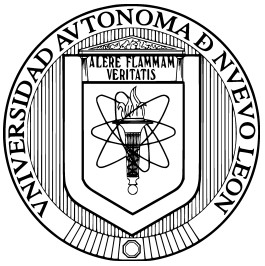
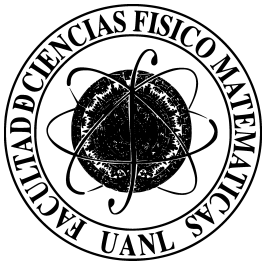


Universidad Autónoma de Nuevo León
Facultad de Ciencias Físico Matemáticas
Artificial Intelligence | Group: 031
Neural Network for Optical Character Recognition
May 15th, 2025



Aldo Hernández
aldo.hernandezt@uanl.edu.mx
Universidad Autónoma de Nuevo León
San Nicolás de los Garza, Nuevo León, MX

Abraham López
abraham.lopezg@uanl.edu.mx
Universidad Autónoma de Nuevo León
San Nicolás de los Garza, Nuevo León, MX

Damián García
gilberto.garciam@uanl.edu.mx
Universidad Autónoma de Nuevo León
San Nicolás de los Garza, Nuevo León, MX

Cristian Antonio
cristian.antoniosnt@uanl.edu.mx
Universidad Autónoma de Nuevo León
San Nicolás de los Garza, Nuevo León, MX

Abstract

This work presents the development of a neural network designed to perform Optical Character Recognition (OCR) on handwritten names. Using a dataset of over 400,000 labeled name images, we built a hybrid architecture combining Convolutional Neural Networks (CNNs) for spatial feature extraction with Bidirectional Long Short-Term Memory (BiLSTM) layers for sequential modeling. The model was trained using the Connectionist Temporal Classification (CTC) loss and evaluated with the Character Error Rate (CER) metric. After multiple iterations involving adjustments to preprocessing, architecture, and hyperparameters, the final model achieved a validation loss below 5 and a CER below 1.2. These results demonstrate the model's effectiveness in handling highly variable handwriting styles. While further improvements such as additional training epochs and architectural enhancements could be explored, the current results validate the approach as a viable solution for handwritten name recognition.

Keywords: OCR, CNN, BiLSTM, handwriting

1 Introduction

Ever since the use of charcoal for cave paintings, writing has become utterly necessary for humans. Whether we may need to write down something important, make a letter for our loved ones, or even just to express feelings we can't say aloud, writing has became a timeless and often recipient-less way of communication that can leave a trace of everything you could ever think of to whoever that can take a glance at any used writing surface.

While text itself can be anonymous, there is something that individualizes text and can create emotional value: the handwriting. Different hand and arm physiology, distinct education, personal preference, individual amount of care–these are some factors that shape the way each individual writes. Aside from being a beautiful and artistic way to showcase a person's personality, this unique trait can sometimes become a challenge.

Have you ever missed a class and struggled to understand your classmate's notes? Or looked at a medical prescription after a checkup only to find it completely unreadable? These are just examples of why handwriting, despite its beauty, can become a problem. Optical Character Recognition (OCR) offers a solution–allowing computers to make up for these issues by taking the handwritten characters and translating them into legible text. In this article we will explore how we built a neural network capable of doing that.

2 Methodology

Before we design the neural network, we need to find a big and useful dataset to build the network on. For this, we used a set of images [Kaggle, 2020] containing more than 400,000 handwritten names collected through charity projects. This dataset was chosen because it accurately depicts the huge variation in individual writing styles while the images have a great quality with none to little visual noise.

2.1 Initial Exploration

Though this dataset seems to be perfect, it could present some problems. For example, it may have an unbalanced count of characters since the words are names and surnames or contain letters that are not present in the English vocabulary. To know if this set can be used or not, we explored the dataset containing all the images and names. The frequency of all the present characters is showcased in Figure 1, we can see that while some letters and all vocals have a high presence in the dataset, there are other characters that have a significantly small count; this might be a problem when creating the model since there will be less sample data for training which could reduce the accuracy in these letters.

