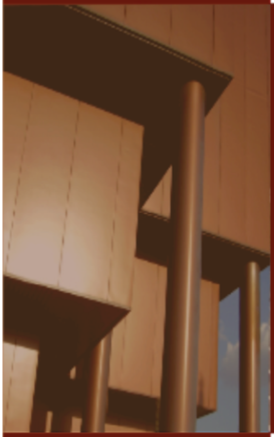
Final Master Project

Master in Telecommunications Engineering

Deep Learning: segmentation of documents from the Archivo General de Indias with DhSegment and NeuralLineSegmenter

Author: Jorge Ugarte Macías

Tutor: Juan José Murillo Fuentes



Dpto. Teoría de la Señal y Comunicaciones

Escuela Técnica Superior de Ingeniería

Sevilla, 2019

Final Master Project

Master in Telecommunications Engineering

Deep Learning: segmentation of documents from the Archivo General de Indias with DhSegment and NeuralLineSegmenter

Author:

Jorge Ugarte Macías

Tutor:

Juan José Murillo Fuentes

Prof. Catedrático de Universidad

Dpto. de Teoría de la Señal y Comunicaciones

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2019

Final Master Project: Deep Learning: segmentation of documents from the Archivo General de Indias with DhSegment and NeuralLineSegmenter

Author: Jorge Ugarte Macías

Tutor: Juan José Murillo Fuentes

The tribunal appointed to judge the above project, composed of the following members:

President:

Members:

Secretary:

They agree to give him the grade of:

Seville, 2019

The Secretary of the tribunal

To my family

To my teachers

To my friends

Acknowledgements

This project concludes my stage as a university student and I qualify as a telecommunications engineer. I have to say that this career, both degree and master, has been an enormous challenge for me, a challenge that represents one of the greatest personal satisfactions. During this period, I have acquired a lot of knowledge, but this is not the only thing that this career has given me. During these years of study at the Escuela Técnica Superior de Ingeniería I have also been trained in other aspects such as maturity, responsibility and dedication. It has also allowed me to enjoy a year of study at the Politecnico di Milano, which I undoubtedly consider an unforgettable experience.

I have to thank the teachers of the college for transmitting me all the necessary concepts to train me as a telecommunications engineer. Special mention to Juan José Murillo Fuentes for the time dedicated to tutoring my master's thesis and introducing me to the fascinating field of deep learning. José Carlos Aradillas Jaramillo has to be also mentioned as he gave me access to the server and helped me with this project advising me.

I am very grateful to my family for their unconditional support during these years. They certainly paved the way to the end of my career.

I am also grateful to my colleagues who, in the course of this career, have become friends. Together we have successfully overcome each and every one of the challenges involved in studying this engineering.

I would also like to mention Sofia Oliveira Ares and Patrick Schone who have helped me in this project.

Thank you.

Jorge Ugarte Macías

Seville, 2019

Overview

The amount of information stored in the form of historical documents is enormous and their treatment is highly tedious. This work is intended to go one step further to facilitate the extraction of information from these documents. This is not easy since many of the historical documents are in bad condition, or their letter is practically illegible to the human eye. The aim of this project is to apply the technique of machine learning, specifically deep learning, to segment digitized images of these documents. That is, differentiate and separate the different areas that make up the document such as text, background or ornaments zones. This will allow each area to be processed separately, which would help to extract the information.

Table of Contents

TABLES

FIGURES

Notation

CNN	Convolutional Neural Network
DNN	Deep Neural Network
AGI	Archivo General de Indias
DL	Deep Learning
CED	Convolutional Encoder-Decoder
FCN	Fully Convolutional Network
HTR	Handwritten Text Recognition
LCN	Local Contrast Normalization

1 INTRODUCTION

Books may look like nothing more than words on a page, but they are actually an infinitely complex imaginotransference technology that translates odd, inky squiggles into pictures inside your head.

-Jasper Forde-

1.1 Project objective

During the history, the way to store information has usually been handwritten documents. These handwritten documents could be considered as a big database in which a lot of information is contained. This information is the key of knowledge through the years; hundreds of years of history covered in these documents. The quantity of information is huge. However, it is difficult to access it for several reasons.

Nowadays, it is possible to generate a document in a computer and send it to the other part of the world in the blink of an eye. What is more, several users can access at the same time the same document and modify it simultaneously. There is a priceless difference between having a digitized document rather than a physical page. Nevertheless, a huge amount the documents that contained so many years of history are not in this format. This makes the study of these documents much more complicated.

It is a big issue the way these documents are stored. Their access is usually limited and, in order to study them, sometimes no more than few people can work with them. In addition, it is normal to find that these physical documents are not in good conditions, as the passage of time does not go unnoticed. This makes them hard to read or interpret.

It is true that it is possible to digitize the documents and have them as images. However, this is still quite far from having a digitized document that can be transformed or modified. It would be great to transcript these documents so that they can be read easily.

Nonetheless, the transcription of these documents is a tedious task. The condition of the document is a major issue, naturally, but it should also be taken into account features such as language, calligraphy, text layout and of course the number of documents to transcript as, for now, this is done by intervention of people. This is not very efficient.

In this project, it is thought that several techniques can be applied to document images in order to help investigators study them. A transcription system for documents is not mundane, different steps must be taken. As mentioned before, the documents conditions are quite random, so they should be processed for a transcription system can act on them. This is the aim of this project.

The processing needed for making de data compatible with a transcription system includes segmenting the images. Namely, differentiating regions in the image. Historical documents can have several regions such as text, graphic content, backgrounds or page region. When digitizing an image, it is important to process it so that the system knows where and what to look for.

For this processing on documents, there has been several methods based on hand-crafted techniques. Although these techniques have done some interesting work, they lack accuracy and generality. As the documents are so diverse, it is normal that these techniques fail on their purpose. The reason is that the rules sometimes are too strict, and they are not able to embrace the variety of handwritten documents.

The solution this project offers is based on detecting certain parts of digitized images of historical handwritten documents using Deep Learning, specifically, Deep Neural Networks (DNN). The use of this DNNs is quite extended: voice transcription, translation, self-driving, image recognition or finance among others. DNNs have been proven to learn accurately certain tasks and perform them afterwards. In this project, some methods based on DNNs are being applied over certain databases like the one provided by the *Archivo General de Indias*. The goal is, by segmenting the documents, to be able to extract the text so that it can be given to a transcription system.

1.2 Memory structure

This memory contains the work done in this project. This work includes investigation on state-of-the-art methods, environment adaptation to use the techniques chosen, python script edition for using the methods and, obviously, interpretation of outputs and results.

To begin with, some background to help the reader understand some concepts is exposed. Then, the investigation of some methods used nowadays is exposed in a state-of-the-art section. In this section, the problem faced in this project is presented so that it can be shown what is this project trying to solve. After this, four articles are exposed in order to position the reader in the development of solutions for the problem just exposed.

Among these solutions, some of them are selected for the project following some criteria that will be presented here. Once the solutions are chosen, there will be an explanation the work that has been done. The present section includes the preparation of the environment to carry out the work, the installation of the methods chosen, some demonstrations of these methods and some code that has been developed.

The next section will present the results of the previous work. For this section there are two subsections: one for the general datasets used during the project and another one for the *Archivo General de Indias* dataset obtained by the *Departamento de Teoría de la Señal y Comunicaciones* of the *Escuela Técnica Superior de Ingeniería* of the *Universidad de Sevilla*.

After the results, there are some conclusions drawn from the work accomplished and some ideas for the future. Among these ideas there is text transcription which has already been presented in the project general objectives.

2 STATE OF THE ART

Any sufficiently advanced technology is equivalent to magic.

-Sir Arthur C. Clarke-

2.1 Problem

The study of historical documents is a crucial issue for better knowing our history. It is not only important just for the history but also for valuable information that can be contained in them. However, as it has been exposed previously, there are various difficulties when studying these documents. Their condition, language, accessibility and treatment are among these difficulties. This makes the extraction of information quite complex.

The intention in this project is to facilitate the study of these documents by extracting specific regions from the digitized images of the documents. Precisely, the objective is to extract the page, the text lines or even the images that might compose the documents. For this, deep learning is applied in the form of neural networks.

Once the regions are extracted, it can be considered that it is easier to study the components of the documents in a separate way. Text lines could be given to a transcript software to extract the text from the document or even ornament could be studied in a deeper way when extracted.

2.2 Background

In this section some concepts are presented so that the reader can follow this project in a better way. Although it is assumed that the reader knows basic concepts of deep learning, it is recommendable to review the following sections. These sections present ideas that are useful when comparing different solutions.

2.2.1 Architectures

Here some architectures of neural networks are explained in a brief way. There are lots of different architectures and models available, however in this section only the architectures that concern to the project will be explained.

2.2.1.1 Convolutional Neural Network (CNN)

“Convolutional Neural Network, also known as convolutional networks, are a specialized kind of neural network for processing data that has a known grid-like topology.” The name indicates that these networks use convolution instead of matrix multiplication in at least one of their layers. So this type of networks are commonly used for image analysis as they scale very well with large widths and heights [1].

A particular case of CNN is the Fully Connected Networks (FCN) in which the neurons of a layer are connected to all the neurons of the next layer. Each connection has its own weight until the last layer which is the output.

2.2.1.2 U-Net

The U-Net comes from the previously commented FCN. This type of neural network takes its name from its shape, see Figure 1. It consists of a contracting path which is in charge of extracting the context and, when it gets to the bottleneck it starts an expanding path which that enables precise location. This type of network relies on a strong data augmentation. [2]

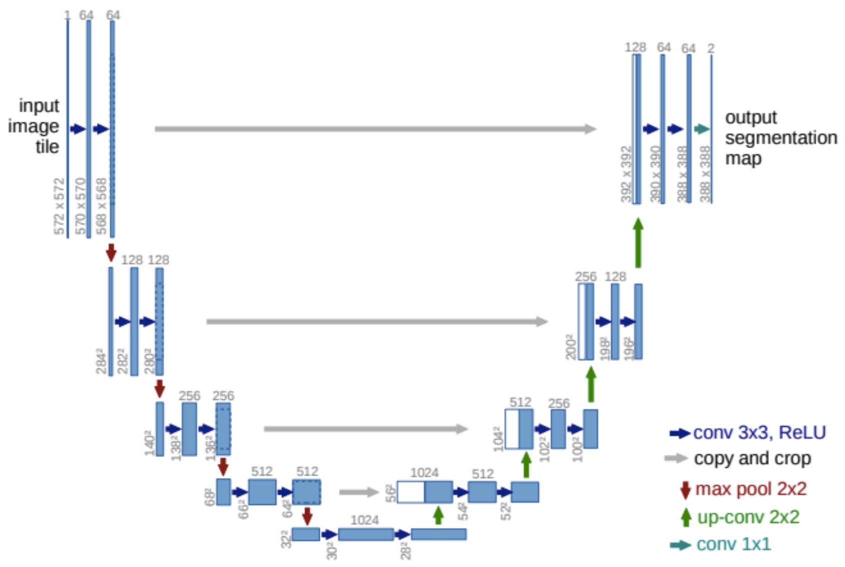


Figure 1 – "U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations." Image from [2], page 2.

2.2.1.3 Encoder-Decoder

Encoder-decoder is a type of CNN. Its architecture consists of two parts: the encoder and the decoder. The encoder is in charge of taking the input and maps it into an encoded representation of the sequence called state (fixed-sized representation internal to the network). The decoder then takes the encoded sequence and generates the output. This means that for each encoder in the architecture, a decoder is needed; so the full architecture can consist of several encoding-decoding pairs. This is widely used in translation applications [3].

CNN and CED learn spatial information in order to avoid rule-based features that have less generalization. [4]

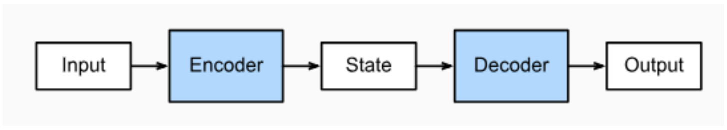


Figure 2 - General architecture of the encoder-decoder CNNs. Image from [3].

2.2.1.4 Atrous

The term "Atrous" indeed comes from French "à trous" meaning hole. The idea is to insert extra "holes" (actually zeros) in the input while executing the convolution so that the output feature map is larger. This helps to get a larger field of view of filters to get a better context [5].

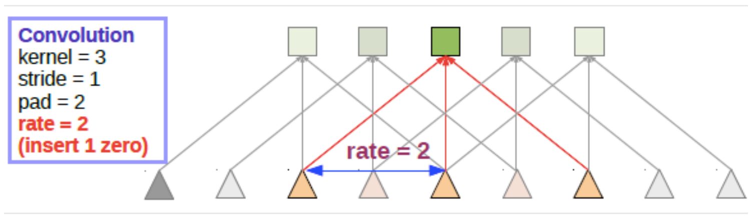


Figure 3 - Example of atrous convolution. It can be seen that when rate = 2, the input signal is sampled alternatively. First, pad=2 means two zeros are padded at both left and right sides. Then, with rate=2, the input signal is sampled every two inputs for convolution. Image from [5].

2.2.2 Evaluation metrics

In order to compare different solutions, it is crucial to evaluate objectively which one is better. In the papers that will be exposed in later sections different metrics are used. The interesting point will be, at least, to have a common criterion to compare all the solutions presented. Some papers will have other metrics, but at least all of them have any of the ones presented: IoU or MIoU.

2.2.2.1 Intersection over Union (IoU)

Intersection over union (also called Jaccard index) is an extended metric to evaluate the accuracy of an object detector (in the case of this project can be lines, pages or other regions of interest). Using set theory, this metric relates the intersection and the union of the ground-truth area with the predicted area. When the areas match perfectly (the case when the ground-truth and the prediction are the same) it results in a 100%. The IoU is calculated for each class separately [6].

In some of the papers presented here it may be also called Recognition Accuracy (RA).

2.2.2.2 Mean Intersection over Union (MIoU)

In the case of multiple classes, IoU needs to be substituted by the mean intersection over union, which actually takes all the IoUs values of each class and calculates the mean [6].

2.3 Papers

In this section, some works are presented and explained in order to try to show the advantages and disadvantages of the different possible solutions. Eventually, some of them will be chosen to develop the project.

2.3.1 Paper 1: Text line segmentation using a fully convolutional network in handwritten document images

In this subsection the following solution is described: Q. N. Vo, S. H. Kim, H. J. Yang, and G. S. Lee, "Text line segmentation using a fully convolutional network in handwritten document images," *IET Image Process*., vol. 12, no. 3, pp. 438–446, 2017. [7]

The purpose of this article is to take a different approach to the problem. The solutions that exist so far are designed by hand or use heuristic methods to detect the lines of a text. The authors of the article present a solution based on a Fully Convolutional Network (FCN) that predicts the structure of lines in document images.

While it is easy to detect a sequence of words on a line when typed, this is not trivial when it comes to handwritten documents. It is a problem to group everything on one line when characters are written differently, with crooked lines or with characters of different lines touching. This reduces the ability to segment lines of text in documents.

Existing methods for text line detection can be divided into three groups: global methods, local methods, and hybrid methods. Global methods first try to estimate the place of the line of text, then form a sequence of characters that is assigned to a line of text and separate those characters of different lines that are joined in some way. Local methods, on the other hand, group characters first and then agglutinate them into separate lines of text. Finally, hybrid methods try to incorporate information from local and global methods to form lines.

One of the biggest problems faced by the authors in this article is the separation of characters that are in contact but belong to different lines. Although some solutions are discussed in the article, there is no perfect solution to that problem.

The main contribution of this article is the FCN application for line detection in handwritten documents. The network learns to generate an energy map to extract the lines from the text given a set of manually labeled images (binary) or binary maps.

According to the authors, compared to the energy map generated by other global methods, the one used in the article improves the connection between components of the text which allows to create the line of the text more easily. Moreover, no manual design or heuristic rules are needed. This is why the system adapts better to different document images.

The architecture used for this system is the Convolutional Neural Network (CNN). Specifically, several layers of CNN have been used resulting in what is known as the Fully Convolutional

Network (FCN). This allows to take into account spatial coordinates on the image unlike a traditional CNN.

The focus of this work for the segmentation of text lines is a system based on FCN as a central module. In Figure 4 it can be observed the scheme of the system.

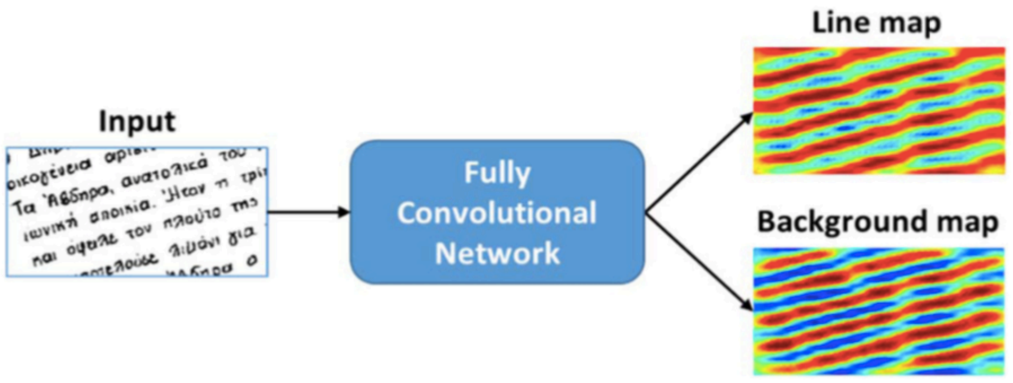


Figure 4 - Overview of the approach for extracting text strings. Adapted from [7], IET Image Process., 2018, Vol. 12 Iss. 3, pp. 438-446 © The Institution of Engineering and Technology 2017.

The network takes a portion of the image and returns two portions of the same size containing a background map on one side and a text line map on the other. In order to detect all the lines of the document, the portion size window is scrolled through the entire image (convolution) and the outputs are integrated into a general map of the size of the original image. In this general map are the lines of text in the form of high energy zones. In these zones, characters of the same line are connected.

Binary maps are used for training. These maps have a label for the lines of text and another for the background. Figure 5 represents this binary map modeling the text labels as white and background as black.

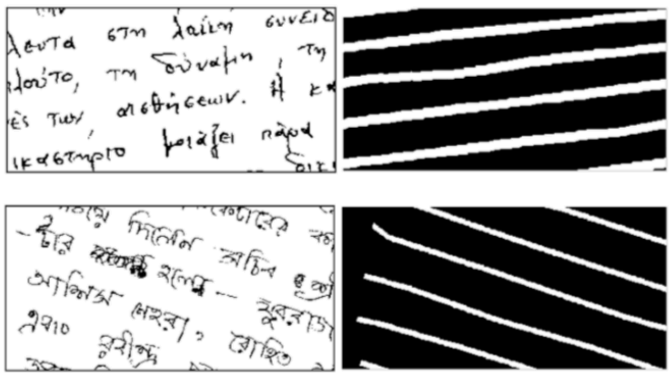


Figure 5 - Left column: training images. Right column: ground truth images. Adapted from [7], IET Image Process., 2018, Vol. 12 Iss. 3, pp. 438-446 © The Institution of Engineering and Technology 2017.

The idea of generating two maps is that the pixels corresponding to text areas and nearby have a high score in the text line map and the rest of the pixels that have a high score in the background map. This is quite redundant, as there are two equivalent outputs. In the paper the authors state that it is enough with the line map to detect the text lines.

In terms of architecture, the authors have tested 3 models (FCN-pool2, FCN-pool3 and FCN-pool4) in which the depth of the networks varies, i.e. they use different quantities of layers. Figure 6 shows the different architectures tested by the authors of the article. Specifically, the layers that will most affect the result will be those of pooling, indicated in orange on the image. These layers reduce the resolution depending on the size of the filter used.

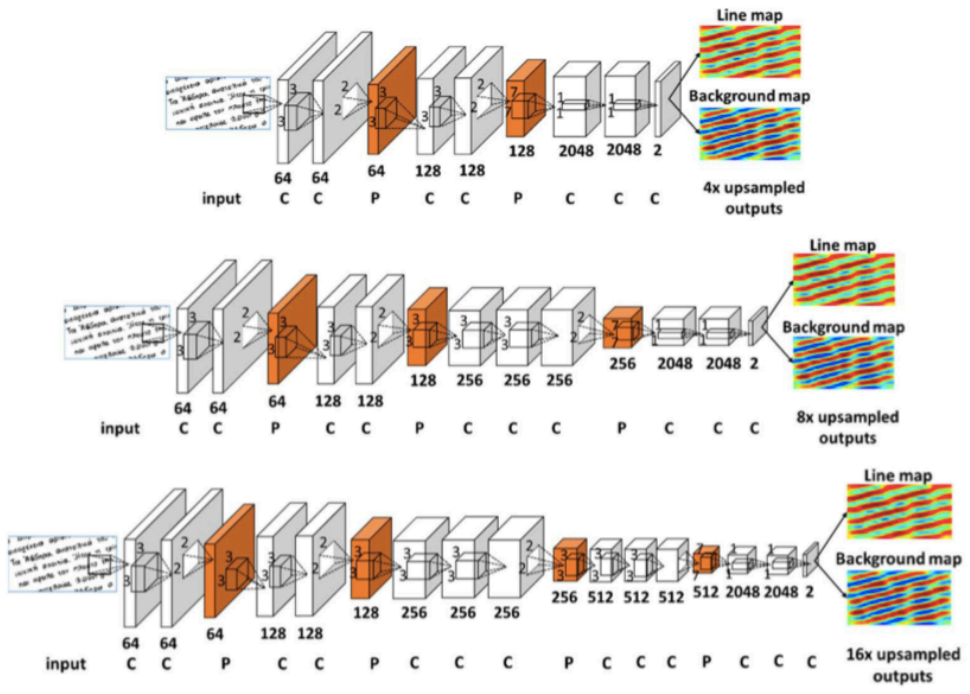


Figure 6 - Networks structures. From top to the bottom: FCN-pool2, FCN-pool3, FCN-pool4. Convolutional layers are denoted as C and max-pooling layers as P. The final classification map is a deconvolution of the last convolution layer. Adapted from [7], IET Image Process., 2018, Vol. 12 Iss. 3, pp. 438-446 © The Institution of Engineering and Technology 2017.

These three candidate models are tested to determine which of them is more suitable for extracting lines of text. In order to test them, they will undergo training with a set of identical data and the output of each model will be studied. Figure 7 shows the results of an example from which sufficient conclusions can be drawn.

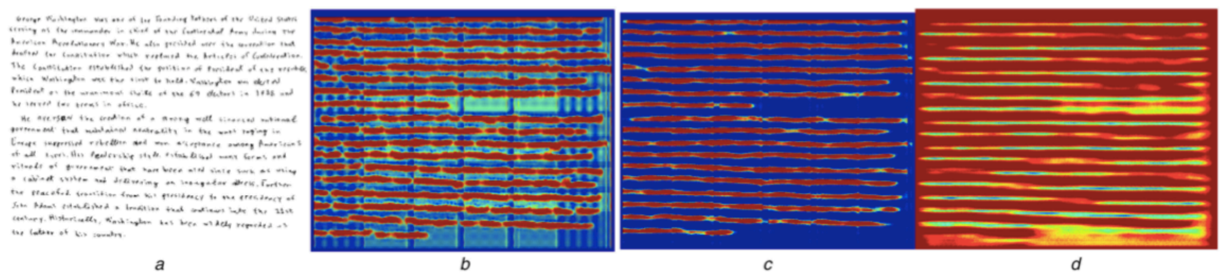


Figure 7 - a) Input image, b) Output of FCN-pool2, c) Output of FCN-pool3, d) Output of FCN-pool4. Adapted from [7], IET Image Process., 2018, Vol. 12 Iss. 3, pp. 438-446 © The Institution of Engineering and Technology 2017.

After analyzing the outputs, it is observed that the output of FCN-pool2 is the result of the high resolution of the max-pooling layer which translates into unconnected lines of text. In the output of FCN-pool2, there are some vertical blue lines (low energy); the reason is not very clear, but it seems that, when there is a big space between two consecutive words in some line and the red regions are not able to concatenate completely, it results in a low probability vertical line (it seems to spread vertically). The green regions (medium energy) seem to come from shorter text lines (last line in a paragraph) where the network thinks that these lines should be as long as the others; there are also some green regions between text lines that probably come from the short distance between two lines. On the other hand, FCN-pool4 has a last max-pooling layer with very little resolution which generates a not precise at all result. It is obvious that the best option is FCN-pool3 since it is in a mid-point that offers quite accurate results.

The article discusses how to extract the line of text. So far what it has been done is to locate the regions where the text is found. Extracting it would mean making a string of pixels that corresponds to a line not to a region. Three main steps are used: line map binarization, line skeleton extraction and line skeleton merging. The first step, line map binarization, consists of labelling those regions with highest energy, as they are supposed to be the text regions. Sometimes, there are several text regions in the same line due to a large separation between characters. This is an issue that the second step, line skeleton extraction, takes into account. In the end, the line skeleton merging step, constructs a line from all the skeletons that might belong to the same line. Its purpose is to make a whole text string.

Sometimes, a character overlaps with more than one text line. For this, the authors separated the problem into two different issues: the splitting of touching characters and text line assignment. To split touching character they use LAG [8], which is a method mentioned in the article which is meant to be used to represent connected components in printed music score images. In the paper: “LAG is first introduced as an efficient way to represent CCs (Connected Components) in printed music score images. The idea is scan de CC horizontally or vertically and extract runs of black pixels.” However, they use this method to segment touching characters in order to be able to separate them. After this, the different components defined by LAG are transferred to the text line assignment issue. For this the nearest Euclidean distance to every text string is computed resulting in assigning each component to a text line.

