Handwriting Recognition

Iris Zero

Zeyad Waheed 6797

Abstract

Handwriting recognition is a challenging problem in the field of deep learning, with applications spanning from digitizing historical documents to enhancing human-computer interaction. This abstract explores the application of the Convolutional Recurrent Neural Network (CRNN) model to address the handwriting recognition problem and achieve accurate text prediction.
The handwritten text recognition problem involves converting handwritten text images into machine-readable text. While traditional methods have made significant progress, deep learning techniques have revolutionized this field by enabling the development of more robust and accurate models.

Week One

1 Introduction

Handwriting recognition is a compelling problem within the realm of deep learning, where the objective is to transform handwritten text into digital form for applications ranging from historical document preservation to efficient data entry. One of the remarkable approaches in this domain involves the utilization of Convolutional Recurrent Neural Networks (CRNN). In this endeavor, we delve into the world of handwriting recognition, employing a CRNN model to convert handwritten text into accurate textual predictions. The dataset that fuels this endeavor is sourced from Kaggle, comprising an extensive collection of more than four hundred thousand handwritten names, generously contributed through charity projects.

While character recognition has shown remarkable prowess in converting machine-printed fonts into digital text, the challenge escalates when dealing with handwritten characters. The inherent variability in individual writing styles introduces complexities that traditional methods often struggle to overcome. This is where deep learning, specifically the CRNN model, steps in as a promising solution to tackle the nuances of handwritten text recognition.

The dataset we employ for this endeavor is both vast and diverse. It encompasses a grand total of 206,799 first names and 207,024 surnames, meticulously collected and curated for this project. To facilitate the development and evaluation of our CRNN model, the dataset is thoughtfully partitioned into three subsets: a training set comprising 331,059 examples, a testing set with 41,382 samples, and a validation set of equal size, ensuring robust model performance across different data splits.

The core of this venture revolves around the analysis of hundreds of thousands of images, each representing a handwritten name. Within this dataset, you'll find these transcribed images thoughtfully organized into distinct sets, each serving a crucial role in the training, testing, and validation of our CRNN model. This effort is not just an exploration of deep learning's capabilities but also a testament to the potential of harnessing advanced technology to bridge the gap between the analog and digital realms of handwritten text recognition.

2 Clean and Preprocess Data

2.1 Overview Data

Using "head" function is to inspect the initial rows of a DataFrame. It allows you to get a quick overview of what the data in the DataFrame looks like by displaying the first few rows.

Using "info" function is to provide a concise summary of the metadata and basic information about the DataFrame. It offers valuable insights into the dataset's structure, including details such as the number of entries, the data types of each column, and the presence of missing values.

2.2 Clean Data

After we sum the NULLS into traning, validation and test set and drop it from the data using "dropna".

Finally, we clean and preprocess the dataset by removing entries (images) that are marked as 'UNREADABLE' in the 'IDENTITY' column which cannot be readable by human eyes as well as. This filtering operation ensures that only readable and relevant data is retained for further analysis or model training.

2.3 Preprocess Data

We begin by importing the necessary modules for text preprocessing from TensorFlow Keras, including ”Tokenizer” and ”pad sequences”. The Tokenizer is used to convert text into numerical sequences, and ”pad sequences” is used to ensure that these sequences have uniform lengths.

We calculate the maximum length variable to determine the maximum length of text labels in the ’IDENTITY’ column of the) DataFrame. This is crucial for defining the maximum sequence length when tokenizing the text.

The Tokenizer is configured to work at the character level, meaning it treats each character as a separate token. It is fitted on the training data to build a vocabulary and generate a word index, which maps characters to numerical indices.

Next, the entire training dataset’s text labels are tokenized using the ”texts to sequences” method, which converts each text label into a sequence of numerical tokens.

To illustrate we selects one sample and converts it back to text using the ”texts to sequences” method.

Finally, the result, including the numerical sequence and the converted text, is printed to the console for verification and inspection.

The purpose of is to prepare text data for further processing. Tokenization converts text into a format that can be fed into neural networks, and padding ensures that all sequences have the same length. So, These preprocessing steps are essential for training models that work with text data effectively.

2.4 Preprocess Images

Here we use Preprocess Images function to play a pivotal role in the preprocessing pipeline for our handwritten recognition problem. Its primary objective is to transform raw image data of handwritten text and corresponding labels into a format suitable for training a machine learning model, specifically designed for handwritten text recognition.

We Can Break it as a FOUR operations :

Data Preparation: The code starts by extracting image file paths (images) and the associated textual labels (labels) from our dataset. These labels represent the handwritten text we aim to recognize.

Image Preprocessing: Handwritten text images undergo several critical transformations. They are loaded and converted to grayscale to simplify processing. Then, the images are resized to a standardized dimension (image height and image width). Additionally, to eliminate potential biases, images are rotated 90 degrees clockwise and normalized by scaling pixel values between 0 and 1.

Label Preprocessing: The textual labels, representing handwritten content, undergo a transformation essential for model compatibility. They are tokenized into numerical sequences using a character-level tokenizer (Tokenizer). To ensure uniform input size, padding is applied to these sequences, extending or truncating them to match a predefined maximum length.

Data Output: The code returns the preprocessed images and labels as a tuple, resulting in a dataset that is well-suited for training a handwritten text recognition model.

2.5 Data Preparation

We is instrumental in preparing the training/validation/test dataset for our handwritten text recognition model. Its primary goal is to process raw image data and corresponding labels, ultimately generating training-ready input and target data for our machine learning model.

it retrieves the image file name and corresponding label for each training/validation/test sample based on the current loop index. These samples are retrieved from the dataset, which contains handwritten text images and their associated labels.

then we calculates the length of each label, representing the number of characters in the handwritten text. which is crucial for later stages of model training and decoding.

The image and label data for each sample are preprocessed using the preprocess single image sample function. This function performs essential operations such as image resizing, rotation, normalization, label tokenization, and padding, ensuring that both images and labels are in suitable formats for training.

Processed images are expanded to include a channel dimension (usually 1 for grayscale images) and added to the x train/valid/test list, while processed labels are added to the y train/valid/test list. This formatting step is essential for creating the input and target data arrays required for model training.

Finally, the code converts the lists x train/valid/test, y train/valid/test, and train/valid/test label length into NumPy arrays, which are more efficient for numerical operations and compatible with machine learning frameworks.