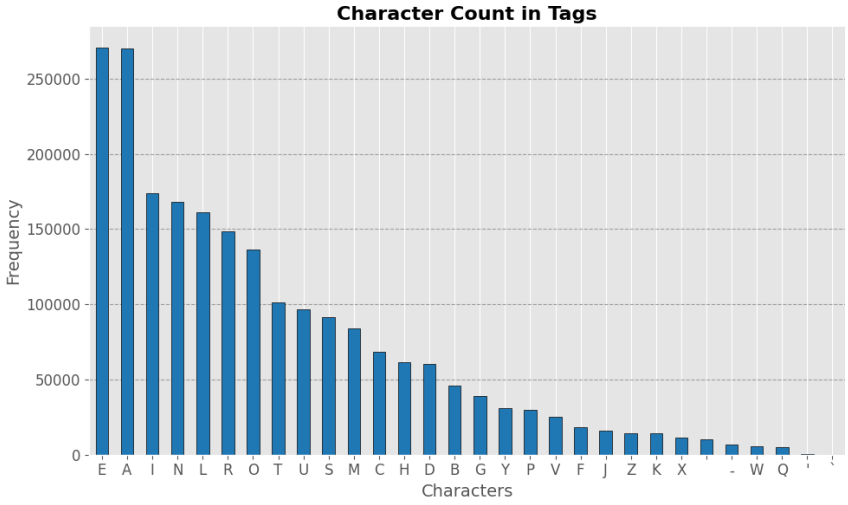


Figure 1: Character presence in dataset.

After detecting this potential issue, we decided to still use this dataset since even if some characters have a relatively small count, it is still a high absolute value. We then proceeded to take a look to the dataset as shown in table 1; the model will later access these images with the function `download_dataset_base_path`.

#	FILENAME	IDENTITY
0	TRAIN_00001.jpg	BALTHAZAR
1	TRAIN_00002.jpg	SIMON
2	TRAIN_00003.jpg	BENES
3	TRAIN_00004.jpg	LA LOVE
4	TRAIN_00005.jpg	DAPHNE

Table 1: Training set head.

2.2 Variables and Hyperparameters

After choosing and exploring the dataset, we decided to use the *Keras* and *TensorFlow* frameworks in order to build our neural network. To start with, we imported the necessary libraries in our script, then we took on setting up the constants and hyperparameters for the model as shown in table 2. Next, we defined some global dynamic variables that will have importance during data preprocessing such as filenames and model callbacks.

Hyperparameter/Constant	Value
dataset_url	https://www.kaggle.com/api/v1/datasets/download/ssarkar445/handwriting-recognitionocr
img_height	60
img_width	128
input_c	1
epochs	200
initial_learning_rate	0.001
batch_size	1024
AUTOTUNE	`tf.data.AUTOTUNE`
empty_prediction_penalty	50.0

Table 2: Definition of hyperparameters and constants.

2.3 Metric

We defined a custom class to compute the **Character Error Rate** (CER) as the metric to evaluate the performance of our model. This value measures the number of character-level errors (insertions, deletions, substitutions) made by the model when predicting a character sequence and comparing it to the target.

This class includes three key fields: `cer_accumulator`, `num_samples`, and `actual_padding_token`. When the metric is updated, the `ctc_greedy_decoder` converts the model's output logits (which represent the probability distribution over characters at each time step) into a predicted sequence by selecting the most likely token at each time step. After decoding, padding is removed from the ground truth (label) and the edit distance is calculated. CTC decoding is necessary for alignment-free predictions, especially since the input and output length may differ.

The total edit distance is added to `cer_accumulator`, and the number of samples in the current batch is added to `num_samples`. The final CER is calculated by dividing the accumulated edit distances by the total number of characters in the target labels.

To discourage behavior such as the model defaulting to blank outputs, we introduced a penalty for empty predictions. This encourages the model to attempt meaningful outputs.

2.4 Data preprocessing

In order to access the data, we defined two functions: `download_dataset_base_path`, and `get_initial_metadata_from_csv`. The first function extracts the dataset in a determined path that is returned, and the second function reads the dataset to retrieve important information such as all the characters present, the length of the biggest label, filenames, and their corresponding labels.

For the *TensorFlow* input pipeline, we use several functions to prepare image-label pairs for efficient model training. The `load_and_process_image` function reads the image file as raw bytes, then decodes it into a grayscale tensor, resizes the image to a consistent height and width using padding to preserve aspect ratio, and normalizes the pixel values from a [0, 255] range into [0, 1]. It returns the processed image with the raw label string and its length.