The model has been tested in an experiment using ICDAR2013 Segmentation Contest data set [9]. This set contains 200 images, from which 10 have been taken as a validation set. The ground truth are binary images of line label which will be used for training. Images are also rotated in order to augment the training data.

The FCN_pool3 is modeled by using Caffe library (deep learning framework developed by Berkeley Vision and Learning Center). The training and testing is done over a single NVIDIA GTX765M. Naming o2o as the number of one-to-one matches, there are several parameters that measure the performance of the model:

$$DR = \frac{o2o}{N} \tag{1}$$

$$IoU = RA = \frac{o2o}{M} \tag{2}$$

$$FM = \frac{2DR \cdot RA}{DR + RA} \tag{3}$$

Where N and M are the number of text lines in the ground truth and the result, respectively. DR is defined as “Detection Rate”, RA as “Recognition Accuracy” and FM as “Performance Metric”. As it has been mentioned before, recognition accuracy will be the method chosen to compare to other methods (recall that recognition accuracy is equivalent to IoU). Having these parameters, the model is compared to other methods:

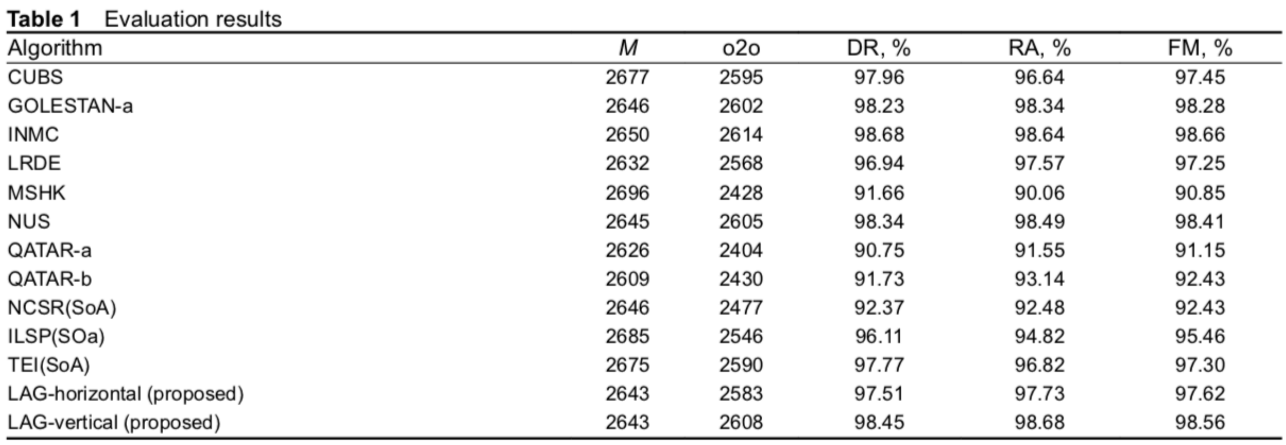


Table 1 Evaluation results

Algorithm	*M*	o2o	DR, %	RA, %	FM, %
CUBS	2677	2595	97.96	96.64	97.45
GOLESTAN-a	2646	2602	98.23	98.34	98.28
INMC	2650	2614	98.68	98.64	98.66
LRDE	2632	2568	96.94	97.57	97.25
MSHK	2696	2428	91.66	90.06	90.85
NUS	2645	2605	98.34	98.49	98.41
QATAR-a	2626	2404	90.75	91.55	91.15
QATAR-b	2609	2430	91.73	93.14	92.43
NCSR(SoA)	2646	2477	92.37	92.48	92.43
ILSP(SOa)	2685	2546	96.11	94.82	95.46
TEI(SoA)	2675	2590	97.77	96.82	97.30
LAG-horizontal (proposed)	2643	2583	97.51	97.73	97.62
LAG-vertical (proposed)	2643	2608	98.45	98.68	98.56

Table 1 - Evaluation results. Adapted from [7], IET Image Process., 2018, Vol. 12 Iss. 3, pp. 438-446 © The Institution of Engineering and Technology 2017.

In Table 1, the results for this method are the corresponding to LAG-horizontal and LAG-vertical (there are two ways of applying LAG). Compared to others in that moment, the results are good. However, there are still some mistakes when solving the touching situations.

In order to compare the neural networks, Table 2 offers a comparison between two models exposed in the article: FCN_pool2 and FCN_pool3:

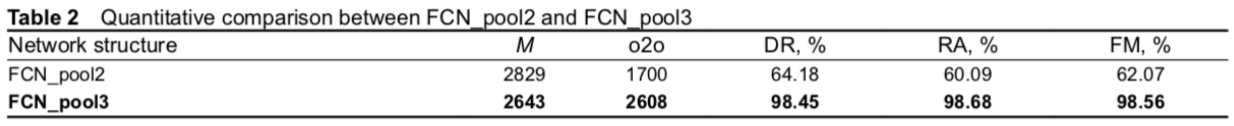


Table 2 Quantitative comparison between FCN_pool2 and FCN_pool3

Network structure	*M*	o2o	DR, %	RA, %	FM, %
FCN_pool2	2829	1700	64.18	60.09	62.07
FCN_pool3	**2643**	**2608**	**98.45**	**98.68**	**98.56**

Table 2 - Quantitative comparison between FCN_pool2 and FCN_pool3. Adapted from [7], IET Image Process., 2018, Vol. 12 Iss. 3, pp. 438-446 © The Institution of Engineering and Technology 2017.

These results presented in the table confirm the results given previously, when comparing the output of the three different models.

The advantages for this model is the ability it has for adaptation to different input images. The main requirement is a suitable training ground truth for the application desired. However, it does not solve perfectly the segmentation of touched characters and it can be improved. In addition, the coding of the model presented in the article has not been found, and for these reasons its use has been discarded for the purposes of this project.

2.3.2 Paper 2: A deep convolutional encoder-decoder network for page segmentation of historical handwritten documents into text zones

In the next subsection this solution is exposed: P. Kaddas and B. Gatos, "A deep convolutional encoder-decoder network for page segmentation of historical handwritten documents into text zones," *Proc. Int. Conf. Front. Handwrit. Recognition, ICFHR*, vol. 2018-Augus, pp. 259–264, 2018. [4]

This article shows the results of a Convolutional Encoder-Decoder based method used for segmenting historical handwritten images into different regions. The method labels each pixel to the different text regions. Specifically, it segments the document in five different classes: main body, comments, decorations, periphery and background. Older methods, traditionally, use prior knowledge of the historical documents and rely on data-oriented features and rules made by experience. The article shows that this method of using Convolutional Encoder-Decoder pairs fits properly to the problem. It will be shown in some results to demonstrate this last statement.

When talking about Handwritten Text Recognition (HTR), segmentation of historical documents is a big step. The goal of the paper is to categorize each pixel into one of the labels mentioned before, resulting in an output were all the different text regions are detected.

Convolutional Encoder-Decoder, has achieved state-of-the-art results in image classification, pixel labelling of natural images or historical handwritten documents among other fields.

The architecture consists of five Encoder-Decoder pairs, taking as input an RGB image of a random size. The output is a labelled image. One of the greatest achievements of this paper is a pre-processing step. When executing this step, the evaluation on public historical handwritten datasets show that this method is in many cases superior to the state-of-the-art techniques.

As the focus of this paper is the segmentation into different text regions on historical handwritten documents, the authors have faced some difficult challenges like layout inconsistencies, different calligraphies, writing style irregularities and low quality.

The authors comment some unsupervised methods that, even though they achieve high performance on the datasets, they lack generality as these methods function with many hand-crafted rules.

On the other hand, there are supervised methods. These supervised algorithms are more efficient and robust for page segmentation.

The proposed method is described as follows. As it has been mentioned before, the authors use a CED architecture in order to classify each pixel in one or more classes (main body, comments, decorations, periphery and background). These results into differentiating different parts of the documents taking into account both spatial and texture information.

One of the main advantages of CED architecture is that pre-processing the input images might result into a much better performance. In the case of this article, the input was 3-channel RGB image. To this image, a Local Contrast Normalization (LCN) algorithm [10] is used. After that, a Gaussian normalization is applied over the image (each channel separately).

The architecture of the CED network used is presented in Figure 8. A random size input is resized to a fixed size (640x416x3). As it can be seen, there are three different images which correspond to the three different channels mentioned before (one for each dimension of the input: height, width and RGB). The authors describe their solution as follows:

“An Encoder consists of a stack of batch-normalized (BN) convolutional layers, which are fed into the element-wise rectified non-linearity unit (ReLU) max(0,x). Then a sub-sampling step is applied (Max Pooling) for dimensionality reduction. Optionally, a dropout layer [11] can be applied to the output in order to prevent over-fitting. In this work, we apply dropout only at the training phase. Each Encoder has a symmetric Decoder, forming an Encoder-Decoder pair.

In the Decoder, up-sampling is applied using pooling indices extracted from the respective encoder in order to preserve boundary details. The last decoder is connected to a logistic regression layer using the softmax function.

The output of the softmax layer is a new image P of shape (640 × 416 × C), where C is the number of classes (C = 5). Each pixel of P is a (1×1×C) vector representing the probability distribution of each class at this position. Image Iout of the predicted labels is calculated by detecting the maximum probability for each pixel.”

After this, the image is resized to its original size and with a single channel.

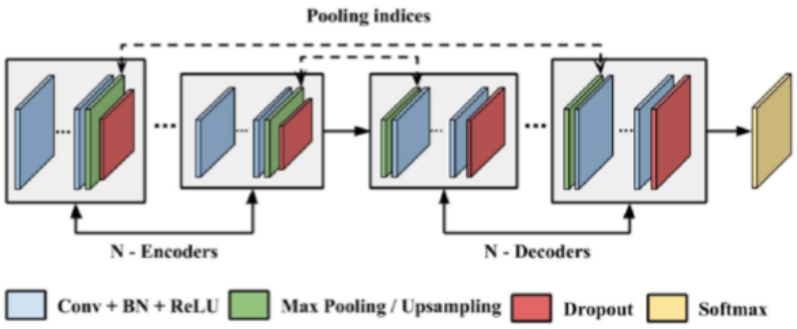


Figure 8 - A schematic of a Convolutional Encoder-Decoder. Adapted from[4], p. 261, ©2018 IEEE.

The network consists of five Encoder-Decoder pairs. The outer pairs are composed of two convolutional layers while inner ones are made with 3 convolutional layers. Also, max-pooling and up-sampling are used (2x2 window with no overlaps).

For the experiments the authors have used six public historical handwritten datasets: G. Washington, Parzival, St. Gall, CB55, CSG18 and CDG863. The measures used for evaluating the different methods are the following ones: Pixel Accuracy(PA), Mean Accuracy (MA), Mean Intersection over Union (mIU) and Frequency weighted Mean IU (fwIU).

The training of the network has been done with Adam optimizer. Weight decay ($5x10^{-5}$) is applied along with dropout on the three deeper Encoder-Decoder pairs (discarding neurons with a probability of 0.5). Constant learning rate at 10^{-4} is used.

The authors state that: “The goal is to minimize the cross-entropy loss between the predicted probabilities of each class and the one-hot encoded ground-truth labels. The model is trained until convergence is reached and we keep the model that performed better on the validation set, based on the mean IU metric.*”*

The comparison in the article has been made in various ways. First, the authors compare their model when using LCN and when not using it. Table 3 shows the results.

CED Variant	*Set-1*				*Set-2*			
	PA	MA	mIU	fwIU	PA	MA	mIU	fwIU
RGB-CED	85	77	61	80	94	60	53	89
LCN-CED	**97**	**86**	**82**	**95**	**96**	**77**	**68**	**92**

Table 3 - Evaluation of CED variants (%). Adapted from[4], p. 262, ©2018 IEEE.

It can be seen that when using LCN-CED model the performance is significantly better. This is expected as the network is taking into account local spatial variability, which is highlighted by LCN. This allows the network to learn and recognize differences more efficiently.

The next step is to compare the method proposed to the state-of-the-art methods. It is important to note that the methods with whom the network is compared, train a different network for each dataset, generating a model for a specific task. The network of this paper is only trained twice, generating two models, one for each of the following sets (it was preferred to train just one network for all the datasets, but ground-truth images are given in different levels of accuracy): Set-1, which includes G. Washington, Parzival and St. Gall (this includes multiple zones in a region level such as paragraph or column among others) and Set-2, which includes CB55, CSG18 and CSG863 (in this case it is just the background and the line region). This allows to show how good the generalization is. The results are shown in Table 4.

Method	*G. Washington*				*Parzival*				*St. Gall*			
	PA	MA	mIU	fwIU	PA	MA	mIU	fwIU	PA	MA	mIU	fwIU
Local MLP [6]	87	89	75	83	91	64	58	86	95	89	84	92
CRF [8]	91	90	76	85	93	70	63	88	97	88	84	94
CNN [1]	91	91	77	86	94	75	68	89	98	90	87	96
FV-CNN [9]	95	93	81	91	**97**	**76**	**71**	**94**	**99**	**91**	**88**	**98**
Proposed (LCN-CED)	**96**	**94**	**83**	**92**	94	75	69	90	98	90	87	97
Method	*CB55*				*CSG18*				*CSG863*			
	PA	MA	mIU	fwIU	PA	MA	mIU	fwIU	PA	MA	mIU	fwIU
Local MLP [6]	83	53	42	72	83	49	39	73	84	54	42	74
CRF [8]	84	53	42	75	86	47	37	77	86	51	42	78
CNN [1]	86	59	47	77	87	53	41	79	87	58	45	79
FV-CNN [9]	95	73	64	91	92	72	60	89	94	71	61	91
Proposed (LCN-CED)	**96**	**75**	**67**	**92**	**96**	**80**	**69**	**92**	**96**	**75**	**66**	**92**

Table 4 - Comparative Evaluation Results of the proposed CED (%). Adapted from[4], p. 262, ©2018 IEEE.

The methods are measured by pixel accuracy (PA), Mean Accuracy (MA), Mean Intersection over Union (mIU in this table), and frequency weighted Mean IU (fwIU).

For Set-1 the proposed method is quite competitive to FV-CNN, actually better in one of the cases. Not for Parzival and St. Gall sets. For Set-2 the method exposed in this article beats all state-of-the-art methods, in some cases by far. It seems that the network of this paper behaves in a better way when there are less classes to segment compared to FV-CNN, for example.

Mean Intersection over Union is chosen in order to have a better comparison with the rest of the methods proposed in this section (in this paper it is called mIU).

The conclusion of the article is that simply by using a specific pre-processing (LCN technique), the performance can be improved in a significant way. Actually, it lets the network to learn spatial and texture variability from the image as a whole instead of patches. It is also remarkable that there is no need for prior knowledge of the dataset or post-processing for the output. Generalization is also worth to be noticed in comparison with other techniques.

However, in this project this method is not chosen. The reason is that there is no code available. It could be developed following the steps the authors share in the paper, but this is not the scope of this project.

2.3.3 Paper 3: DhSegment: A generic deep-learning approach for document segmentation

In the following subsection this solution will be described: S. Ares Oliveira, B. Seguin, and F. Kaplan, "DhSegment: A generic deep-learning approach for document segmentation," Proc. Int. Conf. Front. Handwrit. Recognition, ICFHR, vol. 2018-Augus, pp. 7–12, 2018 [14].

Up to now, document analysis problem has been approached with different tasks, solving problems separately. Actually, these tasks have been usually made with hand-tuned strategies. The authors of this article pretend to handle the diversity of historical document processing with a more generic approach. Specifically, in this article, they attempt to solve different problems: page extraction, baseline extraction, layout analysis, ornament detection or photo collection extraction. The authors have implemented an open-source software based on CNN which predicts each pixel and post-process it in different blocks. In the paper it is shown that a single CNN is enough to solve the different problems exposed before with competitive results.

Lately, there have been serious improvements in semantic segmentation in natural images (pictures, roads, scenes…), however, document segmentation has not been benefited from this yet. The authors believe that, with the actual neural network architectures, some generic approaches are mature enough to outperform some of dedicated systems used up to now.

This work introduces the software . The authors state the following: "a general and flexible architecture for pixel-wise segmentation related tasks on historical documents". This architecture will be tested and compared with other state-of-the-art techniques, and the results will be exposed. The encouraging results may have important influence in the future of historical document segmentation. It is important to remark that it is an open-source software, so anyone can access it and use it.

The system is composed of two steps. The first one, is a Fully Convolutional Neural Network. This network is given an image of the document as an input (the size of the input image is not limited if the memory permits to process it) and outputs a map of probabilities, generated by a ReLU function, of the labels for each pixel. This map will become a mask in the next step, which consists of transforming the prediction map to the desired output of the task. The image processing techniques are standard and depend on the task as the diversity of the outputs is quite diversified. In Figure 14, there is an overview of the system.

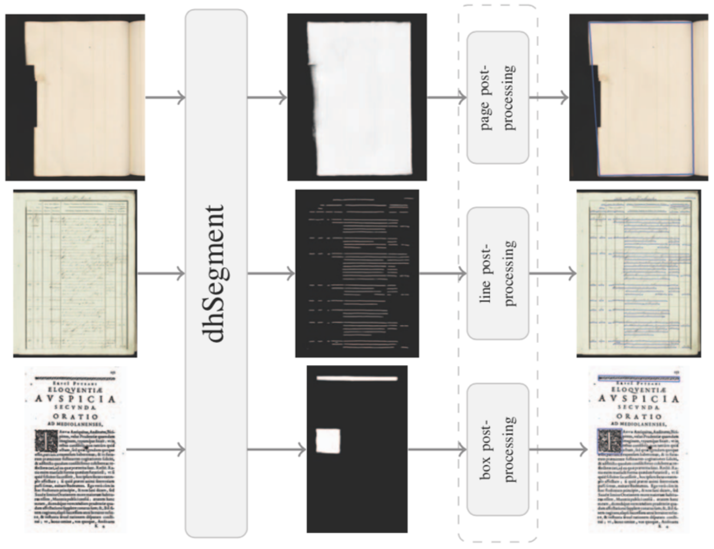


Figure 9 - Overview of the system. From an input image, the generic neural network (DhSegment) outputs probabilities maps, which are then post-processed to obtain the desired output for each task. Adapted from [14], p. 8, ©2018 IEEE.

The network architecture is described in Figure 15. It is composed of two paths forming a U-Net architecture: a contracting and an expanding path. A contracting path consists of repeated convolutional layers which decrease spatial information and increase feature information. On the contrary, an expanding path combines the feature and spatial information through a sequence of up-convolutions and concatenations with high-resolution features from the contracting path [2]. In the case of DhSegment, the contracting path is based on the deep residual network ResNet-50 (it is not exactly the same due to memory efficiency reasons). In the Figure 15 the ResNet-50 is shown in yellow blocks.

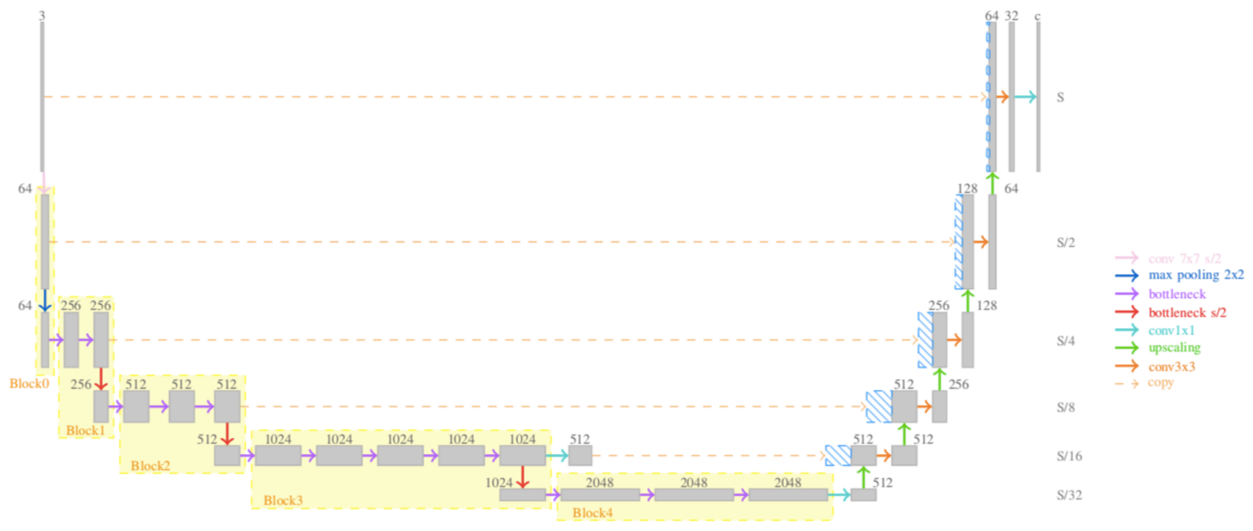


Figure 10 - Network architecture of DhSegment. Adapted from [14], p. 9, ©2018 IEEE.

The authors state the following: “The number of features channels are restricted to 512 in the expansive path in order to limit the number of training parameters, thus the dimensionality reduction in the contracting path (light blue arrows). The dashed rectangles correspond to the copies of features maps from the contracting path that are concatenated with the up-sampled features maps of the expanding path. Each expanding step doubles the feature map’s size and halves the number of features channels. The output prediction has the same size as the input image and the number of features channels constitute the desired number of classes.”

It should be stated that the path uses previously trained weights from the task done in [15] where high level features are learned on a general image classification task.

The architecture has 32.8M of parameters but, thanks to pre-trained weights in the constraint path, only 9.36M need to be trained. These pre-trained weights add robustness and help generalization. The pre-trained weights were trained on a general image classification task that helped the network learn high level features.

After the neural network acts its output will be given to the corresponding post-processing step. The different operations are:

- Thresholding: This is used to obtain a binary map from the output probability map of the network if the classification is binary. If there are several classes, it acts class-wise so instead of a binary map a multiclass map will be obtained.
- Morphological operations: These are non-linear operations used for images and geometrical structures. There are various operators, but the ones used in DhSegment are erosion and dilation which can be applied to binary maps. Erosion makes thick lines

skinny if necessary and dilation is the dual operation of erosion, it makes lines thicker if necessary.

- Connected component analysis: After thresholding and morphological operations are applied, some small connected component may remain. This analysis filters them.
- Shape vectorization: This is used to transform regions into coordinates. For this, the shapes in the binary map are extracted as polygons.

For training part, the authors have used L2 regularization (which means the penalty term is squared) to prevent overfitting with weight decay of 10^{-6}. Learning rate is set to $[10^{-5},10^{-4}]$ with exponential decay. Xavier initialization and Adam optimizer is used [16]. In order to prevent lack of diversity when training, the system uses batch renormalization. Images are also resized for memory issues. The training takes advantage of the data augmentation offered by rotation, scaling and mirroring; this results in less data needed for training.

Training is done on a Nvidia Titan X Pascal GPU. Due to the pre-trained weights, time of training is significantly reduced. During the training, the pre-trained weights help the network to be less sensitive to outliers.

The system has been tested over several tasks in order to demonstrate its generality. Three of these tasks (page extraction, baseline detection and document segmentation) have been compared to other state-of-the-art techniques. Two other tasks (ornament and photograph extraction) are tested in a private way to show the possibilities of the system. These last two tasks will not be mentioned (although they are studied in the article) as it does not offer significant information to this text.

2.3.3.1 Page extraction

Page extraction is the task of locating the original edges of the page document when the image is digitized and there is a residual background. The system is trained with 1635 images. An example is shown in Figure 16.

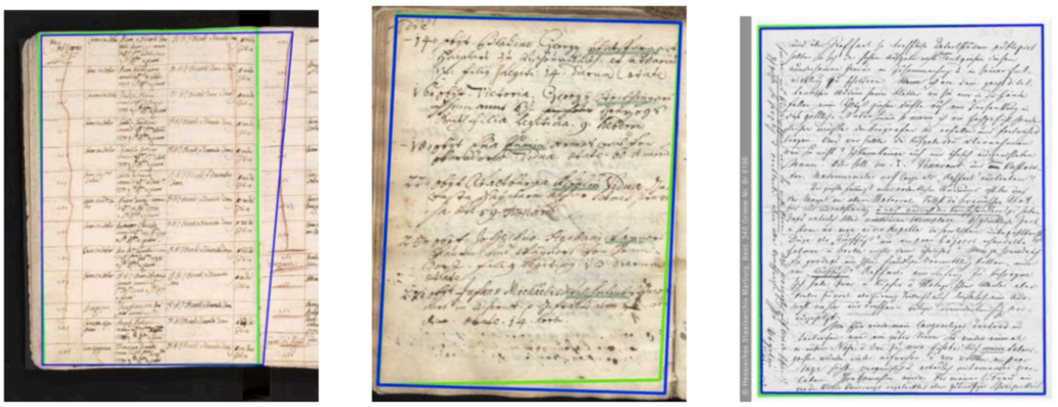


Figure 11 - Example of page detection. Green rectangles indicate the ground-truth pages and blue rectangles correspond to the detections generated by DhSegment. Adapted from [14], p. 10, ©2018 IEEE.

In the first example, page is not correctly detected because part of the side page is also detected as page. The second example shows differences with the ground-truth image, however the decision made is correct. The last example shows a correct page detection.

The results compared to other methods are in Table 6:

Method	cBAD-Train	cBAD-Val	cBAD-Test
Human Agreement	-	0.978	0.983
Full Image	0.823	0.831	0.839
Mean Quad	0.833	0.891	0.894
GrabCut [21]	0.900	0.906	0.916
PageNet [20]	0.971	0.968	0.974
dhSegment (quads)	**0.98**$\pm 6e^{-4}$	**0.98**$\pm 8e^{-4}$	**0.98**$\pm 8e^{-4}$

Table 5 - Results for page extraction (MIoU). Adapted from [14], p. 10, ©2018 IEEE

2.3.3.2 Baseline detection

Text line detection is considered as the previous step of text recognition applications. A baseline is defined as "a virtual line where most characters rest upon and descenders extend below".

