OCR - Sudoku Solver

SudokUwU
Johan TRAN
Adrian GRILLET
Simon THUAUD

7 December 2021

Responsible
David BOUCHET

Contents

1 Group presentation

1.1 Adrian

Since I entered EPITA, every project I was asked to do was more interesting than the last and this one follows that rule. After reading the project's book of specifications, I knew it was going to be great. The neural network part seemed so interesting I knew that I wanted to do that part.

This feeling quickly faded as getting started was really challenging and it took quite some time to get something going. But as time went by, the neural network started to resemble what you would expect and getting the first results really revived my motivation that was slowly fading away.

With the second defense approaching, I did not realise the amount of work needed to be done so did not start working right after the first defense. At some point I realised and became really stressed out and started staying really late after hours in order to finish the work I had to do.

When we finally finished the OCR, all the weight I had accumulated on my shoulders just vanished instantly. I could not believe that we made it together and felt so happy to be done with it and grateful for all the memories we made along the way.

1.2 Simon

I have always been interested in group projects! To me it represents the best experience of what the real life is. In fact working in group is a simulation of what we will encounter in the future.

In this project I mainly worked on the image-processing, at first I was really confused of what I needed to use, what kind of filter I had to apply to my image and how to do so. I have watched and read a lot of articles about image treatment and it started slowly to be clearer. Image treatment and image-processing is something that has always attracted me, we concluded that I will work on that!

As the end is almost there I can clearly affirm, even though we are late on some parts or it does not work correctly, that we did a great job. We are proud that even with one of our comrades leaving the party during the project creation we managed to have a quite good result. The only part that would have been better is the fact that we started working too late. If we had more time we would have done better and with less stress.

To conclude I am happy with what we have done globally, the team was great and I found the subject interesting. We had way more constraints than the S2 Project and it is also what makes the difficulty of this OCR project. I learned a lot of new things, managing a different type of project, working in C and creating Makefile.

1.3 Johan

As far as I can remember, I always play and have fun with new technologies. First, I used them for their main goal. However, when I was bored of the object, I tried to push it over its limit. I was truly fascinating but I broke it after that. Even if I did not know what I wanted to do when I was younger, I knew that I have this attraction to computer science and new technologies in general.

When I discovered EPITA, it was for me a school that had everything I wanted to do and I naturally join the school in the English class because it's mandatory for the future. In general, I'm a person that feel really frustrated when I do not succeed to do something I'm really into, it's why I try to always give my best when I want to succeed.
Finally, I am really happy to have done this project, I was in charged of the splitting and the square detection. I think this part was hard because we did not have many hints to start the detection of the square but it feels really rewarding when we succeed to do one step of the detection. And I was so happy when the last functions works.

In the end, this project was satisfying to do and I proud of our group for what we have done. I've learned many things and I think my understanding of the C language greatly increased thanks to OCR.

1.4 Virgile

During this year I was expecting to do the same project as the past year (a text recognition OCR). I was pleasantly surprised to learn that it changed and is now an OCR Sudoku solver, which is very similar, but I find the Sudoku component more interesting than plain text. I was very happy to work on another project than a video game like we did with my group in S2 and have a more technical experience. When doing the task splitting, We agreed that I would do the Sudoku solver algorithm for the first defense. I also plan on finding ways to improve the solving algorithm I did.

Virgile did not participate for this second defense, because of personal issues he needed to leave.

2 Task distribution

2.1 First defense

At first, we thought that the neural network would be the hardest part so Adrian and Johan started working on it together. But as time passed, we realized how challenging was the image processing part and so Johan joined Simon on this dreadful task. Virgile was in charge of the Sudoku solver and the file reading / writing. This is how the tasks were distributed:

	Simon	Johan	Adrian	Virgile
Image treatment	X			
Image splitting		X		
Neural network			X	
Sudoku solver				X

2.2 Second defense

The second defense schedule was easier to set in place. Actually we lost Virgile at the beginning of the second part of the project so we needed to reorganise a little bit what we decided and organised at the beginning of the project. So the distribution was as follows: Adrian was still in charge of the Neural Network, he upgraded the one it already to make it works with digits. Johan finished the part on Hough Transform and images splitting. And Simon was in charge of the graphical user interface.

	Simon	Johan	Adrian
Image splitting		X	
Digits recognition			X
GUI and filters	X		

3 Progress Made

3.1 Sudoku solver

As we have seen in this project, we need to solve a Sudoku from a given image. We first had to create a program to solve Sudoku, once it's been transformed into a readable text file. As shown in 4.3 figure 8. This implementation with periods made it difficult to handle the transfer from text to the integer array used in the solving algorithm. We implemented this program in C as the rest of the project.

This part did not change for the second defense we just modified a few things to link it with the rest of the work. The input grid and output grid were used to create the output display.

3.2 Image processing

Image processing is an important of this OCR project. Since we need to detect a sudoku grid, our image needs to perfect. In order to do that we apply a lot of different filters. Actually we have all of them:

- Gamma correction
- Constrast correction
- Grayscale filter
- Adaptative treshold (image binarization)
- Noise correction (median filter)

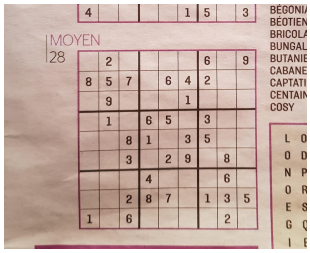
We also have the manual rotation which allows us to rotate manually with a certain angle an image. For example the image 5 need to be rotate by 35 degrees.

In order to have the Hough Transform works better we needed to implement the Sobel filter, filter used to detect edges in the image. It works by calculating the gradient of image intensity at each pixel within the image. It finds the direction of the largest increase from light to dark and the rate of change in that direction.

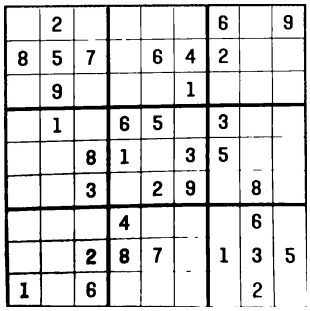
The image processing is not perfect, in order to have it works with every images with would have need to implement a perspective algorithm. For example the image 6 as the lower right corner oriented in a way that Sudoku's borders are not straight. Then Hough Transform is not working and by definition we can not solve the Sudoku...

3.3 Image Splitting

Image splitting is a main part of the project, because it's where we detect the grid and we cut it to send information to our Neural Network. We have completely finish the splitting and we can automatically detect the square to crop for our Neural Network here is the final result for some images :

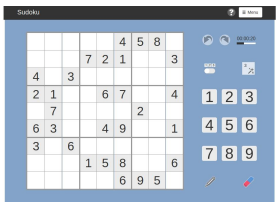


(a) Before processing

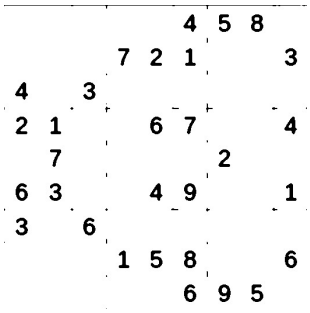


(b) After processing

Figure 1. Image 2

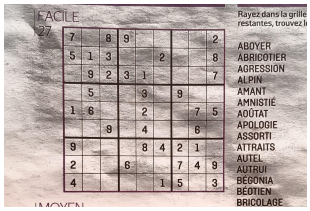


(a) Before processing

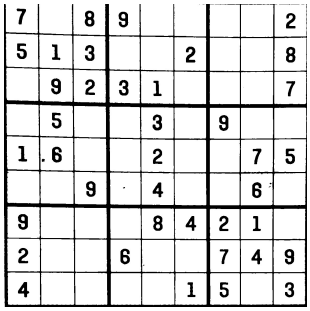


(b) After processing

Figure 2. Image 3

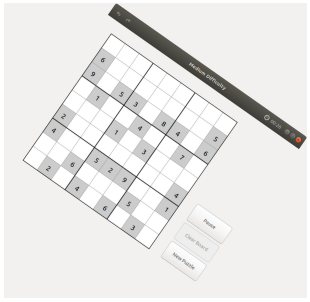


(a) Before processing

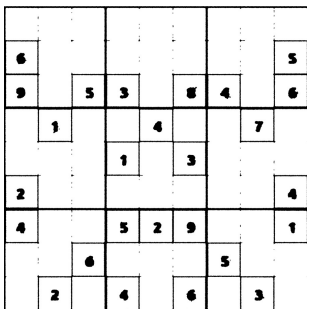


(b) After processing

Figure 3. Image 4



(a) Before processing



(b) After processing

Figure 4. Image 4

3.4 Neural Network

3.4.1 XOR gate

For now, we have a program that after inputting all the needed information, is able to learn anything (limited by size since everything needs to be inputted by hand).

