university of groningen / faculty of science and engineering

Handwriting Recognition on the Dead Sea Scrolls and IAM Datasets

Handwriting Recognition Assignment
Group 20

S. Juneja, s4456580, s.juneja@student.rug.nl
L. Mueller, s3745503, d.l.mueller@student.rug.nl
C. M. Nampouri, s4721810, c.m.nampouri@student.rug.nl
W. de Vries, s2712121, w.de.vries@student.rug.nl

March 7, 2024

Abstract: Offline Handwritten Text Recognition (HRT) is an active field of research that deals with the transcription of handwritten text contained in images. For humans this is mostly trivial (depending on the handwriting), but there are a number of challenges with the automation of this task. We received two datasets, the Dead Sea Scrolls (DSS) and IAM lines. Both differ in type and quality of the data and therefore require separate approaches. The DSS dataset contains text in Hebrew which has decayed due to age and is not segmented. Only single letters are given as labeled data, so the text images have to be segmented into single characters which can be then recognized. In the case of the IAM dataset, all initial text has been segmented into lines. These contain text in English and come with transcriptions as labels, so an end-to-end system can be implemented. Methodologies for solving both tasks are presented in this report.

1 Introduction

Handwriting recognition has for a long time been an interesting field. From the days of Optical Character Recognition, which focussed on the recognition of characters that were very well-defined, handwriting recognition methods focus on text written by humans. The methods used in this field have gotten more and more complex over the years, and have had successes with recognizing different types of handwritten documents, ranging from ancient Arabic writing styles to modern Japanese handwriting. In this paper we will be looking at two types of handwriting recognition; the first is to recognize the handwritten text on the Dead Sea Scrolls collection, and the second is about recognizing text by different authors in the IAM dataset.

Our goal with the Dead Sea Scrolls is to create a pipeline that is able to transcribe the handwritten text of the Dead Sea Scrolls. This pipeline will be able to take as input an image with handwritten Hebrew text from the Dead Sea Scrolls, and output a text document containing the digitalized Hebrew characters occurring in such a text image.

Everything we did related to building this pipeline for the Dead Sea Scrolls task will be discussed extensively in Section 3. In this section, we will first explain our methodology. We will go over the dataset, as well as the methods we use for pre-processing, character segmentation, character recognition, prediction, and transcription. For a quick glance at the pipeline, we refer to Figure 3.1, at the start of the methodology. After going through the methodology, we will go into the experimental setup of the project in Section 3.2. In this section, we will go over the training set, the way we augment the data for training, and the specifics of the model we use. We will also mention the network performance on the training dataset here. Finally, in Section 3.3, we will show some of the results our network has on the test images from the Dead Sea Scrolls dataset.

In the second part of the project we are looking to recognize handwritten text from different authors with different handwriting styles. This task is very different from the research that is being done on the Dead Sea Scrolls, as this task depends more on recognizing sequences of words and characters. Morever, the major problem is varying characteristics of handwritten words such as round, slanted, distorted characters along with alignment between input characters and image pixels.

Our motive for this task is to build a pipeline to transcribe handwritten text images and convert it into textual form. The methodology proposed to undertake this particular task is described in 4.1. In this section, exploratory analysis of IAM lines dataset, spiltting of data, data preprocesing and augmentation techniques, model architecture and evaluation metrics are discussed. In the next section, experimental setup of the task 4.2 is discussed where hardware requirements and model parameters are mentioned followed by model results and insight to model prediction on the novel dataset in Section4.3.

2 Literature Review

General Review To get an overview of what the Dead Sea Scrolls are, as well as some of the difficulties for recognition, we looked at some background on the Dead Sea Scrolls. These include a glance at a book by Qimron (2018), which contains a lot of information about the scrolls, and provides some valuable information about the way Hebrew was written during that time period.

Character-level recognition One very helpful paper was proposed by Likforman-Sulem, Zahour, and Taconet (2007) and reviews all the essential literature in the field of historical handwritten texts. Possible problems with the recognition of these texts, and as possible solutions are addressed. It also discusses various techniques for line segmentation, as well as for word-level segmentation methods. Notably, it suggests the use of connected components for segmentation of Hebrew texts.

Another segmentation-focussed paper is Choudhary, Rishi, and Ahlawat (2013). This paper suggest a new method for character segmantation from words. In essence, it suggests an improvement on methods that use vertical projection for character segmentation, by combining several potential segmentation collumns into one. Although it still has its own problems, it does show a promising direction for character segmentation.

We also wanted to take advantage of n-grams, for which we took a look at a paper by Damashek (1995). It is about the sorting of data based on the n-grams contained within a certain text. Though the methods the paper proposes are not very relevant to our implementation of bigrams, the paper does provide context and understanding of how to use the n-grams data in general. In this paper, the similarity of language is compared between documents based on an exemplar text. The n-grams from this text are then compared to the n-grams of the document we wish to obtain information about.

Line-level recognition There exist various methods to perform handwritten text recognition but mainly due to uncertainity and non-triviality in the hand-written text, each method applied has its own pros and cons.

