

# OCR - First Report

Gaspard Jolly, Robin Paulik, Luca Sarrubi, Tom L'Hotellier

November 4, 2023

# Contents

|  |           |
|--|-----------|
| <b>1 Group members and Organization</b>          | <b>4</b>  |
| 1.1 Tom L'Hotellier . . . . .                    | 4         |
| 1.2 Gaspard Jolly . . . . .                      | 4         |
| 1.3 Luca Sarubbi . . . . .                       | 4         |
| 1.4 Robin Paulik . . . . .                       | 4         |
| <b>2 Pre - Processing</b>                        | <b>5</b>  |
| 2.1 Introduction . . . . .                       | 5         |
| 2.2 Grayscale . . . . .                          | 5         |
| 2.3 Standardisation of lighting . . . . .        | 6         |
| 2.4 Noise Reduction (Median Filter) . . . . .    | 6         |
| 2.5 Reducing parasites (Medium Filter) . . . . . | 7         |
| 2.6 Adaptive Threshold (Binarisation) . . . . .  | 8         |
| 2.7 Smoothing . . . . .                          | 9         |
| 2.8 Inversion . . . . .                          | 10        |
| 2.9 Conclusion . . . . .                         | 10        |
| <b>3 Image manipulation</b>                      | <b>11</b> |
| 3.1 Introduction . . . . .                       | 11        |
| 3.2 Rotation . . . . .                           | 11        |
| <b>4 Grid Recognition</b>                        | <b>13</b> |
| 4.1 Edge Detection . . . . .                     | 13        |
| 4.1.1 Sobel Operator . . . . .                   | 13        |
| 4.1.2 Non-Maxima Removal . . . . .               | 14        |
| 4.1.3 Double Thresholding . . . . .              | 14        |
| 4.1.4 Hysteresis Analysis . . . . .              | 14        |
| 4.2 Hough Transform . . . . .                    | 15        |
| <b>5 Cell Division</b>                           | <b>17</b> |
| <b>6 Solver</b>                                  | <b>18</b> |
| 6.0.1 Backtracking . . . . .                     | 18        |
| <b>7 Character Recognition</b>                   | <b>19</b> |
| 7.1 Global Structure . . . . .                   | 19        |
| 7.1.1 Feedforward . . . . .                      | 19        |
| 7.1.2 BackPropagation . . . . .                  | 19        |

|       |                               |    |
|-------|-------------------------------|----|
| 7.2   | Image training . . . . .      | 19 |
| 7.2.1 | Inputs . . . . .              | 19 |
| 7.2.2 | Training process . . . . .    | 20 |
| 7.3   | Saving and Loading . . . . .  | 20 |
| 7.4   | Note on performance . . . . . | 21 |

# **1 Group members and Organization**

## **1.1 Tom L'Hotellier**

Before studying at EPITA, I never did anything related to code or programming. It was a real surprise for me to discover how interesting was programming and it became a pleasure really fast. I was a bit worried before the begining of the year because I heard that C was trickier to learn than C but overall, I am quite comfortable with C programming now and I find some new concepts such as memory management or multi-threading, very interesting to learn about. Very quickly from the beginning of this project, I became aware of how interesting it is and how, tasks that can seem simple, are in the end more complex than we think about - all the pre-processing procedure for instance.

## **1.2 Gaspard Jolly**

My name is Gaspard Jolly, in highschool when I choose to study math and digital sciences, I thought that I was just going to try those computer things that I saw on internet and maybe it would interest me. Since then I really liked programming, either when I developed a website with a friend in highschool or with the programming of the video game we did last year. This new project allows me to tackle areas that I haven't studied in the past and I've found image analysis very interesting, for example.

## **1.3 Luca Sarubbi**

Hi! I'm Luca Sarubbi. Some day hopefully a low-level software engineer. although it all started from enjoying Minecraft's game engine through custom Java plugins, I then got introduced to lower level languages such as C or Rust. My contribution to the project will mostly be on the AI in order to gain some experience as AIs are heavily used in robotics and IoT.

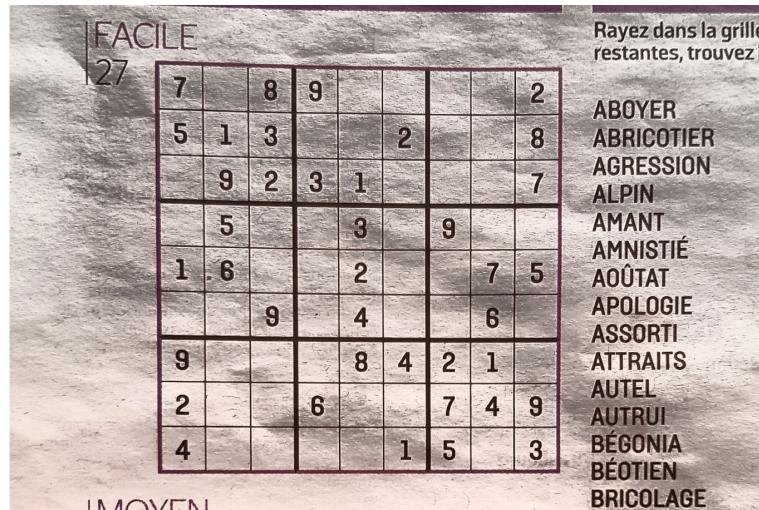
## **1.4 Robin Paulik**

Image processing is something that has intrigued me for some time. I just really like the simple idea of modifying images through formulas. However, I've never had the courage to dig deeply into it. I think this project is a good way to start and learn about ocr, that I will probably really like discovering. I also really enjoy puzzles and enigmas, so the idea of solving sudokus in this way is particularly appealing to me. This project will also greatly improve my skills in the C language.