The inputs needed (for the XOR) are as follow:

- Number of layers (3)

- Number of neurons per layer (2, 4, 1)

- Learning rate (0.15)

- Number of training examples (4)

- Inputs for all the training examples (0 0, 0 1, 1 0, 1 1)

- Desired outputs for all the training examples (0, 1, 1, 0)

Then the program should train the network for 20000 epochs and inform you when it is finished. It then lets you test the results if you input an example. You can therefore check if the training has been successful.

3.4.2 Digit recognition

For the network to be able to decipher which digit it was given an image of, the structure of the network needed to be modified. Now we have a network which has 3 layers, an input layer which has as many neurons as pixels in the image it is given, 1 hidden layer and 1 output layer which has 10 neurons for numbers from 0 to 9.

When executing the network in order for it to train, it gets random images from a folder and knows what output corresponds to the image. This goes on for a while until we are satisfied with the precision of the network. Then, the weights and biases are saved in a text file ready for anyone to use and have good results without even training the network.

4 Technical Aspects

4.1 Image processing first defense

The loaded image goes through a lot of different algorithm in order to have the best image for the grid detection and the grid splitting. It goes through all the filter below:

4.1.1 Gamme filter

Figure 5. Gamma correction

The first filter applied to the image is the gamma filter. It might be useless but in fact it has a great impact on the noise reducing and the edge detection. Gamma is an important but seldom understood characteristic of virtually all digital imaging systems. It defines the relationship between a pixel's numerical value and its actual luminance. We calculate the new R, G and B value by using the following formula:

$$I' = 255 * \left(\frac{I}{255}\right)^{\gamma}$$

In this formula I' will be the new pixel color value (i.e: We change the red value, I' will be the new number of red component and I is the one before the calculation, gamma is the power of the gamma that we want to apply).

4.1.2 Contrast filter

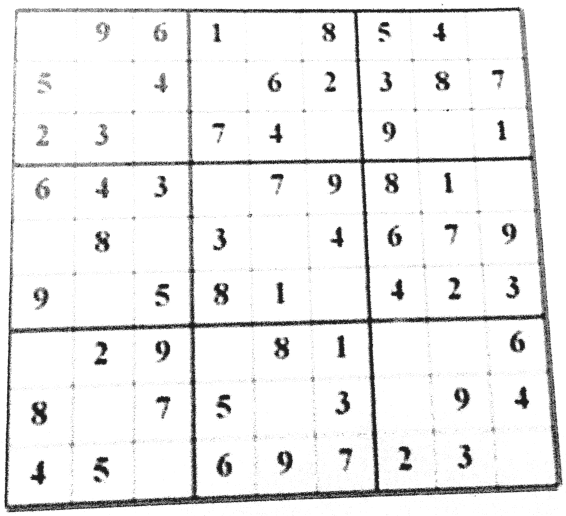


Figure 6. Contrast correction

Sometimes it is desirable to alter the contrast of an image, either to darken faint text or to lighten a dark background, and increase the readability of the data on the image. Contrast is used here to bring out the contours. We use a mathematical formula to compute the contrast correction:

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

The is the factor (the intensity of the contrast we want). It will be used to modify the value of the red pixel with the formula:

$$R' = F * (R - 128) + 128$$

The R (red) value of the current pixel will be change by R'. The combination of these two process gives us a more detailed image, edges are more defined and we have a little less of noise. In fact it just prepare the image for what is coming!

4.1.3 Grayscale filter

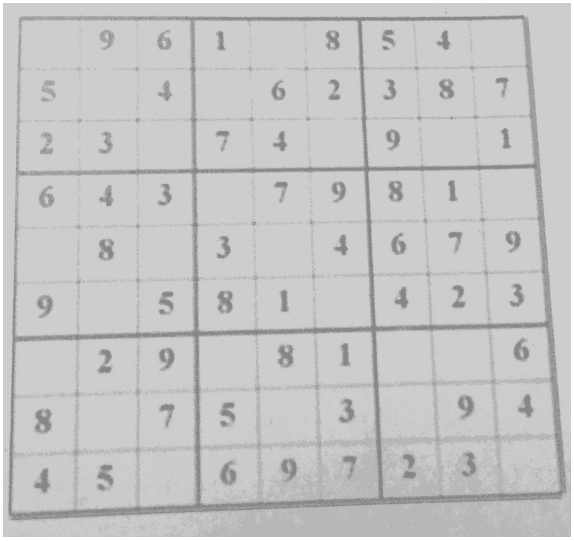


Figure 7. Grayscale correction

The grayscale filter is important for the next filter. Because we compare only one color component of each pixel in the adaptive threshold.

$$average = 0.3 * r + 0.59 * g + 0.11 * b$$

This average value will remplace all the RGB values of the current pixel, which will turn it gray. We use this formula instead of the basic one because by only doing the sum of the RGB component divided by 3 we can have an overflow. But it is also due to the fact that we are more sensitive to green light, less sensitive to red light, and the least sensitive to blue light. That is why we use the weighted method!

4.1.4 Adaptive threshold

For the binarization of the image we do not use a basic black and white algorithm. We use an adaptive threshold technique which consist of: the adaptive threshold technique is a simple extension of Wellner's method. The main idea in Wellner's algorithm is that each pixel is compared to an average of the surrounding pixels. Specifically, an approximate moving average of the last s pixels seen is calculated while traversing the image. If the value of the current pixel is t percent lower than the average then it is set to black, otherwise it is set to white.

This method works because comparing a pixel to the average of nearby pixels will preserve hard contrast lines and ignore soft gradient changes. The advantage of this method is that only a single pass through the image is required. Wellner uses 1/8t of the image width for the value of s and 15 for the value of t.

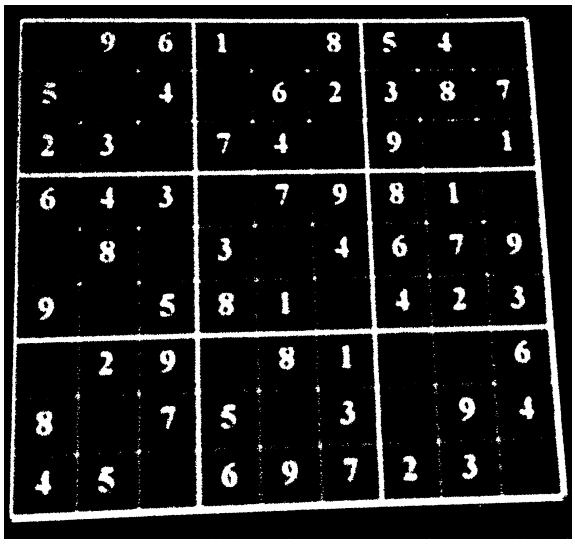


Figure 8. Black and white correction (adaptive threshold)

This method uses a two dimensional array in which we store the sum of the current pixel red component value and the last cell in the two dimensional array. And we have then a comparison between the current pixel and the value in the two dimensional array.

4.1.5 Median filter

To finish, we apply a basic median filter to all the image. It really is one of the easiest algorithm for noise reduction.

$$\begin{matrix} 6 & 7 & 16 \\ 22 & Pixel & 67 \\ 7 & 12 & 2 \end{matrix}$$

The matrix above is just a representation of the current pixel value and the values around. The process is as follow: we take all the 8 pixels value surrounding our current pixel. We store them all in an array which is then sorted in increasing order. We take the median value and we replace our actual current pixel value by the median one.

It has not such a big impact on the final image since it only takes the 8 surrounding pixel but it smooth the image. We will try to implement a better noise reducer algorithm for the next defense.

