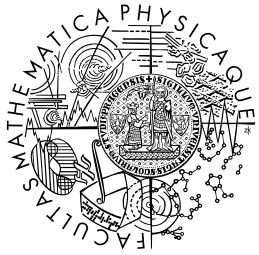


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Štěpán Procházka

Adaptive Handwritten Text Recognition

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: RNDr. Milan Straka, Ph.D.

Study programme: Computer Science

Study branch: Artificial Intelligence

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

To whom it may concern, Thank You.

Title: Adaptive Handwritten Text Recognition

Author: Štěpán Procházka

Institute: Institute of Formal and Applied Linguistics

Supervisor: RNDr. Milan Straka, Ph.D., Institute of Formal and Applied Linguistics

Abstract: The need to preserve and exchange written information is central to the human society, with handwriting satisfying such need for several past millenia. Unlike optical character recognition of typeset fonts, which has been throughly studied in the last few decades, the task of handwritten text recognition, being considerably harder, lacks such attention. In this work, we study the capabilities of deep convolutional and recurrent neural networks to solve handwritten text extraction. To mitigate the need for large quantity of real ground truth data, we propose a suitable synthetic data generator for model pre-training, and carry out extensive set of experiments to devise a self-training strategy to adapt the model to unnanotated real handwritten letterings. The proposed approach is compared to supervised approaches and state-of-the-art results on both established and novel datasets, achieving satisfactory performance.

Keywords: Handwritten Text OCR HTR CTC

Contents

Introduction

The need to preserve and exchange information is central to human society. Over the course of more than four millenia, it has been the handwritting, that played significant role in satisfying such need. From as early as 2600 BCE, when ink was used for writing on papyrus in Ancient Egypt, handwritten text has been a mean to reliable and lasting preservation of information as well as a mean to information exchange, where spoken word was not an option. With the evolution of writing media from the wooden slips, over to the papyrus, the parchment and eventually the paper, together with raising availability of such media and the spread of literacy, the majority of Common Era knowledge and history is captured in manuscripts (i.e., handwritten documents). With the advent of typewriter and digital technologies and their advancements, handwriting is slowly being replaced by typing. Despite such progress, the handwriting maintains its use even in present days, frequently having greater expressive power and often being more accessible to writers.

The digital era not only introduced new means to preserve information in the form of digital documents, it also provided an extensive palette of techniques for further document and natural language processing. Full-text search, topic recognition, semantic parsing or automatic summarization are some but definitely not all examples of such techniques. To be able to apply those advanced tools to handwritten or printed documents, computer scientists formed a field of optical character recognition (OCR) seeking for solutions providing conversion mechanisms from physical documents to their digital counterparts. Due to the social demand — being useful in commercial sector — the task of printed text recognition has been extensively studied and to some degree solved, benefiting from printed document regularities, e.g., fixed glyph shapes and clear composition. Contrary to it, the task of handwritten text recognition (HTR) is being, with small exceptions, rather neglected. Such deficiency can be accounted to substantial complexity of the task, stemming from extensive diversity of handwriting styles and handwriting itself, insufficient quality and quantity of available handwritten documents and considerable cost of annotation of textual contents of the said documents.

As a consequence, our goal is to explore the task of handwritten text recognition and propose a well-performing solution emphasizing modularity, adaptability and ease of application to match the diversity of the task. We want such solution to be based on machine learning models, particularly deep recurrent neural networks. In order to mitigate the lack of annotated data, we want to assess the suitability of data synthesis for the purpose of supervised learning. Apart from that, we expect to need to explore the task of language modelling and to

experiment with self training techniques, both for the purpose of performance gains. The aim is to design, implement and evaluate synthetic data generator, HTR system and methods to adapt it to new, previously unseen data. Based on the cultural context and cultural identity of the author, the implementation and evaluation focuses on Czech cursive handwriting. However, as mentioned earlier, care is taken to design the system preferably with no particular handwriting style or text language in mind, allowing for modifications to meet various needs.

The layout of this thesis is as follows. Chapter 1 gives a detailed overview of the task of optical character recognition, notably its subtasks – text detection (section 1.1) and text extraction (section 1.2). In chapter 2 we discuss dataset synthesis, including an introduction to cursive handwriting (section 2.1.2) and digital typesetting (section 2.1.3), followed by the description of our own data generator (section 2.2). In chapter 3 we propose our handwritten text recognition solution with the description of our model (section 3.1) and measurements of its performance on the synthetic data (section 3.2). Finally in chapter 4 we explore the capabilities of self training to improve the performance of the model on the real handwritten texts (section 4.2) by the means of fine-tuning (section 4.3.1) and by the means of self-training (section 4.3.2), carrying out a multitude of experiments.

1. Optical Character Recognition

This chapter covers the theory of optical character recognition, a field of computer vision studying conversion of physical documents to their digital counterparts. The field concerns a multitude of subtasks, e.g., acquisition, preprocessing and cleanup of digital imagery of physical documents, layout parsing, text detection and text extraction. The extent of particular OCR solution depends on task at hand and may omit or emphasize solving particular subtasks.

In context of this writing, we seek for the solution processing digital greyscale images of scanned pages of documents. The pages are expected to contain a single block of text paragraphs (arbitrarilly aligned and wrapped) occupying the majority of page area both vertically and horizontally. The solution should extract textual contents of said block of text, preserving line wrapping. As a result, we focus on laying basis to methods of text line detection (section 1.1), single-line text extraction (section 1.2) and language modelling (section 1.3).

1.1 Text Detection

One of the preparatory subtasks of OCR systems is text detection, most commonly single text line detection. The need for such component comes from diverse composition of majority of physical documents, with text organically scattered in various positions, often separated using additional graphical elements or interleaved with figures. Various methods can be employed, notably deep learning image detection models seeking rectangular [Zhou et al., 2017] or quadrilateral, perspectively distorted [Liao et al., 2018], regions of text. In simplified cases with no perspective distortion or with regular text alignment (i.e., rectangular block of multiline text), signal processing algorithms such as peak detection may be sufficient.

1.2 Text Extraction

Text extraction as a subtask of OCR can be seen as a task of temporal classification, i.e., a type of sequence to sequence transformation, mapping sequence with, possiby hidden, internal partitioning to sequence of categories of the said parts. In OCR, this translates to mapping sequence of pixel slices, i.e., digital image of concatenation of glyphs, to string of characters.

Extraction of text, particularly handwritten, poses several distinguished challenges. There is no exact relationship between input image dimensions (i.e., the length of the lettering) and the size of output string (i.e., the number of contained

characters). Moreover, when trying to isolate single glyphs, there is a significant degree of fuzziness in detecting glyph boundaries, caused by slanting and use of ligatures (i.e., strokes connecting consecutive glyphs). Finally, in case of handwriting, the glyphs, even for one particular character, are diverse, and change due to the context in which they are being written, as well as simply beacuse they are being produced by hand.

Historically, text extraction was approached using pattern matching techniques, relying on severe restrictions, e.g., small set of fonts with suitable properties, which unfortunately greatly narrowed down the applicability of such solution. Later, hidden Markov models optionally with multilayer perceptrons were adjusted to solve, among others, text extraction. While they were able to improve upon the performance of previous methods, they were principially unable to exploit some inherent properties of data. Nowadays, deep convolutional and recurrent neural networks coupled with connectionist temporal classification successfully overcome the aforementioned difficulties, lifting numerous constraints of previous approaches. Memon et al. [2020] outline the evolution of OCR solutions in greater detail.

1.2.1 Deep Neural Networks

Deep neural networks can be thought of as multilayered structures with layers being parametrized transformations of tensors, often referred to as featuremaps. Different type of layers serve different purposes, e.g., feature extraction, non-linear activation, normalization and reshaping. By composition of multiple layers, neural networks implement composed non-linear parametric functions. Training procedure, i.e., search for optimal parameters exploiting differntiability of neural networks, is employed in order to obtain desired mapping from neural network inputs to its outputs. Provided that the estimation of parameters is stochastic process, neural network outputs often carry probabilistic interpretation of attributes of input data. For introduction to the deep learning refer to Goodfellow et al. [2016].

Like many other computer vision data, digital imagery of written text exhibits high degree of spatial locality of related visual concepts, e.g., strokes of individual glyphs being concentrated in small regions. Moreover, the glyphs form an ordered sequence, following the writing direction. Those inherent properties can be modelled and exploited using deep neural networks composed of convolutional and recurrent layers.

Convolutional layers globally extract local context, using fixed size sliding window (kernel), performing discrete convolution at evenly spaced positions in the input featuremap of the layer. To perform featuremap downsampling, i.e., reduction of its spatial dimensions, larger *stride*, i.e., the distance between sample

positions, is being used.

Recurrent layers extract sequential context, iteratively applying stateful transformation along the temporal direction of input featuremap, outputting transformed sequence and accumulated internal state. Each timestep, the current state is combined with slice of featuremap producing emitted features and new state to be used in next iteration.

1.2.2 Connectionist Temporal Classification

While usage of deep convolutional and recurrent neural networks significantly improves the capability to extract abstract features of written text imagery, it is unable to solve the task of temporal classification on its own. For that, additional measures needs to be taken, carefully designed in a way, which does not hinder the ability to train networks, i.e., preserving their differentiability.

The early attempts to train neural networks to extract written text relied on knowledge of text alignment, i.e., knowledge of boundaries of individual glyphs, allowing to train the network to predict a character for each timestep, based on its position within the said boundaries. While training of such networks is straightforward, it is generally hard, if not impossible, to reliably collect the alignment for training data. Moreover, postprocessing of predictions to obtain labellings is non-trivial as the network models only local, sub character classifications.

Identifying those issues, Graves [2012] came up with Connectionist Temporal Classification (CTC) framework, dropping the need for aligned training data and complex postprocessing steps. This is achieved by convenient reinterpretation of model predictions, modelling the whole label sequences instead of local classifications of timesteps, allowing the model to make label predictions only where necessary and learning the alignment implicitly.

The key concept of CTC is the interpretation of the model outputs. The last layer of the model is expected to be a softmax layer, with the number of channels equal to the number of distinct labels (i.e., characters) plus one, special, *blank* label. The activations of the layer estimate the probabilities of observing corresponding labels at corresponding timesteps given the input, including the extra label, signalling the probability of no proper label being observed. Together, the outputs model the joint conditional probability of all labels at all timesteps, allowing to compute conditional probability of any particular label sequence. Each label sequence, often called *extended labelling*, whose length depends on input dimensions, translates to, potentially shorter, actual *labelling*, such that multiple extended labellings may correspond to the same labelling. The conditional probability of a labelling can then be found by summation of conditional probabilities of all corresponding extended labellings.

Formalization

More formally, the goal is to predict *labelling* l given the input $\boldsymbol{x}$, using model M_θ. It is assumed that $\boldsymbol{l} = (l_1, l_2, \ldots, l_k) \in L^k$ is a finite string of k characters from the fixed character set L and that $L' = L \cup \{\flat\}$ is the extended character set with an extra *blank* label $\flat$. Furthermore, we also assume that $\boldsymbol{x} \in [0,1]^{h \times w}$ is a grayscale bitmap image of a single line of text with height h and width w, and that $M_\theta : [0,1]^{h \times w} \mapsto \mathbb{R}^{T \times |L'|}$ is a neural network (i.e., differentiable parametric function) with parameters θ and stride s, $T = \lceil w/s \rceil$ being the number of output timesteps.

Given the input $\boldsymbol{x}$ and model outputs $\boldsymbol{y} = M_\theta(\boldsymbol{x})$, $y^t_{l'}$ is interpreted as the probability of observing label $l' \in L'$ at time t, given the input sequence $\boldsymbol{x}$ and parameters θ. Together, $\boldsymbol{y}$ estimate the distribution over extended labellings $\boldsymbol{l'} \in L'^T$

$$p(\boldsymbol{l'}|\boldsymbol{x}, \theta) = \prod_{t=1}^{T} y^t_{l'_t} \,, \tag{1.1}$$

implicitly assuming conditional independence of labels at each timestep given $\boldsymbol{x}$ and θ.

Let $\mathcal{B} : L'^T \mapsto L^{\leq T}$ be surjective (i.e., many-to-one) mapping from extended labellings to actual labellings, by first collapsing repeated labels, followed by deletion of blank labels. Figure 1.1 shows all seven extended labellings of labelling `foo` in scenario with $L = \{\mathtt{f}, \mathtt{o}\}$ and $T = 5$. The conditional probability of collapsed labelling $\boldsymbol{l} \in L^{\leq T}$ is a sum of the corresponding extended labellings,

$$p(\boldsymbol{l}|\boldsymbol{x}, \theta) = \sum_{\boldsymbol{l'} \in \mathcal{B}^{-1}(\boldsymbol{l})} p(\boldsymbol{l'}|\boldsymbol{x}, \theta) \,. \tag{1.2}$$

It is mapping $\mathcal{B}$, that allows to drop the requirement for known alignment of data, more specifically it is the invention of the blank label, which serves as both proper label delimiter and null label for timesteps with no particular proper label, e.g., indistinct glyph boundaries or void space in the input data.

Forward-Backward Algorithm

For the purpose of training, relying on pairs $(\boldsymbol{x}, \boldsymbol{l})$ of image and its collapsed labelling, we need to be able to effectively compute probability $p(\boldsymbol{l}|\boldsymbol{x}, \theta)$ of said labelling as defined by equation 1.2. Given that there may be exponentially many extended labellings $\mathcal{B}^{-1}(\boldsymbol{l})$ corresponding to target labelling $\boldsymbol{l}$, such task may seem intractable. Luckily, the task can be solved by a modification of the Viterbi algorithm [Forney, 1973] in polynomial time, by the means of dynamic programming. Moreover, as will be shown later, it can be used to define a differentiable loss function, allowing to propagate gradient to network in each timestep.

