**INTERNSHIP: PROJECT REPORT**

-----------------------------------------------------------------------------------------------------------------------------------------

| | |
|---|---|
| Name Of The Student | Neeraj V B |
| Internship Project Title | Automate extraction of handwritten text from an image |
| Name of the Company | TCS iON |
| Name of the Industry Mentor | Debashis Roy |
| Name of the Institute | ICT Academy of Kerala |

| Start Date | End Date | Total Effort (hrs.) | Project Environment | Tools used |
|---|---|---|---|---|
| 27-06-2023 | 25-09-2023 | 125 | Google Colab with T4 GPU runtime, Python 3.10.12 and Google drive | Dataset – IAM words.tgz & ascii.tgz, TensorFlow 2.12.0 & Keras 2.12.0, OpenCV 4.8.0, NumPy 1.23.5, Levenshtein 0.21.1 |

**Project Synopsis:**
**Title of the Project :** Automate extraction of handwritten text from an image
**Introduction :** This is a TCS iON remote internship - RIO 125-Industry Project. This project aims to develop machine learning algorithms in order to enable entity and knowledge extraction from documents with handwritten annotations, with an aim to identify handwritten words on an image.
When it comes to addressing the challenge of sequence recognition, recurrent neural networks (RNNs) prove to be highly suitable. On the other hand, convolutional neural networks (CNNs) excel in image-related tasks. However, for Optical Character Recognition (OCR) problems like the one at hand, a combination of both CNN and RNN becomes imperative. By leveraging the strengths of both networks, we can effectively tackle the intricacies of handwritten word identification within images.
**Aim**: Utilize machine learning techniques to create a robust solution that can detect and transcribe handwritten content from images, enhancing text digitization and enabling efficient data analysis.

**Solution Approach:**
An optical character recognition problem is basically a type of image-based sequence recognition problem. And for sequence recognition problem, most suited neural networks are recurrent neural networks (RNN) while for an image based problem most suited are convolution neural networks (CNN). To cop up with the OCR problems we need to combine both of these CNN and RNN. So, I used Convolutional Recurrent Neural Network (CRNN) to tackle the both the problems. To implement my project I used words.tgz from IAM dataset, Google Colab T4 GPU, TensorFlow 2.12.0, Keras 2.12.0, OpenCV 4.8.0, NumPy, Jupyter Notebook and python 3.10.12.
We can break the implementation of the project cycle into following steps:

1. Selecting Dataset
2. Pre-processing Data
3. Splitting the Data
4. Building Model Architecture
6. Model Training
7. Model Evaluation
8. Exporting the Model
9. Deploying the Model

1) Collecting Dataset – The characteristics of the data outlined in the project guidelines align with those present in the IAM dataset. This dataset encompasses cursive handwriting, images of subpar quality

extracted from scanned documents, and visuals that might lack perfect alignment. Consequently, the decision was made to procure the original IAM dataset for utilization in this project. I downloaded the words.tgz from the IAM handwritten dataset. This is a large image dataset with a total of 115320 images with a size of 1.09 GB. I also downloaded the ascii.tgz part of the dataset which contains words.txt containing information about the written words. I uploaded both words.tgz and words.txt to my google drive. Out of the total images, 96456 images are labelled as 'ok' by the ascii representation, others have errors. I random sampled 90000 images from this to serve as the training + validation dataset and the remaining 6456 images to serve as testing dataset. I uploaded them both to my google drive for easier access

2) Pre-processing Data –
To preprocess the input images:
The image is read using OpenCV as a gray-scale and passed into a function that does the following:

1. Resize the image to a shape (32, 128), i.e. 128 pixels wide and 32 pixels high
2. Expand image dimension as (32, 128, 1) to make it compatible with the input shape of architecture.
3. Normalize the image pixel values by dividing it with 255.

