

# Text line recognition using a BLSTM neural network

Federico Palai  
Università degli Studi di Firenze  
federico.palai1@stud.unifi.it

Federico Tammaro  
Università degli Studi di Firenze  
federico.tammaro@stud.unifi.it

## Abstract

*In this work we propose a neural network model to transcribe pages from the original Gutenberg's Bible using CNNs and BLSTMs. After expanding the dataset to add the Exodus book by manually aligning the text transcription to the pages of the book, we used this data to refine our model. Then we conducted several experiments to test other models found in literature in order to compare the results.*

## 1. Introduction

The main goal of this work is the training of a neural network in order to transcribe ancient books written with the movable-type printing press technique. In particular we used digital scanned images from the first two books of the Old Testament of the Gutenberg's Bible.

To achieve this, we used a deep learning model based on a combination of convolutional and long-short term memory layers. So far word segmentation is the most common approach to transcribe these kind of documents, instead we chose a line segmentation approach to see if it is a viable option for the manuscripts transcription.

### 1.1. System Overview

Following we present the main steps of our work:

1. **Preprocessing:** detection and segmentation of lines for each page; labels association and creation of the dictionary used for CTC decoding
2. **Neural network:** details of the network used for the experiments
3. **CTC loss and dictionary correction:** details of how the Connectionist Temporal Classification works and dictionary substitution used for prediction

## 2. Dataset

As already said, our dataset is based on high-definition scans of the *Biblia Vulgata* manuscript printed by Guten-

berg in 1450s, digitalized in 2005, and its text transcription.

There were some discrepancies that likely lead to errors in the learning phase of the neural network and consequently in the predictions:

- Abbreviations, which are quite common in the original text because the printing process was still expensive, as shown in Figure 1.
- The unavailability of a trustworthy transcription, hence we were forced to use a common transcription of the latin bible and therefore there is not a perfect match between the two versions in some parts.
- The removal of the punctuation except for the hyphen (the symbol "="), signaling a truncation of the word caused by a line break.

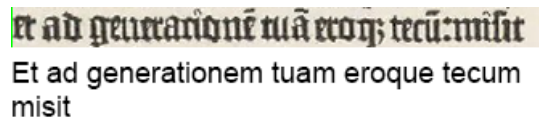


Figure 1: Example of text abbreviations in the dataset

In particular, our dataset is composed by the first two books, *Genesis* and *Exodus* with a total of 7412 text lines. While we already had the *Genesis* text split line by line to match the original pages, we had to apply the same procedure also to the *Exodus*.

## 3. Implementation

### 3.1. Preprocessing

The first step, given a page image, is to identify the two columns and the text lines within them. To do so a previous work has been used [cite here] which produces a new deskewed image and then highlights columns and text lines. From this new image, we extract each line as a separate image which then will be used as input of the neural network. A simplified scheme of this process is shown in Figure 2.

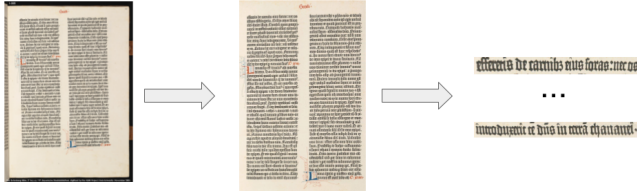


Figure 2: Image deskew and lines extraction exemplification

Each image coming from this procedure is then associated with the corresponding line used as its label after a tokenization process: each letter from the alphabet (also including spaces and line-breaks) is bound with a number and each of its occurrences in the label is replaced with this number. The correspondence between letters and numbers is stored in a dictionary later used for decoding.

Each image might be different both in width and height therefore a padding of the image is necessary to standardize the shape in order to feed it to the network: this way each image will have the same shape of the biggest one. Therefore each input will be (340, 37, 3), meaning that the image has an width of 340 pixels, an height of 37 pixels and 3 channels (RGB). However, lines are greyscaled during the preprocessing phase, still preserving the 3 channels for each image: this choice has been made seeing the results coming from the network for both colored and greyscaled images. Also the labels must be padded to match the longest one.

