Historical Document Automatic Segmentation Using Dynamic Programming and Neural Networks

Lorenzo Agnolucci

lorenzo.agnolucci@stud.unifi.it

Alberto Baldrati

alberto.baldrati@stud.unifi.it

Giovanni Berti

giovanni.berti2@stud.unifi.it

Abstract

In this work we describe a pipeline for training a neural network in segmenting and recognising frequent words in early printed books, in particular we focus on Gutenberg's Bible. First we describe the creation of a dataset, containing only the Genesis book, using dynamic programming techniques and projection profiles with the aid of a line-by-line transcription. Then we leverage this dataset to train a Mask R-CNN model in order to generalize word segmentation and detection in pages where transcription is not available.

1. Introduction

In this work we address the recognition of early printed books by using deep models initially proposed for object detection in scene images. In particular, in this paper we process digital images of the first pages of the Gutenberg's Bible (the Genesis book) and provide a pipeline to train a neural network that can segment words and detect frequent stop words.

When dealing with word segmentation in historical documents such as the Gutenberg's Bible there can be several issues that can impair efficient learning with neural networks, such as poor dataset availability, different text features from the ones found in modern documents (e.g. very little spacing between words), and low quality image scans instead of digital copies.

Also, in the case of Gutenberg's Bible there isn't a proper word-for-word transcription that can be used as training material for a **deep learning model**. To manage this issue, we used a previous text called *Biblia Vulgata*. This text can present some slight differences from the actual printed text (e.g. some missing/reordered words or different spellings). Moreover the text doesn't have any kind of line correspondence. By editing this transcription to match the book's line-by-line structure we managed to get an improved model for word segmentation and stop word detection, build-

ing on previous work from [7].

While the necessity of a line-by-line transcription can be a limiting factor in the applicability of this work, using it in combination with a **dynamic programming algorithm** during the segmentation step allows us to overcome the difficulties described above.

1.1. System Overview

The proposed system is made out of the following parts:

1. **Segmentation preprocessing** (e.g. deskewing, line detection, punctuation detection)

2. **Word segmentation** using dynamic programming based on a line-by-line transcription

3. **Neural network** model trained to segment words and detect stop words and hyphens

2. Proposed approach

2.1. Dataset

We use original images from [6] and an auxiliary source text. The text is the *Biblia Vulgata* and each line has been manually edited to match the original work. Also, **hyphens** were added to the transcription whenever they were present in original image scans.

Because the transcription part of the dataset was manually produced, only the *Genesis* book was available and used in this work and during neural network training. Despite the transcription being a tiny part of the actual source text, we found that due to the regular structure of the printed text and page embellishments it is enough to allow sound inference from the neural network.

2.2. Segmentation Preprocessing

2.2.1 Skew correction

As in [7] at first we perform skew correction. Skew detection and correction is executed in two steps: after

removing the edges of the page we first compute the inclination of the rotated rectangle of minimum area enclosing the foreground of the image. In the second step we refine this angle by computing the horizontal projection profile for different angles and then choosing the angle with the higher peaks.

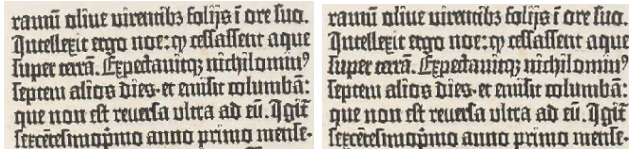An example of skew correction is shown in **figure 1**.

Figure 1. Example of original (right) and deskewed (left) images

2.2.2 Line detection

Line detection preprocessing begins with a preliminary binarization pass. After that we compute a vertical projection to separate the columns present in the image. We then use domain knowledge to choose the columns where text is present by choosing the two largest detected columns.

The line segmentation pass is then executed separately for each column. An *horizontal projection* is computed and each time there is a significant change from high intensity to low intensity colour a line separation is marked. This way whenever there are many black-to-white pixel changes in two subsequent lines the algorithm infers from the projection profile that a line cut can be made. We also apply a constant backoff of 10 pixels to avoid line oversegmentation.

Because we detect black-to-white changes, we manually add the first top line marker using a fixed width offset from the topmost detected line marker.

Our approach differs from the one described in [7], where horizontal projection is performed first to segment lines, followed by a vertical projection to split columns. This should be an improvement because projection of a given row is not affected by the corresponding row in the other column when they have a slightly different vertical alignment.

An example of lines and columns detection is shown in **figure 2**.