Marti and Bunke (2000) proposed a method that treated input as complete handwritten lines of text and as a byproduct of recognition based on hidden Markov models (HMMs), individual words were obtained through segmentation of line. The three main components of the method were preprocessing, feature extraction and recognition. Another novel feature that increased the performance of the method was the addition of Statistical Language Model which was built on Lancaster-Oslo/Bergen corpus (S. Johansson and Goodluck, 1978).

Another hybrid Hidden Markov Model and ANN method for handwritten text recognition was also introduced by Boquera et al S. Espana-Boquera and Zamora-Martinez (April 2011) before the advent of Connectionist Temporal Classification (CTC) (Graves, Fernández, Gomez, and Schmidhuber, 2006) which made use of ANN approach to produce proabilities for the HMM instead of using Gaussian mixture model. For each of the text lines, feature vectors were extracted by applying a sliding window and computing gray-value statistics for each possible position. Next, Viterbi algorithm - a dynamic programming algorithm which estimates a maximum posteriori probability of the most likely sequence of hidden states, is implmented to find the

most suitable label for the image by using emmitted probabilities and language model.

An end to end network was proposed by Shi et al Baoguang Shi (July 2015) which uses 7 Convolutional layers, followed by 2 Bidirectinal LSTM layers with 256 hidden cells and a final CTC layer to produce a prediction. In order to limit the labellings of the set of inputs which were in form of single word images, a dictionary of lexicon based trascription was used. Approximate labelling of the words was estimated by best path decoding and then the one with the highest score among the potential candidates was chosen as final labelling.

Similar approach but with different architecture also exists. Graves Graves (2012) extracts nine handcrafted features from the sliding window applied over the image followed by bidirectional LSTM layer containing 100 hidden cells. The resulting sequence is combnined after processing them forward and backward through two seperate RNN layers. Best Path algorithm is used to decode the output and to calculate word and character error rate while using statistical language model and dictionary contaning 2000 words to imporove accuracy.

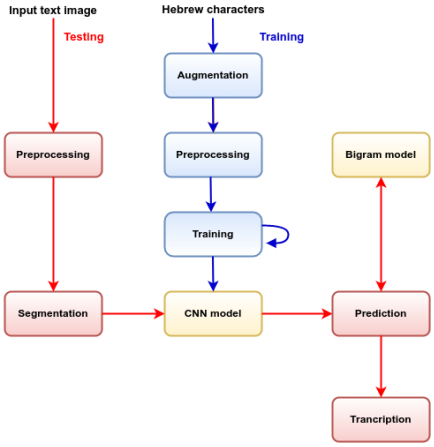

Figure 3.1: Overall pipeline of our character recognition system. The red boxes visualize the stream from raw input documents to their final transcription; the blue boxes, the training process on Hebrew characters; and the yellow boxes, the two models used for the recognition.

3 DSS

3.1 Methodology

In this section, the pipeline of the system, the different components and methods implemented, and the data used are introduced. The overall pipeline of the proposed character recognition system is illustrated in Figure 3.1.

Our approach involves three major processes. First of all, we pre-process the text images in order to increase the readability of their textual contents. Then, we perform segmentation to extract as many individual characters as possible on each image, and finally, we proceed to the prediction part, where all these extracted character-images are recognized to get the whole text of the input image.

3.1.1 DSS Dataset

The Dead Sea Scrolls[1] (DSS) collection consists of ancient manuscripts with immense historical, religious, and linguistic significance. They are approximately two thousand years old, dating from the third century BCE to the first century CE, and discovered in the Qumran Caves in the Judaean Desert near the northern shore of Dead sea.

In this study, we retrieve 20 text images from this collection. These text images are restricted to only Hebrew text. The text images are already binarized to values $\{0, 255\}$ with 0-value being the foreground pixels (text) and 255-value the background ones. Figure 3.2 shows an example of one of such images of the scrolls. From now on, we will use this certain image for presenting our indicative results.

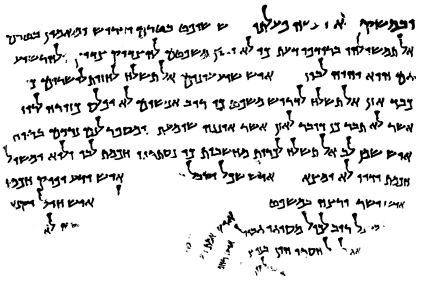

Figure 3.2: An example text image from the Dead Sea Scrolls collection.

[1] https://en.wikipedia.org/wiki/Dead_Sea_Scrolls

3.1.2 Pre-processing

The goal of pre-processing is to improve the quality of an input image so it can be better utilized on the particular task we are working on. Although our images have already been subjected to pre-processing methods, like rotation and binarization, we proceed to some further modifications that will render the segmentation part easier.

To be more precise, since our final task is to recognize single characters, it is of paramount importance to have well-separated and distinguishable characters in our input images. However, we observe that some documents have distorted and erased characters, while others have thicker ones that result in touching characters. To partially resolve this issue, we apply morphological operations to the input images, such as erosion, dilation, and opening, based on their average intensity value. Note that since the values of the images are inverted, i.e., 0-value for the foreground pixels and 255-value for the background pixels, we apply dilation to thin and detach connected objects and opening to enlarge and join broken parts of objects.

