ARTIFICIAL NEURAL NETWORK BASED OPTICAL CHARACTER RECOGNITION

Submitted in partial fulfillment of the requirements of the degree of

Bachelor of Technology

In

Electronics and Communication Engineering with Specialization in Internet of Things and Sensors

By

KARTHIK REDDY T

16BIS0074

Under the guidance of

Dr. Ravi S

SENSE,

VIT Vellore.

December,2020

DECLARATION

I hereby declare that the thesis entitled *“Artificial Neural Network based Optical Character Recognition”,* submitted by me, for the award of the degree of ***Bachelor of Technology in Electronics and Communication with Specialization in Internet of Things and Sensors*** to VIT is a record of bonafide work carried out by me under the supervision of Dr.Ravi S.

I further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Vellore
Date: 28/12/2020

Signature of the Candidate

CERTIFICATE

This is to certify that the thesis entitled ―Artificial Neural Network based Optical Character Recognition‖ submitted by Karthik reddy – 16BIS0074, SENSE, VIT University, for the award of the degree of Bachelor of Technology in Electronics and Communication with Specialization in Internet of Things and Sensors, is a record of bonafide work carried out by him under my supervision during the period, 05. 08. 2020 to 28.12.2020, as per the VIT code of academic and research ethics.

The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The thesis fulfills the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place: Vellore

Date: **Signature of the Guide**

Dr.Ravi S

Internal Examiner **External Examiner**

Dr. M.Arun

Head of the Department

(Department of Embedded Technology)

ACKNOWLEDGEMENTS

I acknowledge the School of Electronics Engineering for giving me an opportunity to do this project.

I express the deepest sense of gratitude to my guide Dr.Ravi S, for his guidance, supervision and constructive criticism for improving and successfully completing the project.

- Karthik reddy

(16BIS0074)

EXECUTIVE SUMMARY

Handwriting recognition is the ability of a machine to receive and interpret handwritten input from multiple sources like paper documents, photographs, touch screen devices etc. Recognition of handwritten and machine characters is an emerging area of research and finds extensive applications in banks, offices and industries.

This project, ‗ANN based Optical Character Recognition‘ is a software algorithm project to recognize any hand written character efficiently on computer with input as an optical image. This is achieved by deep learning algorithms using convolutional neural networks (CNNs) and recurrent neural networks (RNNs). So the neural network model can be viewed as a function which maps a matrix of pixels (images) to a sequence of characters.

$$\mathrm{NN}: \underset{W \times H}{M} \rightarrow \underset{0 \leq n \leq L}{(c_1, c_2, \ldots, c_n)}$$

The neural network consists of CNN, RNN and Connectionist Temporal Classification (CTC) layers. The text is recognized on character-level, therefore words or texts not contained in the training data can be recognized too as long as the individual characters get correctly classified.

The Neural Network (NN) is trained on word-images from the IAM dataset. As the input layer (and therefore also all the other layers) can be kept small for word-images, NN-training is feasible on the CPU (of course, a GPU would be better). The neural network is modeled in Python language making use of deep learning libraries like TensorFlow, Keras and other scientific and numeric computation libraries like NumPy, OpenCV etc.

CONTENTS

List of Figures

List of Abbreviations

AI	Artificial Intelligence
HTR	Handwritten Text Recognition
HCR	Handwritten Character Recognition
OCR	Optical Character Recognition
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
CTC	Connectionist Temporal Classification
RGB Image	Red Green Blue Image
CNTK	Cognitive Toolkit

1. INTRODUCTION

1.1 Background

Artificial intelligence (AI) is a thriving field with many practical applications and active research topics. We look to intelligent software to automate routine labor, understand speech or images make diagnoses in medicine and support basic scientific research. In the early days of artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers—problems that can be described by a list of formal, mathematical rules.

The true challenge to artificial intelligence proved to be solving the tasks that are easy for people to perform but hard for people to describe formally—problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images. The solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all of the knowledge that the computer needs. The hierarchy of concepts allows the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI as ‗deep learning'.

Many of the early successes of AI took place in relatively sterile and formal environments and did not require computers to have much knowledge about the world. For example, IBM‗s Deep Blue chess-playing system defeated world champion Garry Kasparov in 1997 (Hsu, 2002). Chess is of course a very simple world, containing only sixty-four locations and thirty-two pieces that can move in only rigidly circumscribed ways. Devising a successful chess strategy is a tremendous accomplishment, but the challenge is not due to the difficulty of describing the set of chess pieces and allowable moves to the computer. Chess can be completely described by a very brief list of completely formal rules, easily provided ahead of time by the programmer.

Ironically, abstract and formal tasks that are among the most difficult mental undertakings for a human being are among the easiest for a computer. Computers have long been able to defeat even the best human chess player, but are only recently matching some of the abilities of average human beings to recognize objects or speech. A person‘s everyday life requires an immense amount of knowledge about the world. Much of this knowledge is subjective and intuitive, and therefore difficult to articulate in a formal way. Computers need to capture this same knowledge in order to behave in an intelligent way. One of the key challenges in artificial intelligence is how to get this informal knowledge into a computer.

1.2 Motivation

Several artificial intelligence projects have sought to hard-code knowledge about the world in formal languages. A computer can reason about statements in these formal languages automatically using logical inference rules. The difficulties faced by systems relying on hard-coded knowledge suggest that AI systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as _machine learning‘.

Of course, it can be very difficult to extract such high-level, abstract features from raw data. When it is nearly as difficult to obtain a representation as to solve the original problem, representation learning does not, at first glance, seem to help us. Deep learning solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations. Deep learning allows the computer to build complex concepts out of simpler concepts. One such deep learning project is handwritten text recognition.

1.3 Objective

Offline handwriting recognition, often referred to as optical character recognition, is performed after the writing is completed by converting the handwritten document into digital form. The advantage of offline recognition is that it can be done at any time after the document has been written, even years later. The disadvantage is that it is not done in real time as a person writes and therefore not appropriate for immediate text input. Applications of offline handwriting recognition are numerous: reading postal addresses, bank check amounts, and forms. Furthermore, OCR plays an important role for digital libraries, allowing the entry of image textual information into computers by digitization, image restoration, and recognition methods.

Offline handwriting systems generally consist of four processes: acquisition, segmentation, recognition, and post processing. First, the handwriting to be recognized is digitized through scanners or cameras. Second, the image of the document is segmented into lines, words, and individual characters. Third, each character is recognized using OCR techniques. Finally, errors are corrected using lexicons or spelling checkers.

This application is useful for recognizing all the characters (English) present in the input image. Recognition and classification of characters are done by Neural Networks .The main aim of this project is to effectively recognize a particular character in the input image by using the Deep Learning algorithms, which are developed using neural networks.

2. PROJECT DESCRIPTION AND GOALS

This application is useful for recognizing all English characters present in the input image. Once input image of character is given to recognition system, the system recognizes the areas containing handwritten text and then outlines the text and converts it into digital format. Recognition and classification of characters are done by Neural Network. The main aim of this project is to effectively recognize a particular character of type format using the Artificial Neural Network approach.

The goals of this study are:

- To review literature available in various sorts of alphabetical detection systems that can be deployed so that they can carry out hand-written character recognition. The objective behind the review of literature is to figure out strategies that are most appropriate and are efficient with the available resources like running time, space complexity etc.
- To develop hand-written character recognition system in Python by utilizing the deep learning frameworks like TensorFlow, Keras etc.
- To carry out tests on devised algorithm using test pictures (like class notes, posters, etc.) and consequently determine problems within the created algorithm. Assessment is to be based on success rate as well as time consumed by the system to do the recognition.

This project holds great significance since it aims to assist in easing the conversion from physical to electronic type. Such capacity holds significant credibility and its advantages are limitless. This converts hand-written symbols from simple pictures to helpful information that may be utilized in computers. The time used in entering the data and also the storage space required by the documents can be highly reduced by using the HTR Systems.

3. TECHNICAL SPECIFICATION

3.1 Relevant Theory

Artificial Intelligence

Artificial intelligence (AI) refers to the simulation of human intelligence in machines that are programmed to think like humans and mimic their actions. It is a science of finding theories and methodologies that can help machines understand the world and accordingly react to situations in the same way that humans do. The term may also be applied to any machine that exhibits traits associated with a human mind such as learning and problem-solving.

Machine Learning

Machine Learning is a subset of AI. Machine learning allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. A machine learning algorithm is an algorithm that is able to learn from data. Mitchell (1997) provides the definition for learning as ―A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P, improves with experience E.‖

Deep Learning

Deep learning is a specific kind of machine learning. It is a subset of machine learning where algorithms are created and function similar to those in machine learning, but there are numerous layers of these algorithms- each providing a different interpretation to the data it feeds on. The key difference between deep learning and machine learning stems from the way data is presented to the system. Machine learning algorithms almost always require structured data, whereas deep learning networks rely on layers of the ANN (artificial neural networks).

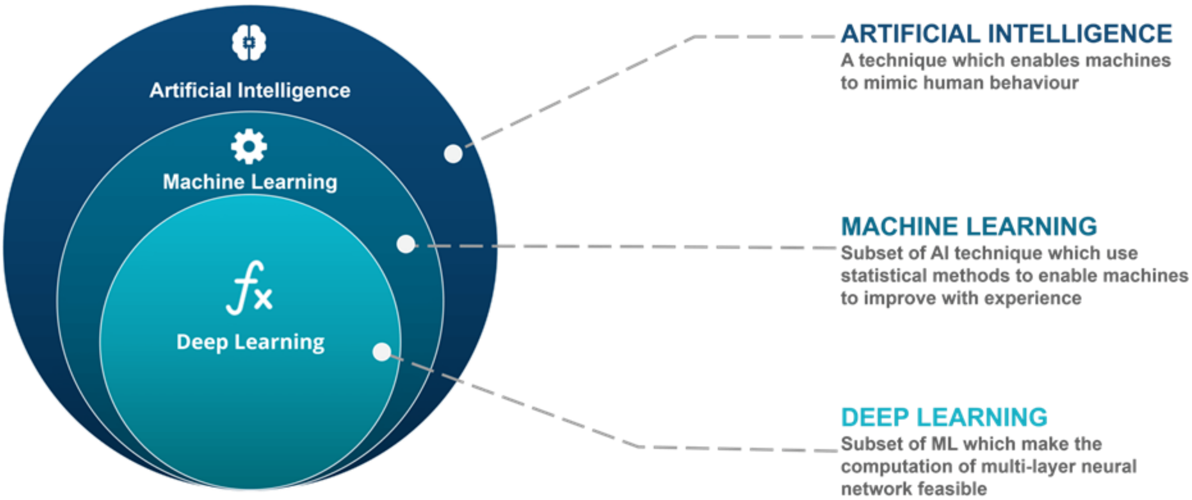


Fig. 1 – Relation between AI, ML and DL

<u>Neural Networks</u>

A neural network is a type of computer system architecture. It consists of data processing by neurons arranged in layers. The corresponding results are obtained through the learning process, which involves modifying the weights of those neurons that are responsible for the error. An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it.

