# Transformative Handwriting Deciphering using ML algorithms

**A MINI PROJECT REPORT**

18CSC305J - ARTIFICIAL INTELLIGENCE

Submitted by

**Anubhav Bhutani [RA2111026010359]**

**Aryan Pandey [RA2111026010360]**

**Rishikesh Bharadwaj [RA2111026010356]**

Under the guidance of

**Dr. Karpagam M**

Assistant Professor, Department of Computational Intelligence ***in partial fulfillment for the award of the degree***

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

Blue and white logo with text

Description automatically generated

**S.R.M. Nagar, Kattankulathur, Chengalpattu District**

**MAY 2024**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**(Under Section 3 of UGC Act, 1956)**

**BONAFIDE CERTIFICATE**

Certified that Mini project report titled “**Transformative Handwriting Deciphering using ML algorithms**” is the bona fide work of **Anubhav Bhutani (359), Rishikesh Bharadwaj (356) and Aryan Pandey (360)** who carried out the minor project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

# SIGNATURE

**Dr.Karpagam M**

# Assistant Professor

CINTEL

# 

# ABSTRACT

# Transformative Handwriting Deciphering using ML algorithms has been a longstanding

# challenge in the field of computer vision, with diverse applications ranging from document digitization to interactive user interfaces. This study presents a robust and efficient approach for handwriting deciphering using computer vision techniques. The proposed system leverages convolutional neural networks (CNNs) for feature extraction and classification, enabling the deciphering of handwritten characters and words. Preprocessing techniques, including image binarization and noise reduction, are employed to enhance the quality of input images. The CNN model is trained on a large and diverse dataset of handwritten samples to learn discriminative features for accurate deciphering. Additionally, the system incorporates post-processing methods, such as language modelling and contextual analysis, to improve deciphering accuracy in real-world scenarios. Experimental results demonstrate the effectiveness of the proposed approach, achieving high accuracy rates across various handwriting styles and languages. The developed system exhibits promising potential for applications in digitizing historical documents, automatic form processing, and enhancing human-computer interaction through intuitive handwriting input methods. Furthermore, the proposed handwriting deciphering system is designed with scalability and adaptability in mind, making it suitable for both offline and online deciphering tasks. It showcases remarkable performance even in challenging conditions, such as skewed or distorted handwriting, ensuring its applicability in diverse real-world environments.

**TABLE OF CONTENTS**

[**ABSTRACT iii**](file:///F:\Yash\College\CourseAndSyllabus\6th%20Semester\Artificial%20Intelligence\AI%20Report%20format.docx#_TOC_250002)

**TABLE OF CONTENTS iv**

**LIST OF FIGURES v**

[**ABBREVIATIONS vi**](file:///F:\Yash\College\CourseAndSyllabus\6th%20Semester\Artificial%20Intelligence\AI%20Report%20format.docx#_TOC_250001)

1. INTRODUCTION 7
2. LITERATURE SURVEY 9
3. REQUIREMENTS 11
4. SYSTEM ARCHITECTURE AND DESIGN 13
5. METHODOLOGY 16
6. CODING AND TESTING 20
7. SREENSHOTS AND RESULTS 34
8. CONCLUSION AND FUTURE ENHANCEMENT 37
   1. Conclusion
   2. Future Enhancement

**REFERENCES 39**

**LIST OF FIGURES**

**4.1 System Architecture 9**

**4.2 Neural Network 10**

**4.2.1 CNN 10**

**4.3 Neural Network for Handwriting Deciphering 10**

**4.4 model architecture 11**

**5 Distribution graph with respect to names in the training set 11**

**5.1 Confusion Matrix with file name and identity 12**

**5.2 Output for the images containing handwriting. 13**

# ABBREVIATIONS

**CNN** Convolutional Neural Networks

**RNN** Recurrent Neural Networks

**LSTM** Long Short-Term Memory

**CTC** Connectionist Temporal Classification

**CRNN** Convolutional Recurrent Neural Network

1. **INTRODUCTION**

Transformative Handwriting Deciphering using ML algorithms, often considered an intersection of computer vision and natural language processing, is a pivotal technology in the age of digital transformation. Despite the prevalent use of typed texts, handwritten documents - be it historical scripts, personal notes, or official forms - remain omnipresent. This project explores the domain of handwriting deciphering with a specific focus on names, leveraging a vast dataset collected from initiatives.

**1.1 Motivation**

In the digital age, where technology is seamlessly integrating with every aspect of our lives, the capacity to interpret and process handwritten information is an invaluable asset. Handwritten data, from historical manuscripts to personal notes, holds immense potential. Deciphering this information and converting it to a digital format not only makes it accessible to a broader audience but also ensures its preservation for future generations. Moreover, with the emergence of interactive smart devices, handwriting deciphering can enhance user experience significantly, opening the door to various innovative applications.

**1.2 Objective**

The primary objective of this project is to harness the power of deep learning to recognize and digitize handwritten names. Using the vast dataset sourced from charity projects, the aim is to achieve a deciphering accuracy of over 85%, transforming handwritten characters into coherent digital text.

**1.3 Problem Statement**

Despite the availability of a vast dataset comprising over four hundred thousand handwritten names, converting them into accurate digital text remains a daunting task. The inherent variability in handwriting styles, irregularities in character formation, and potential for errors or distortions present a unique set of challenges. This project seeks to tackle these challenges by designing and training a Convolutional Recurrent Neural Network (CRNN) model using Connectionist Temporal Classification Loss (CTC Loss).

**1.4 Challenges**

* **Data Quality and Preprocessing:** Handwritten datasets often contain noise, distortions, and inconsistencies that can hinder the model's performance. Cleaning and preprocessing the data effectively are crucial.
* **Variability in Handwriting Styles:** Every individual possesses a unique handwriting style. Accounting for this immense variability to ensure accurate deciphering is a significant challenge.
* **Model Complexity:** Designing a neural network that can handle the intricacies of handwritten text without becoming overly complex or resource-intensive is a delicate balancing act.
* **Achieving High Accuracy:** While the preliminary results are promising, achieving an accuracy of over 85% remains a challenge.