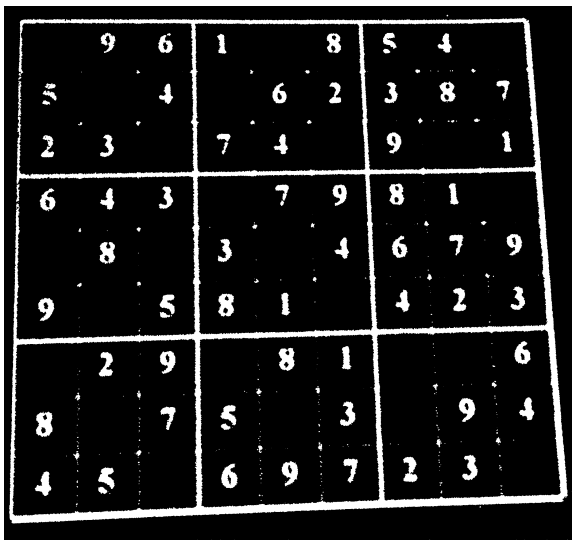


Figure 9. Noise correction

The median filter is a non-linear digital filtering technique, often used to remove noise from an image or signal. Such noise reduction is a typical pre-processing step to improve the results of later processing (for example, edge detection on an image). Median filtering is very widely used in digital image processing because, under certain conditions, it preserves edges while removing noise (but see the discussion below), also having applications in signal processing.

4.2 Image processing second defense

The image processing has evolved quite a bit since last time. In fact we are still using our previous but we upgraded them or we have added some to have a better result.

4.2.1 Gaussian Blur

The first filter that we added is Gaussian Blur. In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function (named after mathematician and scientist Carl Friedrich Gauss).

It is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination.

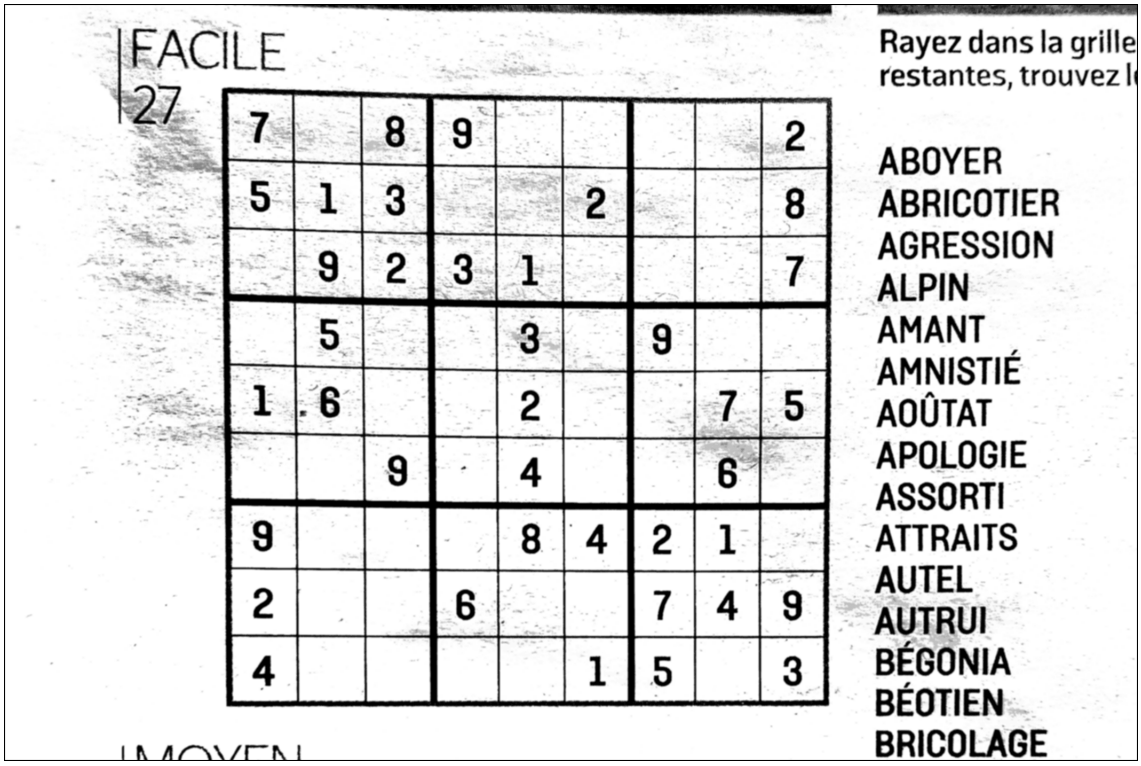


Figure 10. Gauss correction

Gaussian smoothing is commonly used with edge detection. Most edge-detection algorithms are sensitive to noise.Using a Gaussian Blur filter before edge detection aims to reduce the level of noise in the image, which improves the result of the following edge-detection algorithm.

4.2.2 Otsu threshold

Secondly, we changed our threshold method. Previously we were using the adaptive threshold method but was still too noisy. We look for different algorithm and Otsu was the best we have found. The process of separating the foreground pixels from the background is called threshold. There are many ways of achieving optimal threshold and one of the ways is called the Otsu's method. Otsu's method is a variance-based technique to find the threshold value where the weighted variance between the foreground and background pixels is the least. The key idea here is to iterate through all the possible values of threshold and measure the spread of background and foreground pixels. Then find the threshold where the spread is least.

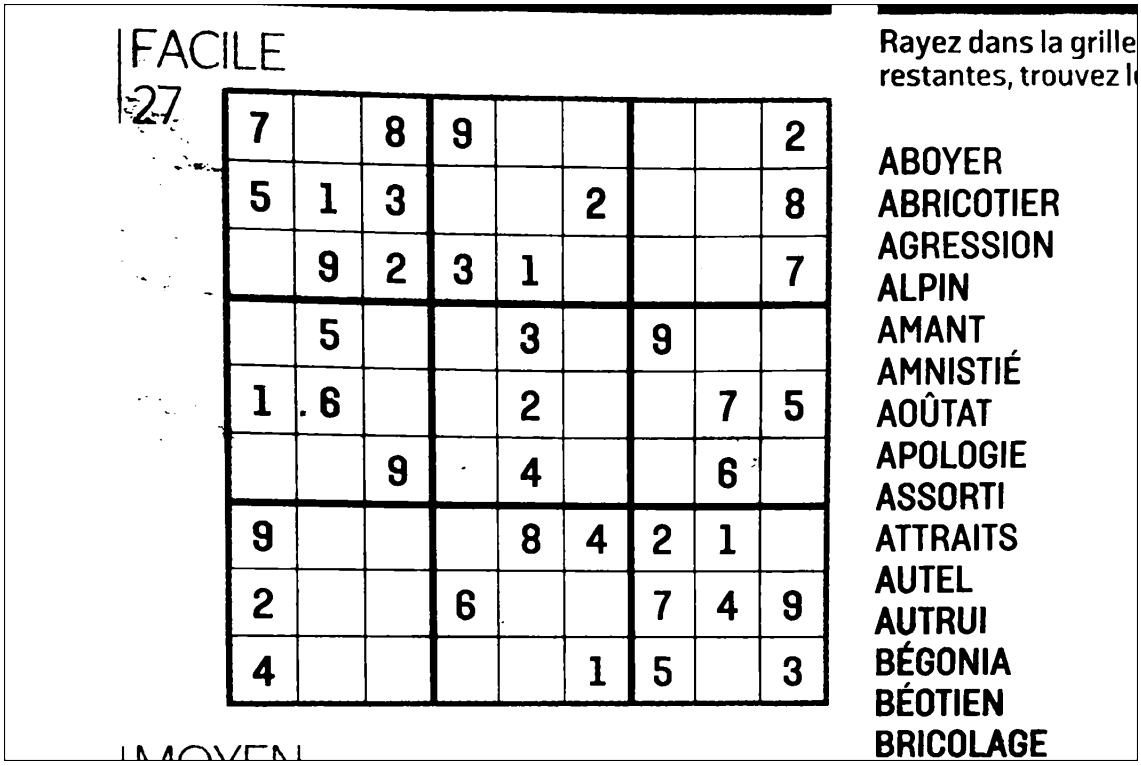


Figure 11. Otsu threshold

The algorithm works as follow: it iteratively searches for the threshold that minimizes the within-class variance, defined as a weighted sum of variances of the two classes (background and foreground). The colors in grayscale are usually between 0-255 (0-1 in case of float). So, If we choose a threshold of 100, then all the pixels with values less than 100 becomes the background and all pixels with values greater than or equal to 100 becomes the foreground of the image.

