HANDWRITTEN TEXT RECOGNITION USING NEURAL NETWORK

A MINI PROJECT REPORT BY

Submitted by

SAI KRISHNA R
SAKETH RAMAN R
SURYA DHANUSH G

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

EASWARI ENGINEERING COLLEGE, CHENNAI

(Autonomous Institution)

affiliated to

ANNA UNIVERSITY:: CHENNAI - 600025

JULY 2021

EASWARI ENGINEERING COLLEGE, CHENNAI

(AUTONOMOUS INSTITUTION)

AFFILIATED TO ANNA UNIVERSITY, CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this project report **"HANDWRITTEN TEXT RECOGNITION USING NEURAL NETWORK"** is the bonafide work of **"SAI KRISHNA R (310618104079),SAKETH RAMAN(310618104080), SURYA DHANUSH (310618104104)"** who carried out the project work under my supervision.

Dr. K M ANANDKUMAR	**Mr K.P.K.DEVAN**
HEAD OF THE DEPARTMENT	**SUPERVISOR**
Professor and Head	Associate Professor
Computer Science and Engineering	Computer Science
Easwari Engineering College	Easwari Engineering College
Ramapuram, Chennai 600089	Ramapuram, Chennai 600089

Submitted for Semester Examination held on ____________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

We hereby place our deep sense of gratitude to our beloved Founder Chairman of the institution, **Dr.T.R.Pachamuthu, B.Sc., M.I.E.,** for providing us with the requisite infrastructure throughout the course. We would also like to express our gratitude towards our Chairman **Dr.R.Shivakumar, M.D., Ph.D.** for giving the necessary facilities.

We convey our sincere thanks to **Dr.R.S.Kumar, M.Tech., Ph.D**. Principal Easwari Engineering College, for his encouragement and support. We extend our hearty thanks to **Dr.V.Elango, M.E., Ph.D.,** Vice Principal (academics) and **Dr.K.M.Anandkumar, M.Tech, Ph.D,** Vice Principal (admin), Easwari Engineering College, for their constant encouragement.

We take the privilege to extend our hearty thanks to **Dr.K.M.Anandkumar, M.Tech, Ph.D.,** Head of the Department, Computer Science and Engineering, Easwari Engineering College for his suggestions, support and encouragement towards the completion of the project with perfection.

We would also like to express our gratitude to our guide **MR K.P.K.Devan**, **M.Tech., (Ph.D),** Associate Professor, Department of Computer Science and Engineering, Easwari Engineering College, for his constant support and encouragement.

Finally, we wholeheartedly thank all the faculty members of the Department of Computer Science and Engineering for warm cooperation and encouragement.

ABSTRACT

After digitalization historical documents are available in the form of scanned images. However, having documents in the form of digital text simplifies the work of historians, as it e.g. makes it possible to search for text. Handwritten Text Recognition (HTR) is an automatic way to transcribe documents by a computer. There are two main approaches for HTR, namely hidden Markov models and Artificial Neural Networks (ANNs). The proposed HTR system is based on ANNs. Preprocessing methods enhance the input images and therefore simplify the problem for the classifier. These methods include contrast normalization as well as data augmentation to increase the size of the dataset. Optionally, the handwritten text is set upright by a deslanting algorithm. The classifier has Convolutional Neural Network layers to extract features from the input image and Recurrent Neural Network layers to propagate information through the image. The RNN outputs a matrix which contains a probability distribution over the characters at each image position. Decoding this matrix yields the final text and is done by the connectionist temporal classification operation. A final text post processing accounts for spelling mistakes in the decoded text. Five publicly accessible datasets are used for evaluation and experiments. Three of these five datasets can be regarded as historical, they are from the 9*th* century, from the 15*th* until the 19*th* century and from around the year 1800. The accuracy of the proposed system is compared to the results of other authors who published their results. Preprocessing methods are optionally disabled to analyze their influence. Different decoding algorithms and language models are evaluated. Finally, the text postprocessing method is analyzed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	**Expansion**
HTR	Handwritten Text Recognition
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
HMM	Hidden Markov Models
LM	Language Model
CTC	Connectionist Temporal Classification

CHAPTER 1
INTRODUCTION

1.1 General

Handwritten Text Recognition (HTR) is the task of transcribing handwritten text into digital text. It is divided into online and offline recognition. Online recognition is performed while the text to be recognized is written (e.g. by a pressure sensitivity device), therefore geometric and temporal information is available. Offline recognition, on the other hand, is performed after the text has been written. The text is captured (e.g. by a scanner) and the resulting images are processed. Online recognition is regarded as the easier problem. Challenges regarding HTR include the cursive nature of handwriting, the variety of each character in size and shape and large vocabularies. Currently, huge amounts of historical handwritten documents are published by online libraries. Transcribing and indexing makes these documents easier to access.

1.2 Objective

To identify handwritten characters with the use of neural networks. We have to construct suitable neural network and train it properly. The program should be able to extract the characters one by one and map the target output for training purpose.

1.3 Scope of the Project

This thesis focuses on the classifier, its parameters, preprocessing methods for the input and a text-postprocessing method for the output. Only offline HTR is discussed because images of handwritten text serve as input. The investigated classifier builds on Artificial Neural Networks (ANNs). Hidden Markov Models (HMMs) are also applied to HTR tasks. However, the published results of HTR contests show that ANNs outperform HMMs, therefore ANNs have been chosen for this thesis. Document analysis methods to detect the text on a page or to segment a page into lines are not discussed. The only exception is word-segmentation, which is presented because the datasets are annotated on line-level. Therefore, one possibility is to segment text-lines into words and classify each word on its own, while another one is to avoid segmentation and feed complete lines into the classifier.

CHAPTER 2
LITERATURE SURVEY

2.1 INTRODUCTION

A literature survey is a documentation of a comprehensive review of the published and unpublished work from secondary sources data in the areas of specific interest to the researcher.

2.2 RELATED WORKS

2.2.1 Offline Handwritten English Numerals Recognition using Correlation Method

In this paper the author has proposed a system to efficiently recognize the offline handwritten digits with a higher accuracy than previous works done. Also previous handwritten number recognition systems are based on only recognizing single digits and they are not capable of recognizing multiple numbers at one time.So the author has focused on efficiently performing segmentation for isolating the digits.

2.2.2 Intelligent Systems for Off-Line Handwritten Character Recognition: A Review

Handwritten character recognition is always a frontier area of research in the field of pattern recognition and image processing and there is a large demand for Optical Character 4 Recognition on handwritten documents. This paper provides a comprehensive review of existing works in handwritten character recognition based on soft computing techniques during the past decade.

2.2.3 Fuzzy Based Handwritten Character Recognition System

This paper presents a fuzzy approach to recognize characters. Fuzzy sets and fuzzy logic are used as bases for representation of fuzzy character and for recognition. This paper describes a fuzzy based algorithm which first segments the character and then using a fuzzy system gives the possible characters that match the given input and then using a defuzzification system finally recognizes the character.

2.2.4 An Overview of Character Recognition Focused on Off-Line Handwriting

Character recognition (CR) has been extensively studied in the last half century and progressed to a level sufficient to produce technology driven applications. Now, the rapidly growing computational power enables the implementation of the present CRmethodologies and creates an increasing demand on many emerging application domains, which require more advanced methodologies.

2.2.5 Image preprocessing for optical character recognition using neural networks

Primary task of this master's thesis is to create a theoretical and practical basis of preprocessing of printed text for optical character recognition using forward-feed neural networks. Demonstration application was created and its parameters were set according to results of realized experiments.

2.2.6 Recognition for Handwritten English Letters: A Review

Character recognition is one of the most interesting and challenging research areas in the field of Image processing. English character recognition has been extensively studied in the last half century. Nowadays different methodologies are in widespread use for character recognition. Document verification, digital library, reading bank deposit slips, reading postal addresses, extracting information from cheques, data entry, applications for credit cards, health insurance, loans, tax forms etc. are application areas of digital document processing. This paper gives an overview of research work carried out for recognition of handwritten English letters. In Hand written text there is no constraint on the writing style. Handwritten letters are difficult to recognize due to diverse human handwriting style, variation in angle, size and shape of letters. Various approaches of hand written character recognition are discussed here along with their performance.

2.2.7 Diagonal Based Feature Extraction For Handwritten Alphabets Recognition System Using Neural Network

An off-line handwritten alphabetical character recognition system using a multi-layer feed forward neural network is described in the paper. A new method, called, diagonal based feature extraction is introduced for extracting the features of the handwritten alphabets. Fifty data sets, each containing 26 alphabets written by various people, are used for training the neural network and 570 different handwritten alphabetical characters are used for testing. The proposed recognition system performs quite well yielding higher levels of recognition accuracy compared to the systems employing the conventional horizontal and vertical methods of feature extraction. This system will be suitable for converting handwritten documents into structural text form and recognizing handwritten names.

2.3 Issues in Existing System // RELATED WORKS which one?

Not enough dataset is available to make accurate predictions. Accuracy of existing systems is lower. The issue is that there's a wide range of handwriting – good and bad. This makes it tricky for programmers to provide enough examples of how every character might look. Also, sometimes, characters look very similar, making it hard for a computer to recognise accurately.

2.4 Proposed System

The proposed system makes use of ANNs, an illustration is given in Figure. Multiple Convolutional Neural Network (CNN) layers are trained to extract relevant features from the input image. These layers output a 1D or 2D feature map (or sequence) which is handed over to the Recurrent Neural Network (RNN) layers. The RNN propagates information through the sequence. Afterwards, the output of the RNN is mapped onto a matrix which contains a score for each character per sequence element. As the ANN is trained using a specific coding scheme, a decoding algorithm must be applied to the RNN output to get the final text. Training and decoding from this matrix is done by the Connectionist Temporal Classification (CTC) operation. Decoding can take advantage of a Language Model (LM).

2.5 Summary

This chapter covers the existing systems, their limitations and their applications that are used as references throughout the proposed system.

CHAPTER 3

STATE OF THE ART

3.1 Introduction

This chapter first introduces the five datasets used to evaluate the proposed HTR system. Afterwards, the foundations of ANNs are presented. The focus lies on the parts relevant for HTR, namely CNNs to extract features and RNNs to propagate information through the image. Loss calculation and decoding are done by the CTC operation. Finally, two successful HTR approaches are presented: the first method builds on HMMs, while the second one uses ANNs.

3.2 Datasets And Evaluation Metric

Five datasets are used to evaluate the proposed HTR system. This section first gives a comparison of the dataset statistics. Afterwards each dataset is presented in detail. Finally, the evaluation method for those datasets is described.

3.2.1 Datasets

A comparison of dataset statistics is given. The number of characters is between 48 (SaintGall) and 93 (Bentham). A similar character distribution can be found for all datasets. It is written in Latin, the occurrence frequency of the character “i” is around 10% while it is 0.01% for the character “X”. Unique words are calculated by splitting a text into words and then only count the unique ones. IAM has the highest number of unique words, while CVL has the lowest number. Out-Of-Vocabulary (OOV) words are words contained in the test-set but not in the training or validation-set. The performance of a word-level LM is influenced by the percentage of unknown words, therefore the number of OOV words gives a hint if using such a LM makes sense.

Some notes on the datasets:

• CVL : this dataset is mainly dedicated for writer retrieval and word spotting tasks. It contains handwriting from 7 texts written by 311 writers. The texts are in English and German language. There is only annotation for words but not for punctuation marks. Data is annotated on word-level.

• IAM : contains English text. The version of the dataset described by Marti and Bunke contains 1066 forms filled out by approximately 400 writers. It has since been extended, the IAM website lists 1539 forms and 657 writers. Data is annotated on word-level and on line-level.

• Bentham : written by Jeremy Bentham (1748-1832) and his secretarial staff. Contains 433 pages. Mostly written in English, some parts are Greek and French. Data is annotated on line-level and partly on word-level.

• Ratsprotokolle : texts from council meetings in Bozen from 1470 to 1805. An unknown number of writers with high variability in writing style created this dataset. It is written in Early Modern German. Data is annotated on line-level.

• SaintGall : this dataset contains 60 pages with 24 lines per page. It is written in Latin by a single person at the end of the 9*th* century. It describes the life of Saint Gall. Data is annotated on line-level

3.2.2 Evaluation Metric

The most common error measures for HTR are Character Error Rate (CER) and Word Error Rate (WER). CER is calculated by counting the number of edit operations to transfer the recognized text into the ground truth text, divided by the length of the ground truth text. The numerator essentially is the Levenshtein edit-distance and the nominator normalizes this edit-distance. For WER, the text is split into a sequence of words. This word sequence can then be used to calculate the WER just as a character sequence is used to calculate the CER. Both CER and WER can take on values greater than 100% if the number of edit operations exceeds the ground truth length [Blu15]: if “abc” is recognized but the ground truth is “xy”, the CER has a value of 3/2 = 150%.

