DMT – Homework3

Vigèr Durand Azimedem Tsafack(1792126)
Malick Alexandre N. Sarr (1788832)

✓ **First Step**: The Pipeline

1. Dataset description

The non-augmented Dataset is a set of **2622** images as **64x64** numpy arrays. Each image represents one of the 94 possible characters in one of the 28 possible fonts. 10 characters are not supported by certain fonts.

After performing data augmentation, we get a new dataset with **55062** images:

	Char	**Font**	**Bold**	**Italic**
	560 images per Char	**1880** images per font	**94x20 = 1880** char per bold font	**94x20 = 1880** char per italic font
TOTALS	**560x94 = 52640** (actually ***55062*** because ***10*** images are not supported) images for all the characters	**1880x28 = 52640** (actually **55062** because **10** images are not supported) images for all the fonts	**1880x12= 22560** bold images	**1880x10 = 18800** italics images.

Our model will deal with scanned images that represent characters. Since the input image to classify will rarely be perfectly defined, we are going to perform some transformation on the pictures initially generated trying to mimic real world deformations:

- Random rotations in the range of **-29°** AND **29°** to capture the case when the paper inclined while inserting it in the scanner.
- the way the images are generated makes the characters stand in the center of the pictures, so we are going to randomly translate them in the range of **-15** and **15** pixels among the image. Since the character could be wherever on the input picture.
- It is also necessary to apply a random zoom in or zoom out from **-0.5** to **0.5**
- Each image is randomly augmented **20** times

2. Model description

Before choosing our best model, we first checked the various known architectures that did well in the past for image recognition (check references). As result, we have tried to train up to **7-8** different models, and the best result we achieved was a partial accuracy of **.57** with **char = .57**, **font = .35**, **bold = .78** and **italic =0 .69**. On our own test data, we had **char= .86**, **font = .67**, **bold = .91** and **italic =0 .79**. The less performant models had a partial accuracy ranging from .31 to .53.

The model that worked the best was a combination of a ***convolutional neural network*** and a ***recurrent neural network*** which was presented in the Keras documentation. The convolution stage has two ***convolutional layers***, each of them followed by a ***maxpooling layer***. On those two ***convolutional layers***, we applied **16 3x3** filters, using a ***'relu'*** activation function. The pooling size was **2**. During the convolutional stage, the model extracts various features (edge detection etc., see extra points 1) from the image to prepare the recurrent stage.

The recurrent stage is made of two successive ***bidirectional Gated Recurrent Units***. One of the advantages of using this method is that it keeps short, long-term memory of the data sampled as it passes through the various layers. Each ***GRU*** were given **512** units. We used some auxiliary layers as well to reshape the convoluted data into a recurrent format, and then merge the two bidirectional blocks. We used ***addition*** method on the first one and ***concatenation*** on the second one. We closed the model by adding ***a fully connected layer*** using a ***softmax*** activation function to generate the various outputs. During the model compilation, ***categorical crossentropy*** was used to identify font and character, and ***binary crossentropy*** was used to identify bold or italic.

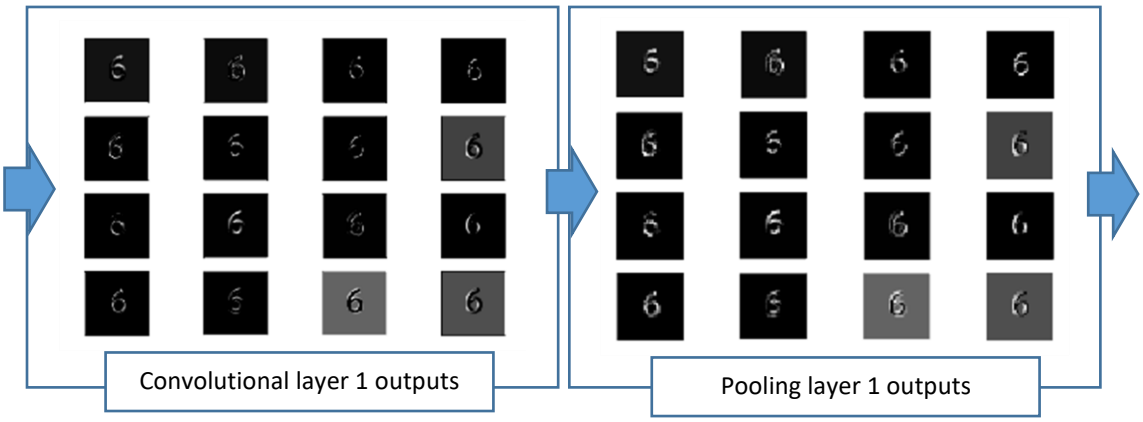
One reason this model work better than the other, might be the fact that there is a lot of written text in the test data (bank scanned document) and since letters in the text appear in a sequential order, this justify the use of an ***RNN*** in the text recognition. One downside of this approach is the fact that we could not train the data on a large epoch set on the cluster since the training cost is really height. Indeed, as opposed to the other pure convolutional model we used, our best performing one took approximately **3** min to render only one epoch, so we might expect better improvement if more epochs could have been used during training.

