

A Project Report on

Text Handwriting Conversion using Neural Networks

Submitted in partial fulfillment of the requirements for the award
of the degree of

Bachelor of Engineering

in

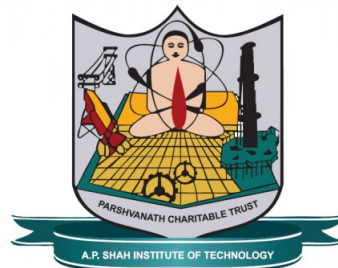
Information Technology

by

Aditya Saini(18104046)
Kunal Sant(18104011)
Sumeet Swain(18104008)

Under the Guidance of

Prof. Neha Deshmukh



Department of Information Technology

NBA Accredited

A.P. Shah Institute of Technology
G.B.Road,Kasarvadavli, Thane(W), Mumbai-400615
UNIVERSITY OF MUMBAI
Academic Year 2021-2022

Approval Sheet

This Project Report entitled “*Text Handwriting Conversion using Neural Networks*” Submitted by “*Aditya Saini*”(18104046), “*Kunal Sant*”(18104011), “*Sumeet Swain*”(18104008) is approved for the partial fulfillment of the requirement for the award of the degree of *Bachelor of Engineering in Information Technology* from *University of Mumbai*.

Prof. Neha Deshmukh
Guide

Prof. Kiran Deshpande
Head of Department of Information Technology

Place: A.P. Shah Institute of Technology, Thane
Date:

CERTIFICATE

This is to certify that the project entitled “*Text Handwriting Conversion using Neural Networks*” submitted by “*Aditya Saini*”(18104046), “*Kunal Sant*”(18104011), “*Sumeet Swain*”(18104008) for the partial fulfillment of the requirement for award of a degree *Bachelor of Engineering* in *Information Technology*, to the University of Mumbai, is a bonafide work carried out during academic year 2020-2021.

Prof. Neha Deshmukh
Guide

Prof. Kiran Deshpande
Head Department of Information Technology

Dr. Uttam D.Kolekar
Principal

External Examiner(s)

1.

2.

Place: A.P. Shah Institute of Technology, Thane

Date:

Acknowledgement

We have great pleasure in presenting the report on **Text Handwriting Conversion using Neural Networks**. We take this opportunity to express our sincere thanks towards our guide **Prof. Neha Deshmukh**, Department of IT, APSIT thane for providing the technical guidelines and suggestions regarding line of work. We would like to express our gratitude towards his constant encouragement, support and guidance through the development of project.

We thank **Prof. Kiran B. Deshpande** Head of Department, IT, APSIT for his encouragement during progress meeting and providing guidelines to write this report.

We thank **Prof. Vishal S. Badgujar** BE project co-ordinator, Department of IT, APSIT for being encouraging throughout the course and for guidance.

We also thank the entire staff of APSIT for their invaluable help rendered during the course of this work. We wish to express our deep gratitude towards all our colleagues of APSIT for their encouragement.

Student Name1: Aditya Saini
Student ID1: 18104046

Student Name2: Kunal Sant
Student ID2: 18104011

Student Name3: Sumeet Swain
Student ID3: 18104008

Declaration

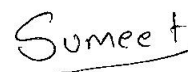
We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, We have adequately cited and referenced the original sources. We also declare that We have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.



(Aditya Saini 18104046)



(Kunal Sant 18104011)



(Sumeet Swain 18104008)

Date:

Abstract

In the current world of automation everything is getting atomized as reducing manual labor is the key to efficiency. This paper focuses on Off-line Handwriting Recognition. Handwriting Recognition is the process of extracting text from handwritten scripts, this is also known as Offline Handwriting Recognition. The purpose of this paper is to attempt to improve the accuracy and efficiency of the system using Neural Networks and datasets used for training the model, as well as detecting and identifying the characters and exporting them in a text format. The proposed system can be used to recognize handwritten characters and convert them into text from the scanned image of a page.

Contents

1	Introduction	1
2	Literature Review	2
3	Project Design	5
4	Project Implementation	7
4.1	Code Snippets	7
4.1.1	Handwriting to Text conversion code	9
4.1.2	Text to Handwriting Conversion	25
5	Testing	31
5.1	Unit Testing	31
5.2	Integration Testing	31
5.3	Compatibility Testing	31
6	Result	32
7	Conclusions and Future Scope	36
	Bibliography	37
7.1	Publication	39

List of Figures

6.1	Input 1	33
6.2	Result 1	33
6.3	Input 2	33
6.4	Result 2	33
6.5	Input 3	33
6.6	Result 3	33
6.7	Text to Handwriting Example 1	34
6.8	Text to Handwriting Example 2	35

List of Abbreviations

CNN:	Convolutional Neural Network
RNN:	Recurrent Neural Network
CTC:	Connectionist Temporal Classification
OCR:	Optical Character Recognition
LSTM:	Long Short Term Memory

