Ontario Tech University
Faculty of Science, Computer Science

Project Group: Adam and Tyson April 6, 2025

Computer Vision Course Project: Optical Character Recognition Libraries and Trained Keras Model On Written Text

by

Adam Kolodziejczak Tyson Grant

Abstract:

The project utilizes and compares multiple Optical Character Recognition (OCR) libraries through benchmarks like Levenshtein distance and time as metrics while applying Computer Vision and Machine Learning techniques to train the recognizer model from Keras. Initially, IAM Handwritten Forms was used as the data set for the project, which contains hundreds of directories with a typed paragraph along with the written comparison on the same PNG file. However, due to the early analysis and limitations of the data set, the Handwriting Recognition Dataset was used to further the study. Benchmarking between the three libraries was performed for both written and typed text, leading to the training of the Keras recognizer model on written text to compare to the OCR baseline. The GitHub repository can be found here.

1 Introduction

The study analyzes three different Optical Character Recognition (OCR) libraries in Python, PyTesseract, Keras-OCR, and EasyOCR, to compare benchmarks to find a model to train on specified handwritten data. The benchmarks analyzed were recognition accuracy (using Levenshtein distance) and time. Based on the results of the benchmarks, the Keras-OCR recognizer model was trained with the specified data and suitable loss function, with the goal of improving its recognition accuracy in handwritten text. The result consisted of a model that recognized the written text better than its pre-trained model.

1.1 Datasets

The study began with utilizing the Kaggle IAM Handwritten Forms dataset [1] with hundreds of PNG files where each contains a typed paragraph and a handwritten paragraph of the same version. Although helpful in exploring the utilized libraries, its manual implementation of separated handwritten and typed text and ground truth proved difficult when further training an OCR model. Based on the result in the early stages and the scope of the project, the Handwriting Recognition dataset [2], obtained from Kaggle, was used to train our Keras-OCR model, as well as benchmark the three libraries. The dataset consists of training, validation, and test folders. Each folder with a corresponding csv file containing the columns "FILENAME", the filename that matches the corresponding number in the respective folder, and "IDENTITY", the ground truth of the text within the image. Some images in the dataset did not have writing, and others were unreadable, corresponding to ground truth labels of "null" and "UNREADABLE", respectively. Some images contained typed text such as "name" before the handwritten text.

1.2 OCR Libraries

PyTesseract - a Python wrapper for Google's Tesseract-OCR Engine [3], Keras-OCR - which uses a detector and recognizer [4], and EasyOCR - which uses a detector and recognizer, were used in this study [5].

2 Methodology

Applying OCR models and benchmarking the models was done using the AMD Ryzen™ 9 7940HS processor. Training a model was done on an Nvidia GeForce RTX 4070 Super. The libraries implemented consisted of Keras-OCR, PyTesseract, EasyOCR, TensorFlow, OpenCV, Pillow and other libraries used to process and handle data, such as NumPy and pandas.

2.1 Cleaning Data

The cleaning process consisted of modifying the IAM Handwritten Forms dataset and Handwriting Recognition dataset. Repeated files, noted with a naming convention, were removed. The separation of handwritten and typed images was done in the IAM handwritten notes dataset, and the ground truth was created. Note that the separation of handwritten and typed images resulted in images with lower resolution, causing issues that resulted in a change in datasets. As the project scope expanded, the Handwriting Recognition dataset was used. Images with ground truth of "UNREADABLE" or "null" were removed.

2.2 Benchmarking

The Levenshtein distance (edit distance) was used to measure the distance between two strings by calculating the minimum number of single-character edits to go from one string to another by inserting, deleting, or substituting additional characters. The built-in function from the Levenshtein library, `.distance(string1, string2)`, returning the edit distance. This distance was then divided by the length of the longest string and turned into a percentage as in Equation (1) [6].

$$similarity = (1 - \frac{distance}{maxlength}) * 100 \tag{1}$$

2.3 PyTesseract

An input directory was looped through calling `image_to_string`, a built-in PyTesseract function for each image, passing in the image and a configuration to specify the engine model and page segmentation style [3]. The output would write the recognized text into a text file.