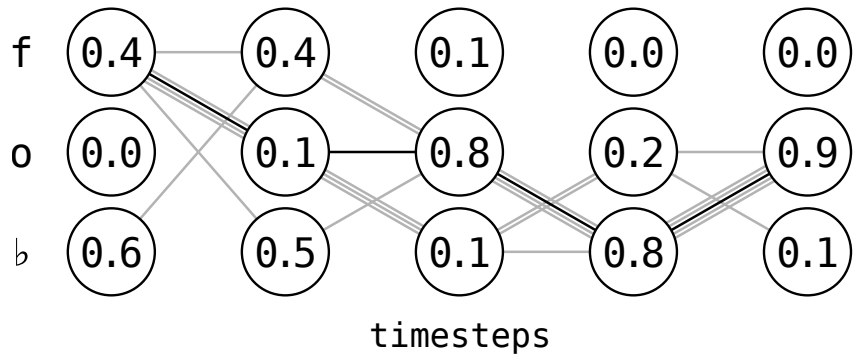


Figure 1.1: **CTC Outputs.** Each node represents the probability $y^t_{l'}$ of corresponding label being predicted at corresponding timestep, the paths show all extended labellings of the most probable labelling `foo` ($p_{\texttt{foo}} = 0.37232$) with one path highlighted, representing extended labelling `foo␣o` ($p_{\texttt{foo␣o}} = 0.02304$).

The main idea is to iterate model outputs $\boldsymbol{y}$ over timesteps, and accumulate probabilities $\boldsymbol{\alpha}$ of prefixes of target labelling $\boldsymbol{l}$ as sums of probabilities of corresponding partial extended labellings, instead of computing conditional probabilities of each whole extended labelling separately. For the needs of training, we will follow a similar procedure, iterating over timesteps in reverse direction, accumulating probabilities $\boldsymbol{\beta}$ of suffixes of target labelling $\boldsymbol{l}$.

Let $\boldsymbol{l'} = (\flat, l_1, \flat, l_2, \ldots, \flat, l_k, \flat)$ be a modified labelling $\boldsymbol{l}$ interleaved with and delimited by blank labels. Moreover, let us define a trellis, an oriented graph $\mathcal{G}$ with nodes $\boldsymbol{v}$ arranged in T columns and $R = |\boldsymbol{l'}|$ rows, v^t_r being a node at row r and column t, and edges $\boldsymbol{e}$ connecting nodes as follows:

$$\forall r \in [1, R], \forall t \in [1, T-1]:$$

$$(v^t_r, v^{t+1}_r) \in \boldsymbol{e} \tag{1.3}$$

$$(v^t_r, v^{t+1}_{r+1}) \in \boldsymbol{e} \quad r < R \tag{1.4}$$

$$(v^t_r, v^{t+1}_{r+2}) \in \boldsymbol{e} \quad l'_r \neq l'_{r+2} \neq \flat \tag{1.5}$$

Each path in graph $\mathcal{G}$ starting in arbitrary node $v^1_?$ in the first column and ending in an arbitrary node $v^T_?$ in the last column corresponds to an extended labelling. Moreover, each extended labelling of target labelling $\boldsymbol{l}$ corresponds to a path in $\mathcal{G}$ starting in v^1_1 or v^1_2 and ending in v^T_{R-1} or v^T_R. This is achieved by edges falling into three categories, representing transitions in extended labelling, i.e., label repetition (1.3), blank to non-blank or non-blank to blank label transition (1.4) and differing non-blank label transition (1.5). See figure 1.2 for example of trellis structure with $\boldsymbol{l} = \texttt{foo}$ and $T = 5$.

Let us move to the definition of accumulated probabilities $\boldsymbol{\alpha}$. Let α^t_r be the probability of all prefixes of extended labellings of length t, ending in either blank

or non-blank label based on parity of r, corresponding to a prefix of the modified labelling $\boldsymbol{l'}$ and transitively to a prefix of labelling $\boldsymbol{l}$. Consequently, the conditional probability of the whole labelling $\boldsymbol{l}$ is

$$p(\boldsymbol{l}|\boldsymbol{x}, \theta) = \alpha_{R-1}^T + \alpha_R^T \,, \tag{1.6}$$

i.e., the sum of probabilities of all complete extended labellings.

The probabilities $\boldsymbol{\alpha}$ can be computed by initialization

$$\alpha_1^1 = y_\flat^1 \tag{1.7}$$
$$\alpha_2^1 = y_{l_1}^1 \tag{1.8}$$
$$\alpha_r^1 = 0 \qquad \forall r > 2 \tag{1.9}$$

and recursion

$$\alpha_r^t = y_{l'_r}^t \begin{cases} \sum_{i=r-1}^{r} \alpha_i^{t-1} & (l'_r = \flat) \vee \left(l'_{r-2} = l'_r\right) \\ \sum_{i=r-2}^{r} \alpha_i^{t-1} & \text{otherwise.} \end{cases} \tag{1.10}$$

The backward probabilities $\boldsymbol{\beta}$ are defined similarly, with β_r^t being the probability of all suffixes of extended labellings of length $T - t$, corresponding to a suffix of modified labelling $\boldsymbol{l'}$ and transitively to a suffix of the labelling $\boldsymbol{l}$. The probabilities can be computed by initialization

$$\beta_R^T = \beta_{R-1}^T = 1 \tag{1.11}$$
$$\beta_r^T = 0 \qquad \forall r < R - 1 \tag{1.12}$$

and recursion

$$\beta_r^t = \begin{cases} \sum_{i=r}^{r+1} \beta_i^{t+1} y_{l'_i}^{t+1} & (l'_r = \flat) \vee \left(l'_{r+2} = l'_r\right) \\ \sum_{i=r}^{r+2} \beta_i^{t+1} y_{l'_i}^{t+1} & \text{otherwise.} \end{cases} \tag{1.13}$$

See figure 1.2 for example of computation of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.

It should be noted that such definition of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ is not coincidental, and has several beneficial properties. First and foremost, for arbitrary timestep t and row r, the product $\alpha_r^t \beta_r^t$ corresponds to the probability of all extended labellings of $\boldsymbol{l}$ going through the node v_r^t. Consequently, for arbitrary timestep t, the aggregation of those per-node probabilities equals to the probability of observing the labelling $\boldsymbol{l}$.

$$\sum_{r=1}^{R} \alpha_r^t \beta_r^t = p(\boldsymbol{l}|\boldsymbol{x}, \theta) \tag{1.14}$$

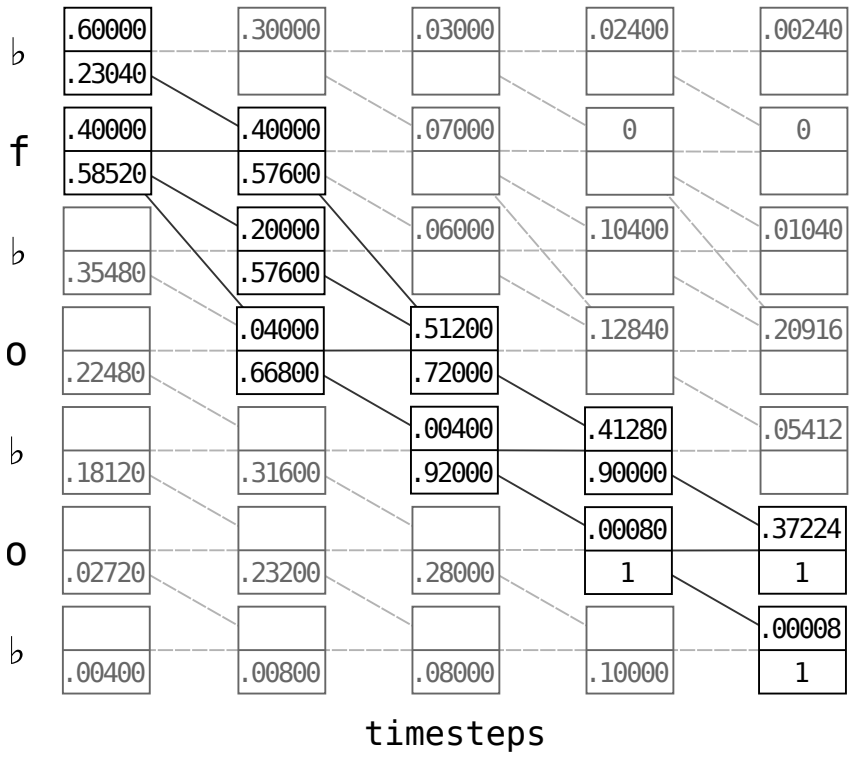


Figure 1.2: **CTC Forward-Backward Computation.** Each node represents pair of variables α and β at corresponding timestep and position in modified label sequence, each path using one transition per timestep corresponds to some extended labelling. Paths formed of highlighted edges correspond to all extended labellings of labelling $\boldsymbol{l} = \texttt{foo}$

Objective Function

The goal is to learn θ using training data $\boldsymbol{T} = \{(\boldsymbol{x}, \boldsymbol{l})_1, (\boldsymbol{x}, \boldsymbol{l})_2, \ldots, (\boldsymbol{x}, \boldsymbol{l})_n\}$, i.e., dataset of pairs – image and its true labelling. To do so, we derive an objective function $\mathcal{L}$ and find its derivatives with respect to model outputs $\boldsymbol{y}$, which allows us to propagate gradient to model parameters θ and use common gradient descent training procedures. As usual, the objective function $\mathcal{L}$ is based on the principle of maximum likelihood, i.e., minimization of negative log probability of correctly labelling the training set $\boldsymbol{T}$:

$$\mathcal{L} = -\ln\left(\prod_{(\boldsymbol{x},\boldsymbol{l})\in\boldsymbol{T}} p(\boldsymbol{l}|\boldsymbol{x},\theta)\right) = -\sum_{(\boldsymbol{x},\boldsymbol{l})\in\boldsymbol{T}} \ln p(\boldsymbol{l}|\boldsymbol{x},\theta)\,. \tag{1.15}$$

Then, the derivative of $\mathcal{L}$ with respect to particular $y^t_{l'}$, assuming only one training example $(\boldsymbol{x}, \boldsymbol{l})$ for simplification, takes the following form:

$$\frac{\partial\mathcal{L}}{\partial y^t_{l'}} = -\frac{\partial \ln p(\boldsymbol{l}|\boldsymbol{x},\theta)}{\partial y^t_{l'}} = -\frac{1}{p(\boldsymbol{l}|\boldsymbol{x},\theta)}\frac{\partial p(\boldsymbol{l}|\boldsymbol{x},\theta)}{\partial y^t_{l'}}\,. \tag{1.16}$$

Now we can make use of probabilities $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ computed using the Forward-Backward algorithm (section 1.2.2). Let us first compute the partial derivations of α^t_r and β^t_r with respect to $y^t_{l'}$, observing definitions (1.7 to 1.10) and (1.11 to 1.13), respectively. The $\boldsymbol{l'}$ is the modified labelling as defined in section 1.2.2.

$$\frac{\partial\alpha^t_r}{\partial y^t_{l'}} = \begin{cases} \frac{\alpha^t_r}{y^t_{l'}} & l' \in \boldsymbol{l'} \\ 0 & \text{otherwise} \end{cases} \tag{1.17}$$

$$\frac{\partial\beta^t_r}{\partial y^t_{l'}} = 0 \tag{1.18}$$

Combining these results, the derivation of the product $\alpha^t_r\beta^t_r$ is

$$\frac{\partial\alpha^t_r\beta^t_r}{\partial y^t_{l'}} = \begin{cases} \frac{\alpha^t_r\beta^t_r}{y^t_{l'}} & l' \in \boldsymbol{l'} \\ 0 & \text{otherwise.} \end{cases} \tag{1.19}$$

Finally, with known derivative of $\alpha^t_r\beta^t_r$, together with the equation 1.14, we can expand the partial derivative of objective function $\mathcal{L}$ (1.16) to

$$\frac{\partial\mathcal{L}}{\partial y^t_{l'}} = -\frac{1}{p(\boldsymbol{l}|\boldsymbol{x},\theta)y^t_{l'}}\sum_{\{r|l'=\boldsymbol{l'}_r\}}\alpha^t_r\beta^t_r\,. \tag{1.20}$$

Decoding

So far we have explored the algorithms for training of neural networks with CTC, relying on efficient computation of conditional probabilities of the target, known

labellings. Contrary to that, the task of prediction naturally lacks such information, and there does not exist any tractable approach to fully compute conditional probabilities of all (exponentially many) possible labellings. Consequently, we need to devise approaches estimating best labellings, often called decoding algorithms.

Greedy Decoding

The simplest method to decode CTC outputs is the greedy decoding, and it is based on the assumption that the best output labelling is the one corresponding to the most probable extended labelling. The best extended labelling, i.e., the sequence of the most probable labels at every timestep, is simply contracted using mapping $\mathcal{B}$ (see section 1.2.2):

$$\boldsymbol{l} = \mathcal{B}\left(\underset{t\in\{1,2,\ldots,T\}}{\operatorname{concat}}\left(\underset{l'\in L'}{\arg\max}\, y^t_{l'}\right)\right). \tag{1.21}$$

Such decoding algorithm is fast and easy to use as it has no parameters. On the other hand, it does not guarantee choosing the actual best labelling, and lacks the means to integrate additional knowledge to the decoding process. See figure 1.1 for an example of the most probable extended labelling bbobo not corresponding to the most probable labelling foo.

Beam Search Decoding

Mitigating the shortcomings of greedy decoding, the beam search decoding gives a better approximation of the most probable labelling, by considering multitude of extended labellings. As the name suggests, the decoding algorithm searches the space of feasible labellings using a beam search algorithm.

In the context of CTC decoding, beam search with beam width w is carried out using dynamic programming approach, effectively generalizing ideas behind forward pass of the Forward-Backward algorithm (section 1.2.2). The algorithm iterates over the timesteps of CTC outputs, keeping track of up to w most probable labelling prefixes, based on probabilities of their corresponding partial extended labellings. At timestep t, partial extended labellings corresponding to candidate labelling prefixes are expanded, updating estimated probabilities of already tracked labelling prefixes, or creating previously unseen ones. At the end of the timestep, candidate labelling prefixes are pruned, leaving only w most probable ones, to be extended in the next timestep.