Chapter 1

Introduction

Handwriting Recognition is a subject which exists for a long period of time but the accuracy of the process can be improved by the usage of more refined technology and better techniques which reduce the error rates and the difference between the actual text and the detected text of the system. Handwriting Recognition is the process of extracting text from handwritten scripts, this is also known as Offline Handwriting Recognition. In Offline Handwriting Recognition the image of a script is captured using a scanner and sent forward for further processing. This is a step into the world of automation and can help us improve efficiency in this field. Handwriting Recognition works on similar techniques of text detection and recognition. Text detection includes various techniques to extract text from an image such as Texture Detection method and Connected Component method. In texture detection the text is extracted which is detected by texture difference from the background and text is registered by further processing of the information. In the Connected Component method text is extracted using the relation between pixel connectivity and pixel intensity to detect text patterns and checks are made to clear non character elements. The next step in extracting text from an image is Text Recognition which is done by OCR (Optical Character Recognition) which depends on identifying the character from the detected text which can be done by training the model using dataset which gives the model the basic characteristics of characters like the height, width, shape and size and font style. This includes all the alphanumeric characters 10 digits (0-9) and 26 alphabets with uppercase and lowercase for a total of 62 characters. Handwritten characters are more challenging to recognize as compared to printed characters as they carry more parameters like the standard and cursive style of writing. These challenges can be overcome by using new technologies like LSTMs, RNNs, CNNs and different training methods with various datasets. RNNs can be trained for sequence generation by processing real data sequences one step at a time and predicting what comes next. Assuming the predictions are probabilistic, novel sequences can be generated from a trained network by iteratively sampling from the network's output distribution, then feeding in the sample as input at the next step. In other words by making the network treat its inventions as if they were real, much like a person dreaming. Although the network itself is deterministic, the stochasticity injected by picking samples induces a distribution over sequences. This distribution is conditional, since the internal state of the network, and hence its predictive distribution, depends on the previous inputs.

Chapter 2

Literature Review

- In paper[1] Feature Set Evaluation for Offline Handwriting Recognition Systems: Application to the Recurrent Neural Network Model,we studied that the goal of features is to remove unnecessary variability, in the form of individual writing style, from a word image,their use goes from word-spotting, where the goal is to retrieve specific keywords within a document, to word recognition, where a document is converted into a symbolic character string. Nevertheless, feature design and keep only the information relevant for recognition.
- In paper [2] An Off-Line Cursive Handwriting Recognition System Enhancements, we studied that in normalization and in the detection and representation of features have led to reduced error rates. A hybrid system using recurrent neural networks and HMMs was found to perform better than a discrete probability HMM system, The system performance was increased by allowing more frames of context in the network inputs and by “untying” the output distributions, to distinguish between the different parts of each character,the error rate has been reduced to 9.8 percent
- In paper[3] Diagonal Based Feature Extraction For Handwritten Character Recognition System Using Neural Network. we learned the use of techniques such as preprocessing, segmentation, and feature extraction that are used to improve the accuracy of text recognition and identification. The first important step in any handwritten recognition system is pre-processing followed by segmentation and feature extraction.
- In paper[4] Text Detection and Recognition Using Enhanced MSER Detection and a Novel OCR Technique, we studied the work of different text detection methods on how they extract text from an image such as MSER and canny edge integration, stroke width variation and OCR for text recognition. The term “extremal region” represents connected component which differentiate the higher or lower intensity of a pixel to the outer boundary pixel. It is also difficult matter that how to group character candidates into text candidates. For the purpose, generally two types of approaches are used: rule based, clustering based.
- In paper[5] Combination of CNN and LSTM showing a better performance for Offline Handwriting Recognition , we studied that it shows the benefits of combining HMM, LSTM, CNN in visual recognition and performance