1. **LITERATURE SURVEY**

The development and success of handwriting deciphering have been well-documented in academic and industry research. Key advancements and strategies from existing literature include:

**Evolution of Transformative Handwriting deciphering Techniques**

* **Pattern Matching:** Initial techniques revolved around basic pattern matching and template-based deciphering systems.
* **Feature Extraction:** Advanced methods began extracting specific attributes or features of handwritten characters, like loops, curves, and slants for deciphering.

**Neural Networks and Handwriting deciphering techniques**

* **Convolutional Neural Networks (CNNs):** With their ability to detect patterns and hierarchies in visual data, CNNs revolutionized handwriting deciphering.
* **Recurrent Neural Networks (RNNs):** Recognizing the sequential nature of text, RNNs and their variants were integrated into handwriting deciphering systems. These networks are adept at processing sequences and maintaining context over longer spans of data.

**Connectionist Temporal Classification Loss (CTC Loss)**

This specific loss function emerged as a game-changer for handwriting deciphering. CTC Loss optimizes both the length and classes of the predicted data, handling the variable lengths of handwriting and the sequencing of characters.

**Trends in Neural Networks for Handwriting Deciphering**

1. **Integration of CNN with RNN:** Convolutional Recurrent Neural Networks (CRNN): By integrating the spatial feature detection capabilities of CNNs with the sequential processing power of RNNs, CRNNs have become the preferred model for many handwriting deciphering tasks.
2. **Long Short-Term Memory Networks (LSTMs):** LSTMs, a type of RNN, remember patterns over extended sequences, making them particularly suitable for tasks like handwriting deciphering where context and sequence matter.
3. **Attention Mechanisms in Handwriting deciphering:** Borrowed from machine translation tasks, attention mechanisms have been applied to handwriting deciphering to allow models to 'focus' on specific parts of the data, enhancing accuracy.
4. **Transfer Learning and Handwriting Deciphering:** Recent studies suggest that models pre-trained on large datasets can be fine-tuned for specific tasks like handwriting deciphering. This method optimizes results while reducing the need for vast amounts of training data and computational power.

**3.** **REQUIREMENTS**

**Data Requirements:** The project requires three datasets: 'written\_name\_train\_v2.csv', 'written\_name\_validation\_v2.csv', and 'written\_name\_test\_v2.csv'. These datasets should contain the file names and identities of the handwritten images. The images should be stored in 'train\_v2/train/', 'validation\_v2/validation/', and 'test\_v2/test/' directories respectively.

**Software Requirements:** The project requires Python programming language with several libraries including os, cv2, random, numpy, pandas, matplotlib, tensorflow, and keras. These libraries are used for data manipulation, image processing, and machine learning tasks.

**Hardware Requirements:** As the project involves training a deep learning model, it requires a computer with a high-performance GPU for faster computation. Also, sufficient storage is required to store the datasets and the trained model.

**Functional Requirements:** The project should be able to perform the following tasks:

**Load and preprocess the data:** This includes reading the data, visualizing the images, cleaning the data by checking and dropping null values, removing unreadable data, converting lowercase labels to uppercase, and preprocessing the images.

**Train the model:** This includes defining the model architecture, compiling the model, and training the model using the training data.

**Validate the model:** This includes predicting the labels for the validation data and calculating the accuracy of the predictions.

**Test the model:** This includes predicting the labels for the test data and visualizing the results.

**Non-functional Requirements:** The project should ensure the accuracy of the handwriting deciphering. The model should be trained and validated properly to achieve high accuracy. The project should also ensure the efficiency of the code. The code should be written in a way that it can be executed in a reasonable amount of time.

**Constraints:** The project is constrained by the quality and quantity of the data. The accuracy of the handwriting deciphering depends on the quality and quantity of the handwritten images and their labels. The project is also constrained by the computational resources. The training of the deep learning model requires high computational resources.

**Maintainability Requirements:** The code should be written in a modular manner, making it easy to update or modify individual components without affecting the rest of the system. This includes using functions or classes to encapsulate functionality and separating the code into different files or modules based on functionality.

**Scalability Requirements:** The system should be designed in a way that it can handle an increase in the size of the data. This includes using efficient data structures and algorithms, as well as taking advantage of parallel processing capabilities of the hardware.

**4.** **SYSTEM ARCHITECTURE AND DESIGN**

The architecture and design of the handwriting deciphering system is primarily based on a deep learning model that leverages Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and a Connectionist Temporal Classification (CTC) layer. This model is responsible for transforming an image of a handwritten text into a sequence of characters.

**4.1 System Architecture**

The handwriting deciphering system is designed as a pipeline of several components, each responsible for a specific task. The main components of the system are:

**Database Component:** This component is responsible for managing the data storage and retrieval. It stores the training, validation, and testing datasets, as well as the trained model. It retrieves the necessary data for the preprocessing component and stores the output of the post-processing component.

**Data Preprocessing Component:** This component is responsible for loading and preprocessing the data. It ensures that the data is in the correct format and structure for the subsequent components.

**Deep Learning Model Component:** This is the core component of the system. It consists of several layers including Convolutional Neural Networks (CNNs) for feature extraction, Recurrent Neural Networks (RNNs) for sequence modelling, and a Connectionist Temporal Classification (CTC) layer for sequence labelling. This component transforms an image of a handwritten text into a sequence of characters.

**Training Component:** This component is responsible for training the deep learning model using the training data. It uses the Adam optimizer and a special loss function for CTC.

**Validation and Testing Component:** This component is responsible for validating and testing the trained model. It measures the performance of the model on separate validation and testing datasets.

**Post-Processing Component:** After the model makes a prediction, this component takes over. It is responsible for interpreting the model's predictions, converting the sequence of characters into a human-readable format, and handling any post-processing tasks such as correcting common mistakes or formatting the output.

