Computer Vision and Image Processing
M

Report by Giada Martina Stivala - 0000820941

Optical Character Recognition with Neural Networks

Introduction

In this project we developed a system to address the problem of CAPTCHA recognition. Most web services use CAPTCHAs to prevent automatic navigation by robots and crawlers: an image with a series of letters and numbers are shown to the user, who has to recognize them and write the corresponding characters in a text box. CAPTCHAs can vary in complexity, showing more or less letters, adding shifts and rotations as well as noise and additional lines and points overlapping with the letters.

For our purposes, we chose to study and process only simple CAPTCHAs, characterized by a low number of letters and digits, little degree of overlapping and occlusion and little noise. Also, all of the letters are written with a capital font and similar dimension, with the only difference being the bold character. This choice reflected also the limited availability of hardware and computing power at our disposal.

Initially, the target of this project work was to perform character recognition entirely with a neural network, trained on a large number of labeled instances retrieved from the web. The task proved though to be too complex for the available resources, thus imposing a simplification of the task.

Consequently, the complexity of CAPTCHAs was lowered, and the entire

project structure was reshaped into three different parts. The idea behind this approach was to leverage on the OpenCV library, freely available, to preprocess CAPTCHA images so to decrease the workload of the neural network. We also considered to tackle the recognition task with a *divide et impera* approach, purposefully we extracted from a single CAPTCHA image four sub-images containing the letters by themselves, and the network was trained on single labeled letters. Considering the learning process required to recognize a whole CAPTCHA image with respect to the recognition of single characters, this simplistic hypothesis turns out to be quite efficient.

In the first part, CAPTCHAs are loaded into memory and divided into their four characters. Then, they are processed and organized in a set for the neural network to use. In the second part, we build the neural network model and complete the training; finally, a CAPTCHA image is randomly picked and fed into the network for recognition. As the network recognizes only alphanumeric characters, images are previously processed to meet this requirement.

First task - image processing

In this section we address the image preprocessing aimed at simplifying network training.

First, we draw a sample from the entire data set, and save it for later. This is to test the network's predictive ability on unseen examples. To do it, we use the Keras function `train_test_split`, which automatically draws a random sample from the provided data set. The size of the subset is 0.15 times the size of the entire dataset, which counts 9955 labeled CAPTCHAs. We then proceed to load the image and convert it into grayscale. CAPTCHA images may be RGB, but colour does not provide any additional information in terms of character recognition, thus we prefer to remove it to simplify the learning process.

To apply the *divide et impera* approach it is necessary to split the image in its four letters, obtaining four sub-images. Thus, we need to identify the contours of the characters, and then separate them into four different images using their bounding box. During this step, the main issue consisted

in separating overlapped characters: in fact, the bounding box happened to contain two characters at the same time as they were detected as a single object. The problem was addressed with a simple solution: if the width of the bounding rectangle was greater than the height (we thresholded their ratio), the rectangle was divided in two equal halves, each assigned to a character. As a result, some characters appear to be missing edges or small pieces, but this was not considered to be a flaw as it could help the network generalize better.

- Since we chose fairly simple CAPTCHAs with a low amount of noise, we can assume that after turning the image into grayscale the histogram is already bimodal.
- We can proceed with thresholding, leveraging in particular on Otsu's automatic thresholding algorithm. In the OpenCV `threshold` function, we select `THRESH_BINARY_INV` and `TRESH_OTSU`, obtaining the negative of the CAPTCHA.
  - Objects are assumed to be bright on a dark background.
- We invoke `findContours`, selecting only the highest level of the contours hierarchy (the most external contour, with the parameter `RETR_EXTERNAL`); furthermore, since we have to use them to construct the bounding rectangle, we select only the end points with `CHAIN_APPROX_SIMPLE`.
- We feed the obtained contour into the `boundingRect` function, obtaining a quadruple of points, the first two being the top-left corner, followed by width and height of the rectangle. The result is shown in Figure 1.
- To split the original image into four sub-images, first we check if the obtained rectangle has valid values, possibly splitting the bounding rectangle in two separate rectangles if two letters have been recognized together. We also checked if, by any chance, the `findContours` function recognized more than four objects, discarding the whole CAPTCHA if this was the case.
- Finally, we applied the bounding rectangles to the original grayscaled image, which was then cut in four sub-images following those guidelines. We can see single letters and digits in Figure 2.