3 Cost Function

In the context of our handwritten recognition problem, the cost function used is the Connectionist Temporal Classification (CTC) loss. The CTC loss is a widely employed technique for sequence alignment problems, particularly in tasks like speech recognition and, pertinent to our research, handwritten recognition. It allows us to align variable-length sequences (handwritten characters) with variable-length labels (ground truth) without the need for explicit alignment information.

Theoretical Operation of the Cost Function

Initialization: The CTC loss layer, denoted as CTCLayer, is a crucial component of our neural network architecture. In this layer, the ”ctc batch cost” function from Keras backend is utilized as our cost function. This function has been specifically designed to handle the intricacies of CTC loss computation.

Forward Pass (Call Method): During the forward pass of our neural network, the CTCLayer takes two inputs, y true and y pred.

y true: This input represents the ground truth labels for the handwritten sequences. Each label corresponds to a sequence of characters, words, or symbols in our handwritten data.

y pred: This input represents the predictions generated by our neural network. It is a sequence of probabilities or scores for each character class at each time step in the sequence.

Batch Size and Sequence Lengths: To prepare for the CTC loss calculation, the layer first determines some crucial parameters:

batch len: This variable is calculated to determine the size of the current batch. It represents the number of handwritten sequences in the batch.

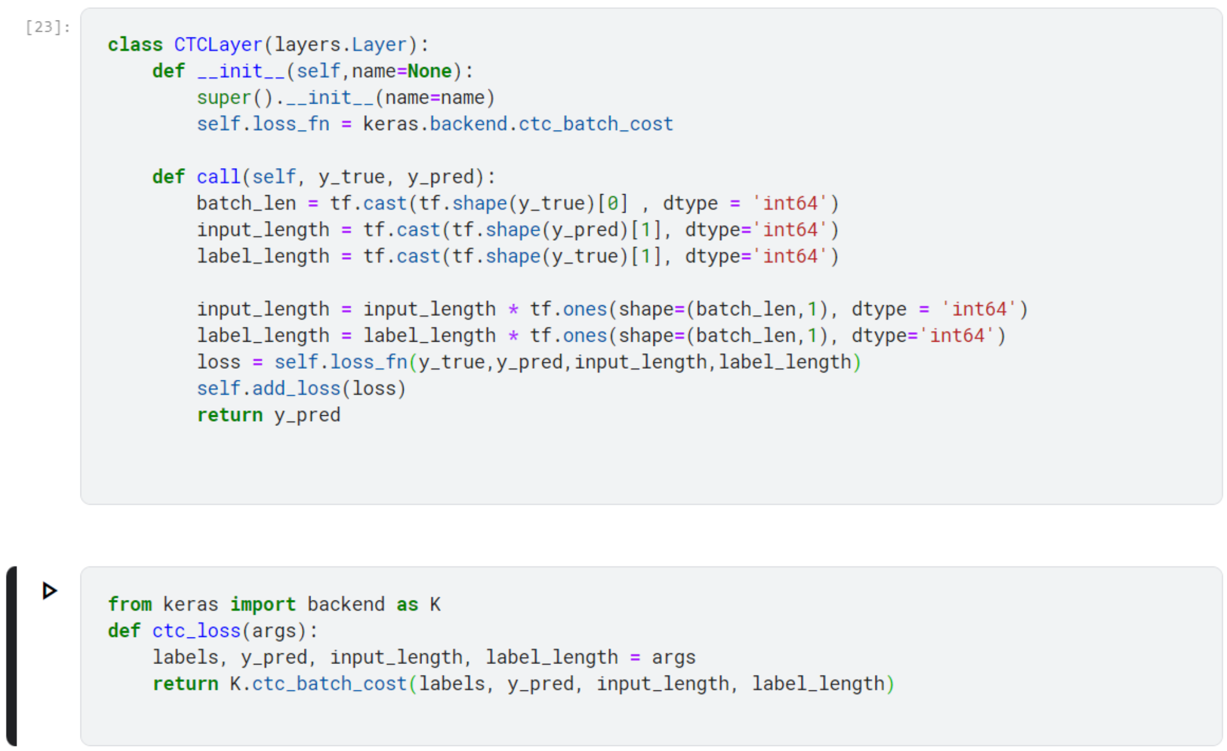
input length: This variable is computed to find the length of the predicted sequence (y pred) for each sample in the batch.

label length: This variable is calculated to find the length of the true label sequence (y true) for each sample in the batch.

Alignment Handling: In CTC, it is common for the input sequence to be longer than the true label sequence due to the inherent variability in handwriting. To account for this, input length and label length are expanded to match the batch size using tf.ones. This ensures that each sequence in the batch has the correct length information.

CTC Loss Computation: The core of the CTC loss calculation is performed using ”self.lossfn(y true, y pred, input length, label length)”. This function computes the CTC loss between the true labels and the predicted outputs. It takes into account the input and label lengths for each sequence in the batch. Integration into Model Loss: To train our neural network effectively, the computed CTC loss is added to the layer's internal losses using self.add loss(loss). This step is crucial as it ensures that the CTC loss contributes to the overall loss that our model is optimized to minimize.

Prediction: Finally, the call method returns y pred, which represents the model's predictions for the handwritten sequences. These predictions are vital for evaluating the model's performance and making predictions on unseen data.



```
[23]:
class CTCLayer(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)
        return y_pred
```

```
from keras import backend as K
def ctc_loss(args):
    labels, y_pred, input_length, label_length = args
    return K.ctc_batch_cost(labels, y_pred, input_length, label_length)
```

Figure 1: CTC Cost Funtion

Custom CTC Loss Function Definition: In our neural network architecture, we define a custom CTC loss function using the provided code snippet. The custom loss function, ctc loss, takes four arguments:

labels: This argument represents the ground truth labels for the handwritten sequences. Each label corresponds to a sequence of characters, words, or symbols in our handwritten data.

CTC Loss Calculation: The core functionality of our custom CTC loss function is encapsulated within the K.ctc batch cost function. This function is provided by Keras backend and has been designed to handle the complexities of CTC loss computation efficiently.

Alignment Handling: The ctc loss function takes care of aligning the predicted sequences (y pred) with their corresponding true labels (labels) while considering the input and label lengths for each sample in the batch. This alignment step is crucial, as handwritten sequences may exhibit variability in length.

Cost Evaluation: The ctc loss function computes the CTC loss, which quantifies the dissimilarity between the predicted sequences and the ground truth labels. The CTC loss takes into account the probabilities or scores assigned to each character class at each time step in the predicted sequence.