2.4 Keras-OCR

The script uses the Keras-OCR library to extract text from images and write it into their corresponding text file. During this process, a function was called to sort the predictions based on the bounding box detected in each image. With this bounding box, it first sorts by the y direction. A threshold for the y direction is checked to ensure slight differences in y don't affect the order. If more significant than the threshold, a new line is created where predictions are grouped. Then, within groups, they are sorted by their x position and are formatted into a text file. Thus, the sorting process was implemented to increase the accuracy of Keras detecting words in paragraph form, aiding in inconsistent handwritten text.

2.5 EasyOCR

The use of the EasyOCR class `easyocr.Reader(['en'])` [5], which contains a recognizer and detector, was used. The detector detects text in an image and creates a bounding box. The recognizer identifies the individual characters within the bounding box. The `readtext()` [5] method combines these steps and was used to parse text from images.

2.6 Trained Model

This sub-section discusses the loss function and training of the Keras recognizer model, utilizing a custom Connectionist Temporal Classification (CTC) function to recognize sequences of characters. The model is trained on only handwritten text. The Keras recognizer base model was wrapped with the CTC loss function due to its ability to overcome recognizing inconsistencies in written text.

2.6.1 CTC Loss

Connectionist Temporal Classification (CTC) loss is used as the loss function in training the model. Due to issues in OCR, such as handwritten text varying in size and spacing, the CTC loss function is used for detected input sequences longer than the label, which may not be perfectly aligned with the output [7]. This allows characters to be matched from one input to an output and deal with the issues mentioned in OCR.

```
k.ctc_batch_cost(labels, y_pred, input_length, label_length)
```

Above is the function provided by Keras for CTC loss [8] that was implemented with the parameters accordingly. As a result, the model can learn how to align input features with output character sequences without manually handling characters, returning a single loss value for each training sample in your batch. Refer to the supplemental document for information on Connectionist Temporal Classification loss and the parameters.

2.6.2 Developing CTC Model

The model used for training handwritten data is the Keras recognizer, a CRNN that extracts sequential features from input images. The model trained consists of the recognizer as the base; however, it also consists of the input parameters needed for CTC loss and is composed of Lambda layers to implement the loss, as seen below [8] (which needs to be specified before training):

```
loss_out = Lambda(ctc_loss_lambda_func, output_shape=(1,), name='ctc')(
[base_output, labels, input_length, label_length])
```

Thus, the code implementation was a function that defined the CTC inputs, got features from the base model, and applied CTC loss using a Lambda layer to create the final model. With the use of the `.compile()` [8] from the Keras library, the recognizer model was compiled with the custom CTC loss function that was written. With the use `.fit()` [8] from the Keras library, the compiled model was trained on the specified data with corresponding inputs and training hyper-features.

2.6.3 Preprocessing Data

All images were converted to grayscale, adding dimension to meet input requirements (height, width, 1). All text labels were encoded into sequences of integers using a character level tokenizer (from Tokenizer) [9], and a padding of zero was applied to keep the shapes consistent for the model. Note that input length and label length padding were ignored for CTC loss. All inputs required for the model, especially CTC loss, were organized into a dictionary and passed as the training input.

2.6.4 Training Model

Training the CTC model (base model being the recognizer from Keras), a batch size of 64 with 20 epochs was used on a training size of 25,000 and a validation size of 4000. Initially, extreme overfitting was evident within five epochs. To diminish this, augmentations were introduced into the training set, allowing for image variability. A function implemented the augmentations by applying a 50% chance of random rotations between -5°and 5°and random noise to the image. The training size was increased to 40,000 with a validation size of 8000. A dynamically adaptive learning rate was also used, `ReduceLROnPlateau` [8], designed to reduce the learning rate, and passed as a parameter to `.fit()` [8], as well as setting shuffling to true. An Adam optimizer [11] was used with a learning rate of 0.00001.