A diagram of a computer system

Description automatically generated

**4.2 Neural Network**

**A diagram of a function

Description automatically generated**

Deep Learning, a subset of Machine Learning, is heavily reliant on **Neural Networks**, which serve as the cornerstone for deep learning techniques and algorithms. Deep learning's name reflects its depth, signifying the substantial number of layers within a neural network. Neural networks are computational systems designed to emulate the human brain's functionality, with their basic building blocks referred to as neurons, drawing inspiration from biological neural networks. The fundamental components of a neural network encompass inputs, weights assigned to connection links, biases, and the ultimate output. In neural network terminology, each computational unit is commonly referred to as a **perceptron**.

A **Convolutional Neural Network** (CNN) represents a specific category within the domain of deep neural networks, tailored for tasks involving visual image analysis. CNNs are characterized by their multilayered architecture, wherein each neuron in one layer establishes connections with all neurons in the subsequent layer, forming a fully connected network. CNNs adopt a three-dimensional structure where clusters of neurons collaborate to analyse specific image regions known as features. This network operates through three key stages.

The initial stage, the convolution layer, scrutinizes the input image's pixels to grasp essential features and yields a feature map as the output. Subsequently, the pooling stage is responsible for both reducing dimensionality within the feature space and preserving critical information. The third and final stage encompasses the flattening of matrices into a vector, which is then fed into a fully connected layer responsible for aggregating features to make informed decisions regarding the image's classification.

A diagram of a diagram of a layer

Description automatically generated with medium confidence

**5.** **METHODOLOGY**

**5.1 Neural Network for Handwriting Deciphering**

The architecture is a combination of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) for the task of handwriting deciphering.

A diagram of a diagram

Description automatically generated

**Input Layer:**

Input shape: (256, 64, 1)

This is the input layer for grayscale images of size 256x64.

**Convolutional Layers:**

**1. Convolutional Layer (conv1):**

* Filters: 32
* Kernel Size: (3, 3)
* Padding: 'same'
* Activation Function: ReLU
* Batch Normalization
* Max Pooling Layer (max1) with pool size (2, 2)
* Output Shape: (128, 32, 32, 32)

**2. Convolutional Layer (conv2):**

* Filters: 64
* Kernel Size: (3, 3)
* Padding: 'same'
* Activation Function: ReLU
* Batch Normalization
* Max Pooling Layer (max2) with pool size (2, 2)
* Dropout Layer with a rate of 0.3
* Output Shape: (64, 16, 64, 64)

**3. Convolutional Layer (conv3):**

* Filters: 128
* Kernel Size: (3, 3)
* Padding: 'same'
* Activation Function: ReLU
* Batch Normalization
* Max Pooling Layer (max3) with pool size (1, 2)
* Dropout Layer with a rate of 0.3
* Output Shape: (64, 16, 32, 128)

**CNN to RNN Transition:**

**1. Reshape Layer (reshape):**

* Reshapes the output from the last convolutional layer to (64, 1024).
* The reshaped tensor is passed to the RNN layers.
* Dense Layer (dense1):
* Units: 64
* Activation Function: ReLU
* Output Shape: (64, 64)

**Recurrent Neural Networks (RNN):**

**1. Bidirectional LSTM Layer (lstm1):**

* Units: 256
* Return Sequences: True (to return sequences for subsequent layers)
* Output Shape: (64, 64, 512)
* Bidirectional LSTM for bidirectional sequence processing.

**2. Bidirectional LSTM Layer (lstm2):**

* Units: 256
* Return Sequences: True
* Output Shape: (64, 64, 512)

**3. Output Layer:**

* Dense Layer (dense2):
* Units: num\_of\_characters (the number of output classes)
* Output Shape: (64, num\_of\_characters)
* Activation Function: Softmax
* The softmax activation assigns probabilities to each character class.

**5.2 Model Architecture**

The model is fed by the image through the Gated CNN, processed using the Bahdanau’s attention, with bidirectional GRU. Finally, GRU’s output matrix is passed to the Connectionist Temporal Classification (CTC [[32](https://www.mdpi.com/2313-433X/6/12/141#B32-jimaging-06-00141)]) to calculate the loss value and decode the output matrix into the final text. The model architecture, which has four primary parts: an encoder, an attention block, a decoder, and CTC, is shown below

A diagram of a diagram of a structure

Description automatically generated with medium confidence

#### **Encoder**

#### **1. Convolutional Blocks:** The encoder receives the input and generates the feature vectors. These feature vectors hold the information and the characteristics that represent the input. The encoder network consists of 3 convolutional blocks that correspond to training to extract relevant features from the images. Each block consists of a convolution operation, which applies a filter kernel of size (3,3) in all the blocks. Parametric Rectified Linear Unit (ReLU) and Batch Normalization are applied. To reduce overfitting, we also use Dropout at some of the convolutional layers (with dropout probability equal to 0.3)

**Decoder**

**1. Sequence Understanding:** The LSTM layers are employed to understand and model the sequential dependencies in the handwritten text. Handwritten text is essentially a sequence of characters, and the order and context of these characters are crucial for accurate deciphering.

**2. Bidirectional Context:** The LSTM layers used in the architecture are bidirectional, meaning they process the input sequence in both forward and backward directions. This bidirectional processing allows the network to capture contextual information from both past and future characters. This is particularly valuable for understanding the formation of characters in handwritten text, as characters can be influenced by neighbouring characters.

**3. Memory and Learning:** LSTM units within these layers have mechanisms for maintaining and updating internal memory. This memory allows the network to remember important information from the past and selectively forget less relevant information. This is crucial for recognizing characters in handwritten text, where strokes and shapes can vary widely.

**4. Output Sequences:** The LSTM layers return sequences of outputs for each input sequence. These sequences are used to maintain information about each character's context, and they serve as input to subsequent layers in the network.

1. **CODING AND TESTING**

Implementing a handwriting deciphering system involves multiple steps such as data preprocessing, feature extraction, model training, and prediction. For implementation using Python and the popular deep learning library TensorFlow and Keras for creating a Convolutional Neural Network (CNN) for handwriting deciphering we need all the required libraries install in the system.