A neural network has the following elements:

- Input Layer: - This layer accepts input features. It provides information from the outside world to the network, no computation is performed at this layer, nodes here just pass on the information (features) to the hidden layer.
- Hidden Layer: - Nodes of this layer are not exposed to the outer world, they are the part of the abstraction provided by any neural network. Hidden layer performs all sort of computation on the features entered through the input layer and transfer the result to the output layer.

- Output Layer: - This layer brings up the information learned by the network to the outer world.

Artificial Neural Networks (ANN) are comprised of a large number of simple elements, called neurons, each of which makes simple decisions. Together, the neurons can provide accurate answers to some complex problems, such as natural language processing, computer vision, and AI. A neural network can be –shallow‖, meaning it has an input layer of neurons, only one –hidden layer‖ that processes the inputs, and an output layer that provides the final output of the model. A Deep Learning Neural Network commonly has between 2-8 additional layers of neurons. Research experts suggest that neural networks increase in accuracy with the number of hidden layers.

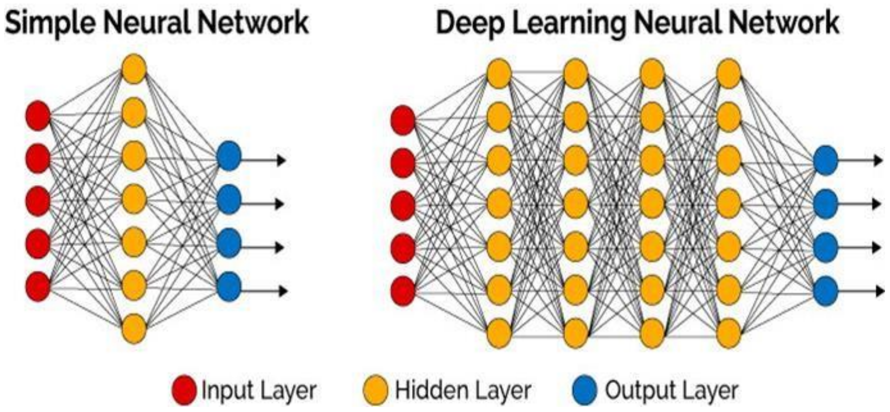


Fig. 2 – Basic Structure of Neural Networks

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks or CNNs in short, are the popular choice of neural networks for different Computer Vision tasks such as image recognition. The name _convolution‘ is derived from a mathematical operation involving the convolution of different functions. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

There are 4 primary steps or stages in designing a CNN:

- Convolution: The input signal is received at this stage and convolution operation is performed.
- Subsampling: Inputs received from the convolution layer are smoothened to reduce the sensitivity of the filters to noise or any other variation.
- Activation: This layer controls how the signal flows from one layer to the other, similar to the neurons in our brain.
- Fully connected: In this stage, all the layers of the network are connected with every neuron from a preceding layer to the neurons from the subsequent layer.

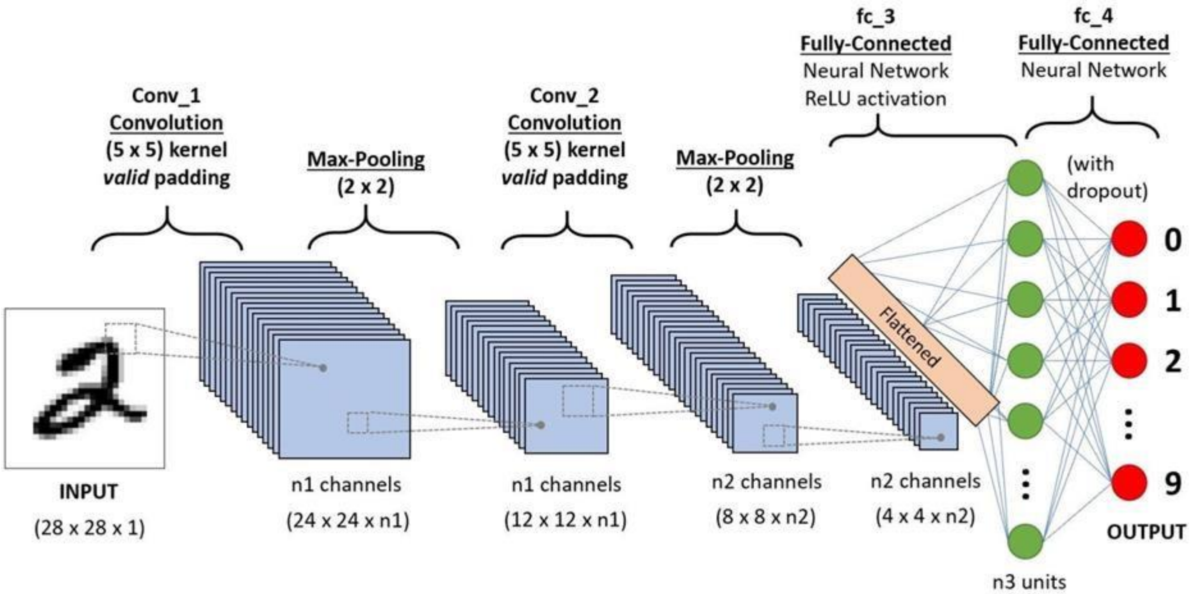


Fig. 3 – Structure of CNN

Recurrent Neural Networks (RNNs)

Recurrent neural networks or RNNs are a family of neural networks for processing sequential data. A recurrent neural network is a neural network that is specialized for processing a sequence of values x(1), . . . , x(t). Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length.

Recurrent Neural Network (RNN) are a type of Neural Network where the output from previous step are fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus, RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is Hidden state, which remembers some information about a sequence.

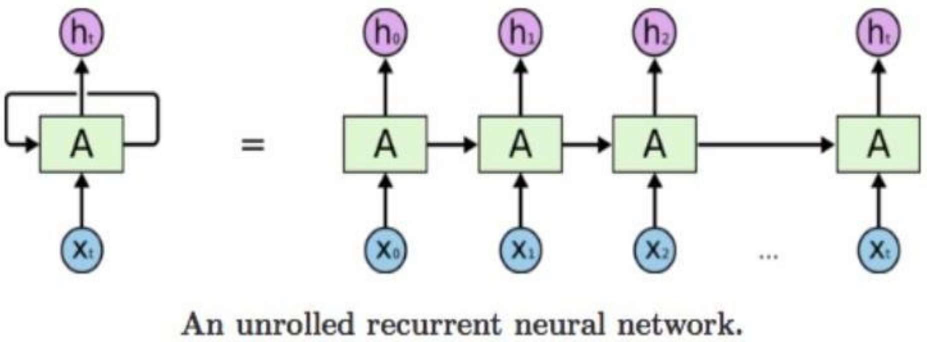


Fig. 4 – Structure of RNN

RNN have a –memory‖ which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

Bidirectional RNNs

Bidirectional recurrent neural networks (RNN) are really just putting two independent RNNs together. The input sequence is fed in normal time order for one network, and in reverse time order for another. The outputs of the two networks are usually concatenated at each time step, though there are other options, e.g. summation.

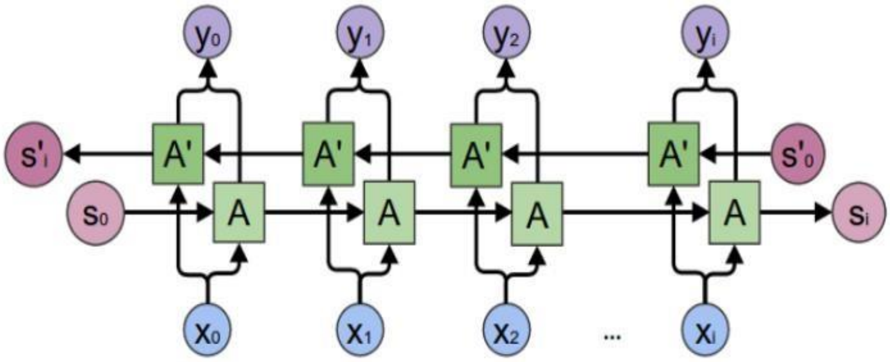


Fig. 5 – Structure of Bidirectional RNN

<u>Long Short Term Memory (LSTM) Networks</u>

Long Short-Term Memory networks – usually just called ―LSTMs‖ – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hoch Reiter & Schmidhuber (1997). LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn. All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single _tanh‘ layer. LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

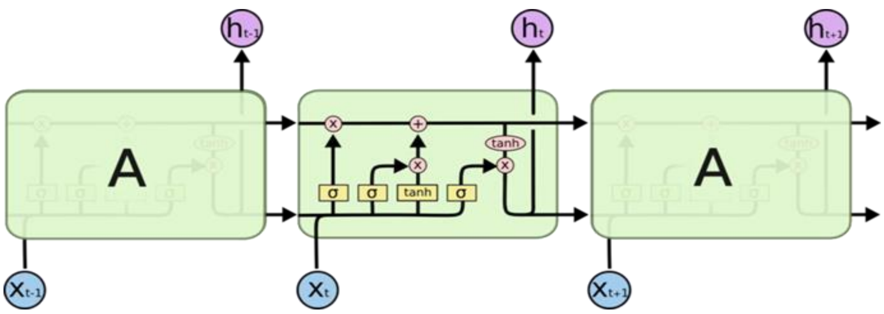
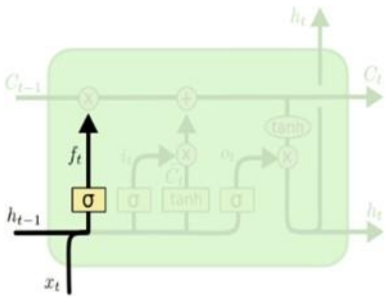


Fig. 6 – Structure of LSTM

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. An LSTM has the following gates, to protect and control the cell state.