Integration into Model Training: The custom CTC loss function is integrated into our neural network model during training. It serves as the objective function that our model aims to minimize. By optimizing this loss, our model learns to align handwritten sequences accurately with their corresponding labels, leading to improved recognition performance.

4 Model

4.1 Model Architecture Description

Our handwritten recognition model is designed to tackle the challenging task of recognizing handwritten characters and words. This model leverages a combination of convolutional neural networks (CNNs) and recurrent neural networks (RNNs) to effectively process and recognize handwritten images.

4.2 Input Layers

Image Input (input image): This layer serves as the entry point for our model and expects grayscale images with dimensions (image height, image width, 1). These images are the handwritten characters we aim to recognize.

Label Input (labels): This input layer accepts ground truth labels for the handwritten sequences, which are sequences of characters or words. The shape of the labels is (maximum length,), accommodating variable-length label sequences.

Input Length (input length): This input layer provides information about the length of the input sequences (images) and has a shape of (1,). It is essential for the CTC loss calculation, which handles sequences of varying lengths.

Label Length (label length): Similar to the input length, this input layer specifies the length of the ground truth label sequences and also has a shape of (1,).

4.3 Convolutional Layers

Convolutional Layer 1 (Conv1): The first convolutional layer applies 32 filters of size (3, 3) with ReLU activation to capture essential features in the input images. It uses the He normal initialization and applies "same" padding to preserve spatial dimensions. Max-pooling with a (2, 2) pool size reduces spatial resolution.

Convolutional Layer 2 (Conv2): The second convolutional layer further refines features with 64 filters of size (3, 3). Like the previous layer, it uses ReLU activation, He normal initialization, and "same" padding. Another max-pooling operation follows.

4.4 Reshaping Layer

Reshape (reshape): This layer reshapes the output from the last convolutional layer into a shape of (imgage height//4, (imgage width//4) * 64). This reshaping prepares the data for the subsequent dense layers and the RNN layers. Dense Layers

Dense Layer 1 (dense1): This fully connected layer consists of 64 neurons with ReLU activation. It serves as an intermediate processing step before entering the RNN layers. Dropout with a rate of 0.2 is applied for regularization. Recurrent Layers (Bidirectional LSTMs)

Bidirectional LSTM Layer 1: The first bidirectional LSTM layer processes the data in both forward and backward directions. It consists of 256 LSTM units and returns sequences. Dropout with a rate of 0.25 helps prevent overfitting.

Bidirectional LSTM Layer 2: Similar to the first LSTM layer, this layer also has 256 units and returns sequences in both directions. Dropout is applied to improve generalization.

4.5 Output Layer

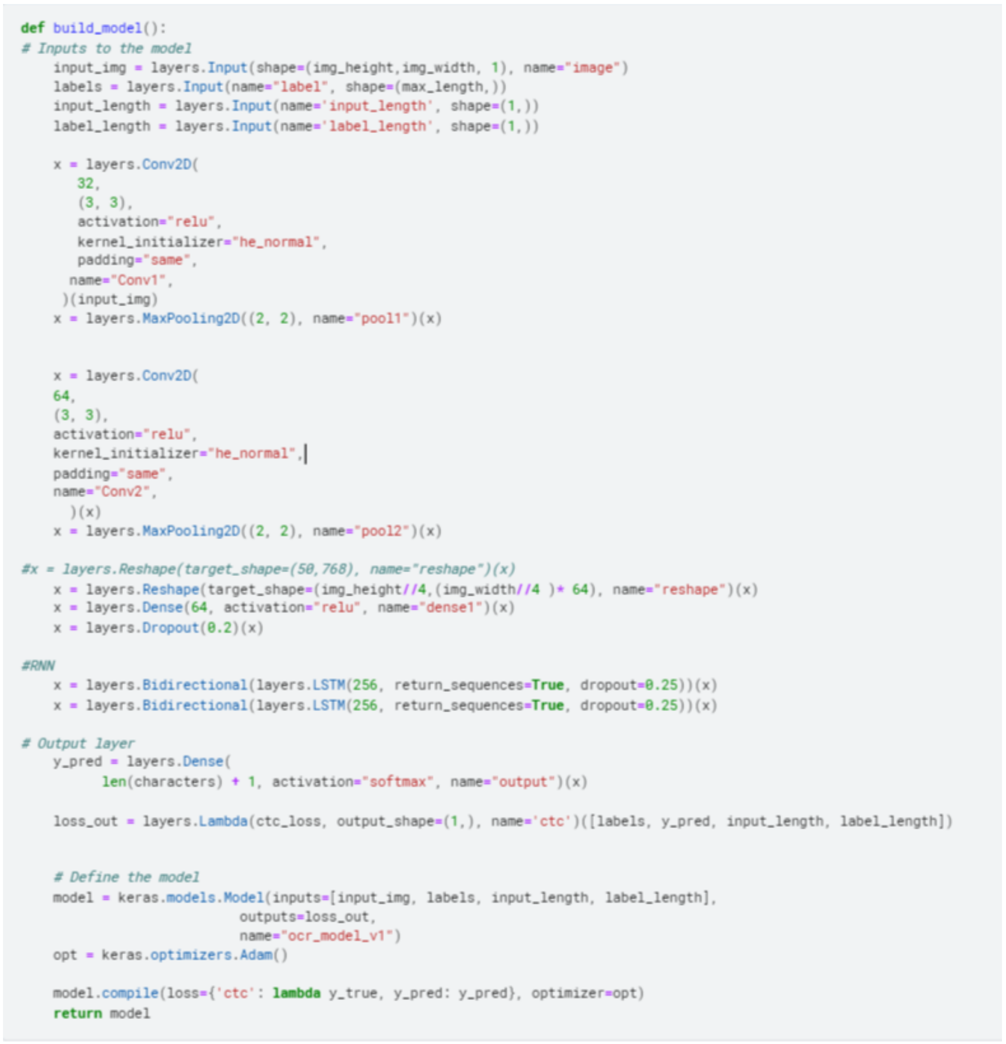
Output (output): The output layer is a dense layer with softmax activation, producing predictions for recognizing handwritten characters. It has a number of neurons equal to the number of unique characters in our dataset, plus one additional neuron for the CTC "blank" label. CTC Loss Layer

CTC Loss (ctc): This layer computes the Connectionist Temporal Classification (CTC) loss, which quantifies the dissimilarity between the predictions and the ground truth labels. It takes inputs from the label input, predicted output, input length, and label length. Model Compilation

4.6 Model Compilation

Finally, the model is compiled with an Adam optimizer and a custom loss function that includes the CTC loss. The model is named ”ocr model v1” and is ready for training to recognize handwritten characters and words effectively.