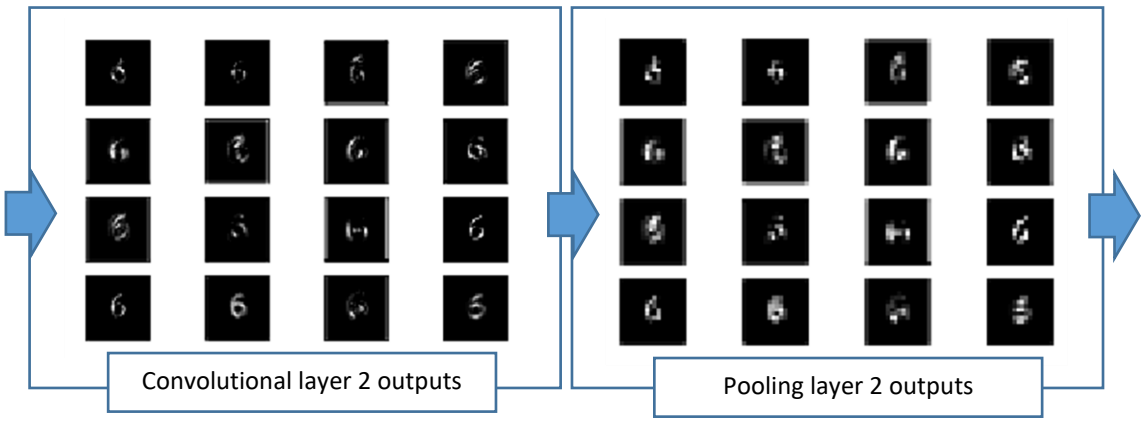
Our second best performing model was made of **3** blocks of **2** ***convolutional layers*** each of them followed by a ***pooling layer*** and a ***dropout layer***. The filters increased from **32** to **128** and was training on **100** epochs until the loss function was fully minimized. That model achieved its full potential unlike our better performing one. We also had implemented ***AlexNet*** and ***ResNet***. They had a very good accuracy on the training data, however they both did quiet poorly on the testing data; **.33** and **.29** as partial accuracy respectively which might be due to overfitting the training data.

✓ **Second Step**: Extra points

1. Part 2.1

The goal of this part is to visualize the intermediate convolutional and pooling layers of our model to show off what is going on in the network. Our model here is made of **14** intermediate layers (excluding input and output layers). Among which we have only **4** convolutional layers (***Convolution2D*** and ***MaxPooling2D***). Hence, we will limit ourselves in displaying and interpreting the outputs of only these **4** layers since they are the only ones that could easily be transformed into images:





In the convolutional layer, each filter scans the picture and learn some particular aspects of it (e.g. Edge detection). The outputs in our case is a set of **16 64x64** images. The original shapes of pictures are kept after convolution because we used ***padding='same'***.

In the ***MaxPooling2D*** layer, we basically reduce the dimension of the input picture by choosing the max value in each **2x2** matrix in the image. This is done to in part to avoid over-fitting by providing an abstracted form of the representation. As well, it reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation. As we see on the images, after applying a pooling function, we lose image definition which could seem useless but prevent us for overfitting.

REFERENCES:

- https://medium.com/@siddharthdas_32104/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5
- https://arxiv.org/pdf/1512.03385v1.pdf
- https://hackernoon.com/latest-deep-learning-ocr-with-keras-and-supervisely-in-15-minutes-34aecd630ed8
- https://medium.com/@14prakash/understanding-and-implementing-architectures-of-resnet-and-resnext-for-state-of-the-art-image-cf51669e1624