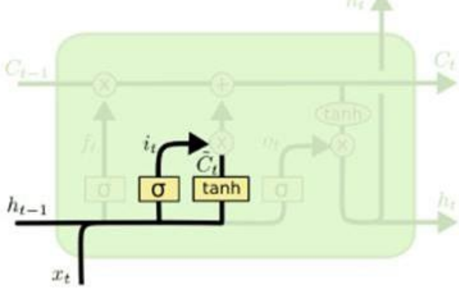
The first step in our LSTM is to decide what information we‘re going to throw away from the cell state. This decision is made by a sigmoid layer called the –forget gate layer.‖ It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents –completely keep this‖ while a 0 represents –completely get rid of this.‖



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Fig. 7 – Forget Gate in an LSTM

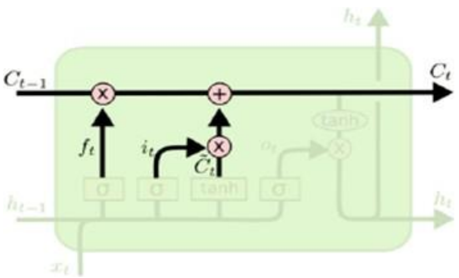
The next step is to decide what new information we‘re going to store in the cell state. This has two parts. First, a sigmoid layer called the –input gate layer‖ decides which values we‘ll update. Next, a _tanh‘ layer creates a vector of new candidate values, C_t, that could be added to the state. In the next step, we‘ll combine these two to create an update to the state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Fig. 8 – Input Gate in an LSTM

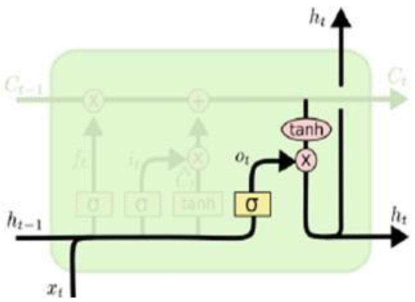
It‘s now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add it∗ C_t. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Fig. 9 – Update Gate in an LSTM

Finally, we need to decide what we‘re going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we‘re going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



$$o_t = \sigma\left(W_o\,[h_{t-1}, x_t] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Fig. 10 – Output Gate in an LSTM

3.2 Software Details

The project is implemented using Python language in the Spyder IDE of the Anaconda Navigator. It is an industry level software providing the required packages and tools for scientific Python programming.

The implementation consists of 4 modules:

1. SamplePreprocessor.py: Prepares the images from the IAM dataset for the NN.
2. DataLoader.py: Reads samples, puts them into batches and provides an iterator-interface to go through the data.
3. Model.py: Creates the model as described above, loads and saves models, manages the TF sessions and provides an interface for training and inference.
4. main.py: Puts all previously mentioned modules together.

The scientific python programming packages used in the coding part are explained below.

TensorFlow

TensorFlow is a software library or framework, designed by the Google team to implement machine learning and deep learning concepts in the easiest manner. It combines the computational algebra of optimization techniques for easy calculation of many mathematical expressions.

The following important features of TensorFlow –

- It includes a feature of that defines, optimizes and calculates mathematical expressions easily with the help of multi-dimensional arrays called tensors.
- It includes a programming support of deep neural networks and machine learning techniques.
- It includes a high scalable feature of computation with various data sets.
- TensorFlow uses GPU computing, automating management. It also includes a unique feature of optimization of same memory and the data used.

Keras

Keras is an open source deep learning framework for python. It has been developed by an artificial intelligence researcher at Google named Francois Chollet. Leading organizations like Google, Square, Netflix, Huawei and Uber are currently using Keras. Keras runs on top of open source machine libraries like TensorFlow, Theano or Cognitive Toolkit (CNTK).

Theano is a python library used for fast numerical computation tasks. TensorFlow is the most famous symbolic math library used for creating neural networks and deep learning models. TensorFlow is very flexible and the primary benefit is distributed computing. Keras is based on minimal structure that provides a clean and easy way to create deep learning models based on TensorFlow or Theano. Keras is designed to quickly define deep learning models. Keras is an optimal choice for deep learning applications.

Keras leverages various optimization techniques to make high level neural network API easier and more performant. It supports the following features —

- Consistent, simple and extensible API.
- Minimal structure — easy to achieve the result without any frills.
- It supports multiple platforms and backends.
- It is user friendly framework which runs on both CPU and GPU.
- Highly scalability of computation.

OpenCV

OpenCV is a cross-platform library using which we can develop real-time computer vision applications. It mainly focuses on image processing, video capture and analysis including features like face detection and object detection.

Computer Vision can be defined as a discipline that explains how to reconstruct, interrupt, and understand a 3D scene from its 2D images, in terms of the properties of the structure present in the scene. It deals with modeling and replicating human vision using computer software and hardware. Computer Vision overlaps significantly with the following fields —

- Image Processing — It focuses on image manipulation.
- Pattern Recognition — It explains various techniques to classify patterns.
- Photogrammetry — It is concerned with obtaining accurate measurements from images.

Apart from these major libraries, other packages like Glob, OS, Numpy, Scipy, Random, etc. are used in the development of the model.

4. DESIGN APPROACH AND DETAILS

4.1 Design Approach

There are four stages in Hand Written Text Recognition System:

1. Image Preprocessing
2. Segmentation
3. Feature Extraction
4. Classification

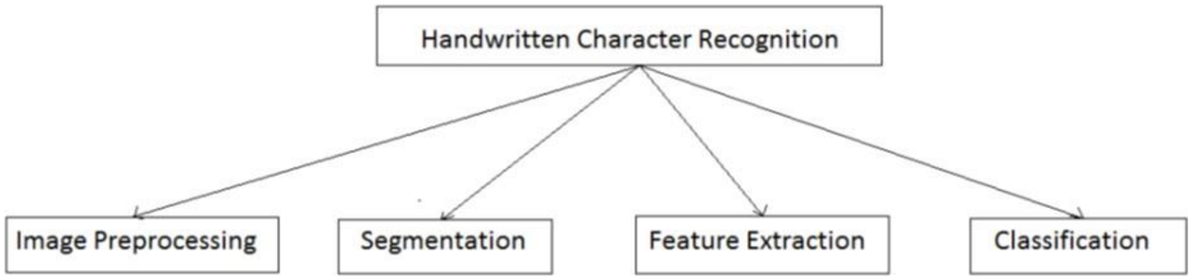


Fig. 11 – Stages in HTR System

Image Preprocessing

The image is preprocessed using different image processing algorithms like Inverting Image, Gray Scale Conversion and Image Thinning. Preprocessing of the sample image involves few steps that are mentioned as follows:

Grey-scaling of RGB image

Grey-scaling of an image is a process by which an RGB image is converted into a black and white image. This process is important for Binarization as after grey-scaling of the image, only shades of grey remains in the image, binarization of such image is efficient.

Binarization

Binarization of an image converts it into an image which only have pure black and pure white pixel values in it. Basically during binarization of a grey-scale image, pixels with intensity lower than half of the full intensity value gets a zero value converting them into black ones. And the remaining pixels get a full intensity value converting it into white pixels.

Inversion

Inversion is a process in which each pixel of the image gets a color which is the inverted color of the previous one. This process is the most important one because any character on a sample image can only be extracted efficiently if it contains only one color which is distinct from the background color. Note that it is only required if the objects we have to identify if of darker intensity on a lighter background.

The flow chart shown below illustrates the physical meaning of the processes that are mentioned above:

RGB => Grey-scaling => Binarization => Inversion

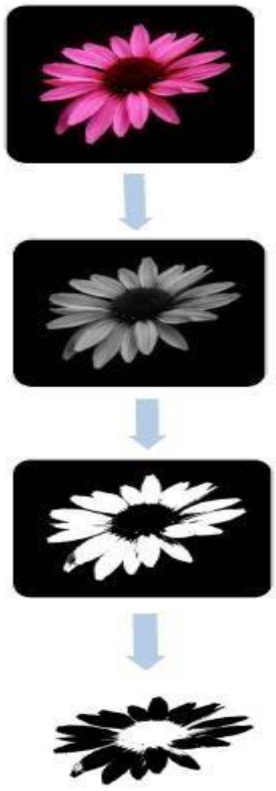


Fig. 12 – Image Preprocessing Steps

Segmentation

After preprocessing of the image segmentation is done. This is done with the help of following steps:

