Beyond Handwritten Text Recognition with CNNs

Deep Learning Final Project
5/6/2024

Bruno Manzano Clotet U199123
Paula Mateos Marín U198733

Table of contents

Introduction

Handwritten text recognition (HTR) is a critical area in deep learning with various applications, including digitizing historical documents, facilitating data entry, and improving accessibility. With this project we aim to develop a HTR system capable of recognizing handwritten text from images and potentially correcting spelling errors.

This project emanates from some already existing projects that will be correctly listed and cited when needed. Our idea is to use they're structure but change the actual CNN they are using to train the model to improve it. For that, we will dive into the best structures for HTR available and study them.

But we also aim to improve the accuracy of this model, for that, we will study various ways of improving the accuracy that will range from hyperparameter optimization to potentially some use of RNNs that we will develop later. Actually this last, is the main objective of our project, since when brainstorming, we came up with the idea to adding an RNN to a HTR CNN to improve its accuracy and general performance, and throughout the investigation process we came to the realization that not only this was a good idea, but that it had been done and studied before.

1. Problem Complexity and Suitability of CNNs

Handwritten text recognition presents a challenging problem for computers due to several factors:

- **Variability:** Handwritten characters exhibit significant variations in style, size, and shape compared to printed text. The same letter can be written in many different ways by different people, or even by the same person over time.
- **Noise:** Images of handwritten text can be noisy due to factors like scanning imperfections, ink bleeding, pen pressure variations or even paper ink absorption and other paper characteristics.
- **Segmentation:** Separating individual characters within a word or sentence can be difficult, especially for cursive handwriting or words written closely together. Some people can encadenate characters differently depending on the concrete character. As an example, one of the two authors of this project encadenates the e and the s when they are consecutive, like in "likes"

2. Why using a CNN?

Why are CNNs the best approach to this problem? Well we can get the answer to that by learning more about their characteristics:

- **Local Connectivity:** CNNs capture local features in an image, making them suitable for identifying patterns in small regions that correspond to individual characters.
- **Parameter Sharing:** By sharing weights across convolutional filters, CNNs can learn efficient representations of features despite the large number of possible variations in handwritten characters.
- **Pooling:** Pooling layers in CNNs help to reduce spatial dimensionality while maintaining essential information, making the model more robust to noise and slight shifts in character position.

Project Structure

The project will be organized into the following sections:

1. Literature Review:

Sub-problems of Character Recognition

Before diving into the training and model design coding, it's essential to understand the subproblems involved in HTR, since it is not only training the model to recognize the characters, we also need to separate the characters of our original input, since the input are phrases or words, so some segmentation needs to be done among others:

1. **Preprocessing:** Image preprocessing usually includes resizing, grayscale conversion, and noise reduction to prepare the image for character recognition.
2. **Line Segmentation:** In some cases, documents might contain multiple lines of text. Techniques like horizontal projection profiles or morphological operations can be used to separate lines.
3. **Word Segmentation:** Words within a line need to be isolated, often using connected component analysis, morphological operations, or density-based approaches.
4. **Character Segmentation:** This is the most crucial step, separating individual characters within a word.
5. **Normalization:** Character images might be resized and normalized to a standard size before feeding them into the CNN for recognition.

Previous Approaches to Character Recognition in HTR

Of course, this problem as big and important as it is has had many approaches throughout the years, some of them are worth mentioning for what they have teached about the matter.

- **LeCun et al. (1998):** Pioneered the use of CNNs for handwritten digit recognition (MNIST dataset).
- **Bluche et al. (2009):** Developed a CNN architecture specifically designed for HTR, achieving high accuracy on character recognition benchmarks.
- **Xiao et al. (2016):** Utilized a combination of CNNs and recurrent neural networks (RNNs) for improved HTR performance.

These are just a few examples, and many other approaches exist. For our project, we will focus and take a lot of inspiration from both Bluche and Xiao's work.

2. Base model

For the problem solving of this task that we have previously explained, we chose to base our work on top of a model we found on the internet. This model can be found at: https://www.kaggle.com/code/aman10kr/offline-handwritten-text-ocr/notebook.

Model basic explanation

The code provided on this notebook utilizes a pre-trained OCR model for character recognition. It is a Convolutional Neural Network (CNN) trained on a large dataset of labeled character images.

