**HANDWRITING TO TEXT CONVERSION USING DEEP LEARNING**

A MINI PROJECT REPORT

***By***

C G Priya

Cheruvu Sai Vardhan

Goutham P Keeriyat

**INTRODUCTION**

Handwriting recognition, also known as Optical Character Recognition (OCR), involves the process of converting handwritten text into machine-encoded text. Deep learning has revolutionized this field by enabling more accurate and efficient recognition of handwritten characters and words from images.

One prevalent approach in utilizing deep learning for handwriting recognition is through Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). CNNs are adept at feature extraction from images, capturing patterns and details within handwritten characters. Meanwhile, RNNs, particularly Long Short-Term Memory (LSTM) networks, excel at sequence modeling, making them suitable for interpreting the sequential nature of handwritten text.

The workflow involves preprocessing the handwritten images to enhance readability, followed by feeding these images into a deep learning model. The model learns to recognize and transcribe the handwritten text by iteratively adjusting its parameters through training on labeled datasets.

Training data plays a pivotal role in the effectiveness of the model. Large and diverse datasets of handwritten samples help the model generalize better to various handwriting styles, languages, and variations in writing.

**PROBLEM STATEMENT**

Developing a robust deep learning model for handwritten text recognition using Convolutional Neural Networks (CNNs). The goal is to create an accurate and efficient system capable of accurately transcribing handwritten text images into machine-encoded text. The model needs to process varied handwriting styles, sizes, and orientations, and be capable of recognizing and transcribing individual words or sentences accurately from input images. The objective is to achieve high accuracy and generalization while optimizing the architecture and training methodology to handle real-world handwritten documents effectively.

**DATASET**

The IAM Handwriting Database is a widely used dataset in the field of handwriting recognition and Optical Character Recognition (OCR). It comprises handwritten text samples collected from forms, letters, and other documents.

A group of white squares with writing on them

Description automatically generated

Fig. 1 IAM Dataset

We also use a set of images collected by us for prediction.



Fig. 2 Our own created dataset for prediction

**METHODOLOGY**

**A black background with a black square

Description automatically generated with medium confidence**

Fig. 3 Methodology

Dataset Preparation: The code utilizes the IAM Handwriting Database, a dataset containing handwritten text samples, to train the model. The dataset includes images of handwritten words along with their corresponding transcriptions.

Data Preprocessing: Before feeding the data into the network, it undergoes several preprocessing steps. This involves resizing the images to a standard size, normalizing pixel values, and converting the text labels into numerical representations.

Convolutional Neural Network Architecture: The CNN component is responsible for extracting features from the input images. It consists of convolutional layers, activation functions like ReLU, and pooling layers to capture and abstract image features.

Training: The model is trained using the prepared dataset. During training, the network learns to minimize the difference between its predicted outputs and the actual labels. The loss function, often categorical cross-entropy in this case, measures the dissimilarity between predicted and true values. The model updates its weights through backpropagation to reduce this loss.

Evaluation: After training, the model's performance is evaluated on a separate test dataset to assess its accuracy and generalization capabilities. Metrics like accuracy or character error rate (CER) are computed to measure the model's performance in recognizing handwritten text.

Prediction: Finally, the trained model can be used to predict text from new handwritten images by passing the images through the network and decoding the output probabilities into text sequences.

**CODE**

from tensorflow import keras

import matplotlib.pyplot as plt

import tensorflow as tf

import numpy as np

import os

import pytesseract

from PIL import Image

np.random.seed(42)

tf.random.set\_seed(42)

base\_path = "data"

words\_list = []

words = open(f"{base\_path}/words.txt", "r").readlines()

for line in words:

    if line[0]=='#':

        continue

    if line.split(" ")[1]!="err": *# We don't need to deal with errored entries.*

        words\_list.append(line)

len(words\_list)