1. Remove the borders
2. Divide the text into rows
3. Divide the rows (lines) into words
4. Divide the word into letters

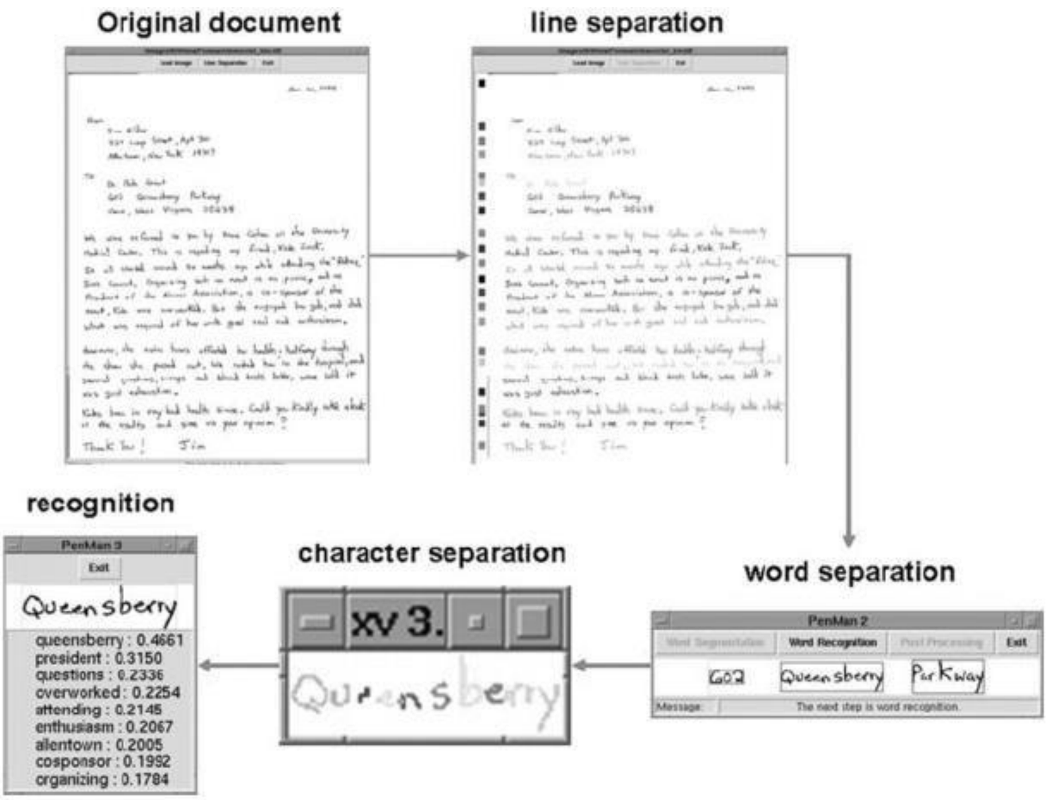


Fig. 13 – Example for Segmentation Process

Feature Extraction

Once the character is segmented we generate the binary graphs and calculate the summation of each row and column values as features. The primary goal of the feature extraction phase is to extricate the pattern that is more appropriate for categorization. These features can be of different types, like horizontal features, vertical features, texture based features, etc. Identification of every segment depends on selection arc type, feature angle, relative position, length ratio and connection angle. Another way to find the feature is (according to biological visual perception) to extract the simple cells and grow these cells on the basis of connected component concept. Some other useful features can be directional features, like size, shape, writing direction, slope, and chart and ending coordinates. The technique of feature extraction algorithm is evident from its designation. It includes an identification of characters or symbols on the basis of their features or aspects that are alike. This concept resembles humans in how they identify characters on the basis of their features or aspects.

0	0	1	0	0
0	1	0	1	0
0	1	0	1	0
1	0	0	0	1
1	1	1	1	1
1	0	0	0	1
1	0	0	0	1

1	1	1	1	0
1	0	0	0	1
1	0	0	0	1
1	1	1	1	0
1	0	0	0	1
1	0	0	0	1
1	1	1	1	0

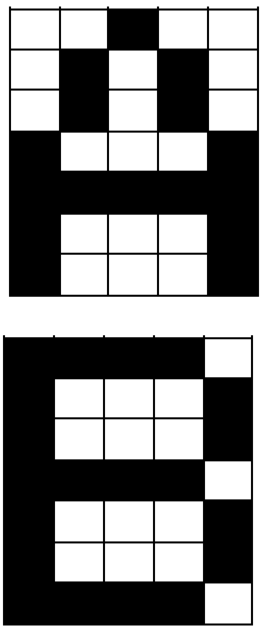


Fig. 14 – Example for Feature Extraction

Classification

In simple words, classification is defined as the process of assigning labels (categories, classes) to unseen observations (instances of data). In machine learning, this is done on the basis of training an algorithm on a set of available data.

Classification is a supervised learning method, where a teacher assigns a label to every student in the class for a particular task. The label is a simple number (or some character) that identifies the class of particular instance.

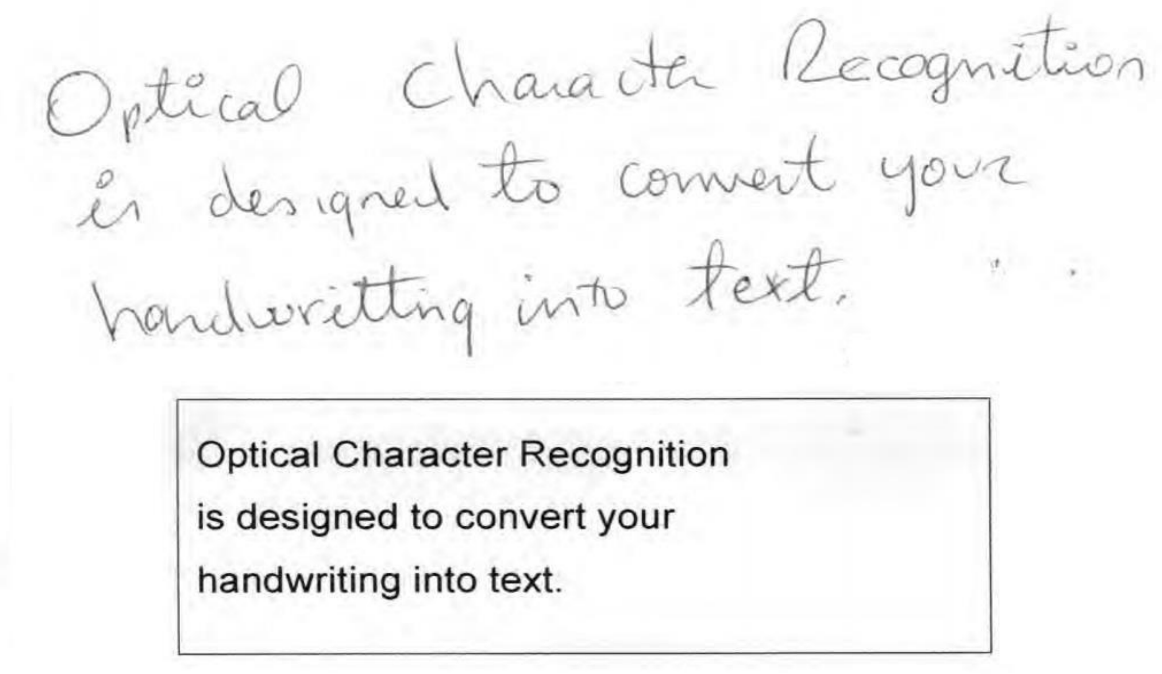


Fig. 15 – Classification Example

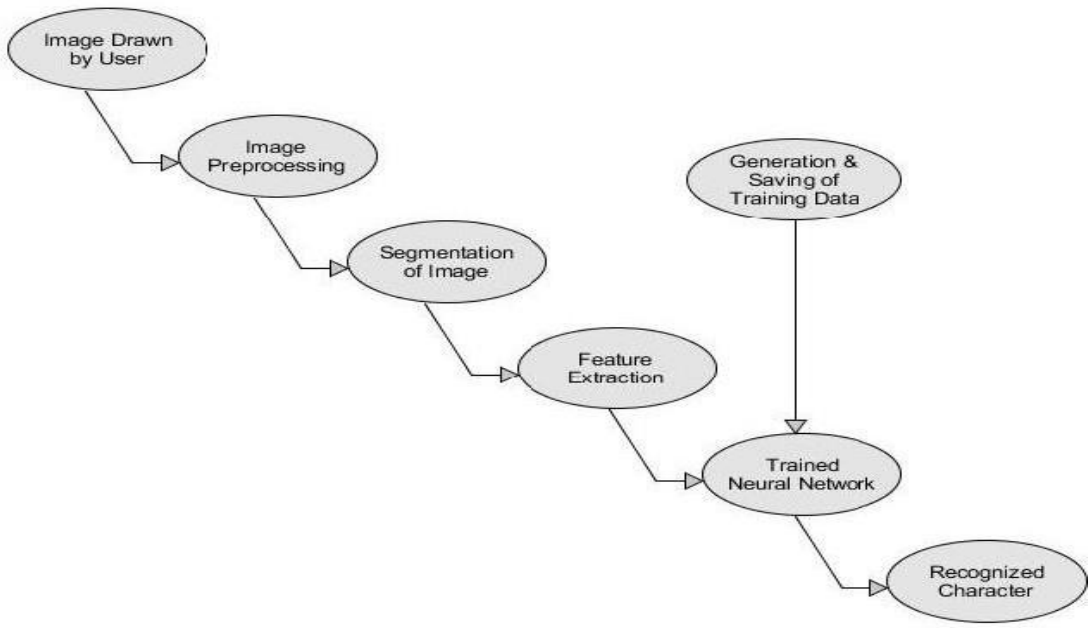


Fig. 16 – Control Flow in the HTR System

4.2 Operations Involved

CNN Layer: The input image is fed into the CNN layers. These layers are trained to extract relevant features from the image. Each layer consists of three operations. First, the convolution operation, which applies a filter kernel of size 5×5 in the first two layers and 3×3 in the last three layers to the input. Then, the non-linear ReLU function is applied. Finally, a pooling layer summarizes image regions and outputs a downsized version of the input. While the image height is downsized by 2 in each layer, feature maps (channels) are added, so that the output feature map (or sequence) has a size of 32×256.

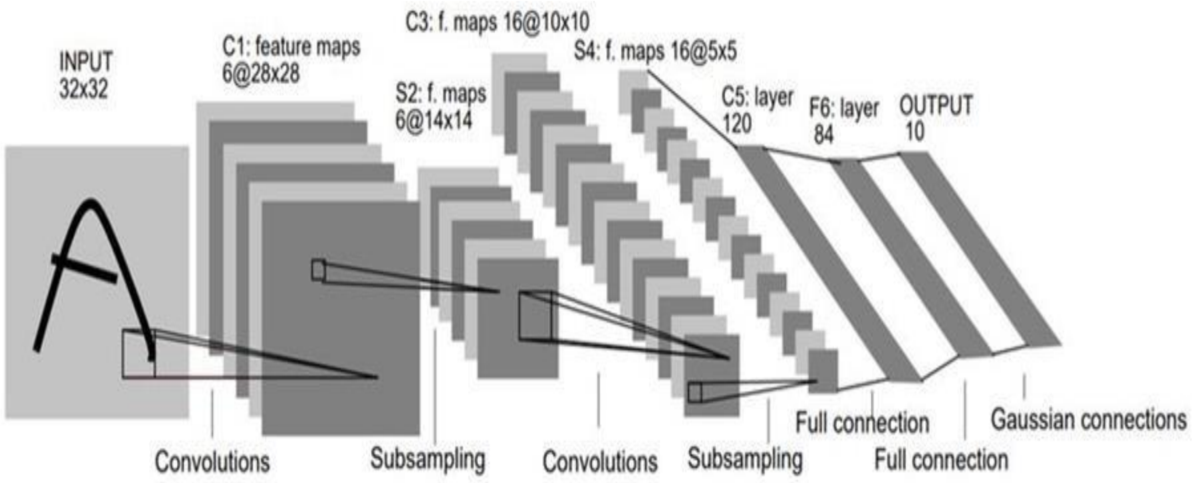


Fig. 17 – Operation at the CNN Layer

RNN Layer: The feature sequence contains 256 features per time step. The RNN propagates relevant information through this sequence. The popular Long Short-Term Memory (LSTM) implementation of RNNs is used, as it is able to propagate information through longer distances and provides more robust training-characteristics than vanilla RNN. The RNN output sequence is mapped to a matrix of size 32×80. The IAM dataset consists of 79 different characters, further one additional character is needed for the CTC operation (CTC blank label), and therefore there are 80 entries for each of the 32 time-steps.