The `vectorize_label_tf` function splits the label into characters and converts them into an integer, then pads the sequence with a -1 token (required for batching variable-length sequences) to a fixed length and returns the padded label with the original length.

Finally, the `create_tf_dataset` function creates the dataset that will be used for the *TensorFlow* model, using the previous helper functions to map the original dataset, preparing it for model use. Then the processed data is cached into the memory.

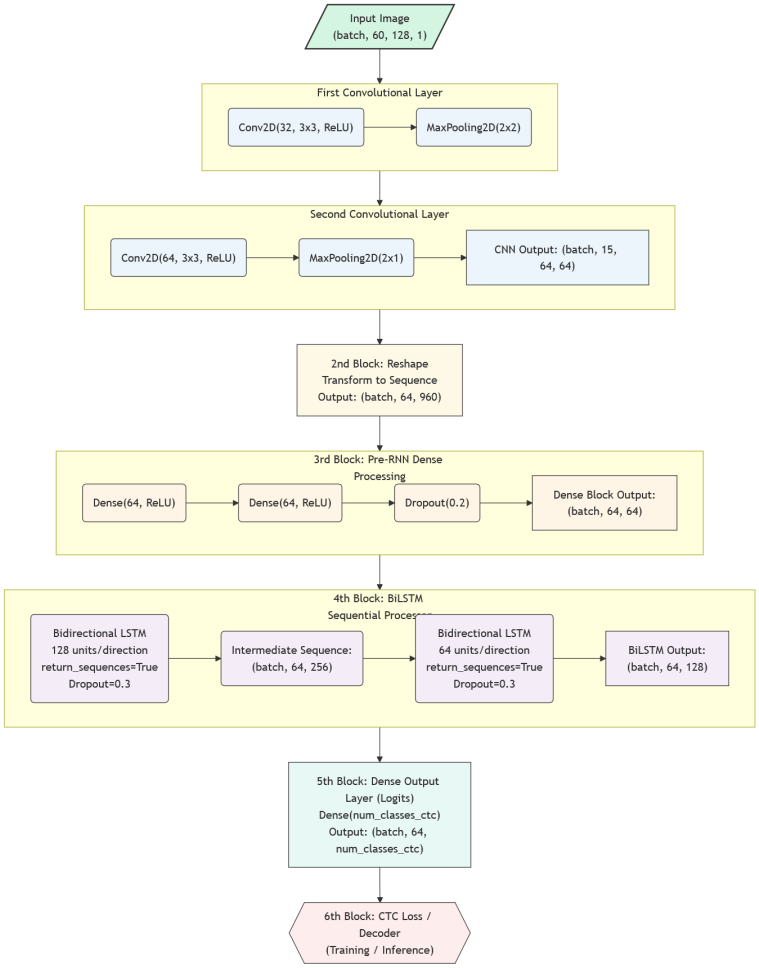
2.5 Model class

For our model we define a custom class **CTCOCRModel** that creates the neural network and manages the metric explained in section 2.3. This class implements custom `train_step` and `test_step` methods allowing control over how loss is computed, direct integration with the metric, and dynamic loss computation with sparse tensors to support variable-length labels. **CTC Loss** is used since it allows training without requiring aligned input and output sequences, which is ideal for OCR tasks where the position of characters in the image isn't strictly aligned with output positions.

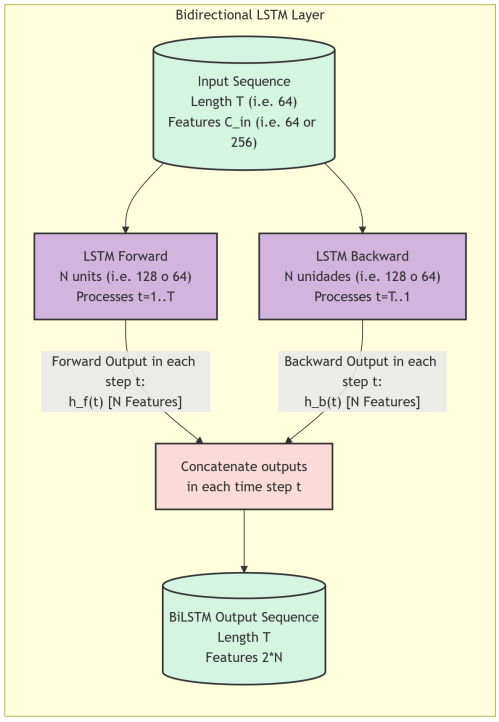
2.5.1 Network Architecture

The implemented network has a **hybrid architecture** that combines a convolutional network (CNN) with a bidirectional long short-term memory network (BiLSTM), as shown in figure 2a.