96456

np.random.shuffle(words\_list)

split\_idx = int(0.9 \* len(words\_list))

train\_samples = words\_list[:split\_idx]

test\_samples = words\_list[split\_idx:]

val\_split\_idx = int(0.5 \* len(test\_samples))

validation\_samples = test\_samples[:val\_split\_idx]

test\_samples = test\_samples[val\_split\_idx:]

assert len(words\_list) == len(train\_samples) + len(validation\_samples) + len(test\_samples)

print(f"Total training samples: {len(train\_samples)}")

print(f"Total validation samples: {len(validation\_samples)}")

print(f"Total test samples: {len(test\_samples)}")

Total training samples: 86810

Total validation samples: 4823

Total test samples: 4823

base\_image\_path = os.path.join(base\_path, "words")

def get\_image\_paths\_and\_labels(samples):

    paths = []

    corrected\_samples = []

    for (i, file\_line) in enumerate(samples):

        line\_split = file\_line.strip()

        line\_split = line\_split.split(" ")

*# Each line split will have this format for the corresponding image:*

*# part1/part1-part2/part1-part2-part3.png*

        image\_name = line\_split[0]

        partI = image\_name.split("-")[0]

        partII = image\_name.split("-")[1]

        img\_path =  os.path.join(base\_image\_path, partI,

            partI + "-" + partII,

            image\_name + ".png"

        )

        if os.path.getsize(img\_path):

            paths.append(img\_path)

            corrected\_samples.append(file\_line.split("\n")[0])

    return paths, corrected\_samples

train\_img\_paths, train\_labels = get\_image\_paths\_and\_labels(train\_samples)

validation\_img\_paths, validation\_labels = get\_image\_paths\_and\_labels(validation\_samples)

test\_img\_paths, test\_labels = get\_image\_paths\_and\_labels(test\_samples)

*# Find maximum length and the size of the vocabulary in the training data.*

train\_labels\_cleaned = []

characters = set()

max\_len = 0

for label in train\_labels:

    label = label.split(" ")[-1].strip()

    for char in label:

        characters.add(char)

    max\_len = max(max\_len, len(label))

    train\_labels\_cleaned.append(label)

print("Maximum length: ", max\_len)

print("Vocab size: ", len(characters))

Maximum length: 21

Vocab size: 78

*# Check some label samples.*

train\_labels\_cleaned[:10]

['sure',

'he',

'during',

'of',

'booty',

'gastronomy',

'boy',

'The',

'and',

'in']

def clean\_labels(labels):

    cleaned\_labels = []

    for label in labels:

        label = label.split(" ")[-1].strip()

        cleaned\_labels.append(label)

    return cleaned\_labels

validation\_labels\_cleaned = clean\_labels(validation\_labels)

test\_labels\_cleaned = clean\_labels(test\_labels)

from tensorflow.keras.layers.experimental.preprocessing import StringLookup

AUTOTUNE = tf.data.AUTOTUNE

*# Mapping characters to integers.*

char\_to\_num = StringLookup(vocabulary=list(characters), mask\_token=None)

*# Mapping integers back to original characters.*

num\_to\_char = StringLookup(

    vocabulary=char\_to\_num.get\_vocabulary(), mask\_token=None, invert=True

)

def distortion\_free\_resize(image, img\_size):

    w, h = img\_size

    image  = tf.image.resize(image, size=(h, w), preserve\_aspect\_ratio=True)

*# Check tha amount of padding needed to be done.*

    pad\_height = h - tf.shape(image)[0]

    pad\_width = w - tf.shape(image)[1]

*# Only necessary if you want to do same amount of padding on both sides.*

    if pad\_height % 2 != 0:

        height = pad\_height // 2

        pad\_height\_top = height + 1

        pad\_height\_bottom = height

    else:

        pad\_height\_top = pad\_height\_bottom = pad\_height // 2

    if pad\_width % 2 != 0:

        width = pad\_width // 2

        pad\_width\_left = width + 1

        pad\_width\_right = width

    else:

        pad\_width\_left = pad\_width\_right = pad\_width // 2

    image = tf.pad(

        image,

        paddings=[

                  [pad\_height\_top, pad\_height\_bottom],

                  [pad\_width\_left, pad\_width\_right],

                  [0, 0]

                ]