For that, the algorithm keeps track of $\boldsymbol{b}$ (blank) and $\boldsymbol{n}$ (non-blank) probability estimates, with the following interpretation. Given the model outputs $\boldsymbol{y}$, labelling $\boldsymbol{l}$ and probabilities $\boldsymbol{\alpha}$ as defined in section 1.2.2, a *blank* probability estimate $b^t_{\boldsymbol{l}}$

approximates the sum of probabilities of partial extended labellings of length t ending with the blank label $\flat$ and corresponding to the labelling $\boldsymbol{l}$:

$$b_{\boldsymbol{l}}^t \approx \alpha_{2|\boldsymbol{l}|+1}^t = \sum_{\{\boldsymbol{l}'|\boldsymbol{l}'\in L'^t, \mathcal{B}(\boldsymbol{l}')=\boldsymbol{l}, l'_t=\flat\}} \left(\prod_{t'=1}^{t} y_{l'_{t'}}^{t'} \right) . \tag{1.22}$$

Similarly, a *non-blank* probability estimate $n_{\boldsymbol{l}}^t$ approximates the sum of probabilities of partial extended labellings of length t ending with non-blank label and coresponding to the labelling $\boldsymbol{l}$:

$$n_{\boldsymbol{l}}^t \approx \alpha_{2|\boldsymbol{l}|}^t = \sum_{\{\boldsymbol{l}'|\boldsymbol{l}'\in L'^t, \mathcal{B}(\boldsymbol{l}')=\boldsymbol{l}, l'_t\neq\flat\}} \left(\prod_{t'=1}^{t} y_{l'_{t'}}^{t'} \right) . \tag{1.23}$$

As a result, it holds that the sum of corresponding estimates at time t for the labelling $\boldsymbol{l}$ approximates probability of the said labelling, restricted to the first t timesteps of the model outputs $\boldsymbol{y}$:

$$b_{\boldsymbol{l}}^t + n_{\boldsymbol{l}}^t \approx p(\boldsymbol{l}|\boldsymbol{y}^{1:t}) . \tag{1.24}$$

The algorithm computes $\boldsymbol{b}$ and $\boldsymbol{n}$, by initialization

$$\boldsymbol{n} = 0 \tag{1.25}$$

$$\boldsymbol{b} = 0 \tag{1.26}$$

$$b_{\varnothing}^0 = 1 \tag{1.27}$$

and recursion

$$b_{\boldsymbol{l}}^t = y_{\flat}^t (b_{\boldsymbol{l}}^{t-1} + n_{\boldsymbol{l}}^{t-1}) \tag{1.28}$$

$$n_{\boldsymbol{l}m}^t = y_m^t \left(b_{\boldsymbol{l}}^{t-1} + n_{\boldsymbol{l}m}^{t-1} \right) + \begin{cases} 0 & \boldsymbol{l} \text{ ends with } m \\ y_m^t n_{\boldsymbol{l}}^{t-1} & \text{otherwise,} \end{cases} \tag{1.29}$$

with $\boldsymbol{l}m$ being the concatenation of the labelling $\boldsymbol{l}$ with the label m.

Computing full $\boldsymbol{b}$ and $\boldsymbol{n}$ would lead to them containing the actual probabilities of labellings. However, this is intractable in practice, due to their exponential size. Consequently, the beam search of beam width w is used, keeping only w best labellings and their corresponding probability estimates at the end of each timestep. The dropped values are considered being zero when being part of further computations. See algorithm 1 for programmatic description.

When processing sequences with inherent structure, e.g., semantics and syntax of natural languages, it may be beneficial to make it part of the decision process of the decoder. Formally, we want to select labelling $\boldsymbol{l}$ maximizing a conditional probability $p(\boldsymbol{l}|\boldsymbol{x}, G)$, G modelling the structure, i.e., grammatics, of labellings. Such task can be approximated by ensembling multiple models, each modelling different aspects of input sequences:

$$\arg\max_{\boldsymbol{l}\in L^{\leq T}} p(\boldsymbol{l}|\boldsymbol{x}, G) = \arg\max_{\boldsymbol{l}\in L^{\leq T}} (p(\boldsymbol{l}|\boldsymbol{x}, G))^2 \approx \arg\max_{\boldsymbol{l}\in L^{\leq T}} \left(p(\boldsymbol{l}|\boldsymbol{x}) \cdot p(\boldsymbol{l}|G) \right) , \tag{1.30}$$

Algorithm 1 Beam Search Decoding [with External Scorer]

Inputs

$\boldsymbol{y} \leftarrow$ model outputs, i.e., label probabilities

Outputs

list of pairs – labelling and its estimated probability

Parameters

$w \leftarrow$ beam width

$\mathcal{S} \leftarrow$ external scoring function or $L^{\leq T} \mapsto 1$

$\gamma \leftarrow$ external score weight or 1

function DECODE($\boldsymbol{y}$)
  $b^0_{\varnothing} \leftarrow 1$ ▷ initial blank probability estimate
  $Beam \leftarrow \{\varnothing\}$ ▷ beam initialization
  for $t \leftarrow 1, T$ **do**
    for $\boldsymbol{l} \in Beam$ **do**
      $b^t_{\boldsymbol{l}} \overset{+}{\leftarrow} y^t_{\flat}(b^{t-1}_{\boldsymbol{l}} + n^{t-1}_{\boldsymbol{l}})$ ▷ extend with the blank
      for $m \in L$ **do**
        $s \leftarrow \mathcal{S}(\boldsymbol{l}m)$
        if $\boldsymbol{l}$ ends with m **then**
          $n^t_{\boldsymbol{l}} \overset{+}{\leftarrow} y^t_m n^{t-1}_{\boldsymbol{l}}$ ▷ repeat a non-blank
          $n^t_{\boldsymbol{l}m} \overset{+}{\leftarrow} y^t_m b^{t-1}_{\boldsymbol{l}} s^{\gamma}$ ▷ extend with a non-blank after the blank
        else
          $n^t_{\boldsymbol{l}m} \overset{+}{\leftarrow} y^t_m (b^{t-1}_{\boldsymbol{l}} + n^{t-1}_{\boldsymbol{l}}) s^{\gamma}$ ▷ extend with a non-blank
        end if
        if $\boldsymbol{l}m \notin Beam$ **then**
          $b^t_{\boldsymbol{l}m} \overset{+}{\leftarrow} y^t_{\flat}(b^{t-1}_{\boldsymbol{l}m} + n^{t-1}_{\boldsymbol{l}m})$ ▷ carry a blank estimate
          $n^t_{\boldsymbol{l}m} \overset{+}{\leftarrow} y^t_m n^{t-1}_{\boldsymbol{l}m}$ ▷ carry a non-blank estimate
        end if
      end for
    end for
    $Beam \leftarrow (\arg\operatorname{sort}_{\boldsymbol{l} \in L^{\leq t}} (b^t_{\boldsymbol{l}} + n^t_{\boldsymbol{l}}))_{1:w}$ ▷ select the w best labellings
  end for
end function

where $p(\boldsymbol{l}|\boldsymbol{x})$ is a CTC trained neural network and $p(\boldsymbol{l}|G)$ an external scorer, e.g., a language model (see the following section 1.3).

Assuming that $p(\boldsymbol{l}m|G) = p(m|\boldsymbol{l}; G)p(\boldsymbol{l}|G)$, we seek a mapping $\mathcal{S} : L \times L^{<T} \mapsto \mathbb{R}_0^+$ from labellings, i.e., sequences of labels, to scores, i.e., non-negative real numbers, approximating the conditional probability distribution $p(m|\boldsymbol{l}; G)$. The beam search decoding, provided its iterative nature, can be adjusted to rely on such external scorer, including the extension score whenever a labelling gets prolonged (see instruction 8 of the algorithm 1).

1.3 Language Modelling

When extracting written text, it is often beneficial to model not only its visual properties, but also its linguistic properties. Language models, i.e., probability distributions $p(\boldsymbol{u})$ of a token sequence $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)$ being an utterance in modelled language, are of particular use. Such models are thoroughly studied by the field of natural language processing.

Historically, rule based models based on expert knowledge were used, but they were hard to maintain and improve, lacking the ability to fit the organicity of natural languages. Later, the so called *n-gram* language models were proposed, proving to be simple yet satisfactorilly performant. Nowadays, deep learning based solutions are being invented [Raffel et al., 2019], outperforming the former approaches at the cost of higher data demand and computational complexity. We will further focus on n-gram models, for their relevance to this writing.

1.3.1 N-gram Language Models

Many types of language models are derived from probabilistic chain rule

$$p(\boldsymbol{u}) = p(u_1)p(u_2|u_1)\ldots p(u_n|u_{n-1}, u_{n-2}, \ldots, u_1)\,. \tag{1.31}$$

However, such derivation, requiring to model sample space of exponential size, still does not allow for tractable definition of applicable language model. Consequently, n-gram models further assume that token u_i depends on a history of bounded size n (often called order), i.e., they assume every utterance u to have Markov property of order n:

$$p(\boldsymbol{u}) = \prod_{i=1}^{n} p(u_i|u_{i-1}, u_{i-2}, \ldots, u_{i-n})\,. \tag{1.32}$$

In order to approximate such conditional probability distribution, the principle of maximum likelihood estimate is applied on training data. This is performed by counting frequencies of n-grams and $(n-1)$-grams, and computing their ratios:

$$p(u_i|u_{i-1}, u_{i-2}, \ldots, u_{i-n}) = \frac{\#(u_{i-n}, u_{i-n+1}, \ldots, u_i)}{\#(u_{i-n}, u_{i-n+1}, \ldots, u_{i-1})}\,. \tag{1.33}$$

While theoretically sound, such definition of n-gram models is hardly applicable, as it assigns zero probability to any utterance u containing n-gram not present in the training data, an event common with usage of, e.g., proper nouns or amounts. A solution to this issue is the use of smoothing, i.e., a method to redistribute probability mass so that no event has zero probability. As with any general concept, there exist a multitude of smoothing approaches, with Kneser-Ney smoothing [Ney et al., 1994] widely accepted as a de-facto standard one.

1.3.2 CTC Decoding with External Scorer

In the context of written text extraction using CTC neural networks, language models can be used as external scorers in the beam search decoding algorithm as described in section 1.2.2, with beneficial impact. The n-gram models are particularly efficient as external scorers, providing the extension score is computed solely from the trailing n-gram.

Care needs to be taken to compensate for length of the labellings, as the probability of labelling $\boldsymbol{l}$ comprised of w words and c characters decays exponentially in w for word-level language models, and in c for char-level language models. Such skewed scores may result in steering the beam search to prefer labellings shorter than the actual one. This issue is usually adressed by introducing length dependent term, optionally weighted, with resulting score being

$$s(\boldsymbol{l}) = p(\boldsymbol{l}) \cdot |\boldsymbol{l}|^{\beta} \,. \tag{1.34}$$

with $|\boldsymbol{l}|$ being either the count of words w or characters c based on model granularity.

2. Data Synthesis

This chapter covers the task of data synthesis for the purpose of supervised learning of HTR models, based on deep neural networks. In the sections to come we explore existing solutions (section 2.1.1), cursive handwriting (section 2.1.2) and digital typesetting (section 2.1.3), allowing us to propose our own approach to artificial lettering synthesis (section 2.2).

The motivation for data synthetization is twofold. Firstly, supervised learning demands substantial amount of annotated data in the form of pairs – an image and its text contents. Artificial data generator may serve as virtually endless source of such data with speed, reliability and cost incomparable to a human annotator or writer. However, care needs to be taken to ensure that artificial data does not lack diversity, inherent to captures of physical documents, their attrition, contents, composition and style. Secondly, with artificial synthetization, the generator possesses full control over generated data properties, e.g., textual contents, visual appearance or composition. Achieving such level of consistency and adaptability to task at hand solely with manpower may be tedious or even impossible, especially for dead languages, intricate compositions or uncommon alphabets.

In spite of the aforementioned benefits, usage of synthetic data is not that common, mainly due to the complexity of a design and an implementation of a faithful data generator. Such task effectively equals to an implementation of realistic document contents and a composition generator, together with an implementation of or a programmatic integration with a typesetting system, providing, among others, non-standard features for non-trivial visual and structural distortion of generated data.

2.1 Preliminaries

Let us define several terms common to the following sections. For our purposes, a *text* is a readable form of piece of information communicated in natural language. It is an ordered sequence of atomic units – *characters* – drawn from a finite set of symbols specific to the language. Such character set can be subdivided into categories, e.g., letters, digits, punctuation marks, etc. The text has its visual representation, i.e., plannar composition of *glyphs*, which are visual representations of individual characters. Such visual representation can be physically realized as, e.g., ink on paper (hand-drawn, printed) or displayed digital image.

2.1.1 Related Work

The idea of written text synthetization is not entirely novel and existing systems have already employed some data synthetization techniques into their workflow.

The well known open source OCR engine Tesseract [Smith, 2007] offers lettering synthesis as part of the model training process. It does so by employing a dictionary with unigram and bigram frequencies, together with a large repository of more than 4500 fonts, in order to synthetize lines of random texts typed in random fonts. To mimic deterioration of real data, the generator optionally applies augmentations to the rendered image, e.g., blurring, adding white noise, rotating or inverting, or changing exposure. In general, the synthetizer is designed to generate typed-like letterings.

2.1.2 Cursive Handwriting

A multitude of handwriting styles has been invented over the course of millenia, improving writing speed and legibility going hand in hand with evolution of writing instruments. With the improvements to quill and later transition to pen between the 16th and 19th centuries, enabling longer, curved and uninterrupted strokes, looped cursive handwriting was invented. This style distinguishes itself from the others having all glyphs of each word actually or at least seemingly written in one stroke. For the sake of better writing speed, glyphs are formed of loops, conveniently minimizing undesirable concentration of ink. Figure 2.1 (excerpt from Blahouš [1902]) shows Czech Cursive Alphabet as taught at schools at the beginning of the 20th century. For non-Czech writers, examining the use of accented characters may be of particular interest.

It should be noted that cursive handwriting does not establish one particular set of glyph shapes, but forms rather a family of derived styles sharing aforementioned properties. Moreover, by the nature of handwriting, each person evolves its own particular writing style or set of styles, resembling ideal shapes only to some degree. On top of that, each written glyph is unique as it is impossible to maintain consistent glyph shapes at all times.

2.1.3 Digital Typesetting

Typesetting is the process of machine conversion of text to its visual representation. In digital typesetting, such conversion processes strings of characters accompanied with style and composition information, outputting visual representation in either vector or bitmap form.