To preprocess the output labels:

1. Created a list of all unique characters from every word in the dataset.
2. Encode each corresponding words(labels) to numeric sequences
3. Compute the maximum length of a word and post-pad with zeroes to make it in a consistent shape

This is done to make it compatible with the output shape of our RNN architecture. Then convert it into a NumPy array. During preprocessing, we also create two important lists: one for label lengths and the other for input lengths for our RNN. These lists are significant for our Connectionist Temporal Classification (CTC) loss. The label length corresponds to the length of each output text label, while the input length remains the same for every input to the LSTM layer, which is set to 31 in our architecture.

3) Splitting the Data –
The main dataset is split into training and validation sets with 85% data for training and 15% for validation. They are them converted into numpy arrays in order to be able to be fed into a neural network.

4) Building Model Architecture –
Architecture: Convolutional Recurrent Neural Network(CRNN)
This network architecture is inspired by this paper. Let's see the steps that we used to create the architecture:
Our images are of height 32 pixels and of width 128 pixels, thus making the architecture's input shape as (32, 128, 1). In this design, we incorporated seven convolutional layers, with six of them employing a kernel size of (3,3), while the last one utilizes a size of (2,2). This progressive arrangement also involves an increase in the number of filters, starting from 64 and gradually reaching 512 as we move through the layers.
To enhance the model's feature extraction capabilities, we introduced two max-pooling layers with dimensions (2,2). Additionally, we included two more max-pooling layers sized (2,1) to capture broader features, a valuable asset for predicting longer texts.
To optimize the training process, we integrated batch normalization layers after the fifth and sixth convolutional layers. This speeds up training and acts as a regularizer.

To accommodate the architecture's structure, we implemented a lambda function to compress the output from the convolutional layers, ensuring compatibility with the subsequent LSTM layer.
Subsequently, we integrated two Bidirectional LSTM layers, each comprised of 128 units. This configuration of Recurrent Neural Network (RNN) components produces an output of dimensions (batch_size, 31, 79). In this context, the number 79 represents the comprehensive count of output classes, including the blank character.

5) Defining CTC loss function – With our model architecture ready, the next step is to pick the right loss function. For our text recognition task, we'll use the CTC (Connectionist Temporal Classification) loss function. This choice is helpful in text recognition situations. It removes the need to label each time step individually and handles cases where a single character spans multiple time steps. By using CTC, we avoid extra steps for handling such situations. The CTC loss function needs four main inputs: the model's predicted results, the actual labels, the input sequence length for the LSTM layer, and the label length. To make this work, we create a custom loss function and include it in the model. Our model is designed to take these inputs and produce the associated loss output, all while ensuring compatibility.

6) Model Training –
For model training, we will employ the SGD (Stochastic Gradient Descent) optimizer. Additionally, leveraging Keras callbacks functionality, we can save the weights of the top-performing model based on validation accuracy. Notably, in the "model.compile()" step, it's evident that only the predicted outputs (y_pred) were included, while the actual ground truth labels (y_true) were omitted. This choice stems from the fact that the labels were already integrated as input during an earlier stage of model development.
With these preparations in place, you can proceed to train your model using a training dataset comprising 76,478 images and a validation dataset of 13,497 images. This training phase is vital to fine-tune the model's performance and enable it to generalize well to new data.
After loading the best weights that were previously saved, the next step involves preserving the entire model configuration and its corresponding weights. This is accomplished by saving the complete model as an "h5" file. This process ensures that the model's architecture, along with its trained parameters, are stored in a single file, making it ready for deployment or further analysis.

7) Model Evaluation –
With the model successfully trained on a dataset of 76,478 images, the focus now shifts to testing its performance. When it comes to testing, our training model isn't applicable due to its reliance on input labels - something unavailable during testing. Thus, to evaluate the model, we'll employ the CRNN model we constructed earlier, which exclusively requires the input of test images.
Given that our model generates probability distributions for each class at every time step, a transcription function is necessary to convert these probabilities into meaningful text. To achieve this, we utilized the CTC decoder, a method adept at transforming these predictions into coherent output text.
For assessing accuracy, we utilize the Jaro Distance and Ratio method. This approach quantifies the similarity between strings, providing valuable insights into the model's performance. To implement this method, we utilize the Levenshtein package, which facilitates the calculation of the Jaro distance and ratio. This measurement process offers a robust evaluation of the model's capability in addressing the task at hand.
Jaro Distance & Jaro Ratio - The Jaro Distance and Ratio are string similarity metrics that quantify the degree of similarity between two strings. These metrics consider both the characters present and their order in the strings. The Jaro Distance measures the dissimilarity between strings, with higher values