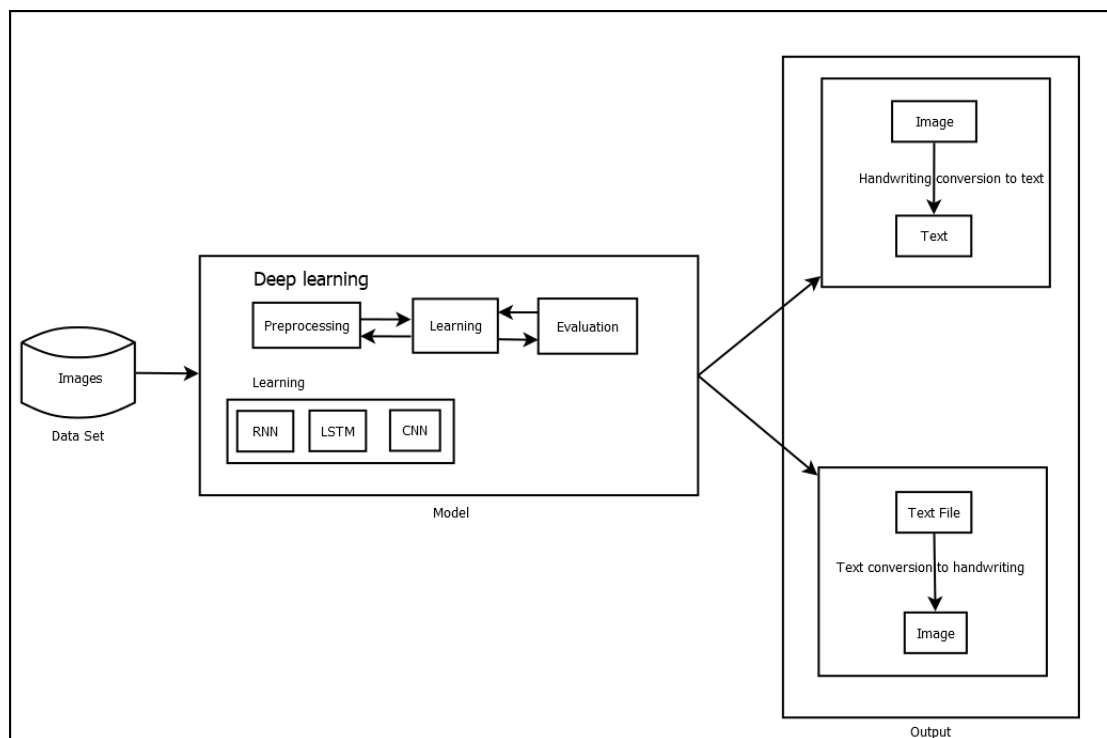
- In paper[6] Convolutional Neural Network Committees For Handwritten Character Classification, we studied that how using CNNs and their training with GPUs to train models can reduce character recognition error rates in offline handwriting. We train five differently initialized nets on each preprocessed dataset as well as on the original data, for a total of 35 CNNs (Table II). Each CNN is trained for 30 epochs by on-line gradient descent. The number of epochs is limited due to the size of NIST SD 19: training a single net for the 62 class problem takes almost six hours
- In paper[7] Fast and robust training of recurrent neural networks for offline handwriting recognition, we studied that LSTM-RNN with Backpropagation Through Time training can be used to speed up the process while increasing the accuracy of recognition process. When recognizing handwritten text, especially the long short-term memory recurrent neural network LSTM-RNN is used in many modern handwriting recognition systems.
- In paper[8] HMM based online handwriting recognition, we studied that in handwriting recognition using hidden Markov modeling and stochastic language modeling methods originally developed for speech applications. These methods are generalized in the AEGIS architecture. Subcharacter models called nebulous stroke models are used to model the basic units in handwriting
- In paper[9] Generating Sequences With Recurrent Neural Networks, we studied that Long Short-Term Memory recurrent neural networks generate both discrete and real-valued sequences with complex, long-range structure using next-step prediction
- In paper[10] The State of the Art in Online Handwriting Recognition, we studied about pre-processing, segmentation, stroke width variation, shape recognition, post-processing, optical character recognition
- In paper[11] Machine Printed Text and Handwriting Identification in Noisy Document Images, we studied that the technique can be used for image enhancement to improve page segmentation accuracy of noisy documents. After noise identification and removal, the zone segmentation accuracy increase from 53 percent to 78 percent using the Docstrum algorithm, segmenting and identifying text from extremely noisy document images. Instead of using simple filtering rules, we treat noise as a distinct class and use statistical classification techniques to classify each block into machine printed text, handwriting, and noise
- In paper[12] A novel connectionist system for unconstrained handwriting recognition, we studied the key features of the network are the BLSTM architecture, which provides access to long-range bidirectional contextual information, and the CTC output layer, which allows the network to be trained on unsegmented sequence data, the new approach outperformed a state-of-the-art HMM-based system and also proved more robust to changes in dictionary size
- In paper [13] Online Handwriting Recognition with Support Vector Machines - A Kernel Approach. they used a technique that combines dynamic time warping (DTW) and

support vector machines (SVM) by integrating DTW into a Gaussian SVM kernel. The benefit of this approach is the absence of restrictive assumptions about class conditional densities, as made in conventional HMM based techniques. The only essential assumption made is the selection of the kernel. A problem of this approach is the complexity.

- In paper [14] full English sentence database for off-line handwriting recognition. we understood that it is potentially useful for recognition of general unconstrained English text utilizing knowledge beyond the lexicon level. Their primary aim was to aid in automatic labeling of the database, but they can be integrated in any recognition system as well.
- In paper [15] Methods of combining multiple classifiers and their applications to handwriting recognition, we studied that the combination of several independent classifiers is a general problem that occurs in various application areas of pattern recognition and the experimental results on the recognition of totally unconstrained handwritten numerals have shown that the performances of individual classifiers could be improved significantly by the combination approaches.

