OCR by Team OT
Final Report

Mathieu Dubrulle
Emile Preumont
Elias Tebbani
Assil Achour

December 2021

Contents

1 Introduction

This report gives an overview of our work on the **Optical Character Recognition** Project (OCR) by **Team OT**. In the scope of this project, an OCR made for detecting and completing a Sudoku grid is made. To do so, the program starts by analyzing the image, then solves the Sudoku grid to finally display it being solved. This project is implemented in C and uses the SDL library. Files are manipulated in the .bmp format but our program can also analyse images in other formats.

2 Image loading and color removal

For this task, the IMG_Load function, that is pre-created by the SDL library, was used.

2.1 Grayscale

To apply grayscale, our image is parsed and is applied the following formula for the value r, g and b and for every pixel :
0.3*r + 0.59*g + 0.11*b;

2.2 Threshold

The main idea in our thresholding processing is that each pixel is compared to an average of the surrounding pixels.If the value of the current pixel is 35 percent lower than the average then it is set to black, otherwise it is set to white.

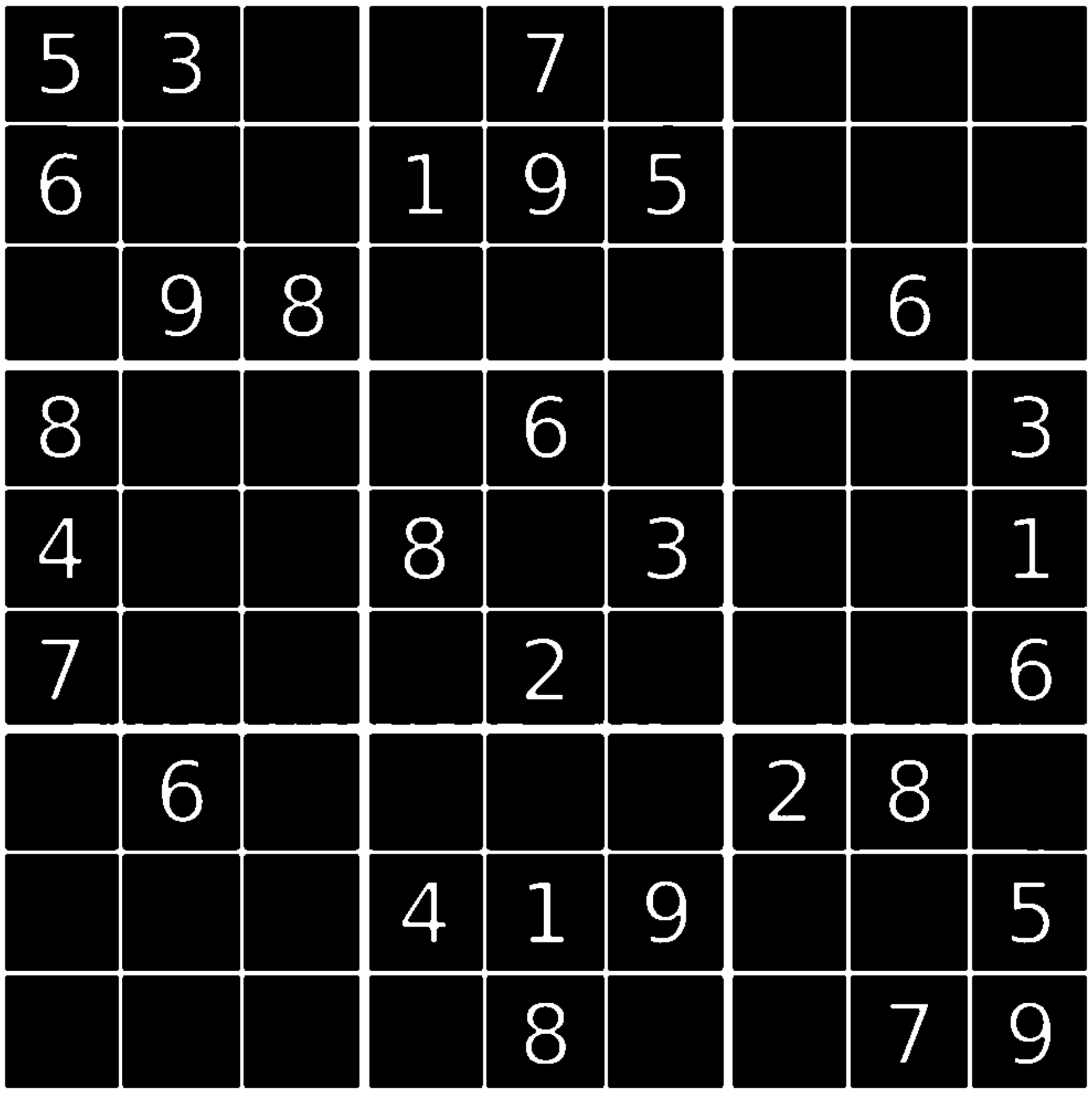
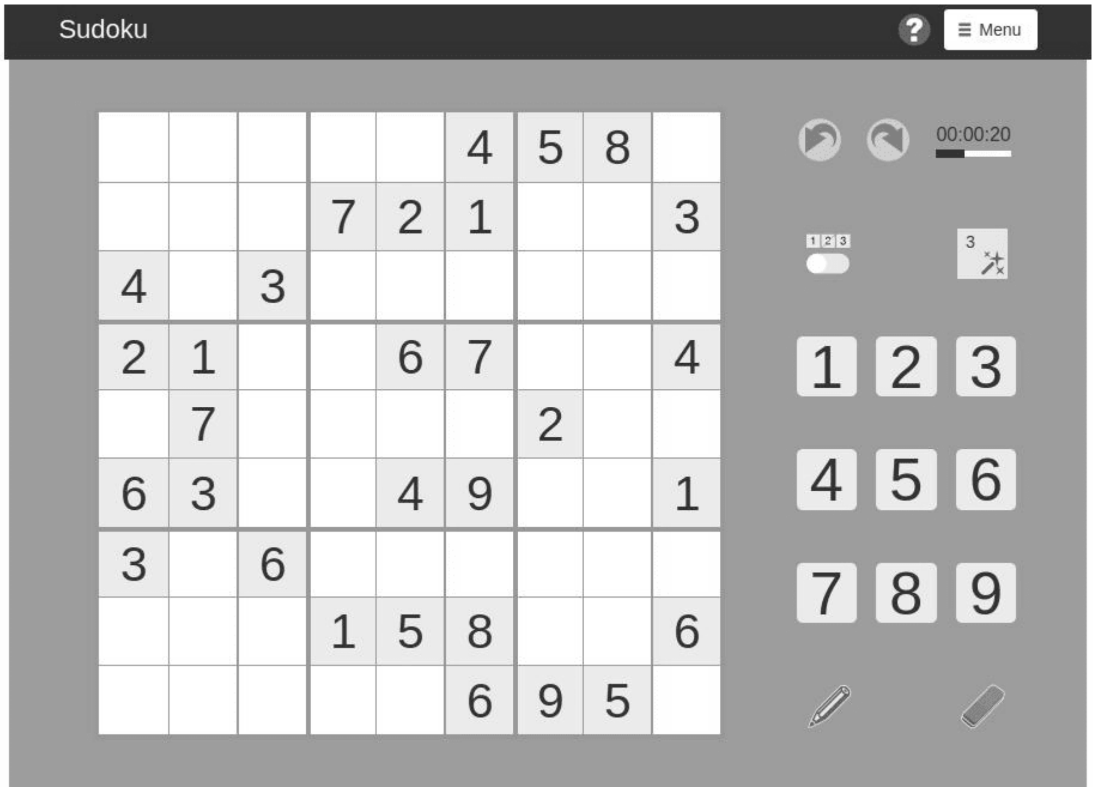


Figure 1: Current output of the adaptive threshold on gamma of image 1



					4	5	8	
			7	2	1			3
4		3						
2	1			6	7			4
	7					2		
6	3			4	9			1
3		6						
			1	5	8			6
					6	9	5	

Figure 2: Current output of grayscale filter on Image 3

3 Detection of the grid

3.1 Noise removing and blur

Gamma : In order to reduce the noise of the given pictures we will brighten the image a lot using gamma correction. The image will become so bright that most of the noise will be removed.

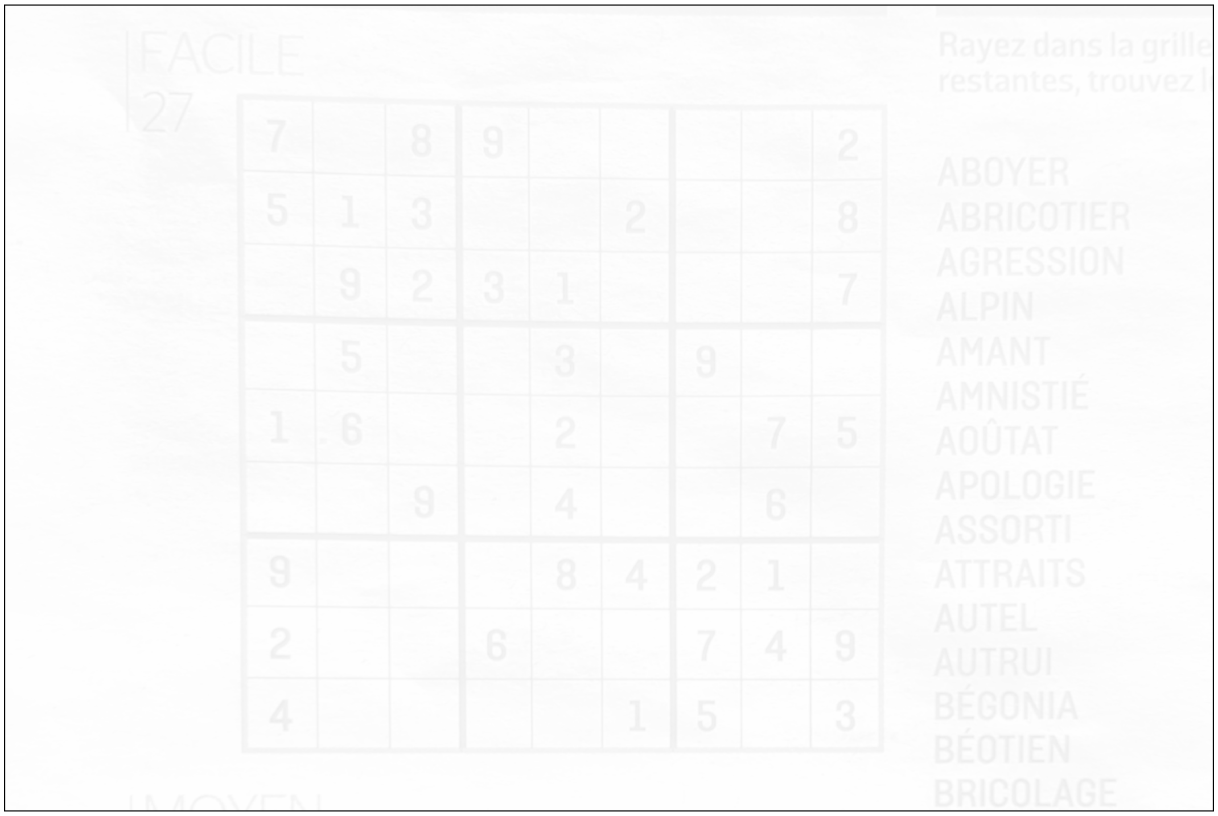
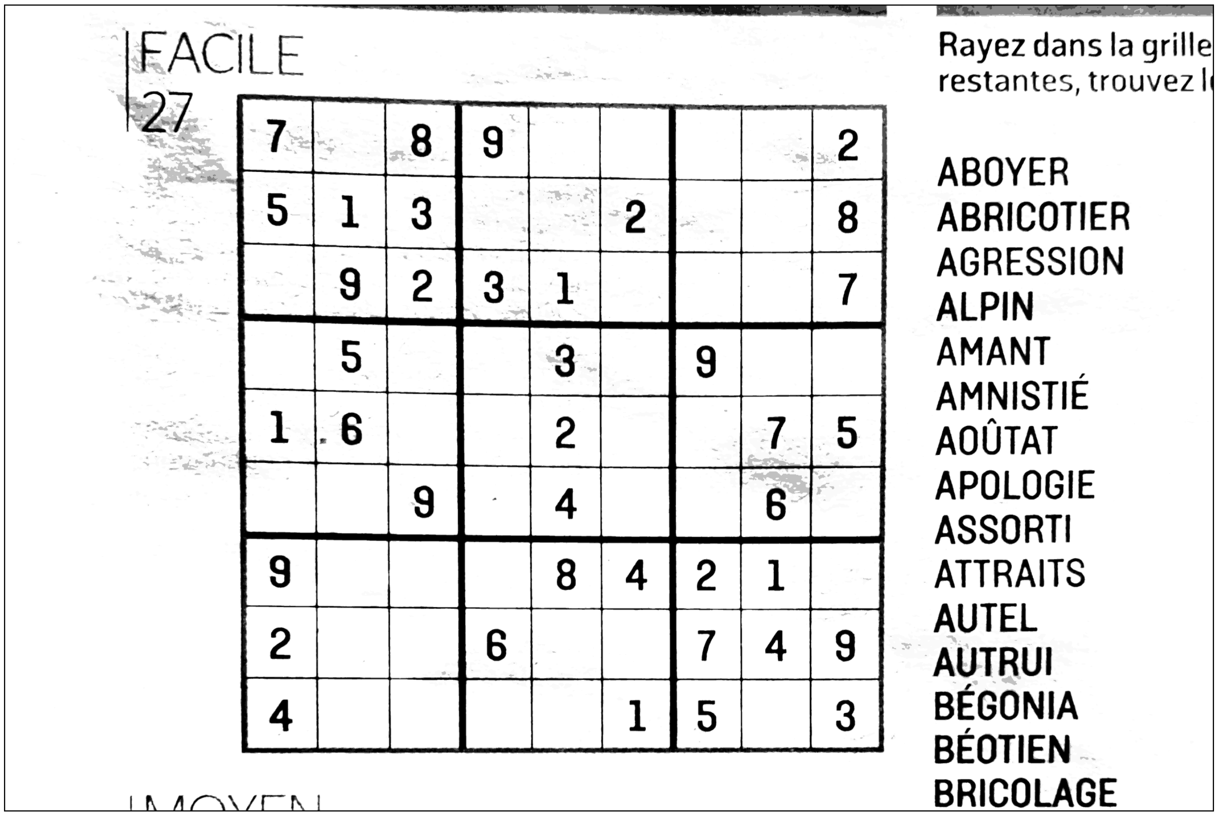