4.2.3 Sobel filter

Finally we applied the Sobel filter. We thought that with all we had already done our pre-processing would be enough but it was not working for images like the third one. We needed sharp edges in order to have Hough working on every type of images. That is why we use Sobel!

Sobel filter is used for edge detection. It works by calculating the gradient of image intensity at each pixel within the image. It finds the direction of the largest increase from light to dark and the rate of change in that direction. The result shows how abruptly or smoothly the image changes at each pixel, and therefore how likely it is that that pixel represents an edge.

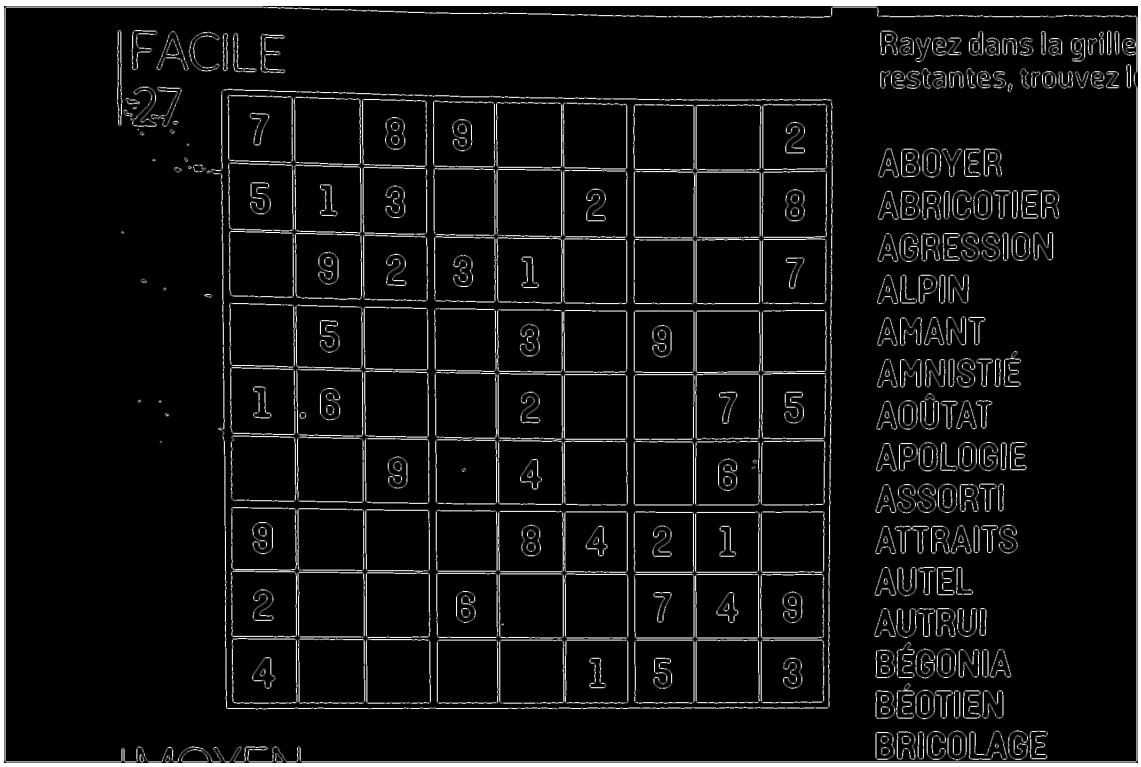


Figure 12. Sobel edges detection

The sobel filter uses two 3 x 3 kernels. One for changes in the horizontal direction, and one for changes in the vertical direction.The two kernels are convolved with the original image to calculate the approximations of the derivatives.

Sobel was really helpful for one image. Our image 3 edges were not recognizable by Hough Transform but now with sobel it can.

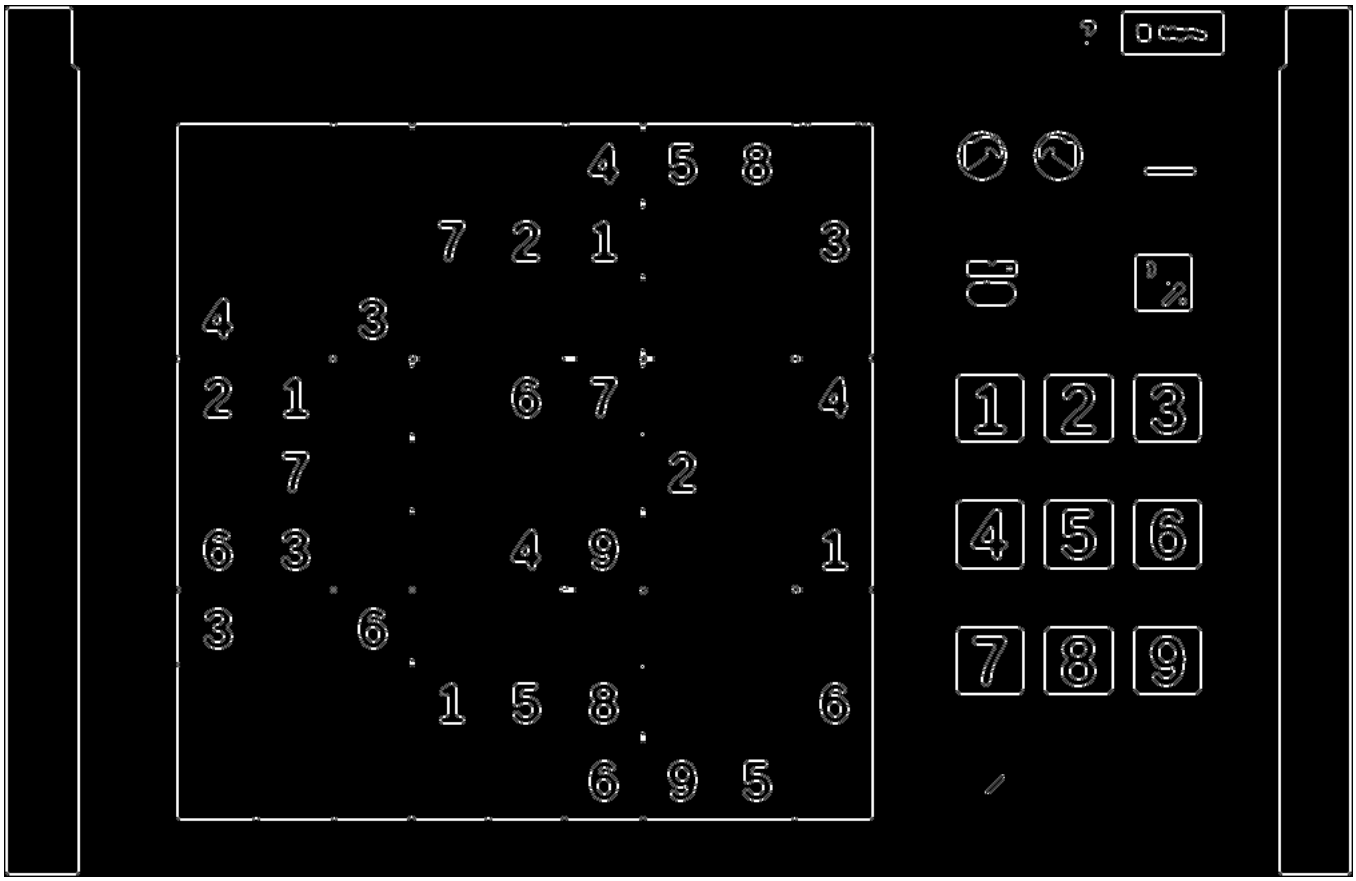


Figure 13. Sobel edges detection on image 3

4.3 Image splitting

4.3.1 Hough Transformation

We can divide the image splitting in two parts:

- The first one is the edge detection. We use Hough Transform to create lines that will be used for detecting the edge. In theory, almost every line of the form y = ax + b is represented as a point in the Hough Space (Horizontal axis is the slope and vertical axis represent the intercept) (a,b). The major issue with representing line with the equation y = ax + b is when there's a vertical edge. At this moment "a" cannot be determined. To solve this problem you have to switch to polar space (rho,theta).