$CER = \#insertions + \#deletions + \#substitutions/\ length(GT)$
As an example the CER and WER of the recognized text “Hxllo World” with the ground truth text “Hello World” is calculated. The character edit-distance is 1, the length of the ground truth text is 11, therefore $CER = 1/11$. For WER the unique words are written to a table with unique identifiers yielding $w1$=“Hxllo”, $w2$=“Hello”, $w3$=“World”. Then, each word is replaced by its identifier from the table, so the two strings become “$w1w3$” and “$w2w3$”. The word edit-distance is 1, the ground truth length is 2, therefore WER=1/2.

3.3 Artificial Neural Networks

This section introduces ANNs with a strong focus on topics which are necessary to understand their application in the context of HTR. The foundations of ANNs are presented. Building on this knowledge, two special types of ANNs, namely CNNs and RNNs are discussed.

3.3.1 Basics

First, the basic building block of an ANN, the neuron, is introduced. Afterwards the combination of neurons to form neural networks with multiple layers and the training of such networks is presented.

Perceptron

First ideas for ANNs date back to the 1940s [GBC16]. Linear models of the form
$y(x, w) = H(\mathrm{P}\ i\ wi \cdot xi)$ are used for binary classification tasks, where x represents the input vector, w the parameter vector, i the indexing variable of the vectors and H the Heaviside step function [GBC16]. Both models presented in the following

use linear functions. The McCulloch-Pitts Neuron has excitatory and inhibitory inputs with fixed (manually set) parameters. Rosenblatt introduces the perceptron which is able to learn the parameters by using training samples for which the target categories are known. As both models build on linear functions they are not able to approximate all possible functions. One of the cases where linear models fail is the XOR function: there is no straight line which can be drawn to separate the two classes..

Multilayer Perceptron

In the 1980s the main idea to improve ANNs was to combine simple units (neurons)
to complex neural networks. Those neural networks are able to approximate arbitrary functions and therefore can solve the XOR problem [GBC16]. Multiple connected layers of neurons are used, which is the reason for the name Multilayer Perceptron (MLP). The first layer of neurons is called input layer, the last layer output layer, and the layers in between are called hidden layers. There are different ways to count the layers, this thesis sticks with the counting method described by Bishop by taking the number of hidden layers plus 1.

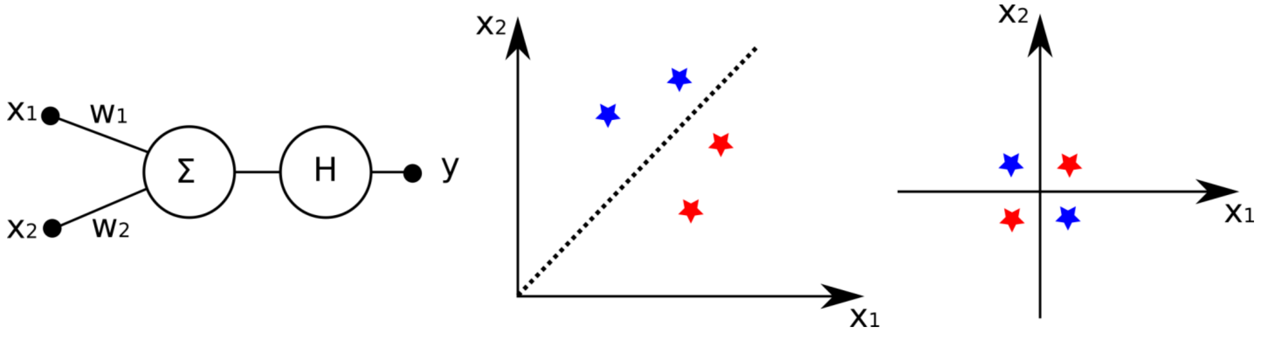

Figure 3.1 Multilayer Perceptron

Convolutional Neural Networks

The traditional image classification approach is to extract local features from the image, which then serve as input to a classifier. An ANN can be used as such a classifier. It is also possible to directly input an image into an ANN and let it learn the features itself. The problems with this approach are :

• Large images require many parameters to be learned. Think of an input image with size 100×100 pixels and a fully connected first layer with 100 hidden neurons. This gives 106 weights to be learned in the first layer. A large number of parameters increases the capacity of the model and therefore also increases the number of samples needed to train the model.

• Invariance to translation or size variance is difficult to achieve. It is possible to learn this invariance by showing training samples at different locations and sizes. This results in similar (redundant) weight patterns at different locations, such that features can be detected wherever they appear in the image. Again, showing the same images at different positions and sizes increases the size of the training-set.

• The topology of images is ignored. Images have a strong local 2D structure, but in a fully connected first layer the ordering of the input neurons (image pixels) does not matter.

CNNs are able to tackle these problems and are tremendously successful in practical applications. CNNs work with data that is organized in a grid-like structure, such as images which have a 2D structure or plain audio data which have a 1D structure. The key difference between MLPs and CNNs is that in CNNs at least one layer uses a convolution operation instead of the usual matrix multiplication.

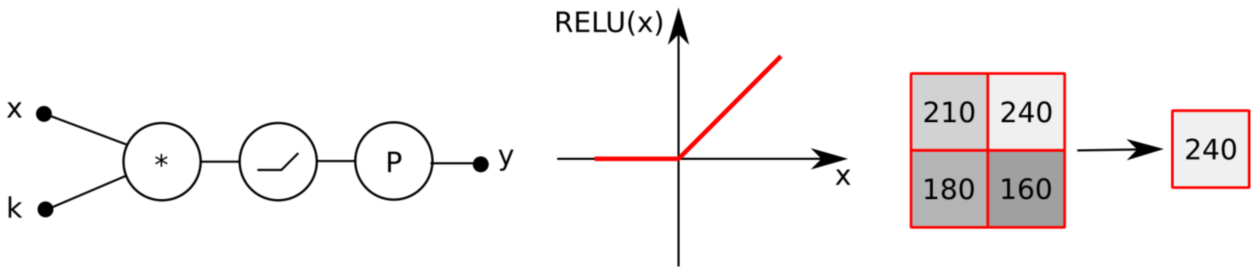

Figure 3.2 CNN

A diagram of a CNN layer can be seen in Figure. It consists of three parts which are afterwards discussed in more detail:

• Convolution: the activation of the neuron is calculated by the convolution of an input and a kernel.

• Nonlinearity: a nonlinear function is applied to the activation of the neuron.

• Pooling: this operation replaces the output at a certain location by a summary statistic of nearby outputs.

Convolution is an operation on two functions and outputs a third function. The operation is commutative, associative and distributive. For computer vision tasks a discrete version of convolution is used. In the context of CNNs, the first argument x is called input, the second argument k kernel and the output y is called feature map. The time-indexes are denoted by t and τ . Because convolution is commutative, the arguments to index x and k can be swapped in a way that is convenient for implementations.

For the 2D structure of images a 2D version of discrete convolution is needed. Indexing is now achieved by i and m in the first dimension and j and n in the second dimension. Only the version used in implementation with swapped arguments is given.

A Rectified Linear Unit (RELU) is used as a nonlinear activation function in CNNs. It is defined as $y = g(a) = max(0, a)$ on the activation a of the neuron. The

RELU essentially zeros out the negative part of its argument while it keeps the positive
part untouched. A plot is shown in Figure 2.8 in the middle.
Pooling is added on top of the nonlinearity: it replaces the output at a certain location by a summary statistic of nearby outputs. An example for pooling is max-pooling on a 2 × 2 neighborhood which is illustrated in Figure 2.8 on the right: the feature map is divided in 2 × 2 squares and for each of them, only the maximum value is kept. This operation reduces the height and width of the feature map by a factor of 2.
Training of CNNs is similar to training MLPs. A key difference between those two ANN types is that a CNN layer shares the weights of its convolution kernel: the kernel is shifted through the image, therefore the same weights are applied to multiple image positions. The backpropagation algorithm to calculate the gradient is slightly modified to account for this weight sharing. This is achieved by first calculating all partial derivatives as in a fully connected layer, and then summing up all partial derivatives belonging to the same kernel weight.

Recurrent Neural Networks

A RNN is a special kind of ANN which is used to model sequential data such as handwritten text or speech. Sequential data can be of arbitrary length along the time axis, where time axis in the context of speech is clear, while in the context of handwritten text it means the axis along which the writing happens. This is from left to right for e.g. English text and the other way round for e.g. Arabic text. The intuition of using RNNs is that relevant information can be scattered around multiple time-steps.

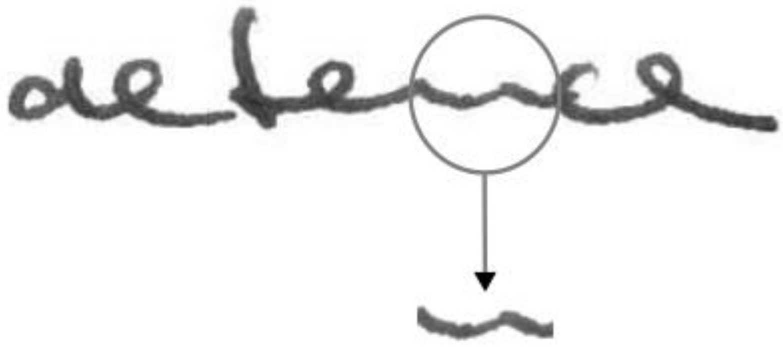

Figure 3.3 RNN

An example is shown in Figure 3.3, where it is easy to identify the character "n" when looking at the whole image, but it is impossible to tell if it is an "n" or "u" when looking at the character on its own.

Additionally to the input and output sequence *xt* and *yt* , RNNs also have an internal state sequence *ht* , which memorizes past events. The internal state *ht* is a function of the current input *xt* and the last internal state *htt* 1, that means *ht* = *f*(*htt* 1, *xt*).The detailed formulas for calculating the activation *at* , the hidden state *ht* , the output *ot* and the normalized output *yt* are shown in Equations.

$at = W \cdot htt\ 1 + U \cdot xt$

$ht = tanh(at)$

$ot = V \cdot ht$

$yt = sof\ tmax(ot)$

The computational graph of a RNN has cycles which represent the connections between neighboring time-steps. Unfolding (i.e. replicating) the graph of a RNN to the length of the input sequence results in a conventional ANN. This ANN is trained by standard methods, i.e.first calculating the gradient of the error function and then updating the parameters with an algorithm like gradient descent. The gradient calculation is called Backpropagation Through Time (BPTT) when applied to an unfolded graph.

Similar as in the case of CNNs, the parameters (*U* and *V*) are shared between different time-steps.When used for practical purposes RNNs have the problem of vanishing gradients. This can be seen when looking at an unfolded graph: long-term memory is achieved by applying the same function over and over again in the form $ht = f(f(f(...), xtt\ 1), xt)$. ---------- (i)
Depending on the function *f* this either vanishes or explodes the back propagated error signal. Goodfellow et al. give a detailed explanation by decomposing the function *f*, with the relevant part being the matrix *W*, into its eigenvalues and eigenvectors. Raising the (diagonal) eigenvalue matrix to the power of *k*, diagonal elements $|dii|$ smaller than 1 tend to 0, whereas values larger than 1 tend to ∞ when increasing *k*.

Long Short-Term Memory (LSTM)

Hochreiter discusses the vanishing gradient problem of RNNs. To avoid them he introduces Long Short-Term Memory Recurrent Neural Networks (LSTM) . A LSTM block learns when to remember and when to forget past events which is achieved by introducing gating neurons. There are two recurrences, an outer and an inner one. The outer recurrence is realized by using the output *yt* from time-step *t* as input for time-step *t* + 1. The inner recurrence is the linear self-loop of the state cell. This inner loop is called Constant Error Carousel (CEC). Gates control the information flow in a LSTM block: the input gate controls how much the input data effects the state cell and the output gate controls how much the internal state effects the output. A problem with the original LSTM formulation is that the state cell accumulates more and more information over time, therefore the internal state has to be manually reset . To solve this problem Gers et al introduces a forget gate to reset or fade out the internal state. Peephole connections are added to a LSTM block such that the cell can take direct control of the gating neurons in the next time-step.