Figure 3: Current output of gamma on image 4

Contrast : Now that the image is really bright we need to reconvert into a "normal" image using a contrast correction factor:

$$\frac{259(C+255)}{255(C-259)}$$

C being the average value of the pixels of our image



FACILE
27

7		8	9					2
5	1	3			2			8
	9	2	3	1				7
	5			3		9		
1	6			2			7	5
		9		4			6	
9				8	4	2	1	
2			6			7	4	9
4					1	5		3

MOYEN

Rayez dans la grille
restantes, trouvez l

ABOYER
ABRICOTIER
AGRESSION
ALPIN
AMANT
AMNISTIÉ
AOÛTAT
APOLOGIE
ASSORTI
ATTRAITS
AUTEL
AUTRUI
BÉGONIA
BÉOTIEN
BRICOLAGE

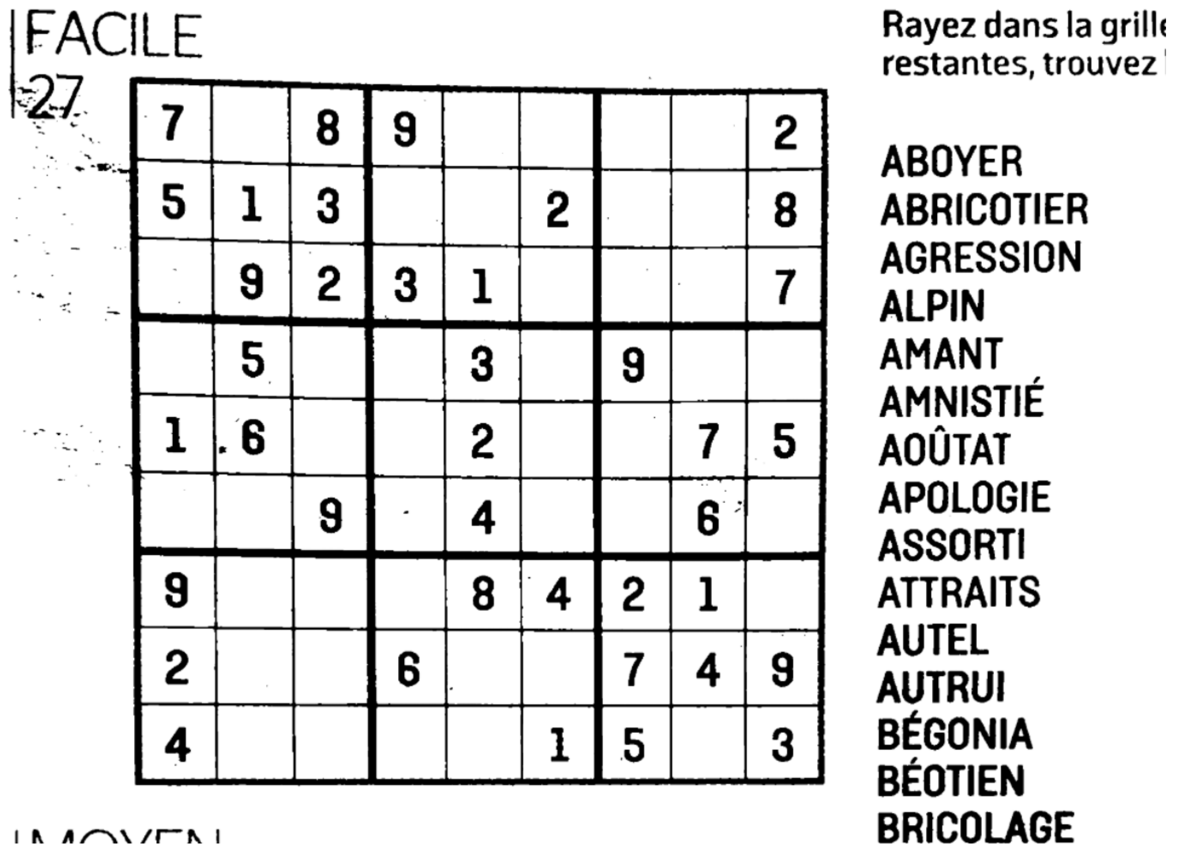
Figure 4: Current output of contrast on gamma of image 4

Otsu Tresholding : At the previous defense, our tresholding method wasn't good enough. It was not working for every image and we still a concerning number of noise on image 4.
So we implemented Otsu tresholding method that can at the same time remove noise but also give us a black and white output without damaging the original outline of our picture. The method for this algorithm is to find a

specific treshold value :
-First we have to use find the histogram of the image.

-Then an algorithm is going to parse our histogram computing the minimal variance of the black and white giving the correct treshold as output.

-We then parse our image and if our current pixel color value is superior to this treshold we set it to white, otherwise we set it to black.



FACILE
27

7		8	9					2
5	1	3			2			8
	9	2	3	1				7
	5			3		9		
1	6			2			7	5
		9		4			6	
9				8	4	2	1	
2			6			7	4	9
4					1	5		3

Rayez dans la grill
restantes, trouvez

ABOYER
ABRICOTIER
AGRESSION
ALPIN
AMANT
AMNISTIÉ
AOÛTAT
APOLOGIE
ASSORTI
ATTRAITS
AUTEL
AUTRUI
BÉGONIA
BÉOTIEN
BRICOLAGE

Figure 5: Current output of Otsu on contrast of image 4

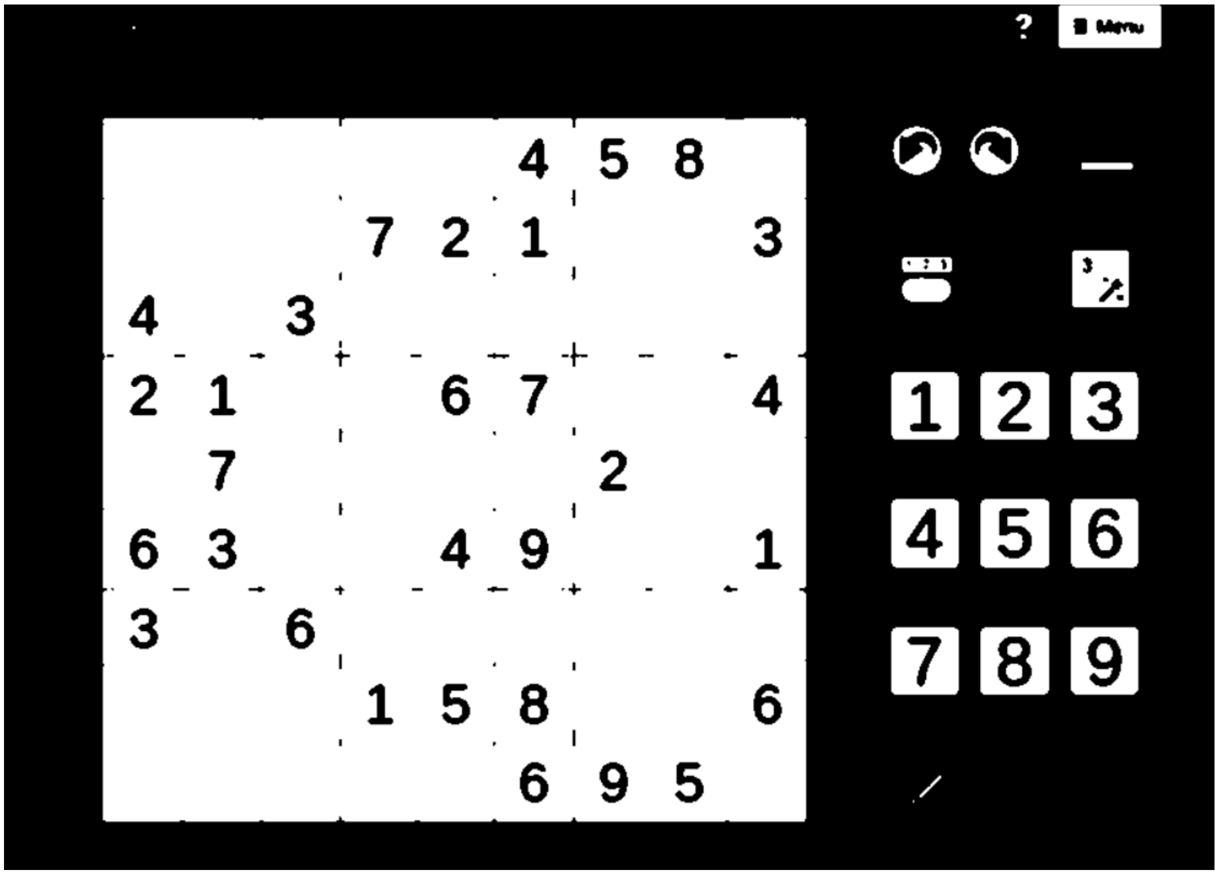


Figure 6: Current output of Otsu on contrast of image 3

Gaussian Blur : A Gaussian blur is applied to the image. It is applied using a mask (or convolution matrix) of 5 by 5 generated with a Gaussian function with the parameter gamma of 1,4. This filter reduces noise and removes small details of the image. This step prepares the next steps of edge and line detection. The matrix used :

$$\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

The result of the convolution is then multiplied by $\frac{1}{159}$ and obtain the following result :