Figure 1: CAPTCHAs with letters and numbers highlighted by a bounding box.

Figure 2: Letters after being cut from whole CAPTCHA images. It is possible to observe as some of them are skewed and missing little angles, or with additional object pieces coming from previously overlapped letters.

Letters were separated from digits and saved separately, to train the network on separate classes.

Second task - neural network

In this section we discuss the architecture and performance of the neural network.

At the beginning of this stage we load the data we processed and add some final touches to start training. These include:

- Choosing a set dimension for all the images, which will then be a parameter for the neural network;
- Creating a `numpy` array with an added dimension;
- Retrieving labels for testing;
- Normalize all images, from interval `[0, 255]` to interval `[0,1]`.

An additional, bigger partitioning of the training set is performed to obtain the validation set, used to calculate accuracy and loss during training. Finally, we instantiate a `Sequential` model for our convolutional neural network, adding as a first layer a `Conv2D` layer applying 20 (5x5) kernels on the 20x20x1 input tensor; the image is padded to obtain an output with the same dimension. Then we have a dropout layer with the `MaxPooling2D` operation, which gives a 10x10x20 tensor as an output. These two layers

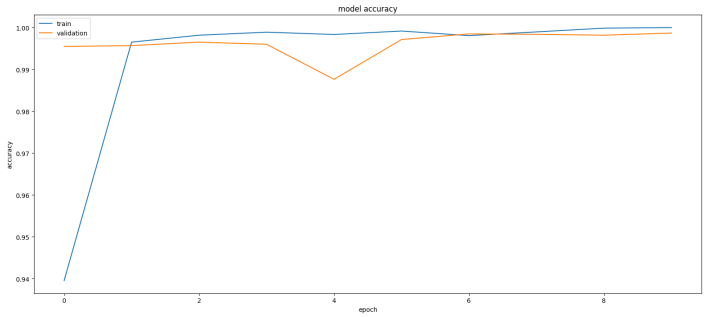

Figure 3: Graph comparing accuracy of the network performing on training set and validation set.

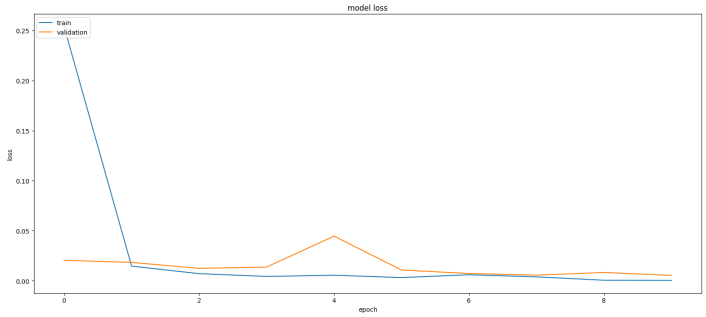

Figure 4: Graph comparing the loss of the network performing on training set and validation set.

are added another time to the network, and followed by a fully-connected layer with activation function `relu`. Ultimately, the output layer is another fully-connected layer, having 32 neurons and `softmax` activation function.

With this configuration, the network trained on 29058 examples, and tested on a validation set of 9686 examples. The network performed well as we can see from the accuracy and loss graphs in Figure 3 and 4.

To study the behaviour of the network and find some possible improvements, we tested two different aspects: first, we changed the structure of the network, adding more neurons in the hidden layer. Specifically, we moved from 300 to 500 neurons in the fully-connected layer, but the results showed a slight decrease of performance. In fact, even if the network seems to be slightly overfit (the accuracy on the training set being higher than the one on the validation set), adding more neurons strengthened this feature of the network, clearly indicating that we were moving in the wrong direction.

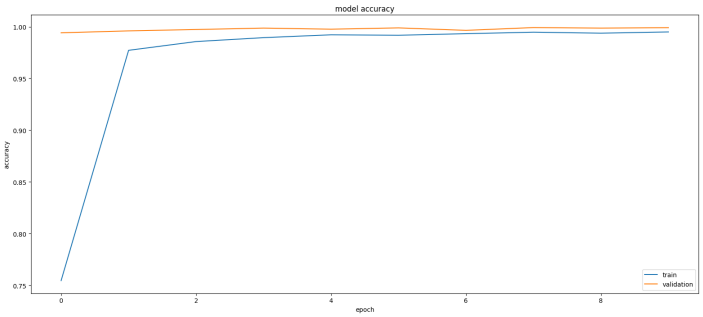

Figure 5: Graph comparing accuracy of the network performing on training set and validation set, after the addition of an ImageDataGenerator.