The vanishing gradient problem is solved by the CEC. When the error signal flows backward in time and no input or error is present (gates are closed), the signal remains constant, neither growing nor decaying. While a vanilla RNN is able to bridge time-lags of 5-6 time-steps between input event and target event, LSTM is able to bridge 1000 time-steps. This and the stable error propagation make LSTMs a popular implementation of RNNs which are widely used in practice when working with sequences .

Multidimensional LSTM (MDLSTM)

RNNs process 1D sequences and therefore have one recurrent connection. Graves et al introduce Multidimensional RNNs (MDRNN) by using *n* recurrent connections to process nD sequences. In the forward pass the hidden layer gets input from the data and from the *n* recurrent connections one time-step back. Suitable ordering of the data points is needed such that the RNN already processed points from which it receives its inputs at a certain time-step. Gradient calculation is done by an extension of the BPTT for *n* dimensions. Also 1D LSTMs can be extended to Multidimensional LSTMs (MDLSTM) by introducing *n* self-loops with *n* forget gates.

Dilated Convolution(DC)

Dilated Convolution (DC) is a modification to standard CNN. However, it is used in tasks which usually use RNNs, therefore the discussion about DC is done in this section. Oord et al. describe DC as “convolutions with holes”. The kernel *k* is not applied to each element of the input *x* as in vanilla convolution, but only to each *nth* element. The formula for 1D convolution is modified as shown in Equation to account for the dilation width .

$y(t) = x(t) *_n k(t) = \infty X \tau = -\infty \; x(t \; t \; n \cdot \tau) \cdot k(\tau)$ ---------(ii)

As shown by Strubell et al multiple layers of DC with exponentially increasing dilation width can be stacked to form an Iterated Dilated Convolutional Network (IDCN). IDCN is used to propagate information through a long sequence with only a few DC layers. Given the kernel size *w*, the width of the receptive field *r* of a CNN increases linearly in the number of layers *l*, as can be seen in Equation (i). For IDCN the width increases exponentially as shown in Equation (ii).

$rCNN = l \cdot (w \; w \; 1) + 1$ - (i)

$rIDCN = (2l \; l \; 1) \cdot (w \; w \; 1) + 1$ - (ii)

CHAPTER 4

METHODOLOGY

4.1. Introduction

The methodology chapter explains the proposed system with the help of an architecture design consisting of the various components, modules and process required to satisfy the system requirements.

The methods used for the proposed HTR system are presented in this chapter. Preprocessing includes deslanting to set the handwritten text upright and random modifications to the original images to enlarge the size of the dataset. Word segmentation can be applied to the line-images to keep the classifier small by only inputting word-images. Four published word segmentation approaches are presented and a new approach using best path decoding is proposed. An ANN is used as a classifier. To propagate information through the feature sequence different RNN types are integrated into the ANN. IDCN has so far only been applied to speech recognition, an architecture using this network layer for HTR is presented. Two state-of-the-art CTC decoding algorithms are discussed which extract the final labeling from the RNN output. Character-level or word-level LMs can improve the recognition performance by introducing knowledge about the language in the decoding step. Finally, a text-postprocessing method is presented which corrects spelling mistakes in the final text.

4.2. Preprocessing

Even tough the HTR system presented in this chapter is end-to-end trainable, preprocessing steps can help to increase the accuracy of the system.

4.2.1 Deslanting

Vinciarelli and Luettin defifine slant as follows: “The slant is the angle between the vertical direction and the direction of the strokes that, in an ideal model of handwriting, are supposed to be vertical”. Deslanting is the process of transforming
the handwritten text so that the slant angle gets minimized. The deslanting technique proposed by Vinciarelli and Luettin builds on the hypotheses that an image containing text is deslanted if the number of image columns with continuous strokes is maximal.

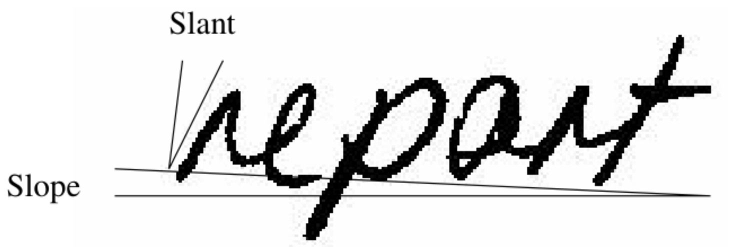

Figure 4.1 Deslanting

Deslanting Algorithm :

Data: image *I*, list of shear factors *A*

Result: deslanted image

```
1 $Ibw = threshold(I)$;
2 $S = \{\}$;
3 for $\alpha \in A$ do
4 $I\alpha = shear(Ibw, \alpha)$;
5 for $i \in colums(I\alpha)$ do
```

```
6 $h\alpha$ = number of foreground pixels in column $i$ of $I\alpha$;
7 $\Delta y\alpha$ = distance between fifirst and last foreground pixel in column $i$ of $I\alpha$;
8 if $h\alpha == \Delta y\alpha$ then
9 $S(\alpha) += h2\alpha$;
10 end
11 end
12 end
13 $\hat{}\alpha = argmax\alpha(S(\alpha))$;
14 return $shear(I, \hat{}\alpha)$;
```

Algorithm shows how this deslanting approach can be implemented. It takes a list of possible shear factors A which can be regarded as the search space of the algorithm. The input image I is mapped to a monochrome image Ibw by applying a threshold (e.g. by using Otsu's method). Afterwards a shear transform for each of the possible shear factors $\alpha \in A$ is applied to the image Ibw. For each sheared image $I\alpha$ and each of its columns i, the distance $\Delta y\alpha$ between the first and last foreground (black) pixel and the number of foreground pixels $h\alpha$ is calculated. A continuous stroke is indicated by the equality of $\Delta y\alpha$ and $h\alpha$. The value of h

2α is then added to the sum $S(\alpha)$, the reason for the square is that short strokes can be due to noise and are therefore lower weighted. The algorithm outputs the input image sheared by the factor α for which the sum $S(\alpha)$ has its maximum.

4.2.2 Word Segmentation

The first part introduces four explicit word segmentation approaches which use classic image processing techniques not involving ANNs. In the second part an implicit segmentation based on best path decoding is proposed.

Explicit Word Segmentation

Manmatha and Srimal propose a scale-space approach which segments a text-line into individual words. This method has a simple implementation and is therefore chosen for the evaluation in Chapter 4. A line-image consists of individual or connected characters. The characters representing a single word should be merged together. A way to achieve this is by transformation the image I into a blob-representation $Iblob$ by convolving it with an appropriate filter kernel F. Manmatha and Srimal use an anisotropic filter kernel F which is derived from a 2D Gaussian function G by summing up the second-order derivatives Gxx and Gyy. The parameters of the kernel are its scale σx in horizontal direction and its scale σy in vertical direction.

Implicit Word Segmentation

For the proposed approach a HTR system based on ANNs trained with the CTC loss is required. The CTC best path decoding algorithm is modified to output the best path without collapsing it. Word boundaries are encoded by (multiply) whitespace characters in the best path. Four bounding boxes can be used to segment this path.

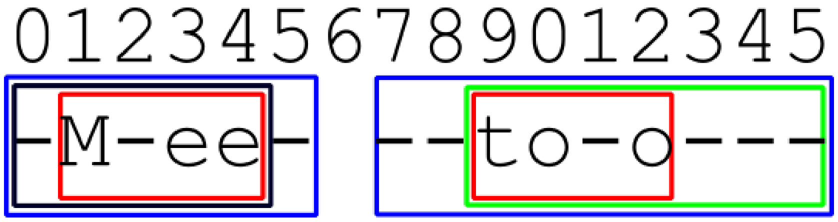

Figure 4.2.2 Implicit Word Segmentation

The first bounding box (blue) simply splits the path at whitespace characters and regards the remaining subpaths as words. The second (green) and third (black) bounding box remove the blanks on the left or on the right of the subpaths. Finally, the fourth (red) one removes the blanks on both sides.

An ANN scales the input image down by a fixed factor, e.g. by the factor 8 for the HTR system presented in this chapter. It is therefore possible to calculate the position of a word in the input image by multiplying the position on the best path by the mentioned scaling factor. The remaining question is if recognized characters on the best path lie near their position in the input image. For example, the RNN could align all characters to the left, regardless of their real position, because the CTC loss works in a segmentation-free manner. This question will be investigated in Chapter 5.

4.3 Language Model(LM)

A LM can be used to guide the CTC decoding algorithms by incorporating information about the language structure. The language is modeled on character-level or on word level, but the basic ideas are the same for both. For simplicity only a word-level LM is discussed. A LM is able to predict upcoming words given previous words and it is also able to assign probabilities to given sequences of words. For example, a well trained LM should assign a higher probability to the sentence “There are ...” than to “Their are ...”. A LM can be queried to give the probability $P(w|h)$ that a word sequence (history) h is followed by the word w.

Such a model is trained from a text by counting how often w follows h. The probability of a sequence is then

$P(h) = P(w1) \cdot P(w2|w1) \cdot P(w3|w1, w2) \cdot ... \cdot P(wn|w1, w2, ..., wnn\ 1)$.

Of course it is not feasible to learn all possible word sequences, therefore an approximation called N-gram is used. Instead of using the complete history, only a few words from the past are used to predict the next word. N-grams with N=2 are called bigrams. Bigrams only take the last word into account, that means they approximate $P(wn|h)$ by $P(wn|wnn\ 1)$. The probability of a sequence is then given by $P(h) = \mathrm{Q}\ |h|\ n{=}2P(wn|wnn\ 1)$.

Another special case is the unigram LM, which does not consider the history at all but only the relative frequency of a word in the training text,

i.e. $P(h) = \mathrm{Q}\ |h|\ n{=}1\ P(wn)$.

If a word is contained in the test text but not in the training text, it is called OOV word. In this case a zero probability is assigned to the sequence, even if only one OOV word occurs. To overcome this problem *smoothing* can be applied to the N-gram distribution, more details can be found in Jurafsky and Martin.

4.4 Prefix Tree

A prefix tree or trie (from re*trie*val) is a basic tool in the domain of string processing. It is used in the proposed CTC decoder to constrain the search space and to query words which contain a given prefix. Figure below shows a prefix tree containing 5 words. It is a tree-datastructure and therefore consists of edges and nodes. Each edge is labeled by a character and points to the next node. A node has a word-flag which indicates if this node represents a word. A word is encoded in the tree by a path starting at the root node and following edges labeled with the corresponding characters of the word. Querying characters which follow a given prefix is easy: the node corresponding to the prefix is identified and the outgoing edge labels determine the characters which can follow the prefix

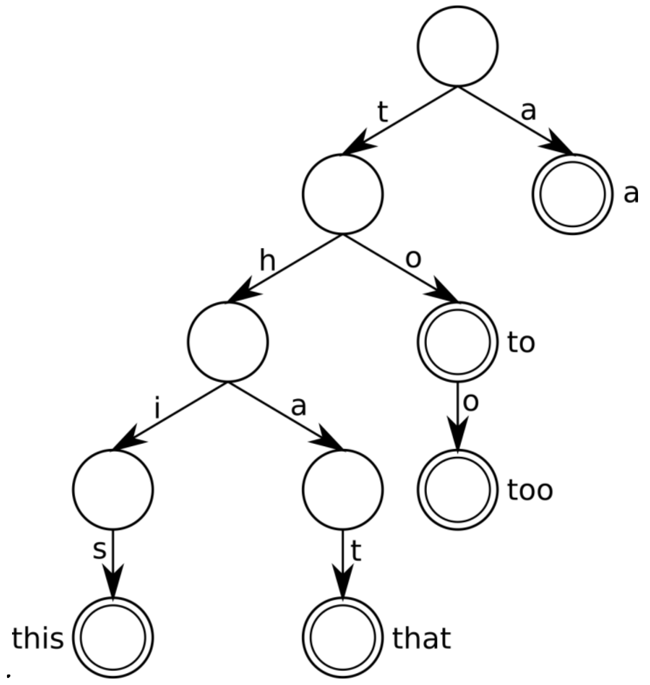

Figure 4.4 Prefix Tree

Prefix tree containing the words “a”, “to”, “too”, “this” and “that”. Double circles indicate that the word-flag is set.

