

Handwritten Text Recognition

S Meena Padnekar

M.Tech, Department Of Computer Science

CUSAT

ABSTRACT

Despite the existence of abundance of technological writing tools, people still prefer to take the notes traditionally. However there are many disadvantages in handwritten text. It is difficult to store and access physical document in an efficient manner. We have thus decided to tackle this problem in our project because we believe the significantly greater ease of management of digital text compared to written text will help people more effectively access, search, share, and analyze their records, while still allowing them to use their preferred writing method.

The aim of this project is to translate the handwritten text document to digital format. In this project, the challenge of classifying the image of any handwritten word, which might be of the form of cursive or block writing. This project can be combined with algorithms that segment the word images in a given line image, which can in turn be combined with algorithms that segment the line images in a given image of a whole handwritten page. We approach this problem with complete word images because CNNs tend to work better on raw input pixels rather than features or parts of an image. Given our findings using entire word images, we sought improvement by extracting characters from each word image and then classifying each character independently to reconstruct a whole word. In summary, the model takes the image of the word and output the name of the word.

SOLUTION

The main objective of this project is to classify the individual words in a handwritten document, so that the handwritten text can be translated to digital form. Convolutional Neural Network is used to classify the words in the handwritten document and for character segmentation, Long Short Term Memory networks (LSTM) is used. LSTM is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. A fully developed model would help the user solve the problem of converting handwritten documents into digital format, by prompting the user to take a picture of a page of notes.

INTRODUCTION

Convolutional neural network

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms.

An image is nothing but a matrix of pixel values. A ConvNet is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

The role of the ConvNet is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction. This is important when we are to design an architecture which is not only good at learning features but also is scalable to massive datasets.

The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the Kernel/Filter. The objective of the Convolution Operation is to extract the high-level features such as edges, from the input image. ConvNets need not be limited to only one Convolutional Layer. Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network which has the wholesome understanding of images in the dataset, similar to how we would.

There are two types of results to the operation — one in which the convolved feature is reduced in dimensionality as compared to the input, and the other in which the dimensionality is either increased or remains the same. This is done by applying Valid Padding in case of the former, or Same Padding in the case of the latter.

Pooling Layer

The Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model.

There are two types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.

Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that Max Pooling performs a lot better than Average Pooling.

The Convolutional Layer and the Pooling Layer, together form the i-th layer of a Convolutional Neural Network. Depending on the complexities in the images, the number of such layers may be increased for capturing low-levels details even further, but at the cost of more computational power.

Recurrent Neural Network

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition. The main and most important feature of RNN is Hidden state, which remembers some information about a sequence.

Working of RNN

RNN converts the independent activations into dependent activations by providing same weight and bias to all the layers, thereby reducing the complexity of the parameters and memorizing each previous outputs by giving each output as input to the next hidden layer. Hence all RNN layers can be joined together such that the weights and bias of all the hidden layers is the same, into a single recurrent layer.