When the network predicts a binary mask of pixels which are in a 5-pixel radius of the training baselines. Figure 17 shows an example of baseline detection.

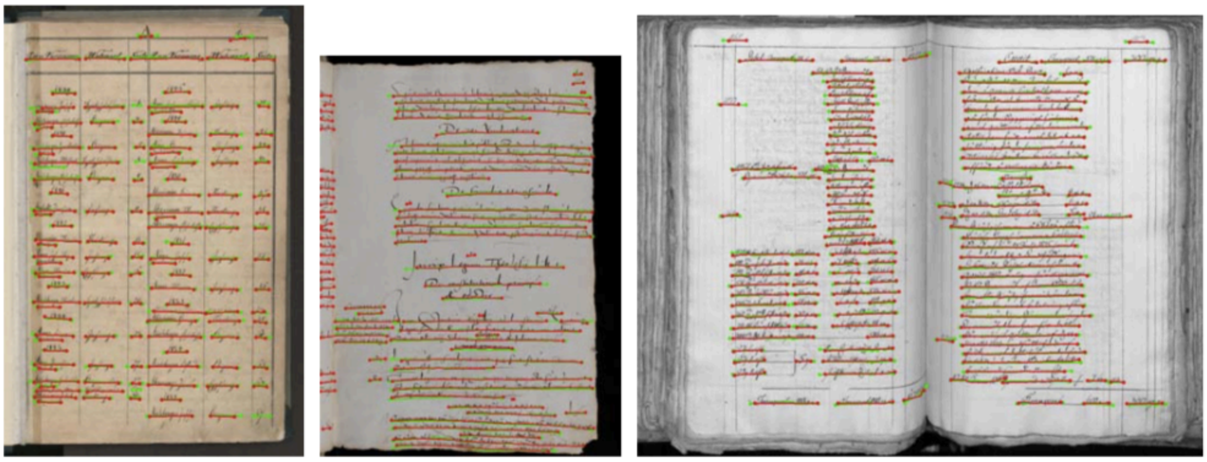


Figure 12 - Examples of baseline detection. The ground-truth and predicted baselines are displayed in green and red respectively. Adapted from [14], p. 10, ©2018 IEEE.

The output of the network compared to the ground-truth, although there are some slight differences, can be considered quite good. The comparison with other state-of-the-art methods are exposed in Table 7.

Method	Simple Track			Complex Track		
	P-val	R-val	F-val	P-val	R-val	F-val
LITIS	0.780	0.836	0.807	-	-	-
IRISA	0.883	0.877	0.880	0.692	0.772	0.730
UPVLC	0.937	0.855	0.894	0.833	0.606	0.702
BYU	0.878	0.907	0.892	0.773	0.820	0.796
DMRZ	**0.973**	**0.970**	**0.971**	**0.854**	0.863	**0.859**
dhSeg.	0.88±.023	**0.97**±.003	0.92±.011	0.79±.021	**0.95**±.005	**0.86**±.011

Table 6 - Results for the cBAD: ICDAR2017 Competition on baseline detection. Adapted from [14], p. 10, ©2018 IEEE

2.3.3.3 Document layout analysis

Document layout analysis is the task of segmenting a document into different meaningful regions. Specifically, this analysis tries to assign a label to each pixel. These labels classes are: text regions, decorations, comments and background. In page extraction and text line detection, only binary classification is needed; in this case two extra classes are needed resulting in a multiclass map. Figure 18 shows an example of layout analysis.

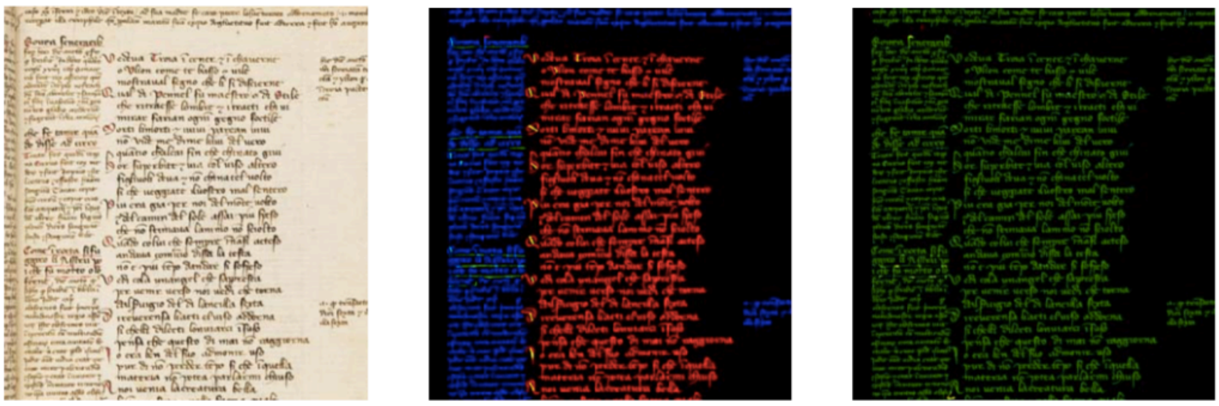


Figure 13 - Example of layout analysis on the DIVA-HisDB test. On the left the original manuscript image, in the middle the classes pixel-wise labelled by the DhSegment and on the right the comparison with the ground-truth. Adapted from [14], p. 11, ©2018 IEEE

The dataset used is DIVA-HisDB [17] (which will also be used to train the models of this project). This data is composed of three groups of manuscripts called CSG18, CSG863 and CB55 (which has higher resolution). Each group has 30 images for training. Table 8 makes a comparison with other state-of-the-art methods in the different groups of the dataset.

Method	CB55	CSG18	CSG863	Overall
System-1 (KFUPM)	.7150	.6469	.5988	.6535
System-6 (IAIS)	.7178	.7496	.7546	.7407
System-4.2 (MindGarage-2)	.9366	.8837	.8670	.8958
System-2 (BYU)	.9639	.8772	.8642	.9018
System-3 (Demokritos)	.9675	.9069	.8936	.9227
dhSegment	.974±.001	.928±.002	.905±.007	.936±.004
dhSegment + Page	.978±.001	.929±.002	.909±.006	.938±.004
System-4.1 (MindGarage-1)	.9864	.9357	.8963	.9395
System-5 (NLPR)	.9835	.9365	.9271	.9490

Table 7 – Results for the ICDAR2017 competition on layout analysis for challenging medieval manuscripts (IoU). Adapted from [14], p. 11, ©2018 IEEE.

To conclude with, the system outputs competitive results for these tasks if not the best ones in some cases. Despite its generality and flexibility, the training is meaningfully quick and efficient thanks to its pre-trained weights. It is also remarkable that the software is open-source, so it can be used, re-trained and tried over different datasets by anyone.

2.3.4 Paper 4: Neural text line segmentation of multilingual print and handwriting with recognition-based evaluation

The solution detailed in this subsection is: P. Schone, C. Hargraves, J. Morrey, R. Day, and M. Jacox, “Neural text line segmentation of multilingual print and handwriting with recognition-based evaluation,” *Proc. Int. Conf. Front. Handwrit. Recognition, ICFHR*, vol. 2018-Augus, pp. 265–272, 2018 [12].

The authors of this article present a new method for detecting text lines in historical handwritten and printed documents images. The technique is based on improving deep neural networks to classify pixels into different classes. Among these classes, there are text and graphic pixels, but it also offers a "glue" technique to unify a whole line in case there are spaces and while also predicting regions where two text lines may merge. The authors believe that their system is the first one to predict the entire perimeter of text lines in documents. In the first place, the system is given a small region of the image and then expands to full images. This aids in scaling and full-scope awareness. The final goal of the project of this article is to segment documents so that the output can be given to a transcription system.

Deep Neural Networks (DNN) have become a qualified method for automatic transcription of handwritten and printed documents. But before transcription occurs, the text lines of the documents must be segmented so that a transcription system can be applied to them.

The network is a 3-stage hybrid that uses CNN, then it has some additional correctional layers and it finishes with a dynamic programming "tie" breaker, which will be explained later. In the first training steps, the network takes small image regions where it learns localized features. As training goes, the batcher takes larger training snippets so that it has entire image scope.

This paper also focusses its work on how good its technique functions when using it with transcription systems. Therefore, the authors evaluate the performance through full recognition systems. There will not be too much explanation of this, as it is not the scope of the article presented here, or even this project.

The ambitious goal of this method is that it seeks to find the actual limits of text regions (it can be done with graphic content too).

As an input to the system, the neural network needs perimeter-based training images so that it can learn the limits of text and graphic regions. The software has been trained and tested with 450 images. The regions of these images are two: text or graphic. In the Figure 9 it is possible to see how the system sort out which region text is and which one is graphic:

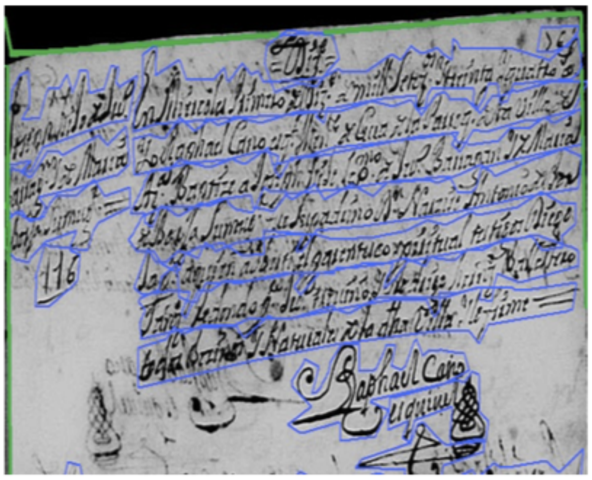


Figure 14 - Annotation of text regions (blue) and marginalia (green) as training data. Adapted from [12], p. 267, ©2018 IEEE.

The model is implemented in Tensorflow for the neural network component, however, it is then folded into Java for post-processing. The core of the segmentation system is a fully-convolutional network. As FCNs are being used, the authors assure that the system goes further than just detecting words, starting or ending points or baselines, it is capable of extracting the whole text line. This is really valuable as it can get the whole text line in order to pass it to a transcription system.

For the architecture of the network it was thought that it should be shaped as an hourglass. As the main goal of the system was to segment a document into text lines rather than finding words or individual text lines, some other elements needed to be added:

- Simultaneous Class Tagging: When the system was trained in the first place, the authors used single regions of images with their corresponding pixel labels in order to know exactly what the class of a specific pixel was. It resulted that the system assigned background class to every pixel. This was solved by making the network predict different layers of information and by ensuring that the batch was big enough to contain representative information.

- Keeping Text Lines Together, Textual Glue: It is normal for the neural network to locate text lines with spaces among the line. Here is where the concept of “glueing” comes up. This element connects different components that are separated in the same text line. In order to do this, the algorithm creates a two-class Gaussian mixture model on a sample of the points in order to know the thresholds for background and foreground. Figure 10 shows how it works on a piece of document.

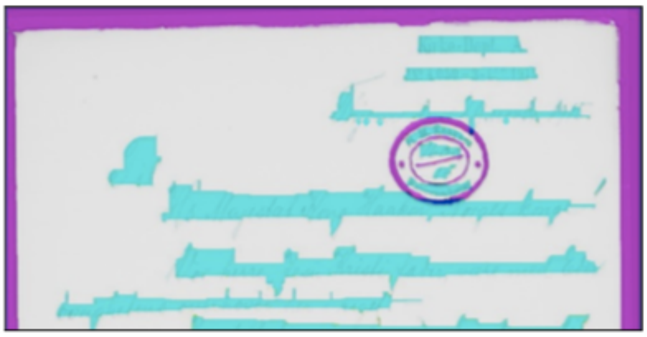


Figure 15 - Piece of an image displaying tight ("glued") regions of text with a covering displayed in cyan color as well as coverings of marginalia, separators and graphics displayed in magenta. Adapted from [12], p. 268, ©2018 IEEE.

- Preventing Textual Line Fusion: This model tends to merge two or more lines whenever they are too close. So, another element is needed to solve this issue. The authors introduce a new label called "buffer" that also needs to be predicted. This new label's role is to indicate regions of the interline background. The effect of this method is shown in Figure 11.

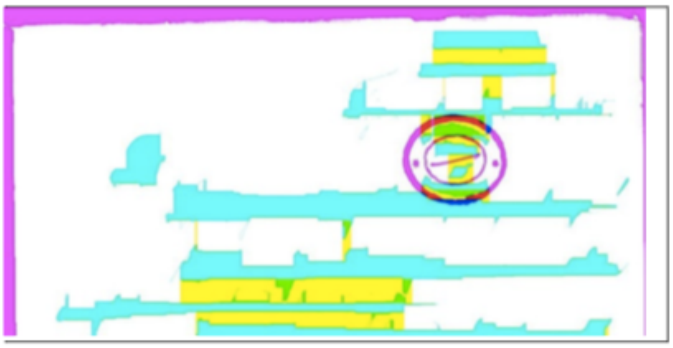


Figure 16 - Same fragment as Figure 10 but showing in yellow the new "buffer" label. Adapted from [12], p. 268, ©2018 IEEE.

- Greater Context Without Needing Size Specifications: Another difference from an hourglass mode is that the mth convolutional layer is connected to the mth deconvolutional one, similar to a U-shaped network. In the interior of the network there is an 8x8 convolutional network which substitutes a fully connected layer and it is not necessary to know previously the size of the input image. At last, it is needed to say that the loss function focusses more on the context rather than making a pixel-by pixel decision. The architecture is shown in Figure 12:

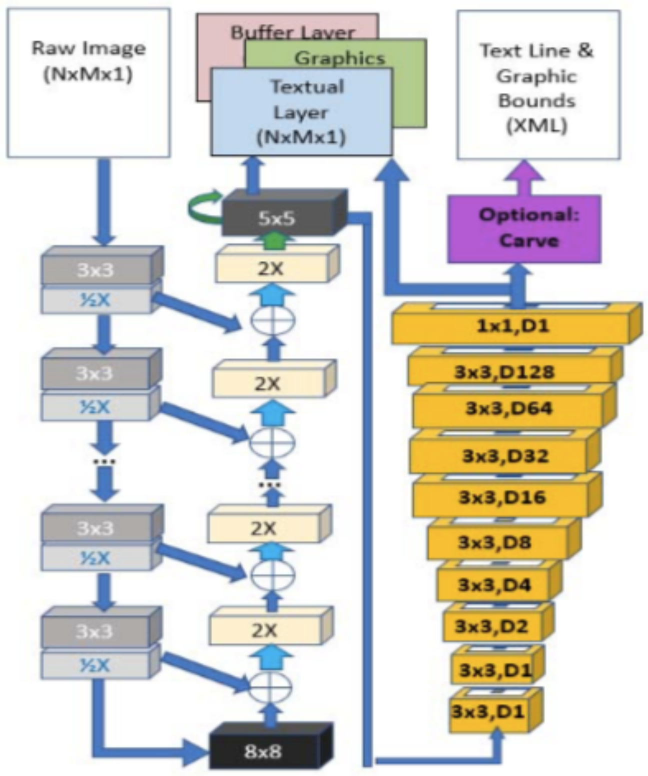


Figure 17 - Graphical architecture of the line segmentation system. NxN boxes represent convolutions; aX boxes represent pooling or deconvolutions; plus-signs are concatenations; and Dn boxes represent atrous convolutions [13] with the associated dilations. Adapted from [12], p. 268, ©2018 IEEE.

The trainer, when it requests the images for each of its batches, tells the batch generator to give a certain percentage of training snippets to have specific labels ("text", "graphic", "buffer" or "generic"). Furthermore, the batcher also asks for the desired snippet size. This is done because when patches are smaller, the results in the first iterations are good; however, when iterates after a few hundred times, the system get confused at the edges. The way to solve this is to grow the image size as the iterations are executed. The network is able to process this change of sizes as a FCN is being used (where no previous knowledge of the image size is needed).

There are situations where this method reaches a tie situation. This happens when the system believes overlapped regions could be either text or background. To overcome this, the system sees if pixels that are together (horizontally) in a tie region are extended to the end of the tie region. Then a dynamic program is used to cut the tie region into upper region and lower region. The Figure 13 shows the final output of the network and the tie-breaking system.

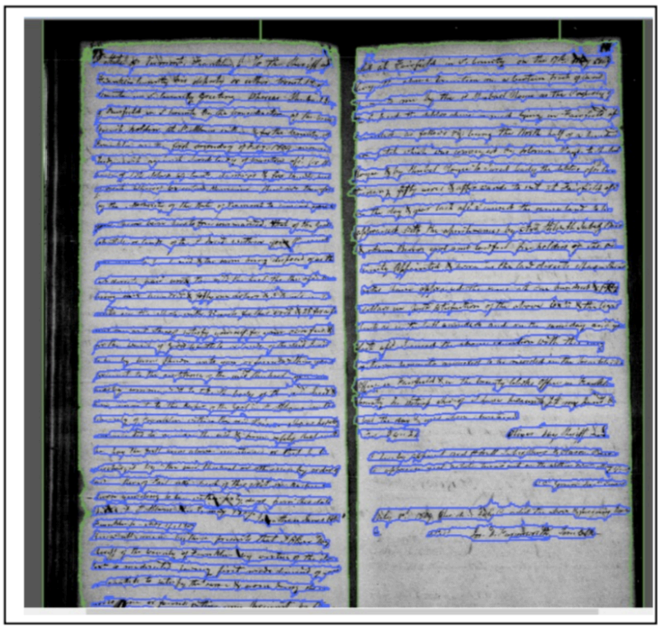
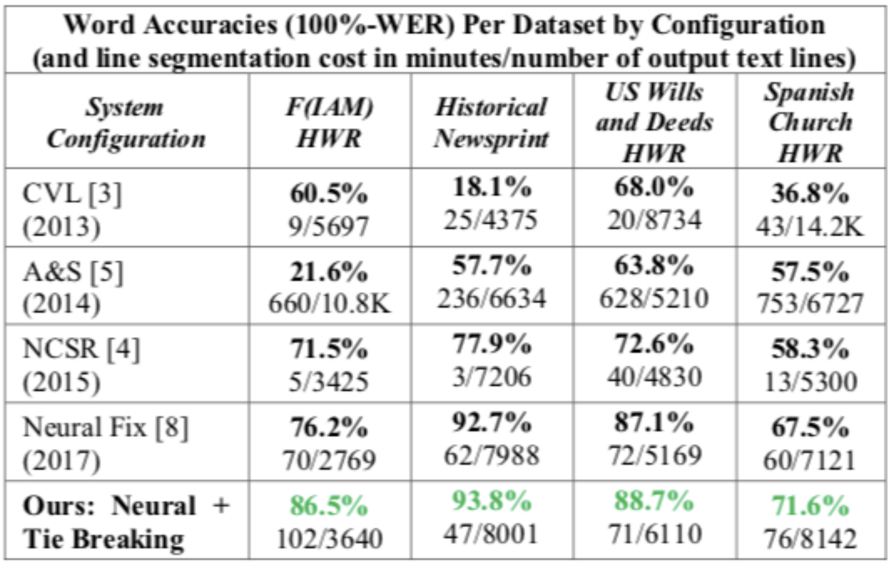


Figure 18 - Example of system output (neural network + tie-breaking system). Adapted from [12], p. 269, ©2018 IEEE.

For the evaluation, several datasets have been used. IAM Data which was complex for its numerous subjects such as cookbooks, news, scientific material and so on. US Wills and Deeds which are approximately 50K words of English handwriting. Spanish Church Records which contents over 40K of hard to read words. For the transcription process (the next one to the segmentation process) some datasets have also been used but the will not be part of this text as it is out of the scope of the project.

Table 5 illustrates the results of comparing this software with other state-of-the-art methods (green indicates the best results per column):



Word Accuracies (100%-WER) Per Dataset by Configuration (and line segmentation cost in minutes/number of output text lines)

System Configuration	*F(IAM) HWR*	*Historical Newsprint*	*US Wills and Deeds HWR*	*Spanish Church HWR*
CVL [3] (2013)	**60.5%** 9/5697	**18.1%** 25/4375	**68.0%** 20/8734	**36.8%** 43/14.2K
A&S [5] (2014)	**21.6%** 660/10.8K	**57.7%** 236/6634	**63.8%** 628/5210	**57.5%** 753/6727
NCSR [4] (2015)	**71.5%** 5/3425	**77.9%** 3/7206	**72.6%** 40/4830	**58.3%** 13/5300
Neural Fix [8] (2017)	**76.2%** 70/2769	**92.7%** 62/7988	**87.1%** 72/5169	**67.5%** 60/7121
Ours: Neural + Tie Breaking	**86.5%** 102/3640	**93.8%** 47/8001	**88.7%** 71/6110	**71.6%** 76/8142

Table 8 - Recognition accuracies based on lines from various text segmentation algorithms plus associated observations. Adapted from [12], p. 271, ©2018 IEEE.

To conclude with, the training system has some new features including technique such as textual glue, neural separation buffers and automatic growth of sample sizing during batches. In the article, authors claim that their perimeter-finding neural line segmentation system with tie-breaking provides better results than other line segmenters.

Again, the code of this work is not available, and no development is intended in this project. However, it has been obtained thanks to the good will of the authors. The results in datasets such as IAM (widely used in text recognition applications) seem very promising.

2.3.5 Comparing and selecting a solution

After presenting the available solutions, it is time to compare them and select the most suitable for this project. The metric used to compare the methods is MIoU. It must be taken into account that the databases on which the methods have been tested are not the same for all of them so there is no way to know which the best is a priori. Additionally, even though one of these solutions might get a high score in a specific database does not mean it will get the best results in the AGI dataset; but naturally, it is a good start. Of course, another essential issue is whether the software is available or not.

Comparing the results presented on the MIoU obtained o Paper 1 [7], Paper 2 [4], Paper 3 [14] and Paper 4 [12] it can be considered that the best solutions are [7] and [14] as they get really high scores in text line detection ([7] could be considered actually the best). The solution in [12] is also quite competitive and could be considered. The worst result would be for the solution exposed in [4], but to be fair, the aim of this method was generality (the results are extracted from sets of databases), so some accuracy can be lost on the way. In addition, the architecture used in this solution, encoder-decoder, is not optimal for image processing; it is normally used for translation and sequence processing. Therefore, this architecture is not optimal for the task intended.

So among the solutions with highest score are [7], [14] and [12], the only ones available were [14] and [12]. In conclusion, these are the solutions chosen for the project.

3 FIRST STEPS IN THE DESIGN OF A LINE SEGMENTATION TOOL

God does arithmetic.

- Carl Friedrich Gauss -

3.1 Software used

After analyzing the previous possibilities, [14] and [12] have been chosen for the project. Both offer desirable characteristics for segmenting the documents of the *Archivo General de Indias* and have very good scores in IoU and MIoU metrics compared to other methods. Specifically, DhSegment [14] offers a lot of post-processing tasks, including page extraction and text line detection. It is also remarkable that is open-source, so there is no limit to use and it can be even modified if necessary. In [18] there is a guide for introducing the software. This guide is a reference to install, try and consult DhSegment. It is also possible to find the reference for the functions used and in case of further information needed, there is a link to the GitHub repository of DhSegment.

The software DhSegment has been studied during the project and is a potential tool for historical document segmentation. It offers some demonstration pre-trained models to try some scripts. It also offers the opportunity to train a model with some images and labels. It has different post-processing tools with some useful functions to modify the outputs of the network. In the following sections it is be explained how to make the software work and how to use these functions available in DhSegment.

On the other hand, NeuralLineSegmenter is the software given in [12] for text line segmentation and extraction and it also has outstanding results and very promising predictions. This software has been provided by the authors of [12] after asking upon request. It has been agreed that it is just for research purposes and it will be deleted after using it. The results will also be shared with these authors when the project is ended. In contrast to DhSegment, this is not an open-source software. This means that the software cannot be modified to make different tasks. The software that has been sent performs the task of text line extraction, allowing some options in the execution. Anyway, this is enough to try it and watch the results on the documents of the *Archivo General de Indias*. For further information refer to the fourth paper in section 2.2 or to [12] where the software is explained in detail.

3.2 Access to the GPU Server

Before commenting on the installation of the chosen software, the environment where the work has been developed will be explained. In the first place, the intention was to do everything locally from a personal computer but given the nature of some of the computation dealt within this text and its high cost it was necessary to install the software and perform the tests on a server with sufficient power to assume the burden of working with this technology.

The server to be accessed belongs to the Departamento de Teoría de la Señal y Comunicaciones de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla. This server is shared in the department to carry out different deep learning experiments that require a higher computing capacity than the one used by an average user. In this project only DhSegment use this server. NeuralLineSegmenter can be run locally so it has been executed in a MacBook Air 2017 as it has not access restrictions.

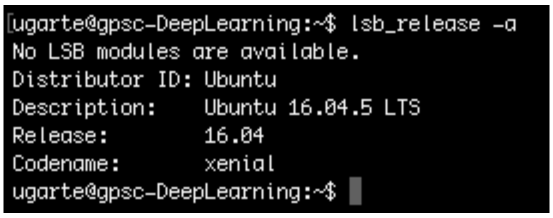
Access to the server is through a ssh tunnel to port 3617 from the command terminal of a personal computer. Access is restricted and can only be made from the department's private network, eduroam network or the VPN network of the University of Seville. Any access outside these networks would be rejected by the firewall of the server.