### 3.2. Neural Network

Since text recognition using neural networks is quite common in literature, and many state of the art models have already been proposed, we looked for the closest to our case study [1], adapting it to our necessities. From various papers, emerges that a good combination of layers for the text recognition is the following one:

- 6 Convolutional Neural Network layers.
- 2 Bidirectional LSTM layers.
- 1 CTC layer.

In Figure 3 is shown a simplified view of the model.

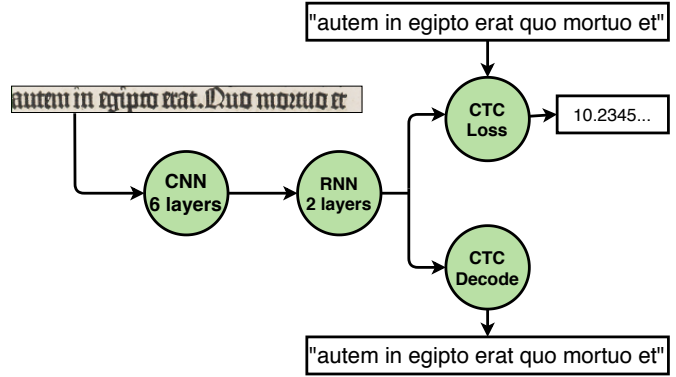


Figure 3: Simplified model view

On the other hand, in Figure 4 is presented a more detailed view of the implemented model, generated by Keras.

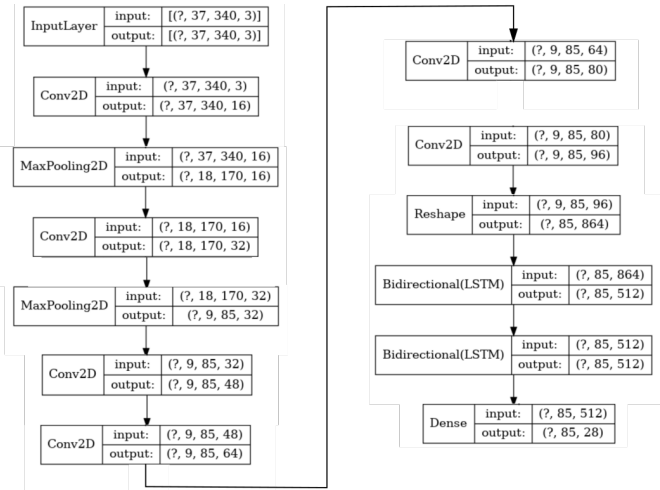


Figure 4: Detailed model view

Here we can notice:

- Each CNN layer is combined with batch normalization and leaky ReLU. The first two layers also have a Max Pooling with a 2x2 window and 2x2 stride. Each CNN layer has a kernel size of 3x3 and an increasing number of filter (16, 32, 48, 64, 80, 96).
- A Reshape layer used to adapt the output of the last CNN layer to fit the input of the first BLSTM layer
- Each LSTM layer has 256 units but since we use Bidirectional LSTMs each layer has 512 units.
- A Dense layer where the number of units is equal to the dictionary size plus one representing the blank label.

Since Keras does not offer any already built-in version of the CTC layer both for the loss and the decoding, in our model we used the implementation proposed by Arthur Flôr [1].

### 3.3. CTC - Connectionist Temporal Classification

As suggested in the literature, in text recognition a good practice is to use the CTC loss. The CTC approach relies on dynamic programming and it is based on the probabilities of all the possible paths which can produce the output sequence. We used the CTC in two different phases of the model: first to evaluate a loss function during training to improve the learning process, then during the prediction phase to decode the outputs of the model which then will be converted back to strings.

#### 3.3.1 CTC - Loss

The CTC loss function, to be calculated, requires:

- The ground truth label and its length
- The label predicted by the last recurrent layer and its length

The CTC loss uses the length of the dictionary of characters plus one and this extra character is a blank which represents a lack of understanding of a character by the network. The last layer of the network contains one unit per character that outputs the probability of having the corresponding character at a particular time step: every possible path leading to the ground truth is discovered and their probabilities are summed up and back-propagated through the RNN and its parameters are updated accordingly.

This loss is used to evaluate both training and validation data and their values are the main metrics considered in the models trained.

#### 3.3.2 CTC - Decode

In the CTC decoding phase, the most probable path is found and each character along this path is the most probable one for that time-step.