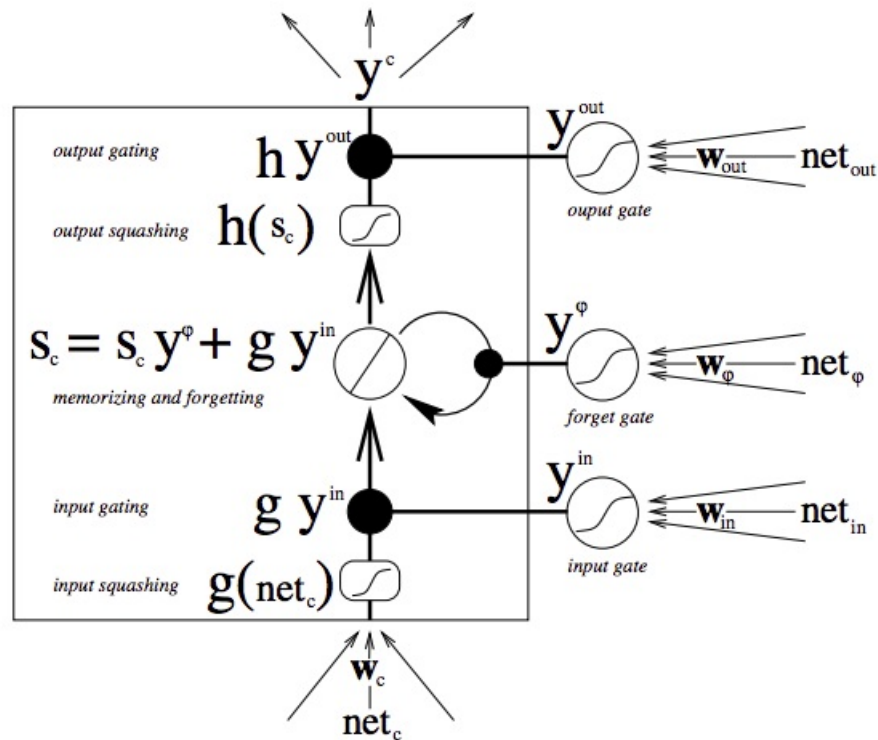
Disadvantages of RNN

Recurrent Neural Networks suffer from short-term memory. If a sequence is long enough, they'll have a hard time carrying information from earlier time steps to later ones. So if you are trying to process a paragraph of text to do predictions, RNN's may leave out important information from the beginning. During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Gradients are values used to update a neural networks weights. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much learning.

LSTM

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture[1] used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can not only process single data points (such as images), but also entire sequences of data. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.

LSTMs contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell, much like data in a computer's memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close. Unlike the digital storage on computers, however, these gates are analog, implemented with element-wise multiplication by sigmoids, which are all in the range of 0-1. Analog has the advantage over digital of being differentiable, and therefore suitable for backpropagation. Those gates act on the signals they receive, and similar to the neural network's nodes, they block or pass on information based on its strength and import, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent. The diagram below illustrates how data flows through a memory cell and is controlled by its gates.



the triple arrows show where information flows into the cell at multiple points. That combination of present input and past cell state is fed not only to the cell itself, but also to each of its three gates, which will decide how the input will be handled. The black dots are the gates themselves, which determine respectively whether to let new input in, erase the present cell state, and/or let that state impact the network's output at the present time step. s_c is the current state of the memory cell, and g_y^{in} is the current input to it. Remember that each gate can be open or shut, and they will recombine their open and shut states at each step. The cell can forget its state, or not; be written to, or not; and be read from, or not, at each time step, and those flows are represented here.

Connectionist temporal classification

Connectionist temporal classification (CTC) is a type of neural network output and associated scoring function, for training recurrent neural networks (RNNs) such as LSTM networks to tackle sequence problems where the timing is variable. It can be used for tasks like on-line handwriting recognition. CTC refers to the outputs and scoring, and is independent of the underlying neural network structure. The input is a sequence of observations, and the outputs are a sequence of labels, which can include blank outputs. The difficulty of training comes from there being many more observations than there are labels. A CTC network has a continuous output (e.g. softmax), which is fitted through training to model the probability of a label. CTC does not attempt to learn boundaries and timings: Label sequences are considered equivalent if they differ only in alignment, ignoring blanks. Equivalent label sequences can occur in many ways – which makes scoring a non-trivial task, but there is an efficient forward-backward algorithm for that. CTC scores can then be used with the back-propagation algorithm to update the neural network weights.

IMPLEMENTATION

The two main approach to build this model is: classification of words and character segmentation. The input to the system is an image of handwritten text. The output from the model is the translated text in digital format.

Dataset

Words in the IAM Handwriting Dataset is used to train the model. IAM Handwriting Dataset is a collection of handwritten passages by several writers. This dataset contain handwritten text over many forms (where a form is an image with lines of text from different writers), sentences, lines, words. The words were then segmented; all associated form label metadata is provided in text file.

Model

Input image to the neural network are resized to 128x32 and are fed into CNN layers.

Convolutional neural network(CNN or ConvNets) is used to train the model to classify the words in the handwritten text. A CNN with five convolutional layers. These layers are trained to extract relevant feature from the image. A kernel filter of 5x5 is used in the first two CNN layers and of 3x3 is used in the next three layers. A pooling layers after each convolutional layer is used to summarize image regions and downsize by two in each layer, and a fully connected layer was used to classify each character. Relu activation function is applied. Feature map of 32x256 is produced as output of CNN layers.

Long Short Term Memory Network, an implementation of Recurrent neural network (RNN) , propagates relevant features/information. LSTM is used as it is able to propagate information through longer distance and provide robust training characteristics. After propagating through LSTM layers a probability matrix of size 32x80 is obtained.

Connectionist temporal classification (CTC) is a type of neural network output and associated scoring function, for training RNNs such as LSTM networks to tackle sequence problems where the timing is variable. In training phase, RNN output is used to compute the loss or error value. And while predicting, it decodes the RNN output matrix to get the final text.

CODE IMPLEMENTATION

Image Preprocessing Module

Grayscale image of size 128x32 is the input to the model.