3 Results

3.1 Pretrained Libraries

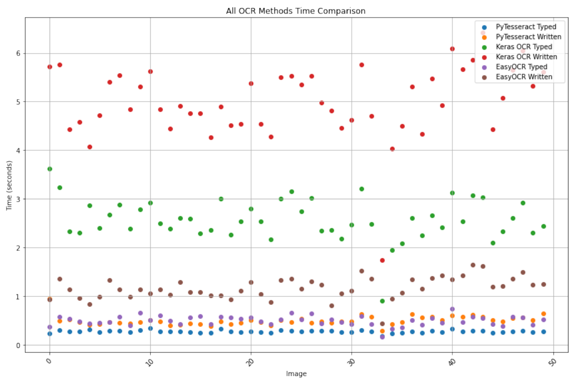
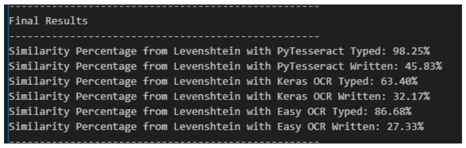


Figure 1: Time Comparison of Each Pretrained Library.



```
----------------------------------------------------
Final Results
----------------------------------------------------
Similarity Percentage from Levenshtein with PyTesseract Typed: 98.25%
Similarity Percentage from Levenshtein with PyTesseract Written: 45.83%
Similarity Percentage from Levenshtein with Keras OCR Typed: 63.40%
Similarity Percentage from Levenshtein with Keras OCR Written: 32.17%
Similarity Percentage from Levenshtein with Easy OCR Typed: 86.68%
Similarity Percentage from Levenshtein with Easy OCR Written: 27.33%
----------------------------------------------------
```

Figure 2: Accuracies measured with Levenshtein distance.

Figures 1 and 2 show that the Keras library and its models perform the worst in both benchmarks for typed and handwritten images. Since Keras loads and runs deep learning models [4], the time to parse text from images is more extensive than in other libraries. For model summaries, please refer to the supplementary material document. Keras' models are designed to be generic, hence the low accuracy. This generic design takes most of the project scope, which will be discussed later. Note that for every library applied, the typed images had faster times and higher accuracies than handwritten images due to these models within the libraries being primarily built and trained for typed text in images.

3.2 CTC Model

Due to the inaccuracies of the Keras model in Section 3.1: Pretrained Libraries, the recognizer model from the Keras pipeline was finetuned and trained to recognize handwritten text. As discussed in Section 2.6: Trained Model, the CTC model of the Keras recognizer base, CTC loss, and input consists of preprocessed images. As a result, the CTC model created consisted of the same number of parameters as the base Keras Model (8,794,267 total, 8,791,707 trainable) [8]. Please refer to the supplementary material document for model summaries.

3.2.1 Accuracy

After trying to minimize the validation loss slightly, to be similar to training loss, the accuracy of the recognizer increased to 84.66% compared to the original Keras model accuracy of 56.27% on the Handwriting Recognition dataset with a size of 8000 test images. When printing the results of some images, it was noted that the lower accuracy tended to be around 60-70%, while higher accuracies tended to hover around 80-100%. Some outliers with an accuracy of 50% were printed too. As such, image TEST_0002.jpg with an accuracy of 50% was printed. The text that was being recognized faded to near illegible. A function created to detect the bounding box was used to if it was an issue regarding the Keras detector locating the text rather than the CTC model for the recognizer.

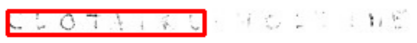
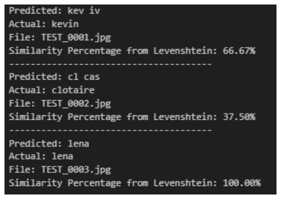


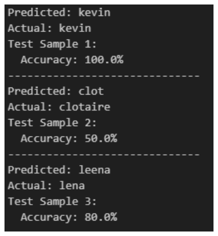
Figure 3: Showing Keras Detector Bounding Box on TEST_0002.jpg

The ground truth for TEST_0002.jpg was “clotaire”, however the predicted was “clot.” As seen in Figure 3, the detector detects the missing values; as such, this was not an issue with the Keras detector. Instead, improvement is still needed in the CTC model, even if it was already performing better than the Keras recognizer. As a result, the augmentation step in preprocessing images was changed to account for the faded text. Steps such as changing the contrast and brightness were done, but a similar overall accuracy of 83.45% was achieved. Note that most of these methods resulted in accuracy identical to or lower than the original. Other methods included CLAHE (adaptive histogram equalization) [10], and an attempt was made to increase the contrast between neighbouring pixels with a kernel convolution. Note that most of these methods resulted in accuracy similar to or lower than the original. Further analyzing the original results, the images without fading text had perfect to near-perfect accuracy. As a result, the model works on text that is written without fading characters seen in Figures 4, 5, 6, and 7. It was also noted that the dataset contains accident characters (those similar to the English alphabet). As a result, the model either learned it as the English version due to ground truth or classified it as two characters.



```
Predicted: kev iv
Actual: kevin
File: TEST_0001.jpg
Similarity Percentage from Levenshtein: 66.67%
--------------------------------------
Predicted: cl cas
Actual: clotaire
File: TEST_0002.jpg
Similarity Percentage from Levenshtein: 37.50%
--------------------------------------
Predicted: lena
Actual: lena
File: TEST_0003.jpg
Similarity Percentage from Levenshtein: 100.00%
```