## 2 Pre - Processing

### 2.1 Introduction

To begin with the treatment of the image, let us firstly see the image we are going to work with.



This image contains all the shapes of colors that the grid had at the moment of the picture. Like that we cannot detect the grid because of all the nuances. We are now going to use different filters so that we can get a grid as readable as possible. The vast majority of this work had been done using SDL, using a lot of his tools, like the "SDL\_Surface" , "SDL\_Format" or "SDL\_GetRGB()".

### 2.2 Grayscale

The first step is to change the image into a grayscale image, that is to say a picture that only contains shapes of gray.

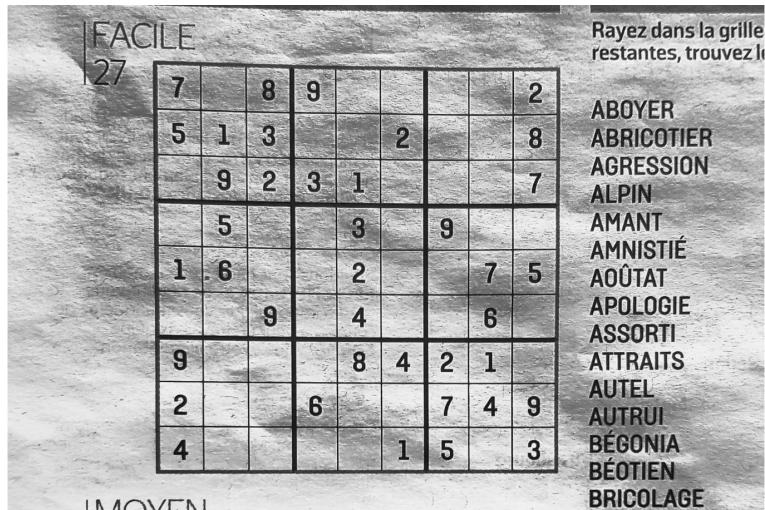
For that we are basically going to go through every single pixels of the image, taking the values of the RGB of each of the pixels using "SDL\_GetRGB()" and apply a basic formula:

$$\text{Red Shape} = \text{Red Shape} * 0.3$$

$$\text{Green Shape} = \text{Green Shape} * 0.59$$

$$\text{Blue Shape} = \text{Blue Shape} * 0.11$$

Here is the result after applying the grayscale filter:



## 2.3 Standardisation of lighting

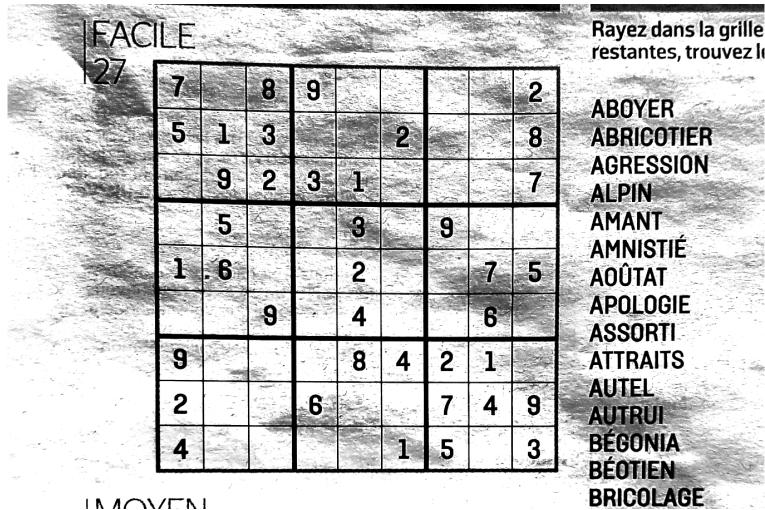
We are now using a contrast filter to help us better distinguish the writing from the distracting elements in the image.

To do this we define a factor, here we choose 2 for our example, we go through each of the pixels of the image and then we use the following formula on each of the RGB of the pixel we are currently on.

$$\text{value} = (\text{value} - 128) * \text{factor} + 128$$

Then we check if the value that we get is between 0 and 255, if not then we set the value to the nearest bound.