Images are resized to 128x32 in this module

In []:

```
import cv2
import random
import numpy as np

def preprocess(img,imageSize,dataAug=False):

    if img is None:
        img = np.zeros([imageSize[1],imageSize[2]])

    if dataAug:
        stretch = (random.random()-0.5)
        wStretched = max(int(img.shape[1] * (1 + stretch)), 1)
        img = cv2.resize(img, (wStretched, img.shape[0]))
# creating an image of size 128x32 and copying the required image
        (wt,ht) = imageSize
        (h,w) = img.shape
        fx = w/wt
        fy = h/ht
        f = max(fx,fy)
        newSize = (max(min(wt, int(w / f)), 1), max(min(ht, int(h / f)), 1))
        img = cv2.resize(img,newSize)

        target = np.ones([ht, wt]) * 255
        target[0:newSize[1], 0:newSize[0]]=img
# transpose
        img = cv2.transpose(target)
# normalizing
        (m, s) = cv2.meanStdDev(img)

        m = m[0][0]
        s = s[0][0]
        img = img - m
        img = img / s if s>0 else img

    return img
```

Data Processing

As the dataset is very large, the dataset is divided into a batch of size 500. Cross Validation method is used and best model is saved.

```
In [ ]:

import os
import random
import numpy as np
import cv2
from imagePreprocessing import preprocess

# fetch sample from the dataset
class Sample:
    def __init__(self, gtText, filePath):
        self.gtText = gtText
        self.filePath = filePath

# batch containing images and its labels
class Batch:
    def __init__(self, gtTexts, imgs):
        self.imgs = np.stack(imgs, axis=0)
        self.gtTexts = gtTexts

class DataLoader:
    # loader for dataset at given location, preprocess images and text according to parameters
    def __init__(self, filePath, batchSize, imgSize, maxTextLen):

        assert filePath[-1]=='/'

        self.dataAugmentation = False
        self.currIdx = 0
        self.batchSize = batchSize
        self.imgSize = imgSize
        self.samples = []

        f=open(filePath+'words.txt')
        chars = set()
        bad_samples = []
        bad_samples_reference = ['a01-117-05-02.png', 'r06-022-03-05.png']
        for line in f:
            # ignore comment line
            if not line or line[0]=='#':
                continue
            lineSplit = line.strip().split(' ')
            assert len(lineSplit) >= 9

            # filename: part1-part2-part3 --> part1/part1-part2/part1-part2-part3.png
            fileNameSplit = lineSplit[0].split('-')
            fileName = filePath + 'words/' + fileNameSplit[0] + '/' + fileNameSplit[0] + '-' + fileNameSplit
[1] + '/' + lineSplit[0] + '.png'

            # GT text are columns starting at 9
            gtText = self.truncateLabel(' '.join(lineSplit[8:]), maxTextLen)
            chars = chars.union(set(list(gtText)))

            # check if image is not empty
            if not os.path.getsize(fileName):
                bad_samples.append(lineSplit[0] + '.png')
                continue

            # put sample into list
            self.samples.append(Sample(gtText, fileName))

        # some images in the IAM dataset are known to be damaged, don't show warning for them
        if set(bad_samples) != set(bad_samples_reference):
            print("Warning, damaged images found:", bad_samples)
            print("Damaged images expected:", bad_samples_reference)

        # split into training and validation set: 95% - 5%
        splitIdx = int(0.95 * len(self.samples))
        self.trainSamples = self.samples[:splitIdx]
        self.validationSamples = self.samples[splitIdx:]

        # put words into lists
        self.trainWords = [x.gtText for x in self.trainSamples]
```

```

self.validationWords = [x.gtText for x in self.validationSamples]

# number of randomly chosen samples per epoch for training
self.numTrainSamplesPerEpoch = 25000

# start with train set
self.trainSet()

# list of all chars in dataset
self.charList = sorted(list(chars))

def truncateLabel(self, text, maxTextLen):
    # 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 > maxTextLen:
            return text[:i]
    return text

# switch to randomly chosen subset of training set
def trainSet(self):
    self.dataAugmentation = True
    self.currIdx = 0
    random.shuffle(self.trainSamples)
    self.samples = self.trainSamples[:self.numTrainSamplesPerEpoch]

# switch to validation set
def validationSet(self):
    self.dataAugmentation = False
    self.currIdx = 0
    self.samples = self.validationSamples

# current batch index and overall number of batches
def getIteratorInfo(self):
    return (self.currIdx // self.batchSize + 1, len(self.samples) // self.batchSize)

def hasNext(self):
    return self.currIdx + self.batchSize <= len(self.samples)

def getNext(self):
    batchRange = range(self.currIdx, self.currIdx + self.batchSize)
    gtTexts = [self.samples[i].gtText for i in batchRange]
    imgs = [preprocess(cv2.imread(self.samples[i].filePath, cv2.IMREAD_GRAYSCALE), self.imgSize, self.dataAugmentation) for i in batchRange]
    self.currIdx += self.batchSize
    return Batch(gtTexts, imgs)