indicating greater similarity. The Jaro Ratio, on the other hand, is obtained by dividing the Jaro Distance by a scaling factor, resulting in a value between 0 and 1, where 1 signifies an exact match.

8) Exporting the Model –
The complete model, in the form of an "h5" file, is stored on Google Drive for the purpose of deployment using Streamlit. To ensure accurate label decoding and word retrieval, the character encoding is also preserved. This encoding information is essential for translating the model's predictions back into meaningful words.

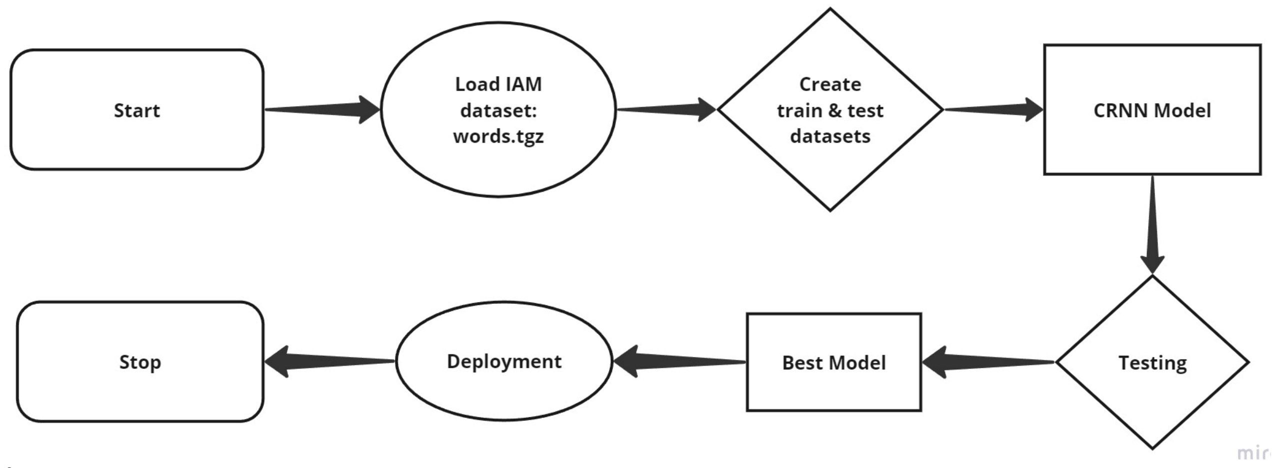
9) Deploying the Model –
The model is deployed on Streamlit cloud with the exported h5 model and character encoding json with the necessary dependencies

**Assumptions:**

- The handwritten text across the image are in English.
- The images are not tilted too much
- The image aspect ratio is below or slightly above 4:1 (ideally 4:1)
- Only image is provided for text recognition
- All dependencies are installed properly