The model begins with a convolutional block composed of convolution and pooling layers that extract spatial features from the input image, then a reshape layer converts the 3D feature map output from the CNN into a sequence of vectors suitable for recurrent layers, next the fully connected dense layers refine the extracted features to allow more abstract representations, after that the dropout layer prevents overfitting by encouraging redundancy in learned features, subsequently the bidirectional LSTM layers process the sequence in both directions (figure 2b), and finally the output dense layer outputs logits for each character class at each time step (position), making the output compatible with the Connectionist Temporal Classification (CTC) loss function.



(a) Model's neural network architecture.



(b) BiLSTM layer process.

Figure 2: Neural network architecture and BiLSTM process.

2.6 Main Workflow and Model Configuration

The principal workflow is defined as follows:

1. Obtain metadata and image/tags pairs from the training dataset.
2. Start the `StringLookup` object to map characters to integers.
3. Create the `tf.data.Dataset` dataset that will actually be used to train, validate, and test the model.
4. Calculate the number of classes for CTC and prediction length.

The output of this cell block can be seen in table 3. As shown, there are 33 different classes: 26 letters, and 1 character for back-tick, apostrophe, space, empty, unknown, and dash.

Parameter	Value
Global Vocabulary (30 chars)	`-ABCDEFGHIJKLMNOPQRSTUVWXYZ``
Max Label Length (from training set)	34
TF StringLookup Vocabulary (32 tokens)	`['', [UNK], ' ', ''', '-', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '`']`

Table 3: Character Vocabulary and Label Parameters.

2.7 Model Build and Loading a Model

To build the model we create an instance of the class defined in section 2.5 with the parameters stated in table 4. After the model is defined, it is compiled with an **Adam** optimizer with an initial learning rate of 0.1, then it is built and the model summary can be consulted in table 3a.

Parameter	Value
Image height	60
Image width	128
Input channel	1
Number of classes for CTC	33
Prediction model output length	64

Table 4: Model Configuration Parameters.

Layer (type)	Output Shape	Param #
`convolutional_block (Sequential)`	(None, 15, 64, 64)	18,816
`reshape_to_sequence (Reshape)`	(None, 64, 960)	0
`dense (Dense)`	(None, 64, 64)	61,504
`dense_1 (Dense)`	(None, 64, 64)	4,160
`dropout (Dropout)`	(None, 64, 64)	0
`bidirectional (Bidirectional)`	(None, 64, 256)	197,632
`bidirectional_1 (Bidirectional)`	(None, 64, 128)	164,352
`output_logits (Dense)`	(None, 64, 33)	4,257

(a) Layer-wise architecture

Total parameters	450,721 (1.72 MB)
Trainable parameters	450,721 (1.72 MB)
Non-trainable parameters	0 (0.00 B)

(b) Parameter summary

Figure 3: Model Summary. Architecture and Parameter Counts.

2.7.1 Loading a previous model

Since we wanted to test the model every 300 epochs we trained on it, we designed a workflow that allows us to load a previous model to restart training on the actual number of epochs. This was essential for the development of the neural network because it let us see how the model was performing at a certain number of epochs and run some tests on the predictions.

This workflow is mainly for previously trained models, but can also take only the weights to run tests on the testing dataset and visualize predictions.

2.8 Training

After the model is successfully built, it is trained on the training dataset for a certain number of epochs. Once training is complete, the model and weights are saved as files to allow future loading as explained in section 2.7.1.

The model is trained with the **CTC Loss** function and the **CER** metric. It also has two callbacks that implement techniques to increase the training efficiency: `EarlyStopping` to stop the training if `val_loss` does not get better after 6 epochs, and `ReduceLROnPlateau` that decreases learning rate when the validation metric no longer reduces.