We get the following image:



## 2.4 Noise Reduction (Median Filter)

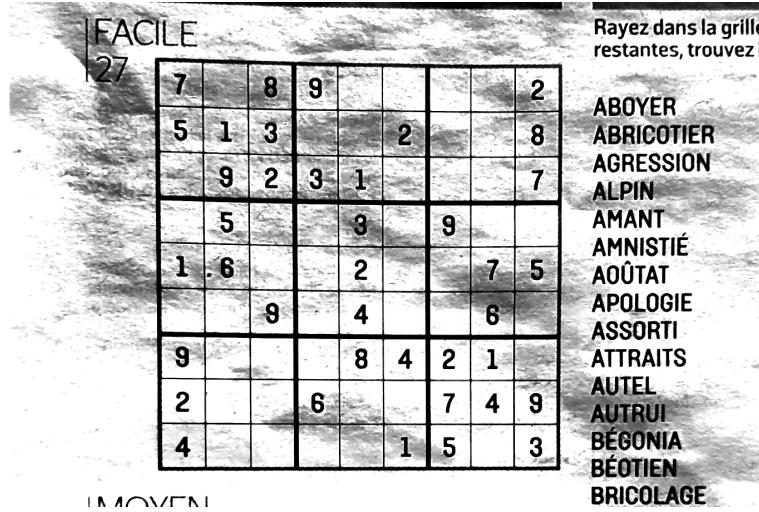
We are now going to get rid of the noise on the picture, which will give us a clearer image that we will be able to put into a threshold filter at the end of the process.

This filter is a bit different from the last one. We are still going through all the pixels, except the border ones. This can be explained because we are going to use a kernel on all the other pixels. A kernel is a matrix (here a 3x3 grid) that looks like this:

$$\frac{1}{9} \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

The kernel will take the shape of the pixel and his 8 neighbors and put them in an array. Then we just sort the array in the ascending order and take the median of the latest.

We then change the values of RGB of the pixel we are currently on to the values of the median we took. Here is the result:



This filter will be improved for the next defense. Indeed we are not taking into account the border pixels, to resolve this problem we will use a padding method so that those pixels will also be pasted in the median filter.

## 2.5 Reducing parasites (Medium Filter)

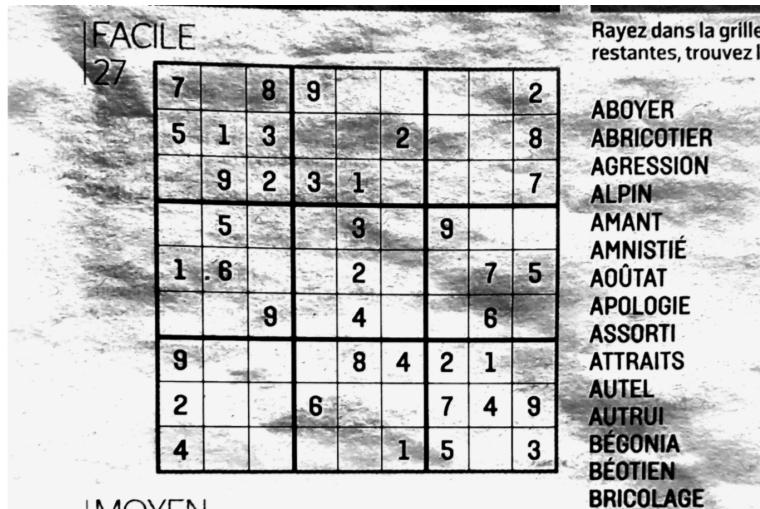
This step is used to blur the image and get rid of unnecessary details on the image that could complicate the detection of the grid.

As we did with the last filter, we are going to take a kernel, but this time we are going to

use the Gauss' blur, which means that we are going to use a particular kernel that will use the pascal triangle:

$$\frac{1}{16} \begin{matrix} & & \\ \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} & & \end{matrix}$$

We then get the next image:



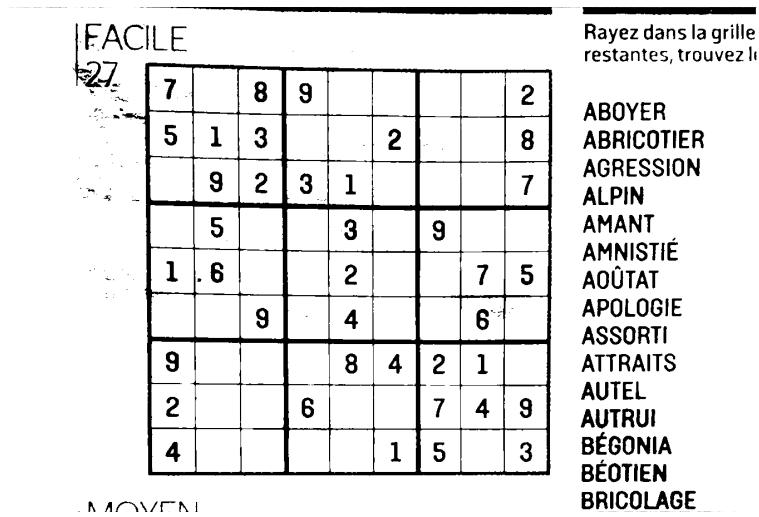
## 2.6 Adaptive Threshold (Binarisation)

For this part, we are going to pass the image into a filter which will reduce the shapes of the image to only black or white. For this purpose we are going to use the principle of the Otsu algorithm. We have to calculate a threshold for the image, in our example we are going to use a particular threshold that we have determined. For the next defense, we will have a function that calculate the threshold of the image that we put in the filter.