Then blank and duplicated characters are removed from the path and the result is a sequence of numbers representing all the characters from the dictionary built during the preprocessing phase in the prediction. Therefore it is now possible to convert this sequence back into a string, obtaining the sentence predicted by the network which represents the transcription of the input image.

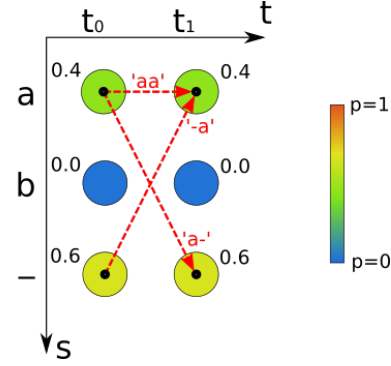


Figure 5: CTC Decode example [2] containing 2 time-steps ( $t_0$  and  $t_1$ ) and 3 labels (a, b and - the CTC-blank). For the labeling "a" these algorithms sum over the paths "-a", "a-" and "aa" with probability  $0.6*0.4+0.4*0.6+0.4*0.4=0.64$ .

### 3.4. Dictionary substitution for predictions

We implemented the use of a dictionary to avoid non-existing words in the prediction: this dictionary is composed of all the words in the Genesis and Exodus books.

In particular, during the decoding phase, we compare all the words of the prediction with those present in the dictionary, taking the one with minor edit distance. The final results are presented in the following figures, where the blue text represents the network prediction without any substitutions, the red text is the network prediction with dictionary substitutions, finally the green text is the ground truth.


  
 deus iacob postquam reversus est de mesopotha=  
 deus iacob postquam reversus est de mesopotha=  
 deus iacob postquam reversus est de mesopotha=

Figure 6: A perfect prediction without dictionary substitutions, in blue the raw prediction, in red the prediction with dictionary substitutions and in green the ground truth

  
 illorum non sit gloria mea quia in furore  
 illorum non sit gloria mea quia in furore  
 illorum non sit gloria mea quia in furore

Figure 7: An example of substitution of non-existent word, in blue the raw prediction, in red the prediction with dictionary substitutions and in green the ground truth

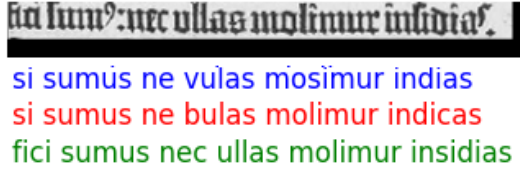


Figure 8: Limits of dictionary substitutions on wrong network prediction, in blue the raw prediction, in red the prediction with dictionary substitutions and in green the ground truth

In Figure 6 we can notice that when the prediction is already consistent with the ground truth, no substitutions are applied and the results are not compromised with a wrong substitution; on the contrary if a non-existent word is present in the original prediction, the substitution might help improve the prediction. This is the case of Figure 7 while Figure 8 shows that dictionary substitution is not always correct because of an incorrect prediction in the first place.

## 4. Experiments

Since all the previous works are based on handwritten text recognition therefore with a really different dataset both considering the irregularity of handwritten characters and the absence of abbreviations, we had no baseline results to compare to. So we experimented different network architectures to find which one works best with the given dataset:

- 6 CNNs and 2 RNNs layers with grayscaled dataset, our proposed network (as described in 3.2)
- 6 CNNs and 2 RNNs layers with colorized dataset
- 5 CNNs and 2 RNNs layers with grayscaled dataset, based on [1]
- 6 CNNs and 5 RNNs layers with grayscaled dataset, based on [3]

### 4.1. Error Rates

To evaluate the predictions generated by the network, we relied on the following error rates with different levels of granularity:

- **Character Error Rate (CER)**: the percentage of wrong characters among all the predictions. Using the Levenshtein edit distance,  $S_c$ ,  $D_c$ ,  $I_c$  and  $C_c$  are, respectively, the number of substituted, deleted, inserted and correct characters among all predicted sentences.

