CRNN for text reading*

*A method for quick high accuracy image-to-text model

1th Iago Casalderrey Casalderrey
Intercontinental business university
Poio, Pontevedra
iago.casalderrey.01@uie.edu

***Abstract*—Convolutional neural networks did a great job at recognizing digits. However, they did not do well at recognizing real text.**

This paper proposes CRNN, an architecture that uses both convolutional layers and recurrent layers. With that, it can read recursively several digits. In addiction, the recurrent layers help it to recognize words.

***Index Terms*—CRNN, OCR, hand-written recognition, image reading, image-to-text**

I. Introduction

Optical Character Recognition (OCR) is a compelling field of research with broad practical applications. It enables the extraction of text from images, addressing challenges posed by handwritten documents or text embedded in visual formats. OCR technology has diverse uses, from digitizing printed materials to enabling accessibility features, such as reading text aloud for visually impaired users.

This paper explores OCR through a multi-stage approach. First, a synthetic dataset was generated using Arial font characters. To enhance robustness, the dataset was augmented using TensorFlow's `ImageDataGenerator`, applying random rotations, zooms, shears, and brightness adjustments. A neural network was then trained on this dataset, employing **Tabu Search**—a metaheuristic optimization algorithm—to fine-tune hyperparameters. The model architecture included a convolutional layer $(64, 128, 3)$ with padding, optimized using the **Adam** optimizer, achieving **98% accuracy** on the test set.

Subsequently, a second dataset was constructed, consisting of **whole words** (*NARANJA, LIMÓN, CIRUELA, MANZANA*, and *PERA*) rather than isolated characters. Preprocessing involved OpenCV-based cropping before applying similar augmentations. A **Convolutional Recurrent Neural Network (CRNN)** was then implemented, utilizing **Connectionist Temporal Classification (CTC) loss** to handle sequential text recognition. The model demonstrated strong performance, with a test loss of **0.0742** and an accuracy of 99.20% in my whole dataset. Trying it in the test dataset would be required in the future, in the same way I calculated the loss.[2]

Identify applicable funding agency here. If none, delete this.

To further improve accuracy, **transfer learning** was applied by fine-tuning one layer of the CRNN using weights from the single-character model. Finally, the system was enhanced with **text-to-speech functionality** for accessibility and compared against a **Hugging Face OCR** model for benchmarking.

This project integrates concepts from multiple disciplines, including advanced machine learning, natural language procesing, intelligent systems, and computer vision. The following sections detail the methodology, experiments, and results in greater depth.

II. The dataset

A. Getting the letters

For our proposes we use a .ttf file. This file can draw Arial letters in an image format. We calculate an approximate image width based on the ammount of letters drawn and the font size. The size of the letters is a number between 50 and 70.

B. Cropping images to avoid too much blank spaces

I used Opencv class knowledge to transform an image into integers. The algorithm is specific but effective to our context. It binarizes the images and gets the highest y, x and the lowest, y and x that is considered black enough. Then it makes a square, so that the image has not too much white fonts, which will help the ctc loss to train quicker. It crops the image and finally we have a complete image

C. Image Preprocesing

ImageDataGenerator from keras or tensorflow adds noise by rotating images, adding zoom and changing shear and brighness randomly. This helps to make a more resilient algorithm.

Fig. 1. Example of dataset result. "CIRUELA" word

Fig. 2. First neural network "Z" letter

III. The single-letter recognizer algorithm

A. Introduction

This algorithm is able to extract key characteristic from the dataset to recognize letters. This is crucial, as some convolutional layers could be used later to train the convolutional-recurrent model faster.

B. Image Preprocesing

Some first changes are needed so that the neural network can process the images in the desired format with RELU activation. Some tensors with values between -1 and 1. First we map the letters into integer numbers, we resize the images, transform the images into grayscale, transform them into tensors, and normalize the values between -1 and 1. This is a good format for RELU activation.