3 Previous Models

The initial model yielded extremely poor performance, likely due to inadequate training duration and suboptimal preprocessing. After some tweaks and fixes to the source code including the following:

- Changes in how data was loaded and processed.
- Adjustments in how the labels were vectorized and padded.
- The implementation of a personalized class for the model (as detailed in section 2.5).
- Tweaks to the CNN block output and the reshape layer.
- Fixes to the metric, resulting in the final version stated in section 2.3.
- Add a class to the total number of classes for the model's output.

Despite these improvements, the performance remained below expectations, particularly with respect to the CER and validation loss. The comparison between said models is displayed in table 5.

Model Version	First	Second
Epochs	100	100
Learning rate	0.001	0.0001
Batch size	1024	1024
val_loss	727.3027	5.26555
CER	N/A	1.2292

Table 5: Comparison between first two models.

4 Results

After picking up the second model to restart training from where it was left off, we added 112 epochs and changed the learning rate to 0.001. The resulting model is the final result of the article, it reached a validation loss of less than 5 per epoch, and a CER per epoch smaller than 1.2, as seen in figure 4. Some of the made predictions over the test set are showcased in figure 5.

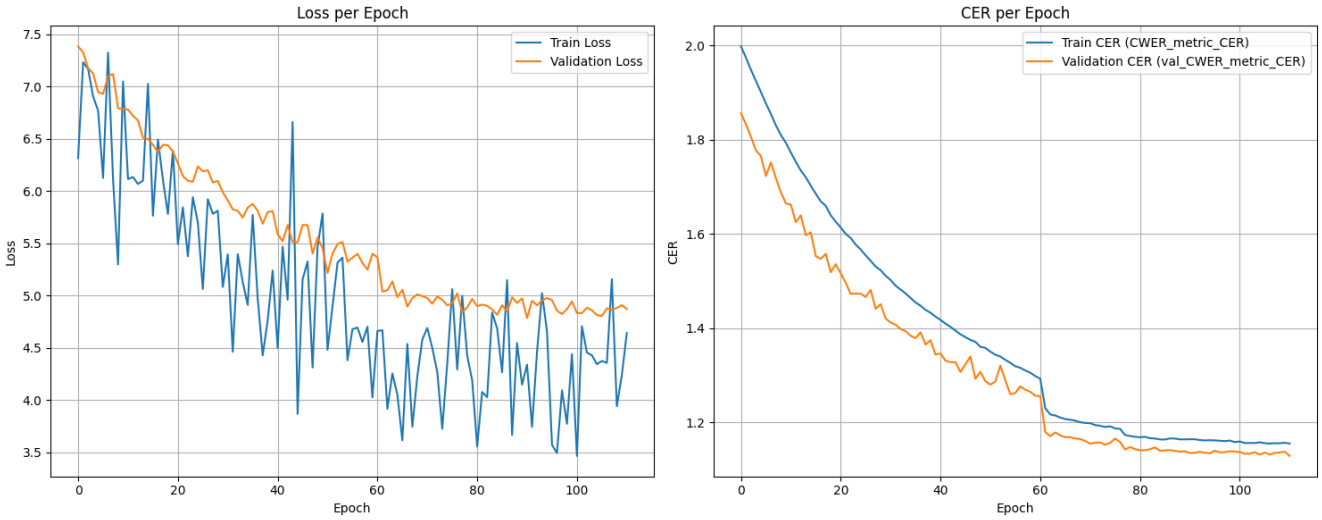


Figure 4: Metrics for the third model.

Figure 5: Predictions made by the model.

5 Conclusions

The results demonstrate that the model is capable of effectively predicting names with a relatively high degree of accuracy. This indicates that the current architecture and training regimen are sufficient for capturing the underlying patterns in the dataset. However, there remains room for further improvement and refinement.

For instance, extending the training process by an additional 100 epochs may allow the model to converge more fully and potentially achieve better generalization. Additionally, increasing the number of filters in the convolutional layers could enable the network to learn more complex and nuanced features

from the input data. These adjustments could lead to enhanced performance, particularly in edge cases or less common naming patterns.

Nonetheless, implementing such modifications goes beyond the scope and intent of this article, which primarily aimed to explore the feasibility of using this specific neural architecture for name prediction tasks. Future work may consider these directions in order to further optimize and expand the capabilities of the model.

References

[Kaggle, 2020] Kaggle (2020). *Handwriting recognition*. Accessed: 2025-05-15. https://www.kaggle.com/datasets/landlord/handwriting-recognition