It is also possible to identify all words which contain a given prefix: starting from the corresponding node of the prefix, all descendant nodes are collected which have the word-flag set. As an example, the characters and words following the prefix “th” in Figure are determined. The node is found by starting at the root node and following the edges “t” and “h”. Possible following characters are the outgoing edges “i” and “a” of this node. Words starting with the given prefix are “this” and “that”. Aoe et al show an efficient implementation of this datastructure. Finding the node for a prefix with length *L* needs *O*(*L*) time in their implementation. The time to find all words containing a given prefix depends on the number of nodes of the tree. An upper bound for the number of nodes is the number of words *W* times the maximum number of characters *M* of a word, therefore the time to fifind all words is $O(W \cdot M)$.

4.5 Classifier

An ANN consisting of CNN and RNN layers is used as a classifier. Different ANN architectures are presented which mainly differ in the used RNN type. Three CTC decoding algorithms with integrated LM are discussed.

4.5.1 Architecture

The architecture of the proposed ANN is mainly inspired by Shi et al and Voigtlaender et al. Three different types of RNNs are tested. The CNN layers and the mapping onto the (vector-valued) 1DArchitecture for the ANN with MDLSTM. Abbreviations: average (avg), bidirectional (bidir), vertical (vert), dimension (dim), batch normalization (BN), convolutional layer (Conv). RNN input sequence are identical for LSTM and IDCN. 7 CNN layers map the input image of size 800×64 onto a sequence of length 100 with 512 features per element. For MDLSTM the CNN layers are modified such that a 2D sequence is created with a width of 100, a height of 8 and 512 features per element. The CTC layer needs a 1D sequence (however, each sequence element itself is a vector containing the corresponding
character probabilities), therefore the output of the MDLSTM is averaged along the height dimension. A linear projection is applied to map the output of the RNN onto the character distribution for each time-step (distribution over all characters contained in the regarded dataset plus one extra character to represent the blank). Finally, the CTC decoding layer outputs a sequence (text) which is at most 100 characters long. This thesis evaluates the purely convolutional RNN type in the

domain of HTR for the first time. The IDCN architecture shown by Oord et al uses a residual block . The input of one layer is fed through a DC and the result
of two different nonlinearities is then multiplied. Skip connections add the result of this layer to an output matrix, while the residual part is fed into the next layer. The IDCN architecture used in the proposed HTR system is illustrated. The basic element is a DC layer with a given dilation width. Three layers of increasing dilation width (1, 2 and 4) are stacked and form a so called *block*. The output of each layer is fed to the next layer as well as to an intermediate output. Multiple blocks can be stacked, to avoid overfitting the kernel parameters are shared layer-wise. All the intermediate outputs are concatenated to form one (large) output matrix. For each time-step the features are projected onto the classes which serve as input for the CTC layer.

4.5.2 Training

The ANN is trained with the RMSProp algorithm which uses an adaptive learning rate. RMSProp divides the learning rate η by an exponentially decaying sum of
squared gradients *avg*. In its original formulation, the suggested value for the learning rate is 0.001 and 0.9 for the decaying factor γ, however Mukkamala and Hein give a more detailed explanation on parameter selection. An advantage over other training algorithms such as gradient descent is that the learning rate is adapted automatically, only an initial value has to be set.
As stated by Goodfellow et al, training a deep ANN can be problematic because the parameter update is done under the assumption that all other layers are unchanged. This of course is not true and the effect increases the more layers are added. Batch normalization tackles this problem by normalizing a batch of input

activations (for the input or any hidden layer) by subtracting the mean and dividing by the standard deviation. While training, the error of the training-set keeps decreasing, while the error of the validation-set starts to increase again at a certain point. This is due to overfitting. The early stopping algorithm as shown by Goodfellow et al is used to prevent overfitting. The patience parameter *p* denotes the number of training epochs the algorithm continues after the epoch with the best result achieved so far on the validation-set. If no better result can be found after *p* trials training terminates.

4.5.3 CTC Decoding with Language Model

The algorithm presented first is token passing and works on word-level. It looks for the most probable sequence of dictionary words and is guided by a word-level LM. The second algorithm is Vanilla Beam Search (VBS) decoding which keeps track of the most promising labelings while iteratively going through all time-steps. A character-level LM helps to consider only labelings with a likely character sequence. Finally, a new algorithm called Word Beam Search (WBS) decoding is proposed, which combines the advantages of the former two.

CTC Token Passing

This algorithm works on word-level. It takes a dictionary containing all words to consider and a word-bigram LM as input. The output word sequence is constrained to the dictionary-words. The token passing algorithm is first described by Young et al in the domain of HMMs and the following discussion is based on this publication. A single word is modeled as a state machine. Each state represents a character of the word. The state machine can either stay in a state or continue to the next state. Going from state i to state j is penalized by pij and a state at time-step t is penalized by dj (t). Aligning a word model to a feature sequence means finding the best path through the state machine. The cost for a possible alignment is $S(i0, i1, ..., iT) = \text{P } i\ (pi + di(t))$. Finding the best path through the states can be formulated as a token passing algorithm:

• Calculate the cost (score) for each start state and put the value into a token.

• Move all tokens one time-step ahead, incrementing the cost value by pij and dj (t).

• Clear the old tokens and then go over all states which hold a (new) token and take the best one for each state. This token represents the minimal cost path to this state.

• If more than one word should be considered multiple word models are put in parallel.

The information in the token has to be extended to save the sequence (history) of words the path goes through. The transition from one word to another can be penalized by a word-bigram LM. For CTC decoding, the original algorithm has to be changed to account for the blank Label. The states of a word model are denoted by s in the pseudo-code, where $s = 0$ is the start state and $s = -1$ is the final state of a word. The extended word $w0$ is the original word w extended by a blank label at the beginning, at the end and in between all characters. For example, “Hello” gets extended to “-H-e-l-l-o-”. A token for segment s of word w at time-step t is denoted by $tok(w, s, t)$ and holds the score and the word-history. Time and words are 1-indexed, e.g. $w(1)$ denotes the first character of a word. The algorithm has a time complexity of $O(T \cdot W2)$, where T denotes the length of the output sequence of the RNN and W the number of words contained in the dictionary.

CTC Vanilla Beam Search

The algorithm described here is a simplified version of the one published by Hwang and Sung. The depth pruning is removed because the number of time-steps is limited for offline HTR. At each time-step a labeling can be extended by a new character from the set of possible characters. This creates a tree of labelings which grows exponentially in time. To keep the algorithm feasible only the best labelings at each time-step are kept and serve as input to the next time-step. A character is encoded by a path which contains one or more instances of the character and is optionally followed by one or more blanks. Repeated characters in a labeling are encoded by a path which separates the repeated characters by at least one blank. It is easy to see that different paths can encode the same labeling, e.g. *B*("a -")=*B*("a a")="a". Therefore, the probabilities of all paths yielding the same labeling must be summed up to get the probability of that labeling the same labeling, e.g. *B*("a -")=*B*("a a")="a". Therefore, the probabilities of all paths yielding the same labeling must be summed up to get the probability of that labeling.

CTC Word Beam Search

The motivation to propose the WBS decoding algorithm is twofold:

• VBS works on character-level and does not constrain its beams (text candidates) to dictionary words.

• The running time of token passing depends quadratically on the dictionary size, which is not feasible for large dictionaries.Further, the algorithm does not handle non-word character strings. Punctuation Marks and large numbers occur in the datasets, however, putting all possible combinations of these into the dictionary would enlarge it unnecessarily. WBS decoding is a modification of VBS and has the following properties:

• Words are constrained to dictionary words.

• Any number of non-word characters is allowed between words.

• A word-level bigram LM can optionally be integrated.

• Better running time than token passing (regarding time-complexity and real running time on a computer).

The dictionary words are added to a prefix tree which is used to query possible next characters and words given a word prefix. Each beam of WBS is in one of two states. If a beam gets extended by a word-character (typically “a”, “b”, ...), then the beam is in the word-state, otherwise it is in the non-word-state. A beam is extended by a set of characters which depends on the beam-state. If the beam is in the non-word-state, the beam-labeling can be extended by all possible non-word-characters (typically “ ”, “.”, ...). Furthermore, it can be extended by each character which occurs as the first character of a word. These characters are retrieved from the edges which leave the root node of the prefix tree. If the beam is in word-state, the prefix tree is queried for a list of possible next characters. The last word-characters form the prefix “th”, the corresponding node in the prefix tree
is found by following the edges “t” and “h”. The outgoing edges of this node determine the next characters, which are “i” and “a” in this example. In addition, if the current prefix represents a complete word (e.g. “to”), then the next characters also include all non-word-characters.

Optionally, a word-level LM can be integrated. A bigram LM is assumed in the following text. The more words a beam contains, the more often it gets scored by the LM. To account for this, the score gets normalized by the number of words.

Four possible LM scoring-modes exist and names are assigned to them which are used throughout the thesis:

• Words: only a dictionary but no LM is used.

• N-grams: each time a beam makes a transition from the word-state to the non word-state, the beam-labeling gets scored by the LM.

• N-grams + Forecast: each time a beam is extended by a word-character, all possible next words are queried from the prefix tree. Figure 3.15 shows an example for the prefix “th” which can be extended to the words “this” and “that”. All beam extensions by possible next words are scored by the LM and the scores are summed up. This scoring scheme can be regarded as a LM forecast.

• N-grams + Forecast + Sample: in the worst case, all words of the dictionary have to be taken into account for the forecast. To limit the number of possible next words, these are randomly sampled before calculating the LM score. The sum of the scores must be corrected to account for the sampling process.

4.6 Postprocessing

The text recognized by the classifier can be further processed to correct spelling mistakes. This post processing has no access to the optical model (RNN output) and therefore works solely on text-level. Tong and Evans [TE96] propose a text correction system which uses a dictionary. The dictionary is created from the training text and holds a list of unique words and their occurrence frequency. First the algorithm splits the recognized text into a list of words. For each word not contained in the dictionary a set of suggestions is created. This is done by edit operations which are applied to the original word: characters are inserted, deleted and substituted. As an example a subset of the suggestions for the misspelled word “Hxllo” are: “Hcllo”, “Hdllo” and “Hello”. The probability of executing a given edit operation for a character is learned while training. To give an example, substituting the characters “a” and “o” happens more often than substituting “x” and “o” because the former ones look more similar. The overall probability *score*(*s.text*) of a suggestion *s* is the product of the edit operation probability *s.prob* and the word frequency *P*(*s.text*). Edit operations yielding non-words (like “Hdllo” in the example above) are removed in this step because these words have zero probability. The best scored suggestion for each misspelled word is selected and the words are concatenated to form the corrected text line. The text correction is trained in parallel to the classifier. The probability of an edit operation (e.g. substituting “o” for “a”) is increased each time this edit operation is able to correct a word.

4.7 System Architecture

System Architecture is a conceptual model that defines the structure and behavior of the program.

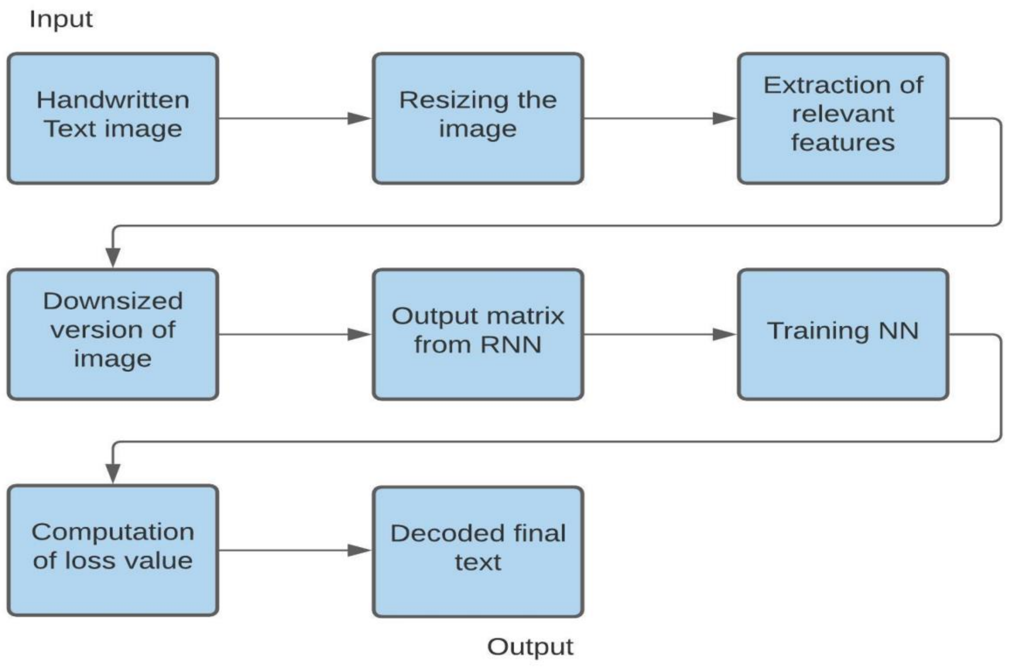

Figure 4.7 System Architecture

4.8 Functional Architecture

Functional Architectur**e** is an architectura**l** model that identifies system functions and their interactions. It defines how the functions will operate together to perform the system mission