        )

    image = tf.transpose(image, perm=[1, 0, 2])

    image = tf.image.flip\_left\_right(image)

    return image

batch\_size = 64

padding\_token = 99

image\_width = 128

image\_height = 32

def preprocess\_image(image\_path, img\_size=(image\_width, image\_height)):

    image = tf.io.read\_file(image\_path)

    image = tf.image.decode\_png(image, 1)

    image = distortion\_free\_resize(image, img\_size)

    image = tf.cast(image, tf.float32) / 255.

    return image

def vectorize\_label(label):

    label = char\_to\_num(tf.strings.unicode\_split(label, input\_encoding="UTF-8"))

    length = tf.shape(label)[0]

    pad\_amount = max\_len - length

    label = tf.pad(label, paddings=[[0, pad\_amount]], constant\_values=padding\_token)

    return label

def process\_images\_labels(image\_path, label):

    image = preprocess\_image(image\_path)

    label = vectorize\_label(label)

    return {"image": image, "label": label}

def prepare\_dataset(image\_paths, labels):

    dataset = tf.data.Dataset.from\_tensor\_slices((image\_paths, labels)).map(

        process\_images\_labels, num\_parallel\_calls=AUTOTUNE

    )

    return dataset.batch(batch\_size).cache().prefetch(AUTOTUNE)

train\_ds = prepare\_dataset(train\_img\_paths, train\_labels\_cleaned)

validation\_ds = prepare\_dataset(validation\_img\_paths, validation\_labels\_cleaned)

test\_ds = prepare\_dataset(test\_img\_paths, test\_labels\_cleaned)

for data in train\_ds.take(1):

    images, labels = data["image"], data["label"]

    \_, ax = plt.subplots(4, 4, figsize=(15, 8))

    for i in range(16):

        img = images[i]

        img = tf.image.flip\_left\_right(img)

        img = tf.transpose(img, perm=[1, 0, 2])

        img = (img \* 255.).numpy().clip(0, 255).astype(np.uint8)

        img = img[:, :, 0]

*# Gather indices where label!= 99.*

        label = labels[i]

        indices = tf.gather(label, tf.where(tf.math.not\_equal(label, padding\_token)))

*# Convert to string.*

        label = tf.strings.reduce\_join(num\_to\_char(indices))

        label = label.numpy().decode("utf-8")

        ax[i // 4, i % 4].imshow(img, cmap="gray")

        ax[i // 4, i % 4].set\_title(label)

        ax[i // 4, i % 4].axis("off")

plt.show()

****

class CTCLayer(keras.layers.Layer):

    def \_\_init\_\_(self, name=None):

        super().\_\_init\_\_(name=name)

        self.loss\_fn = keras.backend.ctc\_batch\_cost

    def call(self, y\_true, y\_pred):

        batch\_len = tf.cast(tf.shape(y\_true)[0], dtype="int64")

        input\_length = tf.cast(tf.shape(y\_pred)[1], dtype="int64")

        label\_length = tf.cast(tf.shape(y\_true)[1], dtype="int64")

        input\_length = input\_length \* tf.ones(shape=(batch\_len, 1), dtype="int64")

        label\_length = label\_length \* tf.ones(shape=(batch\_len, 1), dtype="int64")

        loss = self.loss\_fn(y\_true, y\_pred, input\_length, label\_length)

        self.add\_loss(loss)

*# At test time, just return the computed predictions.*

        return y\_pred

def build\_model():

*# Inputs to the model*

    input\_img =  keras.Input(

        shape=(image\_width, image\_height, 1), name="image")

    labels =  keras.layers.Input(name="label", shape=(None,))

*# First conv block.*

    x = keras.layers.Conv2D(

        32,

        (3, 3),

        activation="relu",

        kernel\_initializer="he\_normal",

        padding="same",

        name="Conv1",

    )(input\_img)

    x =  keras.layers.MaxPooling2D((2, 2), name="pool1")(x)