The code itself doesn't explicitly perform segmentation (separating characters). It relies on OpenCV's contour detection to identify potential character shapes in the image. This approach has limitations and it may not be the most accurate one.

Here's how the code uses the pre-trained model for character recognition in simplified way.

1. Isolates potential character regions based on contours.
2. Preprocesses each region (resizing, normalization).
3. Feeds the preprocessed image to the CNN model.
4. The CNN model predicts a probability distribution for each character class (e.g., A-Z, 0-9).
5. The character class with the highest probability is considered the recognized character.

Here's the layer distribution of the cnn:

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 32, 32, 32)	320
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_4 (Conv2D)	(None, 14, 14, 64)	18,496
max_pooling2d_4 (MaxPooling2D)	(None, 7, 7, 64)	0
conv2d_5 (Conv2D)	(None, 5, 5, 128)	73,856
max_pooling2d_5 (MaxPooling2D)	(None, 2, 2, 128)	0
dropout_2 (Dropout)	(None, 2, 2, 128)	0
flatten_1 (Flatten)	(None, 512)	0
dense_2 (Dense)	(None, 128)	65,664
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 35)	4,515

Convolutional Layers: These layers are designed to extract features from the input image. Convolution involves applying a filter to the image, producing a feature map. The filter highlights specific features in the image, such as edges or lines. By applying multiple filters, the model can learn to detect a variety of features that are important for recognizing characters.

Pooling Layers: The pooling layers (MaxPooling2D) are used to reduce the dimensionality of the data. This is important because it helps to reduce the number of parameters in the model, which can help to prevent overfitting. Overfitting is a problem that occurs when a model memorizes the training data too well and is not able to generalize to new data.

Flattening Layer: The flattening layer (Flatten) is used to reshape the data from a three-dimensional tensor into a one-dimensional vector. This is necessary because the dense layers in the model can only accept one-dimensional input.

Dense Layers: These layers are used to classify the characters in the image. The first dense layer (with 128 units) is followed by a dropout layer (Dropout) with a probability of 0.25. Dropout is a technique that is used to prevent overfitting. The second dense layer (with 35 units) has a softmax activation function. The softmax activation function outputs a vector of probabilities, where each probability corresponds to the probability that the input belongs to a particular class. In this case, the classes are the different characters that the model can recognize.

And this are the parameters of that CNN:

- Total params: 161,690 (631.60 KB)
- Trainable params: 161,690 (631.60 KB)
- Non-trainable params: 0 (0.00 B)

After passing through the CNN, this is how the model does the translation to Digital Outputs:

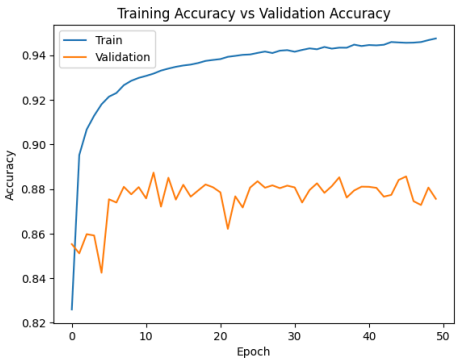
1. The CNN model outputs a vector of probabilities, one for each character class it can recognize.
2. The get_letters function decodes the predicted probability distribution using the LB.inverse_transform method. This likely refers to a Label Binarizer that was used during model training to convert character labels (e.g., "A") to numerical representations.
3. The final output is a list containing the recognized characters for each detected contour (potential character) in the image.

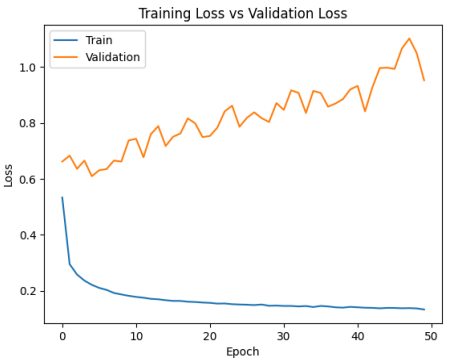
Overall, the code utilizes a pre-trained CNN model for character recognition. However, it relies on external libraries like OpenCV for basic segmentation (contour detection), which can be a source of errors.