Figure 4: Keras pretrained Accuracies: Not Faded, Faded and Accent



```
Predicted: kevin
Actual: kevin
Test Sample 1:
  Accuracy: 100.0%
------------------------------
Predicted: clot
Actual: clotaire
Test Sample 2:
  Accuracy: 50.0%
------------------------------
Predicted: leena
Actual: lena
Test Sample 3:
  Accuracy: 80.0%
```

Figure 5: CTC Model Accuracies: Not Faded, Faded and Accent

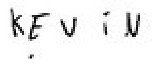


Figure 6: Truth: kevin

Figure 7: Truth: clotaire

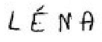


Figure 8: Truth: lena

4 CONCLUSION

The study analyzed pre-trained OCR libraries with standard benchmarks. Benchmarks guided the study in creating a CTC model with the base model of the Keras Recognizer to outperform the pre-trained model in handwritten text recognition. The CTC model outperformed the pre-trained model by applying CTC loss, image pre-processing, and training on the handwritten dataset. To further expand upon this study, a detector model can be trained or created to create an OCR pipeline with the recognizer from this study. Instead of the base recognizer, a custom model for accent characters instead of classifying them to English equivalents. Some possible models are finetuning the existing Keras detector or creating a YOLO model. The YOLO model used as the detector could locate the bounding box, crop and resize the image and use the CTC recognizer created in this study to create an OCR pipeline. However, this study could be applied as the base for programs that detect handwritten notes and parse them into digital formats. Such programs could be beneficial for students, businesses, and day-to-day use

References

[1] Abdalghani, N., *IAM Handwritten Forms Dataset*, Kaggle, Available at: `https://www.kaggle.com/datasets/naderabdalghani/iam-handwritten-forms-dataset/data`.

[2] Landlord, *Handwriting Recognition Dataset*, Kaggle, Available at: `https://www.kaggle.com/datasets/landlord/handwriting-recognition`.

[3] PyPI, *pytesseract*, Available at: `https://pypi.org/project/pytesseract/`.

[4] Robu, R., *Introduction to Keras*, Medium, Available at: `https://medium.com/@RobuRishabh/introduction-to-keras-5337672a96b0`.

[5] Mahajan, A., *EasyOCR: A Comprehensive Guide*, Medium, Available at: `https://medium.com/@adityamahajan.work/easyocr-a-comprehensive-guide-5ff1cb850168`.

[6] Paperspace, *Measuring Text Similarity Using Levenshtein Distance*, Available at: `https://blog.paperspace.com/measuring-text-similarity-using-levenshtein-distance/`.

[7] GeeksforGeeks, *Connectionist Temporal Classification (CTC) for Sequence Modelling*, Available at: `https://www.geeksforgeeks.org/connectionist-temporal-classification/`.

[8] Keras Documentation, *Model Training APIs*, Available at: `https://keras.io/api/models/model_training_apis/`.

[9] TensorFlow Documentation, *Keras: The TensorFlow API*, Available at: `https://www.tensorflow.org/guide/keras`.

[10] OpenCV Documentation, *Histogram Equalization in OpenCV*, Available at: `https://docs.opencv.org/4.x/d5/daf/tutorial_py_histogram_equalization.html`.

[11] GeeksforGeeks, *Adam Optimizer in TensorFlow*, Available at: `https://www.geeksforgeeks.org/adam-optimizer-in-tensorflow/`.

[12] Graves, A., *A Visual Guide to Connectionist Temporal Classification (CTC)*,

Distill, 2017. Available at: `https://distill.pub/2017/ctc/`.

[13] PyTorch Documentation, *torch.nn.CTCLoss*, Available at: `https://pytorch.org/docs/stable/generated/torch.nn.CTCLoss.html`.