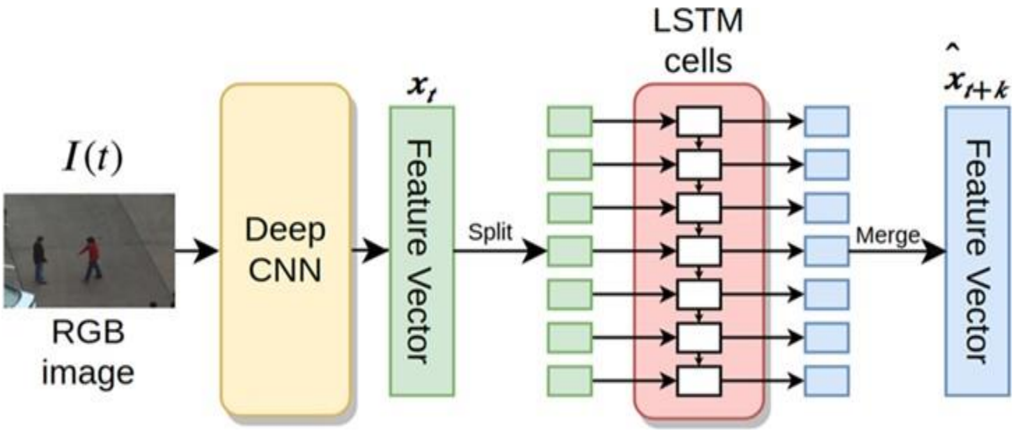


Fig. 18 – Operation at the RNN Layer

CTC Layer: While training the NN, the CTC is given the RNN output matrix and the ground truth text and it computes the loss value. While inferring, the CTC is only given the matrix and it decodes it into the final text. Both the ground truth text and the recognized text can be at most 32 characters long.

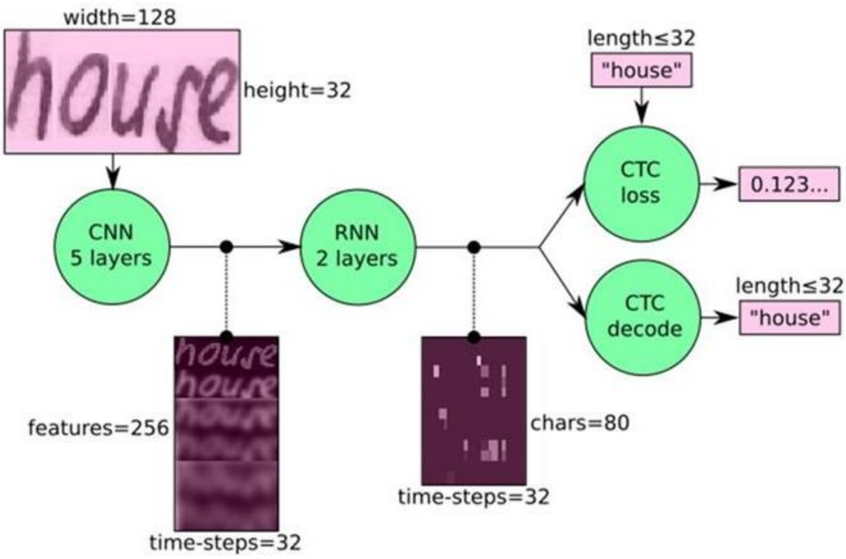


Fig. 19 – Operation at the CTC Layer

4.3 Data Flow

Input: It is a gray-value image of size 128×32. Usually, the images from the dataset do not have exactly this size, therefore we resize it (without distortion) until it either has a width of 128 or a height of 32. Then, we copy the image into a (white) target image of size 128×32. This process is shown in the below figure. Finally, we normalize the gray-values of the image which simplifies the task for the NN. Data augmentation can easily be integrated by copying the image to random positions instead of aligning it to the left or by randomly resizing the image.

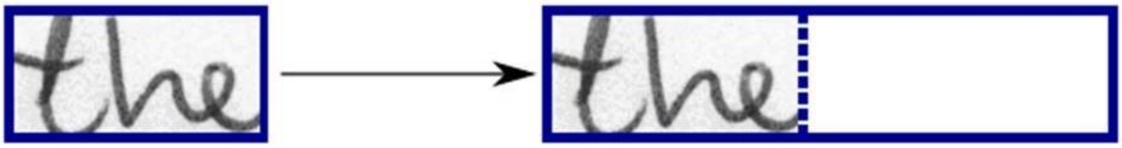


Fig. 20 – Input to CNN Layer

CNN Output: The below figure shows the output of the CNN layers which is a sequence of length 32. Each entry contains 256 features. Of course, these features are further processed by the RNN layers, however, some features already show a high correlation with certain high-level properties of the input image: there are features which have a high correlation with characters (e.g. −e‖), or with duplicate characters (e.g. −tt‖), or with character-properties such as loops (as contained in handwritten −l‖s or −e‖s).

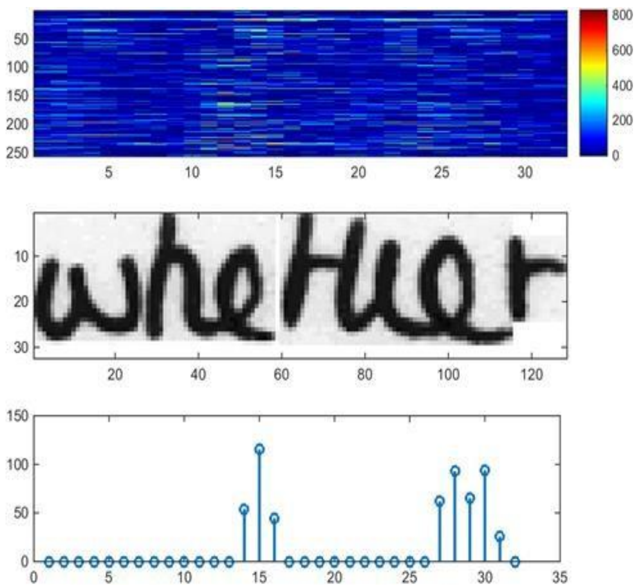


Fig. 21 – CNN Output

RNN Output: The below figure shows a visualization of the RNN output matrix for an image containing the text –little‖. The matrix shown in the top-most graph contains the scores for the characters including the CTC blank label as its last (80th) entry. The other matrix-entries, from top to bottom, correspond to the following characters: ― !‖#&‘()*+, ./0123456789:;?ABCDEFGHIJKLMN OPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz‖. It can be seen that most of the time, the characters are predicted exactly at the position they appear in the image (e.g. compare the position of the –i‖ in the image and in the graph). Only the last character –e‖ is not aligned. But this is OK, as the CTC operation is segmentation-free and does not care about absolute positions. From the bottom-most graph showing the scores for the characters –l‖, –i‖, –t‖, –e‖ and the CTC blank label, the text can easily be decoded: we just take the most probable character from each time-step, this forms the so called best path, then we throw away repeated characters and finally all blanks: –l---ii—t-t—l-…-e‖ → –l---i--t-t--l-…-e‖ → –little‖.

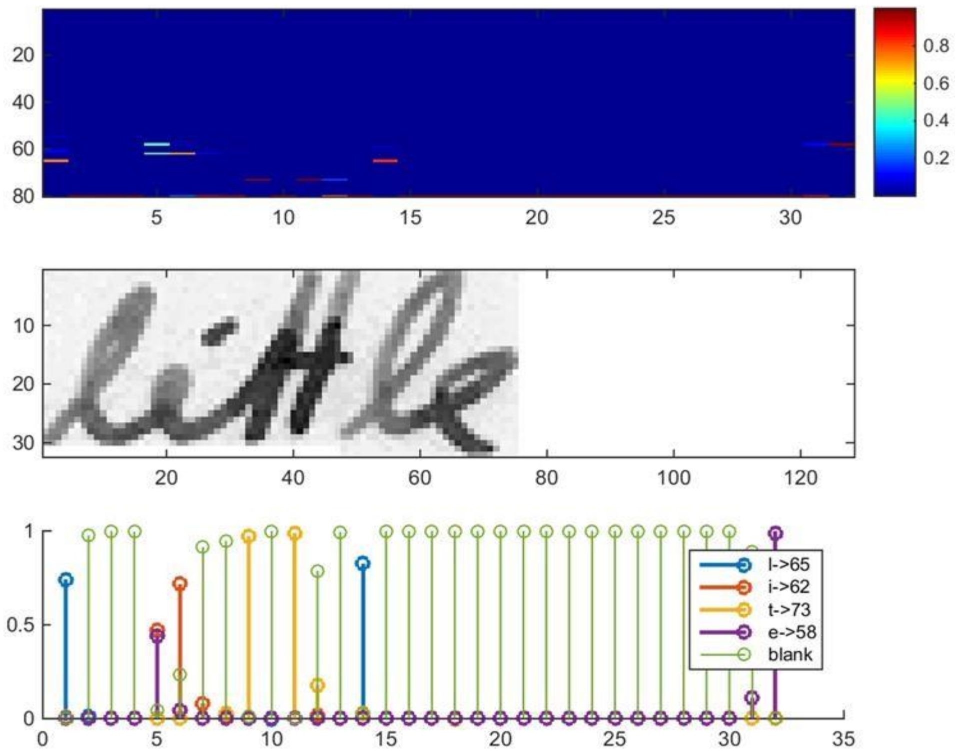


Fig. 22 – RNN Output

1. SCHEDULE, TASKS AND MILESTONES

Initial phases of my work constituted of analysing the literature available on the internet, related to the project title. Literature survey included the selection of specific papers from the huge ocean of the Internet, and then studying the algorithms, methods and tools used by scholars in those papers.