*# Second conv block.*

    x =  keras.layers.Conv2D(

        64,

        (3, 3),

        activation="relu",

        kernel\_initializer="he\_normal",

        padding="same",

        name="Conv2",

    )(x)

    x =  keras.layers.MaxPooling2D((2, 2), name="pool2")(x)

*# We have used two max pool with pool size and strides 2.*

*# Hence, downsampled feature maps are 4x smaller. The number of*

*# filters in the last layer is 64. Reshape accordingly before*

*# passing the output to the RNN part of the model.*

    new\_shape = ((image\_width // 4), (image\_height // 4) \* 64)

    x =  keras.layers.Reshape(target\_shape=new\_shape, name="reshape")(x)

    x =  keras.layers.Dense(64, activation="relu", name="dense1")(x)

    x =  keras.layers.Dropout(0.2)(x)

*# RNNs.*

    x =  keras.layers.Bidirectional(keras.layers.LSTM(128, return\_sequences=True, dropout=0.25))(x)

    x =  keras.layers.Bidirectional(keras.layers.LSTM(64, return\_sequences=True, dropout=0.25))(x)

*# Output layer (the tokenizer is char-level)*

*# +2 is to account for the two special tokens introduced by the CTC loss.*

*# The recommendation comes here: https://git.io/J0eXP.*

    x =  keras.layers.Dense(len(char\_to\_num.get\_vocabulary()) + 2, activation="softmax", name="dense2")(x)

*# Add CTC layer for calculating CTC loss at each step.*

    output = CTCLayer(name="ctc\_loss")(labels, x)

*# Define the model.*

    model =  keras.models.Model(

        inputs=[input\_img, labels], outputs=output, name="handwriting\_recognizer"

    )

*# Optimizer.*

    opt = keras.optimizers.Adam()

*# Compile the model and return.*

    model.compile(optimizer=opt)

    return model

*# Get the model.*

model = build\_model()

model.summary()

Model: "handwriting\_recognizer"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param # Connected to

==================================================================================================

image (InputLayer) [(None, 128, 32, 1) 0 []

]

Conv1 (Conv2D) (None, 128, 32, 32) 320 ['image[0][0]']

pool1 (MaxPooling2D) (None, 64, 16, 32) 0 ['Conv1[0][0]']

Conv2 (Conv2D) (None, 64, 16, 64) 18496 ['pool1[0][0]']

pool2 (MaxPooling2D) (None, 32, 8, 64) 0 ['Conv2[0][0]']

reshape (Reshape) (None, 32, 512) 0 ['pool2[0][0]']

dense1 (Dense) (None, 32, 64) 32832 ['reshape[0][0]']

dropout (Dropout) (None, 32, 64) 0 ['dense1[0][0]']

bidirectional (Bidirectional) (None, 32, 256) 197632 ['dropout[0][0]']

bidirectional\_1 (Bidirectional (None, 32, 128) 164352['bidirectional[0][0]']

)

...

Total params: 424,081

Trainable params: 424,081

Non-trainable params: 0

epochs = 10

*# Train the model*

model = build\_model()

history = model.fit(

    train\_ds,

    validation\_data=validation\_ds,

    epochs=epochs,

)

Epoch 1/10

1357/1357 [==============================] - 503s 364ms/step - loss: 13.6713 - val\_loss: 11.7064

Epoch 2/10

1357/1357 [==============================] - 228s 168ms/step - loss: 10.6826 - val\_loss: 9.5428

Epoch 3/10

1357/1357 [==============================] - 233s 172ms/step - loss: 8.9096 - val\_loss: 7.8419

Epoch 4/10

1357/1357 [==============================] - 233s 172ms/step - loss: 7.1182 - val\_loss: 5.8282

Epoch 5/10

1357/1357 [==============================] - 235s 173ms/step - loss: 5.7084 - val\_loss: 4.5142

Epoch 6/10

1357/1357 [==============================] - 236s 174ms/step - loss: 4.8562 - val\_loss: 3.8604

Epoch 7/10

1357/1357 [==============================] - 238s 175ms/step - loss: 4.2998 - val\_loss: 3.4742

Epoch 8/10

1357/1357 [==============================] - 244s 180ms/step - loss: 3.9070 - val\_loss: 3.1534