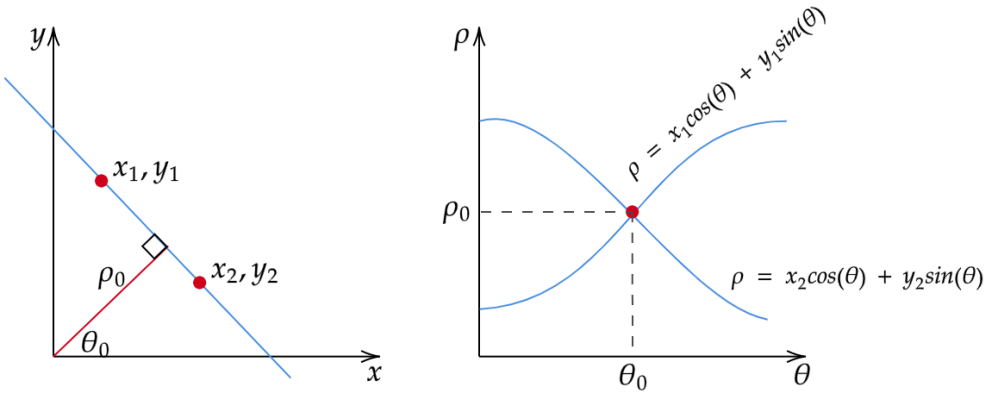


Figure 14. Two ways of representing a line in a space

Now that we have different coordinates in our polar space, we can detect lines that have an amount of intersections greater than a limit fix.

Let's dive into the algorithm:

First, we have to decide of a range for rho and theta, (theta is often between [0,180] and rho is between [-d,d] with d the length of the image diagonal.

With an accumulator that's represented by a 2D array of dimension (num-rhos, num-thetas) each time you detect an edge pixel, you find the values rho and theta and you can increment at the index corresponding.

4.3.2 Automatic rotate

With the rhos and thetas that we get, it becomes easy to find a way to rotate our image automatically. Firstly, we go through our accumulator and when a coordinate is greater than a threshold we increment a temporary value. We search for the biggest temporary value between 0 - 45 and 135 - 180 they will correspond to our horizontal lines and between 45 - 135 they will correspond to our vertical lines. We then call our manual rotate to apply the rotation on the image with the value for the angle the max theta for ours horizontal lines

4.3.3 Square detection

Now to draw our lines we do a traversal of our accumulator. We firstly make a research for local maximums to draw less lines, when we have have less values. This is were we do our threshold to draw lines. We decided to draw only lines that are horizontal or vertical because our image is supposed to be automatically rotated in the good representation.

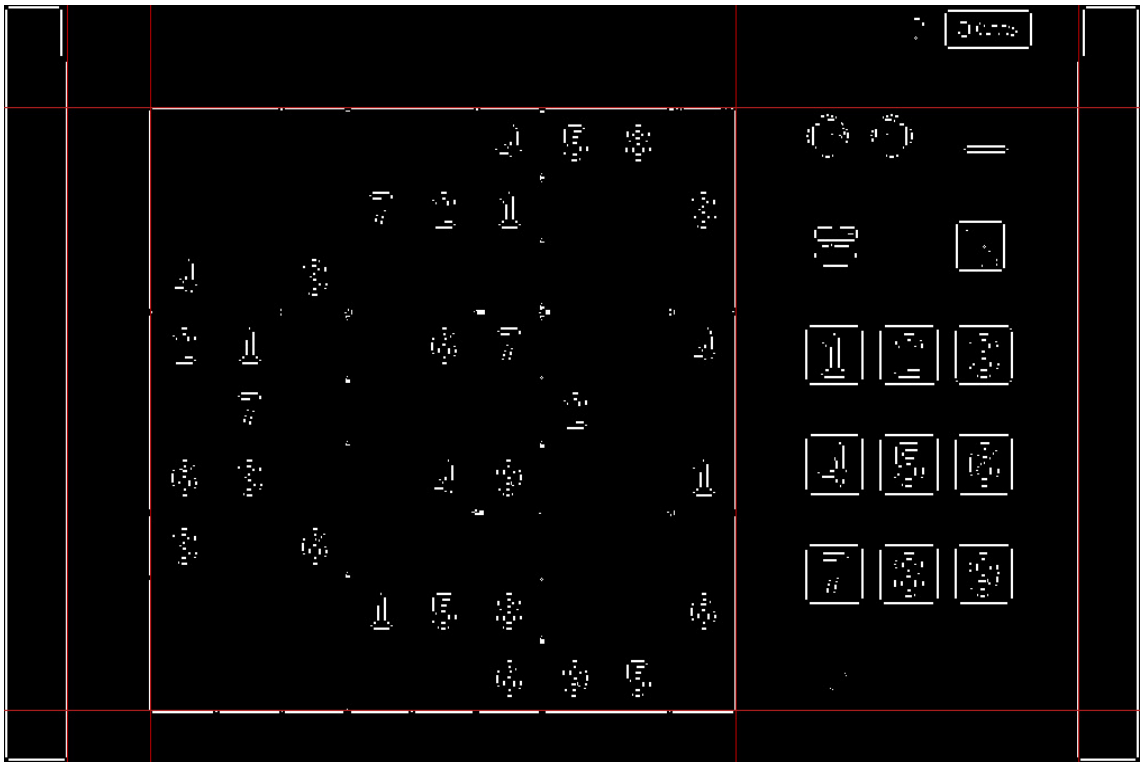


Figure 15. Hough Transform

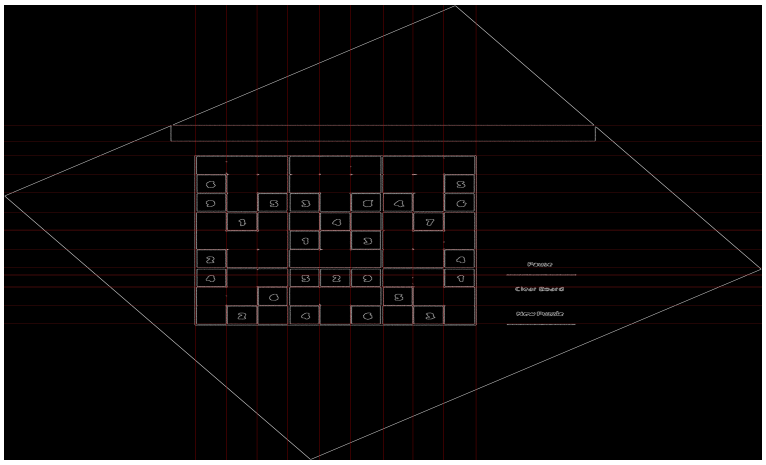


Figure 16. Hough Transform 2

We can now detect the position of each intersection point and have the grid of the Sudoku. It is done when we draw lines when we put a pixel if we find a pixel that have the same color of the line, we store it in a new image.