$$CER = \frac{S_c + D_c + I_c}{S_c + D_c + C_c} \quad (1)$$

- **Word Error Rate (WER)**: the percentage of wrong words among all the predictions. Using the Levenshtein edit distance,  $S_w$ ,  $D_w$ ,  $I_w$  and  $C_w$  are, respectively, the number of substituted, deleted, inserted and correct words among all predicted sentences.

$$WER = \frac{S_w + D_w + I_w}{S_w + D_w + C_w} \quad (2)$$

- **Sentence Error Rate (SER)**: the percentage of wrong sentences among all the predictions. Using the Levenshtein edit distance,  $S_s$ ,  $D_s$ ,  $I_s$  and  $C_s$  are, respectively, the number of substituted, deleted, inserted and correct sentences among all predicted sentences.

$$SER = \frac{I_s + S_s + D_s}{S_s + D_s + C_s} \quad (3)$$

## 4.2. Results

All the experiments have been conducted on the Colab platform provided by Google and on the Iris Laboratory Server, using Tensorflow GPU 2.1.0 to speed up the computation.

The results obtained have the dataset split as following:

- A train set consisting in 95% of total samples, 7041 text lines of which 6336 effectively used for training.
- A validation set consisting in 10% of train set, 705 text lines; to achieve this, we used K-Fold Cross Validation with  $K = 10$  to compute the average results to have a better overview of the model behaviour
- A test set consisting in 5% of total samples, 371 text lines.

Each experiment uses 300 epochs (except for the one with 5 RNNs) since we noticed that after this threshold all the models tend to overfit on training data.

Following we present the results for every model variation we already discussed. In Figure 9 we have the learning curves for all the experiments representing the training and validation losses.

### 4.2.1 6 CNNs and 2 RNNs with grayscaled dataset

This model is the one which yields the best results as shown in Table 1. The *Raw* row represents the prediction error rates, while the *Dict* row considers predictions corrected with dictionary substitutions.

The average time of execution per epoch for this experiment was 30.2 seconds for a total of about 25 hours for the whole experiment.