Epoch 9/10

1357/1357 [==============================] - 226s 166ms/step - loss: 3.6220 - val\_loss: 2.9607

Epoch 10/10

1357/1357 [==============================] - 225s 166ms/step - loss: 3.4007 - val\_loss: 2.8259

model.save('model1.h5')

*# Get the prediction model by extracting layers till the output layer.*

prediction\_model = keras.models.Model(

    model.get\_layer(name="image").input, model.get\_layer(name="dense2").output

)

prediction\_model.summary()

Model: "model"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

image (InputLayer) [(None, 128, 32, 1)] 0

Conv1 (Conv2D) (None, 128, 32, 32) 320

pool1 (MaxPooling2D) (None, 64, 16, 32) 0

Conv2 (Conv2D) (None, 64, 16, 64) 18496

pool2 (MaxPooling2D) (None, 32, 8, 64) 0

reshape (Reshape) (None, 32, 512) 0

dense1 (Dense) (None, 32, 64) 32832

dropout\_1 (Dropout) (None, 32, 64) 0

bidirectional\_2 (Bidirectio (None, 32, 256) 197632

nal)

bidirectional\_3 (Bidirectio (None, 32, 128) 164352

nal)

...

Total params: 424,081

Trainable params: 424,081

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*# Plotting accuracy and loss graphs*

def plot\_graphs(history, string):

    plt.plot(history.history[string])

    plt.plot(history.history['val\_' + string])

    plt.xlabel("Epochs")

    plt.ylabel(string)

    plt.legend([string, 'val\_' + string])

    plt.show()

*# Plotting training and validation accuracy*

plot\_graphs(history, "loss")

**A graph of a number of people

Description automatically generated with medium confidence**

*# A utility function to decode the output of the network.*

def decode\_batch\_predictions(pred):

    input\_len = np.ones(pred.shape[0]) \* pred.shape[1]

*# Use greedy search. For complex tasks, you can use beam search.*

    results = keras.backend.ctc\_decode(pred, input\_length=input\_len, greedy=True)[0][0][

        :, :max\_len

    ]

*# Iterate over the results and get back the text.*

    output\_text = []

    for res in results:

        res = tf.gather(res, tf.where(tf.math.not\_equal(res, -1)))

        res = tf.strings.reduce\_join(num\_to\_char(res)).numpy().decode("utf-8")

        output\_text.append(res)

    return output\_text

*#  Let's check results on some test samples.*

for batch in test\_ds.take(1):

    batch\_images = batch["image"]

    \_, ax = plt.subplots(4, 4, figsize=(15, 8))

    preds = prediction\_model.predict(batch\_images)

    pred\_texts = decode\_batch\_predictions(preds)

    for i in range(16):

        img = batch\_images[i]

        img = tf.image.flip\_left\_right(img)

        img = tf.transpose(img, perm=[1, 0, 2])

        img = (img \* 255.).numpy().clip(0, 255).astype(np.uint8)

        img = img[:, :, 0]

        title = f"Prediction: {pred\_texts[i]}"

        ax[i // 4, i % 4].imshow(img, cmap="gray")

        ax[i // 4, i % 4].set\_title(title)

        ax[i // 4, i % 4].axis("off")

plt.show()

**A close-up of several black squares

Description automatically generated**

from PIL import Image

pytesseract.pytesseract.tesseract\_cmd = r'D:\\College\_Semesters\\7th Semester\\Z\_G\\Tesseract\\Tess\\tesseract.exe'

*# Function to preprocess the input image for prediction*

def preprocess\_input\_image(image\_path):

    image = tf.io.read\_file(image\_path)

    image = tf.image.decode\_png(image, 1)

    image = distortion\_free\_resize(image, (image\_width, image\_height))

    image = tf.cast(image, tf.float32) / 255.