As we did with the latest filters, we are going to cut the images in subimages, using kernel. Then using the threshold that we found we are going to go through each of the pixels, except the ones on the borders, calculate the average of his neighbors and then use the following formula:

If (actual – average) > threshold => white pixel  
 Else => black pixel

Then we get the following image:

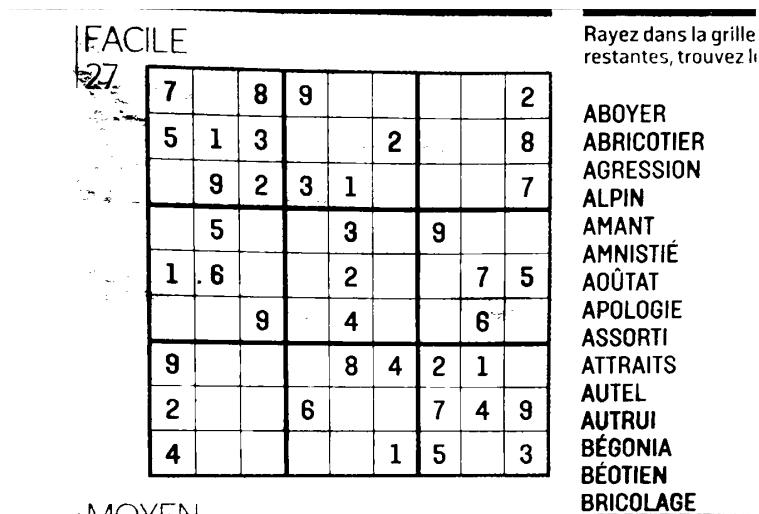


## 2.7 Smoothing

Now that we have a clean image with nothing more than white and black pixels, we are going to thinken up the lines to make it easier to detect the soduku lines.

For that, to begin with we create a new surface that is a copy of the current image, where we are going to change the pixels that need to be changed, because if we used the smoothing filter on the image without using a copy, we would get a totally black image. Then we go throught each of the pixels (again except the border ones) and if there is a black pixels in the neigboors of the current pixel we change this one into a black pixel (if it was not already) in the copied surface. Then we transfer the copied surface into the initial one.

We get the following result:

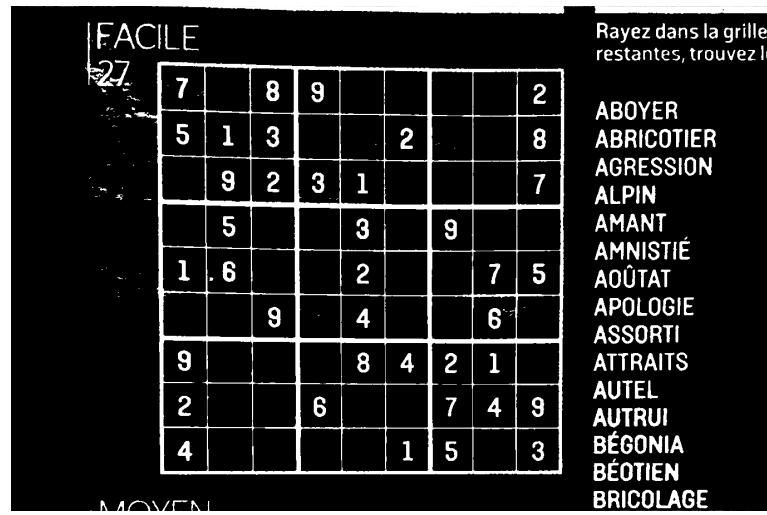


## 2.8 Inversion

Here we just basically inverse the values of each of the pixels on the image using this formula:

$$\text{value} = 255 - \text{value} \quad (1)$$

We finally get this image after using each of the filters



## 2.9 Conclusion

We hence get an image that is now usable for the grid detection as well as recognition with the neural network.

## 3 Image manipulation

### 3.1 Introduction

After pre-processing and before sending the grid to detection and splitting, there is still an important step to do: rotation.

Since the detection process needs the grid to be perfectly straight, rotation should work for any angle.

For this first defense, the rotation works with a user input in the command line with the following structure:

```
./rotate(imagefilepath)(angle)
```

When executed, this command will save the rotated image in a new file, in the 'Rotation' directory.

```
tom@tom-ThinkPad-P15v-Gen-2i:~/OCR/Rotation$ ./rotate sudoku_1.jpeg 28
```

### 3.2 Rotation

Let's now dive into the implementation of this rotation program.