Figure 3.3 shows the histogram of the average intensity value of the provided images. Based on this histogram, we set a threshold value equal to 248 to determine the proper pre-processing method for each image. Specifically, images with an average intensity value lower than 248 have thicker text. Thus, they are subjected to dilation operation in order to shrink the foreground objects (text region) and make them thinner. On the other hand, images with an average intensity value larger than 248 are subjected to opening to join their broken parts.

Our example image from Figure 3.2 has an average intensity value of about 241. Figure 3.4 presents the corresponding output image after pre-processing.

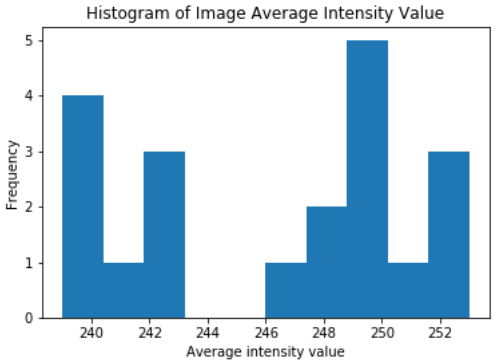

Figure 3.3: Histogram of average intensity value of input text images.

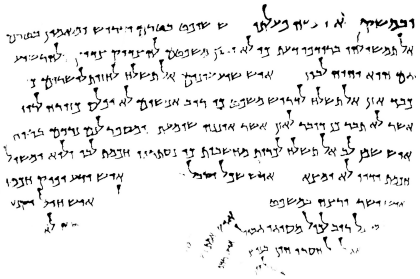

Figure 3.4: The example text image after pre-processing.

3.1.3 Character Segmentation

Once we have the improved version of our input text images, we proceed to the segmentation part. For the character segmentation, we use a contour-based approach which is described in detail below.

3.1.3.1 Contour-based segmentation

Given an input text image, the first step is to detect the borders of all objects/characters and localize them; joining all the points on the boundary of an object, we get a contour. Regarding the technical details, we use the **findContours()** function from the *OpenCV* [2] library and set the retrieval mode to *RETR_TREE* that returns a full hierarchy list of the detected contours. From this contours' hierarchy, we drop the child/internal contours that are inside other larger ones. An indicative example is presented in Figure 3.5.

For the remaining contours, we use the function **boundingRect()** that returns the coordinates (x, y) of their top-left corner, their width w and height h, in order to localize the characters inside.

3.1.3.2 Managing extreme cases

After detecting all the contours of an input image, we evaluate the quality of the segmented characters and of our pre-processing step (Section 3.1.2) with manual inspection. Figure 3.6 illustrates an exam-

[2] https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html

(a) (b)

Figure 3.5: An example contour (a) with the full hierarchy; (b) after dropping the child contour.

ple in which the pre-processing step helps to detach two connected characters.

(a) (b)

Figure 3.6: An example contour (a) without dilation pre-process; (b) with dilation pre-process.

However, there are still examples of extreme cases that need to be resolved. For instance, large contours with characters still attached or very small contours with noise and no actual characters.

Contours with no-characters: To skip the relatively small contours that contain noise, we set a HEIGHT_THRESHOLD = 27 and a WIDTH_THRESHOLD = 12 after having experimented with several values. What we want to ensure is a minimum value for the width of each contour, a minimum value for the height, and also for the circumference of the entire box. Then, we choose to exclude contours that belong to one of the following categories:

1. $h <$ HEIGHT_THRESHOLD or
2. $w <$ WIDTH_THRESHOLD or
3. $h + w < 55$

Contours with connected characters: To handle the larger contours that probably contain connected characters, we pass through all the remaining contours twice. In the first pass, we determine the average width of all of them. In the second, we check the ratio of each contour's width with the average one. If a contour's width is more than double the average width, we assume that we are dealing with two characters under the same contour. This assumption is made after noticing that the characters have, in general, similar widths. If this is the case, we split this contour into two, each with a width equal to the average width. Otherwise we extract the entire contour as it is. At this point, we store those contours in the form $[x, y, w, h]$ as outputted from the **boundingRect()** function.

As it turns out, we only consider the case where a contour contains at most two characters. This approach can also be extended to consider more connected characters for more accurate results.

3.1.3.3 Sorting contours

Having detected all the main contours that correspond to single characters in an input image, the final step is to arrange them so that they are exported in the correct order. *OpenCV* extracts the contours from bottom-to-top considering only the y-coordinate. In our case, we want to extract them from top-to-bottom and from left to right.

To do so, we use the **horizontal projection** approach, which gives a rough estimation of lines. The main problem with this approach by itself is that it cuts off characters in case of skewed text or includes the noise if it exists, as it extracts the entire region around the estimated lines.