* OpenCV
* Pillow (PIL)
* NumPy
* SciPy
* scikit-image
* TensorFlow
* PyTorch
* pip install torch
* Keras
* Tesseract
* Matplotlib

To train the model we used EMINST Dataset which is very popular link for it can be found [HERE](https://www.kaggle.com/datasets/landlord/handwriting-recognition).

[www.kaggle.com/datasets/landlord/handwriting-recognition](http://www.kaggle.com/datasets/landlord/handwriting-recognition)

**CODE**

import os  
import cv2  
import random  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import tensorflow as tf  
from tensorflow.keras.optimizers import Adam  
from keras import backend as K  
from keras.models import Model  
from keras.layers import Input, Conv2D, MaxPooling2D, Reshape, Bidirectional, LSTM, Dense, Lambda, Activation, BatchNormalization, Dropout

! pip install kaggle

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/  
Requirement already satisfied: kaggle in /usr/local/lib/python3.7/dist-packages (1.5.12)  
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.7/dist-packages (from kaggle) (2.8.2)  
Requirement already satisfied: certifi in /usr/local/lib/python3.7/dist-packages (from kaggle) (2022.6.15)  
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.7/dist-packages (from kaggle) (1.15.0)  
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from kaggle) (2.23.0)  
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from kaggle) (4.64.0)  
Requirement already satisfied: urllib3 in /usr/local/lib/python3.7/dist-packages (from kaggle) (1.24.3)  
Requirement already satisfied: python-slugify in /usr/local/lib/python3.7/dist-packages (from kaggle) (6.1.2)  
Requirement already satisfied: text-unidecode>=1.3 in /usr/local/lib/python3.7/dist-packages (from python-slugify->kaggle) (1.3)  
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests->kaggle) (3.0.4)  
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests->kaggle) (2.10)

! mkdir ~/.kaggle

mkdir: cannot create directory ‘/root/.kaggle’: File exists

! cp kaggle.json ~/.kaggle/

! chmod 600 ~/.kaggle/kaggle.json

! kaggle datasets download landlord/handwriting-recognition

Downloading handwriting-recognition.zip to /content  
100% 1.25G/1.26G [00:39<00:00, 37.9MB/s]  
100% 1.26G/1.26G [00:39<00:00, 34.4MB/s]

train = pd.read\_csv('written\_name\_train\_v2.csv')  
valid = pd.read\_csv('written\_name\_validation\_v2.csv')

# Viewing the data  
plt.figure(figsize=(15, 10))  
  
# To view first 6 images  
for i in range(6):  
 # plt.subplot(nrows, ncols, index)  
 ax = plt.subplot(2, 3, i+1)  
 img\_dir = 'train\_v2/train/'+train.loc[i, 'FILENAME']  
 # cv2.imread(path=img\_dir, flag=specifies the way in which image should be read)  
 image = cv2.imread(img\_dir, cv2.IMREAD\_GRAYSCALE)  
 # cv2.imread(X=data of the image, colormap instance)  
 plt.imshow(image, cmap = 'gray')  
 plt.title(train.loc[i, 'IDENTITY'], fontsize=12)  
 plt.axis('off')  
# Adjusts the subplot layout parameters  
plt.subplots\_adjust(wspace=0.2, hspace=-0.8)

A black and white rectangular object with text

Description automatically generated

# Cleaning the data by checking for null values  
print("Number of NaNs in train set : ", train['IDENTITY'].isnull().sum())  
print("Number of NaNs in validation set : ", valid['IDENTITY'].isnull().sum())

Number of NaNs in train set : 565  
Number of NaNs in validation set : 78

# Dropping the null values from the training and testing data  
train.dropna(axis=0, inplace=True)  
valid.dropna(axis=0, inplace=True)

# Removing the data with the label 'UNREADABLE'  
unreadable = train[train['IDENTITY'] == 'UNREADABLE']  
# resets the index of the DataFrame, and uses the default one instead  
unreadable.reset\_index(inplace = True, drop=True)  
  
plt.figure(figsize=(15, 10))  
  
for i in range(6):  
 ax = plt.subplot(2, 3, i+1)  
 img\_dir = 'train\_v2/train/'+unreadable.loc[i, 'FILENAME']  
 image = cv2.imread(img\_dir, cv2.IMREAD\_GRAYSCALE)  
 plt.imshow(image, cmap = 'gray')  
 plt.title(unreadable.loc[i, 'IDENTITY'], fontsize=12)  
 plt.axis('off')  
  
plt.subplots\_adjust(wspace=0.2, hspace=-0.8)

A black and white rectangle with text

Description automatically generated

train = train[train['IDENTITY'] != 'UNREADABLE']  
valid = valid[valid['IDENTITY'] != 'UNREADABLE']

# converting some lowercase labels to uppercase to maintain uniformity  
train['IDENTITY'] = train['IDENTITY'].str.upper()  
valid['IDENTITY'] = valid['IDENTITY'].str.upper()

# resetting the index to maintain uniform leveling  
train.reset\_index(inplace = True, drop=True)  
valid.reset\_index(inplace = True, drop=True)

def preprocess(img):  
 (h, w) = img.shape  
 # blank white image  
 final\_img = np.ones([64, 256])\*255  
  
 # cropping the image if the width and height are greater than  
 # 256 and 64 respectively  
 if w > 256:  
 img = img[:, :256]  
  
 if h > 64:  
 img = img[:64, :]  
  
 # rotating the image clockwise to bring the image shape to (x,y)  
 final\_img[:h, :w] = img  
 return cv2.rotate(final\_img, cv2.ROTATE\_90\_CLOCKWISE)

# training the model 30000 images and the validating it on 3000 images  
train\_size = 30000  
valid\_size= 3000

# training  
train\_x = []  
  
for i in range(train\_size):  
 img\_dir = 'train\_v2/train/'+train.loc[i, 'FILENAME']  
 image = cv2.imread(img\_dir, cv2.IMREAD\_GRAYSCALE)  
 image = preprocess(image)  
 image = image/255.  
 train\_x.append(image)