After a detailed analysis of the advantages and disadvantages of the various methods studied, I started working on the algorithm to be implemented for my project. After the algorithm was finalized I started working on its implementation. Since this is completely an algorithm based project, the implementation doesn‘t require any special hardware other than a working computer, with the required software for implementation (Anaconda Navigator 3.0 → Spyder 4).

So the timeline of my work can be roughly stated as:

- 01/12/19 → 15/12/19: Selecting specific papers from the internet, those are most suited to my project.
- 16/12/19 → 31/12/19: Analysing the tools and methods, advantages and disadvantages of various algorithms described in those papers.
- 01/01/20 → 31/01/20: Designing the algorithm to perform the recognition at a character level, instead of word level, so that the system can recognize even the words which are not present in the training phase.
- 01/02/20 → 29/02/20: Coding and implementing the model.
- 01/03/20 → 01/04/20: Testing the model using various testing datasets. Thesis preparation. Poster preparation. Making presentation.

2. PROJECT DEMONSTRATION

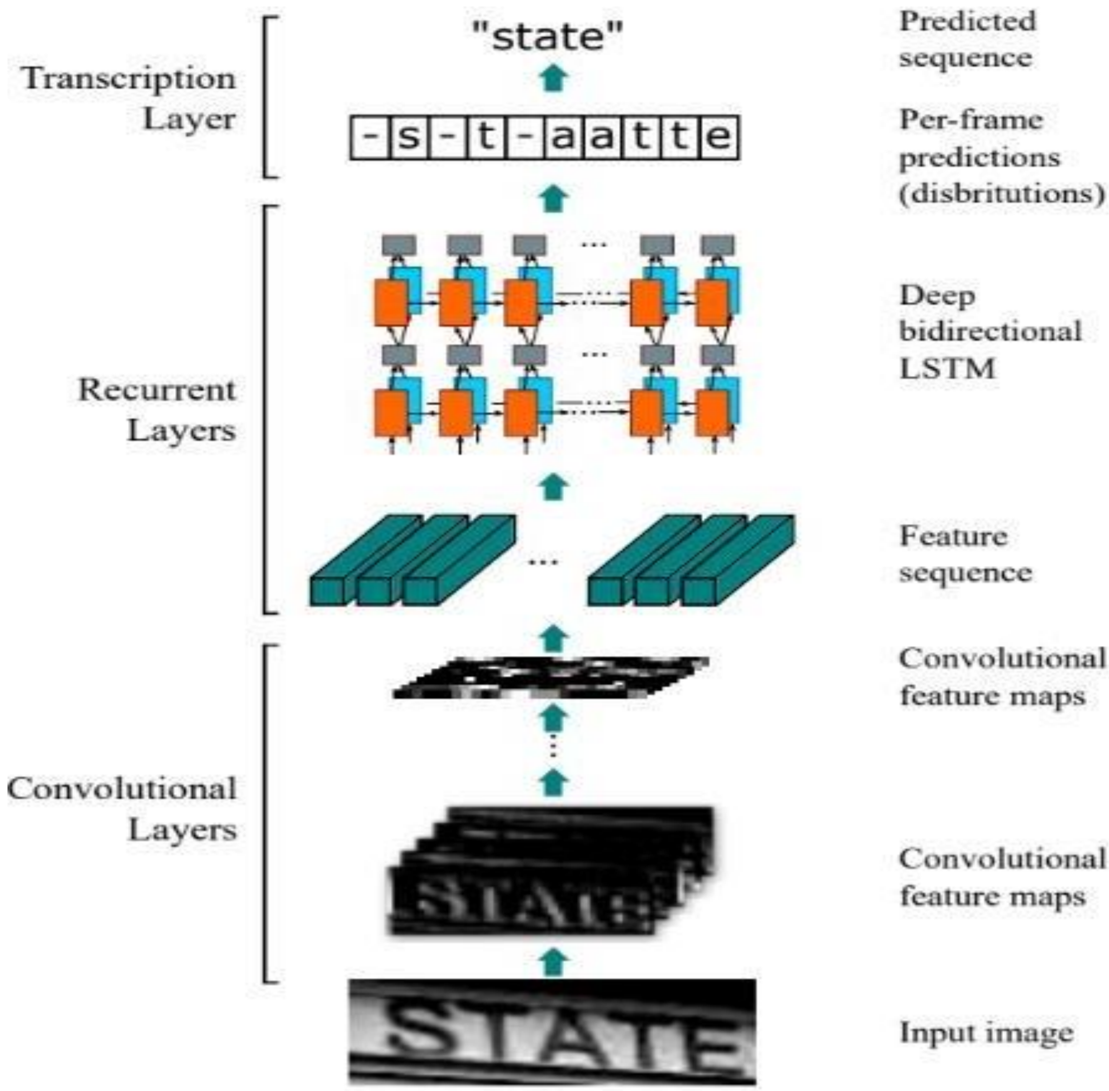


Fig. 23 – Complete Working Example

The above figure demonstrates the stages through which the input image passes and how it gets transformed through these stages and how the text in the input image is represented in the form of sequences in the LSTMs and as feature matrices in the CNNs and thereby how the text is recognized by the system. Since the model is trained on hand written text, we can also use the model for recognizing the printed text, since it is clearer than a hand written text. An example for recognizing hand written text is demonstrated in the below image.

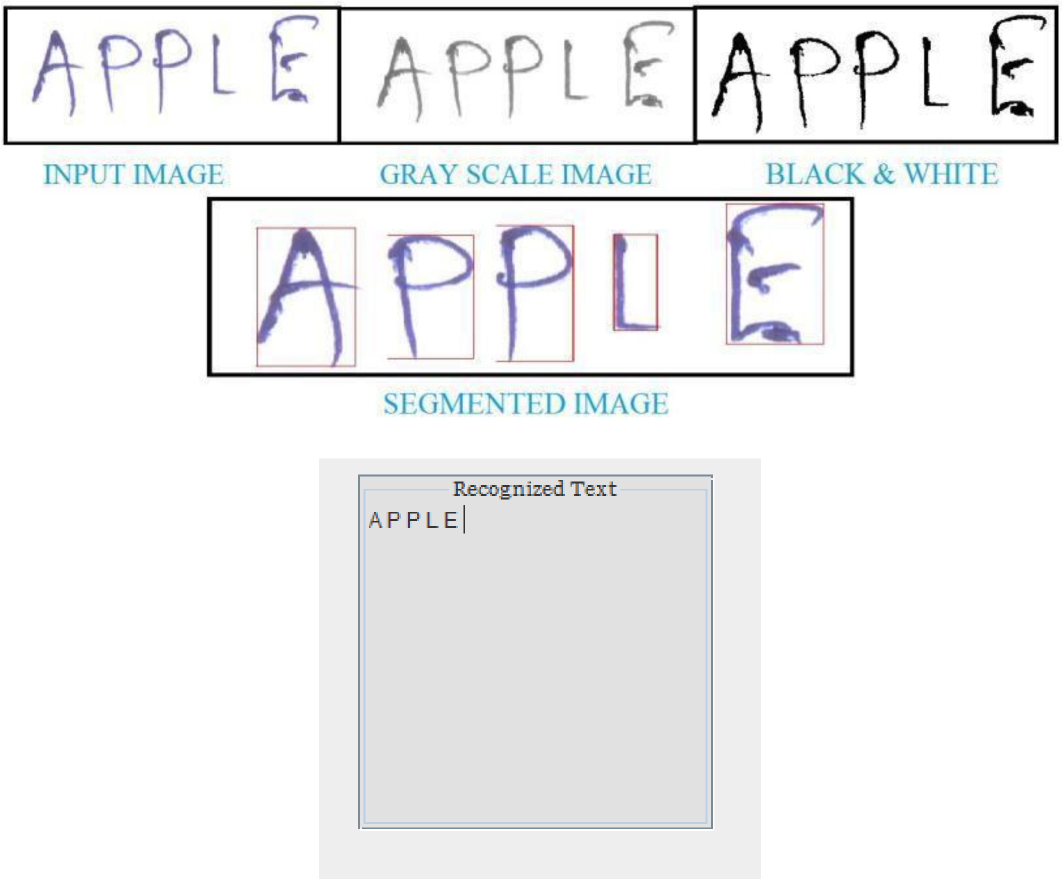


Fig. 24 – Recognizing Words

Fig. 25 – Recognizing Sentences

Another example for the step by step demonstration of the working is given in the below figure.

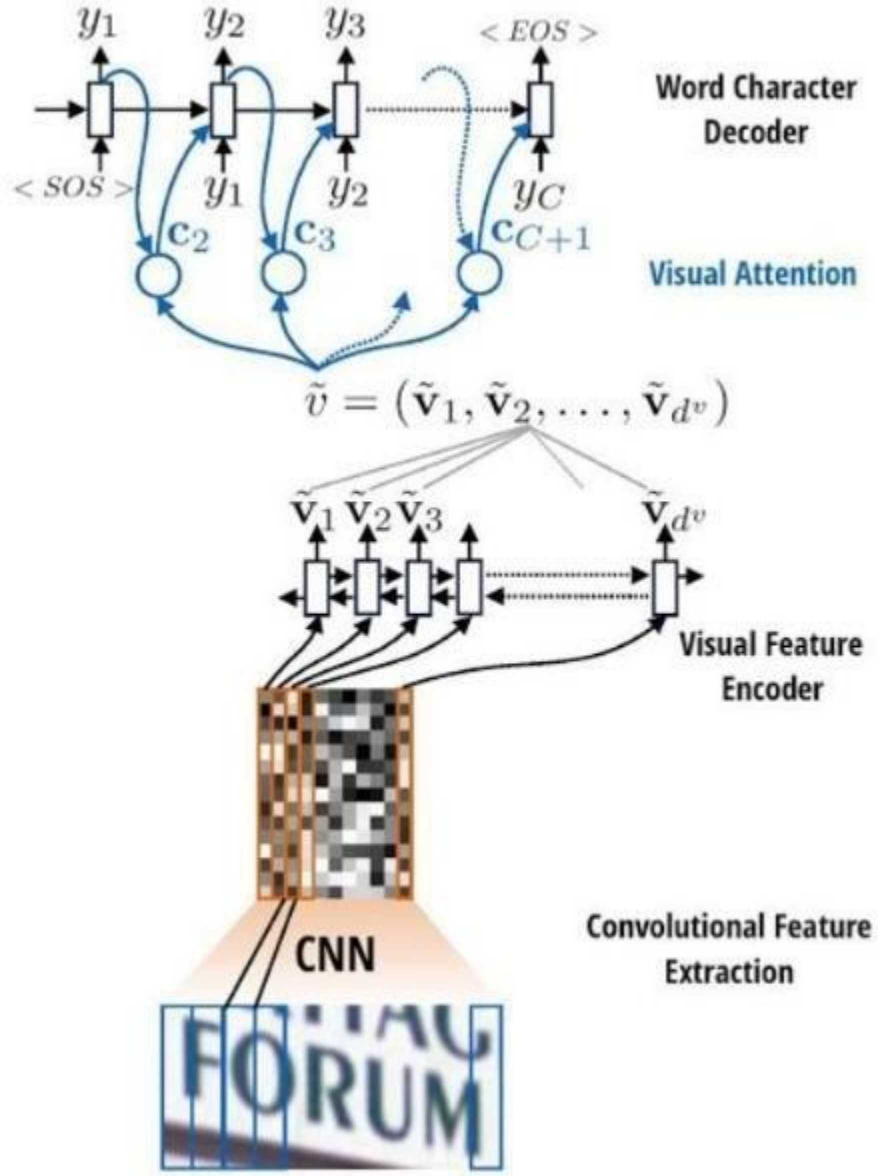


Fig. 26 – Step by Step Working Demonstration

3. RESULTS AND DISCUSSION

The step by step demonstration of the project was described in the previous section. Here I present the results obtained.