C. Train-test split

Its important that we divide our values between train and test using sklearn train-test split. We also need to put our x and y into the torch Dataloader, so they can be easily processed by torch

D. The neural network

The neural network first has a convolutional layer

- First it uses a convolutional layer that transforms a 1x128x128 tensor into a 32x128x128 tensor, extracting key characteristics of the letters
- It normalizes the values using RELU activation so they are in the correct format
- It uses max pooling to reduce the size of the image from 32x128x128 into 32x64x64. to speed up the process and prevent overfitting. It uses a technique we saw in class that extracts the maximun values from the integer's black and white image.
- Then it uses another convolutional layer that extracts key characteristics and transforms the image into a 64x64x64 tensor.
- It uses another RELU activation.
- It uses max pooling to reduce the size of the image again. So we got a size of 128x32x32, getting some key characteristics and blurred image.
- Add RELU activation
- Max Pooling to reduce the size of the image into 128x16x16

Then the neural network uses a linear layer to process the key characteristics that were extacted, using the flattened characteristics:

- It flattenes the characteristics into 128*16*16, and process 256 characteristics
- RELU activation to normilize the values
- Another linear layer that uses the 256 characteristics to get the class number of each letter

E. Tabu search algorithm

The tabu search algorithm is a meta-heuristic algorithm. The algorithm is useful to optimize parameters in neural networks. It uses short-term memory to remember the places it went recently and doesn't have to go again, long-term memory, to learn how many times it went to some places and their results and an aspiration criteria to override the tabu status of a move. We use to find an optimal learning rate, that makes learning quicker. Surprisingly, the result improved when learning rate was very low. This also could be a fascinating area of research.

F. Testing the model

We test our model in the tst dataset. With that, we reach a 99% accuracy with some occasional mistake:

...	B	1.00	1.00	1.00	41
	C	1.00	1.00	1.00	45
	D	1.00	1.00	1.00	45
	E	1.00	1.00	1.00	38
	F	1.00	0.97	0.99	37
	G	1.00	1.00	1.00	38
	H	1.00	1.00	1.00	45
	I	1.00	1.00	1.00	41
	J	1.00	1.00	1.00	37
	K	1.00	1.00	1.00	26
	L	1.00	1.00	1.00	39
	M	0.97	1.00	0.99	37
	N	0.98	0.98	0.98	42
	O	0.98	1.00	0.99	40
	P	0.97	1.00	0.99	37
	Q	1.00	0.98	0.99	43
	R	1.00	1.00	1.00	40
	S	1.00	1.00	1.00	26
	T	1.00	0.98	0.99	48
	U	1.00	1.00	1.00	34
	V	1.00	1.00	1.00	49
	W	1.00	1.00	1.00	45
	X	1.00	1.00	1.00	36
	Y	0.98	1.00	0.99	44
	Z	1.00	1.00	1.00	37
	accuracy			1.00	1040
	macro avg	1.00	1.00	1.00	1040
	weighted avg	1.00	1.00	1.00	1040

Fig. 3. Accuracy results of the first network

G. Saving the model

We save our model. This is important to use one convolutional layer later with transfer learning.

IV. CRNN MODEL

A. Introduction

This model will be able to read words. The recurrent model is crucial because it can identify expected error from the involutional network. For example, the convolutional network may recognize words that doesn't exist, because it doesn't have any natural language understanding. It uses long-short term memory to remember past letters.

B. Preprocessing

It has a very similar structure from the last time. It resizes the images, transforms the images into black and white so they can be converted into integers easily. It also transforms the output letters into a list of integers that can be interpreted by the maths. However, it required a more fine-tuned processing of the loss function, because the CTC loss is more complex to do that cross entropy loss.

The input of the loss function. If we consider:

It has a very similar structure to the last time. It resizes the images, transforms them into black and white so they can be converted into integers easily. It also transforms the output letters into a list of integers that can be interpreted by the model. However, it requires a more fine-tuned processing of the loss function because CTC (Connectionist Temporal Classification) loss is more complex than cross-entropy loss. It's important to add the blanck token as a possible output, so the model can recognize blank or transition spaces. For example if an image is something between an A and a B, the model may output a blank space.

C. Inputs and Outputs for CTC Loss in CRNN Word Recognition

The CTC loss function is critical for training CRNNs on variable-length word sequences. Below, we define the exact tensor shapes and their roles in the context of a CRNN processing word images (e.g., scanned documents or street signs).[1]

Variable Definitions::

- T: Number of timesteps (aligned with the width of the feature map after CNN + RNN).
  Example: For a 100-pixel-wide image, $T = 25$ after 4x downsampling by the CNN.
- N: Batch size (e.g., 32 images processed simultaneously).
- C: Number of character classes + 1 (blank token for CTC).
  Example: For alphanumeric text, $C = 36 + 1 = 37$.
- S: Length of the longest word in the batch (used for padding targets).
  Example: Batch `["CIRUELA", "MANZANA", "PERA"]` $\rightarrow S = 7$.