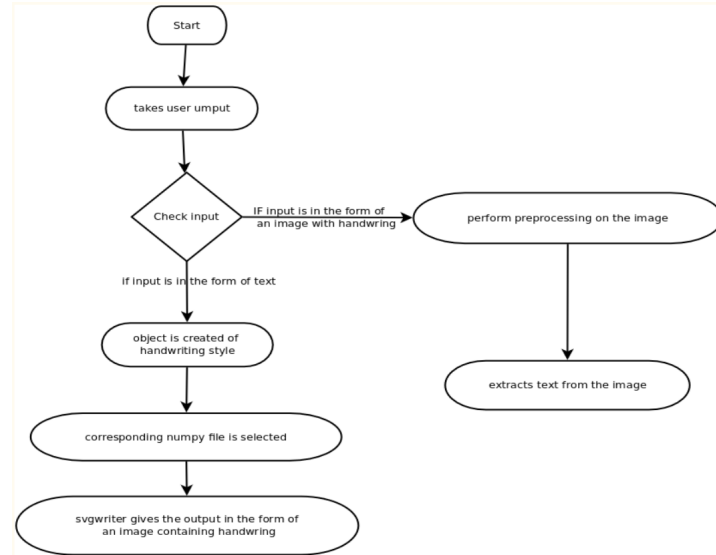
Chapter 3

Project Design



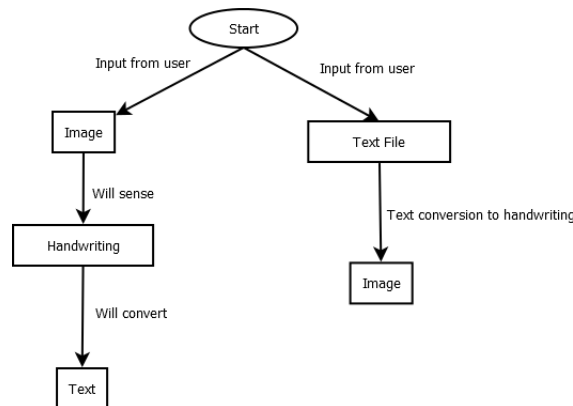
Proposed System Architecture

In our model we have use a data set of hand writings.We have used CNN, RNN, CTC and LSTM. In our project we will push the image into the CNN layer and extract the features of that image and pass them to the RNN layer which will provide us with a matrix which we will pass to the CTC layer.



Activity diagram

In general, handwriting recognition is classified into two types as off-line and on-line handwriting recognition methods. In the off-line recognition, the writing is usually captured optically by a scanner and the completed writing is available as an image. A typical handwriting recognition system consists of pre-processing, segmentation, feature extraction, classification and recognition, and post processing stages. After the required pre-processing we will pass the handwriting into the model which will return digitised text of the handwriting. If the user text as an input we will scan the text file and transfer the text to the model which will then create an object containing text styles. This object defines what style the hand writhing will be and so the model return an image containing generated handwriting.



Workflow

Off-line and on-line handwriting recognition technologies are the two most common types of handwriting recognition. The writing is normally taken optically by a scanner in off-line recognition, and the completed writing is available as an image. Pre-processing, segmentation, feature extraction, classification and recognition, and post-processing are all stages in a typical handwriting recognition system. We'll feed the handwriting into the model, which will provide digitised text of the handwriting after the necessary pre-processing. If the user provides text as input, we will scan the file and send the data to the model, which will build an object with text styles. This object specifies the handwriting style, and the model returns an image of the created handwriting.

Chapter 4

Project Implementation

4.1 Code Snippets

Importing libraries

```
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

import os
import sys
from typing import List, Tuple

import numpy as np
import tensorflow as tf

from dataloader_iam import Batch
```



```

import random
from typing import Tuple

import cv2
import numpy as np

from dataloader iam import Batch

from __future__ import print_function
import os
from xml.etree import ElementTree

import numpy as np
import drawing

import os
import logging

import numpy as np
import svgwrite

import drawing
# import lyrics
from rnn import rnn

```

4.1.1 Handwriting to Text conversion code

Defining size of input image

```
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)
```

Training Neural Network

```
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 validation 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)
```

Validating Neural Network

```
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(f'[OK]' if dist == 0 else f'[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
```

Recognizing the text in the image provided

```
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]}')
```

The main function

```
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/word.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()
```


Init model for CNN, RNN and CTC

```
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()
```

Creating CNN layers

```
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)
```

Creating CTC layer

```
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)
```

Initializing Tensorflow

```
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
```


Extracting text form CTC, Feeding that into NN for training, Dumping output

```
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
```

Feeding batch into NN for text recognition

```
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)
```

Pre-processing

```
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
```

Creating image of a text line by line

```
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)
```


Resizing to target size and applying data augmentation

```
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
```

```

# 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()

```

```
def get_stroke_sequence(filename):  
    tree = ElementTree.parse(filename).getroot()  
    strokes = [i for i in tree if i.tag == 'StrokeSet'][0]  
  
    coords = []  
    for stroke in strokes:  
        for i, point in enumerate(stroke):  
            coords.append([  
                int(point.attrib['x']),  
                -1*int(point.attrib['y']),  
                int(i == len(stroke) - 1)  
            ])  
    coords = np.array(coords)  
  
    coords = drawing.align(coords)  
    coords = drawing.denoise(coords)  
    offsets = drawing.coords_to_offsets(coords)  
    offsets = offsets[:drawing.MAX_STROKE_LEN]  
    offsets = drawing.normalize(offsets)  
    return offsets  
  
def get_ascii_sequences(filename):  
    sequences = open(filename, 'r').read()  
    sequences = sequences.replace(r'%\n%', '\n')  
    sequences = [i.strip() for i in sequences.split('\n')]  
    lines = sequences[sequences.index('CSR:') + 2:]  
    lines = [line.strip() for line in lines if line.strip()]  
    lines = [drawing.encode_ascii(line)[:drawing.MAX_CHAR_LEN] for line in lines]  
    return lines
```