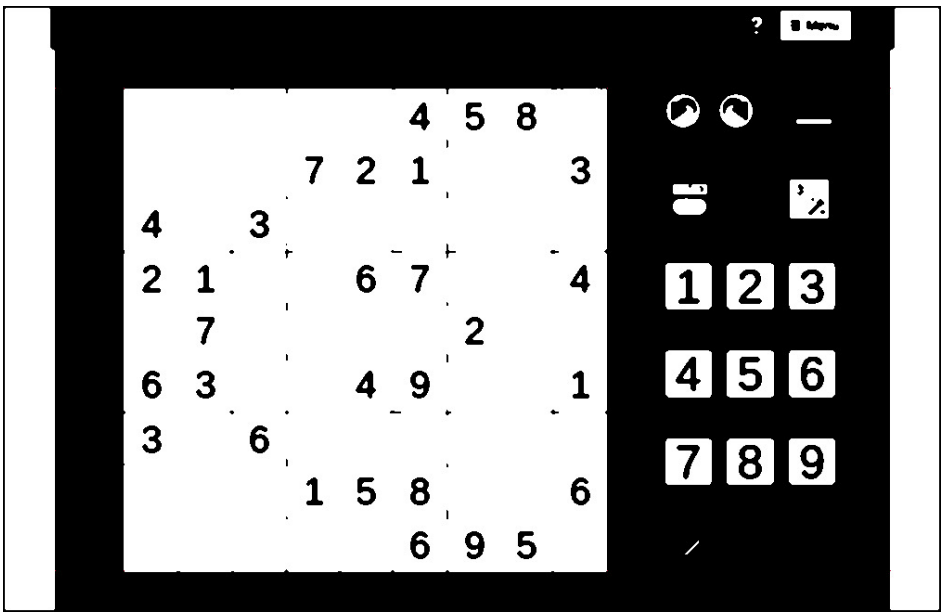


Figure 17. Intersection image 3

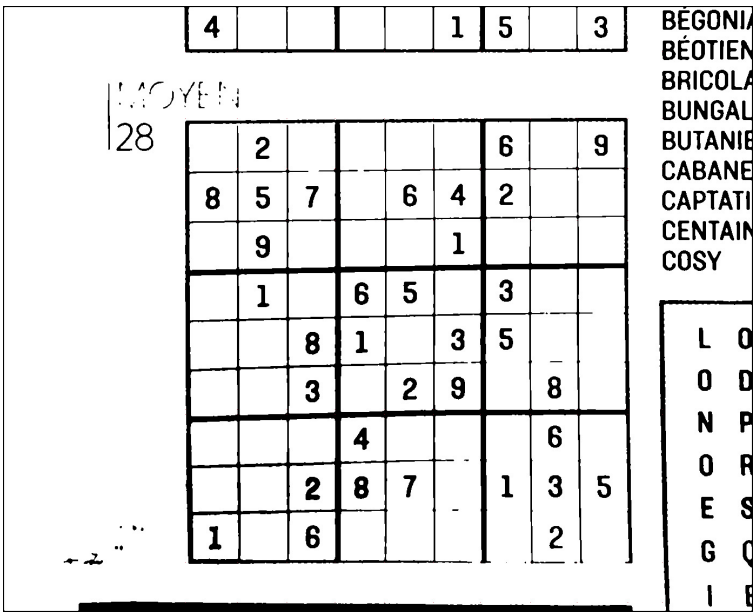


Figure 18. Intersection image 2

After finding the intersections we have to find the biggest square between all intersections. We take each intersections on the image and we check on the axis x and y if there's an other intersection. If we detect one, we add it in a list with the length between the two intersections. When there's two pixels at the same length plus or minus ten pixels, we add the coordinate of the intersection and the length. After, we search for the biggest length and we use it localisation and the length to crop thanks to the SDL fonction blit.

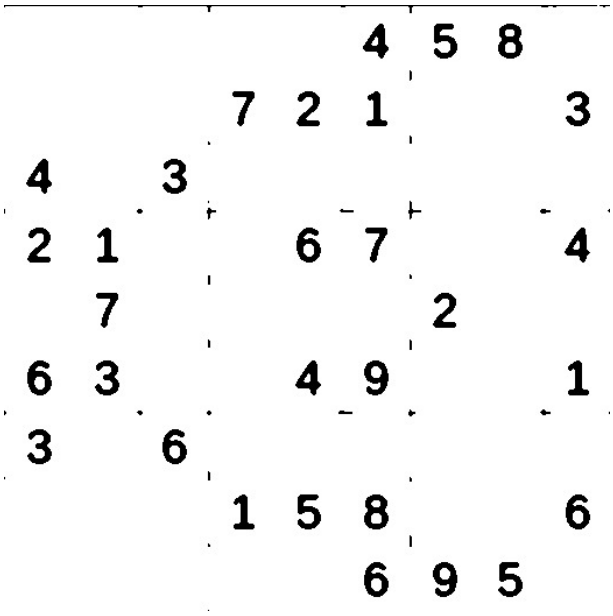


Figure 19. Final square image 3

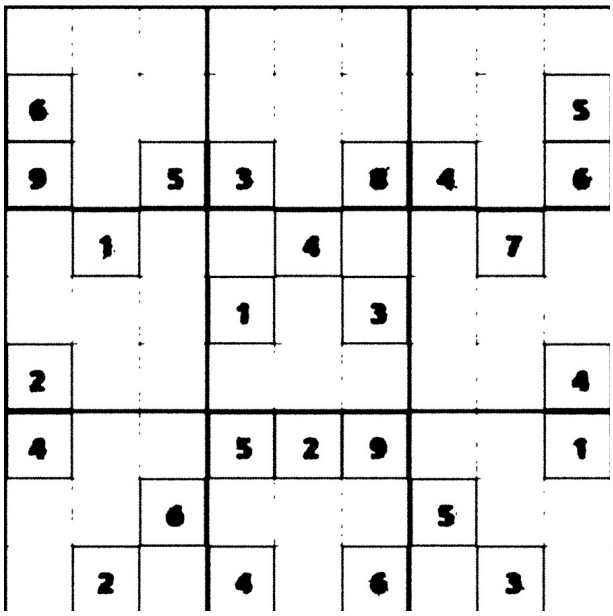
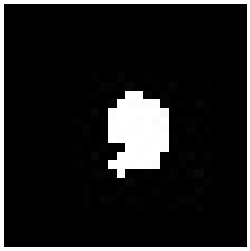


Figure 20. Final square image 5

(a) Case without processing



(b) Case processing

Figure 21. Transition from

4.3.4 Splitting

- We can now crop the grid into 81 cases to separate each number of the grid. We save them, with 81 names that are always name the same in the same order to facilitate the neural network and the creation of the Sudoku in the file. But we need to apply some processing on each case in order for our Neural Network to work properly. article [demo]graphicx caption subcaption

4.4 Neural Network

4.4.1 First defense

In order to recognize the digits that are already present on the grid, a neural network will be needed. Indeed even if you and I can easily recognize the digits, it is significantly harder for a machine to do so. After the image is processed, the pixels can be either black or white which is like 1s and 0s.

Depending on the size of the picture, we will have a different number of inputs corresponding to the number of pixels. To make the calculations smoother, we will implement a hidden layer in the network. Then, the output will be 10 numbers, either 1 or 0. If the network works, inputting the picture of a 1 will return the following output : 0100000000 (approximately, values will be close to 1 and 0 with a unnoticeable margin for error i.e. 0.998, 0.006).

The activation function we are using for this network is the sigmoid function for the output layer:

$$S(z) = 1 / 1+e^{-z}$$

For hidden layers, the Relu function will be used:

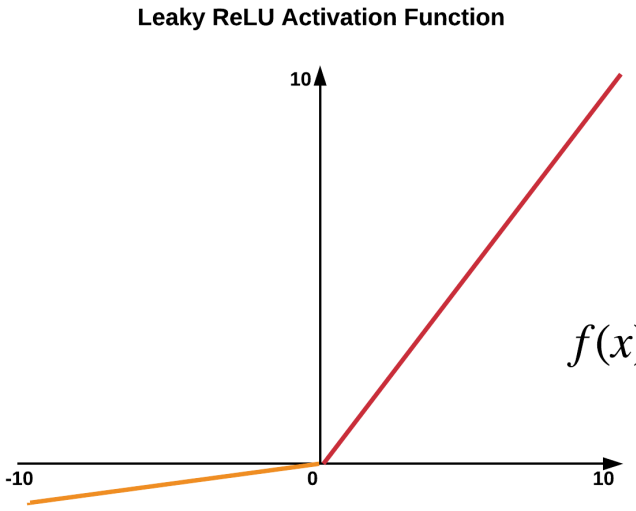
$$R(z) = \begin{cases} z & z > 0 \\ 0 & z <= 0 \end{cases}$$

4.4.2 Second defense

After trying out to recognize the numbers with our existing network, we realised that the sigmoid function, while being an excellent choice for yes or no networks, when the amount of possible answers exceeds 2, it will not be great in terms of precision and will quickly reach its maximum at about 50-60 percent of accuracy. For the second defense, we changed the activation function for the output layer and opted for the softmax function :

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K.$$

After changing from sigmoid to softmax, the results were good but they needed to be perfect. We then decided to change the activation function for the hidden layers and decided to go with the leaky relu function which looks like this:



$$f(x) = \begin{cases} 0.01x & \text{for x} < 0 \\ x & \text{for x} \geq 0 \end{cases}$$

4.5 Sudoku solving algorithm

The solver takes only a file name as input, the file itself contains the Sudoku that we want to solve, with periods representing boxes that we have to fill.