However, this is not the case in our approach since we have the exact coordinates of only four points around the characters. So, what we want to preserve is for just one corner of the contours (the same for all) to be in the correct line. This estimated line will result from the histogram. In our implementation, we use the bottom-right corner as the origin. So, if the bottom-right corner of all contours is assigned to the correct line, then we can just extract for each line the respective entire contours using their coordinates. The steps of our implementation are the following:

1. Detect the valleys of the histogram that correspond to text lines.
2. For every two valleys (text lines), estimate their cut-off line (y-coordinate) and force it to be closer to the later valley (bottom line). The

reason for not using the average of the two valleys, or just the peaks of the histogram, is that we want to ensure that the bottom right corner of all contours will be above their "true" cut-off line. So, we want to have a large margin in case of even skewed text.

3. Sort the contours in ascending order based on their bottom right corner y-coordinate.

4. Assign to each cut-off line the contours whose bottom right corner has a smaller y-value.

Every time we go to the next cut-off line means that we have finished with the contours of the previous line. Thus, given only the contours of one line, the only thing left is to sort them based on their x-coordinate from left to right and export them.

3.1.4 Character Recognition

In this subsection, we will describe the two approaches used for character recognition: a classifier-based approach and a bigram-based approach.

3.1.4.1 CNN-based recognition

The Hebrew language consists of 27 characters in total. To recognize the characters in the segmented-character images, we treat this problem as a multi-class classification task and implement a Convolutional Neural Network (CNN) with 27 output classes. The architecture of this CNN is visualized in Figure 3.7.

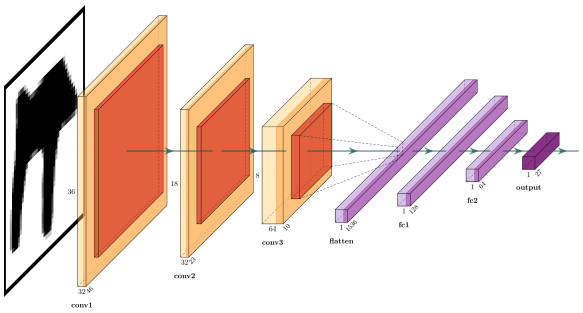

Figure 3.7: The architecture of our convolutional neural network.

This CNN takes as input a 48 by 38 image. It consists of three convolutional layers with 32, 64, and 128 kernels of size 3×3 respectively, each followed by an average pooling over a 2×2 kernel and ReLU activation function. After the convolutional and average pooling layers, the output is flattened. On top of that, we have two dense layers of 128 and 64 units, respectively, followed by a 1×27 output layer. In between the dense layers we still use the ReLU activation, while for the final layer, we use the softmax. The final output is a 27-feature vector with the probabilities of the input image corresponding to each of the Hebrew characters.

3.1.4.2 Bigram-based recognition

For the final recognition, we also make use of bigram data. This data is obtained by modifying the frequency of sequentially occurring characters to create conditional probabilities based on the previously occurring character. This method, therefore, does not apply to the first letter of each line, as there is no character preceding it to base this conditional probability on. We first calculate a frequency matrix of letters occurring after each other. We use the formula:

$$freq_{bigram}(i,j) = \sum_{word} \delta_{word}(c_i, c_j) freq_{word},$$

where $\delta_{word}(c_i, c_j) = 1$ if the i-th and j-th character occur in the sequence we are looking at. To get the probability of character j occurring after character i, we then normalize this amount. This gives us a probability matrix, which we can use to find out the probability of a character, given a vector of probabilies that a recognized character is a certain character k. We do so by doing an element-wise multiplication between the k-th row of the matrix, and the probability vector.

3.1.5 Prediction

Once we have our segmented characters extracted and our model trained (Section 3.2), we proceed to their recognition. Specifically, for each line of a text image, we load all its segmented characters and resize them to 48×38 pixels to fit the input layer of the network. Next, we pass each character in turn from the CNN model and get a 27-feature vector with the output probabilities.

Then, we utilize the bigram model by taking the average between its own probabilities and the model's probabilities and selecting the new argmax. We choose to implement the bigram data in this way, instead of the purely probabilistic Bayesian view, since the way the bigram data is obtained is inherently flawed in some aspects. By taking this mean instead, we make sure to allow for certain character combinations to occur, even though they were not observed in the data. This also minimizes the influence of a wrongly recognized character.

Though the bigram data is available, its use is optional and recommended only for texts with sufficiently high-level recognizable characters. This is because if the first character is incorrectly recognized, then this error will be propagated through their use. In our implementation, we choose to get the argmax index directly from the model's output and this constitutes our final prediction. The last step is to convert this argmax index to the string label it corresponds to.

3.1.6 Transcription

The output of the prediction is a string label for each character. To write the actual character on the output text file, we first have to convert these labels into their actual symbols. To do so, we use a dictionary that matches the names of the characters with their symbols and implement a transcription code that writes the respective symbols in the file given the character labels. Note that the transcription is done per line and after the characters are reversed from right to left, as it is supposed to be in the Hebrew language.