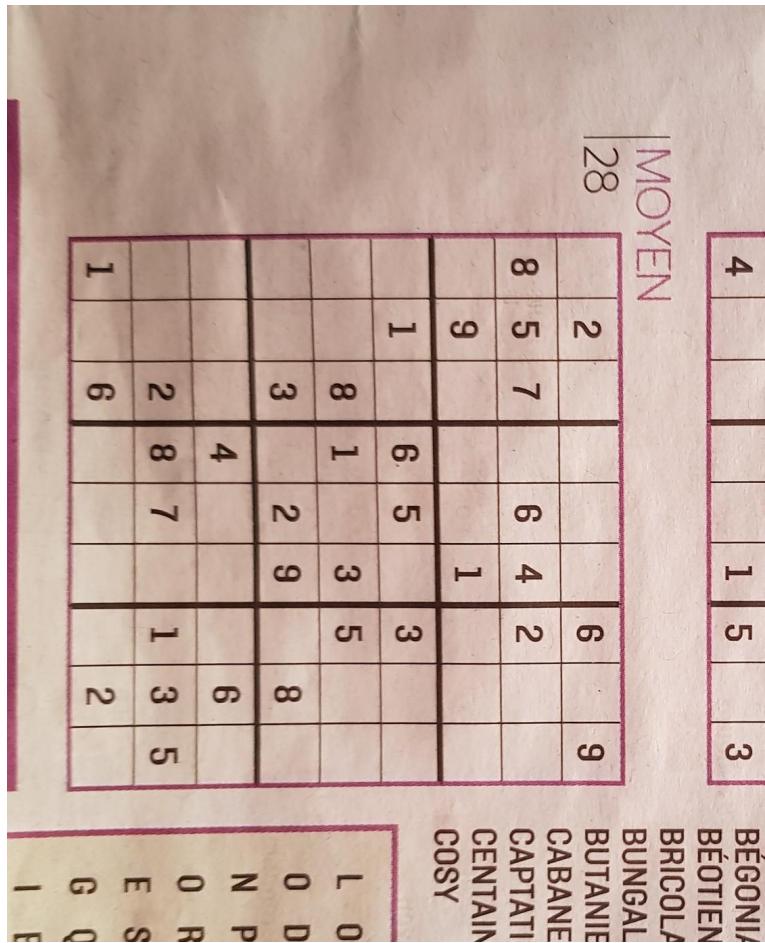
First of all, there are two very different methods to rotate the image.

The first one is used when the angle given by the user is a multiple of 90.

When that is the case, we call an auxiliary function called `rotation90()` the number of times equal to the input divided by 90. For example, if the input is 270, we call the function three times. This method is the simplest one because rotating an image by 90 degrees to the right can be done with matrix manipulation using the following formula.

$$\text{rotatedpixel}[y][\text{height} - 1 - x] = \text{pixel}[x][y] \quad (2)$$

The dimension of the new image have obviously been inverted before. This is the result for a 90 degree rotation.



The second one now, is way trickier and makes the program work for any angle.

Rotating an image by an angle that is not a multiple of 90 is not possible right from the hook, as you cannot save an image that is tilted. To tackle that problem, we hence had to modify the dimension of the new image in order to add some space for the image to rotate correctly and not to lose any pixels. Now for the actual rotation, we obviously had to use trigonometry. The formula was difficult to find but once we got it, it was a quite straightforward implementation.

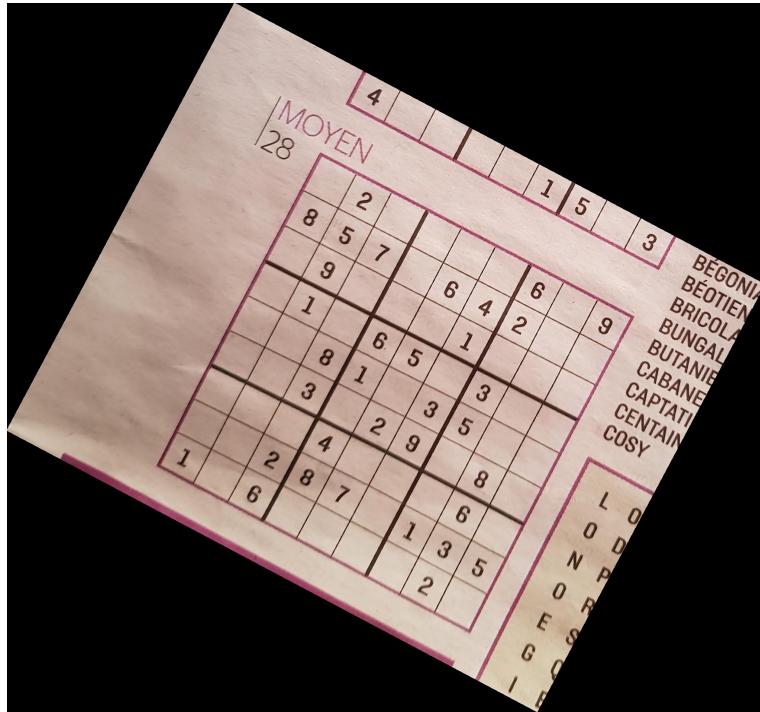
Here is the formula:

$$\text{sourceX} = \left( x - \frac{\text{width}}{2} \right) \cdot \cos(\text{angle}) - \left( y - \frac{\text{height}}{2} \right) \cdot \sin(\text{angle}) + \text{cX} \quad (3)$$

$$\text{sourceY} = \left( x - \frac{\text{width}}{2} \right) \cdot \sin(\text{angle}) + \left( y - \frac{\text{height}}{2} \right) \cdot \cos(\text{angle}) + \text{cY} \quad (4)$$

$$\text{rotatedpixel}[x][y] = \text{pixel}[\text{sourceY}][\text{sourceX}] \quad (5)$$