Collecting the data

```
def collect_data():
    fnames = []
    for dirpath, dirnames, filenames in os.walk('data/raw/ascii/'):
        if dirnames:
            continue
        for filename in filenames:
            if filename.startswith('.'):
                continue
            fnames.append(os.path.join(dirpath, filename))

    # low quality samples (selected by collecting samples to
    # which the trained model assigned very low likelihood)
    blacklist = set(np.load('data/blacklist.npy'))

    stroke_fnames, transcriptions, writer_ids = [], [], []
    for i, fname in enumerate(fnames):
        print(i, fname)
        if fname == 'data/raw/ascii/z01/z01-000/z01-000z.txt':
            continue

        head, tail = os.path.split(fname)
        last_letter = os.path.splitext(fname)[0][-1]
        last_letter = last_letter if last_letter.isalpha() else ''

        line_stroke_dir = head.replace('ascii', 'lineStrokes')
        line_stroke_fname_prefix = os.path.split(head)[-1] + last_letter + '-'

        if not os.path.isdir(line_stroke_dir):
            continue
        line_stroke_fnames = sorted([f for f in os.listdir(line_stroke_dir)
                                     if f.startswith(line_stroke_fname_prefix)])
        if not line_stroke_fnames:
            continue

        original_dir = head.replace('ascii', 'original')
        original_xml = os.path.join(original_dir, 'strokes' + last_letter + '.xml')
        tree = ElementTree.parse(original_xml)
        root = tree.getroot()

        general = root.find('General')
        if general is not None:
            writer_id = int(general[0].attrib.get('writerID', '0'))
        else:
            writer_id = int('0')

        ascii_sequences = get_ascii_sequences(fname)
        assert len(ascii_sequences) == len(line_stroke_fnames)

        for ascii_seq, line_stroke_fname in zip(ascii_sequences, line_stroke_fnames):
            if line_stroke_fname in blacklist:
                continue

            stroke_fnames.append(os.path.join(line_stroke_dir, line_stroke_fname))
            transcriptions.append(ascii_seq)
            writer_ids.append(writer_id)

    return stroke_fnames, transcriptions, writer_ids
```



```

if __name__ == '__main__':
    print('traversing data directory...')
    stroke_fnames, transcriptions, writer_ids = collect_data()

    print('dumping to numpy arrays...')
    x = np.zeros([len(stroke_fnames), drawing.MAX_STROKE_LEN, 3], dtype=np.float32)
    x_len = np.zeros([len(stroke_fnames)], dtype=np.int16)
    c = np.zeros([len(stroke_fnames), drawing.MAX_CHAR_LEN], dtype=np.int8)
    c_len = np.zeros([len(stroke_fnames)], dtype=np.int8)
    w_id = np.zeros([len(stroke_fnames)], dtype=np.int16)
    valid_mask = np.zeros([len(stroke_fnames)], dtype=np.bool)

    for i, (stroke_fname, c_i, w_id_i) in enumerate(zip(stroke_fnames, transcriptions, writer_ids)):
        if i % 200 == 0:
            print(i, '\t', '/', len(stroke_fnames))
            x_i = get_stroke_sequence(stroke_fname)
            valid_mask[i] = ~np.any(np.linalg.norm(x_i[:, :2], axis=1) > 60)

            x[i, :len(x_i), :] = x_i
            x_len[i] = len(x_i)

            c[i, :len(c_i)] = c_i
            c_len[i] = len(c_i)

            w_id[i] = w_id_i

    if not os.path.isdir('data/processed'):
        os.makedirs('data/processed')

    np.save('data/processed/x.npy', x[valid_mask])
    np.save('data/processed/x_len.npy', x_len[valid_mask])
    np.save('data/processed/c.npy', c[valid_mask])
    np.save('data/processed/c_len.npy', c_len[valid_mask])
    np.save('data/processed/w_id.npy', w_id[valid_mask])

```

4.1.2 Text to Handwriting Conversion

Hand class for the recreation of handwriting

```
class Hand(object):

    def __init__(self):
        os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
        self.nn = rnn(
            log_dir='logs',
            checkpoint_dir='checkpoints',
            prediction_dir='predictions',
            learning_rates=[.0001, .00005, .00002],
            batch_sizes=[32, 64, 64],
            patiences=[1500, 1000, 500],
            beta1_decays=[.9, .9, .9],
            validation_batch_size=32,
            optimizer='rms',
            num_training_steps=100000,
            warm_start_init_step=17900,
            regularization_constant=0.0,
            keep_prob=1.0,
            enable_parameter_averaging=False,
            min_steps_to_checkpoint=2000,
            log_interval=20,
            logging_level=logging.CRITICAL,
            grad_clip=10,
            lstm_size=400,
            output_mixture_components=20,
            attention_mixture_components=10
        )
        self.nn.restore()

    def write(self, filename, lines, biases=None, styles=None, stroke_colors=None, stroke_widths=None):
        valid_char_set = set(drawing.alphabet)
        for line_num, line in enumerate(lines):
            if len(line) > 75:
                raise ValueError(
                    (
                        "Each line must be at most 75 characters. "
                        "Line {} contains {}".format(line_num, len(line))
                    )
                )
```

```

def _sample(self, lines, biases=None, styles=None):
    num_samples = len(lines)
    max_tsteps = 40*max([len(i) for i in lines])
    biases = biases if biases is not None else [0.5]*num_samples

    x_prime = np.zeros([num_samples, 1200, 3])
    x_prime_len = np.zeros([num_samples])
    chars = np.zeros([num_samples, 120])
    chars_len = np.zeros([num_samples])

    if styles is not None:
        for i, (cs, style) in enumerate(zip(lines, styles)):
            x_p = np.load('styles/style-{}-strokes.npy'.format(style))
            c_p = np.load('styles/style-{}-chars.npy'.format(style)).tostring().decode('utf-8')

            c_p = str(c_p) + " " + cs
            c_p = drawing.encode_ascii(c_p)
            c_p = np.array(c_p)

            x_prime[i, :len(x_p), :] = x_p
            x_prime_len[i] = len(x_p)
            chars[i, :len(c_p)] = c_p
            chars_len[i] = len(c_p)

    else:
        for i in range(num_samples):
            encoded = drawing.encode_ascii(lines[i])
            chars[i, :len(encoded)] = encoded
            chars_len[i] = len(encoded)

    [samples] = self.nn.session.run(
        [self.nn.sampled_sequence],
        feed_dict={
            self.nn.prime: styles is not None,
            self.nn.x_prime: x_prime,
            self.nn.x_prime_len: x_prime_len,
            self.nn.num_samples: num_samples,
            self.nn.sample_tsteps: max_tsteps,
            self.nn.c: chars,
            self.nn.c_len: chars_len,
            self.nn.bias: biases
        }
    )
    samples = [sample[~np.all(sample == 0.0, axis=1)] for sample in samples]
    return samples