3.2 Experimental Setup

Training dataset The training experiments are conducted on a separate dataset that consists of several sample images for each of the 27 Hebrew characters. These images are already binarized to values $\{0, 255\}$ and labeled. As can be seen in Figure 3.8, the training images are unbalanced; they are balanced towards a realistic distribution of the characters in the final dataset. Since we care more about the total amount of characters being recognized than them being balanced in recognition, we do not need to modify to compensate for this.

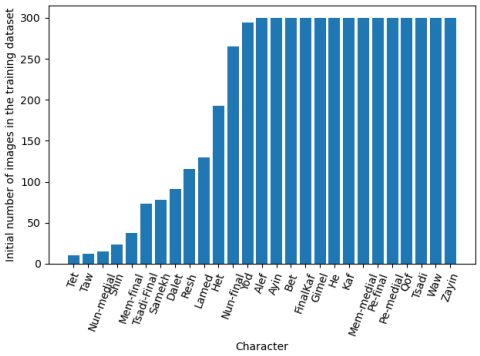

Figure 3.8: Character distribution of the training dataset

The average size of the training images is 48×38 pixels. Hence, this became the input resolution of our CNN as described in Section 3.1.4.1. The provided characters have different qualities; some include parts of other characters; many include a lot of whitespace, mostly to the top or the bottom of the character; characters have different degrees of stroke thickness; some characters are slightly rotated; some include random noise; some are hardly recognizable to the untrained eye. In short, the training set inherently contains some of the problems we expect to find in the Dead Sea Scrolls themselves.

Data augmentation Data augmentation helps a network to become more robust and avoid overfitting. Especially considering that our final test images have occurred after segmenting entire texts, our trained model should be able to deal with different and much-distorted versions of a character. These can be rotated, but are mostly cropped and erased. Thus, for data augmentation, we make use of several different methods. We use the following: rotation, expansion, random cropping, erosion, dilation, random erasing, and adding an area of pixels.

We use these various methods to deal with different problems that can occur with the text within these scrolls. Having rotated images helps deal with characters that were written very skewed, as our bounding boxes will always be a box around a character. Expanding or cropping the image border helps deal with bounding boxes that have been selected wrongly. Eroding and dilating images helps

us deal with thicker or smaller strokes that have been used when writing the text or with dealing with the decay of texts over time. Erasing random pixels helps us deal with noise in data collection and with texts that have decayed over time, leading to incomplete characters. Adding random boxes of data helps us deal with overlapping characters in the bounding boxes drawn around the characters.

Model training Before training the network, each training character image is resized to 48×38 pixels and converted to single channel. Also, we apply binarization using THRESHOLD $= 127$ and normalize the pixel values to $\{0, 1\}$. Finally, we split the entire dataset into train and test with ratio $80 : 20$ and apply label-encoding to the labels of the characters. The network is trained for 25 epochs using a batch size of 64 and a learning rate of 0.001. Adam is adopted as the optimization algorithm, categorical cross-entropy as our criterion and TensorFlow/Keras is used as our deep learning framework.

Model evaluation The results of the network are shown in Figures 3.9 and 3.10. On the augmented training set, the network performs with a validation accuracy of 93.8%. Taking into account the amount of augmentation we have done, we consider this validation accuracy a good baseline, as we expect some of the augmented pictures to be very hard to predict correctly.

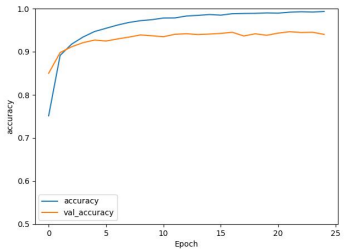

Figure 3.9: The training and validation accuracy of our convolutional neural network.

3.3 Results

The running time of the entire pipeline on the test images given the trained model is about 241 sec.

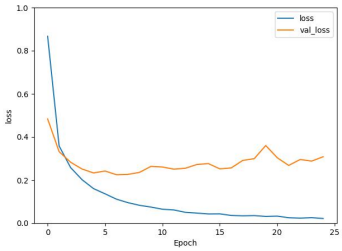

Figure 3.10: The training and validation loss of our convolutional neural network.

We show the intermediate result of segmentation on the example image in Figure 3.11. The output of the entire pipeline on our example image is shown in Figure 3.12 below.

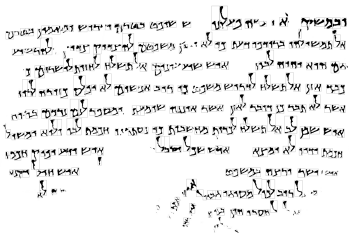

Figure 3.11: The segmented image.

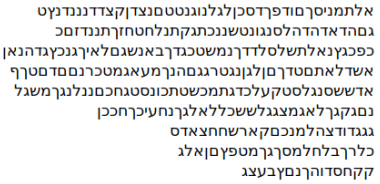

Figure 3.12: The output text image.

4 IAM

4.1 Methodology

This section describes the pipeline and individual components used to solve the IAM line recognition task. An overview is given in Figure 4.1.