This architecture combines CNNs for feature extraction from images, dense layers for intermediate processing, and bidirectional LSTMs for sequence recognition, making it well-suited for our handwritten recognition task.

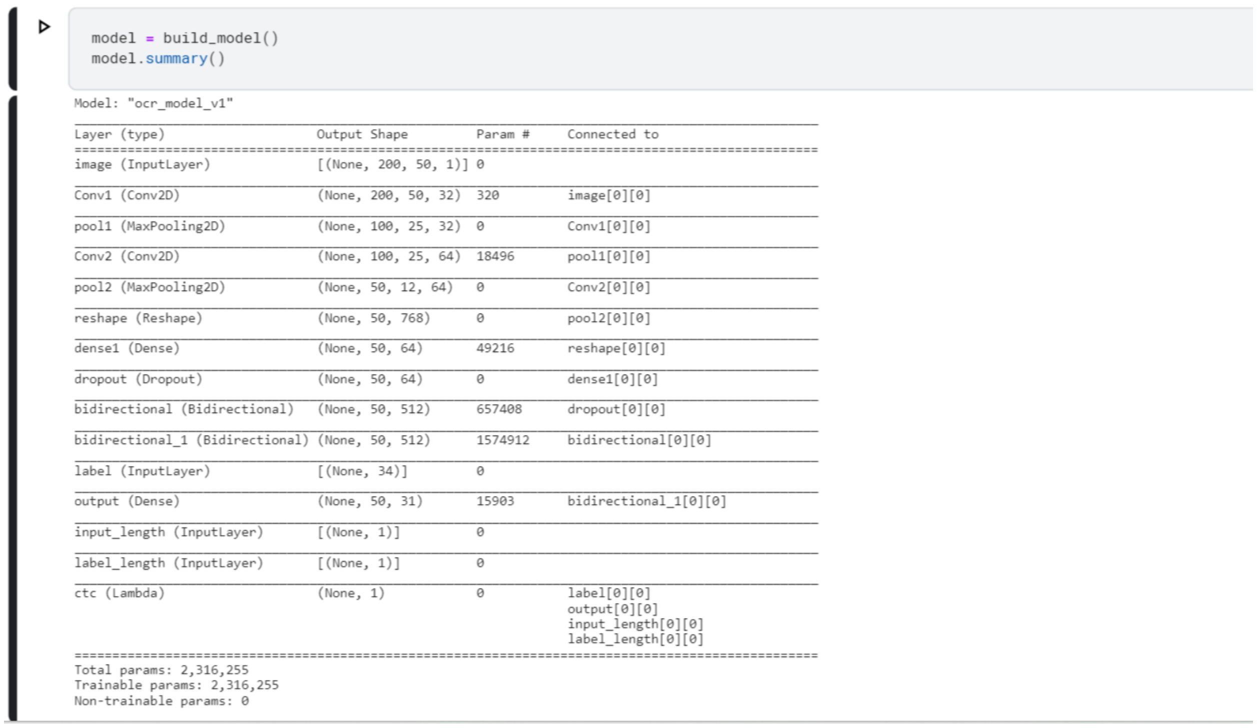


```
def build_model():
# Inputs to the model
    input_img = layers.Input(shape=(img_height,img_width, 1), name="image")
    labels = layers.Input(name="label", shape=(max_length,))
    input_length = layers.Input(name='input_length', shape=(1,))
    label_length = layers.Input(name='label_length', shape=(1,))

    x = layers.Conv2D(
       32,
       (3, 3),
       activation="relu",
       kernel_initializer="he_normal",
       padding="same",
      name="Conv1",
     )(input_img)
    x = layers.MaxPooling2D((2, 2), name="pool1")(x)


    x = layers.Conv2D(
    64,
    (3, 3),
    activation="relu",
    kernel_initializer="he_normal",
    padding="same",
    name="Conv2",
      )(x)
    x = layers.MaxPooling2D((2, 2), name="pool2")(x)

#x = layers.Reshape(target_shape=(50,768), name="reshape")(x)
    x = layers.Reshape(target_shape=(img_height//4,(img_width//4 )* 64), name="reshape")(x)
    x = layers.Dense(64, activation="relu", name="dense1")(x)
    x = layers.Dropout(0.2)(x)

#RNN
    x = layers.Bidirectional(layers.LSTM(256, return_sequences=True, dropout=0.25))(x)
    x = layers.Bidirectional(layers.LSTM(256, return_sequences=True, dropout=0.25))(x)

# Output layer
    y_pred = layers.Dense(
          len(characters) + 1, activation="softmax", name="output")(x)

    loss_out = layers.Lambda(ctc_loss, output_shape=(1,), name='ctc')([labels, y_pred, input_length, label_length])


    # Define the model
    model = keras.models.Model(inputs=[input_img, labels, input_length, label_length],
                               outputs=loss_out,
                               name="ocr_model_v1")
    opt = keras.optimizers.Adam()

    model.compile(loss={'ctc': lambda y_true, y_pred: y_pred}, optimizer=opt)
    return model
```

Figure 2: Baseline Model

4.7 Model Summary

Here’s Summary of the model.