2.2.3 Punctuation detection

Direct matching of transcription text to word images in Medieval text poses a challenge because of extensive use of outdated punctuation and abbreviations. Because of that, a **punctuation detection** stage is applied to the source images. This detection stage covers
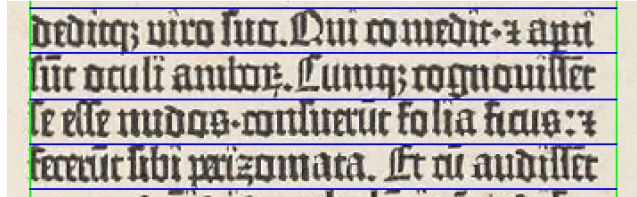

Figure 2. Example of column (green) and line (blue) detection

periods, colons, mid dots, long accents and dots over i's. For all punctuation we first compute connected components over line images and then filter them using metrics such as:

- bounding box size and area (*e.g.* area between 7 and 35 for colons)

- position (*e.g.* bounding box bottom between 18 and 23 pixels from the top of the line and top less than 14 pixels from the top of the line for colons)

- density of black pixels inside the bounding box (*e.g.* density lower than 0.71 for dots over i's)

- number of black pixels above/below the connected component: (*e.g.* number of black pixel greater than 15 for long accents)

We take care of not doubly detecting colons as periods by checking whether periods have a corresponding similar black connected component above. An example of punctuation detection is shown in **figure 3**.
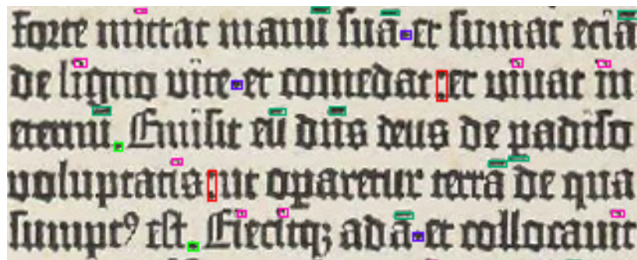

Figure 3. Example of punctuation detection: colons (red), mid dots (purple), periods (light green), long accents (dark green) and dots over i's (pink)

2.3. Word segmentation

Our approach to word segmentation is made by two steps: first a preprocessing one that outputs candidate segmentations, then a dynamic programming-based step that filters those candidates. Unlike [7] we exploit the regular structure of the text as well as the

Figure 4. Line image with original and inverted binarized projection profile

available transcription to accomplish segmentation instead of relying purely on image and projection features. Moreover, our approach ensures that the number of segmented words in a line matches the number of the words in the corresponding transcribed line.

2.3.1 Pixel-wise processing

Prior to line-wise word detection we first delete blue-ish and red-ish connected components in order to get a more regular image. The aim is to target page embellishments and caputs, which often are coloured in red and blue tones. After deleting caputs all the remaining processing chain is done on a line-per-line level.

The first step done over the line image is punctuation deletion. Because only colons, periods and mid dots can affect word segmentation, only the respective subroutines devoted to their detections are executed. After that, we denoise the resulting image by deleting small connected components which area is below a given threshold. Then a *line projection histogram* is computed, binarized (using a fixed threshold) and inverted, and it is then used as input to the second processing stage. Thus each potential *word cut* (whitespace in the original image) is associated to a 1 and each black pixel column (part of a character in the original image) is associated to a 0. We will refer to this histogram as *observed histogram*.

An example of this pipeline is shown in **figure 4**.

2.3.2 DTW preprocessing

To help with word segmentation, the transcription is used as a ground truth that is then matched to the line image. We generate an *expected binarized histogram* from a line transcription by associating to each character an expected black-pixel run. Likewise, whitespace character are associated to white-pixel runs of fixed length.

Both binarized histograms (expected and observed) are then *collapsed* by replacing each run of white pixels with their length and a run of $(length - 1)$ zeros.

In the following diagram we can see an example of this process.

1	1	1	0	$\cdots$

$$\vee$$

0	3	0	0	$\cdots$

We then delete short white runs from the observed histogram in order to reduce false positives when cutting words. If N is the number of words in a certain line, the target white run is computed by taking the smallest of the biggest N white runs. White runs that are strictly smaller than this target white run are replaced with black pixels.

These histograms are then interpreted as *time series*, where x-axis is time/pixel x coordinate and y-axis is a white run length (a "spike") or zero.