The concept central to typesetting is the use of fonts. Digital font is a file containing descriptions of glyphs together with metrics, i.e., geometrical properties relevant to glyph positioning. Based on the type of a font file, each glyph is

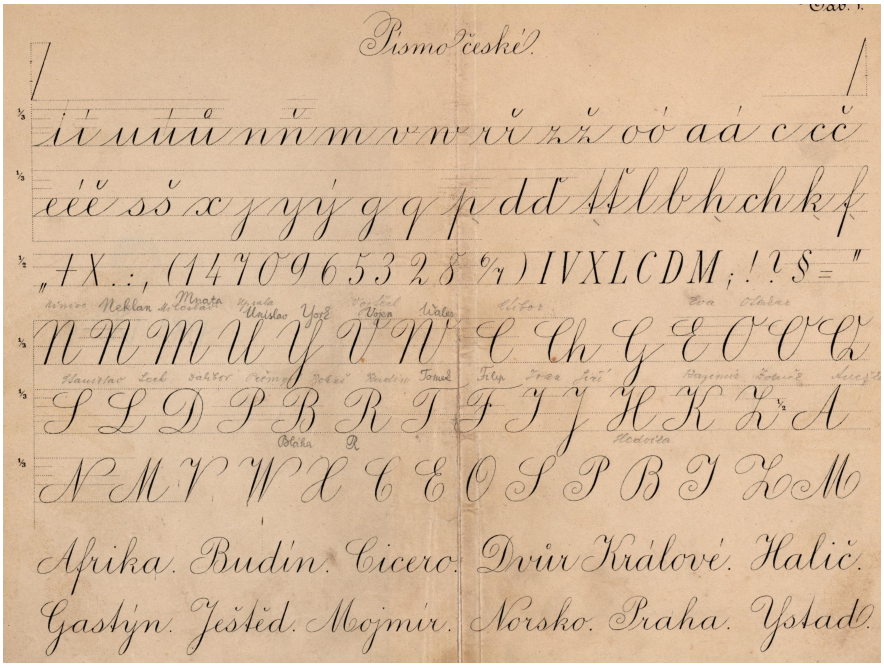


Figure 2.1: Czech Cursive Alphabet by Blahouš [1902].

defined as a dot matrix or a set of Bezier curves (see figure 2.2 and highlighted nodes of the glyph outline). Apart from a shape, a font defines rules for positioning of each character given the surrounding glyphs and desired composition in the form of metrics.

The process of typesetting of each glyph is as follows (see figure 2.2). We maintain a cursor (initially *origin*), a point in space, relative to which the next glyph will be placed. *Bearing* defines the offset from the cursor to the top-left corner of the glyph bounding box and allows for proper glyph placement. After the glyph is placed, the cursor is moved by *advance*, ready for a placement of the next glyph. Some fonts also define *kerning*, i.e., a correction to advance based on the pair of the current and the succeeding glyph. For the purposes of proper line positioning and spacing relative to *baseline*, *ascent*, *descent* and *x-height* are defined, denoting the highest point, the lowest point and the lower-case height, respectively.

2.2 Our Synthetizer

In this section we describe our approach to handwritten text synthetization. Our goal is to mimic high quality captures of well preserved manuscripts with regular text flow (i.e., text paragraphs with fixed line width), e.g., handwritten chronicles captured with a flatbed scanner. We do not aim to simulate wear and tear present in some physical documents, as well as artifacts of some capturing processes, such as perspective distortion or uneven lighting. We expect the outputs to be rendered

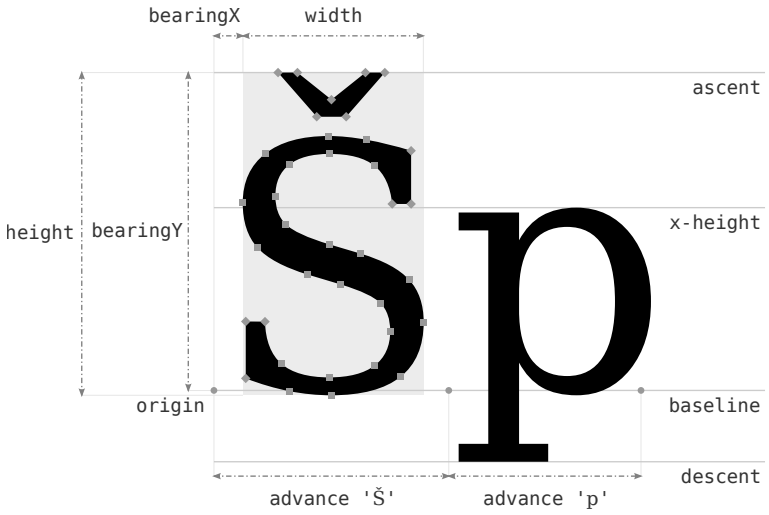


Figure 2.2: Glyph Metrics

lines of text accompanied by metadata, notably textual contents.

We propose a solution based on typesetting of decorative and handwriting-like digital fonts, with subsequent transformations of individual glyphs to simulate unique handwriting styles and their internal inconsistencies. We identify and model four properties of individual handwriting styles, namely *weight* accounting to the stroke width, *slant* controlling the angle of the principal axis of glyphs to the baseline, *scale* and finally *baseline consistency*, which translates to the deviation from an ideal straight horizontal baseline. Each of these properties is represented as a range of admissible values to be drawn from when typesetting each glyph, resulting in its randomized appearance, contrary to the usage of a single value per property, common in typesetting of printed text.

We see several benefits of such setup. Firstly, the generator relies mainly on readily available data, namely decorative digital fonts [Google, 2021] and optionally language corpora [Křen et al., 2016], both easily obtainable from public domain or for research purposes. Secondly, the configuration of the generator is simplistic, comprising only from a few parameters with straightforward interpretation. Finally, while being out of scope of this writing, we see potential in possibility to design a digital font based on a task at hand, i.e., idealized form of its inherent handwriting style, mitigating the need to gather and annotate real data.

2.2.1 Generator Pipeline

The architecture of our solution takes form of a pipeline, i.e., directed acyclic graph of transformations, yielding images with metadata. We choose such approach, for it ensures modularity and maintains the separation of concerns of the

individual components. We present a detailed description of each component in the rest of this section.

Contents Generator

Essential component of the pipeline is the textual contents generator. In our case, we rely on natural language corpus, containing newspaper articles and excerpts from prose, from which we randomly draw *paragraphs* of text. However, any source of text can be used, e.g., a generator based on n-gram distributions, or even a random string generator.

Style Generator

Complementary to the contents generator a style generator is employed. A random font is drawn from the collection of available fonts, and it is transformed into *handwriting font* via a wrapper, which simulates particular handwriting style. The wrapper controls the extent of per-glyph transformations, providing randomly generated ranges for allowed scale, slant, weight and rotation.

When later querrying the handwriting font for a particular glyph, the base glyph is fetched from the underlying font and both its shape and metrics are distorted. First, the glyph weight is adjusted (proportionally to the weight factor drawn uniformly from the corresponding range), simulating thinner or thicker strokes. Then, the weighted glyph is slanted, rotated and scaled in this order, using affine transformations (drawing the slant angle, rotation and scale uniformly from the corresponding ranges). Scaling does not necessarily preserve aspect ratio of glyphs, allowing to simulate dense or conversely sparse handwriting styles. The glyph metrics are transformed appropriately.

Apart from well known affine transformations, we define custom transformations, namely weight adjustment and slanting. Weight adjustment stands for blending, i.e., per-pixel convex combination, of the glyph render with its eroded or dilated counterpart (see figure 2.3). Slanting by angle α is performed as a sequence of affine transformations, namely shearing in y-axis ($(x, y) \mapsto (x, \tan(\alpha)x + y)$), followed by uniform scaling to preserve the original line length and a rotation by the angle α (see figure 2.4).

Typesetter

Having defined both the contents generator and the style generator, their outputs are combined as an input to a typesetting component. The typesetting component types a given paragraph of text using the provided handwriting font, following the principles introduced in section 2.1.3. The result of such transformation is a set of *lines*, forming the whole paragraph. The lines are wrapped on spaces,

(a) $w = -1.0$ (b) $w = -0.5$ (c) $w = 0.0$ (d) $w = 0.5$ (e) $w = 1.0$

Figure 2.3: **Weight transformations using weight factor w.** The original font render ($w = 0$, fig. 2.3c) is either eroded ($w = -1.0$, fig. 2.3a) or dilated ($w = 1.0$, fig. 2.3e), forming the extremes. For other values $w \in (-1, 1)$, the original is blended with its eroded (e.g., $w = -0.5$, fig. 2.3b) or dilated (e.g., $w = 0.5$, fig. 2.3d) counterpart.

(a) `Slant` lettering (b) $shear(0, \tan(\alpha))$ (c) $scale\left(\frac{1}{1+\tan(\alpha)^2}\right)$ (d) $rotate(\alpha)$

Figure 2.4: Decomposition of slanting transformation

maximizing the length of each line, which is being bounded by a desired pixel width. Such set of lines may already be sufficient output of the pipeline, however, we propose one more step enhancing the fidelity of generated samples.

Compositor

Final component improves each rendered line by adding visual context, i.e., partial protrusions of the glyphs of the preceeding and the succeeding line. Such transformation is motivated by phenomena common in manuscripts, where descenders and ascenders of consecutive lines bleed into each other in vertical axis. The component consumes up to three consecutive lines (the amount drawn randomly) and collates them vertically, based on their metrics (i.e., ascender, descender and x-height), creating randomized line spacing. Such composition is then randomly cropped, containing one whole line and parts of context lines if present (see figure 2.5).

2.2.2 Implementation

Not only we designed the hadwritten text letterings synthetization pipeline, we also provide its implementation as a part of this work [Procházka, 2021]. For that, we have chosen Python 3 programming language, which is commonly used in deep learning research, together with well known open source software libraries, namely FreeType [Rougier, 2018] for font processing, NumPy [Harris et al., 2020] and Pillow [Clark, 2021] for bitmap image processing.

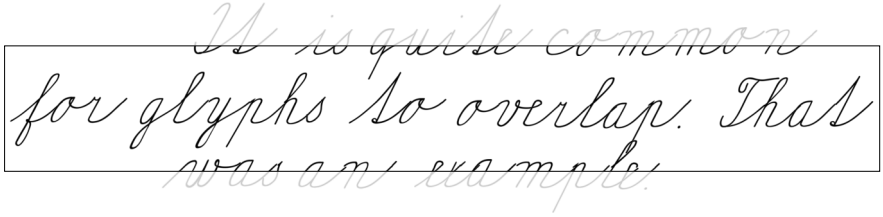


Figure 2.5: **Composed lines.** Three lines of text are collated together, with their ascenders and descenders potentially overlapping; the composition is then cropped to contain a single line and surrounding context.

Lettering Handling

Central to the synthetizer pipeline is the lettering data structure, containing a bitmap image of the lettering, its metrics, textual contents and metadata, e.g., positions of individual glyphs within the image. Having access to all the properties of the lettering at once, such data structure provides a convenient way to implement the affine transformations logic and the letterings concatenation.

When applying affine transformations, we make use of their matrix notation in homogeneous coordinate system. This simplifies both image transformation as well as adjustments to the accompanied metrics, using a single predefined Pillow function and a NumPy matrix multiplication, respectively. Apart from the ease of use, it alows for efficient composition of consecutive transformations, eliminating the need for the intermediate image transformations, greatly improving the time performance. Similarly, concatenation of multiple letterings, often single glyphs, benefits from composing multiple images to single one at once, rather than iteratively extending intermediate results.

Font Rendering

In order to create a lettering, we need to be able to obtain renders of glyphs as bitmap images. To do so, a wrapper around FreeType primitives is created, outputting the lettering data structure when querried for glyph of a particular size. The need for a wrapper comes mainly from the need to convert between units common to typography, i.e., *points* and *points-per-inch*, to units suitable for processing of bitmap images, i.e., *pixels*. Another benefit is the ability to use caching of previously rendered glyphs to speed up the querries.

Apart from these convenience features, the wrapper allows to level out certain inconsistencies between font definitions, and provide a unified interface. The main cause of inconsistencies we observe is handling of accented letters and the varying degree of their support in commonly available fonts. In fonts, where accented letters are not fully supported, some font metrics, e.g., height, ascender or advance, are ill-defined, potentially resulting in typesetting errors. The wrapper

allows to transparently handle the correction of the said metrics and fallback to base glyph, if the accented glyph is not available.

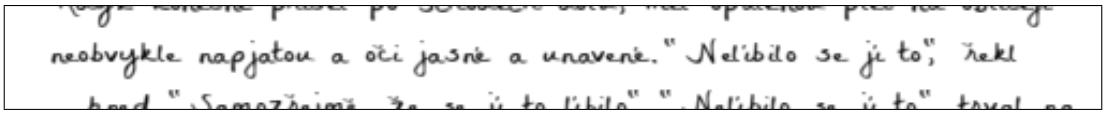
Lastly, the font wrapper turns out to be suitable abstraction for definition of handwriting inspired distortions of individual glyphs (see section 2.2.1), allowing to transparently handle randomized transformations of glyphs of the base font. In order to avoid aliasing when transforming images in rather low resolution, the transformations are performed on supersampled glyphs, which are downsampled using bilinear interpolation, after all transformations have been performed.

Pipeline

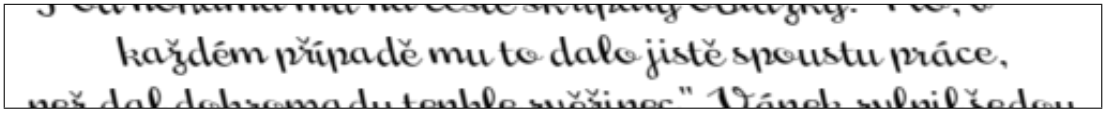
The implementation of the synthetizer pipeline follows its aforementioned design (section 2.2.1). As we are using the synthetizer to provide data to supervised training of deep neural network, benefitting from the diversity of the dataset, and exploiting the ability of the synthetizer to generate large quantity of unique samples without repetition. To do so, care needs to be taken to avoid large memory consumption, and to achieve sufficient speed of sample synthetization, in order not to stall the training process. As a result, we implement the individual components as lazy iterators, generating next sample, e.g., paragraph of text, font to be used or rendered lettering, only when querried by its consumer – succeeding pipeline component, being a lazy iterator itself. Together with speed optimizations mentioned earlier, without parallelizing our code, we manage to generate 10000 letterings of size $768 \times 48\,px$ and average length of 73 characters in $173\,s$, i.e., $17.3\,ms$ per sample on average.