**Project Diagrams:**

1) **Block Diagram**

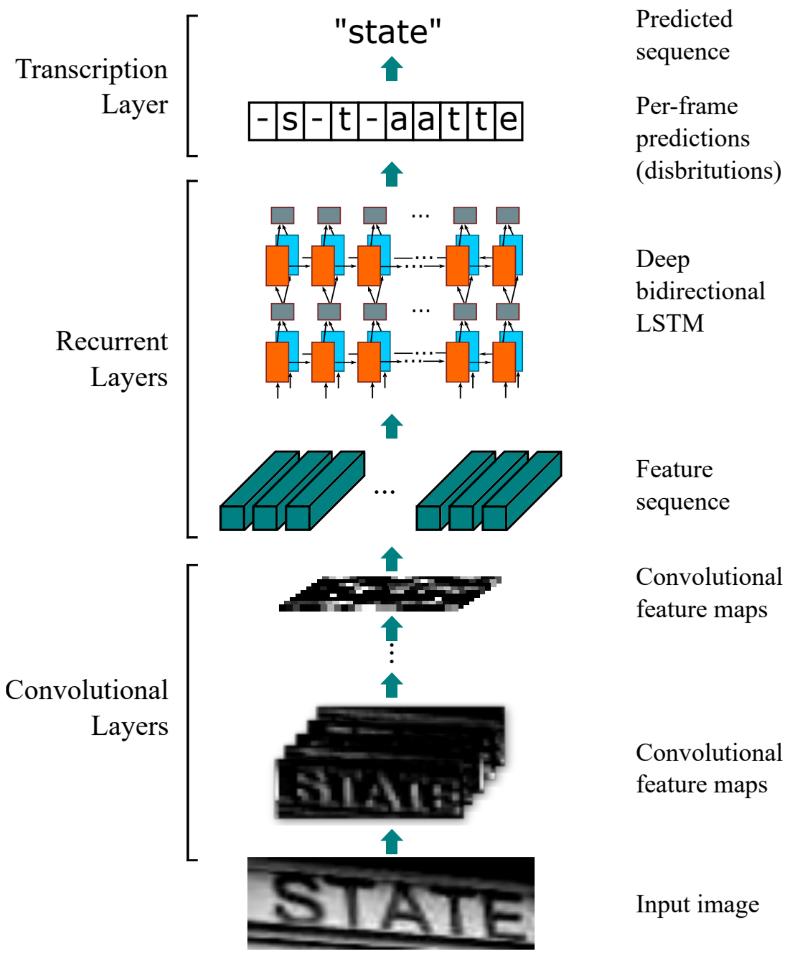


2) **CRNN Model Diagram**
The architecture of this model consists of three parts:
1) Convolutional Layers:
These layers are responsible for extracting relevant features from the input image. They analyze local patterns and shapes within the image, capturing important visual cues like edges, corners, and textures.
2) Recurrent Layers:
These layers incorporate temporal information by considering the sequence of features extracted by the convolutional layers. Recurrent layers help capture sequential dependencies in the data, making them well-suited for tasks like speech recognition or text analysis, where the order of input matters. In the context of CRNN, recurrent layers predict a distribution of labels for each frame of the input sequence.

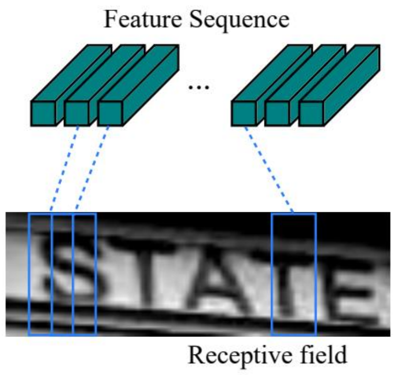
3) Transcription Layer:

After the recurrent layers generate label distributions for each frame, the transcription layer converts these distributions into the final label sequence. This is where the predicted labels are assembled in a coherent manner to form the recognized text. The transcription layer effectively translates the sequence of per-frame predictions into the actual words or characters that constitute the text.

In summary, a CRNN combines convolutional layers for feature extraction, recurrent layers for sequence modeling, and a transcription layer to produce the final output sequence, making it a powerful architecture for tasks like handwriting recognition or text transcription from images.