# validating  
valid\_x = []  
  
for i in range(valid\_size):  
 img\_dir = 'validation\_v2/validation/'+valid.loc[i, 'FILENAME']  
 image = cv2.imread(img\_dir, cv2.IMREAD\_GRAYSCALE)  
 image = preprocess(image)  
 image = image/255.  
 valid\_x.append(image)

# .reshape(a: array\_like = -1(we want numpy to figure out as the dimensions are unknown))  
train\_x = np.array(train\_x).reshape(-1, 256, 64, 1)  
valid\_x = np.array(valid\_x).reshape(-1, 256, 64, 1)

# labels are converted to numbers representing each character  
# labels are then prepared for Connectionist Temporal Classification Loss (CTC Loss)  
alphabets = u"ABCDEFGHIJKLMNOPQRSTUVWXYZ-' "  
# max length of input labels  
max\_str\_len = 24  
# +1 for ctc pseudo blank  
num\_of\_characters = len(alphabets) + 1  
# max length of predicted labels  
num\_of\_timestamps = 64  
  
  
def label\_to\_num(label):  
 label\_num = []  
 for ch in label:  
 label\_num.append(alphabets.find(ch))  
  
 return np.array(label\_num)  
  
def num\_to\_label(num):  
 ret = ""  
 for ch in num:  
 if ch == -1: # CTC Blank  
 break  
 else:  
 ret+=alphabets[ch]  
 return ret

name = 'YASH'  
print(name, '\n',label\_to\_num(name))

YASH   
 [24 0 18 7]

# train\_y contains the true labels converted to numbers and padded with -1.  
# The length of each label is equal to max\_str\_len.  
train\_y = np.ones([train\_size, max\_str\_len]) \* -1  
# train\_label\_len contains the length of each true label (without padding)  
train\_label\_len = np.zeros([train\_size, 1])  
# train\_input\_len contains the length of each predicted label.  
# The length of all the predicted labels is constant i.e number of timestamps - 2.  
train\_input\_len = np.ones([train\_size, 1]) \* (num\_of\_timestamps-2)  
# train\_output is a dummy output for ctc loss.  
train\_output = np.zeros([train\_size])  
  
for i in range(train\_size):  
 train\_label\_len[i] = len(train.loc[i, 'IDENTITY'])  
 train\_y[i, 0:len(train.loc[i, 'IDENTITY'])]= label\_to\_num(train.loc[i, 'IDENTITY'])

valid\_y = np.ones([valid\_size, max\_str\_len]) \* -1  
valid\_label\_len = np.zeros([valid\_size, 1])  
valid\_input\_len = np.ones([valid\_size, 1]) \* (num\_of\_timestamps-2)  
valid\_output = np.zeros([valid\_size])  
  
for i in range(valid\_size):  
 valid\_label\_len[i] = len(valid.loc[i, 'IDENTITY'])  
 valid\_y[i, 0:len(valid.loc[i, 'IDENTITY'])]= label\_to\_num(valid.loc[i, 'IDENTITY'])

print('True label : ',train.loc[100, 'IDENTITY'] , '\ntrain\_y : ',train\_y[100],'\ntrain\_label\_len : ',train\_label\_len[100],  
 '\ntrain\_input\_len : ', train\_input\_len[100])

True label : NOUR   
train\_y : [13. 14. 20. 17. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  
 -1. -1. -1. -1. -1. -1.]   
train\_label\_len : [4.]   
train\_input\_len : [62.]

input\_data = Input(shape=(256, 64, 1), name='input')  
  
inner = Conv2D(32, (3, 3), padding='same', name='conv1', kernel\_initializer='he\_normal')(input\_data)  
inner = BatchNormalization()(inner)  
inner = Activation('relu')(inner)  
inner = MaxPooling2D(pool\_size=(2, 2), name='max1')(inner)  
  
inner = Conv2D(64, (3, 3), padding='same', name='conv2', kernel\_initializer='he\_normal')(inner)  
inner = BatchNormalization()(inner)  
inner = Activation('relu')(inner)  
inner = MaxPooling2D(pool\_size=(2, 2), name='max2')(inner)  
inner = Dropout(0.3)(inner)  
  
inner = Conv2D(128, (3, 3), padding='same', name='conv3', kernel\_initializer='he\_normal')(inner)  
inner = BatchNormalization()(inner)  
inner = Activation('relu')(inner)  
inner = MaxPooling2D(pool\_size=(1, 2), name='max3')(inner)  
inner = Dropout(0.3)(inner)  
  
# CNN to RNN  
inner = Reshape(target\_shape=((64, 1024)), name='reshape')(inner)  
inner = Dense(64, activation='relu', kernel\_initializer='he\_normal', name='dense1')(inner)  
  
## RNN  
inner = Bidirectional(LSTM(256, return\_sequences=True), name = 'lstm1')(inner)  
inner = Bidirectional(LSTM(256, return\_sequences=True), name = 'lstm2')(inner)

## OUTPUT  
inner = Dense(num\_of\_characters, kernel\_initializer='he\_normal',name='dense2')(inner)  
y\_pred = Activation('softmax', name='softmax')(inner)  
  
model = Model(inputs=input\_data, outputs=y\_pred)  
model.summary()