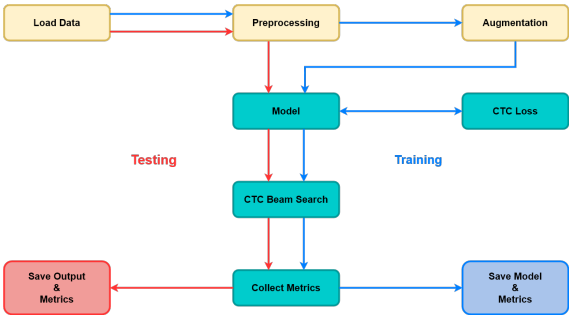

Figure 4.1: IAM pipeline.

4.1.1 IAM Lines Dataset

The IAM database, introduced by Marti and Bunke (2002), consists of handwritten English sentences. Data was collected from 657 different writers who copied a small text on a piece of paper. All text lines were taken from the Lancaster-Oslo/Bergen corpus which makes it possible to generate language models beyond the lexicon level (Favata, Srihari, and Govindaraju (1998) and Kim, Govindaraju, and Srihari (2000), as cited in Marti and Bunke (2002)). The handwritten text has been scanned and the resulting grey-scale imges have been segemented into lines and words.

Our training data is a subset of the original dataset and contains 7458 images and their labels. For the training procedure it is necessary to know the maximal label length (75) and number of characters (79 including space). An overview of the label lengths is given in Figure 4.2 and the number of occurences per character can be seen in Figure 4.3.

We can see that the number of character occurences varies greatly. This was not considered during training but it can help us understand the limitations of our model.

The data was split into three subsets: train, validation and test. The test set contains the last 20% in our data set, the remaining data is shuffled and again 20% are removed for validation. What remains is a train set with 64% of the initial data set.

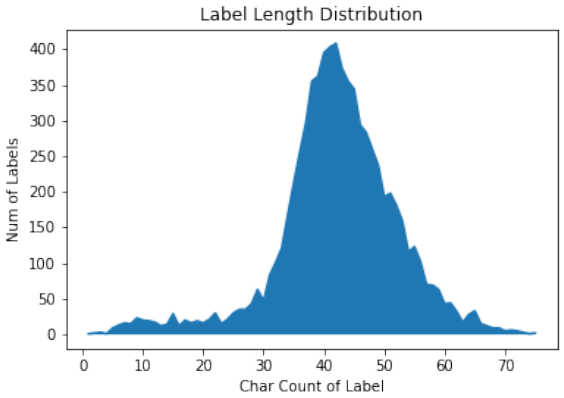

Figure 4.2: Histogram of number of characters per label.

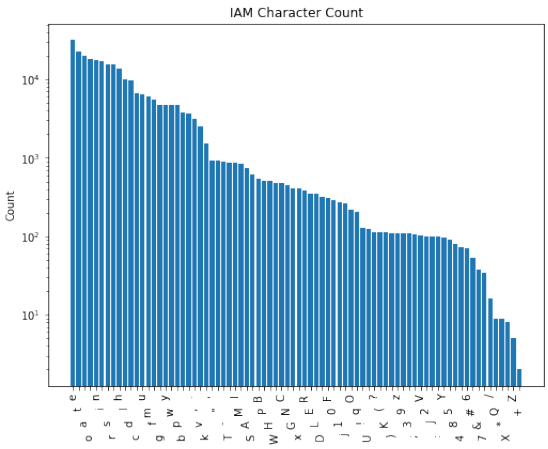

Figure 4.3: Character count for characters in the IAM labels.

4.1.2 Data Preprocessing & Augmentation

Preprocessing for images is done in three steps: First, the color is inverted so that the background has a value of 0 and only the text feeds a signal into the model. Next, the images are resized to a fixed size of 1000×64. The aspect ratio is preserved to prevent distortion and the value 0 is used for padding. Finally, the images are scaled from range [0, 255] to range [0.0, 1.0].

For the labels two preprocessing steps are needed. Labels are encoded by replacing characters with integers and afterwards padded to the same length.

Augmentation is performed on the images during training and consists of four (random) operations. Images are slightly rotated and translated, random contrast is applied and finally a small amount of gaussian noise is added.

4.1.3 Model Architecture

We implemented our model based on the architecture (option 1) presented on page 38 in the PhD thesis by Scheidl (2018). It consists of a CNN, followed by bi-directional LSTMs and a softmax output layer. The convolution and max-pooling operations reduce the height of the input image to one, effectively removing the dimension. This is necessary to feed the image features into the bi-directional LSTM layers as a time-series. Using the output of the softmax layer the CTC loss (Graves et al., 2006) can be calculated to determine the gradients. The output is also used for the CTC decoder which returns a 'path', i.e. a series of (encoded) characters.

A problem we faced with the model was the relatively small number of output timesteps $U = 100$. CTC loss requires that $U > (2 \times T)$ (where T is the label length) which is at least 151 in our case. Scheidl (2018) uses word segmentation algorithms and uses single words as model input. Since we preferred feeding the lines directly into the model it was necessary to increase the number of outputs. The first modification was changing the pool-size of the fourth max-pooling layer from 2×2 to 1×2. This doubled U from `image-width` $\times 0.125$ to `image-width` $\times 0.25$. To further increase U the input width was increased from 800 to 1000, resulting in $U = 250$ which is more than sufficient.