```

Model

Input to the neural network is an image of size 128*32

CNN Layers

There are 5 CNN Layers. These layers are trained to extract relevant features from the image. Kernal filter of size 5x5 is applied on the first 2 layers and then filter of size 3x3 in the last 3 layers. Non linear RELU function is applied. A max pooling layer is used after each CNN layer such that this layer summarize the image regions and downsize by 2 in each layer
The output of CNN layer is feature map of size 32x256

RNN Layers

LSTM (a variation of RNN) is used in this model. There are 2 LSTM layers. The feature sequence contain 256 features per time-step. The RNN layer propagates relevant information through this sequence. The out sequence is a matrix of size 32X80.

CTC Layers

In training phase, the rnn output is used to calculate the loss and while predicting, it decoded the RNN output matrix to get the final text

In []:

```
import sys
import numpy as np
import tensorflow as tf
import os

class DecoderType:
    BestPath = 0
    BeamSearch = 1
    WordBeamSearch = 2

class Model:

    batchSize = 50
    imageSize = (128, 32)
    maxTextLen = 32

    def __init__(self, charList, decoderType=DecoderType.BestPath, mustRestore=False, dump=False):
        self.dump = dump
        self.charList = charList
        self.decoderType = decoderType
        self.mustRestore = mustRestore
        self.snapID = 0

        self.is_train = tf.placeholder(tf.bool, name='is_train')

        self.inputImg = tf.placeholder(tf.float32, shape=(
            None, Model.imageSize[0], Model.imageSize[1]))

        self.setupCNN()
        self.setupRNN()
        self.setupCTC()

        self.batchesTrained = 0
        self.learningRate = tf.placeholder(tf.float32, shape=[])
        self.update_op = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
        with tf.control_dependencies(self.update_op):
            self.optimizer = tf.train.RMSPropOptimizer(
                self.learningRate).minimize(self.loss)

        (self.sess, self.saver) = self.setupTF()

    def setupCNN(self):
        # create cnn layer and return output of these layers
        cnnIn4d = tf.expand_dims(input=self.inputImg, axis=3)

        # list of parameters for the layers
        kernelVals = [5, 5, 3, 3, 3]
        featureVals = [1, 32, 64, 128, 128, 256]
        strideVals = poolVals = [(2, 2), (2, 2), (1, 2), (1, 2), (1, 2)]
        numLayers = len(strideVals)
        # CNN layer
```

```

pool = cnnIn4d

for i in range(numLayers):
    kernel = tf.Variable(tf.truncated_normal(
        [kernelVals[i], kernelVals[i], featureVals[i], featureVals[i + 1]], stddev=0.1))
    conv = tf.nn.conv2d(
        pool, kernel, padding='SAME', strides=(1, 1, 1, 1))
    conv_norm = tf.layers.batch_normalization(
        conv, training=self.is_train)
    relu = tf.nn.relu(conv_norm)
    pool = tf.nn.max_pool(relu, [1, poolVals[i][0], poolVals[i][1], 1], (
        1, strideVals[i][0], strideVals[i][1], 1), 'VALID')

self.cnnOut4d = pool

def setupRNN(self):
    # create rnn layers and return output of these layers
    rnnIn3d = tf.squeeze(self.cnnOut4d, axis=[2])

    numHidden = 256
    cell = [tf.contrib.rnn.LSTMCell(
        num_units=numHidden, state_is_tuple=True) for _ in range(2)] # 2 layers

    stacked = tf.contrib.rnn.MultiRNNCell(cell, state_is_tuple=True)

    # bidirectional rnn
    ((fw, bw), _) = tf.nn.bidirectional_dynamic_rnn(
        cell_fw=stacked, cell_bw=stacked, inputs=rnnIn3d, dtype=rnnIn3d.dtype)

    concat = tf.expand_dims(tf.concat([fw, bw], 2), 2)

    kernel = tf.Variable(tf.truncated_normal(
        [1, 1, numHidden * 2, len(self.charList) + 1], stddev=0.1))

    self.rnnOut3d = tf.squeeze(tf.nn.atrous_conv2d(
        value=concat, filters=kernel, rate=1, padding='SAME'), axis=[2])

def setupCTC(self):
    # calculate loss, decode the word and return
    self.ctcIn3d = tf.transpose(self.rnnOut3d, [1, 0, 2])
    self.gtText = tf.SparseTensor(tf.placeholder(tf.int64, shape=[None, 2]), tf.placeholder(
        tf.int32, shape=[None]), tf.placeholder(tf.int64, shape=[2]))

    # loss for batch
    self.seqLen = tf.placeholder(tf.int32, [None])
    self.loss = tf.reduce_mean(tf.nn.ctc_loss(labels=self.gtText, inputs=self.ctcIn3d, sequence_length=self.seqLen, ctc_merge_repeated=True))

    # loss for each element
    self.savedCtcInput = tf.placeholder(
        tf.float32, shape=[Model.maxTextLen, None, len(self.charList)+1])
    self.lossPerElement = tf.nn.ctc_loss(
        labels=self.gtText, inputs=self.savedCtcInput, sequence_length=self.seqLen, ctc_merge_repeated=True)

    # decoder
    if self.decoderType == DecoderType.BestPath:
        self.decoder = tf.nn.ctc_greedy_decoder(
            inputs=self.ctcIn3d, sequence_length=self.seqLen)
    elif self.decoderType == DecoderType.BeamSearch:
        self.decoder = tf.nn.ctc_beam_search_decoder(
            inputs=self.ctcIn3d, sequence_length=self.seqLen, beam_width=50, merge_repeated=False)
    elif self.decoderType == DecoderType.WordBeamSearch:
        word_beam_search_module = tf.load_op_library('TFWordBeamSearch.so')

        # prepare information about language (dictionary, characters in dataset, characters forming words)
        chars = str().join(self.charList)
        wordChars = open('model/wordCharList.txt').read().splitlines()[0]
        corpus = open('data/corpus.txt').read()

        # decode using the "Words" mode of word beam search
        self.decoder = word_beam_search_module.word_beam_search(tf.nn.softmax(
            self.ctcIn3d, dim=2), 50, 'Words', 0.0, corpus.encode('utf8'), chars.encode('utf8'))

def setupTF(self):
    # print('Python: ' + sys.version)
    # print('Tensorflow: ' + tf.__version__)
    # TF session