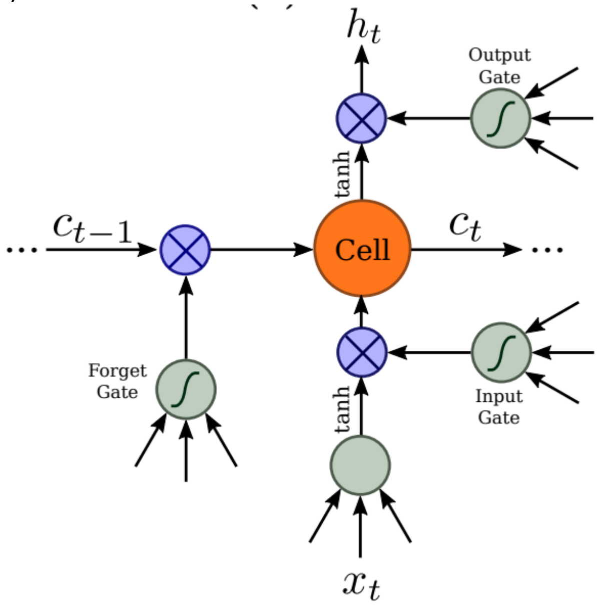
**3) The Receptive Field**



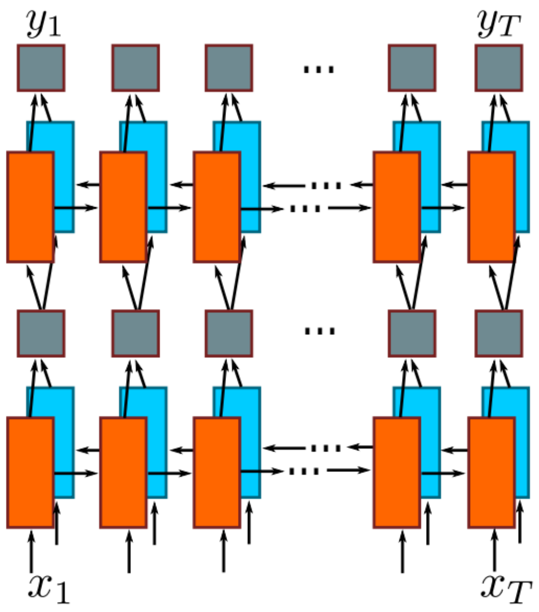
Each vector in the extracted feature sequence is associated with a receptive field on the input image, and can be considered as the feature vector of that field