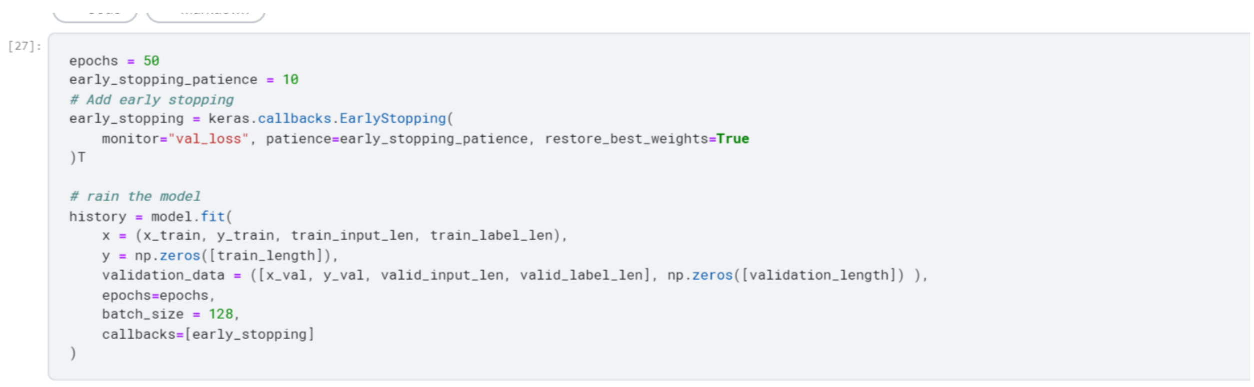
```python
model = build_model()
model.summary()
```

```
Model: "ocr_model_v1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
image (InputLayer)              [(None, 200, 50, 1)] 0
__________________________________________________________________________________________________
Conv1 (Conv2D)                  (None, 200, 50, 32)  320         image[0][0]
__________________________________________________________________________________________________
pool1 (MaxPooling2D)            (None, 100, 25, 32)  0           Conv1[0][0]
__________________________________________________________________________________________________
Conv2 (Conv2D)                  (None, 100, 25, 64)  18496       pool1[0][0]
__________________________________________________________________________________________________
pool2 (MaxPooling2D)            (None, 50, 12, 64)   0           Conv2[0][0]
__________________________________________________________________________________________________
reshape (Reshape)               (None, 50, 768)      0           pool2[0][0]
__________________________________________________________________________________________________
dense1 (Dense)                  (None, 50, 64)       49216       reshape[0][0]
__________________________________________________________________________________________________
dropout (Dropout)               (None, 50, 64)       0           dense1[0][0]
__________________________________________________________________________________________________
bidirectional (Bidirectional)   (None, 50, 512)      657408      dropout[0][0]
__________________________________________________________________________________________________
bidirectional_1 (Bidirectional) (None, 50, 512)      1574912     bidirectional[0][0]
__________________________________________________________________________________________________
label (InputLayer)              [(None, 34)]         0
__________________________________________________________________________________________________
output (Dense)                  (None, 50, 31)       15903       bidirectional_1[0][0]
__________________________________________________________________________________________________
input_length (InputLayer)       [(None, 1)]          0
__________________________________________________________________________________________________
label_length (InputLayer)       [(None, 1)]          0
__________________________________________________________________________________________________
ctc (Lambda)                    (None, 1)            0           label[0][0]
                                                                 output[0][0]
                                                                 input_length[0][0]
                                                                 label_length[0][0]
==================================================================================================
Total params: 2,316,255
Trainable params: 2,316,255
Non-trainable params: 0
```

Figure 3: Model Summary

4.8 Training Model

Training the model with early stopping with 50 Epochs.

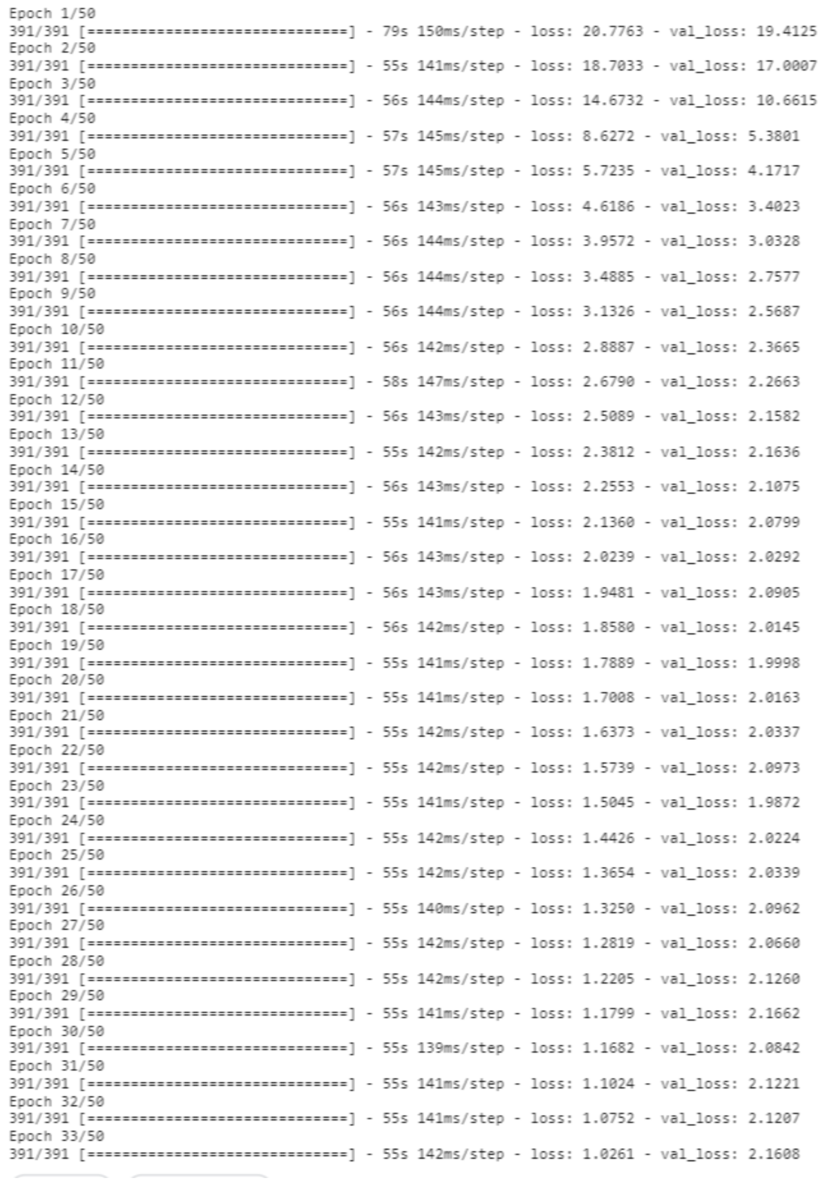


```python
[27]:
epochs = 50
early_stopping_patience = 10
# Add early stopping
early_stopping = keras.callbacks.EarlyStopping(
    monitor="val_loss", patience=early_stopping_patience, restore_best_weights=True
)T

# rain the model
history = model.fit(
    x = (x_train, y_train, train_input_len, train_label_len),
    y = np.zeros([train_length]),
    validation_data = ([x_val, y_val, valid_input_len, valid_label_len], np.zeros([validation_length]) ),
    epochs=epochs,
    batch_size = 128,
    callbacks=[early_stopping]
)
```