```

Demo output code for handwriting to text conversion

```
def _draw(self, strokes, lines, filename, stroke_colors=None, stroke_widths=None):
    stroke_colors = stroke_colors or ['black']*len(lines)
    stroke_widths = stroke_widths or [2]*len(lines)

    line_height = 60
    view_width = 1000
    view_height = line_height*(len(strokes) + 1)

    dwg = svgwrite.Drawing(filename=filename)
    dwg.viewbox(width=view_width, height=view_height)
    dwg.add(dwg.rect(insert=(0, 0), size=(view_width, view_height), fill='white'))

    initial_coord = np.array([0, -(3*line_height / 4)])
    for offsets, line, color, width in zip(strokes, lines, stroke_colors, stroke_widths):
        if not line:
            initial_coord[1] -= line_height
            continue

        offsets[:, :2] *= 1.5
        strokes = drawing.offsets_to_coords(offsets)
        strokes = drawing.denoise(strokes)
        strokes[:, :2] = drawing.align(strokes[:, :2])

        strokes[:, 1] *= -1
        strokes[:, :2] -= strokes[:, :2].min() + initial_coord
        strokes[:, 0] += (view_width - strokes[:, 0].max()) / 2

        prev_eos = 1.0
        p = "M{},{ } ".format(0, 0)
        for x, y, eos in zip(*strokes.T):
            p += '{},{ } '.format('M' if prev_eos == 1.0 else 'L', x, y)
            prev_eos = eos
        path = svgwrite.path.Path(p)
        path = path.stroke(color=color, width=width, linecap='round').fill("none")
        dwg.add(path)

        initial_coord[1] -= line_height

    dwg.save()
```

Demo code for text to handwriting conversion

```
if __name__ == '__main__':  
    hand = Hand()  
    lines = [  
        "Live Demo for presentation"  
    ]  
    biases = [2 for i in lines]  
    stroke_colors = ['black' for i in lines]  
    stroke_widths = [1]  
    styles=[2]  
    filename="img/demo_live_4.svg"  
    hand.write(  
        filename=filename,  
        lines=lines,  
        biases=biases,  
        styles=styles,  
        stroke_colors=stroke_colors,  
        stroke_widths=stroke_widths  
    )
```

Handwriting to text: CNN: Input from the user is first inserted into CNN layers. CNN layers are used to extract the features from the image. Five input layers are used, of which the first two layers apply a filter of 5×5 and the last three of 3×3 for convolution operations. After the input passes through the five layers of CNN, a non-linear RELU function is applied. Finally, a layer is used to summarise image regions, and a downsized output is generated. Even though the image is downsized by 2 in each layer, a feature map is added, which has a size of 32×256 . RNN: The feature sequence contains 256 features per time step; the relevant information is passed on to the sequence by the RNN. The popular Long Short-Term Memory (LSTM) implementation of RNNs is used, as it is able to pass on information over longer distances and provides more powerful training characteristics than vanilla RNN. In a matrix of size 32×38 , the RNN output sequence is mapped. A total of 79 different characters are contained in the IAM dataset further, one additional character is needed for the CTC operation; therefore, there are 80 entries for each of the 32-time steps. Connectionist Temporal Classification (CTC): While training the Neural Network, the CTC is given to the RNN output matrix and the ground truth text, which computes the loss value. While inferring, the CTC is only given the matrix, and from that information, it decodes it into the final text. The maximum length of ground truth and recognized text is 32 characters.

Input Data: Input data is an image of size 128×32 . Images in the dataset do not have this exact size, so we resize it to either a width of 128 or a height of 32. Then the input is converted into a target image (white) of size 128×32 . Data augmentation can be integrated by randomly positioning the picture instead of aligning it to the left or by randomly resizing it. CNN Output: In the output, each entry contains 256 features. These features that we have received are processed further by the RNN layers. However, some of the features already show a high correlation with specific high-level properties of the input image provided. There are some features that have a high correlation with duplicate characters (example: “tt”), or with certain characters (example: “e”) or with character properties like loops and curves (example: in handwritten “l”s or “e”s). Implementation using Tensorflow: The implementation consists of 4 modules: SamplePreprocessor.py: To prepare the images from the dataset for the Neural Network. DataLoader.py: Reading samples, putting them into batches, and providing an iterator-interface to go through the data. Model.py: To create the model from the given architecture, loads and saves models, manages the session simultaneously, providing an interface for training and inference. main.py: This file puts all the previously mentioned modules together. We only look at Model.py, since the other source files are concerned with basic file IO, input and output (DataLoader.py), and image processing (SamplePreprocessor.py). To test whether the prediction network could also be used to generate convincing real-valued sequences, we applied it to online handwriting data (online in this context means that the writing is recorded as a sequence of pen-tip locations, as opposed to offline handwriting, where only the page images are available). Online handwriting is an attractive choice for sequence generation due to its low dimensionality (two real numbers per data point) and ease of visualisation. All the data used for this paper were taken from the IAM online handwriting database (IAM-OnDB). IAM-OnDB consists of handwritten lines collected from 221 different writers using a ‘smart whiteboard’. The writers were asked to write forms from the Lancaster-Oslo-Bergen text corpus, and the position of their pen was tracked using an infra-red device in the corner of the board. Samples from the training data are shown in . The original input data consists of the x and y pen co-ordinates and the points in the sequence when the pen is lifted off the whiteboard. Recording errors in the x, y data was corrected by interpolating to fill in for missing readings, and removing steps