**4) LSTM**

a) Basic LSTM



b) Bidirectional LSTM

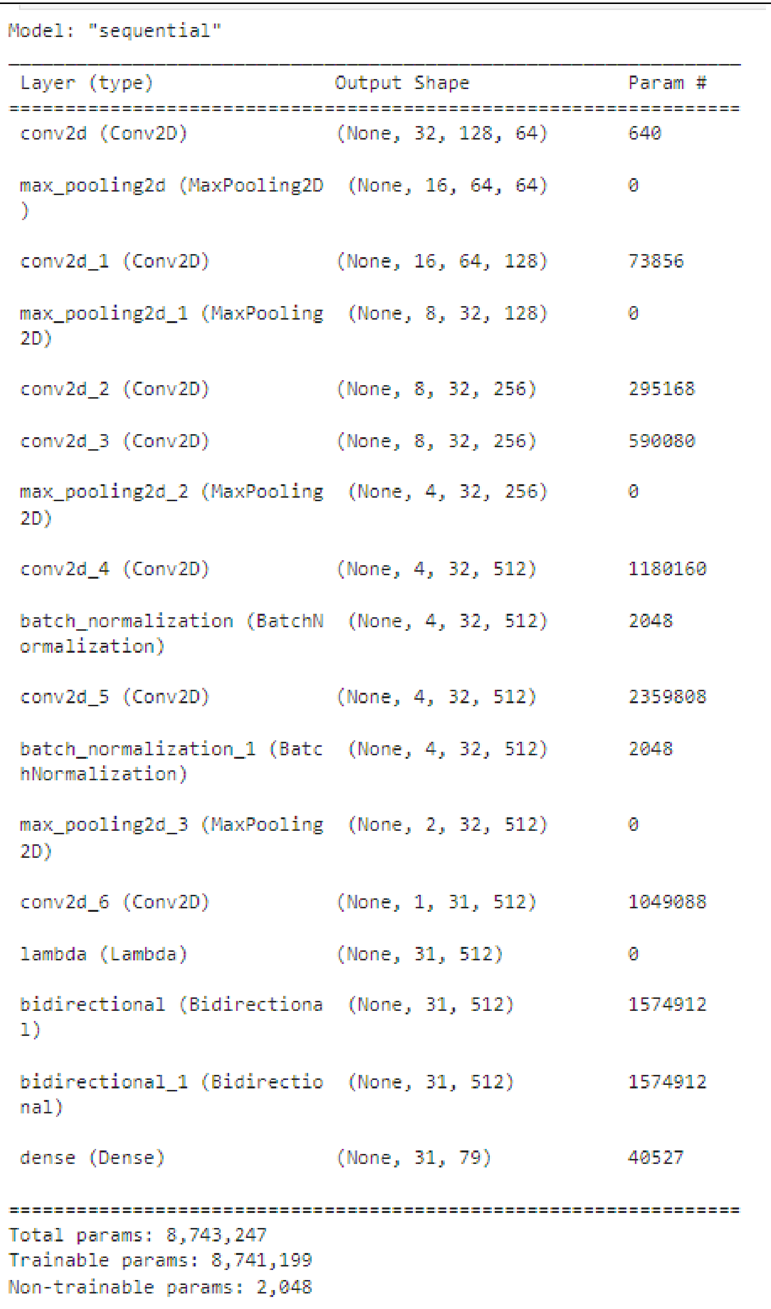


**Algorithms:**

**Model = CNN + RNN + CTC loss**

Our model consists of three parts:

1. The convolutional neural network to extract features from the image
2. Recurrent neural network to predict sequential output per time-step
3. CTC loss function which is transcription layer used to predict output for each time step



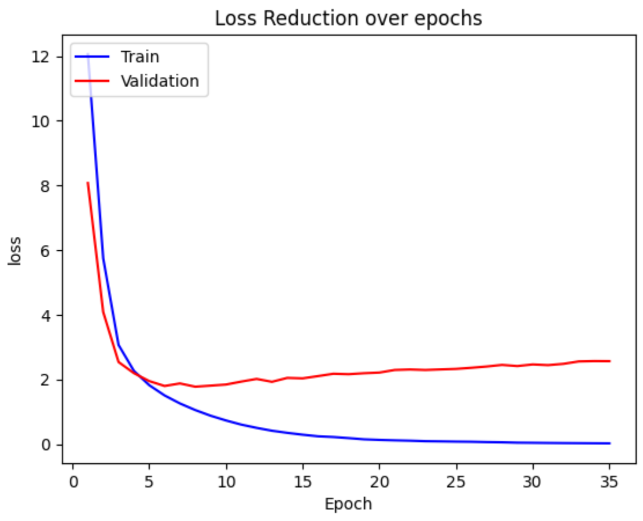
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 32, 128, 64)       640       
                                                                 
 max_pooling2d (MaxPooling2D  (None, 16, 64, 64)       0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (None, 16, 64, 128)       73856     
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 8, 32, 128)       0         
 2D)                                                             
                                                                 
 conv2d_2 (Conv2D)           (None, 8, 32, 256)        295168    
                                                                 
 conv2d_3 (Conv2D)           (None, 8, 32, 256)        590080    
                                                                 
 max_pooling2d_2 (MaxPooling  (None, 4, 32, 256)       0         
 2D)                                                             
                                                                 
 conv2d_4 (Conv2D)           (None, 4, 32, 512)        1180160   
                                                                 
 batch_normalization (BatchN  (None, 4, 32, 512)       2048      
 ormalization)                                                   
                                                                 
 conv2d_5 (Conv2D)           (None, 4, 32, 512)        2359808   
                                                                 
 batch_normalization_1 (Batc  (None, 4, 32, 512)       2048      
 hNormalization)                                                 
                                                                 
 max_pooling2d_3 (MaxPooling  (None, 2, 32, 512)       0         
 2D)                                                             
                                                                 