2.2.3 Examples

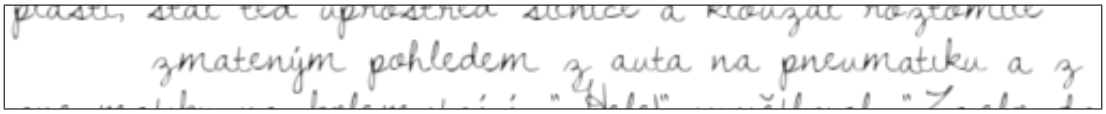
Figure 2.6 shows artificially generated letterings.



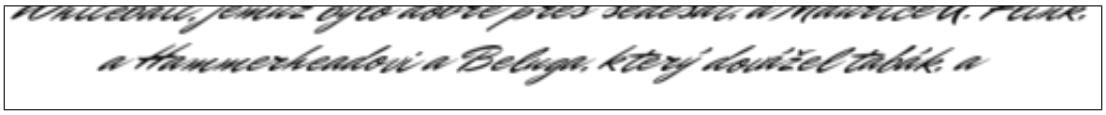
(a) neobvykle napjatou a oči jasné a unavené. "Nelíbilo se jí to," řekl



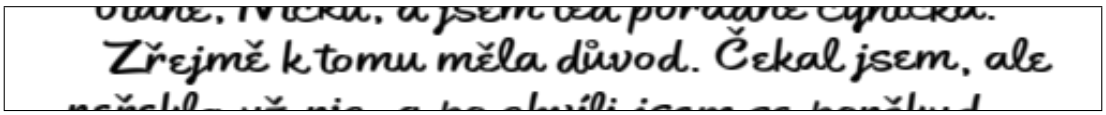
(b) každém případě mu to dalo jistě spoustu práce,



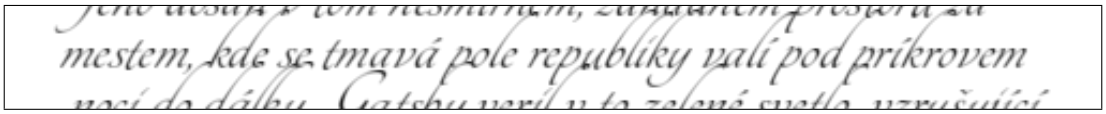
(c) zmateným pohledem z auta na pneumatiku a z



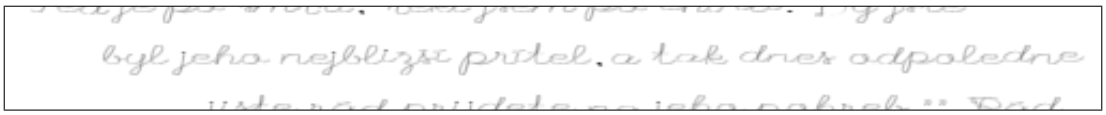
(d) a Hammerheadovi a Beluga, který dovážel tabák, a



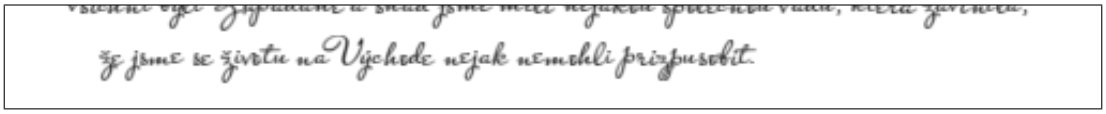
(e) Zřejmě k tomu měla důvod. Čekal jsem, ale



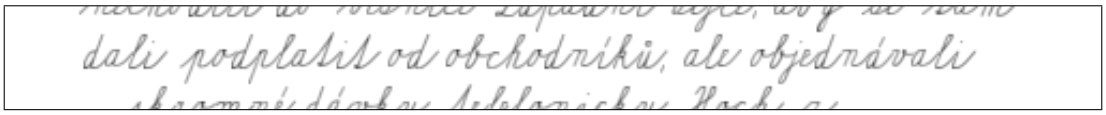
(f) mestem, kde se tmavá pole republiky valí pod príkrovem



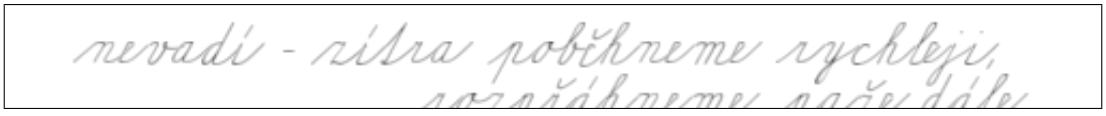
(g) byl jeho nejbližší prítel, a tak dnes odpoledne



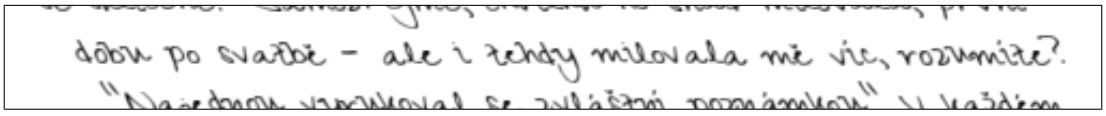
(h) že jsme se životu na Východe nejak nemohli prizpusobit.



(i) dali podplatit od obchodníků, ale objednávali



(j) nevadí - zítra poběhneme rychleji,



(k) dobu po svatbě - ale i tehdy milovala mě víc, rozumíte?

Figure 2.6: Examples of synthetized letterings

3. Our Solution

In this chapter, we propose our own handwritten text extractor, i.e., a chain of components extracting textual contents of an image of a single line of text. To assess its capabilities, we train the extractor using synthetic data (see section 3.2.3). The processing of the real data will be discussed in the next chapter 4.

3.1 Text Extractor

The chain comprises of deep neural network composed of convolutional, recurrent and dense layers and a CTC decoder, either greedy or beam-search based. Figure 3.1 shows the schema of the architecture, with each component described in the following sections.

3.1.1 Visual Features Extractor

As a visual features extractor, we opt for deep neural network with convolutional and recurrent layers loosely inspired by Shi et al. [2015] architecture. We improve upon the said architecture by using strided convolutions instead of max-pooling layers, adding batch normalization and dropout layers to support regularization and finally by adding residual connection around the last recurrent layer.

Figure 3.1 shows the exact configurations of neural network layers, grouped into *cnn*, *rnn* and *classifier* submodels. In case of convolutional layers, the abbreviations k, s and p are used to describe kernel shape, stride and padding, respectively. In the rnn submodel, we use bidirectional LSTM [Hochreiter and Schmidhuber, 1997] with 256 units, concatenating outputs of left-to-right and right-to-left passes at the corresponding timesteps. Finally, the dense layer in the classifier is used to perform classification at each timestep to classes $L' = L + \{\flat\}$, L being a predefined charset.

Regarding dropout, we use spatial dropout [Tompson et al., 2014] in between convolutional layers with rate 0.25, dropping whole channels of featuremaps randomly. We interleave recurrent layers with standard dropout, again with rate 0.25.

In total, the network has $5.5\,M$ trainable parameters and it is supposed to be trained using the CTC loss. The inputs to the network are expected to be greyscale images $[-1, 1]^{48\times w}$, with fixed height $h = 48$ and variable width w being an arbitrary multiple of 2. The outputs of the network are probability distributions over labels L' with CTC interpretation (section 1.2.2). Each output timestep corresponds to 2-pixel columns in input, expecting glyphs in input images to be at least 2 pixels wide for CTC to work properly.

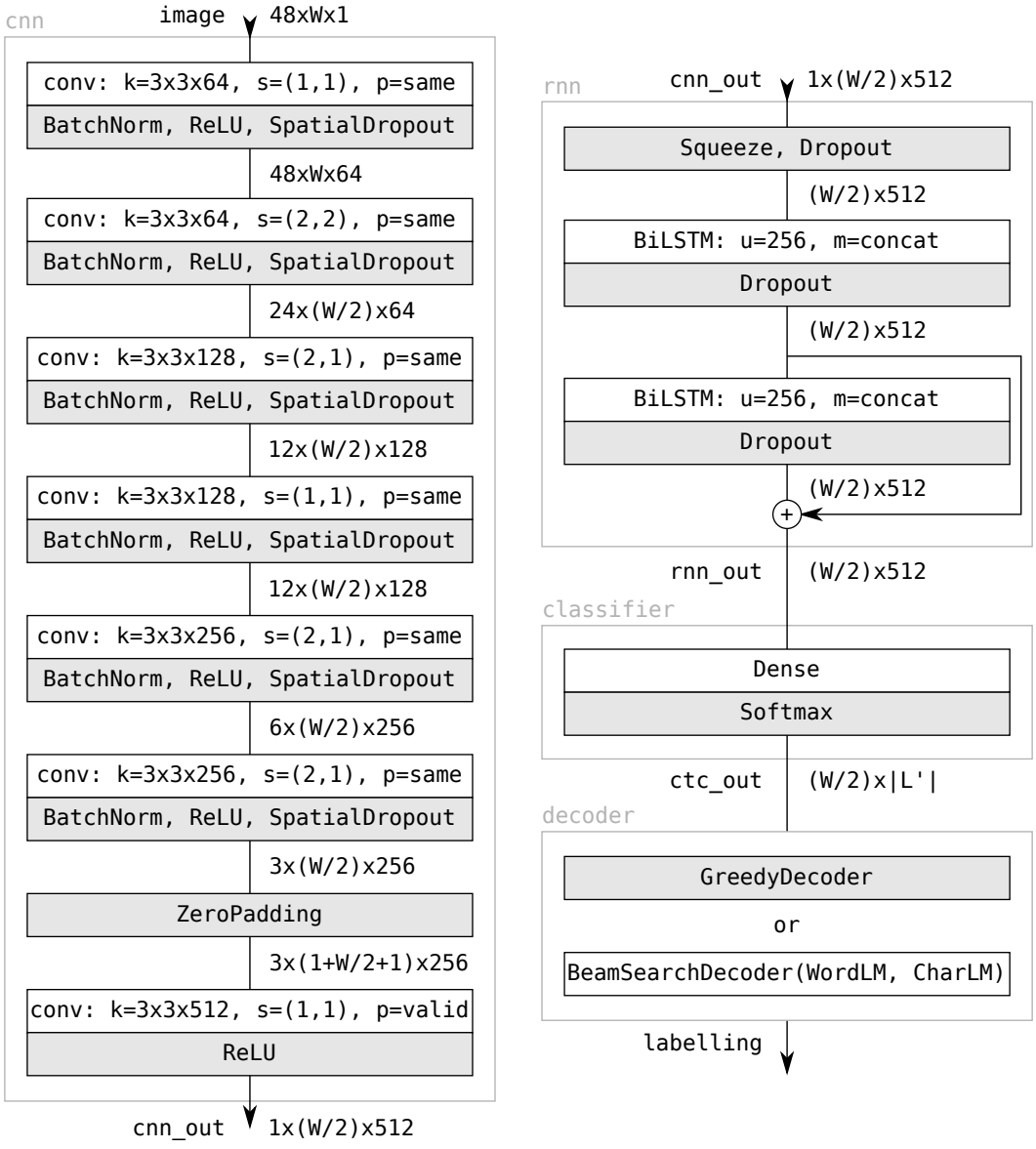


Figure 3.1: Text Extractor Architecture

3.1.2 CTC Decoder

In order to decode outputs of the text extractor, either greedy decoder (section 1.2.2) or beam-search decoder (section 1.2.2) can be used. We do not expect bare beam-search decoder to perform considerably better than a simpler greedy decoder, but we propose to integrate beam-search decoder with an external language model scorer.

While theoretically any well performing language model should improve the decoding performance, we opt to use n-gram language models, for their suitability to be integrated with a beam-search decoder. For the purposes of beam-search decoding, we seek a language model providing credible scores for all, even partial utterances. As word-level language model is principially unable to model partial words and as char-level model provides less reliable scores for word sequences compared to word-level model, we propose to use their combination, in order to utilize the benefits of both types of the models.

For a sequence of n characters $\boldsymbol{l} = (l_1, l_2, \ldots, l_n)$, we suggest multiple approaches to compute score $s(\boldsymbol{l})$. It is possible to simply compute a char-level score $s_c = p_c(\boldsymbol{l})$ and a word-level score $s_w = p_w(\boldsymbol{l})$ and their, optionally adjusted, combined score $s(\boldsymbol{l}) = s_w s_c^\alpha$. However, such approach relies on the value of the word-level probability $p_w(\boldsymbol{l})$, potentially skewed by unfinished word at the end of sequence $\boldsymbol{l}$. Assuming the positions of all K whole word begins $\boldsymbol{b} = (b_1, b_2, \ldots, b_K)$ and ends $\boldsymbol{e} = (e_1, e_2, \ldots, e_K)$ (exclusive) in sequence $\boldsymbol{l}$, we can improve upon former approach by computing $s_w = p_w(\boldsymbol{l}_{[1:e_K]})$. While this alleviates the issue of non-word suffix negatively influencing the word-level probability, it introduces an imbalance, stemming from double counting characters within whole words (both on word-level and char-level) as opposed to characters of non-word suffix (only char-level). Consequently, we propose to account for it by scoring the whole words with word-level model only, more formally:

$$s(\boldsymbol{l}) = \left(\frac{p_c(\boldsymbol{l})}{\prod_{k=1}^{K} p_c(\boldsymbol{l}_{[b_k:e_k]} | \boldsymbol{l}_{[1:b_k]})} \right)^\alpha p_w(\boldsymbol{l}_{[1:e_K]}) \,. \tag{3.1}$$

Such approach can be computed iteratively, accumulating the score of sequence that gets extended, by keeping track of char-level score of its non-word suffix, and swapping it for its word-level score, once it becomes a whole word. This approach can be seen as using char-level language model as a fallback model to estimate probability of partial words.