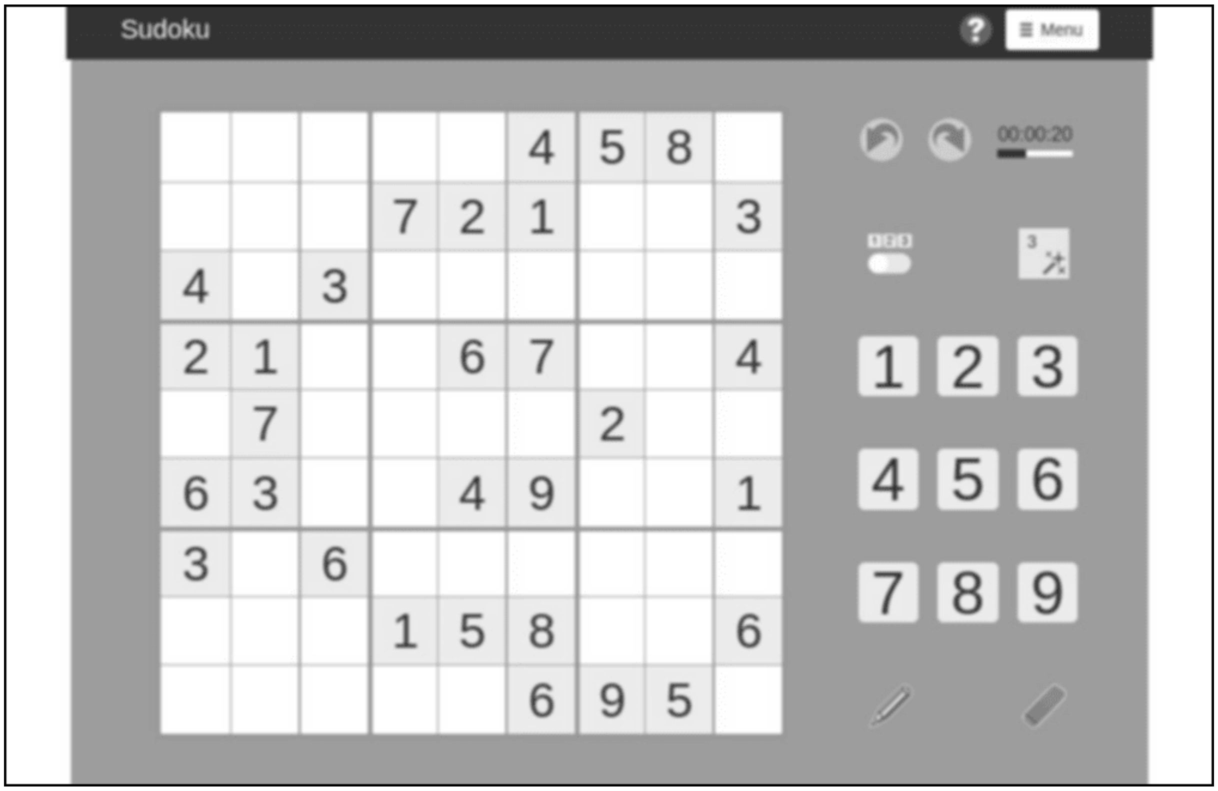


Figure 7: Current output of Gaussian filter on Image 3

3.2 Edge detection

Sobel Filter : The Sobel filter is used to detect the edges of the image. The Sobel filter uses convolution of two matrices to get a gradient of the pixel values calculated as follows :

$$\sqrt{(Gx^2) + (Gy^2)}$$

with :

$$\text{Gx} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and Gy} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The direction of the pixel is calculated as so : Θ = atan2(Gy, Gx).
Sobel filter gives good results but the lines resulting are thick and so not perfect for line detecting. This is why we apply Canny after it.

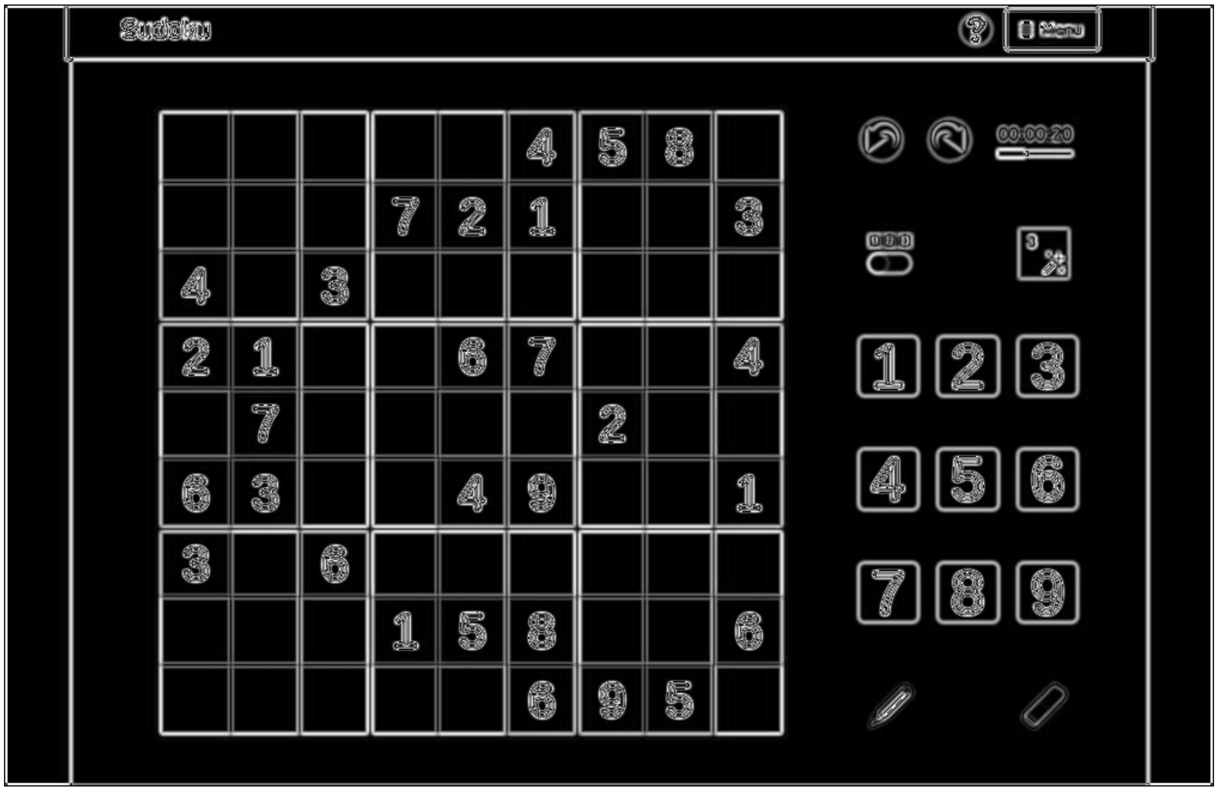


Figure 8: Current output of Sobel filter on Image 3

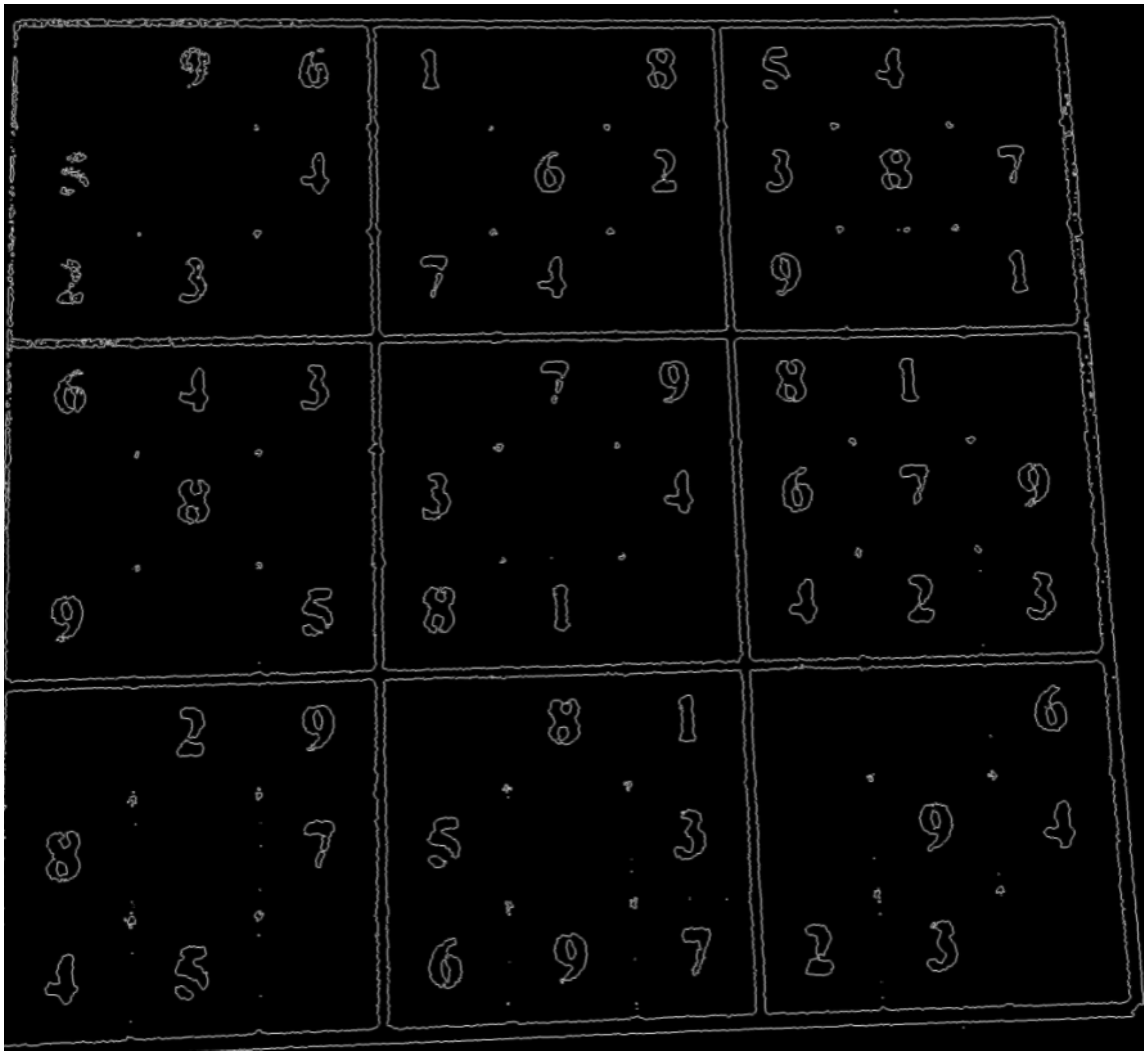


Figure 9: Current output of Sobel filter on Image 6

Canny Filter : Canny filter is used to detect edges and the resulting image has only 1 pixel wide lines. The first step is the lower bound cut-off suppression which removes the non-maximum on each lines. The second step is the double threshold and the Edge tracking by hysteresis which can be resumed as the analysis of which pixel are kept based on their connections and their gradient.