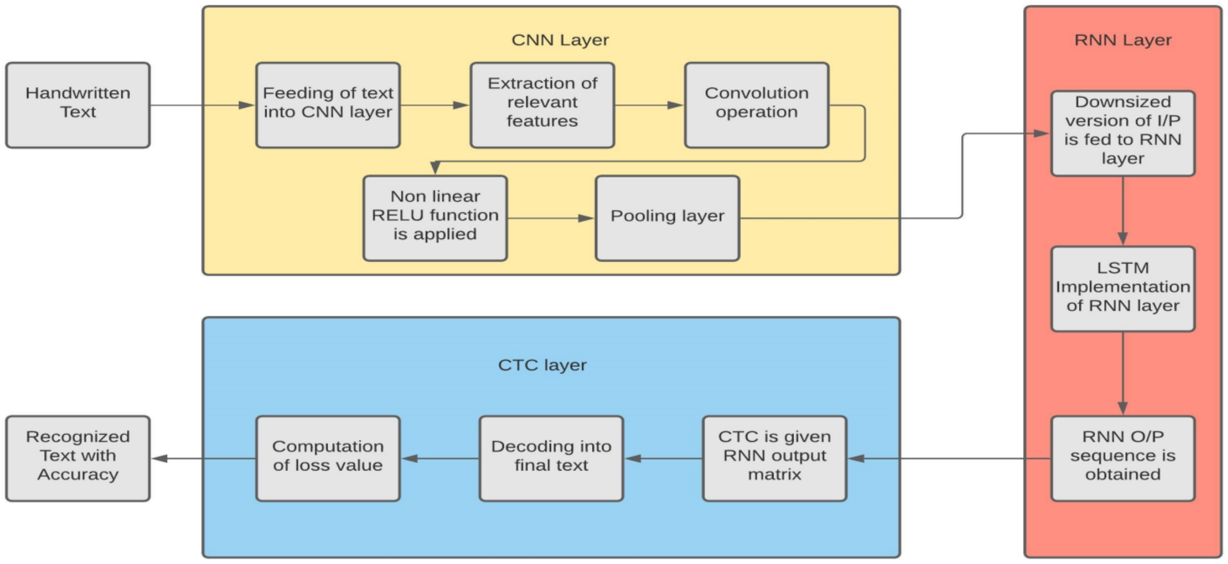
CNN Layer
Handwritten Text
Feeding of text into CNN layer
Extraction of relevant features
Convolution operation
Non linear RELU function is applied
Pooling layer
RNN Layer
Downsized version of I/P is fed to RNN layer
LSTM Implementation of RNN layer
RNN O/P sequence is obtained
CTC layer
CTC is given RNN output matrix
Decoding into final text
Computation of loss value
Recognized Text with Accuracy

Figure 4.8 Functional Architecture

4.9 Summary

This chapter presented the relevant parts of the proposed HTR system. An algorithm is shown which removes the slant angle of handwritten text by searching the sheared image with maximum continuous vertical lines in it. Either complete line-images or word-images can serve as input for the classifier. In the former case, no segmentation is needed for the task of HTR. However, the HTR system can be used for word-segmentation by adapting the best path decoding algorithm. In the latter case, the classifier can be kept small. Four word segmentation approaches are presented which make use of different distance measures to classify gaps between word candidates. Regarding the classifier itself, three architectures are proposed which make use of different RNN types. Multiple decoding algorithms exist which optionally can integrate a LM. The presented text-postprocessing method searches for the most likely correction of a misspelled word and also takes the character confusion probabilities of the HTR system into account, i.e. it learns the weaknesses of the system.

CHAPTER 5

EXPERIMENTATION AND RESULT

5.1 Introduction

This chapter gives details about the experimentation and result of the proposed model and the performance and result analysis.

5.2 Experiments

The first experiments analyze optional data preprocessing. Afterwards, the influence of different RNN types is examined. The hyperparameter selection for IDCN is presented to give an idea which parts mainly influence the result. The decoding algorithms and the text-postprocessing are able to improve the accuracy on noisy data by incorporating language information and are tested on two datasets with different language properties.

5.2.1 Image Preprocessing and Data Augmentation

All dataset images are converted to gray-value images to decrease the size of the database. Keeping the input size of the ANN constant simplifies the implementation of the classifier, however, images from a dataset vary in size. The simplest method as used by Shi et al is to scale the input image to the desired input size which yields distorted images. Instead of resizing, the image can be downsized (horizontally and vertically by the same scaling factor) if needed and then copied into an image of the desired input size. The parts not covered by the copied image are filled with a background color. This background color is calculated by taking the gray-value with the highest bin in the image histogram. Data augmentation happens on-the-fly when the classifier asks for a new training sample and therefore does not increase the database size. As already described, the training image is copied to the input image to avoid distortion. A simple type of data augmentation is

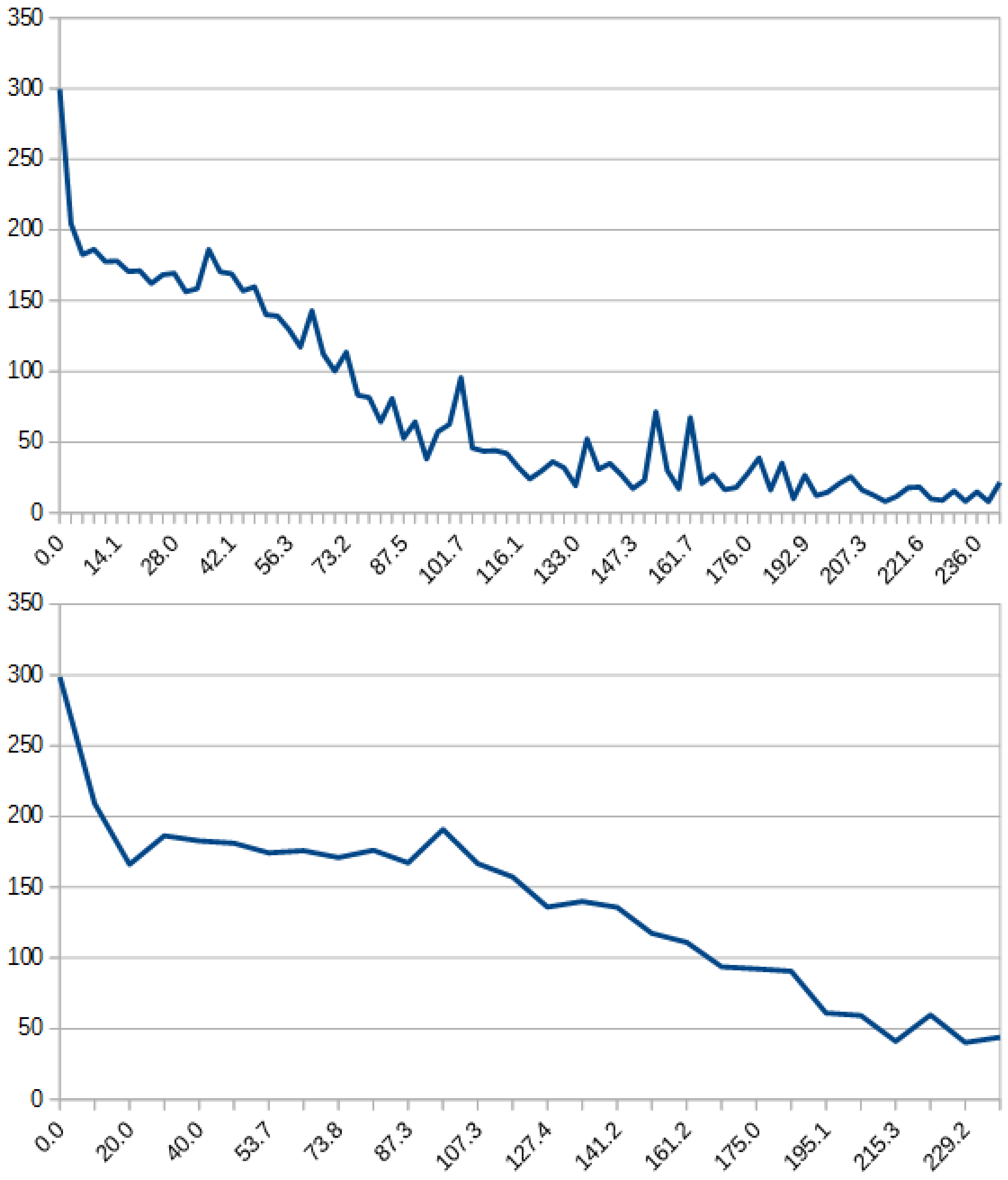

Figure 5.2.1 Image Preprocessing and Data Augmentation

Plot of loss functions for the first 4 minutes of training on the SaintGall dataset. Horizontal axis represents the time in seconds, the vertical axis represents the CTC loss value. Top: batch size is 5. Bottom: batch size is 25.

5.2.2 Deslanting

Deslanting is a computational expensive algorithm as multiple sheared versions of the original image have to be calculated. The implementation has a search space of 9 different shear factors in the range from *m* 1 to 1. The algorithm is implemented with OpenCV for the CPU and with OpenCL for the GPU. As can be seen in Table 4.4 the running time per sample is reduced from 21.7*ms* to 12.3*ms* when moving the computation from the CPU to the GPU. This performance improvement is achieved by parallelizing the loops of the algorithm and doing the summation in a final reduction step. The algorithm is optionally applied when creating a LMDB database.

5.3 Main Results

Only a subset of all possible hyperparameter combinations is evaluated. Line-images are fed into the classifier and no explicit word segmentation takes place. Contrast normalization and data augmentation are enabled. The MDLSTM architecture is chosen because it performs best as shown by the conducted experiments. All presented decoding algorithms are evaluated, however postprocessing is just applied to prefifix search and VBS.
The LM weighting parameter γ of VBS is set to 0.01. The Bentham and Ratsprotokolle datasets are tested in two versions: the original ones and the deslanted ones. To avoid overfitting training is stopped by the early stopping algorithm with patience 5 for IAM and CVL and patience 10 for all others. The concatenated ground-truth text of training and validation-set is used to train the LM of VBS and WBS. The text from the test-set is used to train the LM of token passing because this algorithm has a high running time regarding the dictionary size and therefore is only feasible for the small dictionary created from the test-set.

Further, the test-set is also used for WBS such that a comparison between both algorithms is possible.
The results of the evaluation are shown in Table 4.16. As already shown for best path decoding, deslanting improves the results for Bentham but worsens them for Rats-Protokolle. The same holds true for all other algorithms except token passing, for which the trend is the other way round. The accuracy of WBS depends on the quality of the dictionary. When training the LM with the test-set, WBS always outperforms the other algorithms regarding the CER. This is the setup which can be compared to token passing, because of the same LM source. *Words* mode (i.e. only using a dictionary) is used for WBS while token passing additionally uses N-gram probabilities, which explains why token passing outperforms WBS regarding WER 4 out of 7 times. When training the LM of WBS with the concatenated text of training and validation-set, this algorithm performs worst for all datasets except CVL. As already analyzed, a dictionary with a low OOV word rate must be available when using WBS and the same holds true for token
passing too. Adding a large number of words (e.g. from a word-list of the corresponding language) to the dictionary decreases the OOV rate and therefore increases the decoding accuracy. The text-postprocessing improves the accuracy for CVL, Bentham (deslanted) and Ratsprotokolle (original and deslanted). Again, the high OOV word rate explains why postprocessing fails for SaintGall.