Figure 4: Training

4.9 Training Results

Training the model with early stopping so it ended with 33 Ebochs with loss 1.0261 and validation loss = 2.1608



```
Epoch 1/50
391/391 [==============================] - 79s 150ms/step - loss: 20.7763 - val_loss: 19.4125
Epoch 2/50
391/391 [==============================] - 55s 141ms/step - loss: 18.7033 - val_loss: 17.0007
Epoch 3/50
391/391 [==============================] - 56s 144ms/step - loss: 14.6732 - val_loss: 10.6615
Epoch 4/50
391/391 [==============================] - 57s 145ms/step - loss: 8.6272 - val_loss: 5.3801
Epoch 5/50
391/391 [==============================] - 57s 145ms/step - loss: 5.7235 - val_loss: 4.1717
Epoch 6/50
391/391 [==============================] - 56s 143ms/step - loss: 4.6186 - val_loss: 3.4023
Epoch 7/50
391/391 [==============================] - 56s 144ms/step - loss: 3.9572 - val_loss: 3.0328
Epoch 8/50
391/391 [==============================] - 56s 144ms/step - loss: 3.4885 - val_loss: 2.7577
Epoch 9/50
391/391 [==============================] - 56s 144ms/step - loss: 3.1326 - val_loss: 2.5687
Epoch 10/50
391/391 [==============================] - 56s 142ms/step - loss: 2.8887 - val_loss: 2.3665
Epoch 11/50
391/391 [==============================] - 58s 147ms/step - loss: 2.6790 - val_loss: 2.2663
Epoch 12/50
391/391 [==============================] - 56s 143ms/step - loss: 2.5089 - val_loss: 2.1582
Epoch 13/50
391/391 [==============================] - 55s 142ms/step - loss: 2.3812 - val_loss: 2.1636
Epoch 14/50
391/391 [==============================] - 56s 143ms/step - loss: 2.2553 - val_loss: 2.1075
Epoch 15/50
391/391 [==============================] - 55s 141ms/step - loss: 2.1360 - val_loss: 2.0799
Epoch 16/50
391/391 [==============================] - 56s 143ms/step - loss: 2.0239 - val_loss: 2.0292
Epoch 17/50
391/391 [==============================] - 56s 143ms/step - loss: 1.9481 - val_loss: 2.0905
Epoch 18/50
391/391 [==============================] - 56s 142ms/step - loss: 1.8580 - val_loss: 2.0145
Epoch 19/50
391/391 [==============================] - 55s 141ms/step - loss: 1.7889 - val_loss: 1.9998
Epoch 20/50
391/391 [==============================] - 55s 141ms/step - loss: 1.7008 - val_loss: 2.0163
Epoch 21/50
391/391 [==============================] - 55s 142ms/step - loss: 1.6373 - val_loss: 2.0337
Epoch 22/50
391/391 [==============================] - 55s 142ms/step - loss: 1.5739 - val_loss: 2.0973
Epoch 23/50
391/391 [==============================] - 55s 141ms/step - loss: 1.5045 - val_loss: 1.9872
Epoch 24/50
391/391 [==============================] - 55s 142ms/step - loss: 1.4426 - val_loss: 2.0224
Epoch 25/50
391/391 [==============================] - 55s 142ms/step - loss: 1.3654 - val_loss: 2.0339
Epoch 26/50
391/391 [==============================] - 55s 140ms/step - loss: 1.3250 - val_loss: 2.0962
Epoch 27/50
391/391 [==============================] - 55s 142ms/step - loss: 1.2819 - val_loss: 2.0660
Epoch 28/50
391/391 [==============================] - 55s 142ms/step - loss: 1.2205 - val_loss: 2.1260
Epoch 29/50
391/391 [==============================] - 55s 141ms/step - loss: 1.1799 - val_loss: 2.1662
Epoch 30/50
391/391 [==============================] - 55s 139ms/step - loss: 1.1682 - val_loss: 2.0842
Epoch 31/50
391/391 [==============================] - 55s 141ms/step - loss: 1.1024 - val_loss: 2.1221
Epoch 32/50
391/391 [==============================] - 55s 141ms/step - loss: 1.0752 - val_loss: 2.1207
Epoch 33/50
391/391 [==============================] - 55s 142ms/step - loss: 1.0261 - val_loss: 2.1608
```