To connect to the server, ssh client is needed. In Linux and Mac OS, it is installed by default, in Windows, on the other hand, it would be necessary to install some ssh client such as Putty. The connection in this case has been made from a Mac OS, the connection can be made directly from a command terminal. Specifically executing the following:

```
ssh -p 3617 username@sedna.us.es
```

Once executed, it will ask for a password, if necessary, and it will have made the connection to the server allowing commands to be executed on it. It is necessary to comment that, to be able to make the connection, it is indispensable to be registered in the server with a user (username in the previous command) and password. This registration must be done by the administrator. It is also important to note that the permissions the account used did not have administrator privileges but user ones. This has made that certain commands (those in which it is necessary to have permissions of administrator, for example, any preceded of sudo) and installations carried out have had to be asked to the administrator of the server.

Once the server is accessed, an inspection of the environment is made to know where the work is being done and with what characteristics, as there is no information in the first place. In order to do this, several commands are executed that return important characteristics that must be known in order to carry out the work. For example, the type of hardware being used or the operating system. Therefore, the command *lsb_release -a* will be executed first, returning the following result:

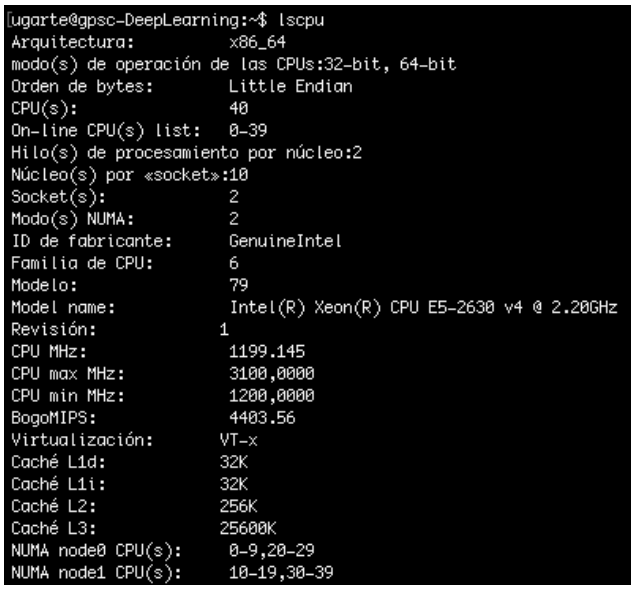


```
ugarte@gpsc-DeepLearning:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.5 LTS
Release:        16.04
Codename:       xenial
ugarte@gpsc-DeepLearning:~$
```

Figure 19 - Command lsb_release -a: returns information about the operating system, distribution and version.

It is now known that the operating system being used is Linux in its Ubuntu distribution. Specifically, version 16.04 of this distribution is being used.

The next step would be to know more about the hardware of this machine. To do this, the command *lscpu* is executed:



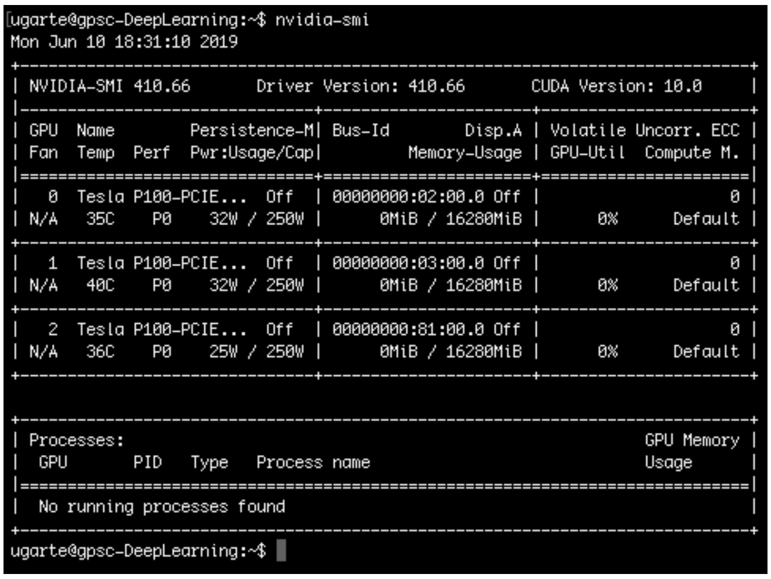
```
ugarte@gpsc-DeepLearning:~$ lscpu
Arquitectura:          x86_64
modo(s) de operación de las CPUs:32-bit, 64-bit
Orden de bytes:        Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Hilo(s) de procesamiento por núcleo:2
Núcleo(s) por «socket»:10
Socket(s):             2
Modo(s) NUMA:          2
ID de fabricante:      GenuineIntel
Familia de CPU:        6
Modelo:                79
Model name:            Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
Revisión:             1
CPU MHz:               1199.145
CPU max MHz:           3100,0000
CPU min MHz:           1200,0000
BogoMIPS:              4403.56
Virtualización:        VT-x
Caché L1d:            32K
Caché L1i:            32K
Caché L2:             256K
Caché L3:             25600K
NUMA node0 CPU(s):     0-9,20-29
NUMA node1 CPU(s):     10-19,30-39
```

Figure 20 - Command lscpu: show system characteristics.

From this command it can be good to know the number of CPUs (40) and the number of cores per socket. This is interesting because the number of parallel processes that the system is able to execute depends on it, yet it will not be decisive.

It is more interesting, on the other hand, to find out which graphics cards are being used. This is because in deep learning graphic cards are usually used over CPUs for better performance. Graphics cards, also called GPU (Graphic Process Unit), are processing units optimized to perform specific operations on a large amount of data, while CPUs (Central Process Unit) are usually a more general processing unit. GPUs, with their hundreds or thousands of small cores, are ideal for working with vectors and arrays that require a high degree of parallelism. That's why they're so interesting when working with neural networks, because it requires working with a large amount of data.

To find out which GPUs the server is equipped with, the following command should be run: *nvidia-smi*. The result is as follows:



```
ugarte@gpsc-DeepLearning:~$ nvidia-smi
Mon Jun 10 18:31:10 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.66       Driver Version: 410.66       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:02:00.0 Off |                    0 |
| N/A   35C    P0    32W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla P100-PCIE...  Off  | 00000000:03:00.0 Off |                    0 |
| N/A   40C    P0    32W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla P100-PCIE...  Off  | 00000000:81:00.0 Off |                    0 |
| N/A   36C    P0    25W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
ugarte@gpsc-DeepLearning:~$
```

Figure 21 - Command nvidia-smi: obtains information about Graphic Process Units (GPUs).

From the output of this command it can be concluded that 3 graphic processing units are available. The model of these cards is Nvidia Tesla P100-PCI of 16GB each. A serial graphics card mounted on an average personal computer has about 1 GB of dedicated memory for processing graphics on the screen among other operations. With this fact is possible to have an idea of the processing capacity of the cards on the server. Therefore, this hardware will be in charge of executing the heaviest operations of the project. The administrator of the server has assigned one GPU for experimenting in this project, particularly GPU 0. This number will have its influence when training a model in future sections.

Once the working environment is known, it is necessary to know how to interact with it. As previously mentioned, it is necessary to create an ssh session or tunnel to connect and execute commands or scripts. However, certain scripts that are not on the server need to be run and also modified for testing. There are several ways to do this. To have scripts on the server they can either be downloaded via the wget command (the packages needed to install DhSegment, for example) or they can be uploaded to the server. To upload files to the server from a personal computer it is necessary to be in an ssh session with the server, have a scp client installed and execute the following:

```
  scp -P 3617 /path/to/local/file username@sedna.us.es:/path/to
/remote/file
```

If necessary, the password of the corresponding user must be entered in order to perform this operation. To specify that, username would be the user name in the server, /path/to/local/file would be the path to the locally stored file and /path/to/remote/file would be the path in which the file will be hosted in the server.

On the other hand, files or scripts must be edited. Two methods are available. The first one is using the nano command that allows you to edit files directly from the command terminal. This is not the most comfortable way to work, but it allows modifications to be made to scripts. The second method is to modify the file locally on the PC and upload it to the server.

This second method may seem more tedious, but it has been chosen for the following reasons. The nano text editor is an editor where mouse cannot be used (just scrolling up and down) so the movement through the script becomes quite complicated. Locally, you can use a text editor that is optimized for the programming language being used and offers tools such as search or substitution of characters and words or auto-filling of variables and functions among others (in the case of this project Sublime Text is being used). Additionally, it is preferred to work in local as there is always an updated copy of the file available; this allows to study and work on the script while not being connected to the server (it is important to remember that access is restricted outside certain university networks).

Knowing how to upload and edit files to the server, it is also necessary to obtain the results of the experiments. Since the server does not have a graphical environment, it is only possible to interact with the server via the command prompt. This makes it impossible, for example, to view images or to show figures and graphics of the processes. This is clearly a problem because there are processed many images and it is necessary to see how they are processed and how they are displayed. The solution is to download the files and check them locally. For this it is necessary to execute the following command (again, it is necessary to have a scp client installed):

```
 scp -P 3617 username@sedna.us.es:/path/to/remote/file/path/to/
local/file
```

As with the upload command, username would be the user on the server, /path/to/remote/file the path where the path is on the server and /path/to/local/file the path where the file will be hosted on the local machine. On many occasions, network trainings or image processing with these networks can be very long lasting. Therefore, it is necessary to keep the session active for a long period of time, however, it is not always possible to be connected to the university network for so long. Therefore, the ideal would be to be able to keep the session active on the server performing tasks even when not connected or having closed the command terminal. There is a way to do this through the screen program. This program is installed on the server and allows a permanent ssh session on the server even when not connected. To do this, it is necessary to execute the screen command in the terminal corresponding to the server. This command opens a permanent session that will remain active until the exit command is executed. To recover the session, simply execute the screen -r command. In the case of having more than one session open, the command will return the open sessions and a one of them will have to be chosen executing again the command along with the session to recover.

3.3 Installation of DhSegment

The installation of DhSegment involves a series of steps and two ways of carrying them out: using the pip package manager or through the Anaconda distribution that allows working in data science and data learning environments for Python and R in different operating systems.

It is necessary to comment that, in the beginning, it was tried to install this software in a personal computer (MacBook Air 2017) to check its operation and to carry out some tests. Although it was possible to install it, it was not possible to use it since DhSegment is developed to work on GPU. It is true that personal computers, such as the one used in this project, have their own GPU for user applications. Nevertheless, there is no GPU support for Mac OS. For this reason and for the performance improvement, it has been decided to work in the server exposed in Section 3.2.1. The following steps are done in the server.

For the installation, in the first place, it was decided to use the pip package manager, because in the first place, it required less space and requirements. However, once installed, it caused library problems that did not allow working correctly.

It was therefore decided to install DhSegment using the Anaconda distribution. This distribution offers a much more stable environment and all the libraries needed to work with DhSegment. Therefore, the first step is to install Anaconda. To do this, from the terminal of the server the following should be executed:

```
 wget https://repo.anaconda.com/archive/Anaconda3-4.6.11-Linux-
x86_64.sh
```

This command downloads the script needed to install Anaconda in its version 4.6.11 that contains Python in its version 3.7.3. Once downloaded, it is necessary to run it, to do so:

```
 bash Anaconda3-4.6.11-Linux-x86_64.sh
```

Once this last command has been executed, Anaconda is considered installed if there have been no problems. With Anaconda installed it is possible to install DhSegment itself.

To begin, the repository that has DhSegment in GitHub should be cloned:

```
git clone https://github.com/dhlab-epfl/DhSegment.git
```

Once the repository has been cloned, it is necessary to create an environment. However, this is done through the conda command that can cause problems when executing because it is not found in the usual command directories. To solve this the following command is introduced in the terminal:

```
export PATH=~/anaconda3/bin:$PATH
```

Now it is possible to create the environment. The environment is made through the file environment.yml that is provided in the repository of DhSegment. This environment is a directory that contains a specific collection of conda packages to run DhSegment. It is created using:

```
conda env create -f environment.yml
```

With the environment created, the next step would be activating it. Activating an environment means moving between environments or changing them. The command to be executed to make DhSegment work is:

```
source activate dh_segment
```

Once the environment is activated it would be interesting to know which packages are in it and what versions are being used. For this, there is a command:

```
conda list
```

It is supposed, after this last command, that the environment is activated. If it is not, it is necessary to specify the environment to the command with the option: *-n dh_segment*. This command is just to know the version of the packages used, it is not essential in the execution of the software. In Table 9 there some interesting packages (not all of them) with their correspondent versions:

Name	**Version**
_tflow_select	2.1.0

cudnn	7.3.1
dh-segment	0.4.0
imageio	2.5.0
numpy	1.16.2
opencv-python	4.0.1.23
python	3.6.8
sacred	0.7.4
scikit-image	0.15.0
scikit-learn	0.20.3
scipy	1.2.1
sphinx	2.0.1
tensorboard	1.13.1
tensorflow	1.13.1
tqdm	4.31.1
zlib	1.2.11

Table 9 - Some important packages and their versions available in the environment.

It might be also necessary to change the ~/.bashrc file to resolve library dependencies by adding the following two lines:

```
  export
LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64:/usr/
local/cuda/extras/CUPTI/lib64"

  export CUDA_HOME=/usr/local/cuda
```

Finally, it is necessary to run the setup.py script that comes in the repository. This script allows to add the DhSegment package into the code by importing dh_Segment, for example.

```
python setup.py install
```

At this moment DhSegment is installed and it is now feasible to use it from the scripts. In fact, there is a directory called DhSegment in the place where the software has been installed. Within this new directory are some of the scripts needed to perform a demonstration, some examples, scripts to train as an example among other files.

Although the software is installed, another step must be taken. It is possible now to use DhSegment with a provided model, but not to train a new model. If a training is executed, it need to be configured first. Here is where *sacred* [20] appears. As the reference says:

"Sacred is a tool to configure, organize, log and reproduce computational experiments. It is designed to introduce only minimal overhead, while encouraging modularity and configurability of experiments. The ability to conveniently make experiments configurable is at the heart of Sacred. If the parameters of an experiment are exposed in this way, it will help you to:

- keep track of all the parameters of an experiment
- easily run your experiment for different settings
- save configurations for individual runs in files or a database
- reproduce results"

This package allows to configure a training process for a model of DhSegment. To install it (administrator privileges are mandatory for this command):

```
pip install sacred
```

3.4 Execution of NeuralLineSegmenter

The package received with the software was a .tar file. There is no need to install NeuralLineSegmenter, it is only necessary to extract the files. Once it is extracted, it appears a directory where the software is contained. Inside this directory there are several files.

- atrousModels.smb: This is a directory where the trained network is. The model is a .pb file along with a directory where the variables are.

- NeuralLineSegmenter-common-1.0-SNAPSHOT.one-jar.jar: This is the executable file for the JAVA software to be run.
- outDir: A directory where the outputs are saved.
- SpanDirList: Plain text file where the path to the input files are listed.
- SPANISH_DIR: This directory contains two example images. In the first place, the images listed in SpanDirList were these two example images.
- aletheiaOutXmlDir: This is the output directory for the .xml files generated by the software.
- README.txt: Text file where some indications can be found. It contains the possible options of execution of NeuralLineSegmenter-common-1.0-SNAPSHOT.one-jar.jar.

The options for the execution are the following:

```
 -list <listname>/-file <filename>: listname is text file with a
list of the images to process while filename would be the file
directly.
 -pageSplit: If two pages are in the same image add this option
in the execution.
 -noSwaths: Do not make images snippets
 -outXml xmldir: Output directory for the xml files generated
 -snipDir snipdir: Output directory for the snippets
 -inDtypeDir dtypeDir: Input only
 -makeSnipsOnly: Snip-ify only from existing XML
 -noVert: Do not do any vertical processing
 -swathHeight: Set height of image
 -noTar: Do not tar up the file-related snippets & metadata
 -copies: Number of files to run simultaneously
 -printVariants: Report {STD|Specific} forms
```

In the next section, demo section, there will be an example of execution using some of these options. The software has been developed with Caffe library [19]. This library: “Is deep learning framework developed by the Berkeley Vision and Learning Center (BVLC). It is implemented in C++ and Python and Matlab interface.” [7]. As this software is executed via CPU, it can be run in any personal computer, however the executable is compiled from Java, so it has to be installed before executing it. Has it has been mentioned before, this software will be executed in a MacBook Air 2017 which is not too powerful for this task. It is true that this program would run better in the GPU Server (although the software does not run in over GPU the server has also a lot of CPU power), nevertheless, it was decided to run it in the personal computer as the access to server was more restrictive and allowed less flexibility whenever the software wanted to be run.

3.5 Demo

3.5.1 DhSegment

Once the software is installed, a test will be performed to see that it works. To do this, it is necessary to enter the DhSegment directory.

From here there are two ways to test the demo.py script: with model trained personally or with a model provided by the developers. For the case that occupies this section, the training part is avoided and the pre-trained model that is provided is used. In later sections, training will be carried out and results will be compared, but now it is only important to check that the installation is correct, and that the software works correctly.

Once it is decided to use a pre-trained model, it must be downloaded. To download, go to the /demo directory (located inside the /DhSegment directory). Once in /demo, the following command is required:

```
 wget  https://github.com/dhlab-epfl/DhSegment/releases/download
/v0.2/ model.zip
 unzip model.zip
```

After these commands there must appear a /model directory, where the pre-trained model should be. Trained neural network models have a .pb termination. Inside /model there is a file called saved_model.pb. The demo.py script will look for files with this ending in the directory indicated, specifically it looks for some model trained in a directory (/demo/page_model/export) and if it does not find it, it looks in /demo/model by default.

The next step is to get the images that are processed by the already downloaded model. These images must also be downloaded separately. Therefore, in /demo, the following is executed:

```
 wget https://github.com/dhlab-epfl/DhSegment/releases/download/
v0.2/ pages.zip
```

```
unzip pages.zip
```

These commands create a /pages directory. Within this directory there are three more: /test_a1, /train and /vl_a1 as well as a classes.txt file. The last two directories, as well as classes.txt, are not necessary as they would be used in case of wanting to train the model personally with the same data used to train the model given. Just out of curiosity, it can be observed that inside /train there are 1635 images with their respective tags to train and that inside /val_a1 there are 199 images also with their respective tags for validation. In any case, the interest of this section falls on the 200 images included in /test_a1. Within /test_a1 there are also the images with their respective tags (separated in two directories /images and /labels), however, the tags in this section are useless because they would only be necessary for the test part in a training.

Once it is known which images will be processed it is time to run the script. It is necessary to do it with Python 3 (from the /DhSegment directory):

```
pyhton3 demo.py
```

The script will process 200 images. Its results can be seen in the /demo/processed_images directory if everything has worked correctly.

The possibilities of this software have been detailed before, that is to say, it is known that it can be trained to get a model that segments different parts of the document. Specifically, it can extract the entire page, detect lines, analyze layout and extract images or ornamentation. This recently executed script performs only one of these operations: page extraction. In addition to applying the model to process the images, the code draws a box on the original image indicating which zone is the page of the document within the processed image.

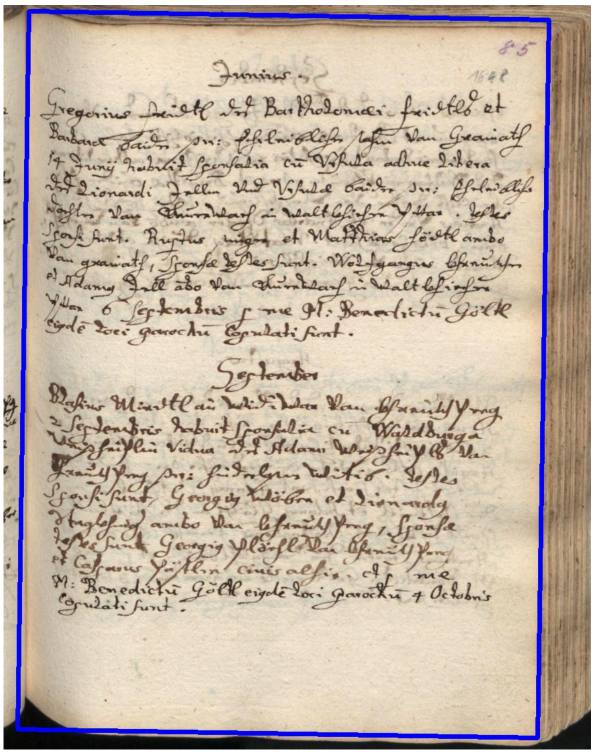


Figure 22 - Example of page extraction with DhSegment made in the demonstration.

3.5.2 NeuralLineSegmenter

As it has been explained before, this software cannot be modified, it is just an executable file with some options available for the execution. The package received from the authors of [12] contained a couple of images to try it and a README.txt file to make a test over these images. In the README.txt file there are some examples of executions with different options. The first example of this README.txt file will be the one used for the demonstration. The command goes like this:

```
 java -Xmx8g -Dfile.encoding=UTF-8 -jar NeuralLineSegmenter-
common-1.0-SNAPSHOT.one-jar.jar -list SpanDirList -snipDir
outDir -outXml outDir -noVert -noTar -pageSplit -swathHeight 60
-copies 2 > out.SpanChr
```

This command runs the executable java file (specifying UTF-8 encoding) where the inputs are in a list called SpanDirList, the output snippets go to outDir and the output -xml files go to outDir. It is also indicated that no vertical processing to the images or tar compression to the snippets. As the Figure 23 shows, there are two pages in the image so the option for splitting the page is in the line to execute (-pageSplit). The height of the snippet is set to 60 pixels as it is done in [12]. In the paper, the authors use 36 pixels when they want to detect print documents and 60 pixels for handwriting, so the same criterion is adapted here. Two copies are processed simultaneously. The standard output for the command is redirected to a file called out.SpanChr.

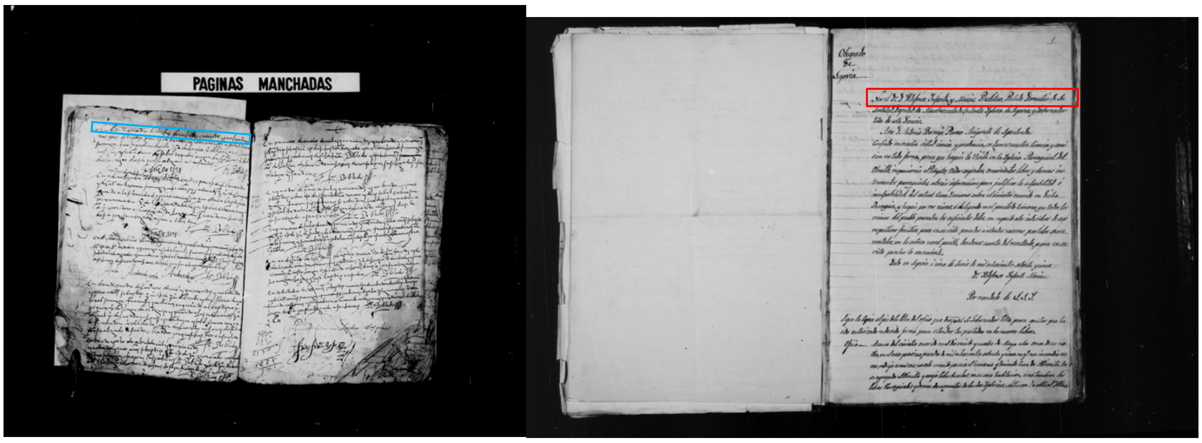

PAGINAS MANCHADAS

Figure 23 - Demonstration images. All the text lines will be extracted from these images. The line in the red box will be shown as an example.

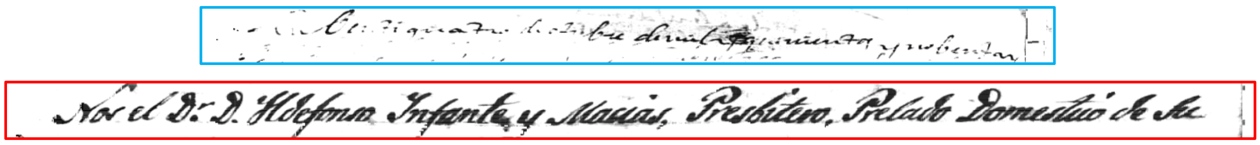

Nos el Dr. D. Ildefonso Infante y Macias, Presbitero, Prelado Domestico de Su

Figure 24 - Lines extracted by NeuralLineSegmenter from the documents provided.

3.6 Code Developed

In this section the code is going to be commented to know what it does. For DhSegment (developed in python), some of it comes from the demo and has been adapted for the dataset and other has been developed in this project (text line detection and text region detection). Apart from the software developed for image processing itself, some other code has been developed in this project for parsing XML files to PNG masks (in python) and for training (this was also based on some given code but has been modified to configure it for the necessities of this project).

As for NeuralLineSegmenter, a script has been developed in python to run the program in a loop to sequentially process all the images.

In this chapter only some examples will be shown to prove the code works, but it is in Chapter 4 where the code is applied to all the AGI dataset, and some others, in order to analyze the results of the two solutions (not all the images will be shown).

3.6.1 DhSegment

This software is very versatile and there has been a lot of time spent in studying it and understanding it due to its complexity. Previous experience in Python is desirable to be able to interact well with this platform.

3.6.1.1 Page extraction for AGI (prueba1.py)

The next thing to do after trying the demo would be to try the same model and script in the images from the AGI. The intention is to extract the pages from the documents of the AGI. But before doing that, it is better to know what the script actually does when it is executed. The code lines that are explained right after are the most significant parts of the code, it is not the whole of it.