5.4 Comparison to other HTR systems

HTR results from other authors are published for IAM, Bentham and Ratsprotokolle. The results achieved under VBS with postprocessing enabled are

used for the comparison. For Bentham the result of the deslanted dataset is taken because it performs better than the original dataset. rows are sorted by CER.
For IAM the WER is about the same as reported by Graves. Interestingly the CER of Graves is much worse. This can be due to the decoder in use: Graves applies token passing to the RNN output, which achieves good results for the WER but not for the CER. For Bentham, the CER is in between the other two methods, but the WER is better than those methods. The reason is that postprocessing improves the WER for VBS from 16.10% to 13.45%. Bentham has a low OOV word rate of 17.63% which enables the postprocessing to correct many spelling mistakes. Finally, for the Rats-protokolle dataset CER and WER are on the 6*th* out of 8 places. These results show that the proposed HTR system is able to achieve state-of-the-art results

5.5 Summary

This chapter presented the results for the proposed HTR system. Multiple experiments are conducted to analyze the influence of different parts of the system. Contrast normalization increases the accuracy, the same applies for data augmentation via random image transformations. When removing the slant angle of the text, the results are improved for one of the two evaluated datasets. Segmenting a text-line into words shows poor performance for the tested preprocessing methods. Using the trained HTR system instead, it is possible to segment most of the words correctly. MDLSTM outperforms LSTM and IDCN. However, even if IDCN shows a slightly worse CER than MDLSTM,
it makes it possible to build a purely-convolutional ANN for HTR. Choosing the right decoding algorithm for an already trained ANN can improve the accuracy by incorporating information about the language. Text-postprocessing corrects spelling mistakes and learns the character confusion probabilities of the HTR

system. It is able to improve the results if the number of OOV words is low. A comparison to published results from other authors shows that the proposed system is able to achieve state-of-the-art performance.

CHAPTER 6

Conclusion and Future Enhancement

6.1 Conclusion

This thesis has four main contributions. First, a purely convolutional replacement for the RNN is proposed. This architecture uses IDCN and has the potential to make HTR available on more systems and frameworks which do not support RNNs but only CNNs. Next, a method to segment a text-line into words using best path decoding is suggested. This method outperforms other methods which rely on classic image processing without exploiting learning techniques. Further, a decoding algorithm is proposed which uses a dictionary and a LM to constrain the recognized words while allowing arbitrary non-word characters between them to account for numbers or punctuation marks. Finally, different settings, architectures, preprocessing and postprocessing methods are analyzed regarding their influence on the recognition accuracy.

6.2 Future Enhancement

Further preprocessing methods to simplify the task for the classifier will be tested. One promising method is to remove the slope of the text, which results in text lying approximately in horizontal direction. This also enables tighter cropping of the text-lines which yields larger text in the input images. More variability will be added to the data by using further data augmentation methods. Deslanting decreased the recognition performance on the Rats-protokolle dataset. It will be tried to first deslant the images and then add a random slant again to augment the data. This might improve the results for the mentioned dataset in case the decrease in recognition performance was due to removing data variability. Regarding the classifier, the suggested IDCN will be further improved by doing hyper parameter search on parameters such as dilation width and number of layers. The concatenation of all intermediate outputs of the IDCN layers uses a lot of memory. Further experiments will be conducted to identify intermediate outputs which can be ignored or downsampled without affecting the accuracy. The downsampling scheme of an image-pyramid is a possible starting point for these experiments.

The text-postprocessing uses edit operations to produce suggestions which have an edit distance of 1 to the original word. This will be extended to recursively produce suggestions with an edit distance of 2 or more. The number of suggestions grows exponentially with the number of recursions, however the frequency of confusing certain characters differs, therefore a large number of suggestions can safely be ignored. It might also improve the results to condition the insert and delete operation on the surrounding characters.

6.3 Summary

Five datasets are used to evaluate the proposed system. Those datasets include two with modern handwriting, one from around the year 1800, one from the late middle-ages and one from around the year 900. Results from other authors are available for three datasets and are used to compare the proposed HTR system to other systems. The preprocessing methods can be divided into those simplifying the problem for the classifier and into those increasing the size of the dataset. Contrast normalization increases the recognition performance for all datasets, for the Bentham dataset the CER is improved by 1%. Deslanting enhances the results for Bentham (CER decreased by 0.3%) but not for Ratsprotokolle (CER increased by 0.9%). To further improve the accuracy the size of the datasets can be artificially increased by random image transformations. This decreases the CER by 0.5% for Bentham. The IAM dataset is fully annotated on word-level, all other datasets only contain annotation on line-level. This suggests two solutions to tackle the problem: either a line-image is segmented into words or the complete text-line is fed into the classifier which additionally has to learn a word-separation label (i.e. whitespace character). Experiments show that the analyzed word segmentation methods perform poor on historical datasets: segmentation methods are compared by counting the number of lines for which the segmentation yields the correct number of words. The explicit method is able to achieve 47% on this task, while the implicit method achieves 91%. As a by-product of these experiments a word segmentation method using best path decoding is developed.

The evaluated ANN architectures mainly differ in the type of RNN used. LSTM propagates information through a 1D sequence which is aligned with the horizontal axis of the image. MDLSTM extends LSTM by using a 2D sequence, therefore information is also propagated along the vertical axis. A new method using only convolutional operations is suggested in the form of the IDCN and achieves state-of-the-art performance. MDLSTM outperforms the other two RNN types and is therefore used to compare the results to those of other authors. Evaluated on the Bentham dataset, MDLSTM performs 0.1% better than LSTM and LSTM again performs 0.1% better than IDCN regarding CER.

The ANN is trained using the CTC loss and outputs the recognized text in a specific coding scheme. Multiple decoding algorithms are evaluated. Using best path decoding as a baseline, VBS decreases the CER from 7.35% to 7.08% while token passing decreases the WER from 26.29% to 7.88% on the Ratsprotokolle dataset. The integration of LMs especially helps decoding noisy data by incorporating information about language structure.

Finally, the recognized text can be checked for spelling mistakes. A text postprocessing method tries to find the most likely word given a misspelled word. This method is trained on the ANN output and is therefore able to learn common mistakes of the ANN such as confusing the characters “a” and “o”. It is able to decrease the CER by 0.2% on the CVL dataset.

Appendices

Source Code

main.py :

```
import argparse
import json
from typing import Tuple, List
import cv2
import editdistance
from path import Path
from dataloader_iam import DataLoaderIAM, Batch
from model import Model, DecoderType
from preprocessor import Preprocessor
class FilePaths:
    """Filenames and paths to data."""
    fn_char_list = '../model/charList.txt'
    fn_summary = '../model/summary.json'
    fn_corpus = '../data/corpus.txt'
def get_img_height() -> int:
    """Fixed height for NN."""
    return 32
def get_img_size(line_mode: bool = False) -> Tuple[int, int]:
    """Height is fixed for NN, width is set according to training mode (single words
or text lines)."""
    if line_mode:
        return 256, get_img_height()
    return 128, get_img_height()
def write_summary(char_error_rates: List[float], word_accuracies: List[float]) ->
None:
    """Writes training summary file for NN."""
    with open(FilePaths.fn_summary, 'w') as f:
        json.dump({'charErrorRates': char_error_rates, 'wordAccuracies':
word_accuracies}, f)
def train(model: Model,
          loader: DataLoaderIAM,
          line_mode: bool,
          early_stopping: int = 25) -> None:
    """Trains NN."""
    epoch = 0  # number of training epochs since start
    summary_char_error_rates = []
    summary_word_accuracies = []
    preprocessor = Preprocessor(get_img_size(line_mode),
data_augmentation=True, line_mode=line_mode)
    best_char_error_rate = float('inf')  # best valdiation character error rate
    no_improvement_since = 0  # number of epochs no improvement of character
error rate occurred
    # stop training after this number of epochs without improvement
    while True:
        epoch += 1
        print('Epoch:', epoch)
        # train
        print('Train NN')
        loader.train_set()
        while loader.has_next():
            iter_info = loader.get_iterator_info()
            batch = loader.get_next()
            batch = preprocessor.process_batch(batch)
            loss = model.train_batch(batch)
            print(f'Epoch: {epoch} Batch: {iter_info[0]}/{iter_info[1]} Loss: {loss}')
        # validate
        char_error_rate, word_accuracy = validate(model, loader, line_mode)
        # write summary
        summary_char_error_rates.append(char_error_rate)
        summary_word_accuracies.append(word_accuracy)
        write_summary(summary_char_error_rates, summary_word_accuracies)
        # if best validation accuracy so far, save model parameters
        if char_error_rate < best_char_error_rate:
            print('Character error rate improved, save model')
            best_char_error_rate = char_error_rate
            no_improvement_since = 0
            model.save()
        else:
            print(f'Character error rate not improved, best so far: {char_error_rate *
100.0}%')
            no_improvement_since += 1
        # stop training if no more improvement in the last x epochs
        if no_improvement_since >= early_stopping:
            print(f'No more improvement since {early_stopping} epochs. Training
stopped.')
            break
def validate(model: Model, loader: DataLoaderIAM, line_mode: bool) ->
Tuple[float, float]:
    """Validates NN."""
    print('Validate NN')
    loader.validation_set()
    preprocessor = Preprocessor(get_img_size(line_mode), line_mode=line_mode)
    num_char_err = 0
    num_char_total = 0
    num_word_ok = 0
    num_word_total = 0
    while loader.has_next():
        iter_info = loader.get_iterator_info()
        print(f'Batch: {iter_info[0]} / {iter_info[1]}')
        batch = loader.get_next()
        batch = preprocessor.process_batch(batch)
        recognized, _ = model.infer_batch(batch)
        print('Ground truth -> Recognized')
        for i in range(len(recognized)):
            num_word_ok += 1 if batch.gt_texts[i] == recognized[i] else 0
            num_word_total += 1
            dist = editdistance.eval(recognized[i], batch.gt_texts[i])
            num_char_err += dist
            num_char_total += len(batch.gt_texts[i])
            print('[OK]' if dist == 0 else '[ERR:%d]' % dist, '"' + batch.gt_texts[i] + '"',
'->',
                  '"' + recognized[i] + '"')

    # print validation result
    char_error_rate = num_char_err / num_char_total
    word_accuracy = num_word_ok / num_word_total
    print(f'Character error rate: {char_error_rate * 100.0}%. Word accuracy:
{word_accuracy * 100.0}%.')
```

```
    return char_error_rate, word_accuracy
def infer(model: Model, fn_img: Path) -> None:
    """Recognizes text in image provided by file path."""
    img = cv2.imread(fn_img, cv2.IMREAD_GRAYSCALE)
    assert img is not None
    preprocessor = Preprocessor(get_img_size(), dynamic_width=True,
padding=16)
    img = preprocessor.process_img(img)
    batch = Batch([img], None, 1)
    recognized, probability = model.infer_batch(batch, True)
    print(f'Recognized: "{recognized[0]}"')
    print(f'Probability: {probability[0]}')
def main():
    """Main function."""
    parser = argparse.ArgumentParser()
    parser.add_argument('--mode', choices=['train', 'validate', 'infer'], default='infer')
    parser.add_argument('--decoder', choices=['bestpath', 'beamsearch',
'wordbeamsearch'], default='bestpath')
    parser.add_argument('--batch_size', help='Batch size.', type=int, default=100)
    parser.add_argument('--data_dir', help='Directory containing IAM dataset.',
type=Path, required=False)
    parser.add_argument('--fast', help='Load samples from LMDB.',
action='store_true')
    parser.add_argument('--line_mode', help='Train to read text lines instead of
single words.', action='store_true')
    parser.add_argument('--img_file', help='Image used for inference.', type=Path,
default='../data/line.png')
```

```
    parser.add_argument('--early_stopping', help='Early stopping epochs.', type=int,
default=25)
    parser.add_argument('--dump', help='Dump output of NN to CSV file(s).',
action='store_true')
    args = parser.parse_args()
    # set chosen CTC decoder
    decoder_mapping = {'bestpath': DecoderType.BestPath,
                       'beamsearch': DecoderType.BeamSearch,
                       'wordbeamsearch': DecoderType.WordBeamSearch}
    decoder_type = decoder_mapping[args.decoder]
    # train or validate on IAM dataset
    if args.mode in ['train', 'validate']:
        # load training data, create TF model
        loader = DataLoaderIAM(args.data_dir, args.batch_size, fast=args.fast)
        char_list = loader.char_list
        # when in line mode, take care to have a whitespace in the char list
        if args.line_mode and ' ' not in char_list:
            char_list = [' '] + char_list
        # save characters of model for inference mode
        open(FilePaths.fn_char_list, 'w').write(".join(char_list))

        # save words contained in dataset into file
        open(FilePaths.fn_corpus, 'w').write(' '.join(loader.train_words +
loader.validation_words))
        # execute training or validation
        if args.mode == 'train':
            model = Model(char_list, decoder_type)
            train(model, loader, line_mode=args.line_mode,
early_stopping=args.early_stopping)
        elif args.mode == 'validate':
            model = Model(char_list, decoder_type, must_restore=True)
            validate(model, loader, args.line_mode)
    # infer text on test image
    elif args.mode == 'infer':
        model = Model(list(open(FilePaths.fn_char_list).read()), decoder_type,
must_restore=True, dump=args.dump)
        infer(model, args.img_file)
if __name__ == '__main__':
    main()
```