Figure 5: Training

Week Two

5 Analysis of Model

We have plotted the loss between training and validation sets with respect to No. of Epochs.

```
[28]:
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Training Loss vs Validation Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

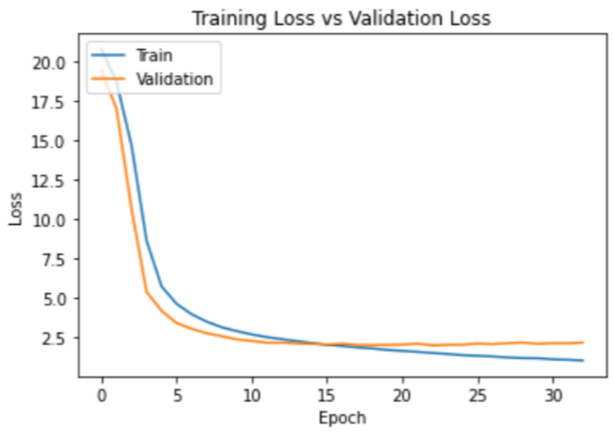


Figure 6: Plot

6 Prediction Model

6.1 Prediction Model Description

In our handwritten recognition system, the prediction model plays a vital role in transforming input images into meaningful text sequences. This model is a crucial component of our overall architecture, leveraging the knowledge learned during training to make predictions on unseen handwritten data.

6.2 Prediction Model Architecture

The prediction model is a streamlined version of our training model, focusing exclusively on the inference phase. It takes an input image and produces predictions for recognizing handwritten characters and words.

Here's an overview of its architecture :

Input Layer (Image Input) : The prediction model's input layer expects grayscale images with dimensions (image height, image width, 1), similar to the training model. These images represent the handwritten characters we want to recognize.

Output Layer (Output): The output layer of the prediction model is a dense layer with softmax activation, producing predictions for character recognition. It has the same number of neurons as the number of unique characters in our dataset, plus one additional neuron representing the CTC "blank" label. The output from this layer is a probability distribution over characters.

This prediction model is ready to take input images and provide predictions for recognizing handwritten characters and words. It encapsulates the knowledge acquired during training and is essential for real-world applications of our handwritten recognition system, where it will be deployed to process and interpret handwritten data.

The prediction model is a key component of our system, enabling us to perform character recognition on new handwritten inputs. Its architecture combines convolutional and recurrent layers, making it well-suited for understanding the spatial and temporal aspects of handwritten data.

6.3 Decoding of the Prediction Model

In our handwritten recognition problem, decoding predictions is a pivotal step in converting the model's output into meaningful text. The "decode predictions" function performs this critical task. It takes the model's predictions (pred), which are typically in a numerical format, and translates them into human-readable text.

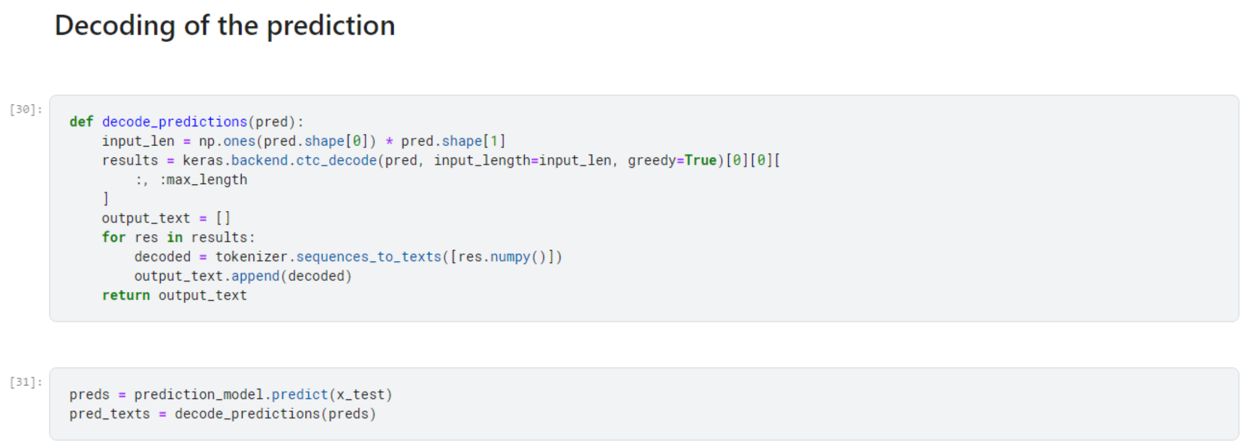
The decoding process is essential to transform numerical predictions from our model into actual text, making them interpretable and usable. The decode predictions function plays a central role in this process.

Input Length Determination: Initially, the function calculates the input length, which represents the number of time steps or columns in the predicted output. This step ensures that we process the model's predictions correctly.

CTC Decoding: We leverage the Connectionist Temporal Classification (CTC) decoding method, provided by Keras, to convert the model's predictions into text. The (keras.backend.ctc decode) function takes the predictions, input length, and employs the "greedy" decoding strategy to find the most likely text sequence.

Filtering and Post-processing: The decoded results are filtered to match the predefined maximum length, ensuring consistency. The filtered results are then processed to convert numerical sequences into text using the (tokenizer.sequences to texts) function, which maps numerical tokens back to characters.

Output Text Generation: The final output of the decode predictions function is a list of recognized text sequences. Each element in this list represents a recognized handwritten text sample.


Decoding of the prediction

```
[30]:
def decode_predictions(pred):
    input_len = np.ones(pred.shape[0]) * pred.shape[1]
    results = keras.backend.ctc_decode(pred, input_length=input_len, greedy=True)[0][0][
        :, :max_length
    ]
    output_text = []
    for res in results:
        decoded = tokenizer.sequences_to_texts([res.numpy()])
        output_text.append(decoded)
    return output_text
```

```
[31]:
preds = prediction_model.predict(x_test)
pred_texts = decode_predictions(preds)
```

Figure 7: Decoding of the Prediction

7 Evaluation

7.1 Accuracy on test set

Calculating the accuracy is a crucial step in evaluating the performance of our handwritten text recognition model. now we describe how accuracy is computed based on the model's predictions and the ground truth labels. Below is a description of this process:

Calculating Accuracy our Handwritten Text Recognition:

Accuracy serves as a fundamental metric to assess how effectively our model recognizes handwritten text. The following illustrates the process of computing accuracy.

1. Initialization and Variables: The process begins by initializing variables to keep track

of the number of correctly predicted words (correct), the number of correctly predicted characters within those words (correct char), and the total number of characters in the ground truth labels (total char). Additionally, the total number of test samples (test length) is determined based on the length of the predictions.

2. Iteration over Test Samples: The code iterates through each test sample, comparing the model's predictions (pr) to the ground truth labels (tr). It also calculates the total number of characters in the ground truth labels (total char) to later determine character-level accuracy.

3. Character-level Comparison: Within each sample, a character-level comparison is performed between the predicted word and the ground truth word. For each character position where the predicted character matches the ground truth character, correct char is incremented.

4. Word-level Comparison: The code also checks if the entire predicted word (pr) matches the ground truth word (tr). If they match, correct is incremented, indicating that the entire word was correctly recognized.

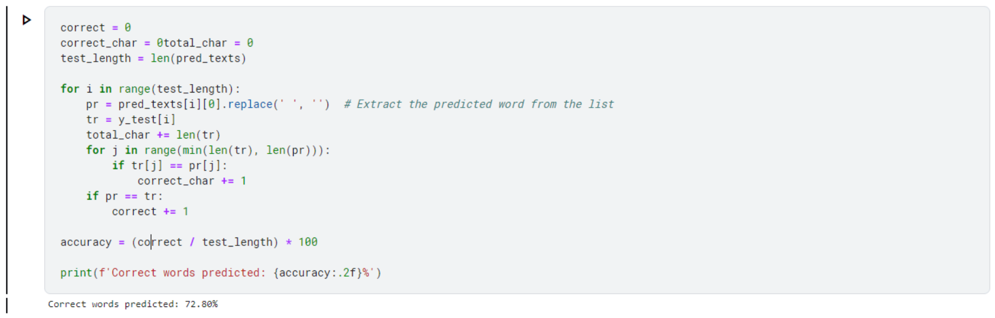
5. Accuracy Calculation: The overall accuracy is calculated as the ratio of correctly recognized words (correct) to the total number of test samples (test length). The result is multiplied by 100 to express accuracy as a percentage.

6. Displaying the Result: The calculated accuracy is then displayed to provide insight into the model's performance in recognizing handwritten text.

Significance of Accuracy Calculation:

The accuracy calculation is a fundamental part of our model evaluation process. It quantifies the model's ability to correctly recognize entire words and individual characters within those words. By assessing both word-level and character-level accuracy, we gain a comprehensive understanding of our model's performance, enabling us to fine-tune and optimize our system for improved handwritten text recognition.

And at the end the accuracy is 72.80 %.