Model results

Now that we know how the model works, it is time to see it's overall results:

1. **Accuracy of train and test (they call test validation)**





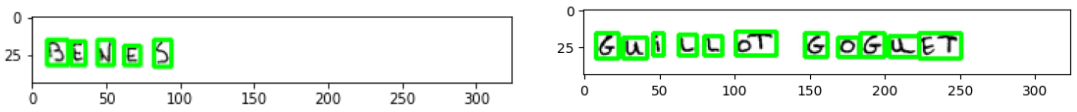
Epoch 50/50

- accuracy: 0.9488
- val_accuracy: 0.8756
- loss: 0.1280
- val_loss: 0.9530

As we can see, this is not the best of models. The train accuracy curve is very good, but the test (or validation) is far from being good. We also observe that the test loss is very bad and in fact increases over time instead of decreasing.

Overall these results help us on our project, since they mean the model has lots of room for improvement, and that is exactly our objective.

2. **Segmentation visualization**
   Luckily this notebook also contains a part where it helps us visualize how the segmentation is being done.



   As we can see the segmentation is not always perfect and some words overlap. We've also realized that this code is not separating words, so when translating the second image the output is (if it had 100% accuracy, that it doesn't) "GUILLOTGOGUET". This also leaves room for improvement, since it can be an interesting improvement for the model to be able to segment by words.

Hyperparameters

Since the optimizer we are working with is adam, there is no concrete defined learning rate. Apart from that, here are some parameters that may be interesting.

```
history = model.fit(train_X,train_Y, epochs=50, batch_size=32,
validation_data = (val_X, val_Y),  verbose=1)
```

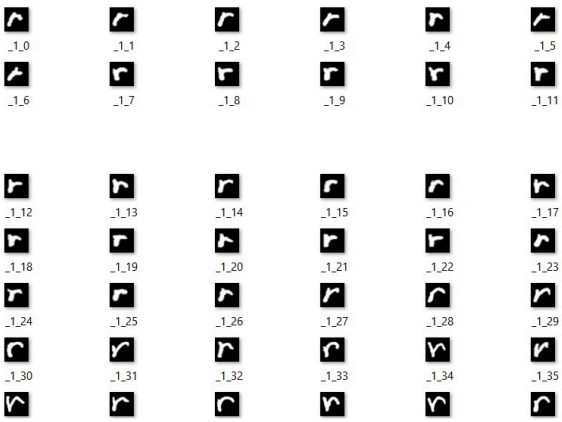
Total params: 162,851 (636.14 KB)

3. Data Acquisition and Preprocessing:

In this section we will talk about the dataset chosen for this project. The model we are bassing this project on, and the reason why we chose it, has a great dataset we will now describe.

Chosen dataset

The idea is that the model is being trained with a dataset with samples of each character on the abecedarium and some special characters as @ as well as numbers. In total, 32 different characters. In total 140000 different samples only to train. And 15000 to train (on the original model). The model itself is only trained to recognize characters.



Once the training is done, what is done is to pass the following database for it to do the recognition. Mainly, the words are segmented into characters and then passed to the model.This dataset has 41370 samples representing 20280 different names. The problem with it, is that it isn't a clean dataset, a huge percentage of it are photos that say: "PRENOM: (and whatever name)", or on other cases prenom is cut in some way that over all difficulties a lot the segmentation and HTR job.

41370 unique values	20280 unique values
TEST_0021.jpg	NAMIZATA FATIM
TEST_0022.jpg	FOURNEL
TEST_0023.jpg	DICINTIO-ILLESCA
TEST_0024.jpg	BARDOT
TEST_0025.jpg	DUVAL
TEST_0026.jpg	ANTONY
TEST_0027.jpg	LISA

The great advantages of these datasets is that both have the ground truths implemented, so we didn't have to spend time writing more than 200000 ground truths for our samples.

Data processing or preprocessing

1. **Data augmentation:**
   For this task, we implemented a classical data augmentation function that included rotation, zoom and even horizontal and vertical flip. Of course, we soon realized that horizontal and vertical flip had no sense at all, since letters as p and q or p and b didn't support it.