Model: "model"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 input (InputLayer) [(None, 256, 64, 1)] 0   
   
 conv1 (Conv2D) (None, 256, 64, 32) 320   
   
 batch\_normalization (BatchN (None, 256, 64, 32) 128   
 ormalization)   
   
 activation (Activation) (None, 256, 64, 32) 0   
   
 max1 (MaxPooling2D) (None, 128, 32, 32) 0   
   
 conv2 (Conv2D) (None, 128, 32, 64) 18496   
   
 batch\_normalization\_1 (Batc (None, 128, 32, 64) 256   
 hNormalization)   
   
 activation\_1 (Activation) (None, 128, 32, 64) 0   
   
 max2 (MaxPooling2D) (None, 64, 16, 64) 0   
   
 dropout (Dropout) (None, 64, 16, 64) 0   
   
 conv3 (Conv2D) (None, 64, 16, 128) 73856   
   
 batch\_normalization\_2 (Batc (None, 64, 16, 128) 512   
 hNormalization)   
   
 activation\_2 (Activation) (None, 64, 16, 128) 0   
   
 max3 (MaxPooling2D) (None, 64, 8, 128) 0   
   
 dropout\_1 (Dropout) (None, 64, 8, 128) 0   
   
 reshape (Reshape) (None, 64, 1024) 0   
   
 dense1 (Dense) (None, 64, 64) 65600   
   
 lstm1 (Bidirectional) (None, 64, 512) 657408   
   
 lstm2 (Bidirectional) (None, 64, 512) 1574912   
   
 dense2 (Dense) (None, 64, 30) 15390   
   
 softmax (Activation) (None, 64, 30) 0   
   
=================================================================  
Total params: 2,406,878  
Trainable params: 2,406,430  
Non-trainable params: 448  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# the ctc loss function  
def ctc\_lambda\_func(args):  
 y\_pred, labels, input\_length, label\_length = args  
 # the 2 is critical here since the first couple outputs of the RNN  
 # tend to be garbage  
 y\_pred = y\_pred[:, 2:, :]  
 return K.ctc\_batch\_cost(labels, y\_pred, input\_length, label\_length)

labels = Input(name='gtruth\_labels', shape=[max\_str\_len], dtype='float32')  
input\_length = Input(name='input\_length', shape=[1], dtype='int64')  
label\_length = Input(name='label\_length', shape=[1], dtype='int64')  
  
ctc\_loss = Lambda(ctc\_lambda\_func, output\_shape=(1,), name='ctc')([y\_pred, labels, input\_length, label\_length])  
model\_final = Model(inputs=[input\_data, labels, input\_length, label\_length], outputs=ctc\_loss)

# the loss calculation occurs elsewhere, so we use a dummy lambda function for the loss  
model\_final.compile(loss={'ctc': lambda y\_true, y\_pred: y\_pred}, optimizer=Adam(learning\_rate = 0.0001))  
  
model\_final.fit(x=[train\_x, train\_y, train\_input\_len, train\_label\_len], y=train\_output,  
 validation\_data=([valid\_x, valid\_y, valid\_input\_len, valid\_label\_len], valid\_output),  
 epochs=60, batch\_size=128)