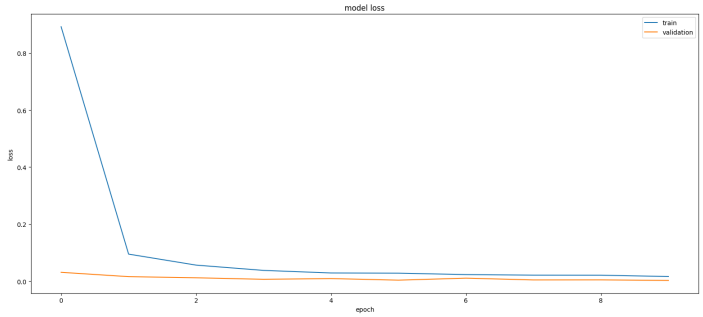

Figure 6: Graph comparing the loss of the network performing on training set and validation set, after the addition of an ImageDataGenerator.

Secondly, we decided to add some more image preprocessing aimed at introducing more variations in the data set. We used the `ImageDataGenerator` available from Keras to insert random shifts both horizontally and vertically, together with rotations. The shifts could be up to 0.2 of the image dimension, and the rotation up to 15 degrees.

This new model performed quite differently from the original one: the accuracy on the training set dropped considerably, while accuracy on the validation set improved, showing a greater ability of the network to generalize. The same can be said for the loss function, as shown in Figure 5 and 6. For testing purposes, we considered the performance of `ImageDataGenerator` also in the network with 500 hidden neurons. The results showed an improvement of performance, which was, nonetheless, poorer than the one obtained with 300 neurons.

We finally tested both networks on the unseen examples. They are 1490

Figure 7: Results of wrong predictions in unseen examples. We can notice, among them, some errors due to spurious objects in the image.

randomly-selected CAPTCHAs, with four characters each, for an overall amount of 5960 predictions. We observed the following results:

- The first network made 24 incorrect predictions on the whole dataset. Even if only a single letter was incorrect among the four composing a CAPTCHA, the entire prediction must be discarded.
- The second network made 29 incorrect predictions on the whole dataset. This goes against our previous considerations of a better performance.

In Figure 7 are shown some of the incorrectly labeled images. We analysed the incorrectly labeled CAPCHAs for both networks and found out that:

- For the first network, 13 over 24 incorrectly labeled characters were Q letters which had inside their bounding rectangle some spurious element, probably due to incorrect binarization or cutting.
- We run the same check on the second network and found out a lower number of errors due to Q incorrect labeling (11 over 29). A comparison of the two errors for this particular case shows a decrease of incorrect labeling from 54% to 37%. The remaining errors were "regular" wrong predictions with an even distribution.

Third task - predicting CAPTCHAs

The third step loads the unseen examples and the trained model, and calculates some statistics on performance. As the model predicts single letters, we repeated the process used in the first step to separate letters within CAPTCHAs.

Final considerations

We have presented a report for a project on Optical Character Recognition for the Computer Vision and Image Processing M university course. We have discussed possible different network architectures and their performance, evaluating them and identifying the best solution. We have also considered and employed ways to preprocess images, so as to simplify the training process and gain in efficiency.

This project makes use of OpenCV functions studied within the course as well as convolutional neural network training, which is not covered in the program. The choice of the topic is related to my own personal interest in Information Security, and I thought that "breaking CAPTCHAs" could be a project that fit both areas.

We focused on a simplified use case, in which the diversity between CAPTCHA characters is limited and there is little to no amount of added noise in the image. It could be a real use case only for web services with a low security risk, while CAPTCHAs employed in services as Amazon or Facebook are at another scale of difficulty.

Further developments and improvements could tackle more complex tasks; these involve using additional image preprocessing, to remove both noise and added lines or objects. Depending on the type of noise (usually both impulse and random noise), median and gaussian filtering can be employed. On the other side, to remove additional lines or objects we could use morphology transformations like opening and closing, available from the OpenCV library.