   Once this was solved, we realized that data augmentation wasn't improving at all the model performance. We believe this was due to the fact that the characters dataset was already very complete and clean, zoom cut through some characters, and rotation changed some, so overall, we decided to not use data augmentation.

   This taught us how, once one has a great dataset, things can simplify a lot.

2. **Modifications on the dataset**
   For the purpose of our experiment we didn't need numbers or special characters like the original dataset had, so we dropped those. So the final character dataset consisted of 26 characters and hundreds of photos of different handwritings of those characters.

   Once the dataset contained the proper classes we investigated a bit more the randomness with which the data was being picked for training and testing and we found out it wasn't random at all. The base project was using only lowercase letters to train and was testing with both lower and uppercase letters. Obviously this was a problem because the model wouldn't be able to predict uppercase letters this way. For this matter we added both lowercase and uppercase letters in training and testing and we shuffled the datasets as well to avoid any weird picking.

   Those were the changes that had more impact on the accuracy of our model.

3. **Character segmentation**

   The base CNN had a segmentation function which didn't perform correctly so we decided to change that. As we explained, segmentation is key in this project because the CNN is only trained with characters and we want to predict larger forms as documents or words in this case.

   We made several changes in the function such as adjusting the "character thickness detection threshold" which at the start was very rough and insensitive to some letters if they were close to one another.

   We also added error handling to check if the image was successfully loaded. If not it printed an error message avoiding potential crashes or weird behaviors.

   The base segmentation function didn't manage images with no contours (the outline of a character) found, so we also error-managed that.

   Finally we did some minor changes to enforce the robustness of the whole function such as converting the bounding boxes coordinates to integers and outputting control messages all along the code.

4. **Word segmentation**

   We tried duplicating our segmentation function and modifying it to allow multiple words and detect spaces but as we were a bit short of time we didn't fully implement it and thus, we didn't use it in our model.

   However, it is worth mentioning that we applied a density based method which computed the distance between the right coordinates of each bounding box and the left coordinates of each next bounding box and added a space to the prediction when it exceeded a certain threshold. Our idea was to make the threshold relative to the average width of the bounding boxes so the length of the spaces made sense.

   As said, this was only theory, we didn't manage to fully implement the new function.

4. Model Design and Implementation:

Arriving at this point on our project, it is now time to talk about the model of the CNN itself. Since as we have already seen, the CNN on the base model may not be the best since the accuracy is quite bad, so we've decided to change it completely. And for that, we've decided to use some information we learned when reading the Bluche et al. (2009) investigation on developing a CNN architecture specifically designed for HTR, achieving high accuracy on character recognition benchmarks and posterior comments.

This is what we learned from our reading:

1. Add more convolutional layers: Convolutional layers are responsible for learning local features of the data. In the context of image processing, these can be edges, textures, shapes, etc. By adding more convolutional layers, our model can learn more complex representations. However, each additional layer adds more parameters to the model, which can slow down training and increase the risk of overfitting. Therefore, it's important to find a balance.

2. Add Batch Normalization: Batch Normalization is a technique for improving the speed, performance, and stability of neural networks. It standardizes the inputs to a layer, which can help to mitigate problems caused by poor initialization and speed up training. It can also have a slight regularization effect, similar to Dropout.

Now that we have all this in mind we can now explain our final CNN, it is a sequential model, which means that each layer in the network feeds into the next one. Here's a step-by-step explanation:

1. Conv2D Layer (32 filters, 3x3 kernel size): This is the first convolutional layer of the network. It takes an input image of size 32x32 with 1 channel (grayscale). The layer applies 32 different filters, each of size 3x3, to the input image. The 'relu' activation function introduces non-linearity to the model, allowing it to learn more complex patterns.

2. MaxPooling2D Layer (2x2 pool size): This layer reduces the spatial dimensions (width, height) of the input by taking the maximum value over the window defined by pool_size for each dimension along the features axis.

3. Dropout Layer (0.3): This layer randomly sets a fraction (0.3) of input units to 0 at each update during training time, which helps prevent overfitting.

These steps are repeated 3 times.

After several convolutional, pooling, and dropout layers, our model flattens the 3D outputs to 1D and then uses two dense (fully connected) layers. The final layer uses a 'softmax' activation function, making it suitable for multi-class classification (26 classes in our case).