The accuracy with which each letter was determined is given in the following figure.

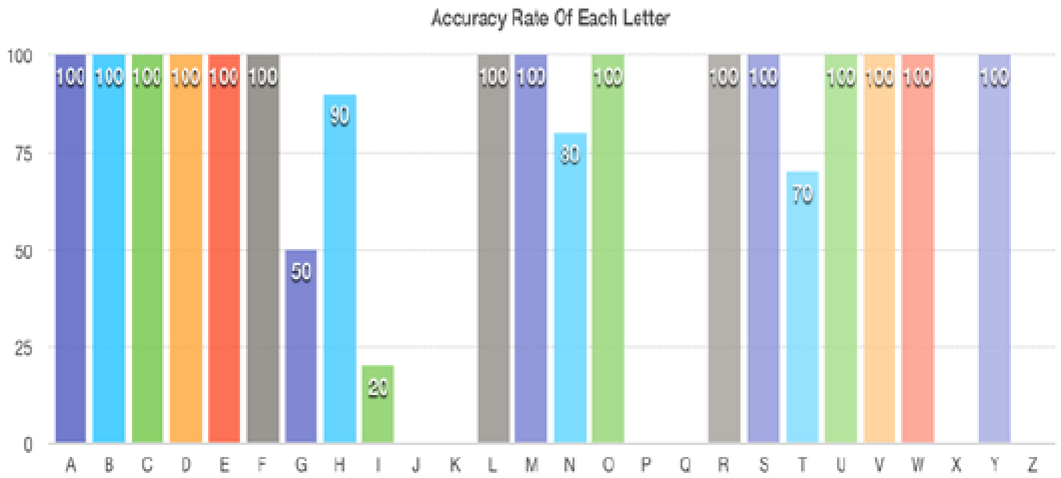


Fig. 27 – Accuracy Rate of each Letter

Accuracy rate of each letter after training the letters and recognizing the image is shown in Figure 16. Accuracy rate is calculated by first calculating the total sum of the given letter and then finding the percentage of given letter where it's false.

As you see, letters such as I and G have a very low accuracy where the accuracy of letters N and T is satisfactory. After training the given letters with more data sets, results are improved, such as the letter I has gained 60 %, and the letter N has gained 10%. However, there has been a small decline in the letters T and H, about 25% and 5% respectively. The main reason for the decline is because of the similarity of T and I and this problem can be solved by writing different types of T and I's. It's still not guaranteed that the accuracy rate will reach 100% but it will slightly increase if the handwriting is well processed by the system.

However, considering the complexity of the project, the total accuracy of the given sentence is more than 80%.

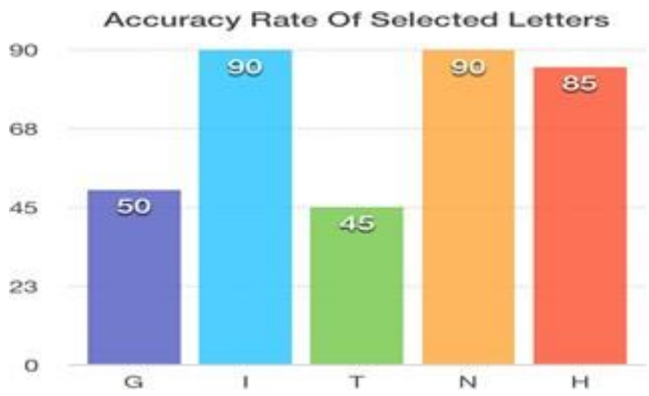


Fig. 28 – Improved Accuracy Rate of Selected Letters

4. SUMMARY

Classification of characters and learning of image processing techniques is done in this project. Also the scheme through which project is achieved is Artificial Neural Network scheme. The result which was got was correct up to more than 90% of the cases, but it would be improved at the end. This work was basically focused on envisaging methods that can efficiently extract feature vectors from each individual character. The method I came up with gave efficient and effective result both for feature extraction as well as recognition.

Many regional languages throughout world have different writing styles which can be recognized with HCR systems using proper algorithm and strategies. We have learning for recognition of English characters. It has been found that recognition of handwritten character becomes difficult due to presence of odd characters or similarity in shapes for multiple characters. Scanned image is pre-processed to get a cleaned image and the characters are isolated into individual characters. Preprocessing work is done in which normalization, filtration is performed using processing steps which produce noise free and clean output. Managing our evolution algorithm with proper training, evaluation other step wise process will lead to successful output of system with better efficiency. Use of some statistical features and geometric features through neural network will provided better recognition result of English characters. This work will be helpful to the researchers for the work towards other script.

Future Scope

The application of this HCR algorithm is extensive. Now-a-days recent advancement in technologies has pushed the limits further for man to get rid of older equipment which posed inconvenience in using. In our case that equipment is a keyboard. There are many situations when using a keyboard is cumbersome like,

- We don‘t get fluency with keyboard as real word writing
- When any key on keyboard is damaged
- Keyboard have scripts on its keys in only one language
- We have to find each character on keyboard which takes time
- In touch-enabled portable devices it is difficult to add a keyboard with much ease

On the other hand if we use OCR software in any device, we can get benefits like,

- Multiple language support
- No keyboard required
- Real world writing style support
- Convenient for touch enabled devices
- Previously hand written record can be documented easily
- Extensive features can also be added to the software like,
    1. Translation
    2. Voice reading

9. REFERENCES

[1] Supriya Deshmukh, Leena Ragha, ―*Analysis of Directional Features - Stroke and Contour for Handwritten Character Recognition*‖, IEEE International Advance Computing Conference (IACC 2009) Patiala, India, 6-7 March 2009.

[2] Sung-Bae Cho, ―*Neural-Network Classifiers for Recognizing Totally Unconstrained Handwritten Numerals*‖, IEEE Transactions on Neural Networks, Vol. 8, No. 1, January 1997

[3] M. Blumenstein, B. K. Verma and H. Basli, ―*A Novel Feature ExtractionTechnique for the Recognition of Segmented Handwritten Characters",* 7th International Conference on Document Analysis and Recognition (ICDAR '03) Eddinburgh, Scotland: pp.137-141, 2003.

[4] Tobias Blanke Michael Bryant Mark Hedges. *"Open source optical character recognition for historical research"*. Journal of Documentation, Vol. 68 Iss 5 pp. 659 - 683.

[5] Anita Pal and Davashankar Singh, *"Handwritten English Character Recognition Using Neural Network"*, International Journal of Computer Science and Communication, 2011, pp: 141-144.

[6] J.Pradeep, E.Srinivasan, S.Himavathi, *"Diagonal Based Feature Extraction For Handwritten Character Recognition System Using Neural Network",* International Conference of Electronics Computer Technology (ICECT), Volume-4, 2011, pp. 364-368.

[7] Sandhya Arora, *"Combining Multiple Feature Extraction Techniques for Handwritten Character Recognition",* IEEE Region 10 Colloquium and the Third ICIIS, Kharagpur, India, December 2008.

[8] C. L. Liu, H. Fujisawa, *"Classification and Learning for Character Recognition: Comparison of Methods and Remaining Problems"*, Int. Workshop on Neural Networks and Learning in Document Analysis and Recognition, Seoul, 2005.

[9] Isha Vats, Shamandeep Singh, *“Offline Handwritten English Numerals Recognition using Correlation Method”*. International Journal of Engineering Research and Technology (IJERT): ISSN: 2278-0181 Vol. 3 Issue 6, June 2014.

[10] Anita Pal and Davashankar Singh, *“Handwritten English Character Recognition Using Neural Network”,* International Journal of Computer Science and Communication, pp: 141-144, 2011.

[11] U. Pal, T. Wakabayashi and F. Kimura, *“Handwritten numeral recognition of six popular scripts”,* Ninth International conference on Document Analysis and Recognition ICDAR 07, Vol.2, pp.749-753, 2007.

[12] Salvador España-Boquera, Maria J. C. B., Jorge G. M. and Francisco Z. M., *“Improving Offline Handwritten Text Recognition with Hybrid HMM/ANN Models”*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 33, No. 4, April 2011.

[13] U. Bhattacharya, and B. B. Chaudhuri, *“Handwritten numeral databases of Indian scripts and multistage recognition of mixed numerals”,* IEEE Transaction on Pattern Analysis and Machine Intelligence, vol.31, No.3, pp.444-457, 2009.

[14] Ashok M. Sapkal Suruchi G. Dedgaonkar, Anjali A. Chandavale. *“Survey of Methods for Character Recognition”*. International Journal of Engineering and Innovative Technology (IJEIT) Volume 1, Issue 5, page 180-188. May 2012.

[15] E. Tautu and F. Leon, *“Optical Character Recognition System Using Support Vector Machines”,* International Journal of Engineering Research and Technology (IJERT): pp. 1-13, 2012.