The specific libraries needed (apart from other more general-purpose libraries) for this script are

- from dh_segment.io import PAGE (imports the Page class)
- from dh_segment.inference import LoadedModel (necessary to load the models of networks)
- from dh_segment.post_processing import boxes_detection, binarization (for postprocessing purposes)

At the beginning of the script there are two definitions of functions: page_make_binary_mask and format_quad_to_string. The first function uses two post-processing methods:

```
 mask = binarization.thresholding(probs, threshold)
 mask = binarization.cleaning_binary(mask, kernel_size=5)
```

These two lines compute the binary mask of the detected Page (object of Page class) from the probabilities output by network and remove and clean small elements from the binary image using mathematical morphology.

The second function defined change the format from coordinates of the image to string.

Once these functions are defined, it is mandatory to define some variables.

```
 model_dir = 'demo/model/'
 input_files                                              =
glob('/media/HDD/Databases/AGI/AGI_INDIFERENTE_0416/*')
 output_dir = 'demo/prueba1'
```

```
os.makedirs(output_dir, exist_ok=True)
```

These variables tell where the model that is used for segmentation (model_dir) is, where the files that are processed by DhSegment are and where are the stored results. The last line is for creating the output directory in case it does not exist.

After these definitions, the script will open a TensorFlow session (tf.Session()). The next lines are executed inside this session. The model needs to be loaded at the beginning as it is in charge of processing the files:

```
m = LoadedModel(model_dir, predict_mode='filename')
```

This function, that comes from DhSegment.inference, loads the model hosted in model_dir. The predict_mode='filename' tells the function to load the image, resize and predict returning the labels, the probabilities and the original. Other predict modes can be found in [18]. The model loaded for this example is the same pre-trained model given for the demo script.

Now, with a loop, the script goes through all the files in the input directory. The name of each file inside the loop is stored in a variable called *filename*.

```
prediction_outputs = m.predict(filename)
probs = prediction_outputs['probs'][0]
original_shape = prediction_outputs['original_shape']
probs = probs[:, :, 1]
probs = probs / np.max(probs)
```

The lines above do the next actions. First, the model uses a predict method to calculate the labels, the probability map for each class and the original shape of the image as it was defined when the model was loaded. The probabilities from the output are then collected by *probs* and the original shape of the image by *original_shape*. The last two lines take the probabilities of class 1 (class 0 is the background) and then normalize those probabilities.

After this, the page_make_binary_function mentioned before is fed with the normalized probabilities calculated in the last step. It returns a binarized page. After that the binary image is resized (resulting in a variable called *bin_upscaled*) and passed to another post-processing method from DhSegment:

```
pred_page_coords=boxes_detection.find_boxes(bin_upscaled.astype
(np.
 uint8, copy=False),mode='min_rectangle', n_max_boxes=1)
```

This find.boxes() method finds the coordinates of the box in the bin_upscaled binary mask. There are three ways of doing this and it is specified through the mode argument: 'min_rectangle' calculates the minimum rectangle rotating it if necessary, 'rectangle' does the same without rotating and 'quadrilateral' finds the minimum polygon approximated by a quadrilateral. The argument n_max_boxes tells the maximum number of boxes that can be calculated with the largest areas. Other arguments can be found in [18].

After all these steps, the script runs some instructions to output the original image with the rectangle calculated on it. Figure 25 shows an example from AGI.

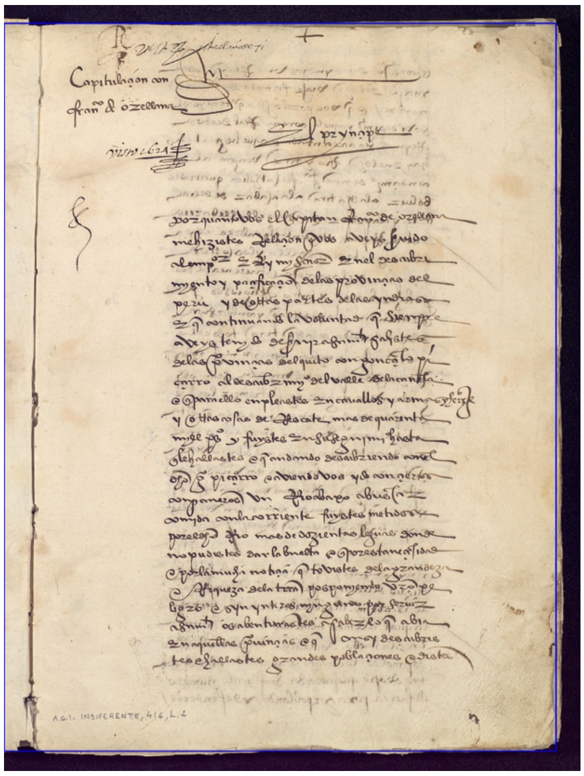


Figure 25 - Example of a document from AGI. In blue the rectangle calculated by DhSegment.

In this case, although the model is given by the authors, the extraction is not achieved in the left side of the document. It does not detect the vertical line that does the division with the other page. It seems that it rather detects the difference in colors (black from the background of the image and brown from the background of the document) than detecting the actual page.

3.6.1.2 Parsing XML files to label images

When training a model, there are two main things needed: input images and their corresponding ground-truths. In order to feed the network, the ground-truth must be in a specific format. However, when working with a database that has images with their ground-truths, usually the ground-truths are given in a generic way. Specifically, in XML (eXtensible Marcup Language) format. This language allows to describe an image using tags and values. For example, it can use tags such as text regions with some values that can contain the coordinates of the region where the text is in the image.

The problem is that neural networks cannot be fed with XML files directly. First, they need to be parsed. Parsing an XML file means to extract the information in it in order to create an image. Depending on the dataset, different information can be contained in a XML file (text regions, text lines, graphic content, background…). In some cases, all the information can be used, but usually only the information of interest is extracted and parsed.

To parse the content of a XML file to an image a python script has been developed in this project. In the script the first thing needed is:

```
 xmldoc = minidom.parse(file)
```

In this moment, the whole XML root with its sons and values is contained in *xmldoc*. Now it is only necessary to extract the data of interest:

```
 width=xmldoc.getElementsByTagName('Page')[0].getAttribute('imag
eWidth')
 height=xmldoc.getElementsByTagName('Page')[0].getAttribute('ima
geHeight')
 itemlist = xmldoc.getElementsByTagName('TextLine')
 coordinates = item.getElementsByTagName('Coords')
 coord_list=coordinates[0].getAttribute('points')
```

These previous lines extract: the size of the image (in order to create the mask where the program will draw the regions), all the text lines sons and their coordinates values. After this the program will have the coordinates for the text lines regions and will be able to draw it in a blank mask previously created.

This script is done for the ICDAR2019 database. It must be said that it might not work in other databases as the tree of the XML files might differ. Therefore, this script has been applied to 984 images, generating 984 PNG ground-truth images (there are even more available, but resources and time consumption were too high).

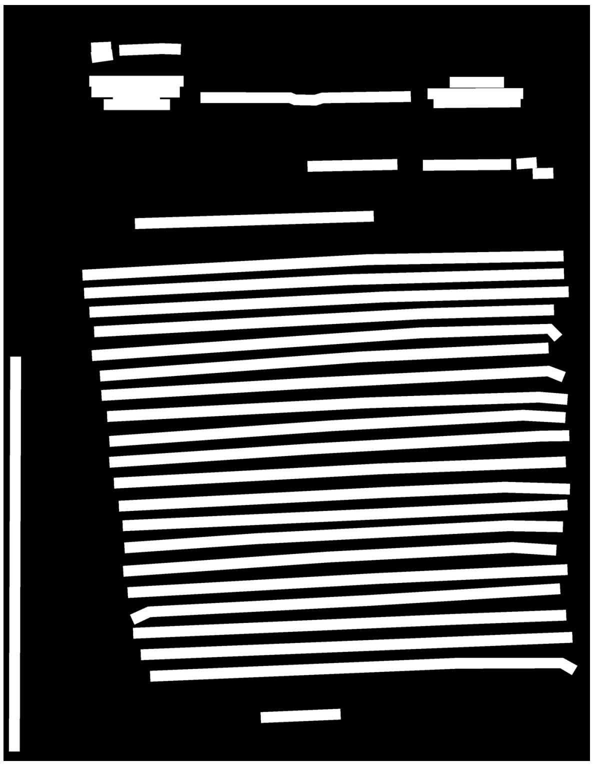


Figure 26 - Example of ground-truth image extracted from a XML file for text line detection training or validation.

3.6.1.3 Training a text line detection model (complete_train.py)

At the moment, the only model available is trained for page extraction so it cannot be used for this task. It is the time to train a model for text line detection then. For this, the material needed is a database with images ant their corresponding labels, a script to execute the training and a class.txt file which indicates the classes a pixel can be classified as. The databases that are used are DIVA-HisDB [17] and ICDAR2019 [21]. DIVA-HisDB has three different groups of images with labels: CB55, CSG18 and CSG863. Each group is composed of 40 images with their corresponding labels. Those 40 images are divided in training (20), test (10) and validation (10). There is an advantage at this point of the project: it can be seen in section 2.2.4 (or in [14]) that this network has already been tested with this images with positive results. This means that there is no need for the testing part. Instead, the testing pairs (image-label) are part of the training set. On the other hand, the validation set is still needed as it is crucial to know if the model generalizes correctly and do not overfit during the training. Taken this into account, the complete training set has 120 images with the contribution of all three groups. From this set a 25% is invested in the validation set. Therefore, 90 pairs of images-labels are for training while 30 pairs are for validation. It must be said that for this dataset no parsing was needed. It offers the ground-truth images shown in Figure 27:

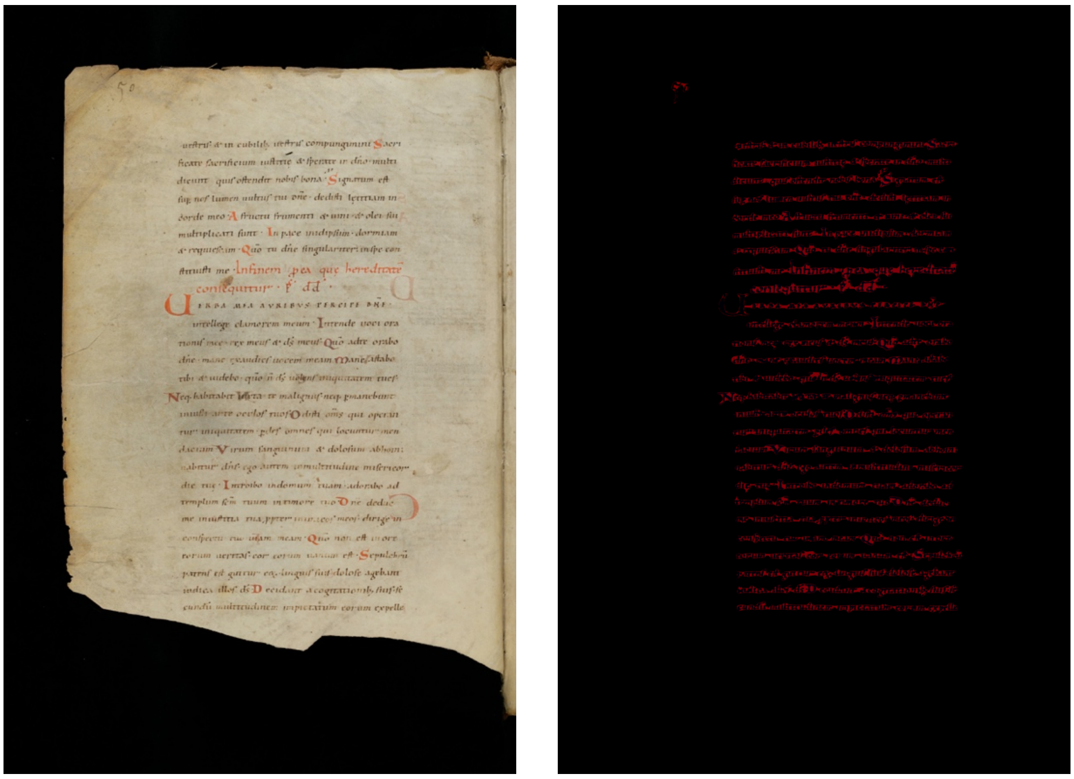


Figure 27 - Example of a pair of image-label from the DIVA dataset.

Once the images and their corresponding labels are organized, they are sent to the server. The network of DhSegment needs to know how many classes there are. The software is able to classify pixels into two or more classes (background, text and ornaments for example). This is indicated via class.txt file. This file contains a row for each class (including background class) and each row has three values for the three RGB values. Of course, each class needs to have a different code. Multiple classes are useful for layout analysis where it is necessary to classify pixels into different regions. It is also possible to assign multiple classes to the same pixel with multilabeling by adding some other parameters. In the case of this section, it is not necessary, with two classes is enough as the task is to differentiate text regions from the rest. Therefore, the class file will be composed of the next lines (the first one is black for class 0 which is background and the next one is red for class 1 which relates to text regions):

```
0 0 0

255 0 0
```

For the training, DhSegment repository has a script which will be necessary to edit and adapt it to the dataset that has been commented previously. The training script has several parts, but the interesting one is the definition of the default configuration (def default_config()). In this section is where the parameters are indicated:

```
train_data = ' /train'
eval_data = ' /validation'
model_output_dir = ' /model'
restore_model = True
classes_file = 'class.txt'
gpu = '0'
prediction_type = utils.PredictionType.CLASSIFICATION
pretrained_model_name = 'resnet50'
```

The variables train_data, eval_data and model_output_dir contain the directories related to training, evaluation and output model respectively. In case the directory of the output model already exists (which can mean that there is a model already inside it), the flag *restore_model* will stop the training if FALSE or will continue if TRUE. The class.txt file is indicated through the classes_file variable. As it was mentioned in the server installation section, out of the three available GPUs, the corresponding to this project was GPU 0 (assigned by the administrator). That is why the *gpu* variable is set to 0, to indicate which GPU must be chosen for training. For the *prediction_type* there are three possible values: CLASSIFICATION, REGRESSION (where no class.txt file is needed) or MULTILABEL. This depend on the problem to solve. In the case of text line detection, what it is necessary to know is whether a pixel belongs to a text region or not, in other words, to classify it in text region or background region. The last variable, *pretrained_model_name*, refers to the part of the neural network of DhSegment that is already trained. For further information refer section 2.2.4, [14] and [22].

Once all this is done and considering the name of the script is *train.py*, all it takes is executing the next command:

```
python3 train.py
```

After the execution of the script (the whole training takes about 18 minutes), a new directory should appear (if not already created) with the value *model_output_dir* has. Inside this directory, there are several files. The interesting one for this project is the directory called /export. Inside it there is another directory whose name is a random number (it changes every time a training process is done). Inside this random number folder the new trained model can be found (the one with the .pb termination). It is now possible to try it on some files to detect some text lines.

The results of the model trained with these images (all the groups) is not very accurate. The reason is probably because there is not enough training data. The binary mask detected is presented in Figure 28. As it can be observed, even though it seems to have learned some aspect, it has not the precision enough to apply some post-processing on it.

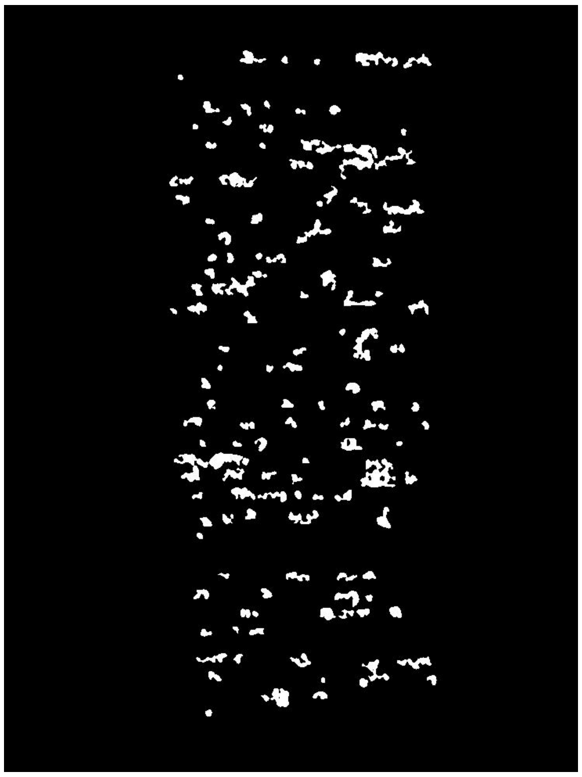
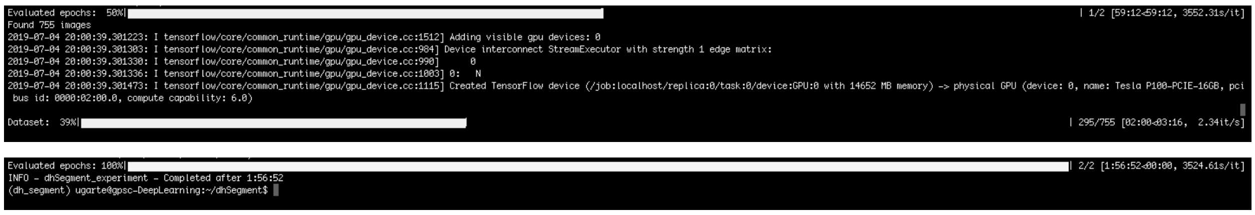


Figure 28 - Binary mask of an AGI document when applying a model trained with [17].

With these results it is tempting to train another model with another dataset. Let's recall the ground-truth images created before. Instead of 120 images, the ICDAR2019 [21] set has 984 available images with their respective labels. Therefore, let's execute the script again with this pairs images-labels:



```
Evaluated epochs:  50%|████████████████████████████████████████                                        | 1/2 [59:12<59:12, 3552.31s/it]
Found 755 images
2019-07-04 20:00:39.301223: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1512] Adding visible gpu devices: 0
2019-07-04 20:00:39.301303: I tensorflow/core/common_runtime/gpu/gpu_device.cc:984] Device interconnect StreamExecutor with strength 1 edge matrix:
2019-07-04 20:00:39.301330: I tensorflow/core/common_runtime/gpu/gpu_device.cc:990]      0
2019-07-04 20:00:39.301336: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1003] 0:   N
2019-07-04 20:00:39.301473: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 14652 MB memory) -> physical GPU (device: 0, name: Tesla P100-PCIE-16GB, pci
 bus id: 0000:02:00.0, compute capability: 6.0)

Dataset:  39%|███████████████████████████████                                                  | 295/755 [02:00<03:16,  2.34it/s]
```

```
Evaluated epochs: 100%|████████████████████████████████████████████████████████████████████████████████| 2/2 [1:56:52<00:00, 3524.61s/it]
INFO - dhSegment_experiment - Completed after 1:56:52
(dh_segment) ugarte@gpsc-DeepLearning:~/dhSegment$ █
```

Figure 29 - Part of the output of a training process of DhSegment.

In Figure 29 there is some information about the training (as it is hard to visualize it is going to be explained). It tells how the process is going and whether it is validating or training. It also outputs the time consumed for generating the model: 1:56:52. The system DhSegment also lets some graphic information to be shown using tensorboard, but as the software is being executed in the server (which has no graphic environment from a ssh tunnel) it is not possible to use this tool.

After the model is trained, it is tried the same way it has been tried the previous one. The results are in Figure 30.

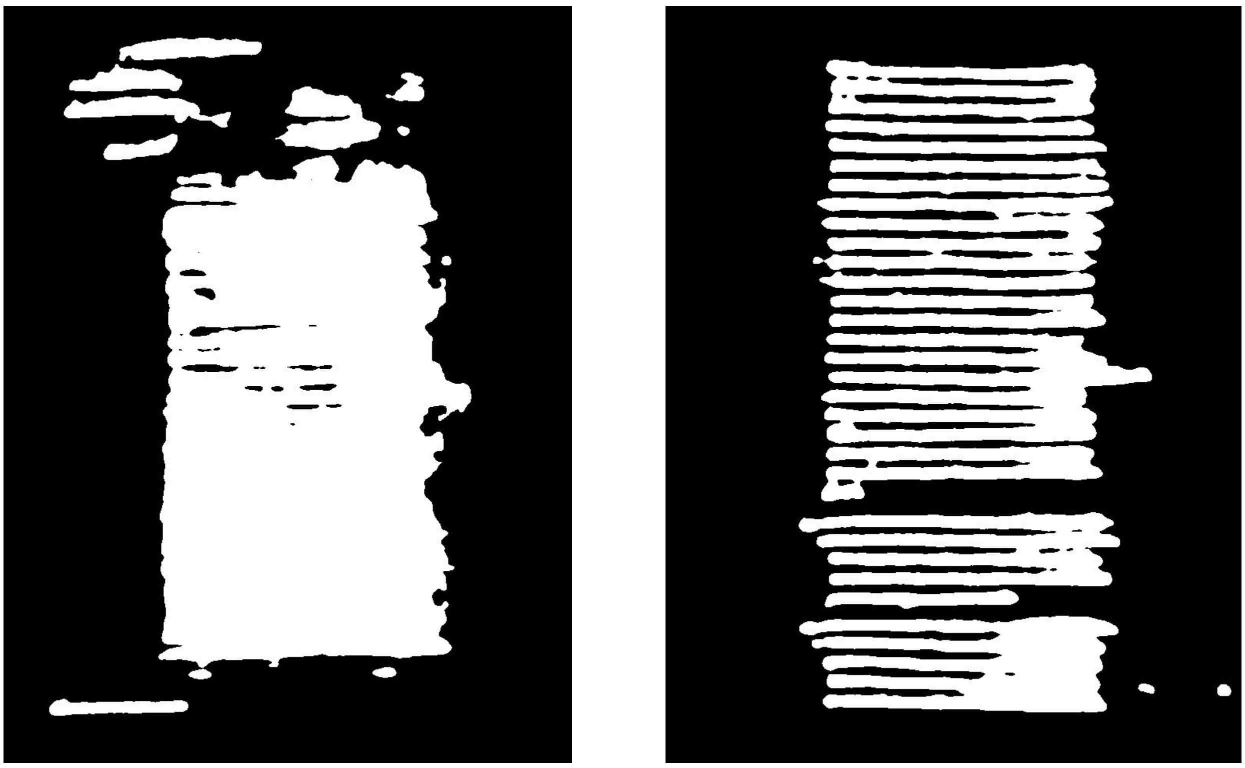


Figure 30 - Mask extracted by the model trained with a portion of ICDAR2019 dataset [21].

In Figure 30 there are two examples of masks extracted from AGI. The mask in the left corresponds to the same image in Figure 25 (and the mask of Figure 28). The result is not good at all. It has been decided to add another one with better results to show that the training and the detection actually work and present better results than the model trained with the DIVA-HisDB set. Nevertheless, none of these models could be used in a serious task of detection. In the next subsection, a model provided by the authors will be used instead as it had better results.

3.6.1.4 Text line detection in images (script3.py)

First, let's create a script that predicts and process the images. The file used for this is accumulative, in other words, it will also detect the page (the same way as in section 3.4.1.1 so no further explanation is needed). As there are two models working simultaneously, two TensorFlow sessions are required. In each session, a model will be loaded (one with the model for page extraction and the other one with the model of text line detection). The model for text line detection is provided in the GitHub repository* of the investigating group of [14], as it has not been achieved a personally trained competitive model. The model for page extraction will be the one used in the demo:

*https://github.com/dhlab-epfl/FDH_Tutorial_ComputerVision_DeepLearning/releases

```
    sess1 = tf.InteractiveSession()
    with sess1.graph.as_default():
      model_page = LoadedModel("demo/model/")
    sess2 = tf.InteractiveSession(graph=tf.Graph())
    with sess2.graph.as_default():
     model_textline=LoadedModel("/polylines”)
```

In this case, as there are two sessions running it is compulsory to use InteractiveSession() and the graph argument for the second session (for initializing). Therefore, there are two variables holding each model, one in each session: *model_page* and *model_textline*. After this, the code is similar to the one in section 3.4.1.1. Let’s assume that the coordinates for the rectangle of the page are stored in *page_coords*.

```
  with tf.Session():
      #... (the page coordinates calculations would be here)
  PAGE_info = PAGE.Page(image_width=img.shape[1],
image_height=img.shape[0],page_border=PAGE.Border(PAGE.Point.lis
t_to_point(list(page_coords))))

      prediction_outputs = model_textline.predict(filename)

      textline_probs = prediction_outputs['probs'][0,:,:,1]

      textline_probs2 = cleaning_probs(textline_probs, 2)

     extracted_page_mask = np.zeros(textline_probs.shape,
dtype=np.uint8)

      PAGE_info.draw_page_border(extracted_page_mask,
color=(255,))

     textline_mask=hysteresis_thresholding(textline_probs2,low_t
hreshold=0.3,high_threshold=0.4,candidates_mask=extracted_page_m
ask>0)
```

The lines above do something very alike to the page extraction. The first line is creating a Page object whose attributes will contain information such as the coordinates of the page or the text lines, among others. Specifically, while creating the object, *PAGE_info* is also taking the coordinates calculated previously (contained in *page_coords*). Essentially, the second line predicts in the filename given (recall that there is a loop going through the folder saving the name of a file in *filename* in each iteration). After predicting, it extracts the probabilities of class 1 (class belonging to text regions) and then it applies the method *cleaning_probs()* which, having sigma=2, will apply a Gaussian filter over the probability map and assign its output to *textline_probs2*. The next two lines are to make a binary mask with the text regions from the probability map given. In order to do so, it applies a *hysteresis_thresholding()* method. With this mask obtained, it is now possible to use the next line:

```
 lines=line_vectorization.find_lines(resize(textline_mask,img.sh
ape[:2]))
```

This line uses a method from dhSegment.post-processing that will find the lines in the binary mask given. The variable *lines* contains now the lines in a Opencv-style list of polygons.

```
 text_region = PAGE.TextRegion()
 text_region.text_lines  =  [PAGE.TextLine.from_array(line)  for
line in lines]
 PAGE_info.text_regions.append(text_region)
```

The code above, is assigning the lines in *lines* to the *text_lines* attribute of *text_region* (an object of class *PAGE.TextRegion()*). The last line allocates the object *text_region* containing the lines to the *PAGE_info.text_regions* attribute of the previously created Page object *PAGE_info*.