This model was way more complex to get to than what we expected, since our first guess was to add more convolutions and more parameters, but all trials overfitted, so finally, we dropped from that idea. We also tried using batch normalization as Bluche recommended, but that also didn't work for us. And after many many trials and errors, we realized the original CNN was a very good fit for the model, since a simple approach was what seemed to be best. Overall we added dropout to the model, which seemed to improve a lot reducing overfitting.

Another thing we realized was that when training and testing the functions prior to the main CNN were "cheating" or doing something that didn't make sense at all, since as said before, they used lowercase letters for the training and uppercase and lowercase letters for the test. So by changing that to make it more representative, shuffling the dataset to make it more random and some other data related tasks, we spend a lot of time cleaning the data and organizing it well.

Finally, here are the parameters of the network:

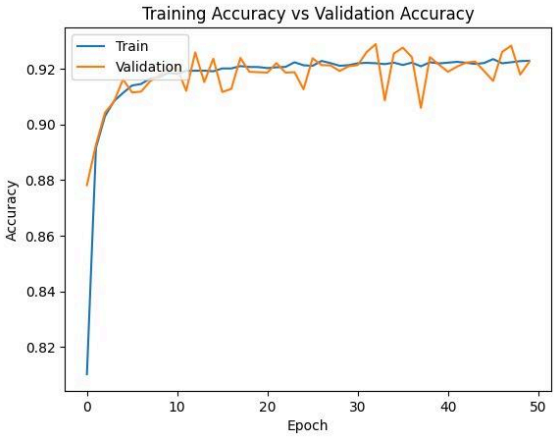
- Total params: 161,690 (631.60 KB)
- Trainable params: 161,690 (631.60 KB)
- Non-trainable params: 0 (0.00 B)

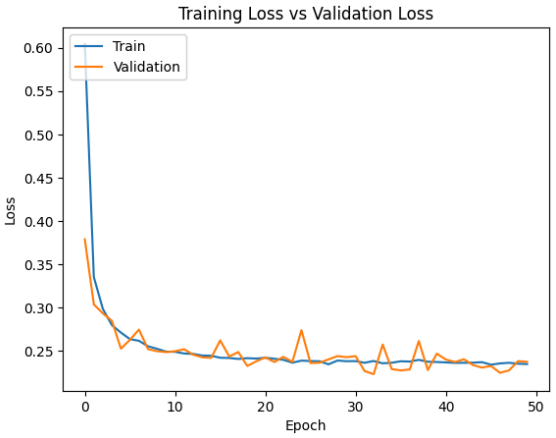
Here you can find a visualization of it and it's parameters:

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 32, 32, 32)	320
max_pooling2d_4 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_3 (Dropout)	(None, 16, 16, 32)	0
conv2d_5 (Conv2D)	(None, 14, 14, 64)	18,496
max_pooling2d_5 (MaxPooling2D)	(None, 7, 7, 64)	0
dropout_4 (Dropout)	(None, 7, 7, 64)	0
conv2d_6 (Conv2D)	(None, 5, 5, 128)	73,856
max_pooling2d_6 (MaxPooling2D)	(None, 2, 2, 128)	0
dropout_5 (Dropout)	(None, 2, 2, 128)	0
flatten_1 (Flatten)	(None, 512)	0
dense_2 (Dense)	(None, 128)	65,664
dropout_6 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 26)	3,354

5. Evaluation

Now that the CNN and it's preprocessing is clear, it is time to visualize and evaluate the results of the model.





As it is clear on the images, this model has way more equilibrium now, since the train and test accuracies are very similar, and so are the losses. In concrete, we have an overall accuracy of 92% and an overall loss of 0,24. Which are overall satisfactory results.

But of course, this accuracy isn't the most important one, and it is time to truly test the model. For that, we have used the names dataset we commented on before. We have created a function to be able to evaluate this results, concretely, the function computes the similarity between CNN and ground truth:

- **Levenshtein Distance Function**: A helper function to compute the Levenshtein distance between two strings.
- **Similarity Calculation**: For each pair of strings, calculate the similarity based on the Levenshtein distance.
- **Global Similarity**: Calculate the global similarity as the average similarity across all string pairs.