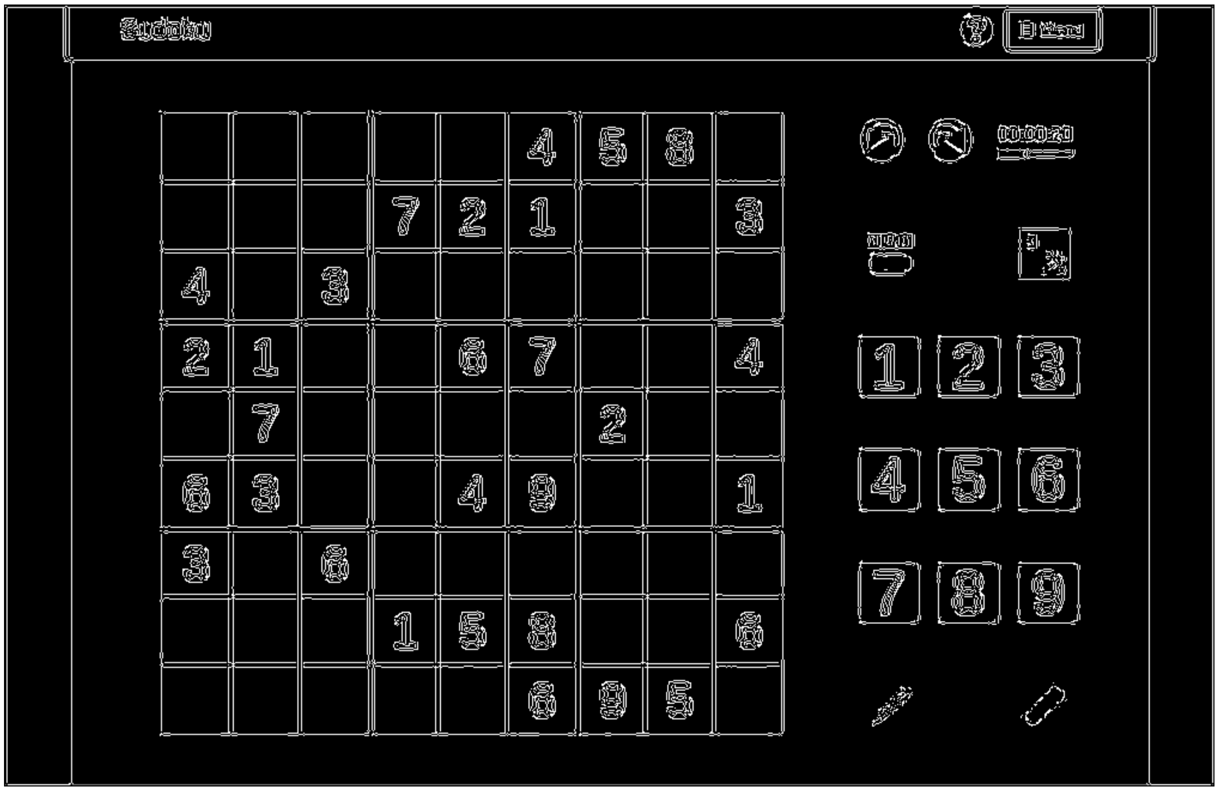


Figure 10: Current output of Canny filter on Image 3

3.3 Line Detection

Hough transformation : In order to simplify the detection of the Sudoku after the image processing, the Hough transformation algorithm has been implemented. This algorithm find the lines of the image and it works as follows:

1. Getting the edges pixel lines polar coordinates. Those pixel are recognisable thanks to previous image processing algorithms making them white.
2. Saving the number of occurrences of the lines into a Hough space which is a matrix with two coordinates, rho and theta from the line polar coordinates.
3. Selecting only the lines having high occurrences in the Hough space and drawing them. Those lines are the new edges which will be used in the Sudoku grid detection later on.
4. Selecting the two angles with the highest occurrences and then using them to automatically rotate the image.
5. We have upgraded our Hough Transform function in order for it to give an output with way less lines than before. Hence the Intersection detection is way easier.