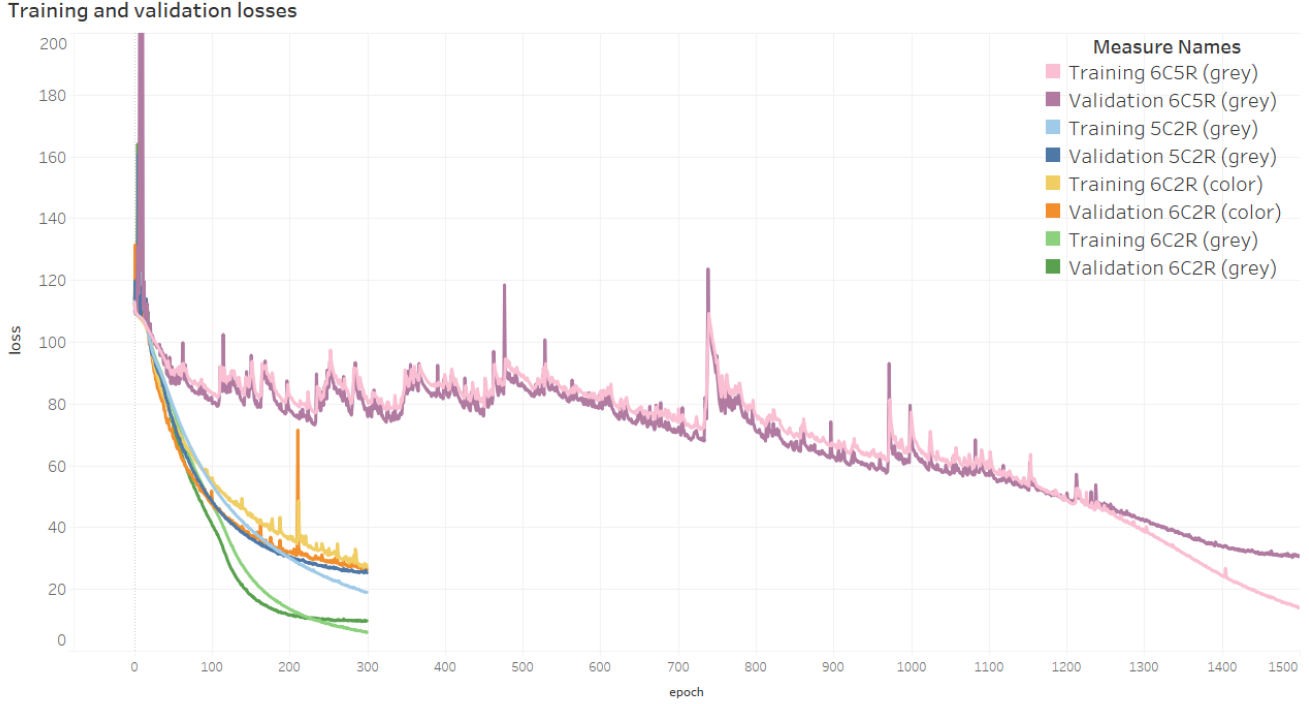


Figure 9: Learning curves for all the experiments. Each experiment has a lighter-colored curve for the training and a darker one for the validation

#### 4.2.2 6 CNNs and 2 RNNs with colored dataset

The same model as before applied with colored dataset yields less accurate results as shown in Table 1.

The average time of execution per epoch for this experiment was 30.8 seconds for a total of about 25 hours for the whole experiment.

#### 4.2.3 5 CNNs and 2 RNNs with grayscale dataset

One of the first experiments was with 5 CNNs but as shown in Table 1 this has been proven to be the worst model we tested.

The average time of execution per epoch for this experiment was 27.6 seconds for a total of about 23 hours for the whole experiment.

#### 4.2.4 6 CNNs and 5 RNNs with grayscale dataset

Training a model with 6 CNNs and 5 RNNs on 300 epochs, we saw that there was room for the learning curves to improve, hence we tried on 1500 epochs. The results are shown in Table 1.

The average time of execution per epoch for this experiment was 48.9 seconds for a total of about 203 hours (~ 8 days) for the whole experiment.

## 5. Conclusions and future developments

The proposed model seems to be very effective for the transcription of manuscript written with the movable-type printing press technique, with 7.1% of Character Error Rate, 18.5% of Word Error Rate and 64.5% of Sentence Error Rate. As observed, the results have greatly improved when

|     | 6 CNNs 2 RNNs grey |              | 6 CNNs 2 RNNs color |       | 5 CNNs 2 RNNs grey |       | 6 CNNs 5 RNNs grey |       |
|-----|--------------------|--------------|---------------------|-------|--------------------|-------|--------------------|-------|
|     | Raw                | Dict         | Raw                 | Dict  | Raw                | Dict  | Raw                | Dict  |
| CER | 7.8%               | <b>7.1%</b>  | 9.6%                | 9.1%  | 12.2%              | 11.7% | 22.6%              | 23.2% |
| WER | 31.3%              | <b>18.5%</b> | 35.3%               | 22.2% | 42.3%              | 27.9% | 55.3%              | 45.7% |
| SER | 87.0%              | <b>64.5%</b> | 89.2%               | 69.5% | 95.2%              | 80.7% | 97.0%              | 90.9% |

Table 1: Results table showing CER, WER and SER for each experiment both with and without dictionary substitution

we expanded the dataset adding the *Exodus* book along the *Genesis*. On the other hand, changing the model structure by adding more layers and/or using a colorized version of the dataset, does not improve the results.

Finally, some observations that might be useful for future developments to improve the results:

- Adding more pages (in this case more books from the *Bibbia Vulgata*)
- Refining the transcriptions of all the pages in the dataset, ensuring that the labels will perfectly match the images lines

## 5.1. Resources

All the code used in this project is available on GitHub [4]

## References

- [1] Arthur Flôr. An implementation of htr using tensorflow 2.0, 2019. <https://github.com/arthurflor23/handwritten-text-recognition>.
- [2] Harald Scheidl. Comparison of connectionist temporal classification decoding algorithms. <https://github.com/arthurflor23/handwritten-text-recognition>.
- [3] Joan Puigcerver. Are multidimensional recurrent layers really necessary for handwritten text recognition? 2017.
- [4] Federico Palai and Federico Tammaro. Gutenberg bible lstm. [https://github.com/palai103/Gutenberg\\_Bible\\_LSTM](https://github.com/palai103/Gutenberg_Bible_LSTM), 2020.