For instance, for a 27 degree rotation, we would get the following image:



## 4 Grid Recognition

### 4.1 Edge Detection

#### 4.1.1 Sobel Operator

The Sobel operator is used for edge detection in digital image processing. It involves convolution with two 3x3 kernels to calculate the gradient in the x and y directions. The Sobel operator helps identify edges in an image.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

X Kernel (X Gradient  $G_x$ )

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Y Kernel (Y Gradient  $G_y$ )

For each pixel, the gradient's norm will be as following :

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2} \quad (6)$$

And the gradient direction, that will be used for the next step :

$$\Theta = \tan(\mathbf{G}_y, \mathbf{G}_x) \quad (7)$$

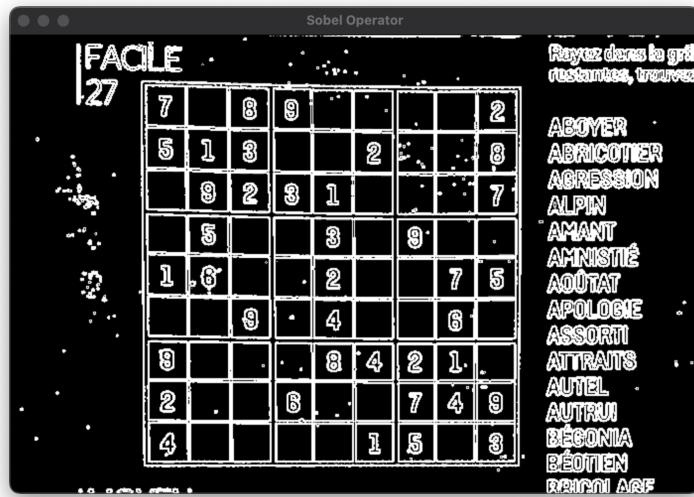


Figure 1: Image after Sobel Operator

#### 4.1.2 Non-Maxima Removal

After applying the Sobel operator, non-maxima suppression is used to thin the edges. This step ensures that only local maxima in the gradient direction are retained while the rest of the points are set to zero.

#### 4.1.3 Double Thresholding

Double thresholding is applied to the magnitude of the gradient. Pixels with a magnitude above a high threshold are considered strong edges, while those below a low threshold are considered non-edges. Pixels with a magnitude between the two thresholds are considered potential edges.

#### 4.1.4 Hysteresis Analysis

Hysteresis analysis is used to connect weak edge pixels to strong edge pixels. It helps to trace continuous edges by considering the connectivity between pixels. If a weak edge pixel is connected to a strong edge pixel, it is considered part of the edge; otherwise, it is removed. Once this step is done, the edge detection is fully done and it is now possible to go onto the next step.



Figure 2: Image after the Edge Detection Process

## 4.2 Hough Transform

The Hough transform is used to detect lines in an image. It creates an accumulator array of two dimensions,  $\rho$  and  $\theta$ , to find the most prominent lines. The Hough transform is commonly used in line detection, thus it was extremely adapted for the project.

The lines are expressed in polar coordinates and can be represented like this :

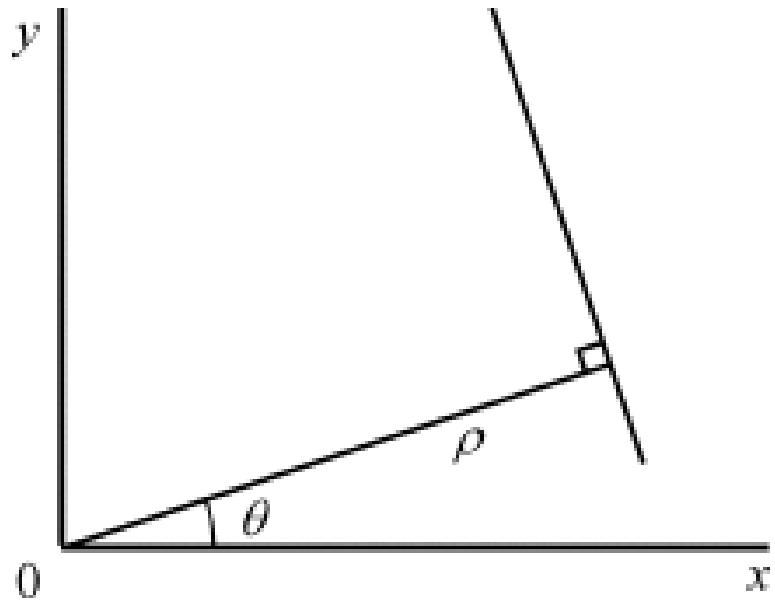


Figure 3: The line expressed with  $\rho$  and  $\theta$

Thus, since we are in polar coordinates, to help for our computations, we can express x and y in cartesian coordinates as functions of  $\rho$  and  $\theta$ :

$$x = \rho \cdot \cos(\theta)$$

$$y = \rho \cdot \sin(\theta)$$

We can also deduce the other related couples to trace our lines, with d a distance large enough , in this case the diagonal:

For vertical lines:

$$x_1 = x_0 + d \cdot \cos(\theta)$$

$$y_1 = y_0 + d \cdot \sin(\theta)$$

For horizontal lines:

$$x_2 = x_0 - d \cdot \cos(\theta)$$

$$y_2 = y_0 - d \cdot \sin(\theta)$$

Once we have our lines, we have to refine them to only keep ten of them, that are in fact the grid lines. To do this, we detect lines that are similar enough and we merge them by only keeping the average of these two lines. Once this process is done, we have this result and our grid is isolated:

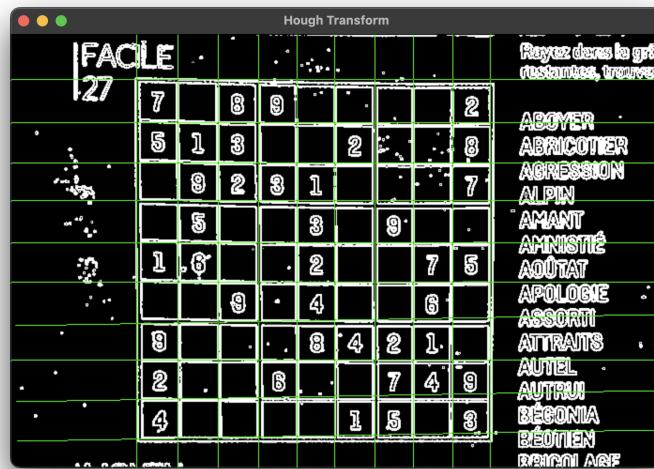


Figure 4: Image after Hough Transform

## 5 Cell Division

Once the lines are traced on the grid, we have to identify the intersection between those lines. We first detect the four vertices of the grid. Then we just extract the part of the surface where the grid is located and render it as a bitmap.

We then repeat the same process for each intersection, and as a result, we have our 81 grid cells rendered as bitmaps.

Each intersection is represented by a red pixel in figure 4. Every cell is rendered like that :

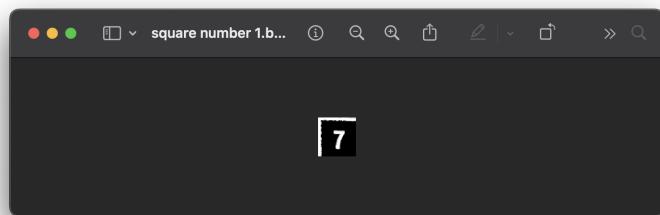


Figure 5: Example of a grid cell

Then, the last step is to remove the lines, with a simple function. If we detect a non black pixel in the edges, then we remove recursively every non black adjacent pixel.

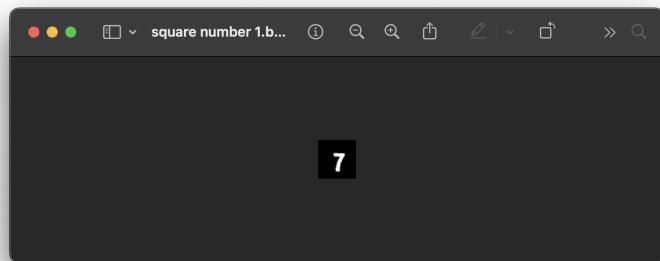


Figure 6: Cleaned grid cell

## 6 Solver

Despite all the fancy technology used in this project the Sudoku still needs to be solved.

### 6.0.1 Backtracking

In order to solve the Sudoku, we use the classic backtracking algorithm. It is a brute-force technique that consists in trying all the possibilities until a valid solution is found. If no solutions are found then the Sudoku grid is deemed unsolvable.

```
[root@localhost ~]# $ cat grid_1
53. .7. ...
6.. 195 ...
.98 ... .6.

8.. .6. ...3
4.. 8.3 ..1
7.. .2. ...6

.6. ... 28.
... 419 ..5
... .8. .79
```

Figure 7: unsolved Sudoku before being fed to the solver program

```
$ cat grid_1.result
534 678 912
672 195 348
198 342 567

859 761 423
426 853 791
713 924 856

961 537 284
287 419 635
345 286 179
```

Figure 8: solved Sudoku under the same format as the previous figure

# 7 Character Recognition

## 7.1 Global Structure

We have chosen to implement a classic structure used for character recognition. It consists (at least for this state of the project) of one 784 neurons wide input layer, a 30 neurons hidden layer as well as a 10 neurons output layer where each value represents the value of its index (for instance: an output layer of [0.0100, 0.0024, 0.9923, 0.0001, ..., 0.1254, 0.0065] means that the AI think the value is a 2 as C arrays are 0-indexed). As for training the Stochastic Gradient Descent (SGD) on the MNIST dataset freely available at <http://yann.lecun.com/exdb/mnist/> is used.

### 7.1.1 Feedforward

The activation function used is the sigmoid function although this is subject to change.

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1 + e^{-x}}$$

### 7.1.2 BackPropagation

As the sigmoid function is used during the feedforward process, its derivative is used during the backpropagation.

$$\text{Sigmoid prime: } \sigma(x)' = x * (1 - x)$$

## 7.2 Image training

in order to train IAs need a dataset so decided to use the MNIST public database as a rich dataset of labeled handwritten digits.