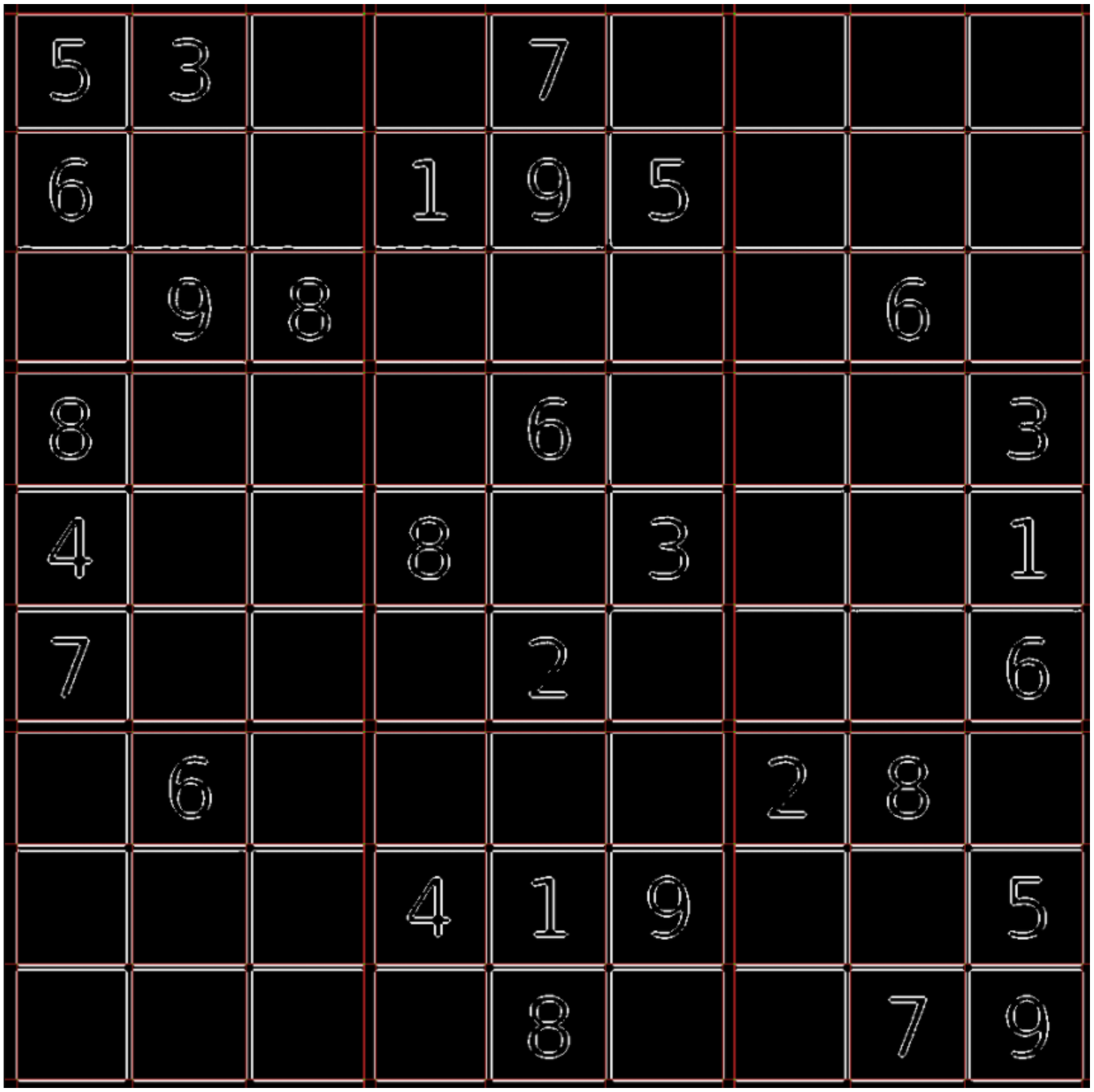


Figure 11: Current output of Hough transform on Image 1

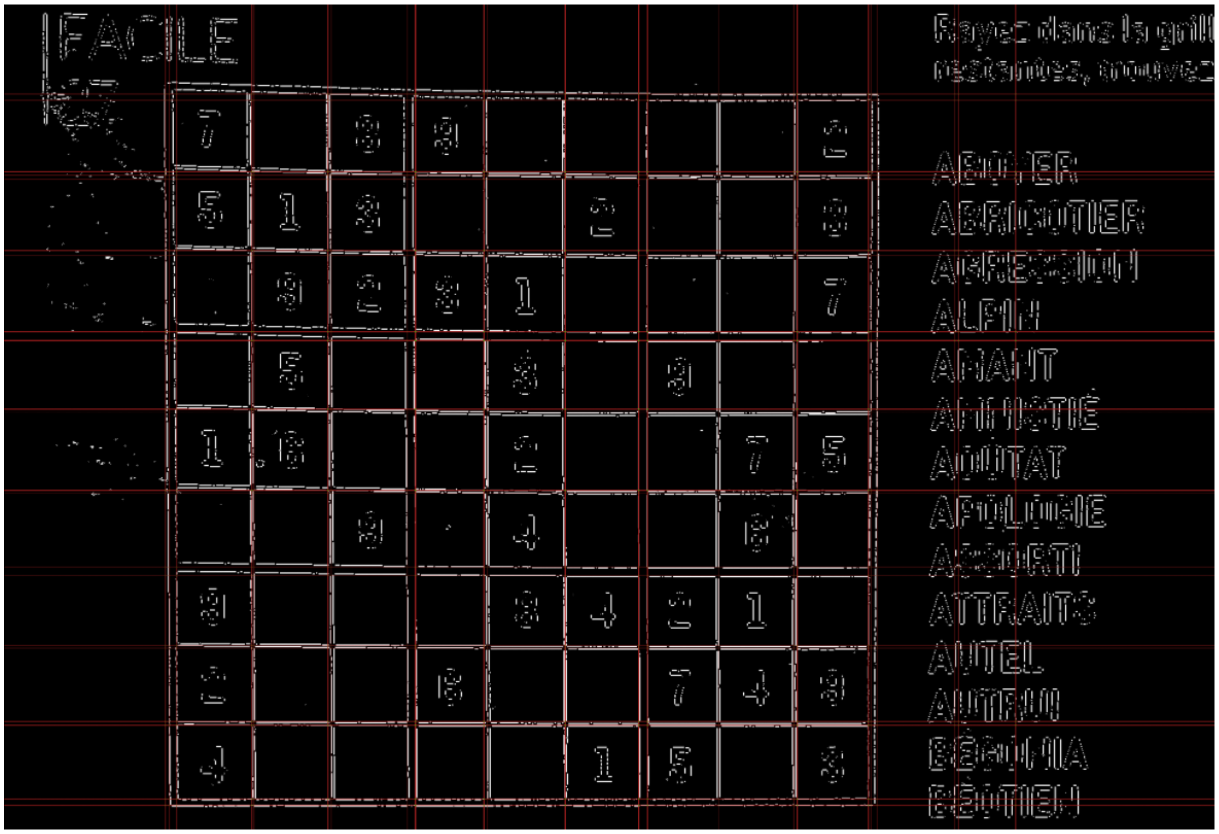


Figure 12: Current output of Hough transform on Image 4

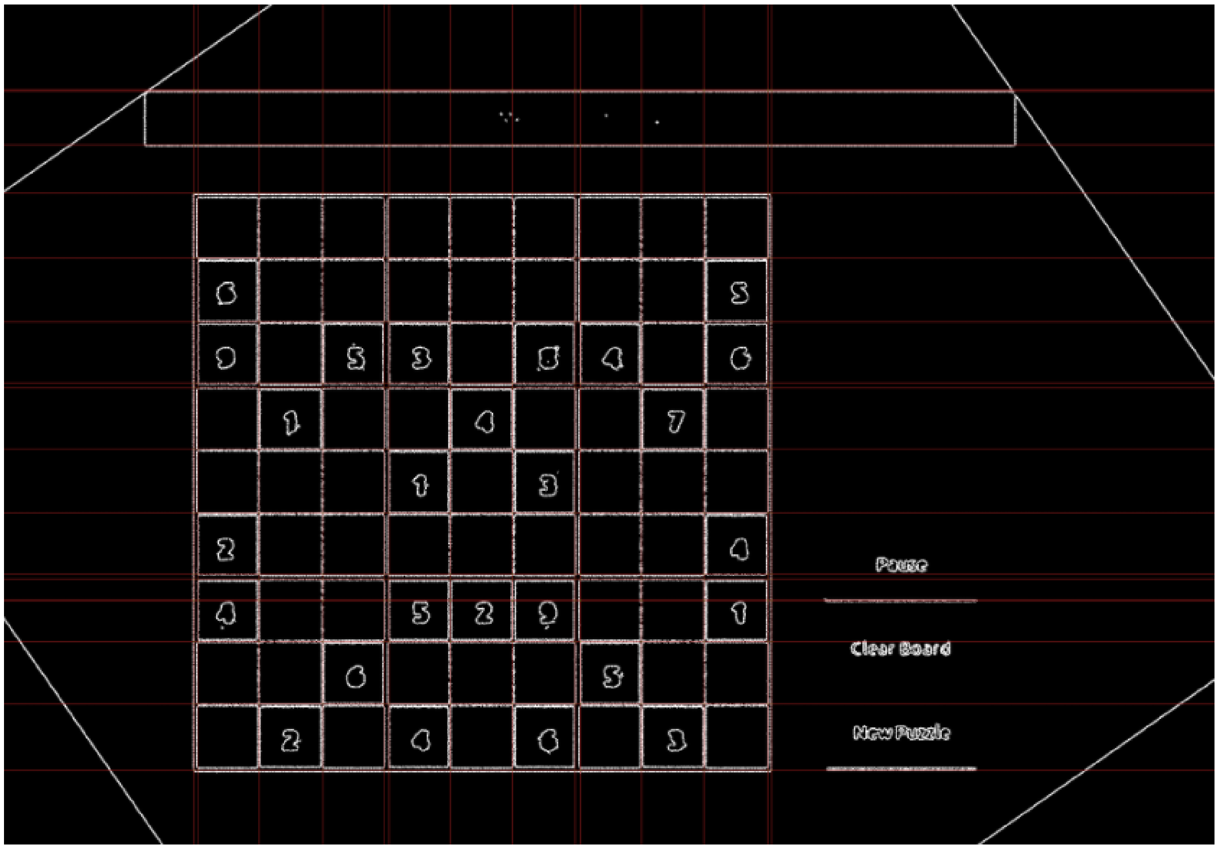


Figure 13: Current output of Hough transform on Image 5

3.4 Square detection

3.4.1 Intersection detection

In order to detect the square, we have to detect the intersections between the lines drawn by the Hough transform function. To do so, we modified the function so when it encounters a pixel that has already been drawn, it draws a pixel in green.

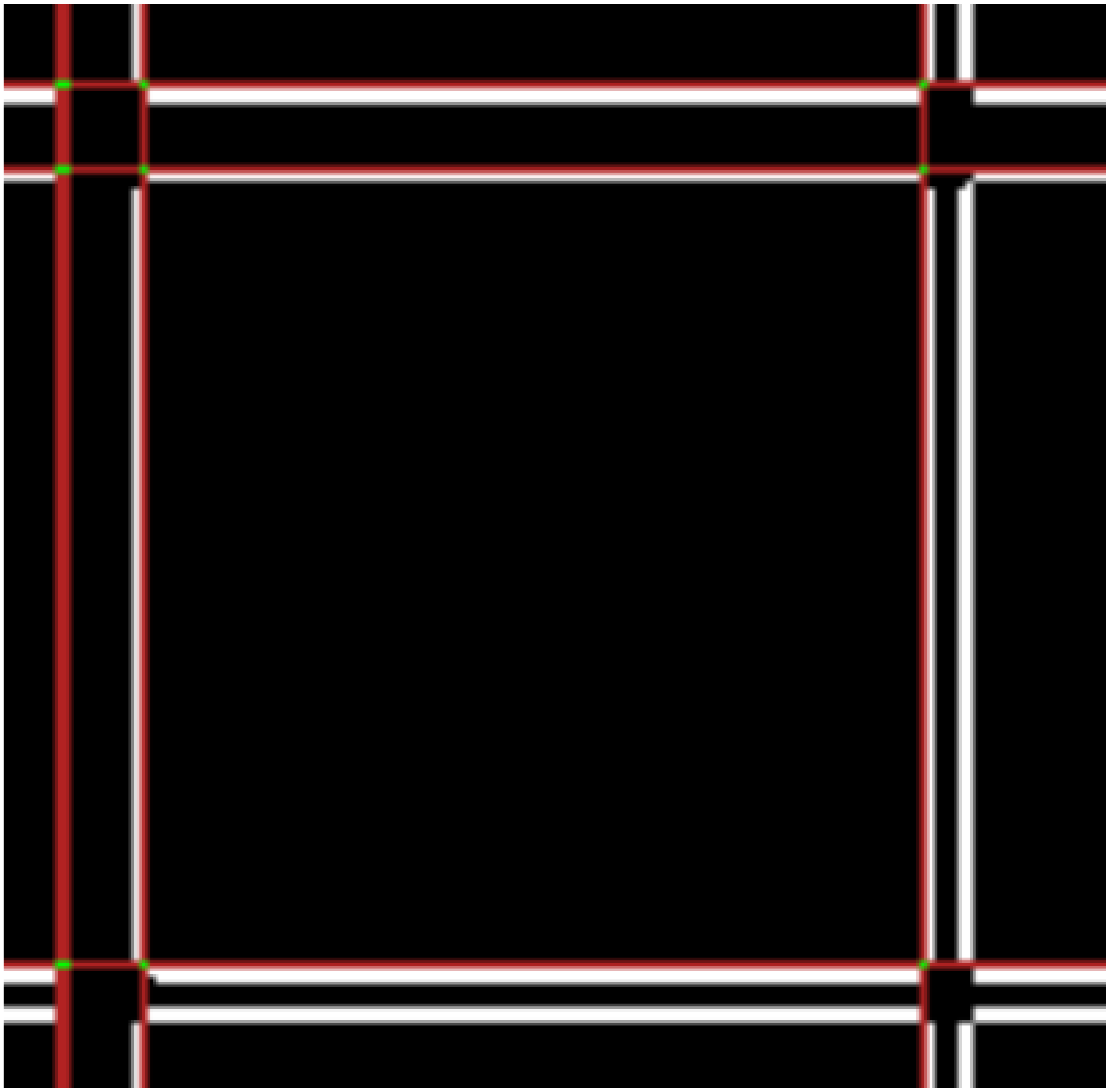
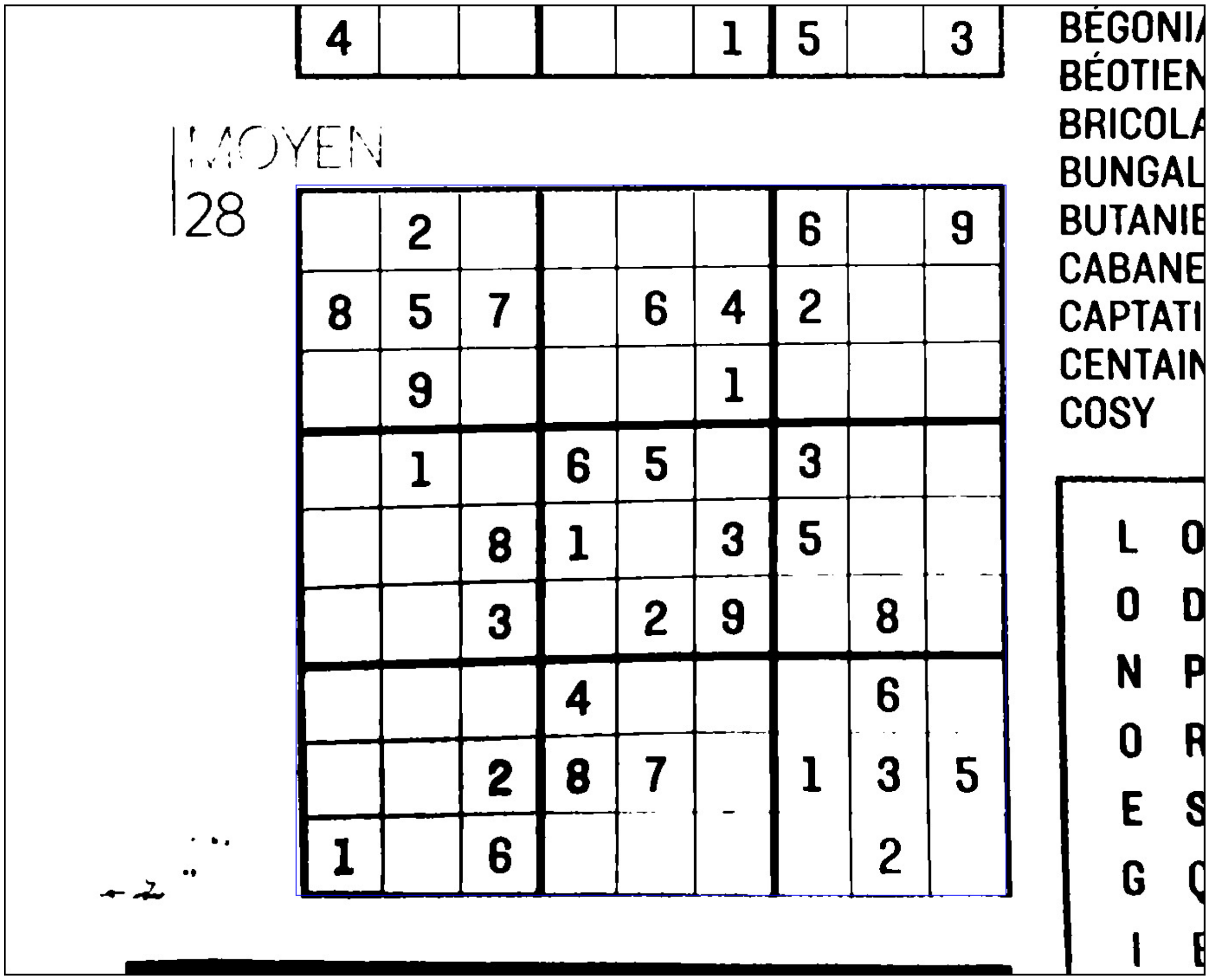


Figure 14: Intersections between lines on Hough of Image 1

3.4.2 Square detection

To detect the square from the intersections, we go through the image looking for the green pixels which represent the intersections. When we encounter one, we look for the pixel in the bottom right part of the image that could form a square with the previous pixel (from the diagonal between the two points). When we find the couple of pixel which defines our square, we split the image.



4					1	5		3

MOYEN
28

	2					6		9
8	5	7		6	4	2		
	9				1			
	1		6	5		3		
		8	1		3	5		
		3		2	9		8	
			4				6	
		2	8	7		1	3	5
1		6					2	

BÉGONI
BÉOTIEN
BRICOL
BUNGAL
BUTANIE
CABANE
CAPTATI
CENTAIN
COSY

Figure 15: Square detected on the image 2 (in blue)

4 Splitting of the picture

A function invoked with the coordinates of the top left pixel of the Sudoku grid and the width and the height of the grid allows one to split the image in 81 sub-images corresponding to the cells of the grid respectively. It uses **this function**. Once the image is splitted in 81 subimages, we still need to prepare the image for the AI. The image has to be resized to 28 by 28 pixel format, should have the number in white and the rest in black, there should be the least noise possible on the image and finally the number should be centered on the image. To scale the image, we use the function rotozoomXY from the gfx library. To center the number, we do an average of all the coordinates of the white pixel and compare the result to 14 to know towards which position we should redraw the number and then redraw it. The rest is just basic image processing (inverting the pixels and cleaning the borders of the image).