Create_lmdb.py :

```
import argparse
import pickle
import cv2
import lmdb
from path import Path

parser = argparse.ArgumentParser()
parser.add_argument('--data_dir', type=Path, required=True)
args = parser.parse_args()
# 2GB is enough for IAM dataset
assert not (args.data_dir / 'lmdb').exists()
env = lmdb.open(str(args.data_dir / 'lmdb'), map_size=1024 * 1024 * 1024 * 2)
```

```
# go over all png files
fn_imgs = list((args.data_dir / 'img').walkfiles('*.png'))
# and put the imgs into lmdb as pickled grayscale imgs
with env.begin(write=True) as txn:
    for i, fn_img in enumerate(fn_imgs):
        print(i, len(fn_imgs))
        img = cv2.imread(fn_img, cv2.IMREAD_GRAYSCALE)
        basename = fn_img.basename()
        txn.put(basename.encode("ascii"), pickle.dumps(img))
env.close()
```

Dataloader_iam.py :

```
import pickle
import random
from collections import namedtuple
from typing import Tuple
import cv2
import lmdb
import numpy as np
from path import Path

Sample = namedtuple('Sample', 'gt_text, file_path')
Batch = namedtuple('Batch', 'imgs, gt_texts, batch_size')
class DataLoaderIAM:
    """
    Loads data which corresponds to IAM format,
    see: http://www.fki.inf.unibe.ch/databases/iam-handwriting-database
    """
    def __init__(self,
                 data_dir: Path,
                 batch_size: int,
                 data_split: float = 0.95,
                 fast: bool = True) -> None:
        """Loader for dataset."""
        assert data_dir.exists()
        self.fast = fast
        if fast:
            self.env = lmdb.open(str(data_dir / 'lmdb'), readonly=True)
        self.data_augmentation = False
        self.curr_idx = 0
        self.batch_size = batch_size
        self.samples = []
        f = open(data_dir / 'gt/words.txt')
        chars = set()
        bad_samples_reference = ['a01-117-05-02', 'r06-022-03-05']  # known broken
images in IAM dataset
        for line in f:
            # ignore comment line
            if not line or line[0] == '#':
                continue
            line_split = line.strip().split(' ')
            assert len(line_split) >= 9
            # filename: part1-part2-part3 --> part1/part1-part2/part1-part2-part3.png
            file_name_split = line_split[0].split('-')
            file_name_subdir1 = file_name_split[0]
            file_name_subdir2 = f'{file_name_split[0]}-{file_name_split[1]}'
            file_base_name = line_split[0] + '.png'
            file_name = data_dir / 'img' / file_name_subdir1 / file_name_subdir2 /
file_base_name
            if line_split[0] in bad_samples_reference:
                print('Ignoring known broken image:', file_name)
                continue
            # GT text are columns starting at 9
            gt_text = ' '.join(line_split[8:])
            chars = chars.union(set(list(gt_text)))
            # put sample into list
            self.samples.append(Sample(gt_text, file_name))
        # split into training and validation set: 95% - 5%
        split_idx = int(data_split * len(self.samples))
        self.train_samples = self.samples[:split_idx]
        self.validation_samples = self.samples[split_idx:]
        # put words into lists
        self.train_words = [x.gt_text for x in self.train_samples]
        self.validation_words = [x.gt_text for x in self.validation_samples]
        # start with train set
        self.train_set()
        # list of all chars in dataset
        self.char_list = sorted(list(chars))
    def train_set(self) -> None:
        """Switch to randomly chosen subset of training set."""
        self.data_augmentation = True
        self.curr_idx = 0
        random.shuffle(self.train_samples)
        self.samples = self.train_samples
        self.curr_set = 'train'
    def validation_set(self) -> None:
        """Switch to validation set."""
        self.data_augmentation = False
        self.curr_idx = 0
        self.samples = self.validation_samples
        self.curr_set = 'val'
    def get_iterator_info(self) -> Tuple[int, int]:
        """Current batch index and overall number of batches."""
        if self.curr_set == 'train':
            num_batches = int(np.floor(len(self.samples) / self.batch_size))  # train set:
only full-sized batches
        else:
            num_batches = int(np.ceil(len(self.samples) / self.batch_size))  # val set:
allow last batch to be smaller
        curr_batch = self.curr_idx // self.batch_size + 1
        return curr_batch, num_batches

    def has_next(self) -> bool:
        """Is there a next element?"""
        if self.curr_set == 'train':
            return self.curr_idx + self.batch_size <= len(self.samples)  # train set: only
full-sized batches
```

```
        else:
            return self.curr_idx < len(self.samples)  # val set: allow last batch to be
smaller
    def _get_img(self, i: int) -> np.ndarray:
        if self.fast:
            with self.env.begin() as txn:
                basename = Path(self.samples[i].file_path).basename()
                data = txn.get(basename.encode("ascii"))
                img = pickle.loads(data)
        else:
            img = cv2.imread(self.samples[i].file_path, cv2.IMREAD_GRAYSCALE)
        return img
    def get_next(self) -> Batch:
        """Get next element."""
        batch_range = range(self.curr_idx, min(self.curr_idx + self.batch_size,
len(self.samples)))
        imgs = [self._get_img(i) for i in batch_range]
        gt_texts = [self.samples[i].gt_text for i in batch_range]
        self.curr_idx += self.batch_size
        return Batch(imgs, gt_texts, len(imgs))
```

Modal.py :

```
import os
import sys
from typing import List, Tuple
import numpy as np
import tensorflow as tf
```

```
from dataloader_iam import Batch
# Disable eager mode
tf.compat.v1.disable_eager_execution()
class DecoderType:
    """CTC decoder types."""
    BestPath = 0
    BeamSearch = 1
    WordBeamSearch = 2
class Model:
    """Minimalistic TF model for HTR."""
    def __init__(self,
                 char_list: List[str],
                 decoder_type: str = DecoderType.BestPath,
                 must_restore: bool = False,
                 dump: bool = False) -> None:
        """Init model: add CNN, RNN and CTC and initialize TF."""
        self.dump = dump
        self.char_list = char_list
        self.decoder_type = decoder_type
        self.must_restore = must_restore
        self.snap_ID = 0

        # Whether to use normalization over a batch or a population
        self.is_train = tf.compat.v1.placeholder(tf.bool, name='is_train')
        # input image batch
        self.input_imgs = tf.compat.v1.placeholder(tf.float32, shape=(None, None,
None))
```

```
        # setup CNN, RNN and CTC
        self.setup_cnn()
        self.setup_rnn()
        self.setup_ctc()
        # setup optimizer to train NN
        self.batches_trained = 0
        self.update_ops =
tf.compat.v1.get_collection(tf.compat.v1.GraphKeys.UPDATE_OPS)
        with tf.control_dependencies(self.update_ops):
            self.optimizer = tf.compat.v1.train.AdamOptimizer().minimize(self.loss)
        # initialize TF
        self.sess, self.saver = self.setup_tf()
    def setup_cnn(self) -> None:
        """Create CNN layers."""
        cnn_in4d = tf.expand_dims(input=self.input_imgs, axis=3)
        # list of parameters for the layers
        kernel_vals = [5, 5, 3, 3, 3]
        feature_vals = [1, 32, 64, 128, 128, 256]
        stride_vals = pool_vals = [(2, 2), (2, 2), (1, 2), (1, 2), (1, 2)]
        num_layers = len(stride_vals)

        # create layers
        pool = cnn_in4d  # input to first CNN layer
        for i in range(num_layers):
            kernel = tf.Variable(
                tf.random.truncated_normal([kernel_vals[i], kernel_vals[i],
feature_vals[i], feature_vals[i + 1]],
                              stddev=0.1))
            conv = tf.nn.conv2d(input=pool, filters=kernel, padding='SAME',
strides=(1, 1, 1, 1))
            conv_norm = tf.compat.v1.layers.batch_normalization(conv,
training=self.is_train)
            relu = tf.nn.relu(conv_norm)
            pool = tf.nn.max_pool2d(input=relu, ksize=(1, pool_vals[i][0],
pool_vals[i][1], 1),
                                    strides=(1, stride_vals[i][0], stride_vals[i][1], 1),
padding='VALID')
        self.cnn_out_4d = pool
    def setup_rnn(self) -> None:
        """Create RNN layers."""
        rnn_in3d = tf.squeeze(self.cnn_out_4d, axis=[2])
        # basic cells which is used to build RNN
        num_hidden = 256
        cells = [tf.compat.v1.nn.rnn_cell.LSTMCell(num_units=num_hidden,
state_is_tuple=True) for _ in
                 range(2)]  # 2 layers
        # stack basic cells
        stacked = tf.compat.v1.nn.rnn_cell.MultiRNNCell(cells, state_is_tuple=True)
        # bidirectional RNN
        # BxTxF -> BxTx2H
        (fw, bw), _ = tf.compat.v1.nn.bidirectional_dynamic_rnn(cell_fw=stacked,
cell_bw=stacked, inputs=rnn_in3d,
                                                                dtype=rnn_in3d.dtype)
        # BxTxH + BxTxH -> BxTx2H -> BxTx1X2H
        concat = tf.expand_dims(tf.concat([fw, bw], 2), 2)
        # project output to chars (including blank): BxTx1x2H -> BxTx1xC ->
BxTxC
        kernel = tf.Variable(tf.random.truncated_normal([1, 1, num_hidden * 2,
len(self.char_list) + 1], stddev=0.1))
        self.rnn_out_3d = tf.squeeze(tf.nn.atrous_conv2d(value=concat,
filters=kernel, rate=1, padding='SAME'),
                                     axis=[2])
    def setup_ctc(self) -> None:
        """Create CTC loss and decoder."""
        # BxTxC -> TxBxC
        self.ctc_in_3d_tbc = tf.transpose(a=self.rnn_out_3d, perm=[1, 0, 2])
        # ground truth text as sparse tensor
        self.gt_texts = tf.SparseTensor(tf.compat.v1.placeholder(tf.int64,
shape=[None, 2]),
                                        tf.compat.v1.placeholder(tf.int32, [None]),
                                        tf.compat.v1.placeholder(tf.int64, [2]))
        # calc loss for batch
        self.seq_len = tf.compat.v1.placeholder(tf.int32, [None])
        self.loss = tf.reduce_mean(
            input_tensor=tf.compat.v1.nn.ctc_loss(labels=self.gt_texts,
inputs=self.ctc_in_3d_tbc,
                                                  sequence_length=self.seq_len,
                                                  ctc_merge_repeated=True))
        # calc loss for each element to compute label probability
        self.saved_ctc_input = tf.compat.v1.placeholder(tf.float32,
                                                        shape=[None, None, len(self.char_list) + 1])
        self.loss_per_element = tf.compat.v1.nn.ctc_loss(labels=self.gt_texts,
inputs=self.saved_ctc_input,
                                                         sequence_length=self.seq_len,
ctc_merge_repeated=True)
        # best path decoding or beam search decoding
        if self.decoder_type == DecoderType.BestPath:
            self.decoder = tf.nn.ctc_greedy_decoder(inputs=self.ctc_in_3d_tbc,
sequence_length=self.seq_len)
        elif self.decoder_type == DecoderType.BeamSearch:
            self.decoder = tf.nn.ctc_beam_search_decoder(inputs=self.ctc_in_3d_tbc,
sequence_length=self.seq_len,
                                                         beam_width=50)
        # word beam search decoding (see
https://github.com/githubharald/CTCWordBeamSearch)
        elif self.decoder_type == DecoderType.WordBeamSearch:
            # prepare information about language (dictionary, characters in dataset,
characters forming words)
            chars = ''.join(self.char_list)
            word_chars = open('../model/wordCharList.txt').read().splitlines()[0]
            corpus = open('../data/corpus.txt').read()

            # decode using the "Words" mode of word beam search
            from word_beam_search import WordBeamSearch
            self.decoder = WordBeamSearch(50, 'Words', 0.0, corpus.encode('utf8'),
chars.encode('utf8'),
                                          word_chars.encode('utf8'))
            # the input to the decoder must have softmax already applied
            self.wbs_input = tf.nn.softmax(self.ctc_in_3d_tbc, axis=2)
    def setup_tf(self) -> Tuple[tf.compat.v1.Session, tf.compat.v1.train.Saver]:
        """Initialize TF."""
        print('Python: ' + sys.version)
        print('Tensorflow: ' + tf.__version__)
        sess = tf.compat.v1.Session()  # TF session
        saver = tf.compat.v1.train.Saver(max_to_keep=1)  # saver saves model to file
        model_dir = '../model/'
        latest_snapshot = tf.train.latest_checkpoint(model_dir)  # is there a saved
model?
        # if model must be restored (for inference), there must be a snapshot
        if self.must_restore and not latest_snapshot:
            raise Exception('No saved model found in: ' + model_dir)
        # load saved model if available
        if latest_snapshot:
            print('Init with stored values from ' + latest_snapshot)
            saver.restore(sess, latest_snapshot)
        else:
            print('Init with new values')
            sess.run(tf.compat.v1.global_variables_initializer())

        return sess, saver
    def to_sparse(self, texts: List[str]) -> Tuple[List[List[int]], List[int], List[int]]:
        """Put ground truth texts into sparse tensor for ctc_loss."""
        indices = []
        values = []
        shape = [len(texts), 0]  # last entry must be max(labelList[i])
    # go over all texts
    for batchElement, text in enumerate(texts):
        # convert to string of label (i.e. class-ids)
        label_str = [self.char_list.index(c) for c in text]
        # sparse tensor must have size of max. label-string
        if len(label_str) > shape[1]:
            shape[1] = len(label_str)
        # put each label into sparse tensor
        for i, label in enumerate(label_str):
            indices.append([batchElement, i])
            values.append(label)
    return indices, values, shape
def decoder_output_to_text(self, ctc_output: tuple, batch_size: int) -> List[str]:
    """Extract texts from output of CTC decoder."""
    # word beam search: already contains label strings
    if self.decoder_type == DecoderType.WordBeamSearch:
        label_strs = ctc_output
    # TF decoders: label strings are contained in sparse tensor
    else:
        # ctc returns tuple, first element is SparseTensor
        decoded = ctc_output[0][0]
        # contains string of labels for each batch element
        label_strs = [[] for _ in range(batch_size)]
        # go over all indices and save mapping: batch -> values
        for (idx, idx2d) in enumerate(decoded.indices):
            label = decoded.values[idx]
            batch_element = idx2d[0]  # index according to [b,t]
```

```
                label_strs[batch_element].append(label)
        # map labels to chars for all batch elements
        return [''.join([self.char_list[c] for c in labelStr]) for labelStr in label_strs]
    def train_batch(self, batch: Batch) -> float:
        """Feed a batch into the NN to train it."""
        num_batch_elements = len(batch.imgs)
        max_text_len = batch.imgs[0].shape[0] // 4
        sparse = self.to_sparse(batch.gt_texts)
        eval_list = [self.optimizer, self.loss]
        feed_dict = {self.input_imgs: batch.imgs, self.gt_texts: sparse,
                     self.seq_len: [max_text_len] * num_batch_elements, self.is_train:
True}
        _, loss_val = self.sess.run(eval_list, feed_dict)
        self.batches_trained += 1
        return loss_val

    @staticmethod
    def dump_nn_output(rnn_output: np.ndarray) -> None:
        """Dump the output of the NN to CSV file(s)."""
        dump_dir = '../dump/'
        if not os.path.isdir(dump_dir):
            os.mkdir(dump_dir)
        # iterate over all batch elements and create a CSV file for each one
        max_t, max_b, max_c = rnn_output.shape
        for b in range(max_b):
            csv = ''
            for t in range(max_t):
                for c in range(max_c):
                    csv += str(rnn_output[t, b, c]) + ';'
                csv += '\n'
            fn = dump_dir + 'rnnOutput_' + str(b) + '.csv'
            print('Write dump of NN to file: ' + fn)
            with open(fn, 'w') as f:
                f.write(csv)
    def infer_batch(self, batch: Batch, calc_probability: bool = False,
probability_of_gt: bool = False):
        """Feed a batch into the NN to recognize the texts."""
        # decode, optionally save RNN output
        num_batch_elements = len(batch.imgs)
        # put tensors to be evaluated into list
        eval_list = []
        if self.decoder_type == DecoderType.WordBeamSearch:
            eval_list.append(self.wbs_input)
        else:
            eval_list.append(self.decoder)
        if self.dump or calc_probability:
            eval_list.append(self.ctc_in_3d_tbc)
        # sequence length depends on input image size (model downsizes width by 4)
        max_text_len = batch.imgs[0].shape[0] // 4
        # dict containing all tensor fed into the model
        feed_dict = {self.input_imgs: batch.imgs, self.seq_len: [max_text_len] *
num_batch_elements,
                     self.is_train: False}
        # evaluate model
        eval_res = self.sess.run(eval_list, feed_dict
        # TF decoders: decoding already done in TF graph
        if self.decoder_type != DecoderType.WordBeamSearch:
            decoded = eval_res[0]
        # word beam search decoder: decoding is done in C++ function compute()
        else:
            decoded = self.decoder.compute(eval_res[0])
        # map labels (numbers) to character string
        texts = self.decoder_output_to_text(decoded, num_batch_elements)
        # feed RNN output and recognized text into CTC loss to compute labeling
probability
        probs = None
        if calc_probability:
            sparse = self.to_sparse(batch.gt_texts) if probability_of_gt else
self.to_sparse(texts)
            ctc_input = eval_res[1]
            eval_list = self.loss_per_element
            feed_dict = {self.saved_ctc_input: ctc_input, self.gt_texts: sparse,
                         self.seq_len: [max_text_len] * num_batch_elements, self.is_train:
False}
            loss_vals = self.sess.run(eval_list, feed_dict)
            probs = np.exp(-loss_vals)
        # dump the output of the NN to CSV file(s)
        if self.dump:
            self.dump_nn_output(eval_res[1])
        return texts, probs
    def save(self) -> None:
        """Save model to file."""
        self.snap_ID += 1
        self.saver.save(self.sess, '../model/snapshot', global_step=self.snap_ID)
```