whose length exceeded a certain threshold. Beyond that, no preprocessing was used and the network was trained to predict the x, y co-ordinates and the end-of-stroke markers one point at a time. This contrasts with most approaches to handwriting recognition and synthesis, which rely on sophisticated preprocessing and feature-extraction techniques. We eschewed such techniques because they tend to reduce the variation in the data (e.g. by normalising the character size, slant, skew and so-on) which we wanted the network to model. Predicting the pen traces one point at a time gives the network maximum flexibility to invent novel handwriting, but also requires a lot of memory, with the average letter occupying more than 25 timesteps and the average line occupying around 700. Predicting delayed strokes (such as dots for ‘i’s or crosses for ‘t’s that are added after the rest of the word has been written) is especially demanding. IAM-OnDB is divided into a training set, two validation sets and a test set, containing respectively 5364, 1438, 1518 and 3859 handwritten lines taken from 775, 192, 216 and 544 forms. For our experiments, each line was treated as a separate sequence (meaning that possible dependencies between successive lines were ignored). In order to maximise the amount of training data, we used the training set, test set and the larger of the validation sets for training and the smaller validation set for early-stopping. The lack of independent test set means that the recorded results may be somewhat overfit on the validation set; however the validation results are of secondary importance, since no benchmark results exist and the main goal was to generate convincing-looking handwriting. The principal challenge in applying the prediction network to online handwriting data was determining a predictive distribution suitable for real-valued inputs. The following section describes how this was done.

Chapter 5

Testing

5.1 Unit Testing

Unit Testing is the first level of testing, which is typically performed by the developers themselves. We tested the working of both modules, text to handwriting as well as handwriting to text. It helped us understand the desired output of each module, which we had broken down into separate units. This approach came in handy for us to understand each unit.

5.2 Integration Testing

Integrated tests are usually made up of a mix of automated functional and manual tests, and they can be carried out by developers or independent testers. It is the phase in software testing in which individual software modules are combined and tested as a group. Integration testing is conducted to evaluate the compliance of a system or component with specified functional requirements. The next step is to put them all together into one module. This testing is essential for establishing which devices will perform flawlessly together.

5.3 Compatibility Testing

A compatibility test is an assessment used to ensure a software application is properly working across different browsers, databases, operating systems (OS), mobile devices, networks and hardware. The system does not require more than 4GB RAM even when both of the modules are running. The system can run on very low specification devices.

Chapter 6

Result

Our model contains two parts, in the first part the user enters their own handwriting as an input, the model will perform pre-processing on the input then will return the digitised text version of the given input. In the latter part the user enters digitised text as an input and the model will return a generated handwriting in the form of an image as an output. In our model we have implemented neural networks, we have achieved an accuracy of 64% for our model.