The final layer of the model performs CTC Beam Search (in Scheidl (2018) *Vanilla Beam Search*). It uses a beam width of 100 and returns only the best path.

4.1.4 Evaluation

As error measures we use the Character Error Rate (CER) and Word Error Rate (WER) which are the most common metrics in HTR (Bluche (2015), as cited in Scheidl (2018)). CER can be calculated by deviding the Levenshtein edit distance by the length of the ground truth label. The Levenshtein edit distance is the number of insertions, deletions and substitutions required to transform the model output into the ground truth. WER is calculated analogous to CER, using words instead of characters.

4.2 Experimental Setup

Hardware The hardware specifications of the machine on which the model was trained are shown in Table 4.1.

Param	Value
CPU	AMD Ryzen 3700x, 8 cores, 3.6 GHz
RAM	16 GB, 2666 Hz
GPU	Nvidia RTX 2060 Super, 8 GB
OS	Windows 10

Table 4.1: Hardware.

Model Training In general the training procedure follows that of Scheidl (2018). The model uses the RMSprop optimizer, however the learning rate η was changed from 0.001 to 0.00025 which yielded better results in our case. Batch normalization and early stopping were adopted as described and the batch size used is 24. For the final version of the model we used the entire data set that was provided to us and trained for 25 epochs.

4.3 Results

The evaluation of validation set after training the model for 30 epochs is measured in loss, CER and WER.
A steep descent in curve,starting above 160, is observed in the validation loss curve Figure 4.4 during the period of five epochs after which it gradually decreases before becoming stable around 16.

Similar steep descent, as observed in the loss curve, is noticed in the mean character error rate measure 4.5 for the validation dataset. The final CER of the model was noted to be below 11% by the end of epochs.

However, the mean word error rate of the model was observed to be much higher than the CER with

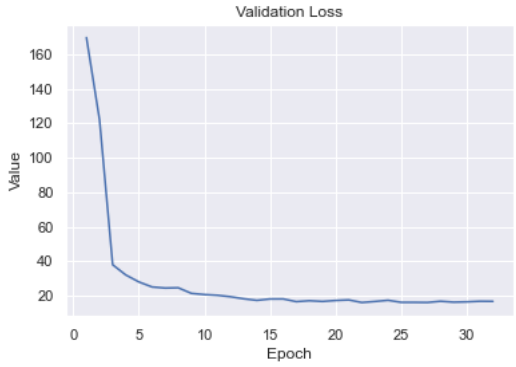

Figure 4.4: Validation loss during training.

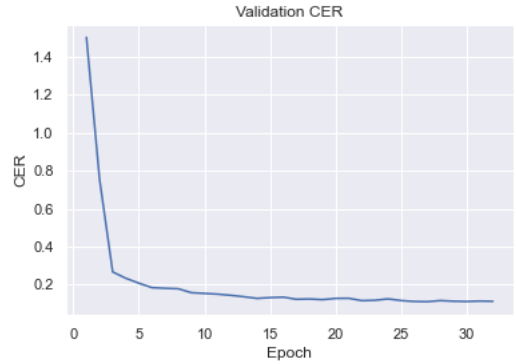

Figure 4.5: Validation CER during training.

the final value of 39.16% as can be seen in Figure 4.6.

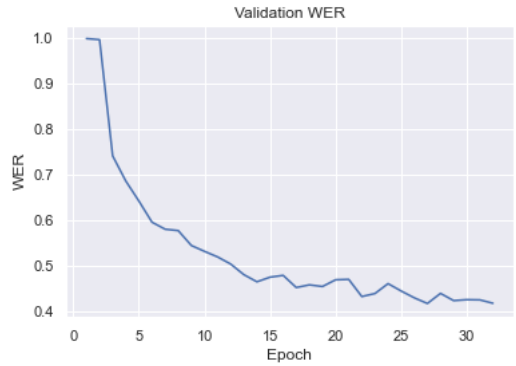

Figure 4.6: Validation WER during training.

Table 4.2 describes the final model metrics for 22 epochs due to early stopping.

	Epochs	CER	WER	Loss
Ours	22	10.96	39.16	15.93
Scheidl	N/A	8.63	28.30	N/A

Table 4.2: Model metrics. Note that Scheidl (2018) used a character language model whereas we did not.

Some of the model's prediction's were absolutely correct even though the handwritten texts were by different authors and styles. It can be affirmed from the Figure 4.7 that model was able to learn and recognize text from images the random thickness of the strokes and variation in style writing.

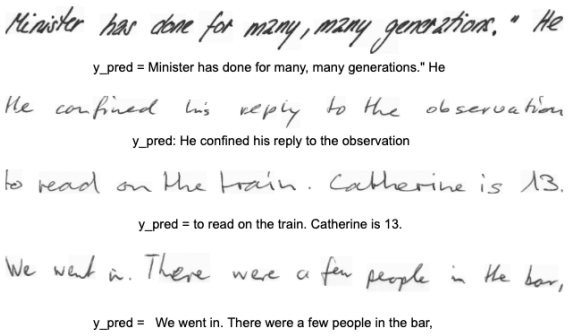

Figure 4.7: Correctly recognized labels

However, some model predcitions were completely wrong while some were partially incorrect due to a missing character or incorrectly recognized character in a word. In some cases where there was presence of similar double character, the model was able to pick up only one resulting in wrong prediction. A few incorrect predictions can be seen in Figure 4.8.