In **figure 5** an example of collapsed and filtered histogram is shown.
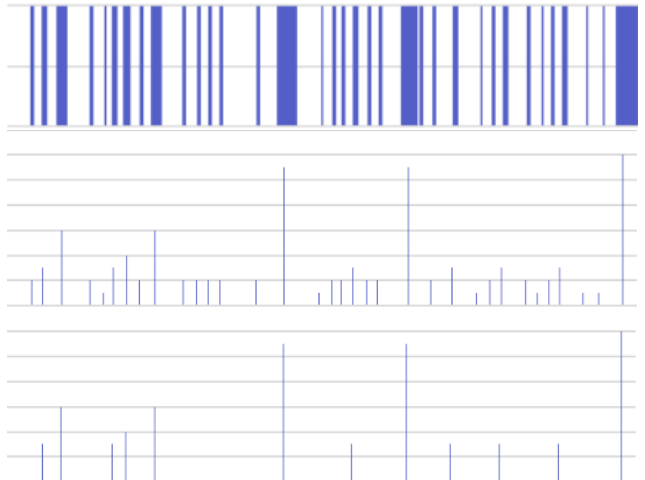

Figure 5. Binarized (from figure 4), collapsed and filtered histograms

The observed histogram is then manipulated to maximize the probability of cutting where periods, colons or mid dots are detected. To do so, we double the spike value in order to increase the probability of matching in the *Dynamic Time Warping* step.

After dealing with periods, we account for long accents in the observed sequence. In Latin texts such as the Gutenberg Bible, long accents' purpose is most often to signal abbreviations (e.g. *veritatē* for *veritatem*, *ī* for *in*). As such, it is very important to correct the observed sequence in light of accent detection. Also,
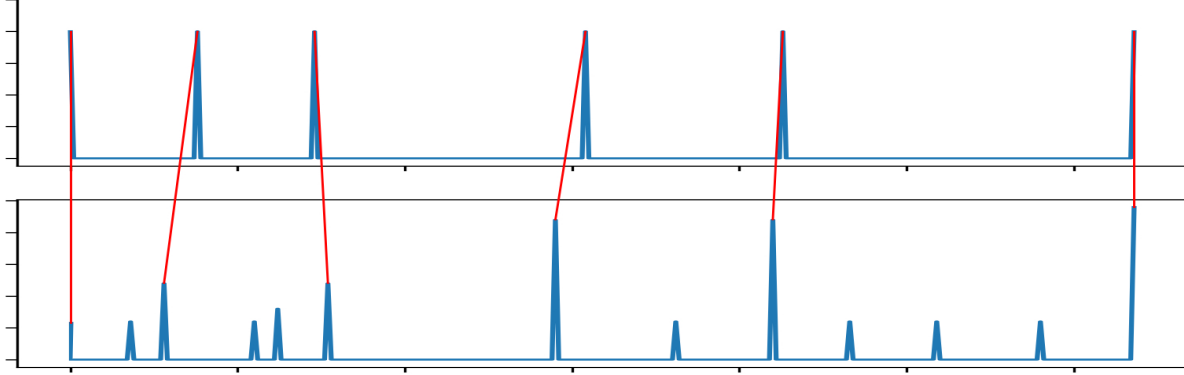
Figure 6. Example of mapping between expected (above) and observed (below; corresponds to the filtered histogram from figure 5 on the preceding page) sequences with DTW

long accents can appear not only at the end of a word, but can be written in the middle or at the start of it. These edge cases are handled by not modifying spike values whose original value is too low (thus by trusting only white runs big enough that can be considered actual word separators) while extending the other runs where long accents are present.

In addition, *alternate forms* of recurring words (*ligatures* in typographical parlance) such as *et* and *quod* are often used throughout the source text. We account for those by producing all possible combinations of expected sequences by substituting letter-by-letter run encoding with their ligature encoding (e.g. 'e' → 7 pixels, 't' → 7 pixels, 'et' can be either 14 pixels in one combination and 10 pixels in the one with the ligature version).

2.3.3 Dynamic Time Warping

Thus, each line of text potentially outputs many possible expected sequences (depending on the presence of potential ligatures), and we would like to both get the most probable expected sequence and to match whitespaces in this sequence to whitespaces in the observed sequence. Note that both the expected sequence and the observed sequence are constructed such that potential whitespaces are encoded with non-zero values ("spikes" which correspond to white-pixel run length in the observed sequence) and word characters are encoded with zeros (black pixels runs in the observed sequence). We interpret these sequences as time series and match them using a *Dynamic Time Warping* algorithm.