 conv2d_6 (Conv2D)           (None, 1, 31, 512)        1049088   
                                                                 
 lambda (Lambda)             (None, 31, 512)           0         
                                                                 
 bidirectional (Bidirectiona  (None, 31, 512)          1574912   
 l)                                                              
                                                                 
 bidirectional_1 (Bidirectio  (None, 31, 512)          1574912   
 nal)                                                            
                                                                 
 dense (Dense)               (None, 31, 79)            40527     
                                                                 
=================================================================
Total params: 8,743,247
Trainable params: 8,741,199
Non-trainable params: 2,048
```

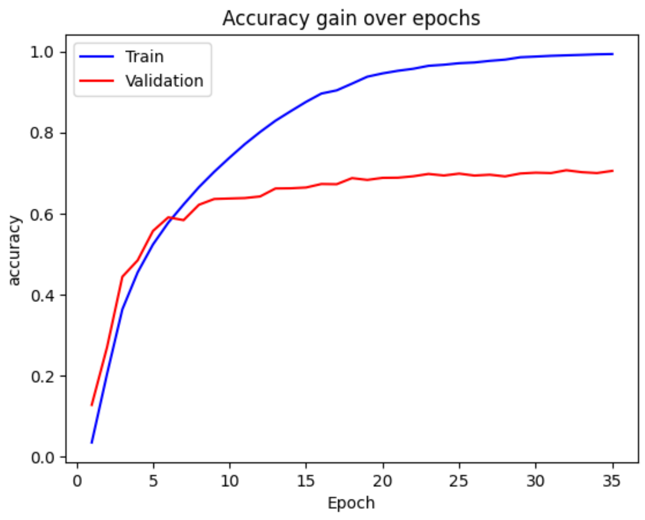
Algorithm of CRNN Model:

1. Initialize a keras sequential model
2. Begin with an input layer of shape (32,128,1) to accommodate images of height 32 and width 128 pixels.
3. Implement a sequence of seven convolutional layers, with six having a kernel size of (3, 3), while the final one employs a size of (2, 2). The number of filters progressively expands from 64 to 512 as we move through the layers.
4. Enhance feature extraction by integrating two max-pooling layers with dimensions (2, 2). Subsequently, include two additional max-pooling layers of size (2, 1) to focus on broader features, particularly useful for predicting longer texts.

5. Improve training efficiency by incorporating batch normalization layers following the fifth and sixth convolutional layers.
6. Adjust the output from the convolutional layer using a lambda function, ensuring compatibility with the forthcoming LSTM layer.
7. Integrate two Bidirectional LSTM layers, each containing 128 units. This Recurrent Neural Network (RNN) component yields an output size of (batch_size, 31, 79), where 79 signifies the complete count of output classes, including the blank character.

**Outcome:**
Based on the depicted plots below, it's noticeable that the model starts exhibiting signs of overfitting at an approximate accuracy of 70%. The strategy of saving the best weights during each epoch has proven advantageous.

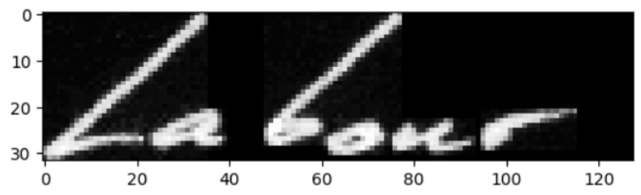




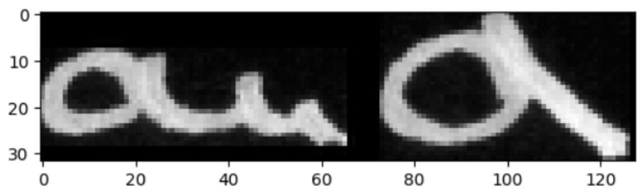
But for a more precise estimation of word accuracy, we measured jaro distance and jaro ratio using levenshtein package. And the results were :