```



```

sess = tf.Session()

saver = tf.train.Saver(max_to_keep=1, reshape=True) # saver saves model to file
modelDir = 'model/'
latestSnapshot = tf.train.latest_checkpoint(modelDir) # is there a saved model?

# if model must be restored (for inference), there must be a snapshot
if self.mustRestore and not latestSnapshot:
    raise Exception('No saved model found in: ' + modelDir)

# load saved model if available
if latestSnapshot:
    print('Init with stored values from ' + latestSnapshot)
    saver.restore(sess, latestSnapshot)
else:
    print('Init with new values')
    sess.run(tf.global_variables_initializer())

return (sess, saver)

def toSparse(self, texts):

    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)
        labelStr = [self.charList.index(c) for c in text]
        # sparse tensor must have size of max. label-string
        if len(labelStr) > shape[1]:
            shape[1] = len(labelStr)
        # put each label into sparse tensor
        for (i, label) in enumerate(labelStr):
            indices.append([batchElement, i])
            values.append(label)

    return (indices, values, shape)

def decoderOutputToText(self, ctcOutput, batchSize):

    # contains string of labels for each batch element
    encodedLabelStrs = [[] for i in range(batchSize)]

    # word beam search: label strings terminated by blank
    if self.decoderType == DecoderType.WordBeamSearch:
        blank = len(self.charList)
        for b in range(batchSize):
            for label in ctcOutput[b]:
                if label == blank:
                    break
            encodedLabelStrs[b].append(label)

    # TF decoders: label strings are contained in sparse tensor
    else:
        # ctc returns tuple, first element is SparseTensor
        decoded = ctcOutput[0][0]

        # go over all indices and save mapping: batch -> values
        idxDict = {b: [] for b in range(batchSize)}
        for (idx, idx2d) in enumerate(decoded.indices):
            label = decoded.values[idx]
            batchElement = idx2d[0] # index according to [b,t]
            encodedLabelStrs[batchElement].append(label)

        # map labels to chars for all batch elements
        return [str().join([self.charList[c] for c in labelStr]) for labelStr in encodedLabelStrs]

def trainBatch(self, batch):
    # feed a batch into the NN to train it
    numBatchElements = len(batch.imgs)
    sparse = self.toSparse(batch.gtTexts)
    rate = 0.01 if self.batchesTrained < 10 else (0.001 if self.batchesTrained < 10000 else 0.0001) # decay learning rate
    evalList = [self.optimizer, self.loss]
    feedDict = {self.inputImg: batch.imgs, self.gtText: sparse, self.seqLen: [Model.maxTextLen] * numBatchElements, self.learningRate: rate, self.is_train: True}
    (_, lossVal) = self.sess.run(evalList, feedDict)
    self.batchesTrained += 1
    return lossVal