Epoch 1/60  
235/235 [==============================] - 69s 203ms/step - loss: 25.1180 - val\_loss: 20.6663  
Epoch 2/60  
235/235 [==============================] - 45s 190ms/step - loss: 20.1380 - val\_loss: 20.2393  
Epoch 3/60  
235/235 [==============================] - 46s 195ms/step - loss: 19.8006 - val\_loss: 19.7221  
Epoch 4/60  
235/235 [==============================] - 44s 189ms/step - loss: 19.3337 - val\_loss: 19.0609  
Epoch 5/60  
235/235 [==============================] - 45s 191ms/step - loss: 18.2791 - val\_loss: 19.1203  
Epoch 6/60  
235/235 [==============================] - 46s 194ms/step - loss: 16.5852 - val\_loss: 16.7888  
Epoch 7/60  
235/235 [==============================] - 45s 191ms/step - loss: 14.5121 - val\_loss: 14.0343  
Epoch 8/60  
235/235 [==============================] - 45s 191ms/step - loss: 12.1958 - val\_loss: 12.1487  
Epoch 9/60  
235/235 [==============================] - 45s 190ms/step - loss: 9.6578 - val\_loss: 9.0626  
Epoch 10/60  
235/235 [==============================] - 46s 196ms/step - loss: 7.6627 - val\_loss: 6.4522  
Epoch 11/60  
235/235 [==============================] - 45s 190ms/step - loss: 6.4474 - val\_loss: 5.6289  
Epoch 12/60  
235/235 [==============================] - 45s 193ms/step - loss: 5.6533 - val\_loss: 4.8463  
Epoch 13/60  
235/235 [==============================] - 46s 197ms/step - loss: 5.0811 - val\_loss: 4.4321  
Epoch 14/60  
235/235 [==============================] - 45s 191ms/step - loss: 4.6071 - val\_loss: 4.0151  
Epoch 15/60  
235/235 [==============================] - 45s 192ms/step - loss: 4.2636 - val\_loss: 3.8602  
Epoch 16/60  
235/235 [==============================] - 49s 208ms/step - loss: 3.9406 - val\_loss: 3.4950  
Epoch 17/60  
235/235 [==============================] - 45s 191ms/step - loss: 3.6948 - val\_loss: 3.2250  
Epoch 18/60  
235/235 [==============================] - 45s 193ms/step - loss: 3.4926 - val\_loss: 3.1590  
Epoch 19/60  
235/235 [==============================] - 46s 195ms/step - loss: 3.3047 - val\_loss: 3.0205  
Epoch 20/60  
235/235 [==============================] - 45s 191ms/step - loss: 3.1578 - val\_loss: 2.8847  
Epoch 21/60  
235/235 [==============================] - 45s 191ms/step - loss: 3.0171 - val\_loss: 2.7828  
Epoch 22/60  
235/235 [==============================] - 46s 195ms/step - loss: 2.8873 - val\_loss: 2.6700  
Epoch 23/60  
235/235 [==============================] - 45s 190ms/step - loss: 2.7744 - val\_loss: 2.5988  
Epoch 24/60  
235/235 [==============================] - 45s 192ms/step - loss: 2.6840 - val\_loss: 2.4984  
Epoch 25/60  
235/235 [==============================] - 46s 196ms/step - loss: 2.5949 - val\_loss: 2.5271  
Epoch 26/60  
235/235 [==============================] - 45s 192ms/step - loss: 2.5156 - val\_loss: 2.4908  
Epoch 27/60  
235/235 [==============================] - 45s 191ms/step - loss: 2.4429 - val\_loss: 2.3946  
Epoch 28/60  
235/235 [==============================] - 46s 197ms/step - loss: 2.3541 - val\_loss: 2.3357  
Epoch 29/60  
235/235 [==============================] - 45s 191ms/step - loss: 2.2949 - val\_loss: 2.2851  
Epoch 30/60  
235/235 [==============================] - 45s 191ms/step - loss: 2.2447 - val\_loss: 2.3537  
Epoch 31/60  
235/235 [==============================] - 45s 190ms/step - loss: 2.1856 - val\_loss: 2.3244  
Epoch 32/60  
235/235 [==============================] - 46s 196ms/step - loss: 2.1482 - val\_loss: 2.2789  
Epoch 33/60  
235/235 [==============================] - 45s 190ms/step - loss: 2.0748 - val\_loss: 2.1897  
Epoch 34/60  
235/235 [==============================] - 45s 190ms/step - loss: 2.0337 - val\_loss: 2.2398  
Epoch 35/60  
235/235 [==============================] - 46s 194ms/step - loss: 1.9799 - val\_loss: 2.1154  
Epoch 36/60  
235/235 [==============================] - 44s 189ms/step - loss: 1.9277 - val\_loss: 2.1708  
Epoch 37/60  
235/235 [==============================] - 45s 190ms/step - loss: 1.9009 - val\_loss: 2.1084  
Epoch 38/60  
235/235 [==============================] - 46s 194ms/step - loss: 1.8664 - val\_loss: 2.0915  
Epoch 39/60  
235/235 [==============================] - 45s 191ms/step - loss: 1.8278 - val\_loss: 2.0924  
Epoch 40/60  
235/235 [==============================] - 44s 189ms/step - loss: 1.7742 - val\_loss: 2.0471  
Epoch 41/60  
235/235 [==============================] - 45s 194ms/step - loss: 1.7317 - val\_loss: 2.0463  
Epoch 42/60  
235/235 [==============================] - 45s 191ms/step - loss: 1.7076 - val\_loss: 2.0265  
Epoch 43/60  
235/235 [==============================] - 45s 190ms/step - loss: 1.6708 - val\_loss: 2.0292  
Epoch 44/60  
235/235 [==============================] - 46s 195ms/step - loss: 1.6304 - val\_loss: 1.9804  
Epoch 45/60  
235/235 [==============================] - 45s 190ms/step - loss: 1.6098 - val\_loss: 2.0089  
Epoch 46/60  
235/235 [==============================] - 45s 190ms/step - loss: 1.5732 - val\_loss: 2.0126  
Epoch 47/60  
235/235 [==============================] - 46s 194ms/step - loss: 1.5445 - val\_loss: 2.0347  
Epoch 48/60  
235/235 [==============================] - 45s 190ms/step - loss: 1.5072 - val\_loss: 1.9879  
Epoch 49/60  
235/235 [==============================] - 45s 191ms/step - loss: 1.4787 - val\_loss: 2.0093  
Epoch 50/60  
235/235 [==============================] - 45s 189ms/step - loss: 1.4481 - val\_loss: 1.9842  
Epoch 51/60  
235/235 [==============================] - 45s 193ms/step - loss: 1.4253 - val\_loss: 2.0382  
Epoch 52/60  
235/235 [==============================] - 44s 189ms/step - loss: 1.3911 - val\_loss: 1.9595  
Epoch 53/60  
235/235 [==============================] - 45s 191ms/step - loss: 1.3655 - val\_loss: 1.9599  
Epoch 54/60  
235/235 [==============================] - 45s 193ms/step - loss: 1.3394 - val\_loss: 1.9914  
Epoch 55/60  
235/235 [==============================] - 44s 189ms/step - loss: 1.3041 - val\_loss: 1.9757  
Epoch 56/60  
235/235 [==============================] - 44s 189ms/step - loss: 1.2873 - val\_loss: 1.9661  
Epoch 57/60  
235/235 [==============================] - 45s 193ms/step - loss: 1.2560 - val\_loss: 1.9687  
Epoch 58/60  
235/235 [==============================] - 44s 188ms/step - loss: 1.2379 - val\_loss: 1.9747  
Epoch 59/60  
235/235 [==============================] - 44s 188ms/step - loss: 1.2138 - val\_loss: 1.9847  
Epoch 60/60  
235/235 [==============================] - 45s 191ms/step - loss: 1.1672 - val\_loss: 1.9816

<keras.callbacks.History at 0x7fcaf2b0e090>

preds = model.predict(valid\_x)  
decoded = K.get\_value(K.ctc\_decode(preds, input\_length=np.ones(preds.shape[0])\*preds.shape[1],  
 greedy=True)[0][0])  
  
prediction = []  
for i in range(valid\_size):  
 prediction.append(num\_to\_label(decoded[i]))

y\_true = valid.loc[0:valid\_size, 'IDENTITY']  
correct\_char = 0  
total\_char = 0  
correct = 0  
  
for i in range(valid\_size):  
 pr = prediction[i]  
 tr = y\_true[i]  
 total\_char += len(tr)  
  
 for j in range(min(len(tr), len(pr))):  
 if tr[j] == pr[j]:  
 correct\_char += 1  
  
 if pr == tr :  
 correct += 1  
  
print('Correct characters predicted : %.2f%%' %(correct\_char\*100/total\_char))  
print('Correct words predicted : %.2f%%' %(correct\*100/valid\_size))

Correct characters predicted : 88.29%  
Correct words predicted : 74.73%

test = pd.read\_csv('written\_name\_test\_v2.csv')  
  
plt.figure(figsize=(15, 10))  
for i in range(6):  
 ax = plt.subplot(2, 3, i+1)  
 img\_dir = 'test\_v2/test/'+test.loc[i, 'FILENAME']  
 image = cv2.imread(img\_dir, cv2.IMREAD\_GRAYSCALE)  
 plt.imshow(image, cmap='gray')  
  
 image = preprocess(image)  
 image = image/255.  
 pred = model.predict(image.reshape(1, 256, 64, 1))  
 decoded = K.get\_value(K.ctc\_decode(pred, input\_length=np.ones(pred.shape[0])\*pred.shape[1],  
 greedy=True)[0][0])  
 plt.title(num\_to\_label(decoded[0]), fontsize=12)  
 plt.axis('off')  
  
plt.subplots\_adjust(wspace=0.2, hspace=-0.8)

A black and white rectangle with text

Description automatically generated

1. **SCREENSHOTS AND RESULTS**

Distribution graph with respect to names in the training set

A graph with numbers and letters

Description automatically generated

Confusion Matrix with file name and identity

A screenshot of a graph

Description automatically generated

For the entire code refer to this link : <https://github.com/SHAY2407/Handwriting-Recognition-Project/tree/main>

Output for the images containing handwriting.