```
Jaro Distance : 94.12%
Jaro ratio : 92.55%
```

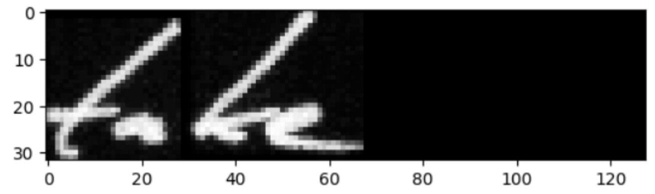
Thus we can conclude that our model has an approximate 93% accuracy in predicting the words from images. Here are a few testing examples:



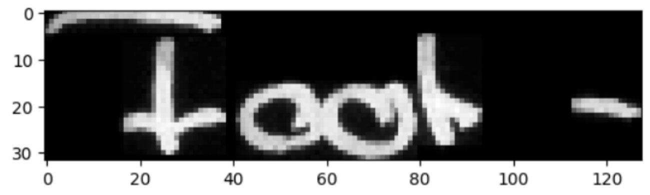
```
original_text =   Labour
predicted text =  Labour
```



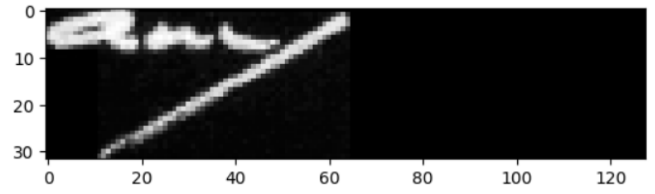
```
original_text =   and
predicted text =  and
```



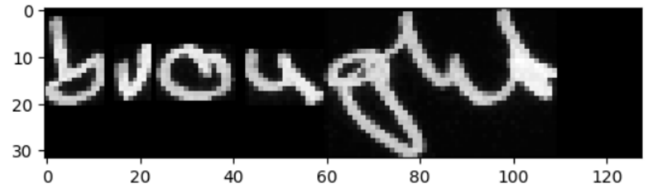
```
original_text =   take
predicted text =  tabe
```



```
original_text =   Foot-
predicted text =  Foot-
```



```
original_text =   any
predicted text =  ayy
```



```
original_text =   brought
predicted text =  brought
```

The model processes the image removes the noise from the image the convolution recurrent neural network algorithm predicts the text. As we can see the model is quite accurate and successfully able to extract the handwritten text. So we can conclude that the model is ready for deployment

**Exceptions considered:**

Exceptions taken into account are outlined below:

1) Uniform coloration is necessary for the text present on the input image, as opposed to being composed of multicolored handwritten text.
2) The image should not feature overly intrusive multicolored backgrounds that extend across the text within the image.
3) The aspect ratio of the input image shouldn't be much higher than 4:1
4) Objects of any nature should not be present in the background intersecting with the text in the image.
5) The image should maintain its original alignment and orientation, without any tilting or rotation

**Enhancement Scope:**

The potential areas of improvement within this project encompass the following:

1) Elevating the model's accuracy can be achieved by integrating predefined models and leveraging robust machine learning GPU processors, ultimately leading to a substantial accuracy enhancement.
2) In subsequent phases, the algorithm could be extended to encompass multiple languages, broadening its applicability and utility.
3) By augmenting the CNN and RNN layers and implementing meticulous data preprocessing, the model can be extended for paragraph extraction.
4) Exploring the integration of CRNN and OpenCV concepts holds the potential for utilizing the model in text extraction from videos.

**Link to Code and executable file:**

Github repository: https://github.com/CaptainHippie/Handwriting-extraction

Main Notebook: https://github.com/CaptainHippie/Handwriting-extraction/blob/main/Handwriting_Extraction_using_CRNN.ipynb

Streamlit cloud deployment: https://handwriting-extraction-n4yrkjqzmdryz58gkuimad.streamlit.app/

Preview video: https://github.com/CaptainHippie/Handwriting-extraction/blob/main/preview.mp4

Sample test images: https://github.com/CaptainHippie/Handwriting-extraction/tree/main/Test%20sample%20images