Inputs::

- **`outputs`** (CRNN Predictions): A tensor of shape (T, N, C):
  - Logits for each character class at each timestep.
    Structure: [Timesteps × Batch × Classes].
- **`targets`** (Ground Truth Labels): A tensor of shape (N, S):
  - Integer-encoded words, padded to length S.
    Example: "CAT" → [3, 1, 20, -1] (padded with -1).
- **`input_lengths`**: A tensor of shape $(N,)$ where all entries are T.
  Implies all images in the batch have the same width after processing.
- **`target_lengths`**: A tensor of shape $(N,)$ with actual word lengths.
  Example: For `["CAT", "DOG", "BIRD"]`, lengths = $[3, 3, 4]$.

Output::

- **`loss`**: A scalar tensor.
  Example: `loss = 0.24` (CTC loss averaged over the batch).

CRNN-Specific Notes::

- **Timesteps vs. Letters**: T (CNN/RNN output steps) and S (letters in word) are independent.
  CTC bridges them via dynamic alignment.
- **Padding**: Targets are padded to S, but `target_lengths` ensures padding is ignored.
- **Blank Token**: Class index $C-1$ is reserved for CTC's blank symbol (critical for repeats/alignment).

D. The network

Convolutional Layers (Feature Extraction):

- First Conv Layer: Transforms input from '1x224x224' to '64x224x224' using 3x3 kernels with padding=1
- **ReLU Activation:** Applies non-linearity while maintaining spatial dimensions ('64x224x224')
- **Max Pooling:** Reduces to '64x112x112' with 2x2 window (stride=2)
- **Second Conv Layer:** Increases depth to 128 channels ('64x112x112' → '128x112x112')
- **ReLU + Max Pooling:** Reduces to final CNN output of '128x56x56'

Recurrent Layer (Sequence Processing): Feature Preparation for LSTM:

- Bidirectional LSTM (2 layers, 128 units): Input reshape: '128x56' → '(batch, 7168, 128)'

Linear Layer (Classification):

- FC Layer: 256 → number of classes
- Final output: (batch, 56, number of classes)

E. Fine Tunning

The fine tunning was quite simple. Simply used the stored weights and bisases from a shared layer. In out case, the third layer from the simple network, was the same than the second layer from the CRNN network, being a tensor with shape (64, 128, 3) and padding = 1

```
state_dict = torch.load("letter_classifier.pth", map_location="cpu")

model.cnn[3].weight.data = state_dict['conv_layers.6.weight'].clone()
model.cnn[3].bias.data   = state_dict['conv_layers.6.bias'].clone()
```

Fig. 4. The simple code to fine tune

F. Testing the model

The model got an accuracy of 99.2% in the full dataset and a loss of 0.0742 in the test dataset. The reason why I didn't do everything in the test is because of incompatibility issues while trying to add new libraries that are not included in this paper, however, trying to get the accuracy in the test dataset would be an interesting research project.

G. Storing the model

We store the model in a .pth file. Key for future uses or developments with fine tunning or to use our model in the future.

H. Text to audio model

With the text we get from the sample image we can transform text to speech We used gtts, a local python library that transforms the output text into audio format. Then we use AudioSegment from pydub, to say the video outloud. This library can execute the file several times, which is very useful to execute our python file several times.

I. Comparing with the hugging face model

The OCR hugging face model reached a 96.40% of accuracy in the whole dataset. That's very good for a computer, but a human could do better. In fact, our model, with 99,2% of accuracy, but overfitting in 80% of images, mightdo better

REFERENCES

[1] Paul Viola and Michael Jones. "Rapid Object Detection using a Boosted Cascade of Simple Features". In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2001, pp. 511–518. DOI: 10 . 1145 / 1143844 . 1143891. URL: https://dl.acm.org/doi/10.1145/1143844.1143891.

[2] Yuqing Xie et al. "An interpretable artificial intelligence system for the histopathological diagnosis of gastric cancer". In: *Scientific Reports* 14.1 (2024), p. 8326. DOI: 10.1038/s41598-024-67738-8. URL: https://doi.org/10.1038/s41598-024-67738-8.