With the aforementioned changes, our decoder relies on two n-gram language models, one modelling character sequences, the other modelling word sequences. The decoder has in total four parameters for weighting the language model probabilities, namely the weight α of the char-level model, length compensating weights β_c and β_w, and finally the weight γ of the importance of the whole language model

scorer. Formally, the score s of the labelling $\boldsymbol{l}$ with K whole words and the total of C characters is

$$s(\boldsymbol{l}) = \left(\prod_{k=0}^{K} p_c(\boldsymbol{l}_{[e_k:b_{k+1}]} | \boldsymbol{l}_{[1:e_k]}) (b_{k+1} - e_k)^{\beta_c} \right)^{\alpha} p_w(\boldsymbol{l}_{[1:e_K]}) K^{\beta_w} \,, \tag{3.2}$$

setting $b_{K+1} = C$ and $e_0 = 1$ for convenience.

The extension score of labelling $\mathcal{S}(\boldsymbol{l}m) = \frac{s(\boldsymbol{l}m)}{s(\boldsymbol{l})}$ for the purpose of the beam search decoder implementation (see instruction 8 of the beam search algorithm 1) can be computed efficiently.

3.2 Implementation

In this section we experimentally assess the capabilities of the proposed approach using synthetic handwritten letterings. We also cover notable implementation details and parameters of the experiment.

3.2.1 Dataset

As a dataset, we use artificial letterings generated using the approach discussed in section 2.2. We source the texts randomly from either the corpus of written Czech [Křen et al., 2016] or corpus of written English [Graff et al., 2003]. To render the letterings, we use a manual selection of 32 handwriting-like fonts from Google Fonts [Google, 2021] listed in section A.1, with the addition of Abeceda font [Fila, 2004]. For distortions of fonts, we define base ranges for each distortion axis, from which we randomly draw the subranges for each instance of distorted font. The subranges cover at most 0.1 of the base range, with base ranges being $(-8, 8)$ degrees for rotation, $(0.5, 1.5)$ for horizontal scale, $(0.75, 1.25)$ for vertical scale, $(-45, 30)$ degrees for slant and $(-0.5, 0.5)$ for weight.

3.2.2 Extractor Implementation

The visual features extractor is implemented and trained using TensorFlow 2.

The beam search decoding is implemented [Procházka, 2021] as a Python extension module in C++ using PyBind [Jakob, 2021] for performance reasons, depending on KenLM [Heafield, 2011] implementation of n-gram language models. The char-level language models are 6-gram language models pruning singleton n-grams with the exception of unigrams. The word-level language models are 5-gram language models pruning n-grams occuring less then 8 times with the exception of unigrams, which are not being pruned.

Language	Decoder	Min	Median	Max
cs	greedy	0.83	2.11	14.28
	beam	0.23	0.99	8.85
en	greedy	1.31	4.48	16.74
	beam	0.29	0.74	8.09

Table 3.1: Performance of the model trained on synthetic data.

3.2.3 Supervised Training

The *cnn*, *rnn* and *classifier* components are trained jointly using the CTC loss (section 1.2.2) and Adam optimizer with recommended defaults [Kingma and Ba, 2014]. The training runs for 256 epochs with 128 gradient steps per epoch, each gradient step being estimated on a batch of 80 samples, drawn from Czech and English synthetic dataset. After the training is finished, the snapshot of model weights from the epoch with the lowest validation loss is selected. The training is independent on a *decoder* and its configuration.

To set the parameters α, β_c, β_w and γ of the beam search decoder, we carry out a grid search over the space of candidate configurations. The grid search minimizes the edit distance between predicted and true labellings, using formerly trained feature extractor applied to synthetic letterings in corresponding language, rendered in font unseen during the training phase. For Czech language, the grid search yields $\alpha = 0.8$, $\beta_c = 0.5$, $\beta_w = 14.0$ and $\gamma = 1.0$. For English, the parameters are $\alpha = 1.2$, $\beta_c = 0.5$, $\beta_w = 5.0$ and $\gamma = 1.0$.

The table 3.1 shows micro-averaged character-level edit distance, often called character error rate (CER), measured on the test set, i.e., letterings synthetized using a distinct set of fonts disjoint with fonts used in training. For each combination of source language and decoding method, the best, the worst and the median error rate among the test fonts is presented. It can be seen that on average, the model achieves satisfactory performance for both languages, namely with beam search decoding, keeping the median error rate under 1%.

4. Real Data Adaptation

Having implemented a handwritten text recognition system trained on artificially synthetized data (chapter 3), we want to assess its performance on the real handwritten letterings, and explore the possibilities of further accuracy improvements. In this chapter, we measure the accuracy of the system on several handwritten text datasets, and we carry out a multitude of transfer learning and self-training experiments, aiming to improve upon the base accuracy of the system. As a result, we propose a self-training strategy offering significant performance gains with no demand for ground truth annotations.

4.1 Preliminaries

Machine learning models, notably deep neural networks, together with their training procedures, are valued for their ability to generalize higher level concepts from training data, allowing to process unseen data with satisfactory performance. Moreover, those generic features prove to be useful in solving related tasks, when transfer learning techniques are employed. Those techniques allow to reuse parts of original pre-trained models and their parameters, reducing the complexity of training the model to solve the task at hand [Goodfellow et al., 2016].

4.1.1 Fine-tuning

Stemming from the iterative nature of the gradient descent based training procedures, one of the most straight-forward transfer learning techniques is fine-tuning.

In its simplest form, the task to be solved is structurally same as the task solved by the pre-trained base model, only the distribution of the actual data is shifted. In such case, it is sufficient to continue the training procedure of the base model with new data, effectively seeding the training procedure with better than random model parameters. Such training procedure often demands significantly lower amount of data as the concepts shared between the original and the actual task are already known to the model. In case of considerably smaller datasets, it is often beneficial to lower the learning rate of the stochastic gradient descent training procedure, or make a part of the original model parameters constant.

4.1.2 Self-training

When lacking sufficient amount of ground truth data for supervised learning, self-training techniques allow to adapt pre-trained base model to unnanotated corpora. The base model serves as a generator of noisy labels, which, when

optionally cleaned-up or pruned, may be used for its fine-tuning or training of a new model. Self-training techniques are currently subject to ongoing research, forming specialized branches, e.g., contrastive learning, clustering learning or self-supervised learning.

In the domain of OCR, an independent concurrent work of Kiss et al. [2021] explore self-training on both printed and handwritten datasets with focus on data augmentation.

4.2 Datasets

AHMP

Czech handwritten letterings are sourced from Prague City Archives [AHMP, 2021] collections of manuscripts, which contain vast amount of scanned municipal and school chronicles, civil registers, etc. As the data lack any kind of annotation, neither text segmentation nor textual contents, we employ our own simplistic preprocessing steps and we manually annotate two experimental datasets, with the focus on chronicles for their suitable layout and substantial amount of contained text.

The preprocessing concerns mainly text line detection using signal processing techniques, relying on simple layout of chronicles, being block of wrapped paragraphs of text. First, the raw image is adaptively thresholded to suppress the background noise and trimmed to remove excess horizontal margins. Second, the vertical coordinates of individual lines are searched for by finding peaks in counts of background to foreground transitions in each row of image pixels (see figure 4.1). Once the vertical line positions are computed, they delimit each line from top and bottom using the neighboring line positions. It is no coincidence that we designed our data synthetizer (section 2.2) to produce similar artificial letterings with parts of adjacent lines.

This way we obtain text lines from two manuscripts, namely the Přibyl [1933] school chronicles and Bruder et al. [1955] municipal chronicles, which we refer to as *Strasnice* and *Reporyje* datasets, respectively. The Strasnice dataset contains 6531 automatically detected lines, the Reporyje dataset contains 17913 lines. For both datasets we provide manual annotations of first 1024 valid lines (see figure 4.2 for a sample line). We also define random training and test splits with 640 and 384 samples, respectively. Both datasets are made publicly available as Procházka and Straka [2021].

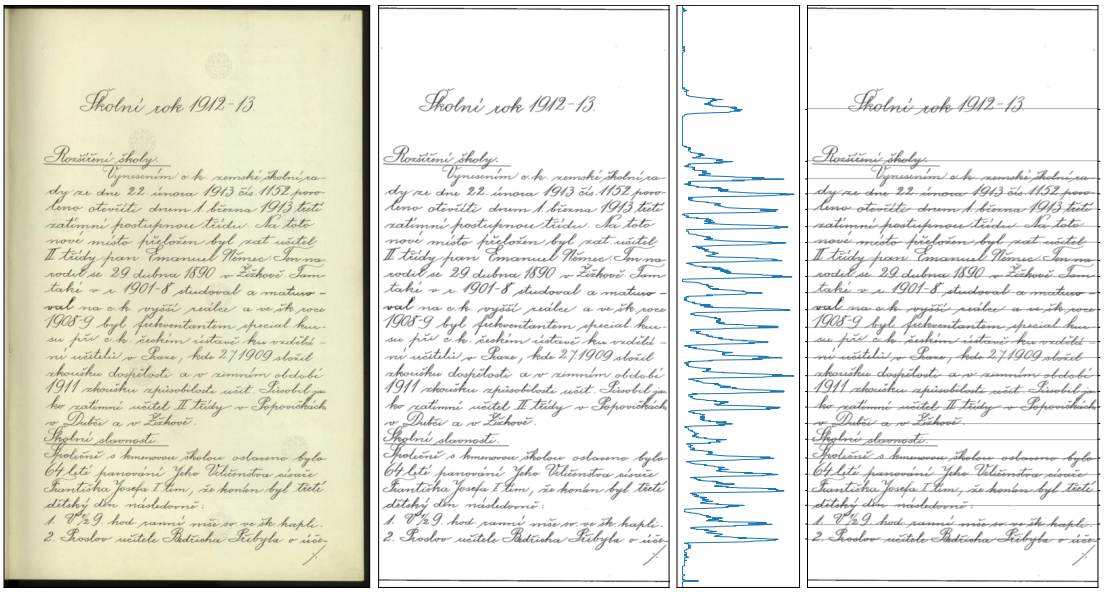
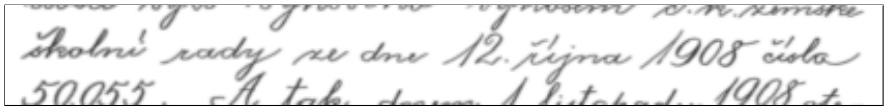
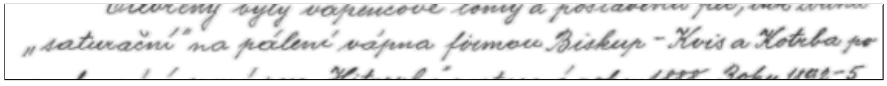


Figure 4.1: **Line detection.** The original image is thresholded and horizontally trimmed, then the line positions are searched for using signal peak detection techniques.

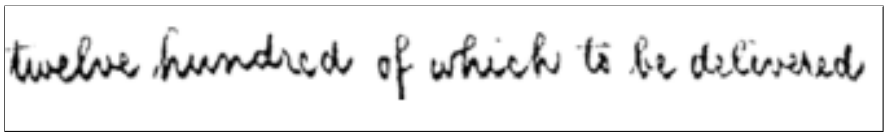


(a) `školní rady ze dne 12. října 1908 čísla`



(b) `"saturační" na pálení vápna firmou Biskup-Kvis a Kotrba po`

Figure 4.2: Samples from the Strasnice and Reporyje datasets.



`twelve hundred of which to be delivered`

Figure 4.3: Sample line from the Washington dataset.

Washington Database

In order to provide results allowing comparison with previous works, we also measure the performance of the system using well established Fischer et al. [2012] English handwritten dataset. The dataset contains 656 text manually segmented and annotated text lines (see figure 4.3 for a sample line).

Several previous works targeted Washington dataset by the means of supervised fine-tuning of a pre-trained network. Notably, Aradillas et al. [2018, 2020b,a] managed to achieve 5.3% character error rate using greedy decoding.

Few preprocessing steps are applied to the original data in order to ensure compatibility with our system. The images are uniformly scaled to height of 28 pixels and padded with 8 and 12 pixels from top and bottom, respectively, to achieve expected fixed height of 48 pixels, with a centered line and vertical margins.

4.3 Experiments

We carry out an extensive series of experiments in order to observe performance gains of fine-tuning and self-training under various conditions, e.g., varying size of training set, portion of model weights to be fine-tuned, training set selection criteria, etc. Apart from supervised fine tuning, we experiment with self-training, using the model itself to annotate the data.

4.3.1 Baseline

In order to assess the possible gains of fine-tuning strategies, we first measure the performance of raw, i.e. synthetically pretrained, model from chapter 3. We measure micro mean normalized edit distance, i.e. overall percentage of spelling errors in the resulting labellings, using predefined test splits of the datasets and available decoders.

Having established the measurement of the raw performance of the system, we measure the accuracy after supervised fine-tuning on training splits of available datasets. For that we use learning rate of 0.0001, 2048 gradient steps and batch size 8. We try both training only the last layer (i.e., the classifier) with the rest of the weights frozen and training the whole network, with results marked $\text{Tuned}_{\text{Last}}$ and $\text{Tuned}_{\text{Whole}}$ respectively, organized in table 4.1.

It can be seen that fine-tuning the whole network is consistently better than fine-tuning only the last layer, with beam search decoding providing better predictions in all cases compared to greedy decoding. Regarding the Washington dataset, we outperform previous works of Aradillas et al. [2018, 2020b,a], improving the original 5.3% error rate to 4.0%.

Dataset	Decoder	Raw	$\text{Tuned}_{\text{Last}}$	$\text{Tuned}_{\text{Whole}}$	Previous
Reporyje	greedy	34.66	21.99	3.38	
	beam	27.16	15.28	2.53	
Strasnice	greedy	28.34	16.99	3.34	
	beam	22.92	12.09	3.11	
Washington	greedy	33.26	18.21	4.01	*5.3*
	beam	20.53	11.49	3.66	

Table 4.1: Supervised fine-tuning.