```

```

def dumpNNOutput(self, rnnOutput):
    # dump the output of the NN to CSV file(s)
    dumpDir = 'dump/'
    if not os.path.isdir(dumpDir):
        os.mkdir(dumpDir)

    # iterate over all batch elements and create a CSV file for each one
    maxT, maxB, maxC = rnnOutput.shape
    for b in range(maxB):
        csv = ''
        for t in range(maxT):
            for c in range(maxC):
                csv += str(rnnOutput[t, b, c]) + ';'
            csv += '\n'
        fn = dumpDir + 'rnnOutput_'+str(b)+'.csv'
        print('Write dump of NN to file: ' + fn)
        with open(fn, 'w') as f:
            f.write(csv)

def inferBatch(self, batch, calcProbability=False, probabilityOfGT=False):
    # feed a batch into the NN to recognize the texts

    # decode, optionally save RNN output
    numBatchElements = len(batch.imgs)
    evalRnnOutput = self.dump or calcProbability
    evalList = [self.decoder] + ([self.ctcIn3d] if evalRnnOutput else [])
    feedDict = {self.inputImg: batch.imgs, self.seqLen: [
        Model.maxTextLen] * numBatchElements, self.is_train: False}
    evalRes = self.sess.run(evalList, feedDict)
    decoded = evalRes[0]
    texts = self.decoderOutputToText(decoded, numBatchElements)

    # feed RNN output and recognized text into CTC loss to compute labeling probability
    probs = None
    if calcProbability:
        sparse = self.toSparse(batch.gtTexts) if probabilityOfGT else self.toSparse(texts)
        ctcInput = evalRes[1]
        evalList = self.lossPerElement
        feedDict = {self.savedCtcInput: ctcInput, self.gtText: sparse, self.seqLen: [Model.maxTextLen] *
            numBatchElements, self.is_train: False}
        lossVals = self.sess.run(evalList, feedDict)
        probs = np.exp(-lossVals)

    # dump the output of the NN to CSV file(s)
    if self.dump:
        self.dumpNNOutput(evalRes[1])

    return (texts, probs)

def save(self):
    # save model to file
    self.snapID += 1
    self.saver.save(self.sess, 'model/snapshot', global_step=self.snapID)

```

Main Module

The program works based on the command-line arguments.

By default, it loads the saved model and predict the text in data/test.png image

--train Trains the model using the dataset and saves the model in model/ folder

--validate Validates the trained model

Note: Use a terminal to run this code

In []:

```

from __future__ import division
import numpy as np
import os
import argparse
import cv2

import editdistance
from imagePreprocessing import preprocess
from model import Model, DecoderType
from DataProcessing import DataLoader, Batch

```

File path

class FilePath:

```
input = 'data/test.png'
charList = 'model/charList.txt'
accuracy = 'model/accuracy.txt'
train = 'data/'
corpus = 'data/corpus.txt'
```

def train(model, loader):

```
epoc = 0
bestCharErrorRate = float('inf')
noImprovement = 0
earlyStopping = 50
```

while True:

```
    epoc += 1
    print('Epoc', epoc)
    loader.trainSet()
    print('Training Neural Network')
    while loader.hasNext():
        iterInfo = loader.getIteratorInfo()
        Batch = loader.getNext()
        loss = model.trainBatch(Batch)
        print('Batch : ', iterInfo[0], '/', iterInfo[1], ' Loss = ', loss)
```

```
    print('Validate')
    charErrorRate = validate(model, loader)
```

```
    if charErrorRate < bestCharErrorRate:
        print('Increase in accuracy. Saving Model')
        bestCharErrorRate = charErrorRate
        noImprovement = 0
        model.save()
        open(FilePath.accuracy, 'w').write('Validation Character error rate of the saved model%f%%'%(best
```

CharErrorRate*100))

```
    else:
        print('No increase in Accuracy')
        noImprovement += 1
```

stopping if no improving in acc after 5 epoc

```
if noImprovement >= earlyStopping:
    break
```

def validate(model, loader):

```
loader.validationSet()
numCharErr = 0
numCharTotal = 0
numWordOK = 0
numWordTotal = 0
while loader.hasNext():
    iterInfo = loader.getIteratorInfo()
    print('Batch:', iterInfo[0], '/', iterInfo[1])
    batch = loader.getNext()
    (recognized, _) = model.inferBatch(batch)
```

```
    for i in range(len(recognized)):
        numWordOK += 1 if batch.gtTexts[i] == recognized[i] else 0
        numWordTotal += 1
        dist = editdistance.eval(recognized[i], batch.gtTexts[i])
        numCharErr += dist
        numCharTotal += len(batch.gtTexts[i])
        print('[OK]' if dist==0 else '[ERR:%d]' % dist, '"' + batch.gtTexts[i] + '"', '->', '"' + recogni
```