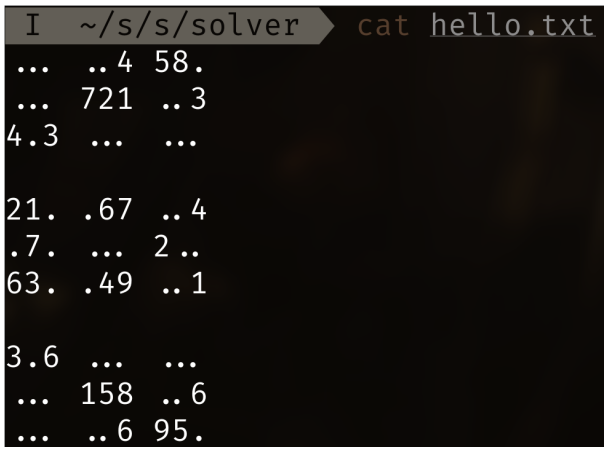


Figure 22. Input file

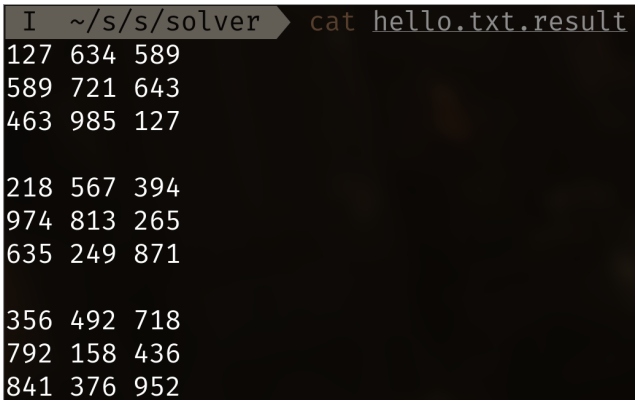


Figure 23. Output file

We must subsequently open this file and transform it into an array of integers, to facilitate solving the Sudoku.
We do this by first allocating memory for the 9 by 9 integer array and reading each character from the input file into the array, skipping newlines and spaces.

We also make sure to transform the periods into the integer (-1), that will represent an empty space in our array.

We then use a recursive algorithm to solve the integer array, and then write it to a file we created, using the original file name followed by ".result"

This part has not changed at lot for the second defense.

4.6 Creating the output image

This section was not so hard to implement, it just took a bit of time to think about it and to have something pretty. We decided to separate the digits that were already in the Sudoku from those that we add with the solver.

4.6.1 Base grid

We have a sample of an empty grid that will be fill with digits after. The grid is 1800x1800 pixels and the little digits are 150x150 pixels each.

The grid is as follow:

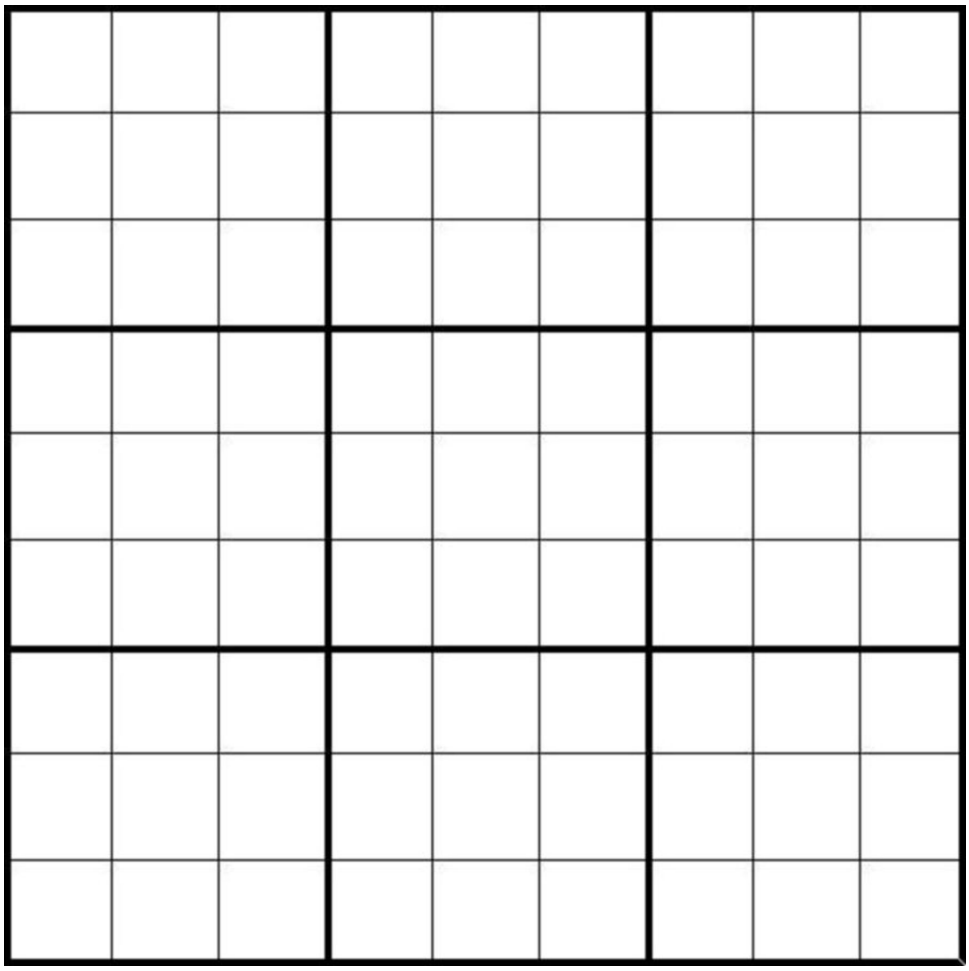
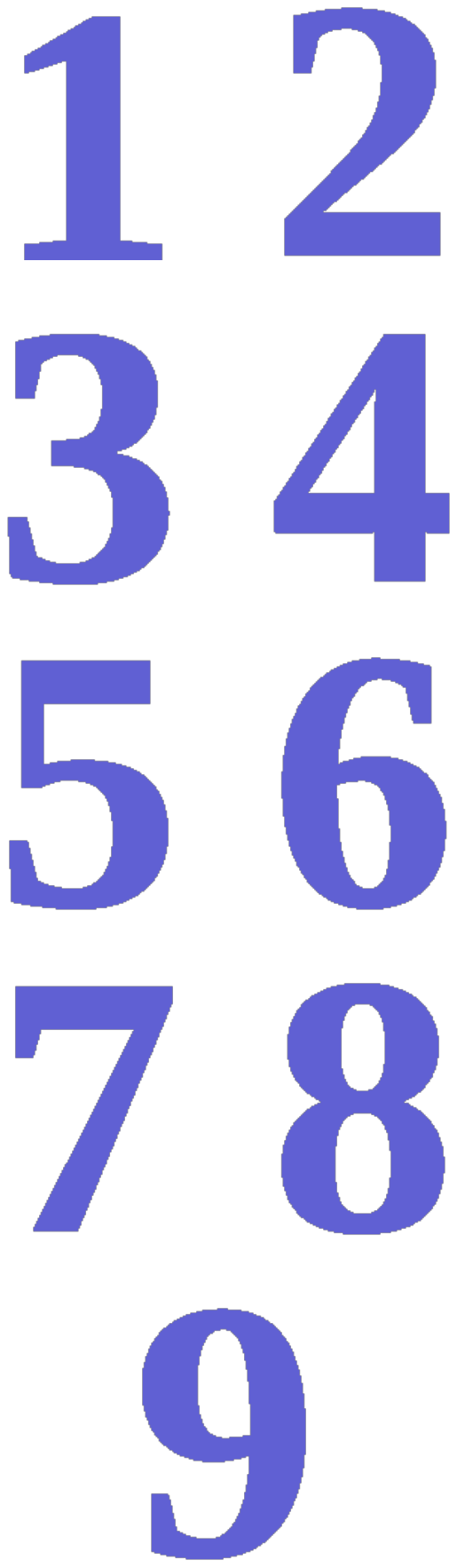