And the output of it has been:

- Total Pairs: 9592,
- Global Similarity: 55.27753121701624}

It is also important to note that, with the base model, this similarity was 37.386678. We now see that due to the huge quantity of noise and the low quality of the dataset, the similarity is very low. Nevertheless, an improvement over the base model has clearly happened

6. Improvements and additions:

Now that the model is complete, it is time to go into detail about the addition of the RNN, but first of all, let’s talk about why it makes sense to add one:

1. **Capturing Sequential Information:** CNNs excel at extracting spatial features from images, which is useful for recognizing individual characters in HTR. However, they struggle to capture the sequential nature of handwriting, where the order of strokes matters. RNNs, on the other hand, are specifically designed to handle sequential data. By adding an RNN on top of the CNN features, we can leverage the strengths of both architectures. The CNN can extract character-level features, and the RNN can learn the relationships between these features across the sequence, improving recognition accuracy, especially for cursive or complex handwriting.

2. **Contextual Dependence:** In HTR, recognizing a character can be influenced by its surrounding characters. For instance, the letter "o" can look quite similar to the letter "a" depending on handwriting style. An RNN can analyze the sequence of characters, allowing it to consider contextual information and make more accurate predictions.

Creating the dataset for the RNN

For us, one of the key parts of this idea of adding an RNN is to be able to use the outputs of the CNN as inputs for the RNN. This will be possible due to the fact that the CNN outputs when we pass it the name dataset is the transformation and the ground truth both in string format, so from there we want to convert it to a dataset and input it to the RNN so it learns.

For this we will use 10000 sample, 7000 for training and 3000 for test.

The code we used to create this dataset for the RNN worked by leveraging the predictions from a pre-trained CNN model. It focuses on a subset of the original data, processing 10,000 samples to ensure training efficiency. Here's how it we achieve this:

1. **Iterating Through Labeled Images:** The code iterated through a CSV file containing two key pieces of information for each sample: the filename of the handwritten image and its corresponding true label.
2. **Extracting Information and Predictions:** For each image listed in the CSV file, the code first verified its existence. If the image was found, it employed a separate function to extract the individual letters from the image. Then, another function used the pre-trained CNN model to predict the entire word based on those extracted letters.
3. **Building the RNN Dataset:** Crucially, both the CNN's predicted word for the image and the true label from the CSV file were captured. This pairing of predicted and true information creates the foundation for training the RNN. The RNN will learn from these pairings, aiming to improve upon the CNN's predictions by incorporating the sequential nature of the handwritten text.

Implementation of the RNN

Now that we have a dataset specifically designed for the RNN, let's talk about how the RNN itself works. This dataset pairs the CNN's predicted words (for each image) with the corresponding true labels from the original data. This pairing should allow the RNN to study the sequential nature of the text.

This is the exact process it follows:

1. **Loading Predictions and Labels:** The code first loads the pre-processed predictions from the CNN model and the true labels from a text file. These are then split into training and testing sets for the RNN.
2. **Character-Level Tokenization:** An important step involves tokenization. Here, the tokenizer breaks down both the predicted words and true labels into individual characters. This "char_level=True" setting ensures the RNN considers the order of characters within a word.
3. **Converting Text to Numbers:** The RNN struggles to directly process text data. So, the tokenizer converts the character sequences into numerical sequences. Each unique character is assigned a corresponding integer value.
4. **Padding Sequences:** Since handwritten words can vary in length, the code employs "padding." This adds special characters (padding) to shorter sequences, ensuring all sequences have the same length. This allows the RNN to process them efficiently.
5. **Converting Labels to Categories:** Finally, the true labels are converted into a categorical format. This format is more suitable for the RNN's loss function used during training.

By following these steps, the RNN will analyze the sequential relationships between characters in the predicted words, aiming to improve upon the CNN's initial predictions by incorporating the context of the entire word or phrase.

This code is the process of many trials and investigations, at the end, it seemed to be the one that worked the best, but as you will now see, it was not satisfactory.

7. Results and Discussion

Unfortunately, the handwriting recognized text after passing through the RNN was worst than before, the names made less sense. But why is this? well the RNN is trained with a database with really low similarity or accuracy, which makes it really hard for it to learn how to correct names or any other text. The idea is that, since the data to train is so bad, it wasn’t able to actually learn enough to be able to correct, so the corrections are almost random.