Comparing the raw performance and best fine-tuned performance, it can be seen that even with a small dataset and minimalistic fine-tuning procedure, the performance can be substantially improved. This motivates us to proceed with studying self-training capabilities.

4.3.2 Self-Training

In this section we explore various self-training strategies. To reduce computational demands, we restrict the experiments to the Strasnice dataset. Moreover, we focus on fine-tuning of the whole network as it proves to yield significantly better results in supervised scenario. The batch size and learning rate is kept the same, and all presented error rates are again micro averages of edit distance over the test split.

Baseline

In order to assess the impact and the importance of various parameters of self-training strategies, we carry out a series of experiments with some degree of ground truth knowledge.

Notably, we explore the impact of the dataset size and the number of gradient steps of the fine-tuning, together with the accuracy drop caused by use the of noisy targets. We establish a baseline using randomly sampled train sets from ground truth data, referred to as *true*. Next, we use the base model and beam search decoder to suggest noisy labels, and we select training subsets minimizing the label noise. This is done by sorting the samples based on normalized edit distance to true labels and taking their portion of a given size. We refer to those data as *ned*.

Table 4.2 shows resulting performance after fine-tuning the model for a given amount of gradient steps and a given decoder. In case of *true* data, the performance improves with larger dataset and longer training as expected, yielding 3.02% character error rate with dataset of size 512 after 2048 gradient steps using

Dataset		Size					
Decoder	Steps	32	64	128	256	512	Best
true		0.00	0.00	0.00	0.00	0.00	
greedy	1024	7.21	6.40	5.37	4.81	4.27	4.27
	2048	7.18	6.09	4.88	3.96	3.41	3.41
beam	1024	5.00	4.79	4.06	3.64	3.43	3.43
	2048	5.06	4.50	3.62	3.36	3.02	3.02
	best	5.00	4.50	3.62	3.36	3.02	3.02
ned		0.00	0.80	2.94	7.10	16.29	
greedy	1024	11.87	10.71	10.42	12.41	15.30	10.42
	2048	11.22	10.17	9.35	12.23	15.32	9.35
beam	1024	8.38	7.90	8.35	10.99	14.42	7.90
	2048	7.95	7.43	7.37	10.65	14.62	7.37
	best	7.95	7.43	7.37	10.65	14.42	7.37

Table 4.2: Self-training baseline experiments.

the beam search decoder. In case of *ned* data, the performance improves with longer training, but the training does not benefit from the largest dataset, caused by a trade-off in target labels impurity. The best reached performance is character error rate of 7.37% achieved with dataset of size 128 and 2.94% label noise (see table 4.2). Overall, the results on *ned* data show substantial improvements of model performance, being 2-3 times lower than the raw 22.92% error rate.

Sorting Criterion

Noting that *ned* experiments still rely on knowledge of true labels needed to compute the edit distance to noisy labels, we need to devise an unsupervised sorting criterion, in order to be able to self-train the system. To this end, we measure correlation of normalized edit distance with various hand-crafted features derived from model predictions. We focus on CTC decoder and language model scores, optionally length normalized. Namely, we try *score* (raw decoder score), *cscore* and *wscore* (*score* normalized by character length and word length, respectively), and finally *clm score* and *wlm score*, which are language model scores of the predicted labelling. Figure 4.4 shows correlation matrices of the aforementioned metrics, with *cscore* clearly predicting *ned* the best. It should be noted that as we are interested solely in ordering the predicted samples, the actual magnitude of the metric is not important.

Choosing the *cscore* sorting criterion, we are able to measure self-training performance of the system. In the following table we show test set character level error rates, when using predicted labels as fine-tuning targets, relying on

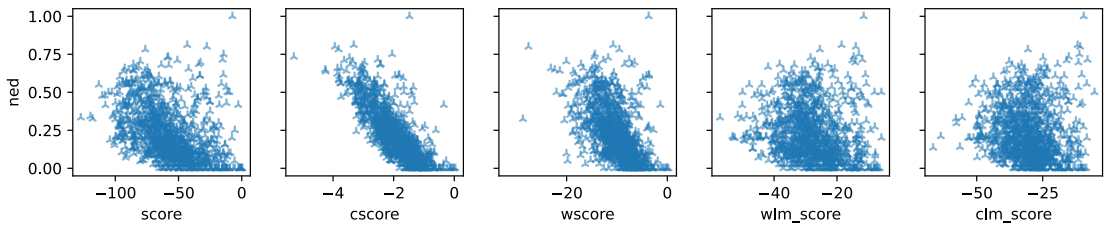


Figure 4.4: Correlation of normalized edit distance with various features derived from model predictions.

Dataset		Size					
Decoder	Steps	32	64	128	256	512	Best
cscore		4.11	3.59	6.20	9.70	16.94	3.59
greedy	1024	14.47	12.70	12.98	12.78	15.03	12.70
	2048	13.68	11.79	12.27	12.49	15.35	11.79
beam	1024	11.17	9.77	10.56	11.18	13.96	9.77
	2048	10.17	9.15	10.15	10.66	14.89	9.15
	best	10.17	9.15	10.15	10.66	13.96	9.15

Table 4.3: Self-training based on *cscore* criterion.

cscore criterion to choose the suitable training samples. Again, the fine-tuning generally benefits from more gradient steps and trades-off dataset size for label purity, achieving the lowest error rate 9.15% using 64 samples with the lowest label noise, i.e., 3.59% (see table 4.3). It should be noted that such improvement is achieved with no knowledge of ground truth data.

Iteration

In the previous sections, we established the means to fine-tune the base model in an unsupervised manner, improving upon its original performance. In this section we explore the capabilities of iterating such approach, using the fine-tuned model as a better predictor of noisy targets.

First, we run an experiment estimating the limits of the iterated approach. In each iteration, we fine-tune the model using training sets of various sizes, and we select the fine-tuned model with the lowest error rate on the test set. This model serves as a predictor of the noisy targets for the next iteration. Note that the network fine-tuning in each iteration starts with the original raw weights. We experiment with decoders used in the process, selecting either the best one each iteration, restricting the whole iteration to single decoder or alternating between beam search decoder and greedy decoder in each iteration. In table 4.4 and figure 4.5 we can see the progression of performance of each strategy throughout 8 iterations. Surprisingly, the best results are not achieved by the strategy selecting

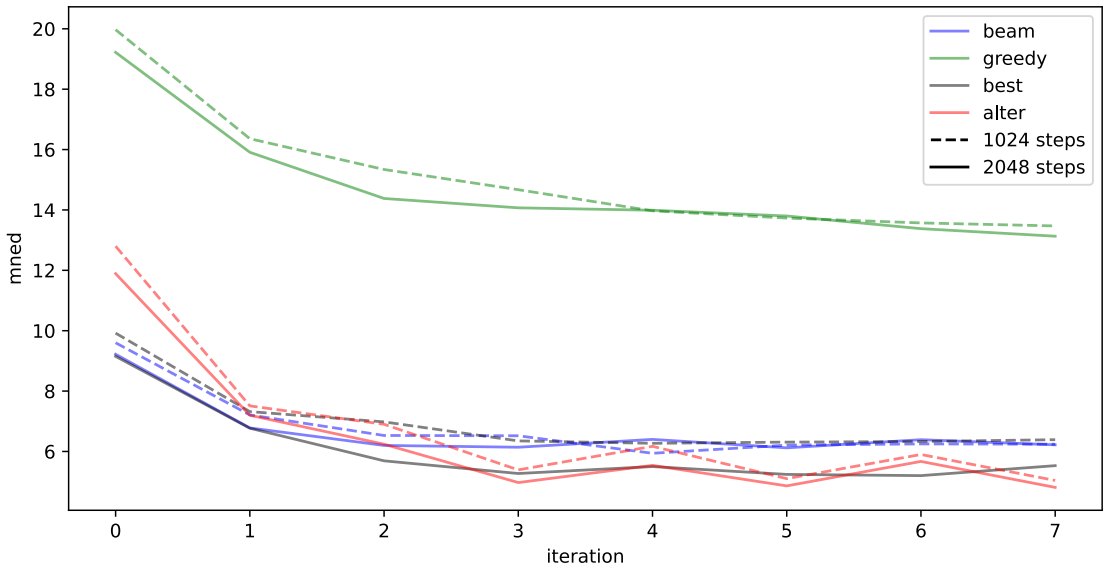


Figure 4.5: Comparison of self-training strategies.

		Iteration								
Decoder	Steps	0	1	2	3	4	5	6	7	Best
best	1024	9.92	7.32	6.98	6.35	6.27	6.31	6.33	6.39	6.27
	2048	9.15	6.77	5.69	5.27	5.50	5.24	5.20	5.53	5.20
greedy	1024	19.97	16.36	15.34	14.67	13.97	13.73	13.57	13.47	13.47
	2048	19.22	15.91	14.38	14.07	13.99	13.80	13.38	13.13	13.13
beam	1024	9.60	7.21	6.53	6.52	5.94	6.21	6.25	6.24	5.94
	2048	9.22	6.78	6.20	6.14	6.40	6.12	6.39	6.22	6.12
alter	1024	12.80	7.51	6.90	5.39	6.17	5.10	5.90	5.04	5.04
	2048	11.89	7.20	6.24	4.97	5.54	4.86	5.67	4.81	4.81
	best	9.15	6.77	5.69	4.97	5.50	4.86	5.20	4.81	4.81

Table 4.4: Self-training strategies comparison.

more performant decoder in each iteration, but with strategy that alternates the decoders, achieving 4.81% error rate. We assume the alternating strategy to exhibit some degree of regularization, preventing to converge prematurely to a sub-optimal solution.

We proceed with experiments not relying on the knowledge of performance of the candidate models. We do so by running 8 self-training iterations with a fixed dataset size. The results are presented in table 4.5 and figure 4.6. Consistent with the previous experiments, the alternating strategy outperforms the usage of a single decoder, resulting in 5.62% character error rate. Figure 4.6 visualizes the progress of self-training, showing the overall superiority of alternated strategy, together with the benefits of using medium sized datasets.

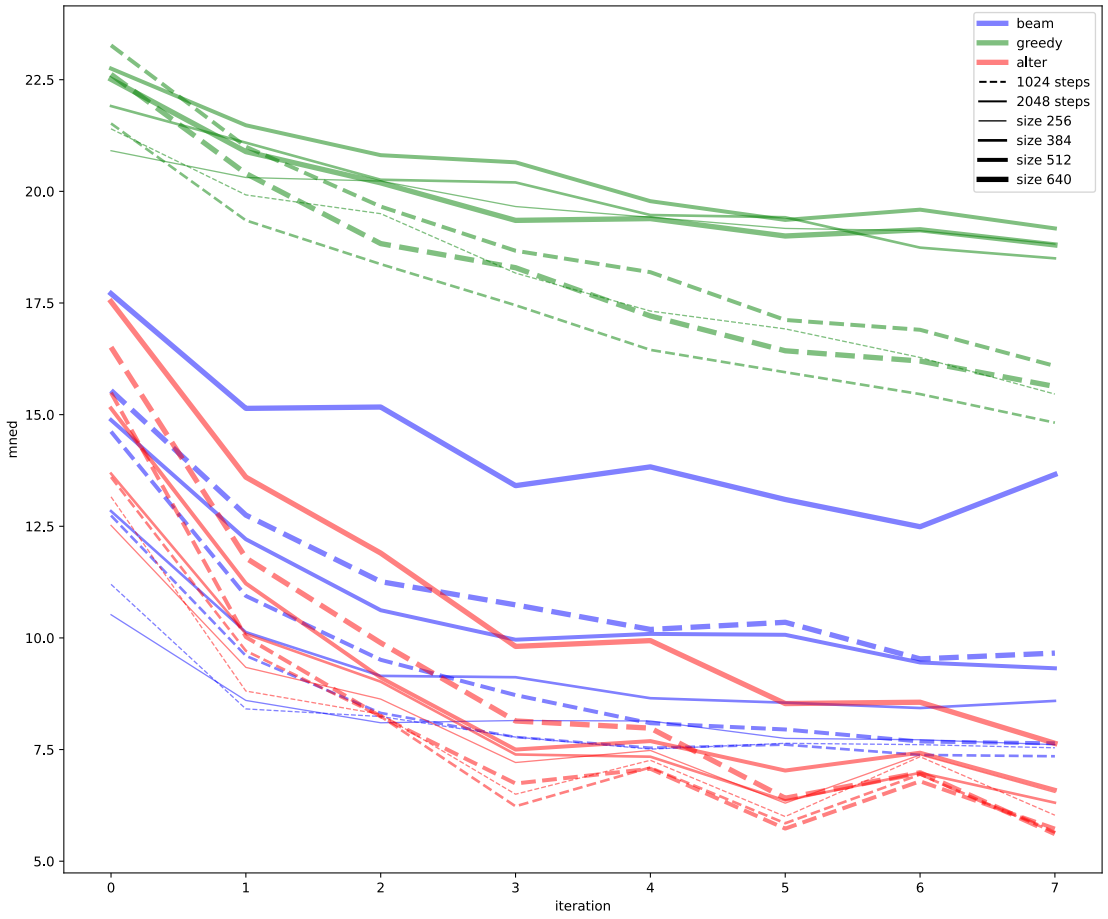


Figure 4.6: Comparison of *cscore* strategies.