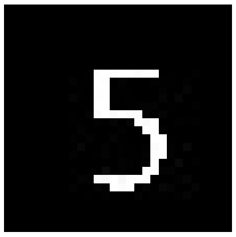


Figure 16: first cell of first image given by splitting

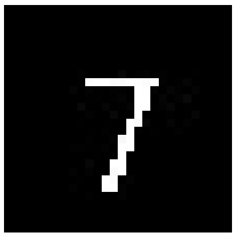


Figure 17: another cell from the split of image 1

5 XOR Network

5.1 The Network

A first neural network has been made : a **XOR** network. This has been made to try and first get an idea of how neural networks work in general. It will then be adapted to work as a digit recognizer, taking as input an image instead of numbers. The back propagation aspect of this network uses the **Stochastic Gradient Descent**. It has two input layers, two hidden layers and one output layer.

The back propagation aspect of this network uses the **Stochastic Gradient Descent**.

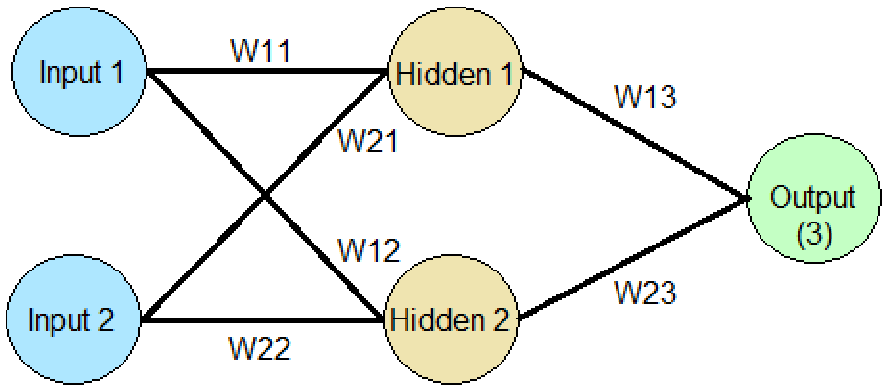
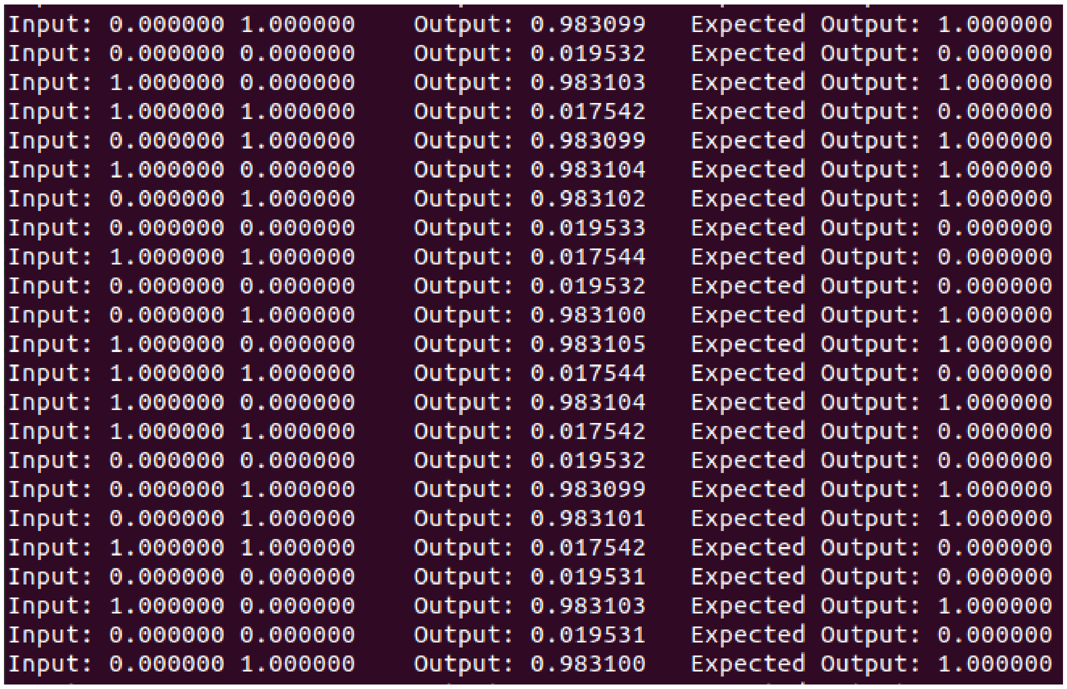


Figure 18: XOR Network

By iterating over a given number of epochs, this XOR Network can minimize its errors and make it learn the XOR operation efficiently.

Here is an example of the algorithm going through many different epochs.

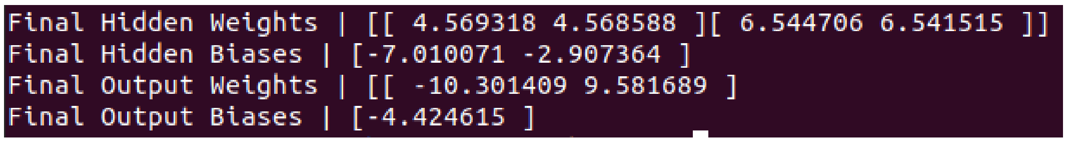


```
Input: 0.000000 1.000000    Output: 0.983099   Expected Output: 1.000000
Input: 0.000000 0.000000    Output: 0.019532   Expected Output: 0.000000
Input: 1.000000 0.000000    Output: 0.983103   Expected Output: 1.000000
Input: 1.000000 1.000000    Output: 0.017542   Expected Output: 0.000000
Input: 0.000000 1.000000    Output: 0.983099   Expected Output: 1.000000
Input: 1.000000 0.000000    Output: 0.983104   Expected Output: 1.000000
Input: 0.000000 1.000000    Output: 0.983102   Expected Output: 1.000000
Input: 0.000000 0.000000    Output: 0.019533   Expected Output: 0.000000
Input: 1.000000 1.000000    Output: 0.017544   Expected Output: 0.000000
Input: 0.000000 0.000000    Output: 0.019532   Expected Output: 0.000000
Input: 0.000000 1.000000    Output: 0.983100   Expected Output: 1.000000
Input: 1.000000 0.000000    Output: 0.983105   Expected Output: 1.000000
Input: 1.000000 1.000000    Output: 0.017544   Expected Output: 0.000000
Input: 1.000000 0.000000    Output: 0.983104   Expected Output: 1.000000
Input: 1.000000 1.000000    Output: 0.017542   Expected Output: 0.000000
Input: 0.000000 0.000000    Output: 0.019532   Expected Output: 0.000000
Input: 0.000000 1.000000    Output: 0.983099   Expected Output: 1.000000
Input: 0.000000 1.000000    Output: 0.983101   Expected Output: 1.000000
Input: 1.000000 1.000000    Output: 0.017542   Expected Output: 0.000000
Input: 0.000000 0.000000    Output: 0.019531   Expected Output: 0.000000
Input: 1.000000 0.000000    Output: 0.983103   Expected Output: 1.000000
Input: 0.000000 0.000000    Output: 0.019531   Expected Output: 0.000000
Input: 0.000000 1.000000    Output: 0.983100   Expected Output: 1.000000
```

Figure 19: XOR Network going through epochs

Overtime, the output slowly gets closer to the expected output. By comparing the obtained output with the expected one, the algorithm is able to get better results.

Finally, the final weights and biases are printed on the screen



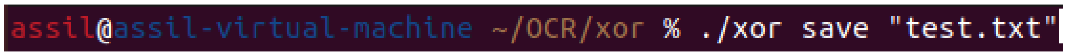
```
Final Hidden Weights | [[ 4.569318 4.568588 ][ 6.544706 6.541515 ]]
Final Hidden Biases | [-7.010071 -2.907364 ]
Final Output Weights | [[ -10.301409 9.581689 ]
Final Output Biases | [-4.424615 ]
```

Figure 20: Final Results

5.2 Save and Load

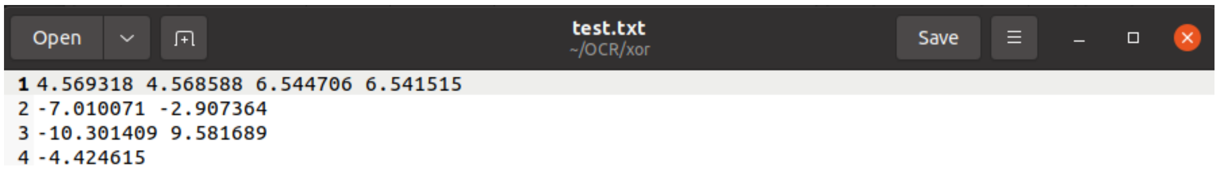
The **XOR Network** can also save the obtained **weights** and **biases** after going through the algorithm.

By using the "**save**" keyword, the final results of the algorithm can be saved into a specified path. (Can be non-exist).



```
assil@assil-virtual-machine ~/OCR/xor % ./xor save "test.txt"
```

Figure 21: Starting the XOR Network and saving



```
4.569318 4.568588 6.544706 6.541515
-7.010071 -2.907364
-10.301409 9.581689
-4.424615
```

Figure 22: Weights and Biases saved into the test.txt file

The same process can be done with the load keyword. The **XOR Network** can start with a specific set of weights and biases instead of randomly generated ones.

6 Character Recognition Neural Network

6.1 The Network

The neural network is used for **digit recognition**. With a 28x28 pixels image, our network can detect what numbers it is recognizing automatically. It is made of 3 layers. It has 784 input nodes, 15 hidden nodes and 10 output nodes. An input layer represents a pixel of the image, this is why 784 inputs nodes are used. By giving it an image, the network will return an integer number.

6.2 Image to Array

After the main image has been processed and split, each image is turned into an array of zeros and ones. Zeros are used to represent black pixels while ones are used to represent white pixels. Once the image has been turned into an array, it is ready to be used by the neural network.

6.3 Initialization of the network

Before starting the network, the training set, composed of a directory of images along a text file representing the expected outputs, are loaded in the network. (Using the image to array function previously mentioned)

The training set is then shuffled to assure that the network receives images in a random manner and to make sure the network does not "remember" the order of images and not the image itself.

Weights and biases of each neuron are randomly generated using **He initialization** to avoid the exploding gradients problem.

6.4 Front Propagation

The front propagation uses the **leaky relu** function as an activation function. For the output layer, it uses the softmax function to compute the probability of each output neurons. The highest probability of a neuron represents the output (between 0 and 9) of the network.

6.5 Back Propagation

For the back propagation, the **cross entropy loss** function along the **softmax / leaky relu** derivative are used to compute the amount of errors in weights and biases. Weights and biases are then slowly updated (with an adapted learning rate).

6.6 Save and Load

Like in the XOR Network, save and loads functions were implemented. Once the network is finished training, it will print the error rate. If the error rate is not optimal, the network can be launched again using previously computed weights and biases.

6.7 Final Network

In the final OCR software, only the front propagation is implemented along a file with weights and biases. The software loads weights and biases onto the network and then run it on all the given split images. If the image is representing a number, it will be given to the network and it will return its associated number.

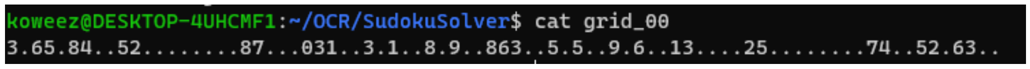
7 Implementation of the sudoku solving algorithm and save and load

7.1 Sudoku Solving algorithm

The sudoku solver algorithm uses backtracking. Basically it works making assumptions on the possible solution of the sudoku. It iterates through the grid and make an assumption of a legal value at the first non-empty cell. Then it follows this assumption until realising it does not work and tries another value. This mechanism goes on until the soduku is solved.