zed[i] + '"')

print validation result

```
charErrorRate = numCharErr/numCharTotal if numCharTotal !=0 else 0
wordAccuracy = numWordOK/numWordTotal if numWordTotal !=0 else 0
print('Character error rate: %f%%. Word accuracy: %f%%.' % (charErrorRate*100.0, wordAccuracy*100.0))
return charErrorRate
```

def recognize(model, InImage):

```
img = preprocess(cv2.imread(InImage, cv2.IMREAD_GRAYSCALE), Model.imageSize)
batch = Batch(None, [img])
(recognized, probability) = model.inferBatch(batch, True)
print('Recognized:', '"' + recognized[0] + '"')
print('Probability:', probability[0])
```

```

def main():
    parser = argparse.ArgumentParser()

    parser.add_argument('--train', help='train the Neural network', action='store_true')
    parser.add_argument('--validate', help='test the Neural network', action='store_true')
    parser.add_argument('--beamsearch', help='use beam search instead of best path decoding', action='store_true')
    parser.add_argument('--wordbeamsearch', help='use word beam search instead of best path decoding', action='store_true')
    parser.add_argument('--dump', help='store the NN weights', action='store_true')

    args = parser.parse_args()

    decoderType = DecoderType.BestPath
    if args.beamsearch:
        decoderType = DecoderType.BeamSearch
    elif args.wordbeamsearch:
        decoderType = DecoderType.WordBeamSearch

    if args.train or args.validate :
        # load training data
        # execute training and validation
        loader = DataLoader(FilePath.train, Model.batchSize, Model.imageSize, Model.maxTextLen)
        open(FilePath.charList, 'w').write(str().join(loader.charList))
        open(FilePath.corpus, 'w').write(str(' ').join(loader.trainWords + loader.validationWords))

        if args.train:
            # training
            model = Model(loader.charList, decoderType)
            train(model, loader)
        elif args.validate:
            # validate
            model = Model(loader, charList, decoderType, mustRestore=True)
            validate(model, loader)
    else:
        # print accuracy
        print(open(FilePath.accuracy).read())
        model = Model(open(FilePath.charList).read(), decoderType, mustRestore=True, dump=args.dump)
        recognize(model, FilePath.input)

if __name__ == '__main__':
    main()

```

Analyse Module

In []:

```

import sys
import math
import pickle
import copy
import numpy as np
import cv2
import matplotlib.pyplot as plt
from DataProcessing import Batch
from model import Model, DecoderType
from imagePreprocessing import preprocess

class Constants:
    #filenames and paths to data
    fnCharList = 'model/charList.txt'
    fnAnalyze = 'data/analyze.png'
    fnPixelRelevance = 'data/pixelRelevance.npy'
    fnTranslationInvariance = 'data/translationInvariance.npy'
    fnTranslationInvarianceTexts = 'data/translationInvarianceTexts.pickle'
    gtText = 'are'
    distribution = 'histogram' # 'histogram' or 'uniform'

def odds(val):
    return val / (1 - val)

def weightOfEvidence(origProb, margProb):
    return math.log2(odds(origProb)) - math.log2(odds(margProb))

def analyzePixelRelevance():