Dynamic Time Warping is a dynamic programming algorithm used to compute similarity between two discrete time series. A *DTW* algorithm also produces a *warping path*, that is, an optimal mapping that minimizes a distance-like measure computed by the algo-

rithm. It operates under some constraints, such as:

- Every index from the first sequence must be matched with one or more indices from the other sequence, and vice versa

- The first (and last) index from the first sequence must be matched with the first (and last) index from the other sequence (but it does not have to be its only match)

- The mapping of the indices from the first sequence to indices from the other sequence must be monotonically increasing, and vice versa, i.e. if $j > i$ are indices from the first sequence, then there must not be two indices $l > k$ in the other sequence, such that index i is matched with index l and index j is matched with index k, and vice versa

We run *DTW* over all possible expected sequences against the observed sequence and consider the one with the *lowest distance*. After that, we mark as word separators the spikes in the observed sequence that get mapped to spikes in the expected sequence.

Unlike standard *DTW*, we have two additional constraints:

- No expected spike should be mapped to a **zero** in the observed sequence

- If N is the number of words in the line text, exactly N spikes should be matched in the observed sequence to the expected sequence (which by construction has N spikes). That is, we would like to have an *injective matching* from expected spikes to observed spikes.

For both of these constraints it proved sufficient to proportionally stretch the expected sequences to have the same length of the observed sequence. In **figure 6**

we can see an example of mapping between expected and observed sequence using DTW.

Thanks to this segmentation pass we can associate each segmented word in the image with the corresponding word in the transcription in the order they appear: the first with the first, the second with the second and so on. This is possible due to the *injective matching* constraint, which ensure that the number of words in the transcription and in the image is the same. Moreover, whenever the transcription of a line ends with a = (a hyphen) we add an *extra word* (marked as a hyphen) between the column limit and the last word cut computed with the DTW.

3. Dataset building

Before building the dataset, because of memory limitations and performance issues we split each column of each page in *six chunks* of *seven rows* each. Using the segmentation and the association between the GT transcription and the image portions we build a dataset in COCO object detection format.
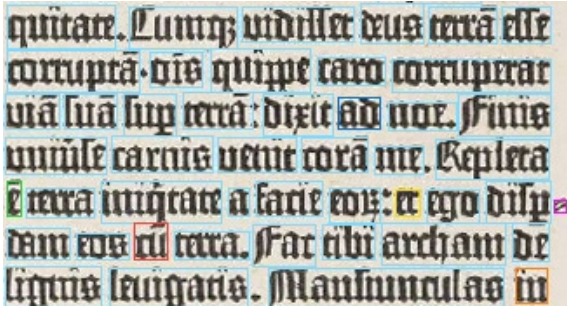
Figure 7. Visualization of the dataset annotations

The dataset contains **seven** categories (background excluded):

1. et
2. est
3. ad
4. cum
5. in
6. other
7. =

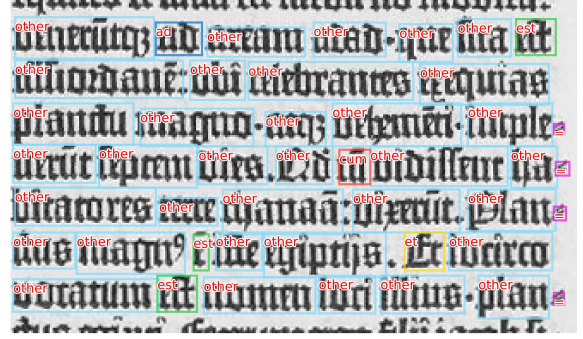The categories from 1 to 5 (included) represent the

Figure 8. Visualization of the output of WALL·E Net

five most frequent words found in the Gutenberg Bible, which we will refer to as **stop words**.

The *other* category corresponds to **any other word** which is not a stop word. Along with the stop words categories this category is useful for segmentation purposes.

Lastly, the = category corresponds to **hyphens** as found in the original images. Each annotation includes several fields such as: bounding box, category id, chunk image.

In **figure 7** an example of the annotations of a chunk image is shown.

4. WALL·E Net

Using the dataset built in the previous section we can set up a neural network, which we will refer to as **WALL·E Net** (**W**ord **A**nd **L**andmark **L**ocation **E**xtended Network), capable of segmenting words and detecting hyphens and stop words.