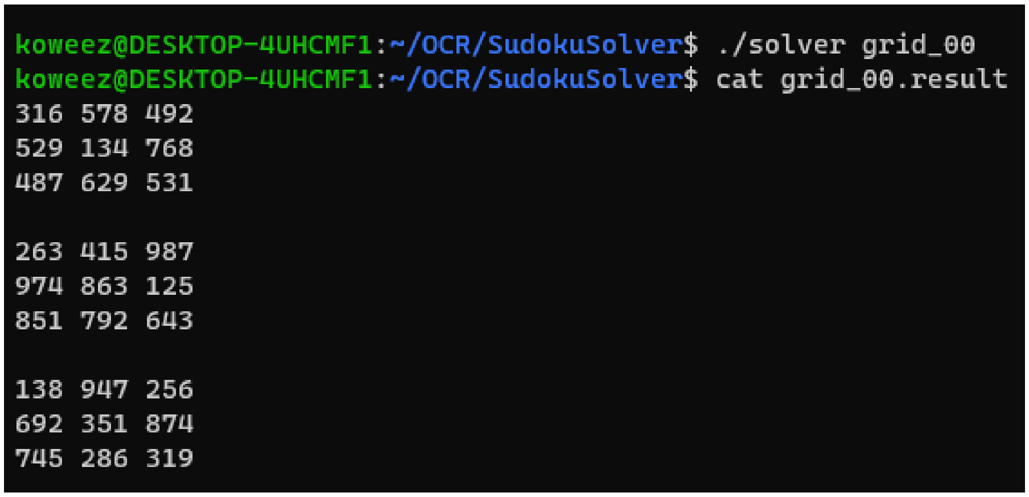
7.2 File manipulation and Save and Load of the results

As a result of the whole image processing part, the Neural Network returns a series of numbers corresponding to the sudoku grid. Those numbers are saved in a file which will be retrieved for the sudoku solver algorithm which will itself return another file corresponding the solved sudoku grid. As for now the neural network is not ready yet to work on numbers, a file with an unsolved sudoku grid has been created to work as the input of the sudoku solver. The input file is specified as an argument of the main and the output file is created in the program.



```
koweez@DESKTOP-4UHCMF1:~/OCR/SudokuSolver$ cat grid_00
3.65.84..52........87...031..3.1..8.9..863..5.5..9.6..13....25........74..52.63..
```

Figure 23: input file

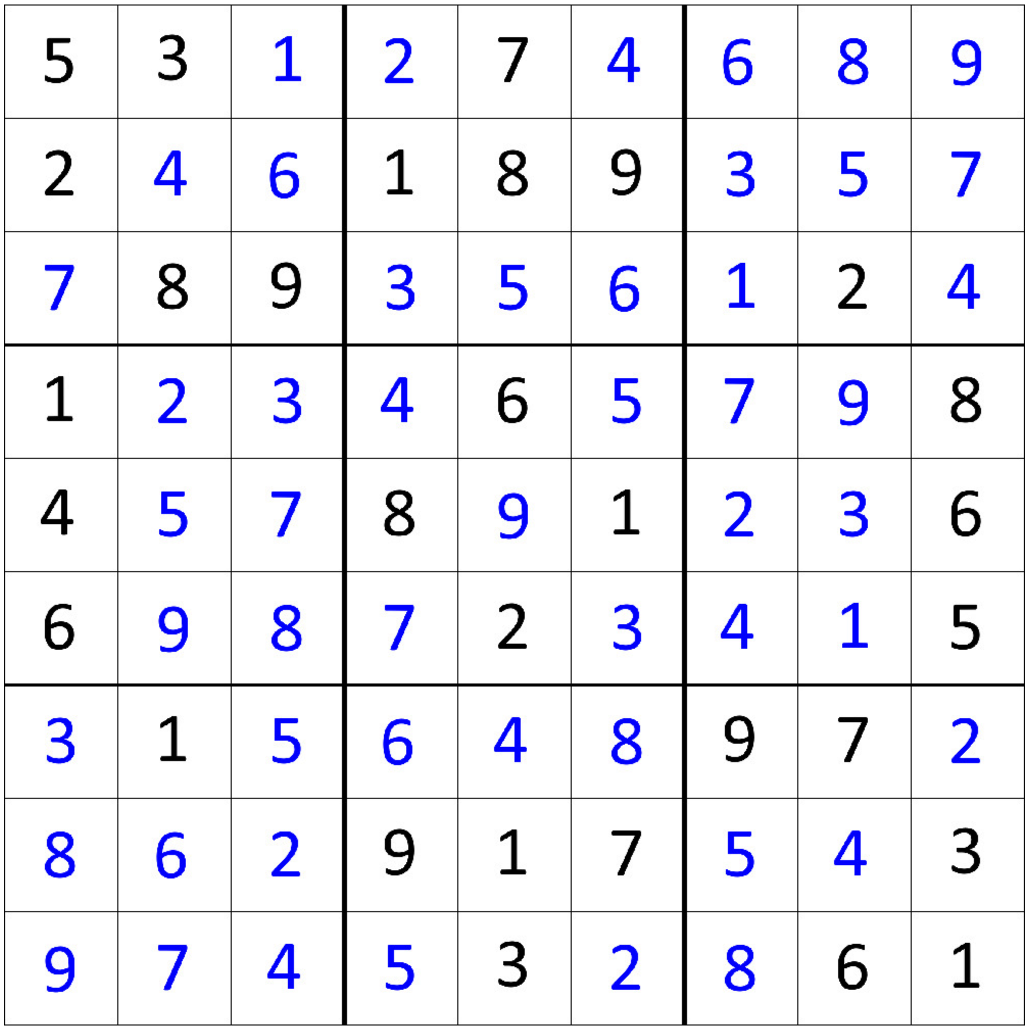


```
koweez@DESKTOP-4UHCMF1:~/OCR/SudokuSolver$ ./solver grid_00
koweez@DESKTOP-4UHCMF1:~/OCR/SudokuSolver$ cat grid_00.result
316 578 492
529 134 768
487 629 531

263 415 987
974 863 125
851 792 643

138 947 256
692 351 874
745 286 319
```

Figure 24: output file

8 Image reconstruction

Once we have the solved grid in a text file, we can use this information to reconstruct the grid. As the SDL library does not have a paste function, we coded one. Then we created a folder with a sample of number images in format 256 by 256. From there we print the images sample on a 1000 by 1000 white surface and print a grid above the numbers to get the final grid. The new numbers are drawn in blue so we can see clearly the solving of the grid. To detect the new numbers, we simply compare the input and output file of the solver.



5	3	1	2	7	4	6	8	9
2	4	6	1	8	9	3	5	7
7	8	9	3	5	6	1	2	4
1	2	3	4	6	5	7	9	8
4	5	7	8	9	1	2	3	6
6	9	8	7	2	3	4	1	5
3	1	5	6	4	8	9	7	2
8	6	2	9	1	7	5	4	3
9	7	4	5	3	2	8	6	1

Figure 25: the output image of the OCR on an example grid

9 Graphical User Interface

The graphical user interface allowing us to link all the different parts of the project has been made using the **GTK** library and **Glade**. The window is a 1280x720 window and is named **Sudoku Solver**. For each operation in the OCR, a page in a **GtkStack** is attributed and different buttons allow the user to back and forth in the process as much as he wants to. Buttons are also used to actually execute the code or to close the window.

9.1 Starting page

This page is just the launching page, the **Start** button allows the user to go to the next page.

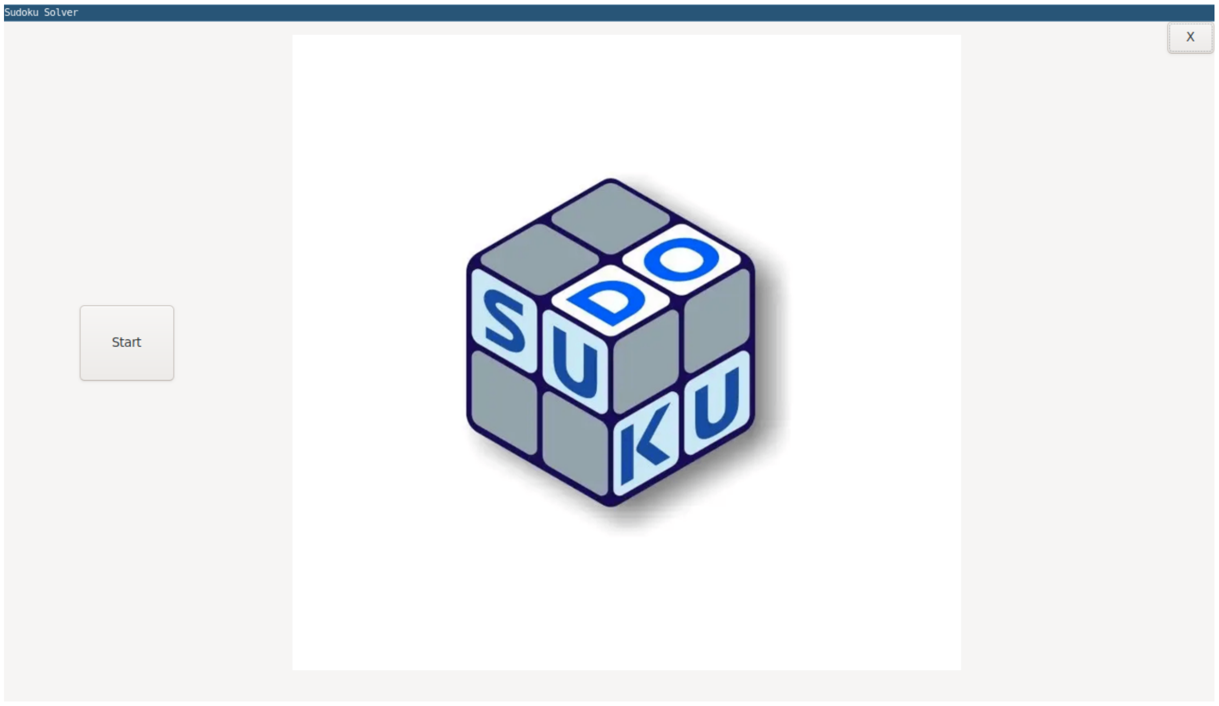


Figure 26: page 0 - the starting page

9.2 Input page

This page manages the input of the OCR. Clicking on the above button allows the user to select an image from his computer and clicking on the **Process Image** button launches the preprocessing on the image and goes to the next page.