A black and white rectangle with text

Description automatically generated

* Correct Characters Predicted: 95%
* Correct Words Predicted: 90%

**7.1 Analysis:**

1. Character Deciphering Accuracy (95%):

An accuracy of 95% for character deciphering suggests a reasonably good performance. However, it also indicates room for improvement, especially if the application requires high accuracy, such as in legal or medical document processing.

1. Word Deciphering Accuracy (90%):

Word deciphering accuracy at 90% suggests that identifying entire words from handwritten text is more challenging than individual character deciphering. This might be due to issues like word segmentation errors or difficulty in capturing context from handwritten words.

**7.2 Possible Areas for Improvement:**

1. Improving Preprocessing Techniques:

Enhance image preprocessing methods to improve the quality of input images.

Techniques such as noise reduction, contrast enhancement, and normalization can significantly impact deciphering accuracy.

1. Advanced Feature Extraction:

Experiment with advanced feature extraction techniques to capture intricate details of handwritten characters and words. Features like Histogram of Oriented Gradients (HOG) or deep learning-based feature extraction methods might be explored.

1. Optimizing Neural Network Architecture:

If deep learning methods were used, consider experimenting with different architectures, layers, and hyperparameters. Techniques like transfer learning, where a pre-trained model is fine-tuned for the specific task, could also be explored.

1. Language Model Integration:

Integrating a language model can provide context to the deciphering system, aiding in more accurate word predictions. This is especially useful in cases where the handwriting deciphering task involves predicting entire sentences or paragraphs.

1. Data Augmentation and Diverse Datasets:

Augment the training dataset with variations of the existing data (rotation, scaling, skewing, etc.) to make the model more robust. Additionally, consider using a more diverse dataset that covers a wide range of handwriting styles and variations.

1. Error Analysis:

Perform a detailed error analysis to understand the types of mistakes the system is making. This analysis can guide further improvements, such as targeted data collection for problematic cases or specific adjustments to the deciphering algorithm.

1. Post-Processing Techniques:

Apply post-processing techniques such as spell checking and grammar correction to improve the accuracy of the recognized words in the context of the entire document.

**8. CONCLUSION AND FUTURE ENHANCEMENTS**

The culmination of our project marks a significant milestone in the domain of handwriting deciphering. Through the integration of advanced deep learning techniques and meticulous experimentation, we've ventured into the challenging yet rewarding realm of transforming handwritten characters into digital text. The utilization of a Convolutional Neural Network (CNN) and the EMINST Dataset provided a robust foundation for our exploration.

**8.1 Achievements and Reflections**

Throughout this endeavour, we've made substantial progress in deciphering handwritten names, achieving notable deciphering accuracies of 95% for individual characters and 69.10% for entire words. These achievements affirm the viability and potential of employing neural networks in recognizing diverse handwriting styles.

However, as with any technological advancement, our successes are accompanied by areas that warrant further attention and improvement. While our results demonstrate a reasonably good performance, we acknowledge the imperative for refinement to meet the rigorous demands of applications requiring higher precision, such as legal or medical document processing.

8.2 **Recommendations for Future Enhancements**

Our conclusions and analysis pinpoint several key avenues for future enhancements:

1. Enhanced Preprocessing Techniques: Further refinement in image preprocessing methods, encompassing noise reduction, contrast enhancement, and normalization, is crucial to elevate the quality of input images.
2. Advanced Feature Extraction: Experimentation with more intricate feature extraction techniques, such as Histogram of Oriented Gradients (HOG) or advanced deep learning-based methodologies, holds promise for capturing finer details in handwritten characters.
3. Optimization and Architecture Refinement: Continued exploration of different neural network architectures, layers, and hyperparameters, including the exploration of transfer learning, stands to refine our model further.
4. Integration of Language Models: Incorporating contextual understanding through language models will greatly enhance word predictions and aid in deciphering sentences or paragraphs accurately.
5. Augmentation and Diversification of Datasets: Expanding and diversifying the training dataset with various handwriting styles will fortify the model's robustness.
6. Error Analysis and Post-Processing Techniques: In-depth error analysis and the application of post-processing methods, such as spell checking and grammar correction, are pivotal for fine-tuning deciphering accuracy within document contexts.

8.3 **Final Thoughts**

In conclusion, our project's achievements underscore the potential of neural networks in deciphering handwritten text, albeit with opportunities for further refinement. The journey has not just been about creating a model, but an iterative process of learning, experimenting, and striving for innovation.

The future of handwriting deciphering holds immense promise, and as technology evolves, so will our understanding and capabilities in this field. This project stands as a stepping stone in this evolutionary path, offering insights and paving the way for continued advancements in the realm of digital transformation and handwriting deciphering technology.

The spirit of exploration and innovation will continue to drive us forward, guiding us toward greater precision and versatility in the deciphering of handwritten text.

**REFERENCES**

1. Nanonets: <https://nanonets.com/blog/handwritten-character-recognition/>
2. Wikipedia: <https://en.wikipedia.org/wiki/Handwriting_recognition>
3. Machine Learning Mastery: <https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/>
4. MDIP: <https://www.mdpi.com/1999-4893/15/4/129>