The network is based on *Mask R-CNN* [3] and implemented with Keras [2] and TensorFlow [1][4]. We used the Convolutional Neural Network *ResNet101* as the backbone with pre-trained *ImageNet* weights.

The network takes a three channel chunk image in input and outputs the bounding box of each detected instance with the corresponding category name. Even though *Mask R-CNN* is able to provide the segmentation mask for each instance we rely only on the inferred bounding box because it is sufficient to fulfill the segmentation task.

5. Experimental results

To evaluate **detection and segmentation results**, we split the dataset generated above into a training part, a test part and a validation part. The division of the generated annotations into the three parts is done randomly. The training dataset consists of **38** pages out of **48** annotated pages, while testing and validation datasets are made both of **5** pages. All the

	ad	*cum*	*est*	*et*	*in*	total
occurences	48	17	33	215	103	416
WALL·E	48 (**100%**)	16 (**94.1%**)	33 (**100%**)	211 (**98.1%**)	90 (**87.3%**)	398 (**95.7%**)
LW-Net [7]	**96.3%**	**80.0%**	**91.7%**	**88.3%**	**80.3%**	**87.0%**

Table 2. Results of landmark detection

following results refer to metrics computed on the testing dataset.

5.1. Segmentation Results

We evaluate the results of segmentation task using **COCO object detection metric** that is designed to evaluate visual object detection and recognition models [5]. A prediction is considered as correct (true positive) if the Intersection over Union of the predicted bounding box and of the GT is greater than a given threshold. The **Average Precision** is computed considering the objects of all the categories and averaging the values. In order to have a more complete picture of the network capabilities, we compute AP with IoU thresholds from 0.5 to 0.95 with a step size of 0.05. Moreover, we compute AP_m and AP_s, which are APs computed for small (area $< 32^2$ pixels) and medium ($32^2 <$ area $< 96^2$ pixels) objects.

The comparisons between the results obtained by our network and the reference network WH-Net [7] are displayed in **table 1**. Concerning hyphens, their recognition accuracy with our network is **90.3%**, while the reference model has an accuracy of **76,4%**.

Metric	AP	$AP_{0.5}$	$AP_{0.75}$	AP_s	AP_m
WALL·E	0.727	0.931	0.854	0.718	0.733
WH-Net [7]	0.682	0.843	0.817	0.651	0.795

Table 1. Results of segmentation task

From **table 1** we can see that thanks to our *DTW*-based algorithm, which resulted in an empirically-measured high-quality segmentation, WALL·E Net manages to improve on previous work [7] in most metrics.

Although we have compared WALL·E result with the WH-Net of [7], there are some key differences from which methodologies and models differ.

- The dataset on which we train WALL·E is built such that each word is annotated with its **bounding rectangle** (see **figure 7 on the previous page**) instead of a generic rectangle in which each word is contained, the task of word segmentation is actually more difficult.

- The dataset we used for training and testing is **larger**

- WALL·E Net detects and classifies **seven** categories while WH-Net only **two** (words and hyphens). Because of this in the comparison WALL·E Net can be penalized.

5.2. Landmark Location Results

To evaluate our segmentation, and our **association** of the segmented text to the transcription from a different perspective, we measure the ability of the WALL·E Net to spot Landmark words. In **table 2** are displayed the comparisons between the results obtained by WALL·E network and the reference network LW-Net [7].

6. Conclusion and future work

We proposed the combined use of dynamic programming and neural networks to fulfill a segmentation task on early printed book, in our case Gutenberg's Bible. The proposed techniques have been evaluated on the *Genesis* book, that share important features with other similar works, like the extensive use of different abbreviations for some words and the regular word spacing.

We achieve promising results like a **72.7% AP** in segmentation task and a **95.7% recall** when detecting stop words. Possible future developments may include automatic alignment of the source transcription using the neural network model developed in this work.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] F. Chollet. keras. `https://github.com/fchollet/keras`, 2015.

[3] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[4] M. Inc. Mask R-CNN, 2018. `https://github.com/matterport/Mask_RCNN/tree/3deaec5d902d16e1daf56b62d5971d428dc920bc`.

[5] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing.

[6] H. Project. Munich gutenberg bible. `https://www.themorgan.org/collections/works/gutenberg/humi`.

[7] Z. Ziran, X. Pic, S. U. Innocenti, D. Mugnai, and S. Marinai. Text alignment in early printed books combining deep learning and dynamic programming. *Pattern Recognition Letters*, 2020.