```
correct = 0
correct_char = 0total_char = 0
test_length = len(pred_texts)

for i in range(test_length):
    pr = pred_texts[i][0].replace(' ', '')  # Extract the predicted word from the list
    tr = y_test[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

accuracy = (correct / test_length) * 100

print(f'Correct words predicted: {accuracy:.2f}%')
```

```
Correct words predicted: 72.80%
```

Figure 8: Accuracy in testset : 72.80 %

7.2 Evaluation Metrics

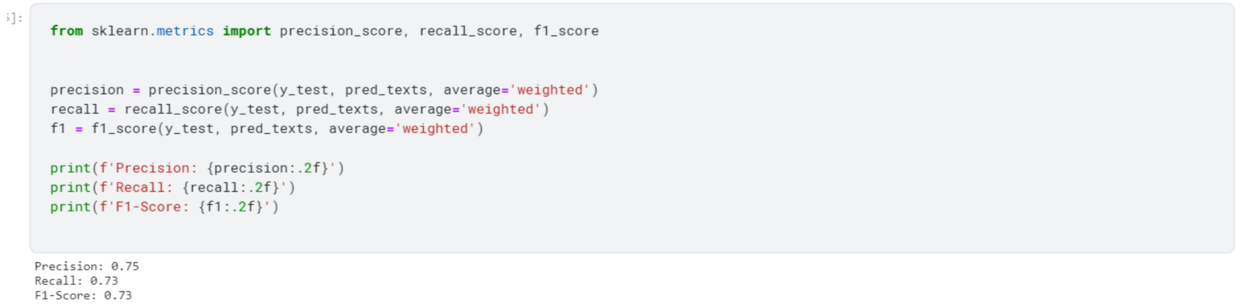
Metric Computation: Three key evaluation metrics are computed:

1 - Precision: Precision measures the accuracy of positive predictions, indicating the percentage of correctly recognized positive (handwritten text) instances among all predicted positive instances.
2 - Recall: also known as sensitivity or true positive rate, quantifies the ability of the model to correctly identify positive instances among all actual positive instances.
3 - F1-Score: The F1-score is the harmonic mean of precision and recall, providing a balanced assessment of a model's performance. It takes both false positives and false negatives into account.

The average='weighted' parameter is specified for each metric. This option calculates metrics for each class and then computes the weighted average based on class support. Weighted averages consider class imbalances in the dataset, making them particularly suitable for multiclass problems.

The code displays the calculated precision, recall, and F1-score values. These values offer insights into how well our model performs in recognizing handwritten text.

Precision: 0.75%
Recall: 0.73%
F1-Score: 0.73%



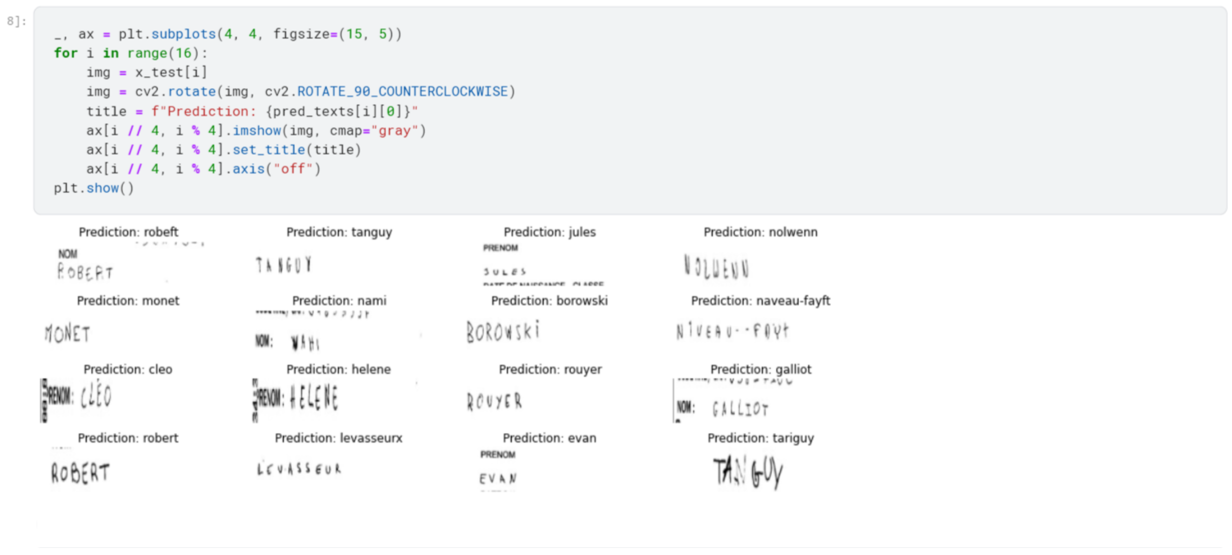
```
from sklearn.metrics import precision_score, recall_score, f1_score


precision = precision_score(y_test, pred_texts, average='weighted')
recall = recall_score(y_test, pred_texts, average='weighted')
f1 = f1_score(y_test, pred_texts, average='weighted')

print(f'Precision: {precision:.2f}')
print(f'Recall: {recall:.2f}')
print(f'F1-Score: {f1:.2f}')
```

```
Precision: 0.75
Recall: 0.73
F1-Score: 0.73
```

Figure 9: Evaluation Metrics

The calculated evaluation metrics play a pivotal role in assessing the quality of our handwritten text recognition model. They provide a comprehensive view of the model's performance, considering both precision (accuracy of positive predictions) and recall (ability to identify positive instances). The F1-score, as a balanced metric, combines these aspects, ensuring a well-rounded assessment.

7.3 Some Predicted Examples



```
8]:
_, ax = plt.subplots(4, 4, figsize=(15, 5))
for i in range(16):
    img = x_test[i]
    img = cv2.rotate(img, cv2.ROTATE_90_COUNTERCLOCKWISE)
    title = f"Prediction: {pred_texts[i][0]}"
    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()
```

Figure 10: Predicted Examples.