Preprocessor.py :

```
import random
from typing import Tuple
import cv2
import numpy as np
from dataloader_iam import Batch
class Preprocessor:
    def __init__(self,
                 img_size: Tuple[int, int],
                 padding: int = 0,
                 dynamic_width: bool = False,
                 data_augmentation: bool = False,
                 line_mode: bool = False) -> None:
        # dynamic width only supported when no data augmentation happens
        assert not (dynamic_width and data_augmentation)
        # when padding is on, we need dynamic width enabled
        assert not (padding > 0 and not dynamic_width)
        self.img_size = img_size
        self.padding = padding
        self.dynamic_width = dynamic_width
        self.data_augmentation = data_augmentation
        self.line_mode = line_mode
    @staticmethod
    def _truncate_label(text: str, max_text_len: int) -> str:
        """
        Function ctc_loss can't compute loss if it cannot find a mapping between text
label and input
        labels. Repeat letters cost double because of the blank symbol needing to be
inserted.
        If a too-long label is provided, ctc_loss returns an infinite gradient.
        """
        cost = 0
        for i in range(len(text)):
            if i != 0 and text[i] == text[i - 1]:
                cost += 2
            else:
                cost += 1
            if cost > max_text_len:
                return text[:i]
        return text
    def _simulate_text_line(self, batch: Batch) -> Batch:
        """Create image of a text line by pasting multiple word images into an
image."""
        default_word_sep = 30
        default_num_words = 5
        # go over all batch elements
        res_imgs = []
        res_gt_texts = []
        for i in range(batch.batch_size):
            # number of words to put into current line
            num_words = random.randint(1, 8) if self.data_augmentation else
default_num_words
            # concat ground truth texts
            curr_gt = ' '.join([batch.gt_texts[(i + j) % batch.batch_size] for j in
range(num_words)])
            res_gt_texts.append(curr_gt)
            # put selected word images into list, compute target image size
            sel_imgs = []
            word_seps = [0]
            h = 0
            w = 0
            for j in range(num_words):
                curr_sel_img = batch.imgs[(i + j) % batch.batch_size]
                curr_word_sep = random.randint(20, 50) if self.data_augmentation else
default_word_sep
                h = max(h, curr_sel_img.shape[0])
                w += curr_sel_img.shape[1]
                sel_imgs.append(curr_sel_img)
                if j + 1 < num_words:
                    w += curr_word_sep
                    word_seps.append(curr_word_sep)
            # put all selected word images into target image
            target = np.ones([h, w], np.uint8) * 255
            x = 0
            for curr_sel_img, curr_word_sep in zip(sel_imgs, word_seps):
                x += curr_word_sep
                y = (h - curr_sel_img.shape[0]) // 2
```

```
                target[y:y + curr_sel_img.shape[0]:, x:x + curr_sel_img.shape[1]] =
curr_sel_img
                x += curr_sel_img.shape[1]

            # put image of line into result
            res_imgs.append(target)
        return Batch(res_imgs, res_gt_texts, batch.batch_size)
    def process_img(self, img: np.ndarray) -> np.ndarray:
        """Resize to target size, apply data augmentation."""
        # there are damaged files in IAM dataset - just use black image instead
        if img is None:
            img = np.zeros(self.img_size[::-1])
        # data augmentation
        img = img.astype(np.float)
        if self.data_augmentation:
            # photometric data augmentation
            if random.random() < 0.25:
                def rand_odd():
                    return random.randint(1, 3) * 2 + 1
                img = cv2.GaussianBlur(img, (rand_odd(), rand_odd()), 0)
            if random.random() < 0.25:
                img = cv2.dilate(img, np.ones((3, 3)))
            if random.random() < 0.25:
                img = cv2.erode(img, np.ones((3, 3)))
            # geometric data augmentation
            wt, ht = self.img_size
            h, w = img.shape
            f = min(wt / w, ht / h)
            fx = f * np.random.uniform(0.75, 1.05)
            fy = f * np.random.uniform(0.75, 1.05)
            # random position around center
            txc = (wt - w * fx) / 2
            tyc = (ht - h * fy) / 2
            freedom_x = max((wt - fx * w) / 2, 0)
            freedom_y = max((ht - fy * h) / 2, 0)
            tx = txc + np.random.uniform(-freedom_x, freedom_x)
            ty = tyc + np.random.uniform(-freedom_y, freedom_y)
            # map image into target image
            M = np.float32([[fx, 0, tx], [0, fy, ty]])
            target = np.ones(self.img_size[::-1]) * 255
            img = cv2.warpAffine(img, M, dsize=self.img_size, dst=target,
borderMode=cv2.BORDER_TRANSPARENT)
            # photometric data augmentation
            if random.random() < 0.5:
                img = img * (0.25 + random.random() * 0.75)
            if random.random() < 0.25:
                img = np.clip(img + (np.random.random(img.shape) - 0.5) *
random.randint(1, 25), 0, 255)
            if random.random() < 0.1:
                img = 255 - img
        # no data augmentation
        else:
            if self.dynamic_width:
                ht = self.img_size[1]
                h, w = img.shape
                f = ht / h
                wt = int(f * w + self.padding)
                wt = wt + (4 - wt) % 4
                tx = (wt - w * f) / 2
                ty = 0
            else:
                wt, ht = self.img_size
                h, w = img.shape
                f = min(wt / w, ht / h)
                tx = (wt - w * f) / 2
                ty = (ht - h * f) / 2
            # map image into target image
            M = np.float32([[f, 0, tx], [0, f, ty]])
            target = np.ones([ht, wt]) * 255
            img = cv2.warpAffine(img, M, dsize=(wt, ht), dst=target,
borderMode=cv2.BORDER_TRANSPARENT)
        # transpose for TF
        img = cv2.transpose(img)
        # convert to range [-1, 1]
        img = img / 255 - 0.5
        return img
      def process_batch(self, batch: Batch) -> Batch:
        if self.line_mode:
            batch = self._simulate_text_line(batch)
        res_imgs = [self.process_img(img) for img in batch.imgs]
        max_text_len = res_imgs[0].shape[0] // 4
```

```
        res_gt_texts = [self._truncate_label(gt_text, max_text_len) for gt_text in
batch.gt_texts]
        return Batch(res_imgs, res_gt_texts, batch.batch_size)
def main():
    import matplotlib.pyplot as plt
    img = cv2.imread('../data/test.png', cv2.IMREAD_GRAYSCALE)
    img_aug = Preprocessor((256, 32), data_augmentation=True).process_img(img)
    plt.subplot(121)
    plt.imshow(img, cmap='gray')
    plt.subplot(122)
    plt.imshow(cv2.transpose(img_aug) + 0.5, cmap='gray', vmin=0, vmax=1)
    plt.show()
if __name__ == '__main__':
    main()
```