Kunal Sant

Figure 6.1: Input 1

Recognized: "Kunal Sant"
Probability: 0.17584793269634247

Figure 6.2: Result 1

Neha is a teacher

Figure 6.3: Input 2

Recognized: "Neha is a teaches"
Probability: 0.05906381085515022

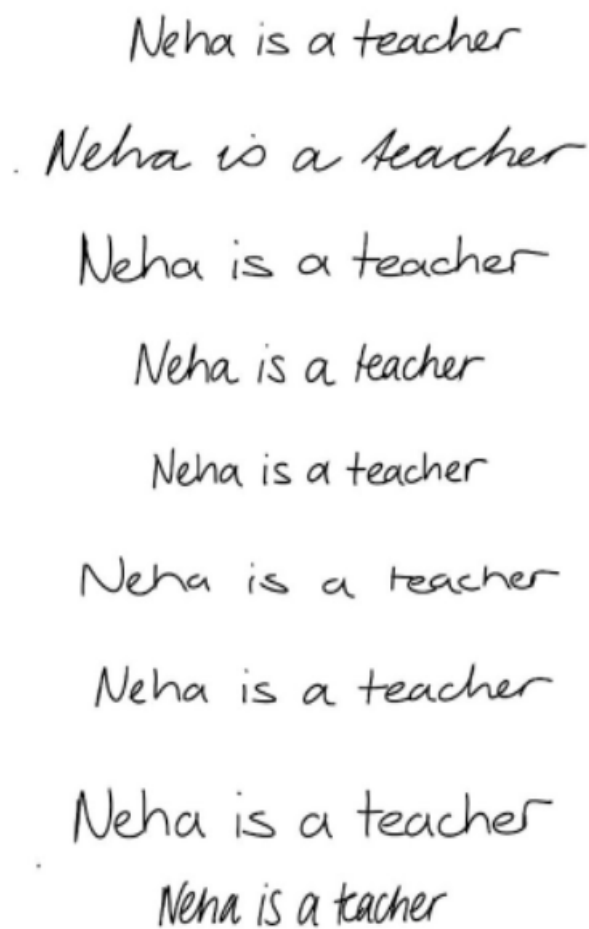
Figure 6.4: Result 2

Kunal is a Student

Figure 6.5: Input 3

Recognized: "Kunel is ce Student "
Probability: 0.025477349758148193

Figure 6.6: Result 3



Neha is a teacher
Neha is a teacher
Neha is a teacher
Neha is a teacher
Neha is a teacher
Neha is a teacher
Neha is a teacher
Neha is a teacher
Neha is a teacher
Neha is a teacher

Figure 6.7: Text to Handwriting Example 1



Figure 6.8: Text to Handwriting Example 2

Chapter 7

Conclusions and Future Scope

We discussed a Neural Network which is able to recognize text in images. Our model can successfully convert the handwriting's given by the user into text and visa with an accuracy of 64%. We aim to create a more efficient handwriting recognition model. We discussed a Neural Network which is able to recognize text in images. The Neural Network consists of 5 CNN and 2 RNN layers and outputs a character-probability matrix. This matrix is either used for CTC loss calculation or for CTC decoding. An implementation using TensorFlow is provided.

The handwriting recognition model's accuracy can be enhanced. To further its versatility, the model may be trained to detect cursive and special characters. Scanning the entire page for handwriting recognition might potentially be included as a feature. Changing the text to handwriting output file to a ruled page rather than a blank page. This model can be used to fill out forms that are cumbersome to fill out by hand. For example, a PAN card or a passport.

Bibliography

- [1] Y. Chherawala, P. P. Roy and M. Cheriet, "Feature set evaluation for offline handwriting recognition systems: Application to the recurrent neural network model", IEEE Trans. Cybern., vol. 46, no. 12, Dec. 2015.
- [2] A. Senior and T. Robinson, "An Off-Line Cursive Handwriting Recognition System", IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 20, no. 3, Mar. 1998.
- [3] J. Pradeep, E. Srinivasan and S. Himavathi, "Diagonal based feature extraction for handwritten character recognition system using neural network", Electronics Computer Technology (ICECT) 2011 3rd International Conference on., vol. 4, 2011
- [4] M. R. Islam, C. Mondal, M. K. Azam and A. S. M.J. Islam, "Text detection and recognition using enhanced MSER detection and a novel OCR technique", Informatics Electronics and Vision (ICIEV) 2016 5th International Conference, 2016
- [5] D. Suryani, P. Doetsch and H. Ney, "On the benefits of convolutional neural network combinations in offline handwriting recognition", 2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR), 2016.
- [6] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Convolutional Neural Network Committees for Hand-written Character classification", In International Conference on Document Analysis and Recognition, 2011.
- [7] Patrick Doetsch, Michal Kozielski and Hermann Ney, "Fast and robust training of recurrent neural networks for offline handwriting recognition", Frontiers in Handwriting Recognition (ICFHR) 2014 14th International Conference on, 2014.
- [8] Jianying Hu, M. K. Brown, and W. Turin, "HMM based online handwriting recognition," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 18, no. 10, October 1996.
- [9] Graves, A. (2013) Generating Sequences With Recurrent Neural Networks. arXiv preprint arXiv:1308.0850. 1–43. Retrieved from <https://arxiv.org/abs/1308.0850>.
- [10] C.C. Tappert, C.Y. Suen and T. Wakahara, "The State of the Art in Online Handwriting Recognition", IEEE Transaction Pattern Analysis and Machine Intelligence, vol. 12, no. 8, August 1990.
- [11] Yefeng Zheng, Huiping Li and David Doermann, "Machine Printed Text and Handwriting Identification in Noisy Document Images", Pattern Analysis Machine Intelligence, vol. 26, no. 3, 2004.

- [12] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke and J. Schmidhuber, "A novel connectionist system for unconstrained handwriting recognition", IEEE Trans. Pattern Anal. Mach. Intell., vol. 31, no. 5, May 2009.
- [13] C. Bahlmann, B. Haasdonk and H. Burkhardt, "Online Handwriting Recognition with Support Vector Machines - A Kernel Approach", Proceedings of the 8th Int. Workshop on Frontiers in Handwriting Recognition (IWFHR), 2002
- [14] U. Marti and H. Bunke. A full English sentence database for off-line handwriting recognition. In Proc. of 5th Int. Conf. on Document Analysis and Recognition, Bangalore, India, 1999.
- [15] L. Xu, A. Krzyzak and C. Y. Suen, "Methods of combining multiple classifiers and their applications to handwriting recognition", IEEE Trans. Syst. Man Cybern., vol. 22, 1992.

7.1 Publication

Paper entitled “**Text Handwriting Conversion using Neural Networks** ” is Submitted and under review at “ **Seventh International Conference on ICT for Sustainable Development**” by “**Aditya Saini, Kunal Sant, Sumeet Swain, Neha Deshmukh**”.