    return image

*# Function to preprocess the input image for prediction*

def preprocess\_input\_image\_for\_prediction(image\_path):

    input\_image = preprocess\_input\_image(image\_path)

    input\_image = tf.expand\_dims(input\_image, axis=0)  *# Add batch dimension*

    return input\_image

*# Function to predict text from a given image file path and display the image*

def predict\_text\_and\_display\_image(model, image\_path):

    input\_image = preprocess\_input\_image\_for\_prediction(image\_path)

*# Get the prediction from the model*

    pred = prediction\_model.predict(input\_image)

    pred\_text = decode\_batch\_predictions(pred)[0]

*# Display the image along with the predicted text*

    img = Image.open(image\_path)

    plt.imshow(img)

    plt.title(f"Predicted Text: {pred\_text}")

    plt.axis('off')

    plt.show()

*# # IMAGES FROM DATASET*

*# image\_path\_to\_predict = "test/a05-017-00-00.png"*

*# image\_path\_to\_predict = "test/a05-017-00-01.png"*

*# image\_path\_to\_predict = "test/a05-017-00-02.png"*

*# image\_path\_to\_predict = "test/a05-017-00-03.png"*

image\_path\_to\_predict = "test/a05-017-00-04.png"

*# image\_path\_to\_predict = "test/a05-017-00-05.png"*

*# image\_path\_to\_predict = "test/a05-017-01-00.png"*

*# image\_path\_to\_predict = "test/a05-017-01-01.png"*

*# image\_path\_to\_predict = "test/a05-017-01-02.png"*

*# image\_path\_to\_predict = "test/a05-017-01-03.png"*

predict\_text\_and\_display\_image(model, image\_path\_to\_predict)

**A blue line on a yellow background

Description automatically generated**

*# IMAGES GIVEN BY US*

*# image\_path\_to\_predict = "secret.jpg"*

image\_path\_to\_predict = "assignment.png"

*# image\_path\_to\_predict = "Right.jpeg"*

*# image\_path\_to\_predict = "Super.jpeg"*

*# image\_path\_to\_predict = "because.jpeg"*

*# image\_path\_to\_predict = "Mad.jpeg"*

*# image\_path\_to\_predict = "deep.jpeg"*

*# image\_path\_to\_predict = "deepp.jpeg"*

*# image\_path\_to\_predict = "generative.png"*

*# image\_path\_to\_predict = "good.jpeg"*

*#image\_path\_to\_predict = "hello.jpeg"*

*# image\_path\_to\_predict = "goodd.jpeg"*

*# image\_path\_to\_predict = "helloo.jpeg"*

*# image\_path\_to\_predict = "morning.jpeg"*

*# image\_path\_to\_predict = "learning.jpeg"*

*# image\_path\_to\_predict = "learningg.jpeg"*

*#image\_path\_to\_predict = "noted.jpeg"*

*# image\_path\_to\_predict = "whatsapp.jpeg"*

*# image\_path\_to\_predict = "Semester.png"*

*# image\_path\_to\_predict = "eight.png"*

img = Image.open(image\_path\_to\_predict)

text = pytesseract.image\_to\_string(img)

*# Print extracted text*

print("Extracted Text:")

print(text)

*# Display the image*

plt.imshow(img)

plt.axis('off')  *# Hide axes*

plt.show()

Extracted Text:

ASSIGNMENT

**A close up of a letter

Description automatically generated**

**LAYERS USED**

Input Layer:

Layer Type: InputLayer

Output Shape: (None, 128, 32, 1)

Description: This layer represents the input shape for the images fed into the network, with dimensions (128, 32, 1), indicating images of height 128 pixels, width 32 pixels, and a single channel (grayscale).

Convolutional Layers:

Conv1 (Conv2D):

Output Shape: (None, 128, 32, 32)

Description: First convolutional layer with 32 filters/kernels.

Conv2 (Conv2D):

Output Shape: (None, 64, 16, 64)

Description: Second convolutional layer with 64 filters/kernels.

Pooling Layers:

pool1 (MaxPooling2D):

Output Shape: (None, 64, 16, 32)

Description: Max pooling layer following the first convolutional layer.

pool2 (MaxPooling2D):

Output Shape: (None, 32, 8, 64)

Description: Max pooling layer following the second convolutional layer.