```

```

def analyzePixelRelevance():
    model = Model(open(Constants.fnCharList).read(), DecoderType.BestPath, mustRestore=True)

    # read image and specify ground-truth text
    img = cv2.imread(Constants.fnAnalyze, cv2.IMREAD_GRAYSCALE)
    w: object
    (w, h) = img.shape
    assert Model.imgSize[1] == w

    # compute probability of gt text in original image
    batch = Batch([Constants.gtText], [preprocess(img, Model.imgSize)])
    (_, probs) = model.inferBatch(batch, calcProbability=True, probabilityOfGT=True)
    origProb = probs[0]

    grayValues = [0, 63, 127, 191, 255]
    if Constants.distribution == 'histogram':
        bins = [0, 31, 95, 159, 223, 255]
        (hist, _) = np.histogram(img, bins=bins)
        pixelProb = hist / sum(hist)
    elif Constants.distribution == 'uniform':
        pixelProb = [1.0 / len(grayValues) for _ in grayValues]
    else:
        raise Exception('unknown value for Constants.distribution')

    # iterate over all pixels in image
    pixelRelevance = np.zeros(img.shape, np.float32)
    for x in range(w):
        for y in range(h):

            # try a subset of possible grayvalues of pixel (x,y)
            imgsMarginalized = []
            for g in grayValues:
                imgChanged = copy.deepcopy(img)
                imgChanged[x, y] = g
                imgsMarginalized.append(preprocess(imgChanged, Model.imgSize))

            # put them all into one batch
            batch = Batch([Constants.gtText]*len(imgsMarginalized), imgsMarginalized)

            # compute probabilities
            (_, probs) = model.inferBatch(batch, calcProbability=True, probabilityOfGT=True)

            # marginalize over pixel value (assume uniform distribution)
            margProb = sum([probs[i] * pixelProb[i] for i in range(len(grayValues))])

            pixelRelevance[x, y] = weightOfEvidence(origProb, margProb)

            print(x, y, pixelRelevance[x, y], origProb, margProb)

    np.save(Constants.fnPixelRelevance, pixelRelevance)

def analyzeTranslationInvariance():
    # setup model
    model = Model(open(Constants.fnCharList).read(), DecoderType.BestPath, mustRestore=True)

    # read image and specify ground-truth text
    img = cv2.imread(Constants.fnAnalyze, cv2.IMREAD_GRAYSCALE)
    (w, h) = img.shape
    assert Model.imgSize[1] == w

    imgList = []
    for dy in range(Model.imgSize[0]-h+1):
        targetImg = np.ones((Model.imgSize[1], Model.imgSize[0])) * 255
        targetImg[:,dy:h+dy] = img
        imgList.append(preprocess(targetImg, Model.imgSize))

    # put images and gt texts into batch
    batch = Batch([Constants.gtText]*len(imgList), imgList)

    # compute probabilities
    (texts, probs) = model.inferBatch(batch, calcProbability=True, probabilityOfGT=True)

    # save results to file
    f = open(Constants.fnTranslationInvarianceTexts, 'wb')
    pickle.dump(texts, f)
    f.close()
    np.save(Constants.fnTranslationInvariance, probs)

def showResults():
    # 1. pixel relevance

```

```
pixelRelevance = np.load(Constants.fnPixelRelevance)
plt.figure('Pixel relevance')

plt.imshow(pixelRelevance, cmap=plt.cm.jet, vmin=-0.25, vmax=0.25)
plt.colorbar()

img = cv2.imread(Constants.fnAnalyze, cv2.IMREAD_GRAYSCALE)
plt.imshow(img, cmap=plt.cm.gray, alpha=.4)

# 2. translation invariance
probs = np.load(Constants.fnTranslationInvariance)
f = open(Constants.fnTranslationInvarianceTexts, 'rb')
texts = pickle.load(f)
texts = ['%d:%i + texts[i] for i in range(len(texts))]
f.close()

plt.figure('Translation invariance')

plt.plot(probs, 'o-')
plt.xticks(np.arange(len(texts)), texts, rotation='vertical')
plt.xlabel('horizontal translation and best path')
plt.ylabel('text probability of "%s"'%Constants.gtText)


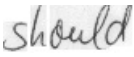

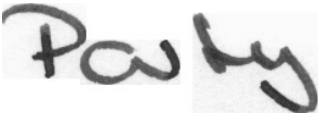
# show both plots
plt.show()

if __name__ == '__main__':
    if len(sys.argv)>1:
        if sys.argv[1]=='--relevance':
            print('Analyze pixel relevance')
            analyzePixelRelevance()
        elif sys.argv[1]=='--invariance':
            print('Analyze translation invariance')
            analyzeTranslationInvariance()
    else:
        print('Show results')
        showResults()
```

As for the first phase, image preprocessing module and the design of network(Model Module) was done.

Implementation for predicting the text in the image, training the model and analyse module to check various parameter was implemented in the second phase of development.

RESULT

			
Recognized: "little"	Recognized: "should"	Recognized: "lastight-"	Recognized: "Party"

RESULT

A model that can recognize words of size upto 32 characters was implemented.
Validation Character error rate of the saved model is 16.507551%

REFERENCE

1. Handwritten Text Recognition, Batuhan Balci, Dan Saadati, Dan Shiferaw

LINKS

Full code : <https://github.com/smeenapadnekar/HandWritten-Text-Recognition> (<https://github.com/smeenapadnekar/HandWritten-Text-Recognition>)

Dataset : <http://www.fki.inf.unibe.ch/databases/iam-handwriting-database> (<http://www.fki.inf.unibe.ch/databases/iam-handwriting-database>)

TUTORIALS : <https://www.tensorflow.org/tutorials/images/cnn> (<https://www.tensorflow.org/tutorials/images/cnn>)