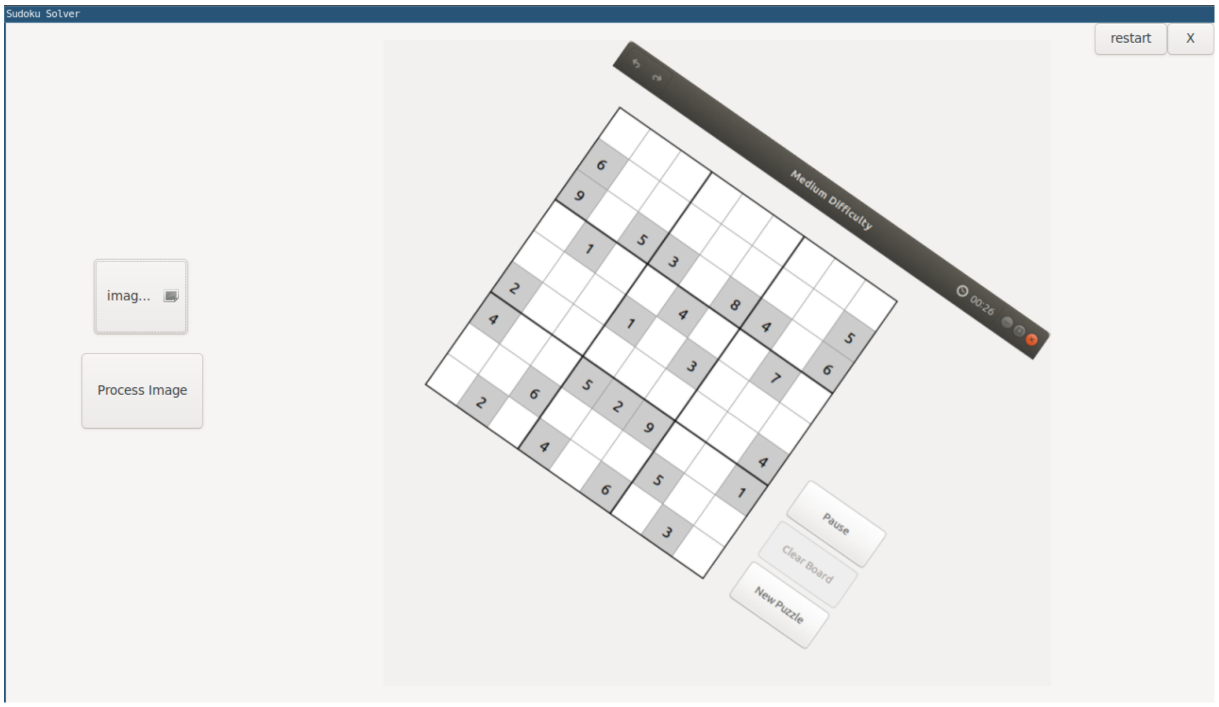


Figure 27: page 1 - the input page

9.3 Square detection page

This page is about the detection of the Sudoku grid. Clicking on the **Detect Square** button detects the square, displays a black and white version of the image with blue lines (not always visible) around the detected square and goes to the next page.

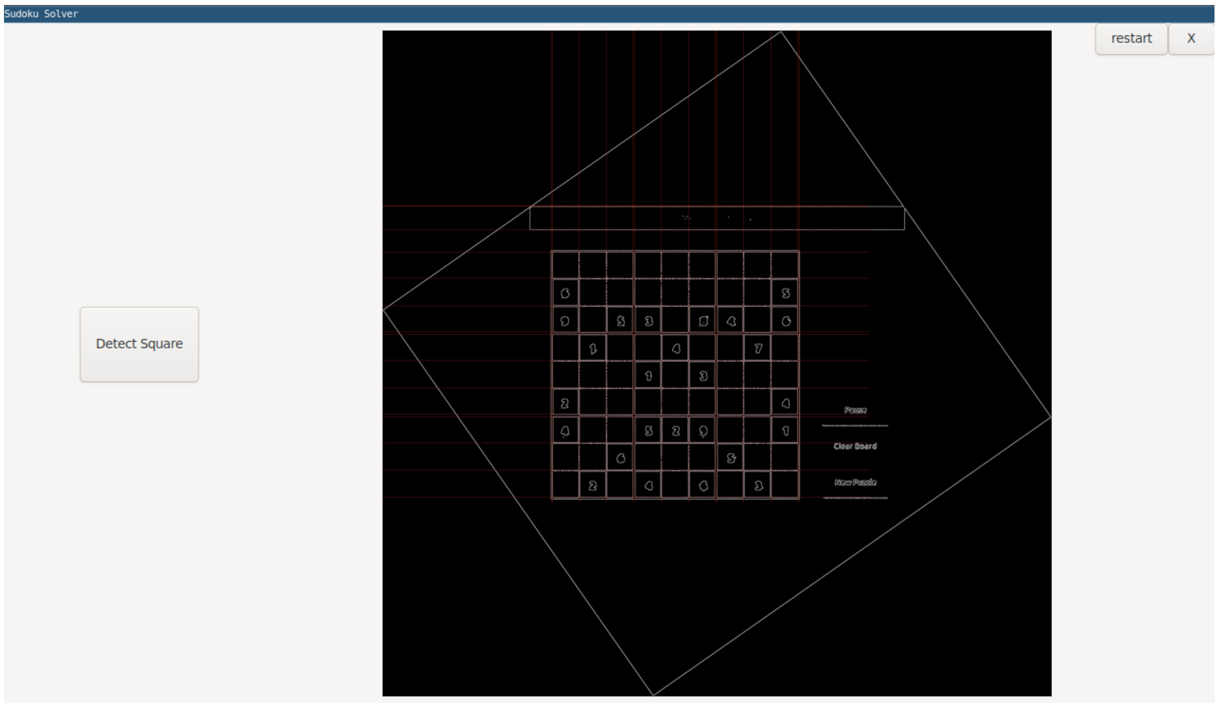


Figure 28: page 2 - the square detection page

9.4 AI and Solving page

This page is here to finally generate an output solved. Clicking on the **Load AI** button launches the AI, then clicking on the **Solve** button launches the Sudoku solving algorithm, creates an output and goes to next page already displaying the input.

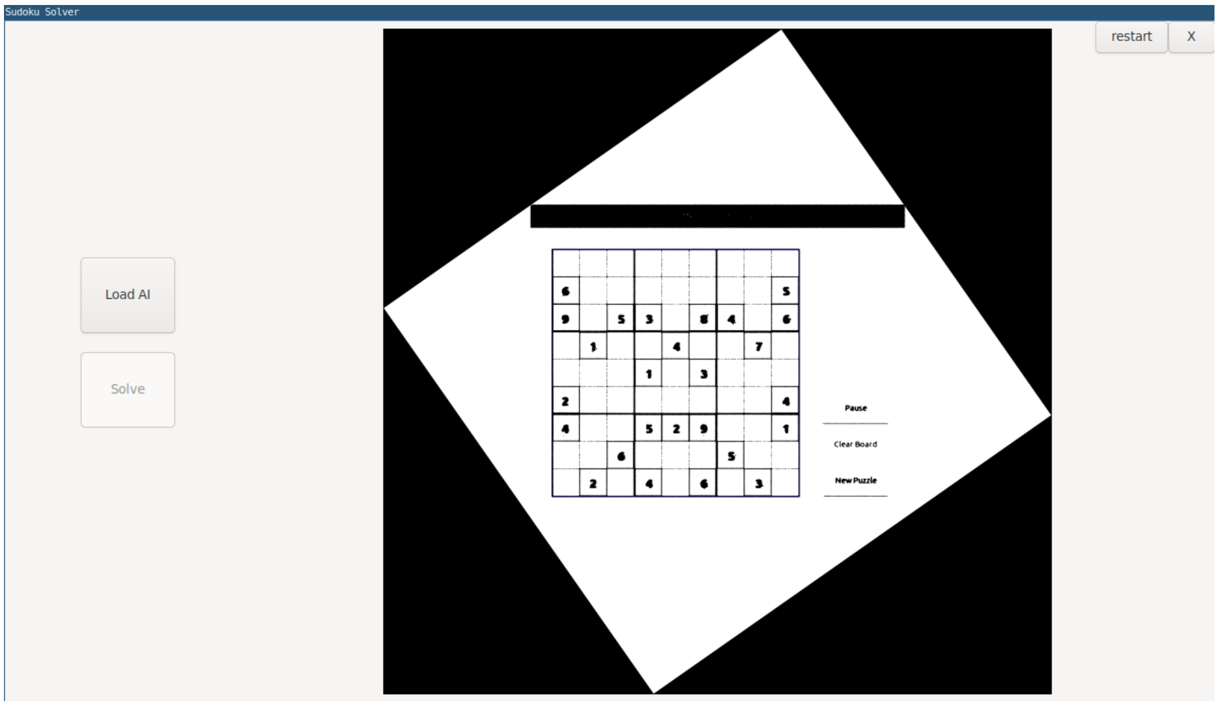
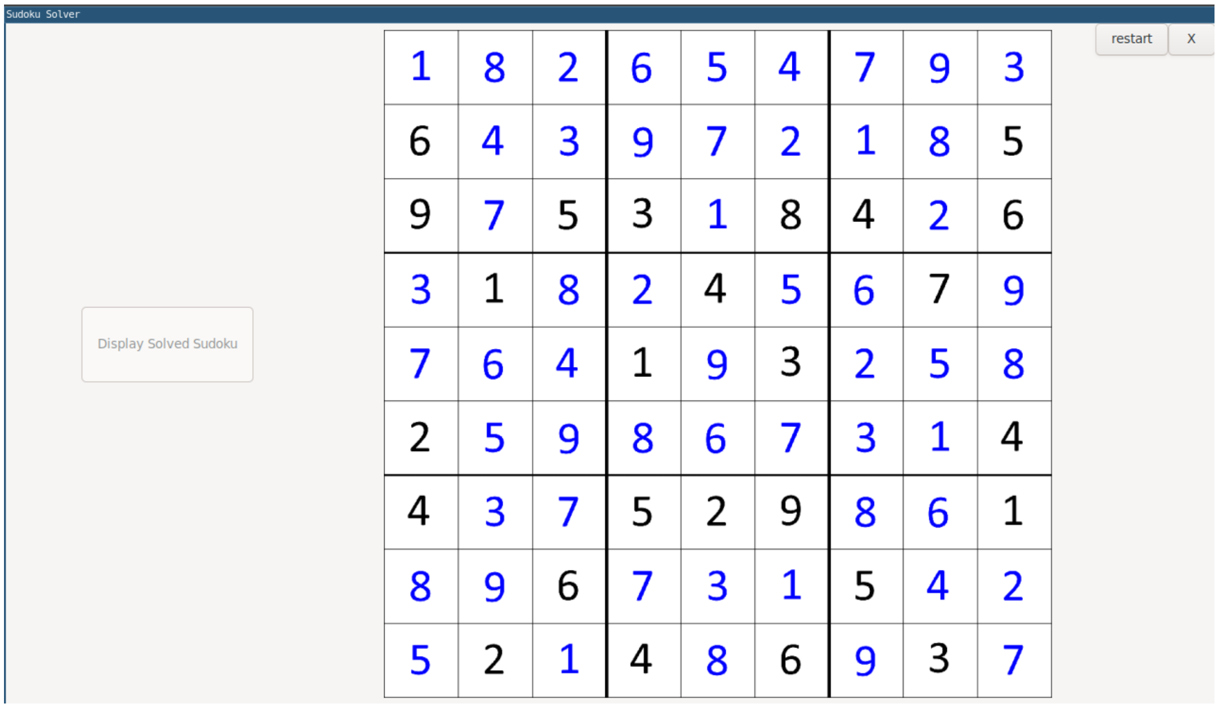


Figure 29: page 3 - the AI and solving page

9.5 Output display page

This page is the final page and it is here to display the final output already created. Clicking on the **Display Solved Sudoku** button passes from the input to the output. The added numbers in the grid are displayed in blue.



1	8	2	6	5	4	7	9	3
6	4	3	9	7	2	1	8	5
9	7	5	3	1	8	4	2	6
3	1	8	2	4	5	6	7	9
7	6	4	1	9	3	2	5	8
2	5	9	8	6	7	3	1	4
4	3	7	5	2	9	8	6	1
8	9	6	7	3	1	5	4	2
5	2	1	4	8	6	9	3	7

Figure 30: page 4 - the output display page

9.6 Additional buttons

The close button **X** is visible in every page, it closes the application. The **restart** button is visible only from page 1 and it actually goes back to this page.

10 Conclusion

Starting this project, we were intimated by all the work that had to be completed. It took us a lot of time, coordination and efforts to get the software we have today and we are proud of the result. In the end, everything planned was completed. This OCR project allowed us to improve our knowledge in programming and obviously in creating a project from scratch. But, let's also not forget that we also improved our skills in working with images, artificial intelligence and in how to tackle and try to solve a problem. It also showed us how important it was to communicate and to work together as a team. As a team, we can complete so much more than by our own self. This project was a great experience.