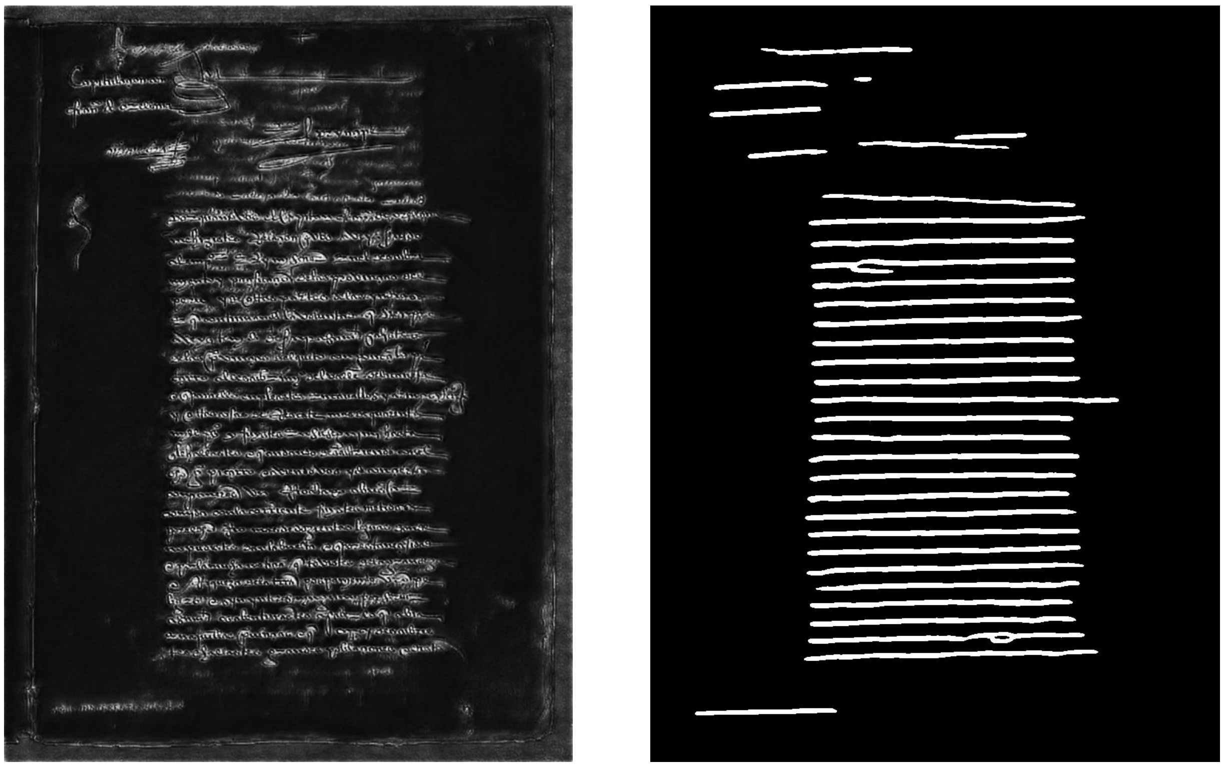


Figure 31 - On the left a probability map (textline_probs2). On the right a binary mask (textline_mask).

To conclude:

```
PAGE_info.draw_lines(plot_img,autoscale=True,fill=False,thickne
ss=15,color=(0,255,0))
```

This last line draws the lines in a copy of the original image (*plot_img* is a variable containing a copy of the image read from *filename*). Consequently, when visualizing plot_img the lines will be detected as in figure 32. It can be seen that the algorithm draws a line beneath the written line. In some cases, this post-processing ignores some parts that may have been detected as text (but are not) or concatenate some parts that might be very separated (although they belong to the same text line). In the binary mask of Figure 31, in the fourth line of the main body there is a point where the line divides into two different lines. The text line detection post-processing detects this and draws a unique line as it can be observed in Figure 32.

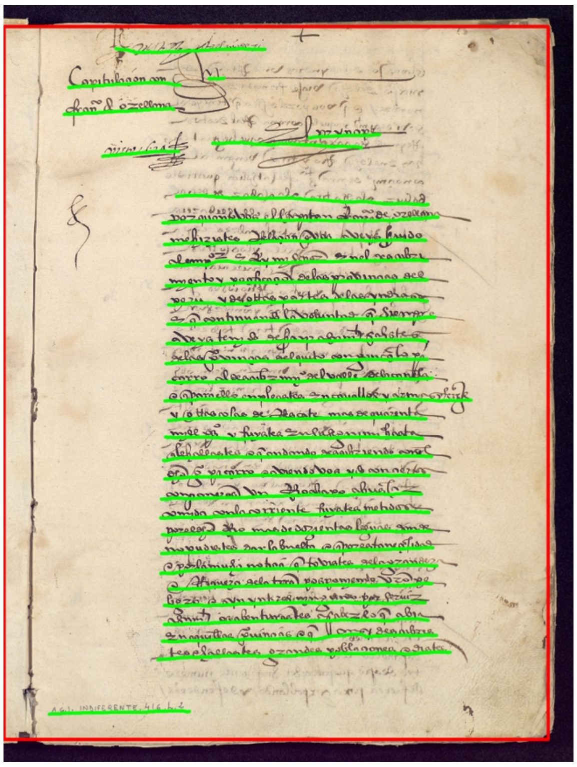


Figure 32 - Example from AGI where both tasks have been applied: a) page extraction (red), b) text line detection (green).

It has not been mentioned that the AGI dataset is divided into different folders corresponding to different authors. The way of accessing automatically to each directory, process all the images contained and then deposit the output in the output directory, replicating the file structure of the original dataset is not presented as it is not the scope of the project. In Section 4, this code will be applied to all the AGI documents and some of the results will be exposed. Figure 32 is to show how the code works.

3.6.1.5 Text region detection (script_region.py)

A different approach is presented in this section. Although it is interesting to locate with a line the text lines, it would be preferable to locate the region in which the text is contained. The text line post-processing uses an algorithm that draws a line beneath the written line, as it has been explained previously. Text region detection, on the other hand, indicates the regions where text has been detected by englobing it in a polygon. For this, the same model given by the authors will be used.

The script developed here does that. First of all, it is necessary to get the coordinates of the polygons in which the text is (*textline_mask* is the same as in text line detection section):

```
 regions1=polygon_detection.find_polygonal_regions(resize(textli
ne_mask, img.shape[:2]).astype(np.uint8), min_area=0)
```

Once the coordinates are extracted, they are assigned to a TextRegion class object, as it was done for detecting the text lines. Nevertheless, the coordinates are assigned to a different attribute (coords). The way to get all the regions is with the following loop:

```
 for cont in range(len(regions1)):
     regions = PAGE.Point.array_to_list(regions1[cont-1])
     text_region.coords = PAGE.Point.list_to_point(regions)
     PAGE_info.draw_text_regions(plot_img,       autoscale=True,
fill=False, thickness=8, color=(0,255,0))
     PAGE_info.text_regions.append(text_region)
```

It is similar to the text line detection, but it differs in some ways. Two methods from *Point* class are needed to convert the regions into compatible coordinates. The method *draw_text_regions()* draws the adapted coordinates in the image region by region in each iteration of the loop. The results are showed in Figure 33.

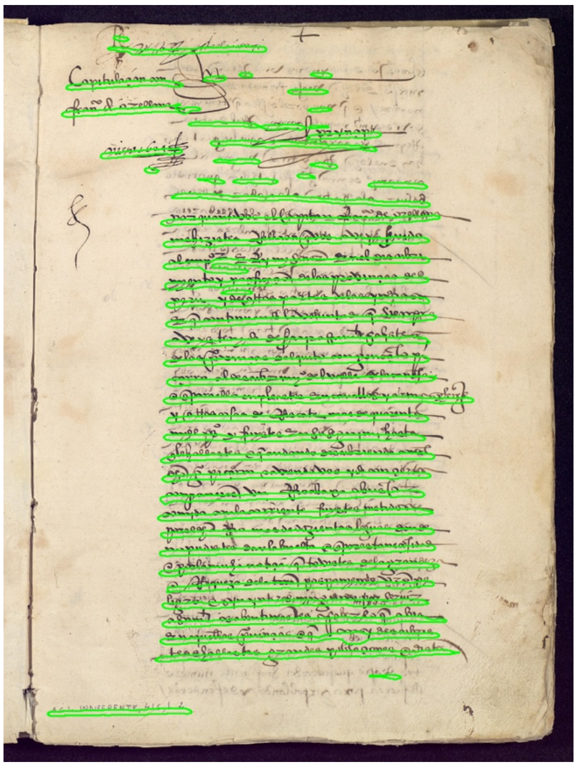


Figure 33 - Text regions (in green) calculated for an AGI document.

In this case, the are some mistakes. First of all, it seems that the model is trained for detecting the region beneath the text line, as the line do not englobe the text line well. Comparing it with the precision of the text line detection, in Figure 31, in the fourth line of the main body appear a division in a point of two text regions, for example. Text line detection corrected this, however text region detection has not. It looks like it takes objectively the white regions of the binary mask of Figure 31.

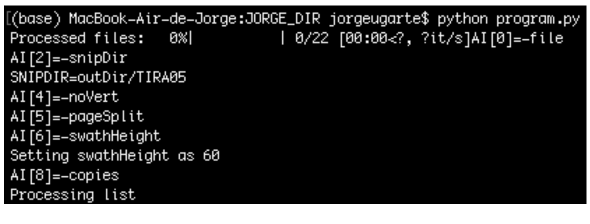
3.6.2 NeuralLineSegmenter

In the case of NeuralLineSegmenter, the process is much simpler as there is nothing to specify rather than the options of execution. Nevertheless, some code has been developed in this project just to process all the images in a unique execution of the developed script. It is mandatory to recall that the only way to make this software work is by running an executable file provided in the terminal. Therefore, if the intention is to run this program in a python script the next line is needed. The script will run in a MacBook Air 2017, the outputs are shown in Figure 34:

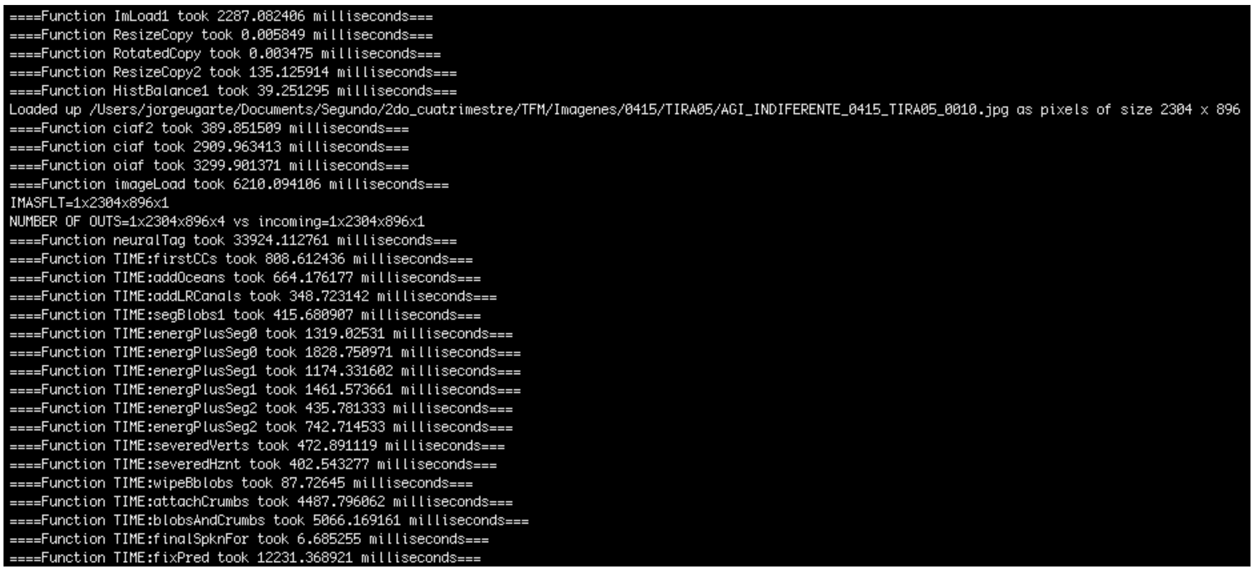
```
 subprocess.call(["java","-Xmx8g","-Dfile.encoding=UTF-8","-
jar","NeuralLineSegmenter-common-1.0-SNAPSHOT.one-jar.jar","-
file",filename,"-snipDir",outDir,"-noVert","-pageSplit","-
swathHeight","60","-copies","1" ])
```

As it can be seen in the instruction from above, all the arguments approximately correspond to the different arguments of the command run in the demo. The method *subprocess.call()* takes the first element of the list as the command to execute in a Linux environment (Mac OS is also compatible) and the rest of the elements in the list would be the arguments. The differences with the command executed in the demo are as follows: instead of specifying the directory, the name of the file (*filename*) is specified (there is a loop going through all the files in the different folder from AGI); the name of the file and the output directory for the snippets (*outDir*) are given as variables that will change along the loop; there is no -noTar option. The last change is done due to the nature of the work that is being done. If there are around 40-50 lines per page, and there is an image being generated for each line, it is a colossal number of images generated. To remediate this there is a compressed file (.tar) generated with all the images of the lines for each document processed.

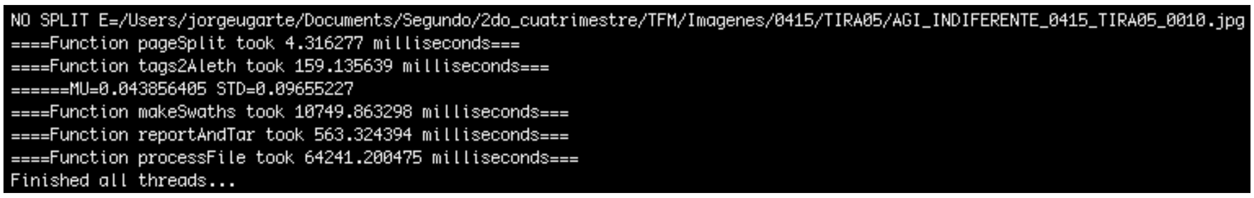
The output in the terminal for each image is the following:



```
[(base) MacBook-Air-de-Jorge:JORGE_DIR jorgeugarte$ python program.py
Processed files:   0%|          | 0/22 [00:00<?, ?it/s]AI[0]=-file
AI[2]=-snipDir
SNIPDIR=outDir/TIRA05
AI[4]=-noVert
AI[5]=-pageSplit
AI[6]=-swathHeight
Setting swathHeight as 60
AI[8]=-copies
Processing list
```



```
====Function ImLoad1 took 2287.082406 milliseconds===
====Function ResizeCopy took 0.005849 milliseconds===
====Function RotatedCopy took 0.003475 milliseconds===
====Function ResizeCopy2 took 135.125914 milliseconds===
====Function HistBalance1 took 39.251295 milliseconds===
Loaded up /Users/jorgeugarte/Documents/Segundo/2do_cuatrimestre/TFM/Imagenes/0415/TIRA05/AGI_INDIFERENTE_0415_TIRA05_0010.jpg as pixels of size 2304 x 896
====Function ciaf2 took 389.851509 milliseconds===
====Function ciaf took 2909.963413 milliseconds===
====Function oiaf took 3299.901371 milliseconds===
====Function imageLoad took 6210.094106 milliseconds===
IMASFLT=1x2304x896x1
NUMBER OF OUTS=1x2304x896x4 vs incoming=1x2304x896x1
====Function neuralTag took 33924.112761 milliseconds===
====Function TIME:firstCCs took 808.612436 milliseconds===
====Function TIME:addOceans took 664.176177 milliseconds===
====Function TIME:addLRCanals took 348.723142 milliseconds===
====Function TIME:segBlobs1 took 415.680907 milliseconds===
====Function TIME:energPlusSeg0 took 1319.02531 milliseconds===
====Function TIME:energPlusSeg0 took 1828.750971 milliseconds===
====Function TIME:energPlusSeg1 took 1174.331602 milliseconds===
====Function TIME:energPlusSeg1 took 1461.573661 milliseconds===
====Function TIME:energPlusSeg2 took 435.781333 milliseconds===
====Function TIME:energPlusSeg2 took 742.714533 milliseconds===
====Function TIME:severedVerts took 472.891119 milliseconds===
====Function TIME:severedHznt took 402.543277 milliseconds===
====Function TIME:wipeBblobs took 87.72645 milliseconds===
====Function TIME:attachCrumbs took 4487.796062 milliseconds===
====Function TIME:blobsAndCrumbs took 5066.169161 milliseconds===
====Function TIME:finalSpknFor took 6.685255 milliseconds===
====Function TIME:fixPred took 12231.368921 milliseconds===
```



```
NO SPLIT E=/Users/jorgeugarte/Documents/Segundo/2do_cuatrimestre/TFM/Imagenes/0415/TIRA05/AGI_INDIFERENTE_0415_TIRA05_0010.jpg
====Function pageSplit took 4.316277 milliseconds===
====Function tags2Aleth took 159.135639 milliseconds===
======MU=0.043856405 STD=0.09655227
====Function makeSwaths took 10749.863298 milliseconds===
====Function reportAndTar took 563.324394 milliseconds===
====Function processFile took 64241.200475 milliseconds===
Finished all threads...
```

Figure 34 - Output of NeuralLineSegmenter for an image (some data has not been presented because it had no interest for the project).

In Figure 34 several conclusions can be extracted. It can be seen that in the first place, the program outputs the options included in the execution. After this, the process begins. The software returns the amount of time consumed for each of the functions used. The total amount is at the end and is about 64 seconds. When analysing, it can be concluded that the most expensive function in terms of time (and resources too but there is no output for that) is the one that applies the neural network, as it can be expected.

4 RESULTS

A man without patience is a lamp without oil.

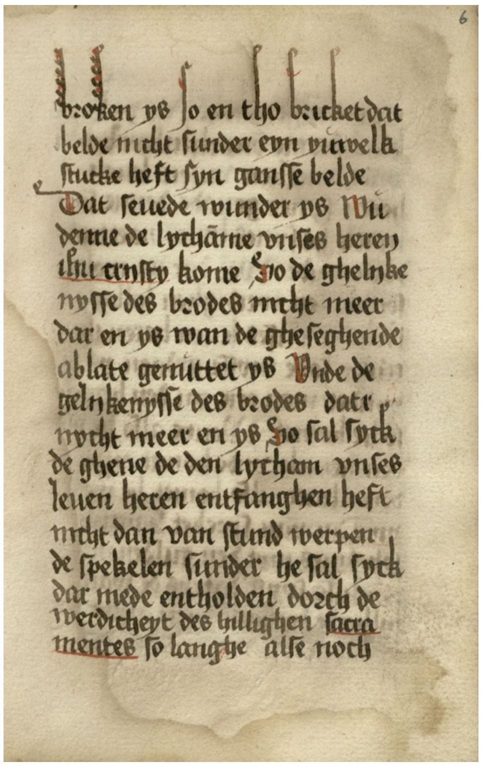
- André Segovia -

4.1 Results for existent datasets

Before exposing the results in AGI dataset, some other known sets are presented in this section. Some of them are basically historical documents. However, there are sets that contain also normal handwritten documents.

4.1.1 ICDAR 2019 Competition

This database is the most recent one. It has also been used for training in this project. This dataset contains, not only historical documents but a huge set of handwritten documents. Figure 35 shows an example of this set. To maintain the scope of the project it has been chosen an image that seems to be an old document.


6

broken ys So en tho bricket dat
belde nicht ſunder eyn yuwelk
ſtucke heft ſyn ganſſe belde
Dat ſeuede wunder ys Wu
denne de lycharme vnſes heren
ihu crıſty kome So de ghelyke
nyſſe des brodes nicht meer
dar en ys wan de gheſeghende
ablate genuttet ys Unde de
gelykenyſſe des brodes dat
nycht meer en ys So ſal ſyck
de ghene de den lycham vnſes
leuen heren entfanghen heft
nicht dan van ſtund werpen
de ſpekelen ſunder he ſal ſyck
dar mede entholden dorch de
werdicheyt des hilllighen ſacra
mentes ſo langhe alſe noch

Figure 35 – Example of an image from the ICDAR2019 set [21].

4.1.1.1 DhSegment

In Figure 36 it can be observed that DhSegment detects the baseline well (slight problems appear at the end of one line). The problem of the text region detection is quite obvious here as it can be seen that it does not detect the text area but the region beneath the line. Page extraction is done perfectly, although it must be said that the task for this image is not very challenging.

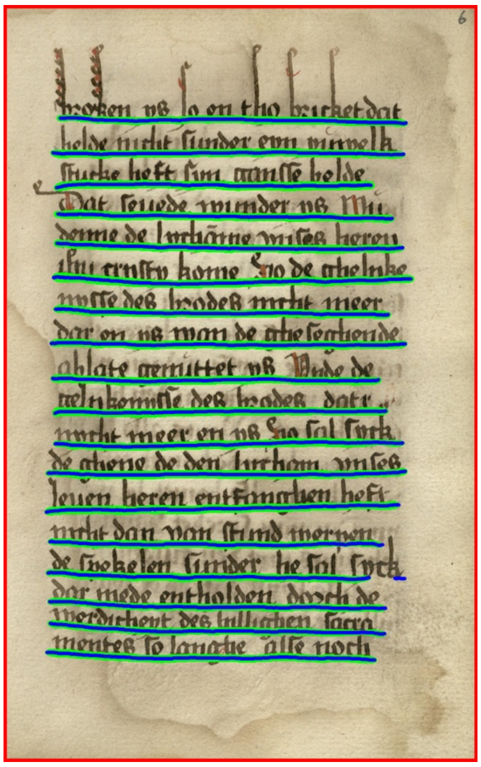


Figure 36 - Extraction of DhSegment of an ICDAR2019 image [21].

4.1.1.2 NeuralLineSegmenter

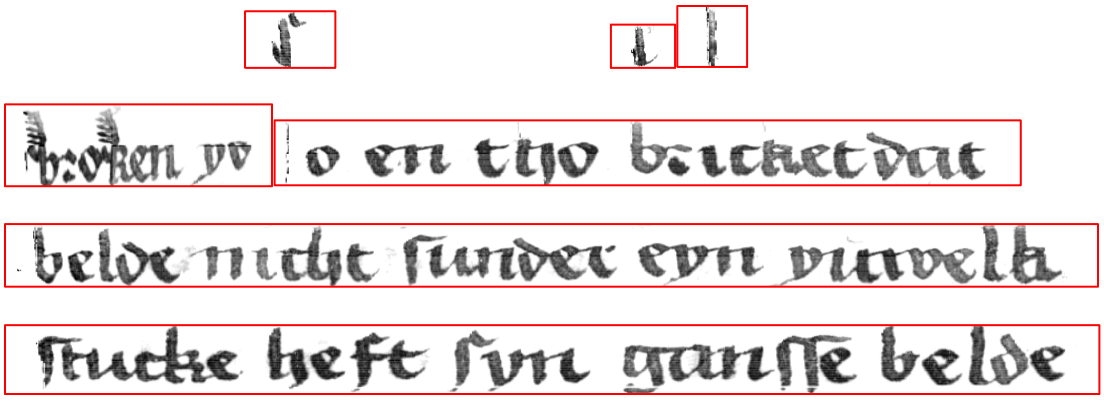


Figure 37 - Some of the snippets extracted by NeuralLineSegmenter of an ICDAR2019 image [21].

NeuralLineSegmenter presents good extractions (Figure 37). It is able to get the lines quite precisely. However, it gets confused when getting the long vertical lines from some letters of the first line. After that, the extraction works outstandingly.

4.1.2 DIVA-HisDB

The DIVA-HisDB [17] set has also been used for training with worse results than the ICDAR2019 set. It is probably due to the lack of images and generality. The set, as it has been mentioned before consists of three groups. As all three of them are very much alike, only one example will be exposed.

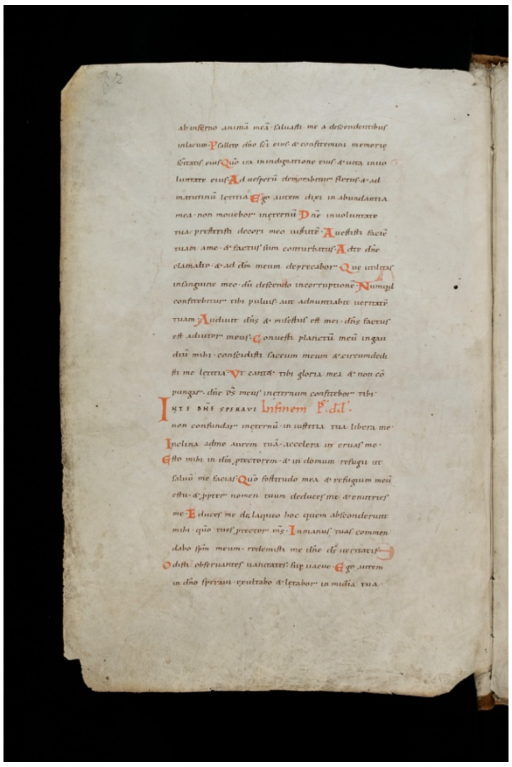


Figure 38 - Example of an image from the DIVA-HisDB set [17].