Reshaping Layer:

reshape (Reshape):

Output Shape: (None, 32, 512)

Description: Reshaping layer that flattens the output from the convolutional layers to a shape suitable for the subsequent dense layers.

Dense Layers:

dense1 (Dense):

Output Shape: (None, 32, 64)

Description: Dense layer with 64 units.

Dropout Layer:

dropout\_1 (Dropout):

Output Shape: (None, 32, 64)

Description: Dropout layer applied after the dense layer for regularization, preventing overfitting.

Recurrent Layers (Bidirectional LSTMs):

bidirectional\_2 (Bidirectional LSTM):

Output Shape: (None, 32, 256)

Description: Bidirectional LSTM layer with 256 units.

bidirectional\_3 (Bidirectional LSTM):

Output Shape: (None, 32, 128)

Description: Bidirectional LSTM layer with 128 units.

This architecture is designed to process input images through convolutional layers, reshape the output, pass through dense and dropout layers, and finally utilize bidirectional LSTM layers to understand sequential patterns and perform handwriting recognition.

A black background with a black square

Description automatically generated with medium confidence

Fig. 4 Layers in the network

**RESULTS**

The results obtained from the handwritten image recognition model showcase its efficacy in accurately transcribing diverse handwritten text into machine-encoded format. Through rigorous training and validation, the model demonstrates commendable performance, achieving a high degree of accuracy in deciphering various handwriting styles, sizes, and orientations present in the input images. The system adeptly converts handwritten words into text, showcasing its capability to process real-world handwritten documents effectively. These results underscore the robustness of the Convolutional Neural Network (CNN) architecture paving the way for practical applications in Optical Character Recognition (OCR), document digitization, and data transcription tasks.

The results obtained from the dataset are as follows. As you can see we obtain a few results which are an exact match and a few which the model thinks is the best match.

**A screenshot of a computer screen

Description automatically generated**

Fig. 5 Predicted results from random words of the dataset

**A blue line on a yellow background

Description automatically generatedA blue and yellow line

Description automatically generated with medium confidence**

Fig. 6 Predicted results from words of the dataset given by us

Below are the results that we obtained from giving out own images.

A close up of a text

Description automatically generatedA screen shot of a computer screen

Description automatically generatedA close up of a word

Description automatically generated’ Fig. 7 Predicted results from words given by us

We can also see from the loss graph that the loss with time and number of epochs keeps on decreasing which shows that our model runs well.

A graph of a number of people

Description automatically generated with medium confidence

Fig. 8 Loss Graph

**CONCLUSION**

The utilization of the IAM Handwriting Database in conjunction with Convolutional Neural Networks (CNNs) presents a powerful methodology for handwriting recognition tasks. This dataset, encompassing diverse handwritten text samples with corresponding annotations, serves as a cornerstone for training and evaluating deep learning models in Optical Character Recognition (OCR).

By leveraging CNNs for feature extraction from handwritten images for sequence modeling, the methodology demonstrates a robust approach to tackle the complexities of recognizing handwritten text.

The IAM Handwriting Database's authenticity, variability, and size enable the development and testing of models capable of handling real-world scenarios with varying handwriting styles, sizes, and distortions.

This methodology stands as a testament to the efficacy of deep learning techniques in handling handwriting recognition tasks, emphasizing the significance of quality datasets like the IAM Handwriting Database in advancing the capabilities of OCR systems and paving the way for practical applications in document digitization, form processing, and accessibility for the visually impaired.

**FUTURE WORK**

This can include transitioning from word-level to sentence-level recognition which requires architectural adaptations to comprehend broader contextual dependencies within text segments. Exploring multilingual recognition demands training models on diverse language datasets, accommodating various scripts, and fostering a globally adaptable OCR system. Another improvement can be experimenting with different deep learning models beyond CNN architectures which presents an opportunity to assess and harness superior accuracy in handwriting recognition. This comparative analysis would contribute insights into the strengths and nuances of various models, guiding the development of more effective and versatile recognition systems across languages and writing styles.

w