Mode		Iteration								
Steps	Size	0	1	2	3	4	5	6	7	Best
greedy										
1024	256	21.40	19.92	19.50	18.17	17.32	16.92	16.28	15.46	15.46
	384	21.52	19.35	18.37	17.45	16.45	15.95	15.46	14.82	14.82
	512	23.27	21.00	19.66	18.67	18.19	17.12	16.90	16.09	16.09
	640	22.62	20.40	18.83	18.29	17.21	16.43	16.20	15.63	15.63
2048	256	20.91	20.31	20.23	19.66	19.42	19.17	19.12	18.82	18.82
	384	21.91	21.09	20.26	20.20	19.47	19.42	18.74	18.50	18.50
	512	22.75	21.48	20.81	20.65	19.78	19.36	19.59	19.17	19.17
	640	22.51	20.89	20.19	19.35	19.39	19.00	19.14	18.80	18.80
	best	20.91	19.35	18.37	17.45	16.45	15.95	15.46	14.82	14.82
beam										
1024	256	11.20	8.41	8.24	7.78	7.51	7.64	7.61	7.54	7.51
	384	12.74	9.60	8.32	7.78	7.54	7.61	7.38	7.35	7.35
	512	14.62	10.94	9.51	8.72	8.09	7.95	7.68	7.64	7.64
	640	15.54	12.75	11.26	10.74	10.19	10.35	9.53	9.66	9.53
2048	256	10.52	8.60	8.10	8.15	8.14	7.75	7.72	7.61	7.61
	384	12.84	10.13	9.15	9.12	8.65	8.55	8.43	8.59	8.43
	512	14.88	12.21	10.62	9.96	10.09	10.07	9.45	9.32	9.32
	640	17.71	15.14	15.17	13.41	13.83	13.10	12.49	13.66	12.49
	best	10.52	8.41	8.10	7.78	7.51	7.61	7.38	7.35	7.35
alter										
1024	256	13.16	8.81	8.29	6.50	7.26	6.00	7.34	6.03	6.00
	384	13.60	9.71	8.24	6.23	7.10	5.85	6.94	5.64	5.64
	512	15.49	10.02	8.24	6.74	7.07	5.73	6.79	5.73	5.73
	640	16.51	11.79	9.89	8.14	7.98	6.41	6.98	5.62	5.62
2048	256	12.52	9.34	8.63	7.21	7.48	6.30	7.38	6.54	6.30
	384	13.68	10.09	9.02	7.39	7.34	6.36	6.98	6.31	6.31
	512	15.14	11.22	9.11	7.50	7.69	7.03	7.43	6.60	6.60
	640	17.53	13.60	11.90	9.81	9.94	8.53	8.56	7.64	7.64
	best	12.52	8.81	8.24	6.23	7.07	5.73	6.79	5.62	5.62
		10.52	8.41	8.10	6.23	7.07	5.73	6.79	5.62	5.62

Table 4.5: Self-training iteration.

Dataset	Decoder	Model				
		Reporyje	Strasnice	Washington	Raw	Tuned
Reporyje	greedy	**7.94**	11.45	43.24	34.66	3.38
	beam	**5.29**	7.86	38.93	27.16	2.53
Strasnice	greedy	11.80	**5.70**	34.42	28.34	3.34
	beam	8.31	**4.84**	30.73	22.92	3.11
Washington	greedy	49.59	43.42	**6.13**	33.26	4.01
	beam	46.06	33.45	**5.49**	20.53	3.66

Table 4.6: **Self-training comparison.** Columns contain error rates of models self-trained on datasets corresponding to column names, with the exception of *Raw* and *Tuned* columns provided for faster comparison with the base model and supervised fine-tuning, respectively. Bold values highlight cases with matching self-training and testing data.

4.4 Results

Based on the experiments seeking appropriate self-training strategies, we process all three available datasets. With no need for ground truth annotation, we use whole datasets (test sets without labels included). We run 16 iterations with alternating decoders, selecting half of available samples (sorted by *cscore*) as the training dataset in each iteration.

Table 4.6 provides comparison between micro average of edit distance between raw model trained solely on synthetized letterings, model fine-tuned using ground truth annotations and iteratively self-trained models. We succeeded in adapting the model to all datasets by the means of self-training, reaching error rates competitive with the supervised fine-tuning scenario.

We also measure knowledge transfer of self-training across datasets, showing positive results for Reporyje and Strasnice datasets, most likely due to both sharing the same language and, to some degree, handwriting style. On the contrary, Washington dataset and its corresponding model does not transfer knowledge to other datasets.

When observing the resulting predictions in detail, substantial amount of remaining errors forms three distinct categories, all stemming from properties of n-gram based language modelling. Firstly, the digits, mainly dates and counts, are often mispredicted. Secondly, partial words at the beginning and end of text lines caused by word splitting are mangled or sometimes even missing. Lastly, the model does not predict well the abbreviations and unique proper nouns unknown to the language model. Future work may improve the language model based beam search decoding to handle digits as a single category, modelling rather the presence of any digit in general than an exact numeric value. Apart from that,

it may be beneficial to explore the means to use context of neighbouring lines in the decoding process.

Conclusion

In this work we focused on the task of handwritten text recognition, with the emphasis on decreasing the required number of ground truth annotations.

To fulfil our goal, we proposed and implemented an artificial letterings synthetizer, taking advantage of publicly available decorative fonts resembling handwriting, coupled with suitable set of randomized visual distortions on a glyph level. This enabled us to generate large amount of data with known textual contents, close to real handwritings in terms of complexity and appearance.

Securing access to large amount of data suitable for supervised learning, we designed and trained a deep convolutional and recurrent neural network, making use of connectionist temporal classification framework and language model based decoding techniques.

With model pre-trained on artificial data we observed a degree of knowledge transfer to the domain of real handwritten letterings. We carried out a multitude of experiments adapting the base model to handwriting style of particular manuscripts by the means of fine-tuning and self-training, allowing us to formulate an unsupervised self-training strategy yielding satisfactory low error rates.

Partly as a mean to verify our findings and partly as their direct result, we managed to produce a novel dataset of Czech handwritten letterings, which we make publicly available.

Stemming from the modular nature of the design of our solution, we see numerous opportunities for further improvements in future work. Regarding text detection, it may be beneficial to make use of the synthetic data generator, providing detailed positional information about artificial letterings up to the glyph level. In the domain of CTC decoding, we see potential in exploring a context-aware beam search algorithm, making use of predictions of surrounding lines. Regarding the iterated self-training, dynamics of noisy labels evolution may be studied in greater detail to improve training dataset selection.

Overall, the techniques proposed and studied in this work may be suitable to process large collections of manuscripts, accumulating the knowledge of various handwriting styles with minimal need for human intervention and providing means to bootstrap the annotation process. Such application may greatly improve the accessibility of those collections in machine-readable form.

Bibliography

AHMP. Prague city archives, 2021. URL http://katalog.ahmp.cz/pragapublica/.

José Carlos Aradillas, Juan José Murillo-Fuentes, and Pablo M. Olmos. Boosting handwriting text recognition in small databases with transfer learning. *CoRR*, abs/1804.01527, 2018. URL http://arxiv.org/abs/1804.01527.

José Carlos Aradillas, Juan José Murillo-Fuentes, and Pablo M. Olmos. Boosting offline handwritten text recognition in historical documents with few labeled lines. *CoRR*, abs/2012.02544, 2020a. URL https://arxiv.org/abs/2012.02544.

José Carlos Aradillas, Juan José Murillo-Fuentes, and Pablo M. Olmos. Improving offline htr in small datasets by purging unreliable labels. In *2020 17th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 25–30, 2020b. doi: 10.1109/ICFHR2020.2020.00016.

Vincenc Blahouš. *Methodika krasopisu*. Alois Wiesner, Praha, 1902. URL https://cdk.lib.cas.cz/uuid/uuid:d7f255c8-9091-43c1-a3fb-d2d6e0ab494c.

Rudolf Bruder, Štěpánka Krotovičová, Karel Peterka, and Eliška Lišková. Kronika obce Řeporyje, 1. díl, 1955. URL http://katalog.ahmp.cz/pragapublica/permalink?xid=2A3D50BD9C7411E8A0AE00505694BE33.

Alex Clark. Pillow (pil fork) documentation, 2021. URL https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf.

Stanislav Fila. Abeceda, 2004. URL http://www.cpppap.svsbb.sk/files/im_font_abc.html.

Andreas Fischer, Andreas Keller, Volkmar Frinken, and Horst Bunke. Lexicon-free handwritten word spotting using character hmms. *Pattern Recognition Letters*, 33(7):934–942, 2012. ISSN 0167-8655. doi: https://doi.org/10.1016/j.patrec.2011.09.009. URL https://www.sciencedirect.com/science/article/pii/S0167865511002820. Special Issue on Awards from ICPR 2010.

G.D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973. doi: 10.1109/PROC.1973.9030.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Google. Google fonts, 2021. URL https://fonts.google.com/?category=Handwriting.

David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English gigaword. *Linguistic Data Consortium, Philadelphia*, 4(1):34, 2003.

Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-24797-2. doi: 10.1007/978-3-642-24797-2_2. URL https://doi.org/10.1007/978-3-642-24797-2_2.

Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi: 10.1038/s41586-020-2649-2.

Kenneth Heafield. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland, July 2011. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/W11-2123.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Wenzel Jakob. pybind11 documentation, 2021. URL https://pybind11.readthedocs.io/_/downloads/en/latest/pdf/.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

Martin Kiss, Karel Benes, and Michal Hradis. AT-ST: self-training adaptation strategy for OCR in domains with limited transcriptions. *CoRR*, abs/2104.13037, 2021. URL https://arxiv.org/abs/2104.13037.

Michal Křen, Václav Cvrček, Tomáš Čapka, Anna Čermáková, Milena Hnátková, Lucie Chlumská, Tomáš Jelínek, Dominika Kováříková, Vladimír Petkevič, Pavel Procházka, Hana Skoumalová, Michal Škrabal, Petr Truneček, Pavel Vondřička, and Adrian Zasina. SYN v4: large corpus of written czech, 2016.

URL http://hdl.handle.net/11234/1-1846. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Minghui Liao, Baoguang Shi, and Xiang Bai. Textboxes++: A single-shot oriented scene text detector. *CoRR*, abs/1801.02765, 2018. URL http://arxiv.org/abs/1801.02765.

Jamshed Memon, Maira Sami, and Rizwan Ahmed Khan. Handwritten optical character recognition (OCR): A comprehensive systematic literature review (SLR). *CoRR*, abs/2001.00139, 2020. URL http://arxiv.org/abs/2001.00139.

Hermann Ney, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech and Language*, 8(1):1–38, 1994. ISSN 0885-2308. doi: https://doi.org/10.1006/csla.1994.1001. URL https://www.sciencedirect.com/science/article/pii/S0885230884710011.

Štěpán Procházka. Adaptive handwritten text recognition, 2021. URL https://gitlab.com/proste/ahtr.

Štěpán Procházka and Milan Straka. HaCzech: Dataset of Handwritten Czech, 2021. URL http://hdl.handle.net/11234/1-3739. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Bedřich Přibyl. Pamětní kniha druhé obecné školy ve starých strašnicích, 1933. URL http://katalog.ahmp.cz/pragapublica/permalink?xid=1A4E05EC3E5A11E4AD89002185109406.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019. URL http://arxiv.org/abs/1910.10683.

Nicolas P. Rougier. Freetype python documentation, 2018. URL https://freetype-py.readthedocs.io/_/downloads/en/latest/pdf/.

Baoguang Shi, Xiang Bai, and Cong Yao. An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *CoRR*, abs/1507.05717, 2015. URL http://arxiv.org/abs/1507.05717.

Ray Smith. An overview of the tesseract ocr engine. In *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*, pages 629–633, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2822-8. URL `http://www.google.de/research/pubs/archive/33418.pdf`.

Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. *CoRR*, abs/1411.4280, 2014. URL `http://arxiv.org/abs/1411.4280`.

Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. EAST: an efficient and accurate scene text detector. *CoRR*, abs/1704.03155, 2017. URL `http://arxiv.org/abs/1704.03155`.

List of Figures

List of Tables

A. Appendix

A.1 Handwritten-like Fonts

List of Google fonts used by artificial letterings synthetizer. The paths are relative to common prefix `https://github.com/google/fonts/raw/main/`.

- `ofl/aguafinascript/AguafinaScript-Regular.ttf`
- `ofl/alexbrush/AlexBrush-Regular.ttf`
- `ofl/allura/Allura-Regular.ttf`
- `ofl/arizonia/Arizonia-Regular.ttf`
- `ofl/bilbo/Bilbo-Regular.ttf`
- `apache/calligraffitti/Calligraffitti-Regular.ttf`
- `ofl/cedarvillecursive/Cedarville-Cursive.ttf`
- `ofl/clickerscript/ClickerScript-Regular.ttf`
- `ofl/cookie/Cookie-Regular.ttf`
- `ofl/damion/Damion-Regular.ttf`
- `ofl/dawningofanewday/DawningofaNewDay.ttf`
- `ofl/euphoriascript/EuphoriaScript-Regular.ttf`
- `ofl/greatvibes/GreatVibes-Regular.ttf`
- `ofl/herrvonmuellerhoff/HerrVonMuellerhoff-Regular.ttf`
- `apache/homemadeapple/HomemadeApple-Regular.ttf`
- `ofl/italianno/Italianno-Regular.ttf`
- `ofl/kristi/Kristi-Regular.ttf`
- `ofl/labelleaurore/LaBelleAurore.ttf`
- `ofl/leaguescript/LeagueScript-Regular.ttf`
- `ofl/leckerlione/LeckerliOne-Regular.ttf`
- `ofl/marckscript/MarckScript-Regular.ttf`

- ofl/meddon/Meddon.ttf
- ofl/monsieurladoulaise/MonsieurLaDoulaise-Regular.ttf
- apache/montez/Montez-Regular.ttf
- ofl/mrdafoe/MrDafoe-Regular.ttf
- ofl/mrdehaviland/MrDeHaviland-Regular.ttf
- ofl/mrssaintdelafield/MrsSaintDelafield-Regular.ttf
- ofl/niconne/Niconne-Regular.ttf
- ofl/norican/Norican-Regular.ttf
- ofl/pacifico/Pacifico-Regular.ttf
- ofl/parisienne/Parisienne-Regular.ttf
- ofl/petitformalscript/PetitFormalScript-Regular.ttf
- ofl/pinyonscript/PinyonScript-Regular.ttf
- ofl/qwigley/Qwigley-Regular.ttf
- ofl/reeniebeanie/ReenieBeanie.ttf
- apache/rochester/Rochester-Regular.ttf
- ofl/rougescript/RougeScript-Regular.ttf
- ofl/sacramento/Sacramento-Regular.ttf
- apache/satisfy/Satisfy-Regular.ttf
- ofl/tangerine/Tangerine-Regular.ttf
- ofl/vibur/Vibur-Regular.ttf
- apache/yellowtail/Yellowtail-Regular.ttf
- ofl/yesteryear/Yesteryear-Regular.ttf
- ofl/zeyada/Zeyada.ttf