4.1.2.1 DhSegment

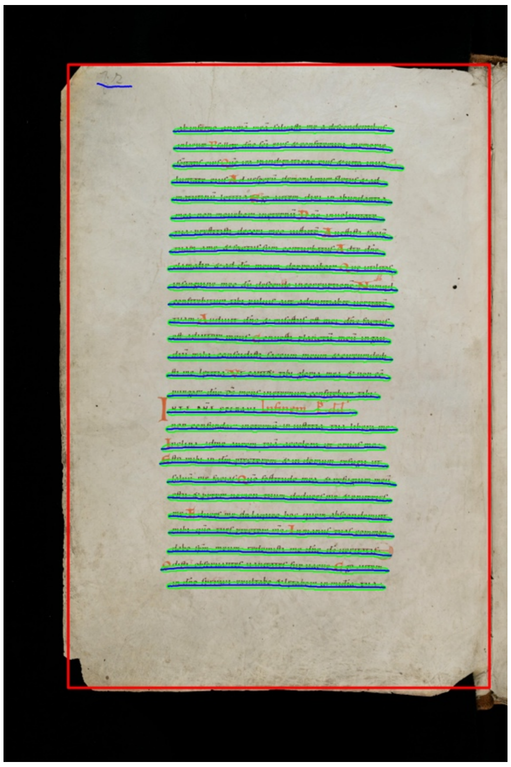
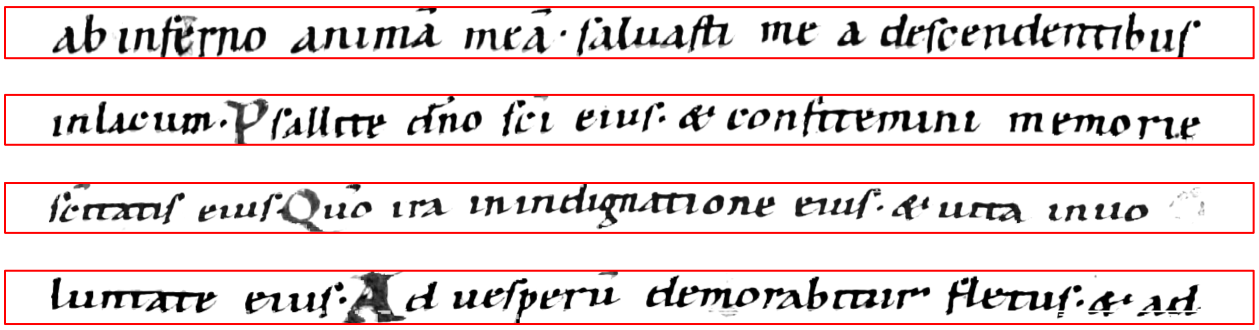


Figure 39 - Extraction of DhSegment in an image of the DIVA set [17].

The results here are almost perfect (Figure 39). It is true that the document is very well presented and optimal conditions. Text line detection here works fine except for the top-left corner, where it has detected a line where there is not. Text region detections keeps working the same way. In this case page extraction seem to have work well when detecting the vertical line that separates the page from the one besides it. This shows that the model is able to detect those type of lines, but maybe it does not generalize enough.

4.1.2.2 NeuralLineSegmenter



ab inferno animā meā· ſaluaſti me a deſcendentibuſ

in lacum·Pſallite dno ſci eiuſ· & confitemini memorie

ſcitatiſ eiuſ Quo ira in indignatione eiuſ· & uita in uo

luntate eiuſ· Ad ueſperū demorabitur fletuſ· & ad

Figure 40 – Some of the snippets extracted by NeuralLineSegmenter [17].

As with DhSegment, the results (Figure 40) here are wonderful. The conditions of the document and the presentation allow NeuralLineSegmenter to extract the lines perfectly.

4.1.3 Serena

These images have been given by José Carlos Aradillas who has helped in some aspects of this project. Again, this set has multiple groups. For the generalization of the results, it has been chosen a document in really bad conditions to see how the methods work on it.

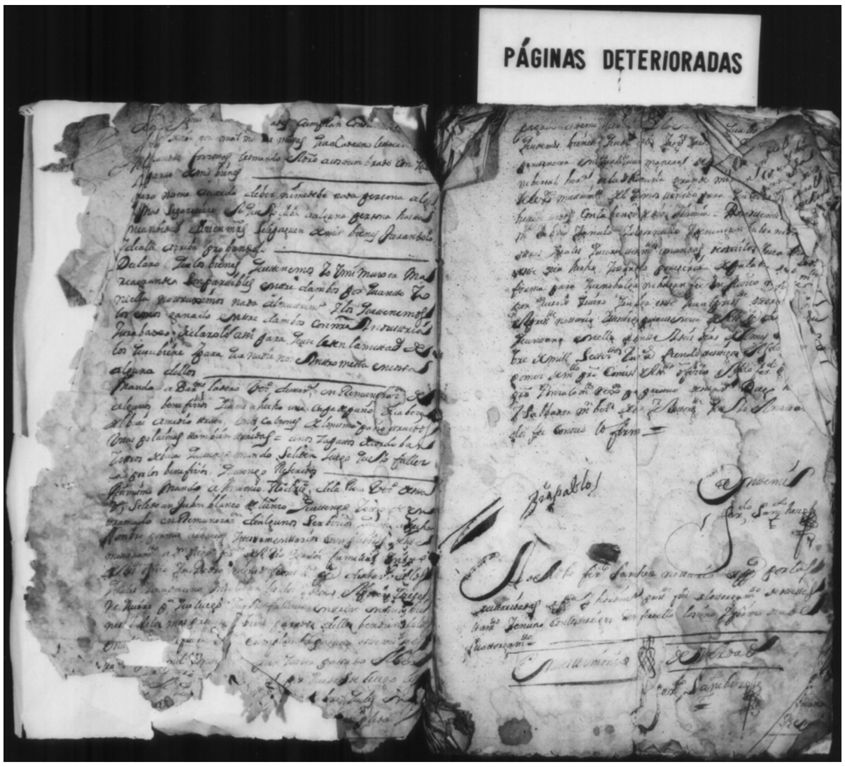


Figure 41 – Example of an image in bad conditions from Serena set.

4.1.3.1 DhSegment

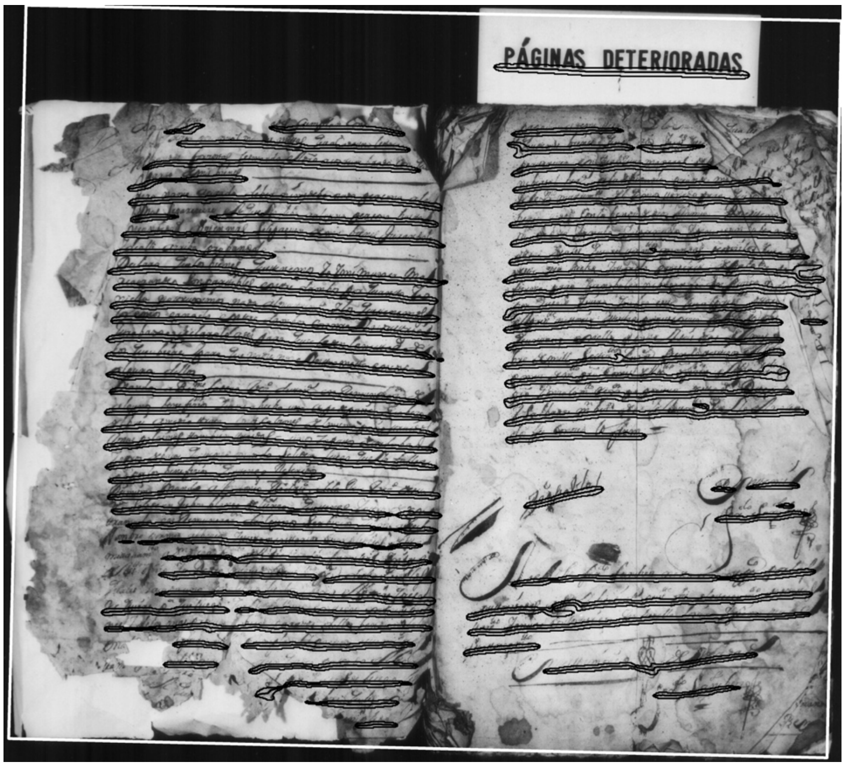


Figure 42 - Extraction by DhSegment on a Serena set image.

It must be admitted that, taking into account the conditions of the document, the results are quite notorious. The text lines are detected accurately, making some mistakes in the top-right corner; nevertheless, those lines look quite difficult to detect. It even ignores the long lines at the end of the first two paragraphs. Text regions, as expected, keep detecting the region beneath the line. About page extraction, the result is not bearable. It does not detect the two pages and it gets confused with the top blank rectangle.

4.1.3.2 NeuralLineSegmenter

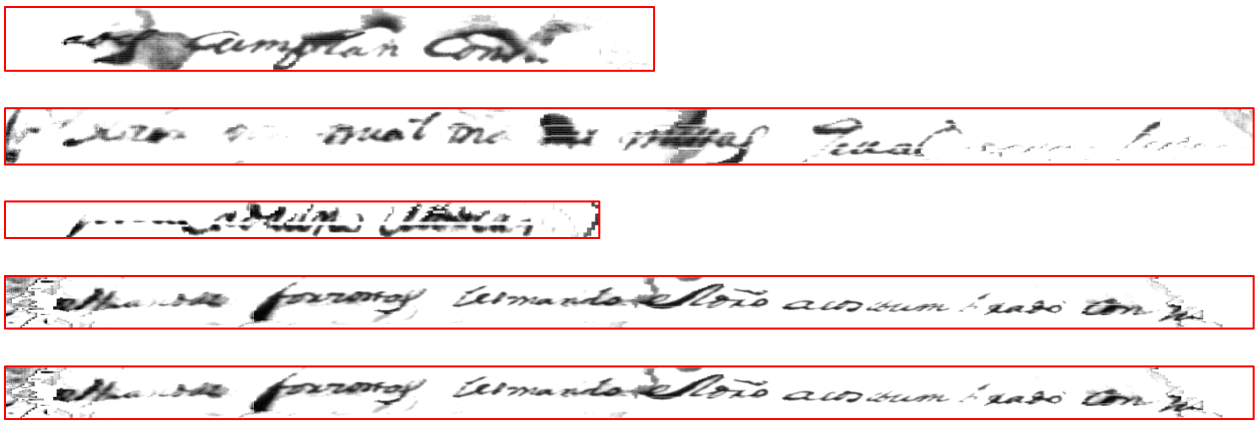


Figure 43 - Some of the snippets extracted by NeuralLineSegmenter.

Taking into account the conditions of the documents it must be admitted that NeuralLineSegmenter works fine extracting the snippets (Figure 43). For these images (the one in the example and the rest of the dataset), the option *-pageSplit* has been added so that it detects if the document has one or two pages. So, the extractions of NeuralLineSegmenter do not take a whole line through both pages, but separately from both pages.

4.2 Results for documents in *Archivo General de Indias*

The *Archivo General de Indias* dataset is the main dataset of this project. As it has been mentioned before, the goal of the project is to segment the documents of the AGI dataset. These documents have been purchased from the *Archivo General de Indias*. Actually, this dataset can be found in the official website in low quality and anyone can download it and work on the documents. The reason they had to be purchased is that the ones received are high quality images. The size of these images is approximately 2900x4000.

Even though the work done in this project is general enough to apply it to diverse datasets, the end of it comes when it is applied to this dataset. This data set consists of different groups of documents. They are divided as follows. Three groups called *INDIFERENTE_0415,* *INDIFERENTE_0416, INDIFERENTE_0422.* Inside the first group there are three subgroups called *TIRA03* (6 images), *TIRA04* (5 images) and *TIRA05* (22 images). The second group has directly 12 images. The third group has two categories called *TIRA01* (2 images) and *TIRA02* (2 images).

As there are too many documents to show in this text, the result of one of each group or subgroup will be presented. Both software will be tried on the dataset.

4.2.1 DhSegment

For DhSegment, the script_region.py is used. As the model trained in this project did not have the expected results, the models offered in the repository* are used for page extraction one of them and text line detection and text regions the other one. The script is modified to make all three tasks at the same time so that the output images have everything detected: page, text lines and text regions.

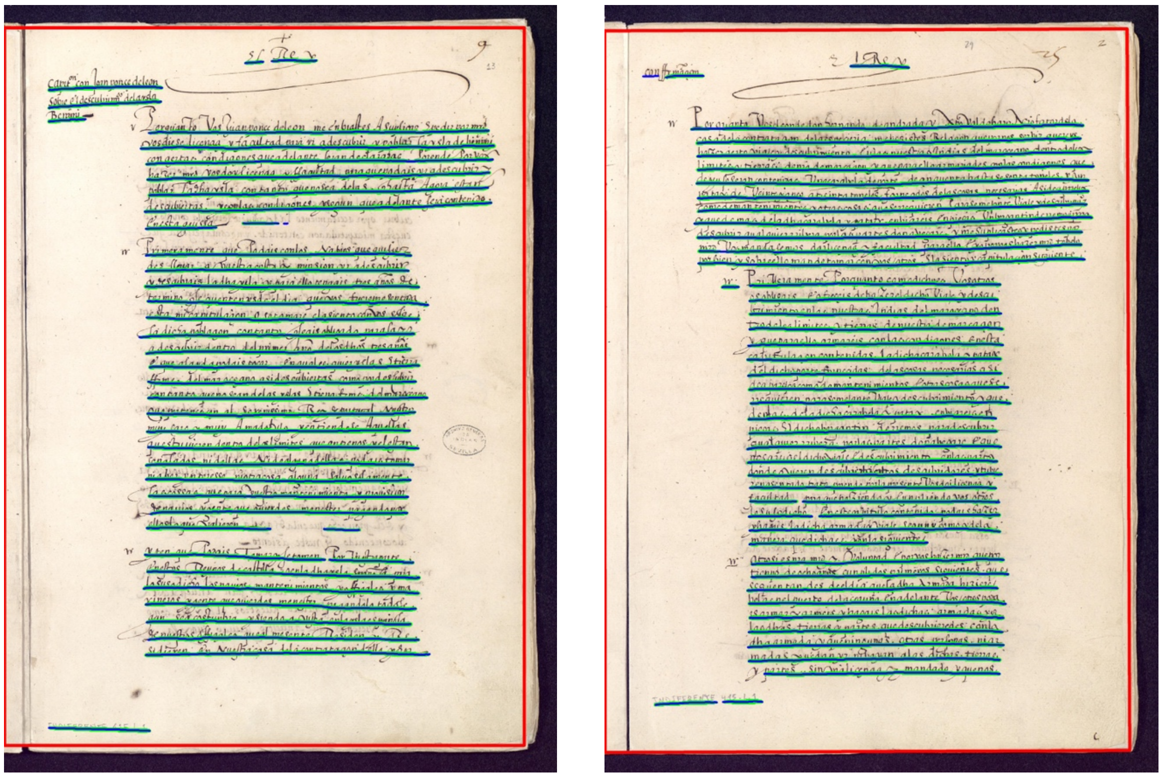


Figure 44 - AGI_INDIFERENTE_0415_TIRA03_0001 (left) and AGI_INDIFERENTE_0415_TIRA04_0001(right) processed with DhSegment. In red the page extraction, in blue the text line and in green the text region.

*https://github.com/dhlab-epfl/FDH_Tutorial_ComputerVision_DeepLearning/releases

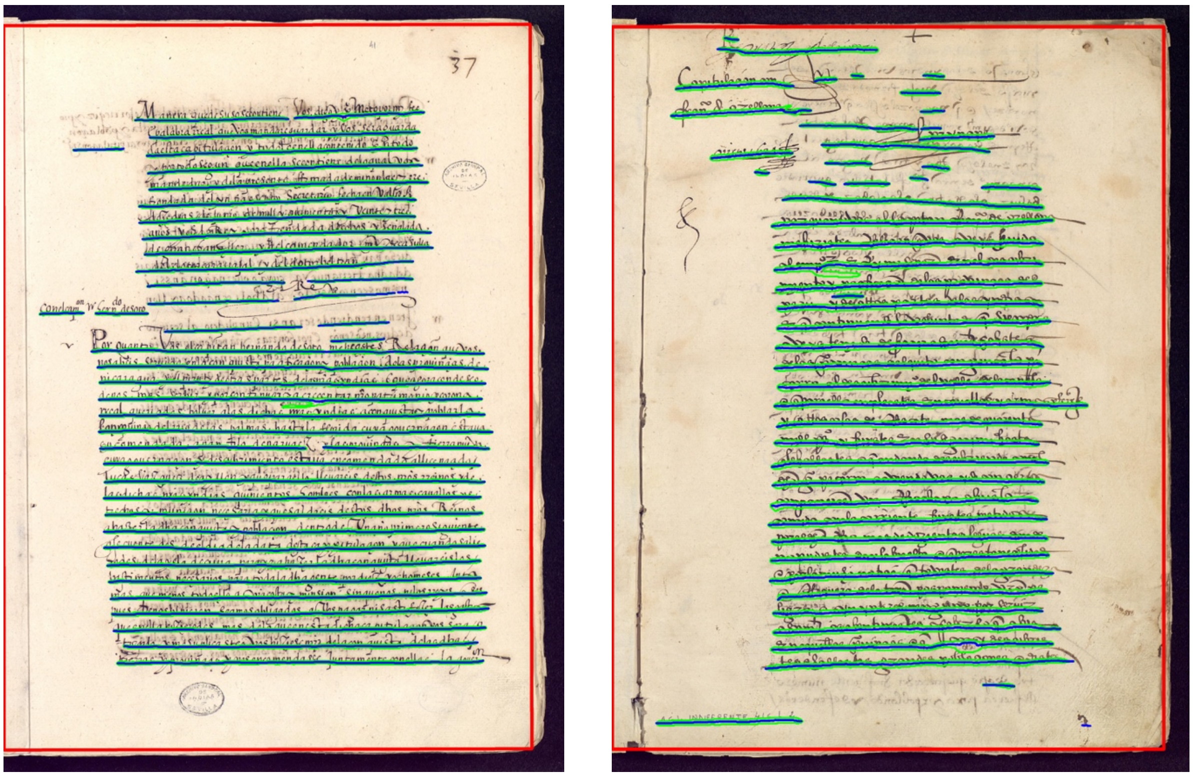


Figure 45 - ***AGI_INDIFERENTE_0415_TIRA05_0001 (left) and AGI_INDIFERENTE_0416_TIRA06_0001(right) processed with DhSegment. In red the page extraction, in blue the text line and in green the text region.***

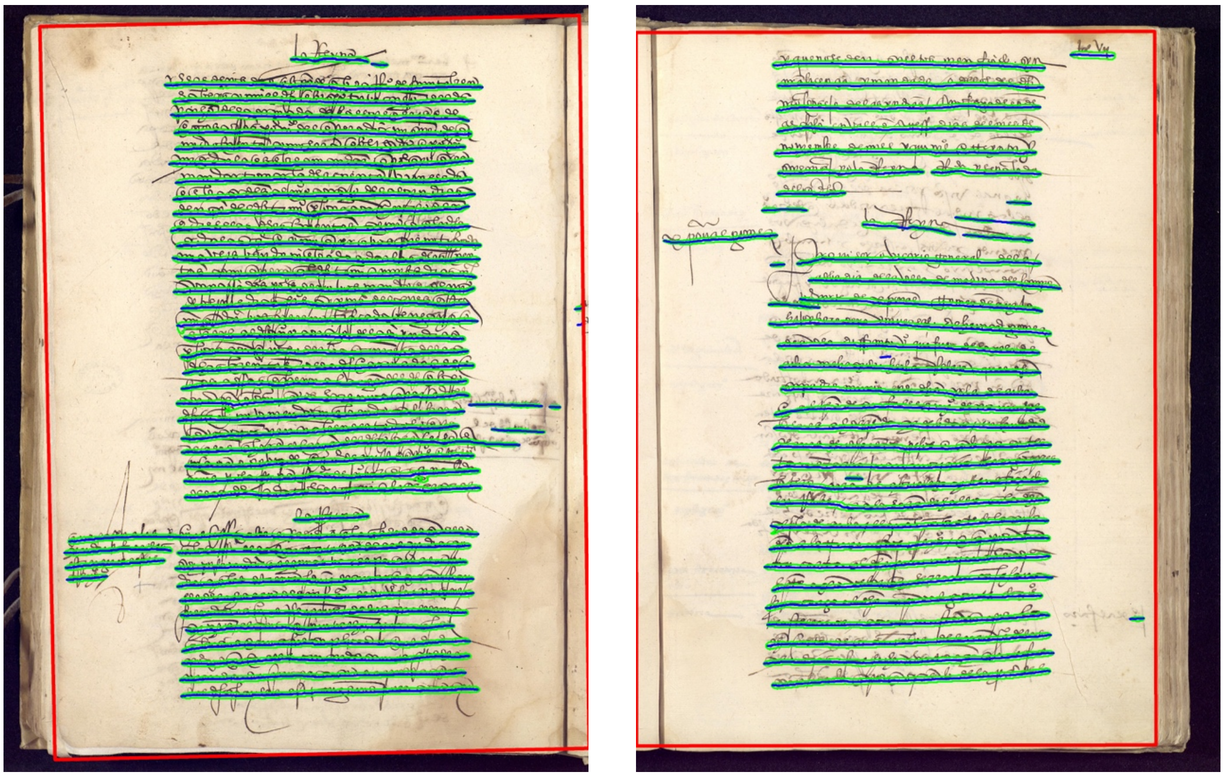


Figure 46 - AGI_INDIFERENTE_0422_TIRA01_0001 (left) and AGI_INDIFERENTE_0422_TIRA02_0001(right) processed with DhSegment. In red the page extraction, in blue the text line and in green the text region.

The conclusions that can be extracted from the outputs shown are diverse. First of all, it has to be taken into account that the software is presented in its article, but it is here when it is applied to a real case. In the article, there some comparison made with other software and some data that shown that it was a promising solution. Nevertheless, the output of DhSegment to the AGI dataset was slightly unpredictable (as it is the nature of neural networks). For the case, the model has shown to work well detecting pages and lines, although a lot of data is needed to train a model, like it has been noted when a new model has been tried to be trained.

The results for page extraction are quite precise. In general, it locates really well the top and the bottom edges. On the other hand, it slightly fails when locating the side edges. In this case, it does not locate the middle line that divides the book into two different pages. Nonetheless, when some pages edges appear behind the front one, it manages fairly fine to locate the edge corresponding to the front page.

When talking about text line detection, DhSegment functions satisfactorily detecting the line and drawing the base line. Almost every time, the software gets it right when identifying the text in middle (could be considered the principal text) and some comments round it. However, it gets confused sometimes by lighted written parts. This light text appears to be whether text from the other side of the documents or text that has been deteriorated.

In the text region detection part, it seems that is not working very fine. It happens to perceive the text region below the text instead of above it. In addition, it does not seem to embrace the whole text contained in the line (the text "escapes" from below and above). This seems to be problem of the post-processing rather than the model because the text is detected, but the region is not well assigned in the document.

It must be added that in TIRA03, TIRA04 and TIRA05, the threshold to extract the binary mask from the probability map are more exigent. This prevent from some false positives (for example, the lines that are transparent from the other side of the page). Note that there are slight (but important) differences with TIRA01, TIRA02 and TIRA06.

4.2.2 NeuralLineSegmenter

On the other hand, the other software applied to these documents is NeuralLineSegmenter. The way of processing the images in this case is different and it is done locally in the CPU. Also, the outputs are different. With this software the images are split into a bunch of text lines snippets. The examples exposed here correspond to the same in the DhSegment section, however they presented in different way. In this case, some extracted lines from each image is shown as there are too many lines to present and each of them is an actual image.

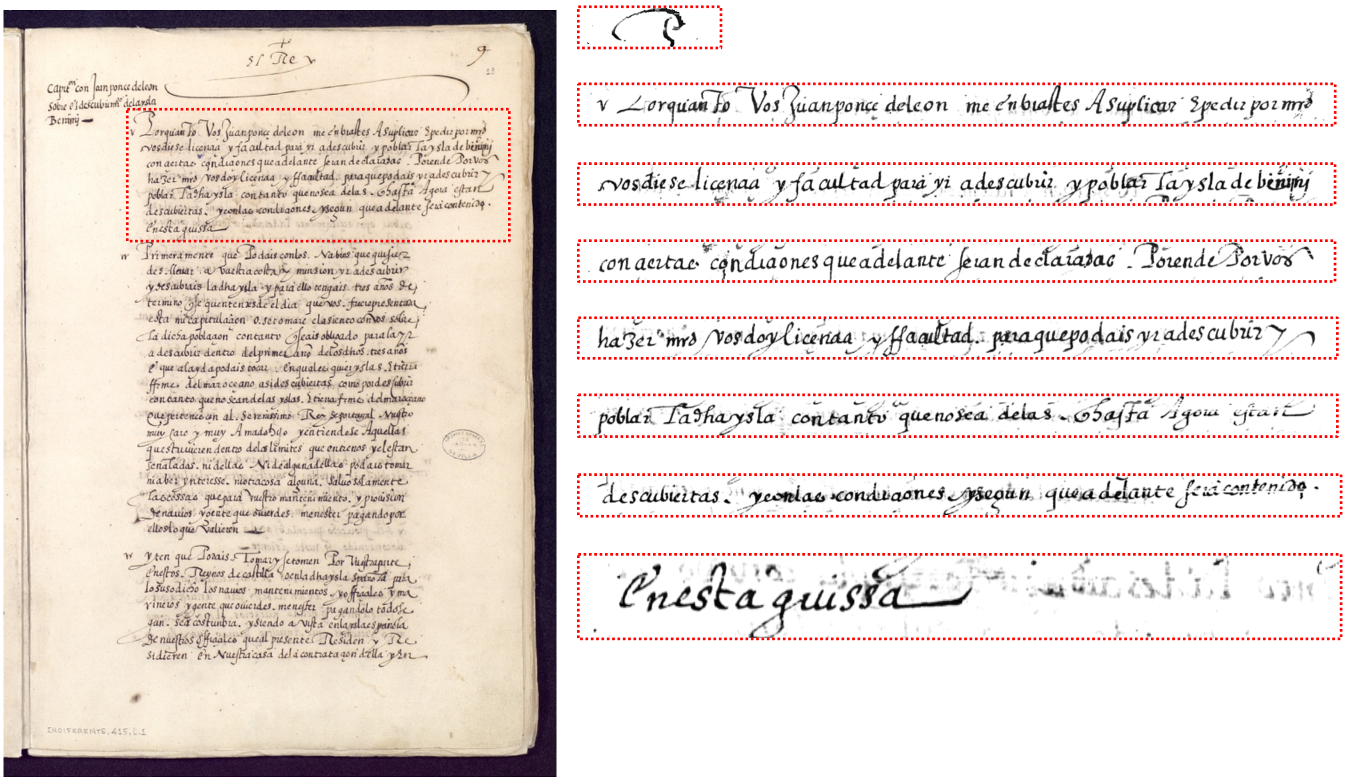


Figure 47 - AGI_INDIFERENTE_0415_TIRA03_0001 (left) processed with NeuralLineSegmenter. The red square (left) correspond to the extracted text lines (right).