APPENDIX

Sample Preprocessor.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Feb  4 19:13:24 2020

@author: Tarun Kumar
"""

from___future___import division
from___future___import print_function

import random
import numpy as np
import cv2


def preprocess(img, imgSize, dataAugmentation=False):
	"put img into target img of size imgSize, transpose for TF and
	normalize gray-values"

	# there are damaged files in IAM dataset - just use black
	image instead
	if img is None:
		img = np.zeros([imgSize[1], imgSize[0]])

	# increase dataset size by applying random stretches to the
	images
	if dataAugmentation:
		stretch = (random.random() - 0.5) # -0.5 .. +0.5
		wStretched = max(int(img.shape[1] * (1 + stretch)), 1)
		# random width, but at least 1
		img = cv2.resize(img, (wStretched, img.shape[0]))
		# stretch horizontally by factor 0.5 .. 1.5

	# create target image and copy sample image into it
	(wt, ht) = imgSize
	(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)) # scale according to f (result at least 1 and at most
	wt or ht)

	img = cv2.resize(img, newSize)
	target = np.ones([ht, wt]) * 255
	target[0:newSize[1], 0:newSize[0]] = img

	# transpose for TF
	img = cv2.transpose(target)
```

```
        # normalize
        (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 Loader.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Feb  4 19:01:44 2020

@author: Tarun Kumar
"""

from___future___import division
from___future___import print_function

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


class Sample:
        "sample from the dataset"
        def_init_(self, gtText, filePath):
                self.gtText = gtText
                self.filePath = filePath


class Batch:
        "batch containing images and ground truth texts"
        def_init_(self, gtTexts, imgs):
                self.imgs = np.stack(imgs, axis=0)
                self.gtTexts = gtTexts


class DataLoader:
        "loads data which corresponds to IAM format, see:
        http://www.fki.inf.unibe.ch/databases/iam-handwriting-
        database"

        def_init_(self, filePath, batchSize, imgSize, maxTextLen):
                "loader for dataset at given location, preprocess images
                and text according to parameters"

                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


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


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


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


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

```
	def getNext(self):
		"iterator"
		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.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Feb  4 19:15:15 2020

@author: Tarun Kumar
"""

from___future___import division
from___future___import print_function

import sys
import numpy as np
import tensorflow as tf #import tensorflow.compat.v1 as tf
import os


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


class Model:
	"minimalistic TF model for HTR"

	# model constants
	batchSize = 50
	imgSize = (128, 32)
	maxTextLen = 32

	def_init_(self, charList, decoderType=DecoderType.BestPath, 
mustRestore=False, dump=False):
		"init model: add CNN, RNN and CTC and initialize TF"
		self.dump = dump
		self.charList = charList
		self.decoderType = decoderType
		self.mustRestore = mustRestore
		self.snapID = 0
```

```
        # Whether to use normalization over a batch or a
population
        self.is_train = tf.placeholder(tf.bool, name='is_train')

        # input image batch
        self.inputImgs = tf.placeholder(tf.float32, shape=(None,
Model.imgSize[0], Model.imgSize[1]))

        # setup CNN, RNN and CTC
        self.setupCNN()
        self.setupRNN()
        self.setupCTC()

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

        # initialize TF
        (self.sess, self.saver) = self.setupTF()


    def setupCNN(self):
        "create CNN layers and return output of these layers"
        cnnIn4d = tf.expand_dims(input=self.inputImgs, 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)

        # create layers
        pool = cnnIn4d # input to first CNN layer
        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])

            # basic cells which is used to build RNN
            numHidden = 256
            cells = [tf.contrib.rnn.LSTMCell(num_units=numHidden,
state_is_tuple=True) for _ in range(2)] # 2 layers

            # stack basic cells
            stacked = tf.contrib.rnn.MultiRNNCell(cells,
state_is_tuple=True)

            # bidirectional RNN
            # BxTxF -> BxTx2H
            ((fw, bw), _) =
tf.nn.bidirectional_dynamic_rnn(cell_fw=stacked, cell_bw=stacked,
inputs=rnnIn3d, dtype=rnnIn3d.dtype)

            # BxTxH + BxTxH -> BxTx2H -> BxTx1X2H
            concat = tf.expand_dims(tf.concat([fw, bw], 2), 2)

            # project output to chars (including blank): BxTx1x2H ->
BxTx1xC -> BxTxC
            kernel = tf.Variable(tf.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):
            "create CTC loss and decoder and return them"
            # BxTxC -> TxBxC
            self.ctcIn3dTBC = tf.transpose(self.rnnOut3d, [1, 0, 2])
            # ground truth text as sparse tensor
            self.gtTexts = tf.SparseTensor(tf.placeholder(tf.int64,
shape=[None, 2]) , tf.placeholder(tf.int32, [None]),
tf.placeholder(tf.int64, [2]))

            # calc loss for batch
            self.seqLen = tf.placeholder(tf.int32, [None])
            self.loss =
tf.reduce_mean(tf.nn.ctc_loss(labels=self.gtTexts,
inputs=self.ctcIn3dTBC, sequence_length=self.seqLen,
ctc_merge_repeated=True))

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

            # decoder: either best path decoding or beam search
decoding
            if self.decoderType == DecoderType.BestPath:
```

```
                self.decoder =
tf.nn.ctc_greedy_decoder(inputs=self.ctcIn3dTBC,
sequence_length=self.seqLen)
           elif self.decoderType == DecoderType.BeamSearch:
                self.decoder =
tf.nn.ctc_beam_search_decoder(inputs=self.ctcIn3dTBC,
sequence_length=self.seqLen, beam_width=50, merge_repeated=False)
           elif self.decoderType == DecoderType.WordBeamSearch:
                # import compiled word beam search operation (see
https://github.com/githubharald/CTCWordBeamSearch)
                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.ctcIn3dT
BC, dim=2), 50, 'Words', 0.0, corpus.encode('utf8'),
chars.encode('utf8'), wordChars.encode('utf8'))


      def setupTF(self):
           "initialize TF"
           print('Python: '+sys.version)
           print('Tensorflow: '+tf._version_)

           sess=tf.Session() # TF session

           saver = tf.train.Saver(max_to_keep=1) # 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):
		"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)
			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):
		"extract texts from output of CTC decoder"

		# 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.inputImgs : batch.imgs, self.gtTexts : 
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.ctcIn3dTBC] if 
evalRnnOutput else [])
		feedDict = {self.inputImgs : 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.gtTexts : 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.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Feb  4 19:19:01 2020

@author: Tarun Kumar
"""

from __future__ import division
from __future__ import print_function

import sys
import argparse
import cv2
import editdistance
from DataLoader import DataLoader, Batch
from Model import Model, DecoderType
from SamplePreprocessor import preprocess


class FilePaths:
      "filenames and paths to data"
      fnCharList = 'Data/charList.txt'
      fnAccuracy = 'Data/accuracy.txt'
      fnTrain = 'Data/'
      fnInfer = 'Data/test.png'
```

```
	fnCorpus = 'Data/corpus.txt'


def train(model, loader):
	"train NN"
	epoch = 0 # number of training epochs since start
	bestCharErrorRate = float('inf') # best valdiation character 
error rate
	noImprovementSince = 0 # number of epochs no improvement of 
character error rate occured
	earlyStopping = 5 # stop training after this number of epochs 
without improvement
	while True:
		epoch += 1
		print('Epoch:', epoch)

		# train
		print('Train NN')
		loader.trainSet()
		while loader.hasNext():
			iterInfo = loader.getIteratorInfo()
			batch = loader.getNext()
			loss = model.trainBatch(batch)
			print('Batch:', iterInfo[0],'/', iterInfo[1], 
'Loss:', loss)

		# validate
		charErrorRate = validate(model, loader)

		# if best validation accuracy so far, save model 
parameters
		if charErrorRate < bestCharErrorRate:
			print('Character error rate improved, save model')
			bestCharErrorRate = charErrorRate
			noImprovementSince = 0
			model.save()
			open(FilePaths.fnAccuracy, 'w').write('Validation 
character error rate of saved model: %f%%' % (charErrorRate*100.0))
		else:
			print('Character error rate not improved')
			noImprovementSince += 1

		# stop training if no more improvement in the last x 
epochs
		if noImprovementSince >= earlyStopping:
			print('No more improvement since %d epochs. Training 
stopped.' % earlyStopping)
			break


def validate(model, loader):
	"validate NN"
	print('Validate NN')
	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)

		print('Ground truth -> Recognized')
		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] + '"', '->', '"' + recognized[i] + '"')

	# print validation result
	charErrorRate = numCharErr / numCharTotal
	wordAccuracy = numWordOK / numWordTotal
	print('Character error rate: %f%%. Word accuracy: %f%%.' %
(charErrorRate*100.0, wordAccuracy*100.0))
	return charErrorRate


def infer(model, fnImg):
	"recognize text in image provided by file path"
	img = preprocess(cv2.imread(fnImg, cv2.IMREAD_GRAYSCALE),
Model.imgSize)
	batch = Batch(None, [img])
	(recognized, probability) = model.inferBatch(batch, True)
	print('Recognized:', '"' + recognized[0] + '"')
	print('Probability:', probability[0])


def main():
	"main function"
	# optional command line args
	parser = argparse.ArgumentParser()
	parser.add_argument('--train', help='train the NN',
action='store_true')
	parser.add_argument('--validate', help='validate the NN',
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='dump output of NN to CSV
file(s)', action='store_true')

	args = parser.parse_args()

	decoderType = DecoderType.BestPath
```

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

	# train or validate on IAM dataset
	if args.train or args.validate:
		# load training data, create TF model
		loader = DataLoader(FilePaths.fnTrain, Model.batchSize,
Model.imgSize, Model.maxTextLen)

		# save characters of model for inference mode
		open(FilePaths.fnCharList,
'w').write(str().join(loader.charList))

		# save words contained in dataset into file
		open(FilePaths.fnCorpus, 'w').write(str('
').join(loader.trainWords + loader.validationWords))

		# execute training or validation
		if args.train:
			model = Model(loader.charList, decoderType)
			train(model, loader)
		elif args.validate:
			model = Model(loader.charList, decoderType,
mustRestore=True)
			validate(model, loader)

	# infer text on test image
	else:
		print(open(FilePaths.fnAccuracy).read())
		model = Model(open(FilePaths.fnCharList).read(),
decoderType, mustRestore=True, dump=args.dump)
		infer(model, FilePaths.fnInfer)


if___name___== '_main_':
	main()
```