### 7.2.1 Inputs

The MNIST database is made with images of 28 by 28 pixels ranging from 0 to 255 saved under the idx3-ubyte format and all the labels are saved under the idx1-ubyte format. So a separate loader has been made to load data from the separate files, map all pixels from the range [0, 255] to [0, 1] and pack all images with their corresponding labels inside a designated data structure. Which is why we choose 28 \* 28 neurons as our input layer.

### 7.2.2 Training process

Our Stochastic Gradient Descent (SGD) provides 3 hyper parameters:

- The number of epochs
- The mini batch size
- The learning rate (often specified as 'eta')

We have tried many different values for each of them and have found that we can obtain 95.58% accuracy on the test dataset of the MNIST database with 30 epochs, a batch size of 100 and 3.0 for the learning rate. The training took 67 seconds on an AMD Ryzen 5 1500x.

### 7.3 Saving and Loading

In order to avoid having to train the AI each time the process is stopped and restarted, it is need to create a way to save and write all its weights and biases to a file. For this task we made a simple plain text custom file format: The first value  $n$  is always an integer and it specifies the number of layers the AI has. The value is followed by  $n$  integers  $n_i$ , with  $i$  being the index of the layer, that specify the amount of neurons of each layer. those values are enough to deduce the number of weights and biases that the AI will contains so the file will then contain  $\sum_{k=1}^{n-1} n_k$  biases and  $\sum_{k=0}^{n-2} (n_k * n_{k+1})$  weights. The format is also in plain text because it enables the use for debugging purposes.

```
|3 784 30 10 -1.301793 0.005424 -1.792563 -1.847729 -1.258441 0.395521 -0.446648 0.654379
-0.768026 -0.538065 -0.734470 -0.117964 -0.785015 0.083502 -1.097018 -0.368624 -1.045627
-0.485394 -1.122907 -1.330792 -0.419250 0.905501 -0.339100 -0.867621 -1.016356 -1.262261
-1.024899 -1.351308 -1.823998 -1.575985 -1.729217 -0.196558 -0.482185 -1.939564 -1.124072
-0.056885 -0.814465 -1.905381 -2.604399 -1.033782 0.000367 0.005699 -0.001646 0.007046 -
0.006823 -0.008372 -0.004663 0.008230 0.003898 0.004666 -0.003394 -0.004656 -0.006440 -
0.008799 0.004298 0.007853 -0.003793 0.002093 -0.002412 0.003629 0.001976 0.005369
0.006363 0.000446 0.005270 0.004762 0.008840 0.005610 0.008280 -0.008640 0.008762 0.006892
0.000709 0.006998 -0.001942 0.008499 0.006359 0.007066 -0.004208 0.003199 0.010210
0.006950 -0.007940 -0.009253 -0.014933 -0.000974 -0.001586 -0.012056 0.005217 0.001227
0.009439 0.008799 -0.002935 0.009977 -0.001768 0.001824 0.007995 0.002579 -0.002270 -
0.001639 0.001600 -0.008791 -0.002448 0.010134 -0.032619 -0.054488 -0.041746 -0.091187 -
0.128778 -0.160518 -0.206360 -0.208832 -0.183828 -0.128409 -0.180009 -0.077561 -0.040836
-0.020736 -0.012311 0.000772 0.008615 -0.000293 -0.005533 0.001563 0.008089 -0.009590 -
0.008276 0.008222 0.009312 -0.032424 -0.025816 -0.060433 -0.034472 -0.133796 -0.217373 -
0.261102 -0.078227 -0.037873 -0.016649 -0.069150 -0.034517 -0.031898 -0.264413 -0.158502
-0.204738 -0.356070 -0.239219 -0.061261 0.007084 -0.004133 0.010375 0.009016 0.004314 -
0.007327 -0.008036 0.008166 -0.026495 -0.118276 -0.134776 -0.045249 -0.083213 -0.106096 -
0.098539 0.001099 -0.160585 -0.244843 -0.300797 0.195786 -0.290279 -0.153622 0.010702 -
0.334157 -0.322236 -0.285692 -0.149635 0.041110 0.013514 -0.004188 -0.008384 0.005394 -
0.008146 -0.005346 -0.001073 -0.016316 -0.140879 -0.288739 -0.218045 -0.213568 -0.256679
```

Figure 9: Initial values of saved AI trained on the MNIST dataset (95% accuracy)

## 7.4 Note on performance

Because of the nature of neural networks and the Stochastic Gradient Descent algorithm, AIs require huge amounts of computing power, especially about matrix multiplications. In order to try to gain performance our AI code base follows a no-malloc after startup policy where any function used during training or normal usage (such as network\_SGD() or network\_evaluate()) do not allocate any data on the heap as to avoid any potential memory fragmentation, memory leaks or use-after-free bugs. Another step taken to help performance is the use of a BLAS library (in our case openblas). BLAS (or Basic Linear Algebra Subprograms) libraries provide heavily optimized functions to handle matrix such as sgemm() and dgemm(). Our AI can seamlessly compile with and without linking to the library so it can be trained on a more powerful machine that has a BLAS library and then used or trained on smaller dataset on another computer.