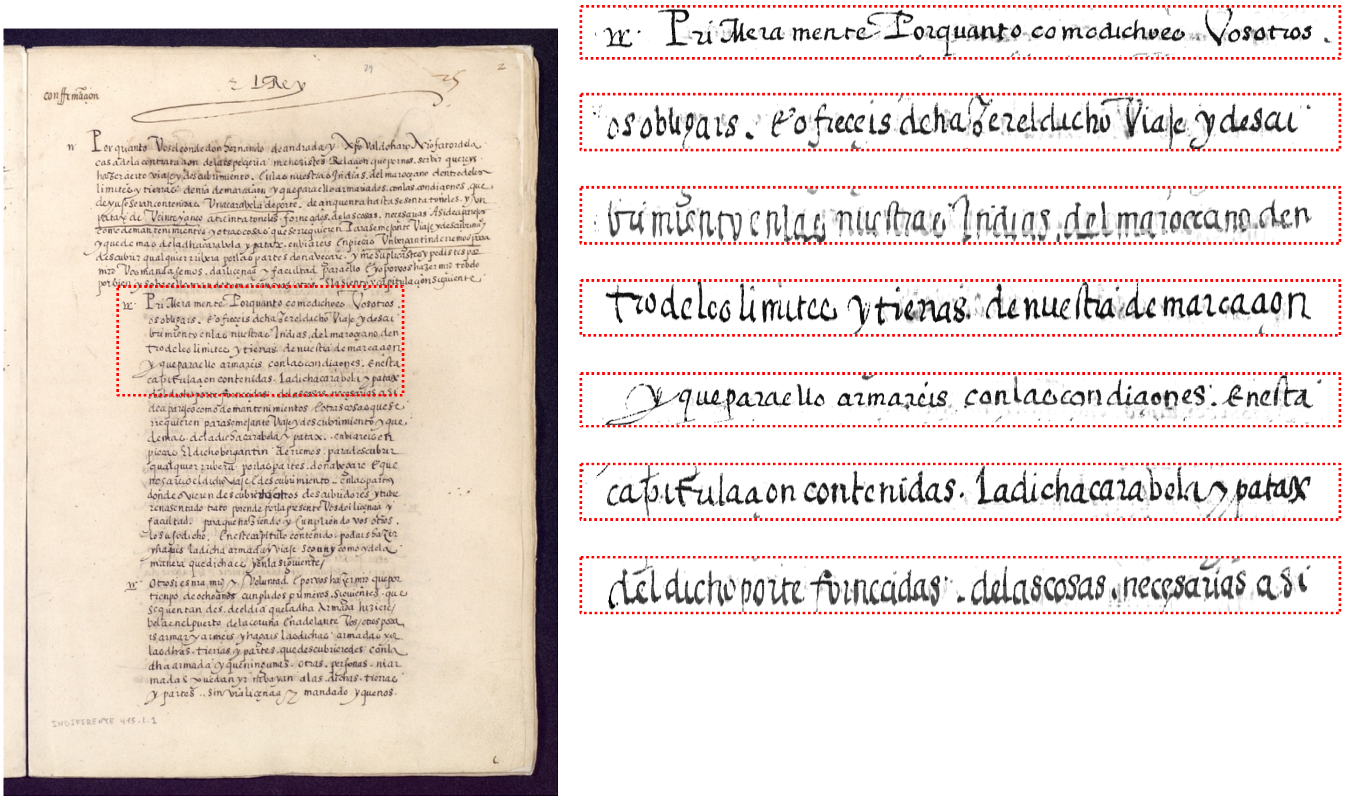


Figure 48 - AGI_INDIFERENTE_0415_TIRA04_0001 processed with NeuralLineSegmenter. The red square (left) correspond to the extracted text lines (right).

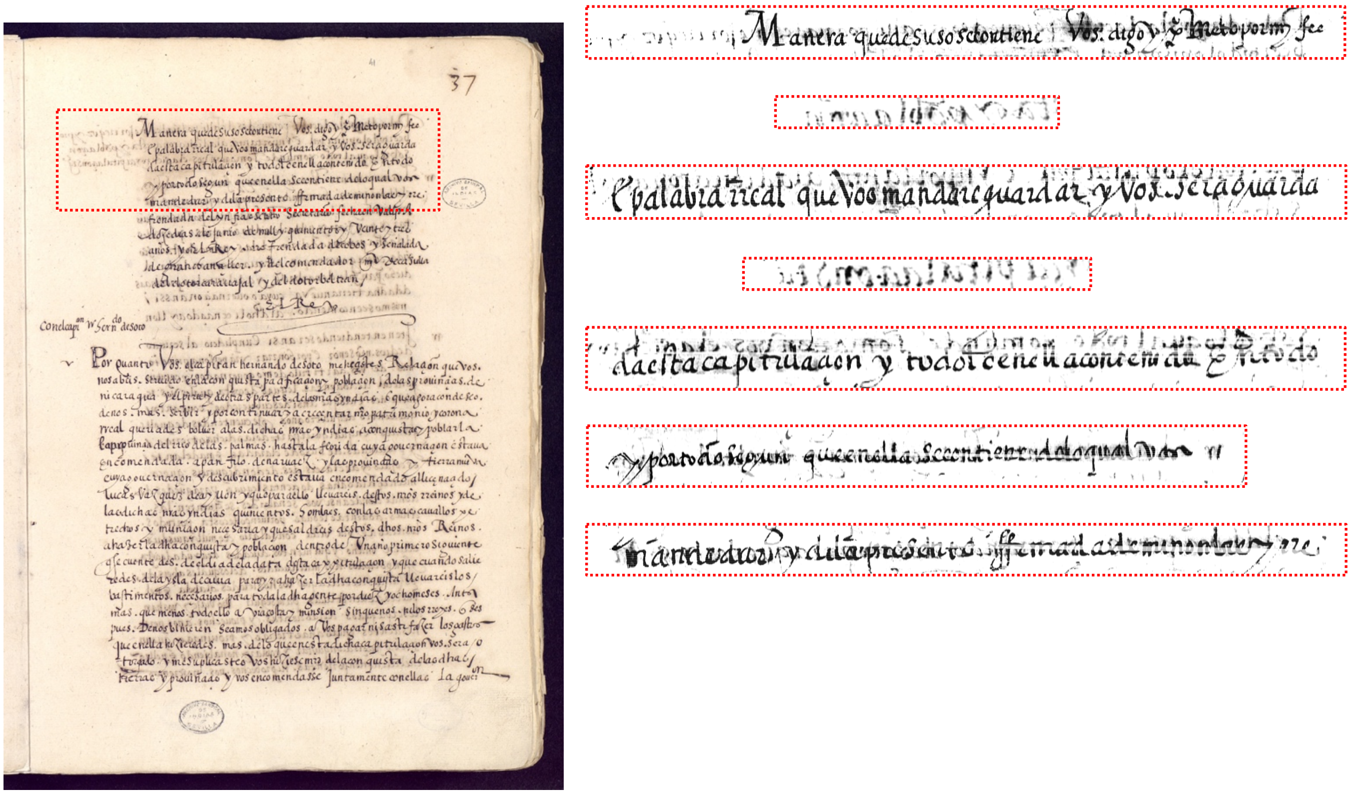


Figure 49 - AGI_INDIFERENTE_0415_TIRA05_0001 processed with NeuralLineSegmenter. The red square (left) correspond to the extracted text lines (right).

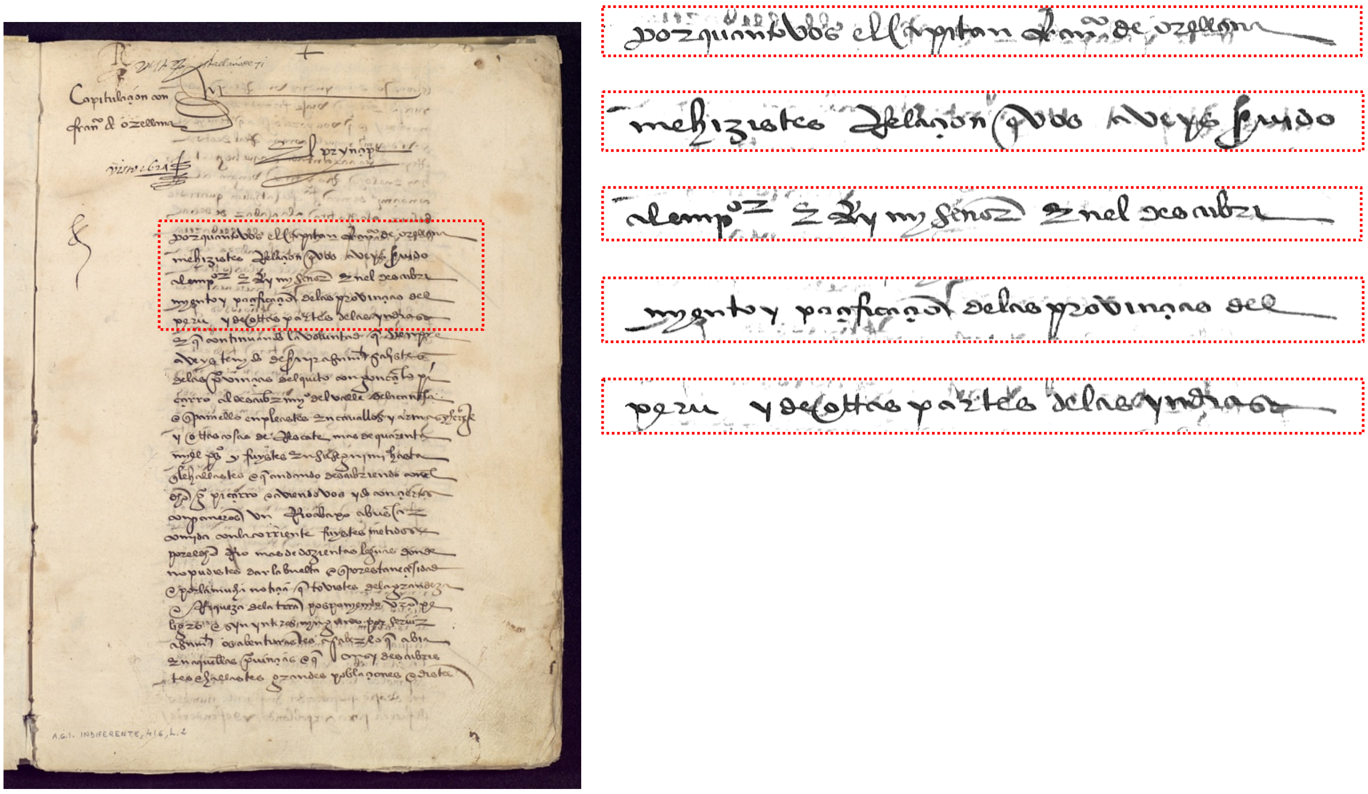


Figure 50 - AGI_INDIFERENTE_0416_TIRA06_0001 processed with NeuralLineSegmenter. The red square (left) correspond to the extracted text lines (right).

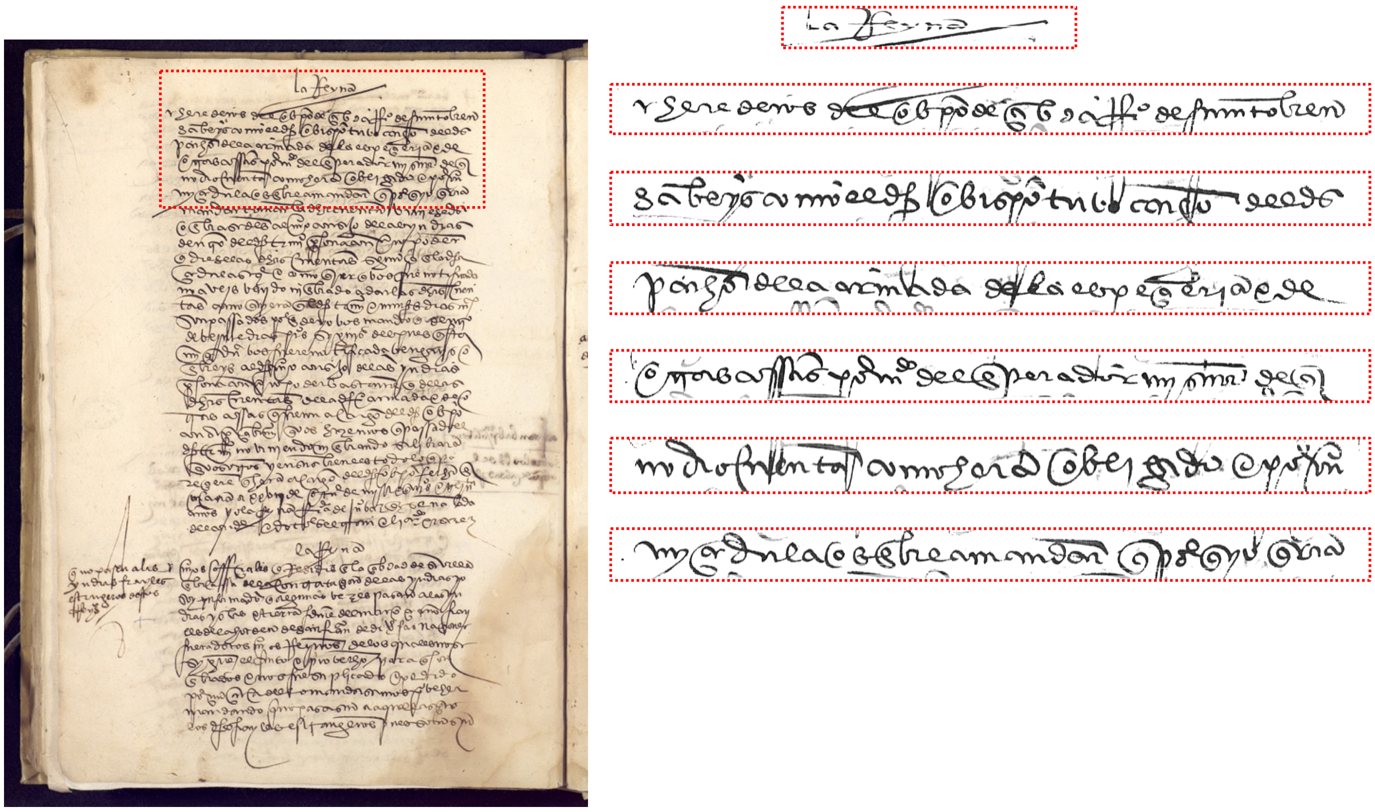


Figure 51 - AGI_INDIFERENTE_0422_TIRA01_0001 processed with NeuralLineSegmenter. The red square (left) correspond to the extracted text lines (right).

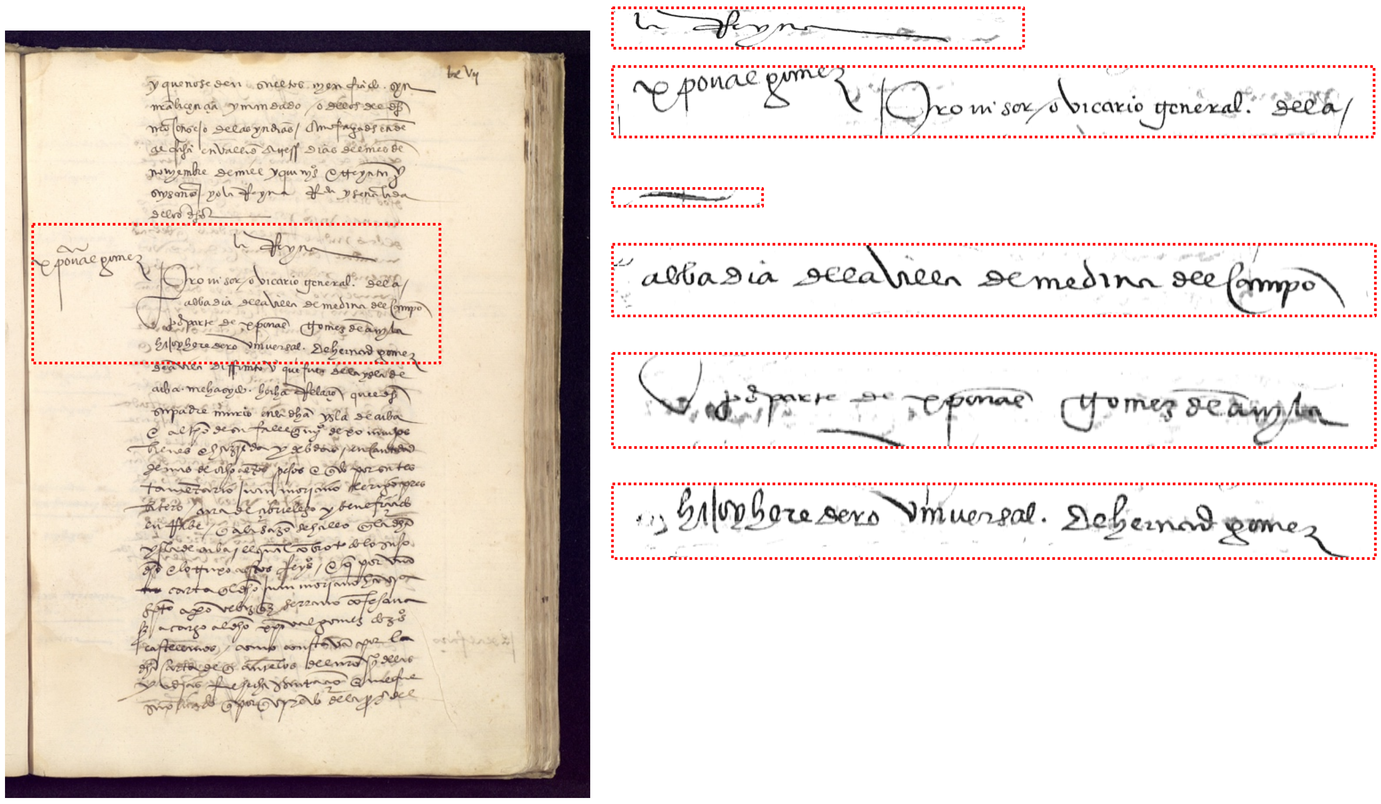


Figure 52 - AGI_INDIFERENTE_0422_TIRA02_0001 processed with NeuralLineSegmenter. The red square (left) correspond to the extracted text lines (right).

For the case of NeuralLineSegmenter, the results are outstanding. It can be observed that the lines are accurately extracted. It works really fine when detecting a line and extracting, although it is true that sometimes it detects the snippet differently when there is light text beside the principal text: sometimes it assigns separate snippets for each type of text and sometimes it takes the whole line including both types of text in the same snippet. As it happened with DhSegment, the algorithm confuses light text. Another slight issue happens to occur. When there is a letter (normally capital letter at the beginning of a paragraph) it appears to split it in different snippets. This issue can be considered for future work because the software, as it is, seems to be working good enough for the purposes of this project.

5 CONCLUSIONS AND FUTURE IDEAS

Artificial Intelligence is likely to became either the best or the worst thing to happen to humanity.

- Stephen Hawking -

To conclude, let’s say that more time has been invested in working and developing the side of DhSegment than the time spent on NeuralLineSegmenter. This is obvious as DhSegment is much more flexible to use and has more applications. When comparing the two software in the AGI dataset, better results have been given by NeuralLineSegmenter. It does really well the detection and extraction of text lines. Although, it is not flexible at all and it is not possible to modify it if pretended, and it confuses sometimes when the text is lighter or with big capital letters; it should be the software chosen for the task of extracting text lines. Again, it is important to mention that to get the software the authors from [12] should be contacted.

On the other hand, DhSegment has a great generality and it detects really fine the regions of interest. It did a really good job in page extraction (discounting the non-detection of the middle line of a book). It also behaved splendidly when locating text lines. The problem came when post-processing the image as the text regions detected were located below the text line. Solving that issue would permit DhSegment to be a solid solution for the task of segmentation. It is also an advantage that DhSegment is an open-source solution, so anyone could use or even edit it.

As for the next step, it would be interesting to process the lines extracted in the images to a transcription system. This would be a nice step in historical document understanding. The solution would, presumably, apply deep learning too. If someone aim to do this interesting task, this project is a great option to help with a solution.

REFERENCES

[1] I. G. and Y. B. and A. Courville, *Deep Learning*. MIT Press, 2016.

[2] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[3] "8.13. Encoder-Decoder Architecture — Dive into Deep Learning 0.7 documentation." [Online]. Available: https://www.d2l.ai/chapter_recurrent-neural-networks/encoder-decoder.html. [Accessed: 04-Jul-2019].

[4] P. Kaddas and B. Gatos, "A deep convolutional encoder-decoder network for page segmentation of historical handwritten documents into text zones," *Proc. Int. Conf. Front. Handwrit. Recognition, ICFHR*, vol. 2018-Augus, pp. 259–264, 2018.

[5] "Review: DeepLabv1 &amp; DeepLabv2—Atrous Convolution (Semantic Segmentation)." [Online]. Available: https://towardsdatascience.com/review-deeplabv1-deeplabv2-atrous-convolution-semantic-segmentation-b51c5fbde92d. [Accessed: 05-Jul-2019].

[6] "Evaluating image segmentation models." [Online]. Available: https://www.jeremyjordan.me/evaluating-image-segmentation-models/. [Accessed: 04-Jul-2019].

[7] Q. N. Vo, S. H. Kim, H. J. Yang, and G. S. Lee, "Text line segmentation using a fully convolutional network in handwritten document images," *IET Image Process.*, vol. 12, no. 3, pp. 438–446, 2017.

[8] B. R. A. Carter N.P., "Automatic Recognition of Printed Music. In: Baird H.S., Bunke H., Yamamoto K. (eds) Structured Document Image Analysis. Springer, Berlin, Heidelberg."

[9] M. Diem, F. Kleber, S. Fiel, T. Gruning, and B. Gatos, "CBAD: ICDAR2017 Competition on Baseline Detection," in *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR*, 2018.

[10] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?," in *Proceedings of the IEEE International Conference on Computer Vision*, 2009.

[11] C. Tensmeyer, B. Davis, C. Wigington, I. Lee, and B. Barrett, "PageNet," 2018.

[12] P. Schone, C. Hargraves, J. Morrey, R. Day, and M. Jacox, "Neural text line segmentation of

multilingual print and handwriting with recognition-based evaluation," *Proc. Int. Conf. Front. Handwrit. Recognition, ICFHR*, vol. 2018-Augus, pp. 265–272, 2018.

[13] L.-C. Chen, G. Papandreou, K. Murphy, and A. L. Yuille, "SEMANTIC IMAGE SEGMENTATION WITH DEEP CON-VOLUTIONAL NETS AND FULLY CONNECTED CRFS."

[14] S. Ares Oliveira, B. Seguin, and F. Kaplan, "DhSegment: A generic deep-learning approach for document segmentation," *Proc. Int. Conf. Front. Handwrit. Recognition, ICFHR*, vol. 2018-Augus, pp. 7–12, 2018.

[15] Jia Deng, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[16] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic gradient descent," *ICLR Int. Conf. Learn. Represent.*, 2015.

[17] F. Simistira, M. Seuret, N. Eichenberger, A. Garz, M. Liwicki, and R. Ingold, "DIVA-HisDB: A precisely annotated large dataset of challenging medieval manuscripts," in *Proceedings of International Conference on Frontiers in Handwriting Recognition, ICFHR*, 2017.

[18] "dhSegment : Generic framework for historical document processing — dhSegment documentation." [Online]. Available: https://dhsegment.readthedocs.io/en/latest/index.html. [Accessed: 19-Jun-2019].

[19] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," Jun. 2014.

[20] "Welcome to Sacred's documentation! — Sacred 0.7.5 documentation." [Online]. Available: https://sacred.readthedocs.io/en/latest/. [Accessed: 03-Jul-2019].

[21] D. Markus, K. Florian, and G. Basilis, "ICDAR 2019 Competition on Baseline Detection (cBAD)," Feb. 2019.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2016-Decem, pp. 770–778, 2016.