The model was able to recognize and predict the majority of the characters correctly but due to the presence of single incorrect character there was in increase in word error rate. Nevertheless, considering the complexity of the task with different style and fewer training data, our model performed decently with the provided dataset.

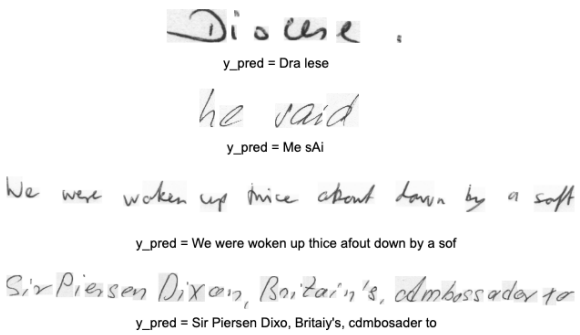

Figure 4.8: Incorrectly recognized labels.

5 Contributions

5.1 Shray Juneja

Worked on the IAM task :

1. Conducted research on model architecture and pipeline.
2. Code for data preprocessing, augmentation along with CTC loss layer & documentation.
3. Inspection of Results.
4. Introduction, Literature review & Results section.

5.2 Luca Mueller

Contributed the following to the IAM task:

1. Research on model architecture & CTC loss
2. Code (except image resizing, CTC loss layer & augmentation module) & documentation
3. Testing & submission of code
4. Model training
5. Abstract, Methods & Experimental Setup section
6. Plots & pipeline schematic

5.3 Chryssa M. Nampouri

Worked on the DSS recognition system:

1. **Conducted research on** handwriting character recognition.
2. **Designed & built** the entire methodology, the end-to-end running pipeline (from DSS loading to transcription) & the documentation.
3. **Implemented the following modules:** main function, pre-processing, segmentation (i.e., contours detection, managing smaller contours, and contours sorting), train (excluding model architecture), prediction, transcription, settings & utility functions.
4. **Report DSS sections:** *Methodology* (DSS Dataset, Pre-processing, Character Segmentation, CNN-based recognition, Prediction, Transcription), *Experimental Setup* (Model training, Model evaluation) & *Results*
5. **Plots & pipeline schematic**

5.4 Wopke de Vries

Worked on the DSS recognition system:

1. **Conducted research on** handwriting character recognition.
2. **Worked on** the bigram model, the CNN model architecture, and on the training experiments and fine-tuning of the model.
3. **Implemented the following modules:** n-grams, data augmentation, segmentation (i.e., managing larger contours).
4. **Report DSS sections:** *Introduction*, *Literature Review*, *Methodology* (Character Recognition, Prediction), *Experimental Setup* (Training dataset, Data augmentation) & *Results*

References

Cong Yao Baoguang Shi, Xiang Bai. An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. July 2015.

Théodore Bluche. *Deep neural networks for large vocabulary handwritten text recognition*. PhD thesis, Paris 11, 2015.

Amit Choudhary, Rahul Rishi, and Savita Ahlawat. A new character segmentation approach for off-line cursive handwritten words. *Procedia Computer Science*, 17:88–95, 2013.

Marc Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.

JT Favata, SN Srihari, and V Govindaraju. Off-line handwritten sentence recognition. *Proceedings of the 6th IWFHR, Taegon, South Korea*, pages 171–176, 1998.

A. Graves. Supervised sequence labelling with recurrent neural networks. 2012.

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

Gyeonghwan Kim, Venu Govindaraju, and Sargur N Srihari. Architecture for handwritten text recognition systems. *Series in Machine Perception and Artificial Intelligence*, 34:163–172, 2000.

Laurence Likforman-Sulem, Abderrazak Zahour, and Bruno Taconet. Text line segmentation of historical documents: a survey. *International Journal of Document Analysis and Recognition (IJDAR)*, 9(2):123–138, 2007.

U.-V. Marti and H. Bunke. Handwritten sentence recognition. 2000. doi: 10.1109/ICPR.2000.903584.

U-V Marti and Horst Bunke. The iam-database: an english sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition*, 5(1):39–46, 2002.

Elisha Qimron. *The Hebrew of the Dead Sea Scrolls*. Brill, 2018.

J. Gorbe-Moya S. Espana-Boquera, M. Castro-Bleda and F. Zamora-Martinez. Improving offline handwritten text recognition with hybrid hmm/ann models. pages 767 – 779, April 2011. doi: 10.1109/TPAMI.2010.141.

G. Leech S. Johansson and H. Goodluck. Manual of infor- mation to accompany the lancaster-oslo/bergen corpus of british english, for use with digital computers. 1978.

Harald Scheidl. *Handwritten text recognition in historical documents*. PhD thesis, Wien, 2018.