Nevertheless, we strongly believe that with a good dataset to train from, like a dictionary of words for example, it would have been able to learn patterns and words, to be able to implement later on that sequentiality learned.

Conclusion

We obviously can't say we are satisfied, since the results weren't the ones we were waiting for. Regardless, we want to start this conclusion by talking about the things we did right and that actually improved the base model we took from the internet. For starters, the segmentation, which is a key part of the HTR problem, has improved a lot thanks to the various changes we have done to it. Also, the accuracy and loss of the predictions that the CNN made have also improved a lot, and that is thanks to the new CNN we created and the dataset cleaning we did, we can now say, we have created a cleaner and more robust model that will be able to recognise handwriting better.

That said, it is time to talk about the RNN. This has honestly been a disappointment, since we have worked so hard on this project but wanted to finish it with the cherry on top of combining different knowledge obtained from this course in a way that actually made sense. Although we have realized our idea wasn't new, we are still proud of having been able to reach this kind of understanding of the subject that has taken us to the idea of convincing RNN and CNN for HTR.

We also want to take this RNN failure as an opportunity to learn new things about RNNs and HTR and actually contribute to the investigations in a small way stating that, adding an RNN to a CNN to better solve the HTR problem is a solution that works and makes a lot of sense only if we have a clean, meaningful and big dataset. On the other hand, this model doesn't improve a thing when the data is really bad or the CNN isn't working great. An RNN simply needs a lot of information when dealing with words because there is a lot of variability and no meaningful patterns or learnings can be extracted from a poor dataset. We believe RNN is great to solve small errors of (for example) an "e" and a "c", or an "i" and an "l"... when this are sort of the only errors on the word, but when the errors are deeper, the RNN isn't the model to go to to solve the problem.

Finally, although this project has been very time consuming and even frustrating at times, we have enjoyed it and learned a lot. We aimed high regardless of the fact that we are only two members on the team, but we still think we did a good project.

References

Wang, Y., Xiao, W., Li, S.: Offline handwritten text recognition using deep learning: A review. In: Journal of Physics: Conference Series, IOP Publishing, p 012015 (2021)

Grygoriev, A. et al. (2021). HCRNN: A Novel Architecture for Fast Online Handwritten Stroke Classification. In: Lladós, J., Lopresti, D., Uchida, S. (eds) Document Analysis and Recognition – ICDAR 2021. ICDAR 2021. Lecture Notes in Computer Science(), vol 12822. Springer, Cham. https://doi.org/10.1007/978-3-030-86331-9_13

Abdurahman, F., Sisay, E. & Fante, K.A. AHWR-Net: offline handwritten amharic word recognition using convolutional recurrent neural network. *SN Appl. Sci.* **3**, 760 (2021). https://doi.org/10.1007/s42452-021-04742-x

Shashank, B.N., Nagesh Bhattu, S., Sri Phani Krishna, K. (2023). Improvising the CNN Feature Maps Through Integration of Channel Attention for Handwritten Text Recognition. In: Gupta, D., Bhurchandi, K., Murala, S., Raman, B., Kumar, S. (eds) Computer Vision and Image Processing. CVIP 2022. Communications in Computer and Information Science, vol 1777. Springer, Cham. https://doi.org/10.1007/978-3-031-31417-9_37

Bluche, T., Ney, H., & Kermorvant, C. (2014). Feature extraction with convolutional neural networks for handwritten word recognition. In 11th IAPR International Workshop on Document Analysis Systems (DAS).

Bluche, T., Messina, R., & Kermorvant, C. (2017). Dropout improves Recurrent Neural Networks for Handwriting Recognition. In 14th IAPR International Conference on Document Analysis and Recognition (ICDAR).

Bluche, T., Louradour, J., & Knibbe, F. (2018). Gated Convolutional Recurrent Neural Networks for Multilingual Handwriting Recognition. In 15th IAPR International Conference on Document Analysis and Recognition (ICDAR).

https://ictactjournals.in/paper/IJSC_Vol_12_Iss_1_Paper_1_2457_2463.pdf