Figure 24. Empty grid

4.6.2 Digits

The following images will be integrated to this grid:


1 2
3 4
5 6
7 8
9

Here some examples of what it looks like after solving the grid:

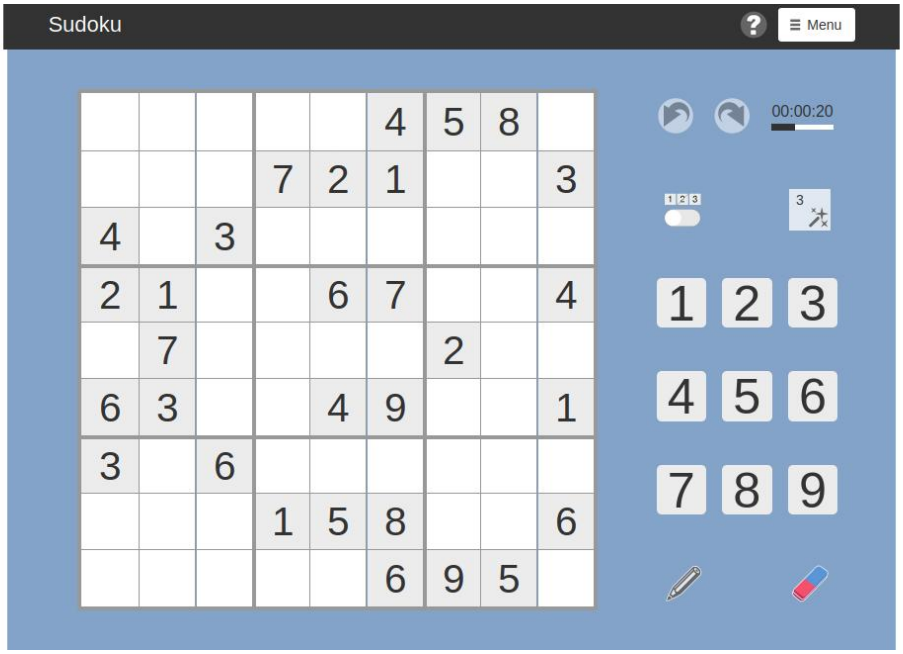


Figure 25. Original grid (3)

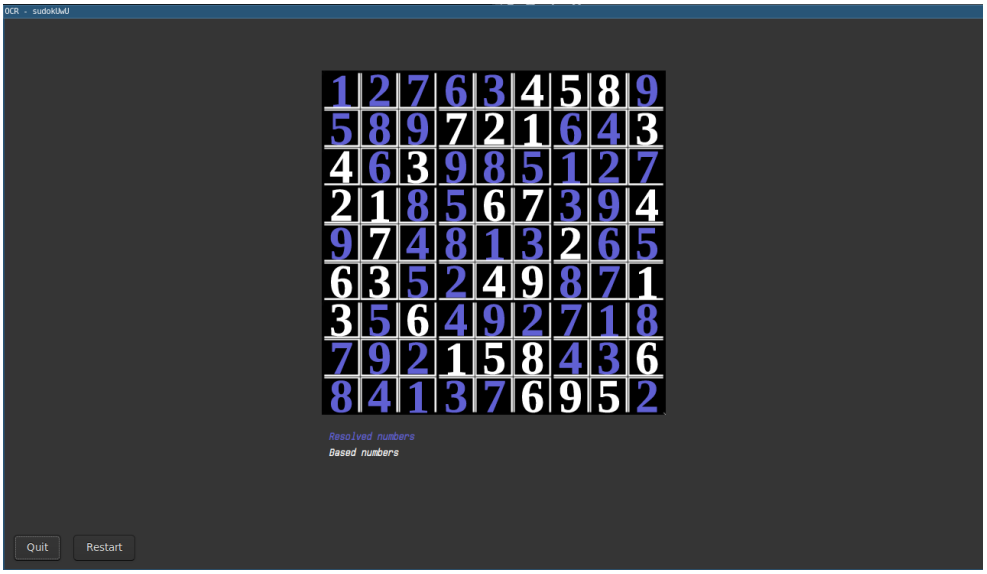


Figure 26. After solving grid 3

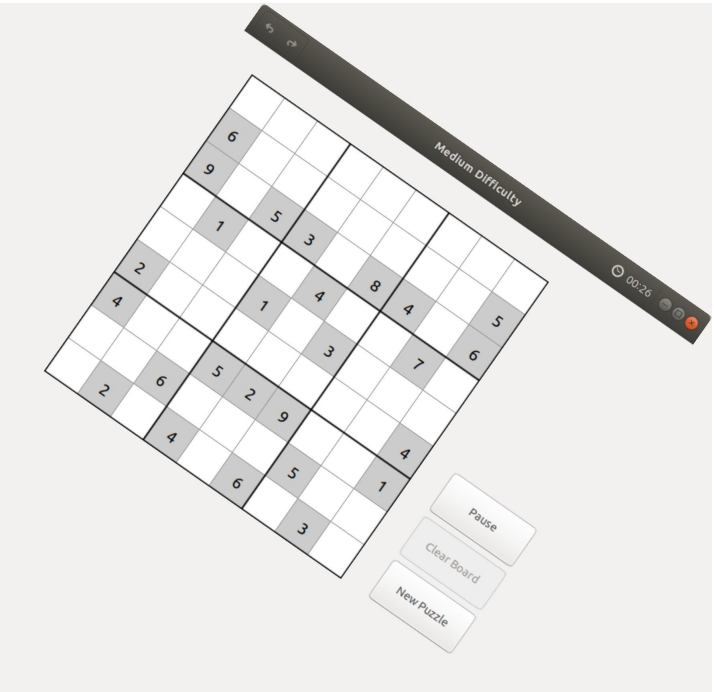


Figure 27. Original grid (5)

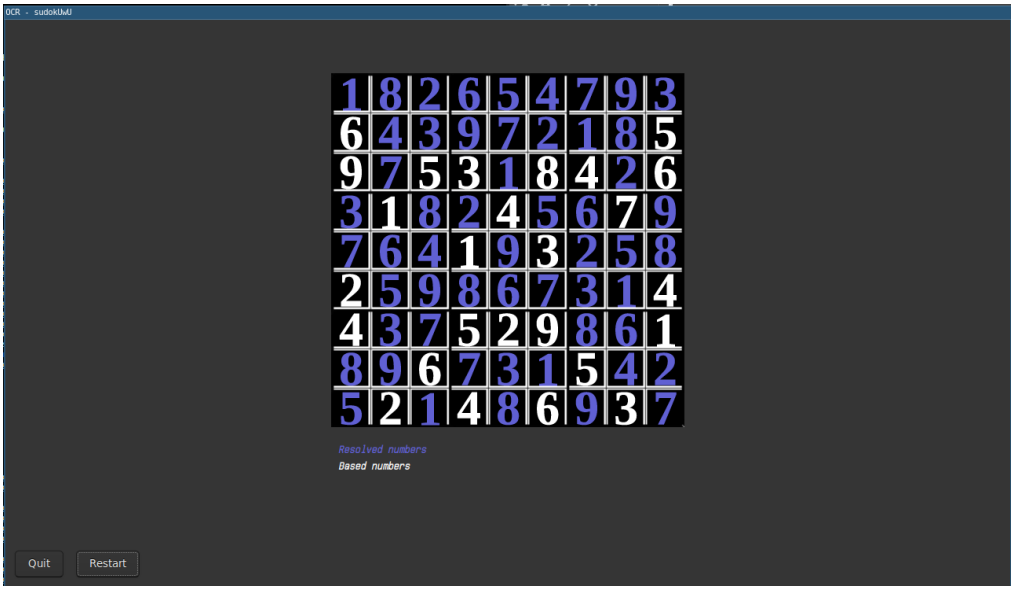


Figure 28. After solving grid 5

4.7 Graphical user interface

The GUI is the main part of the project. It has to link every other sub-parts and make it works well. We decided to use the Glade tool and we then link everything in our main-gui.c using C language. Instead of having something only practical we decided to make it pleasant to use. We spent a lot of time doing the functional section but also the graphical one.

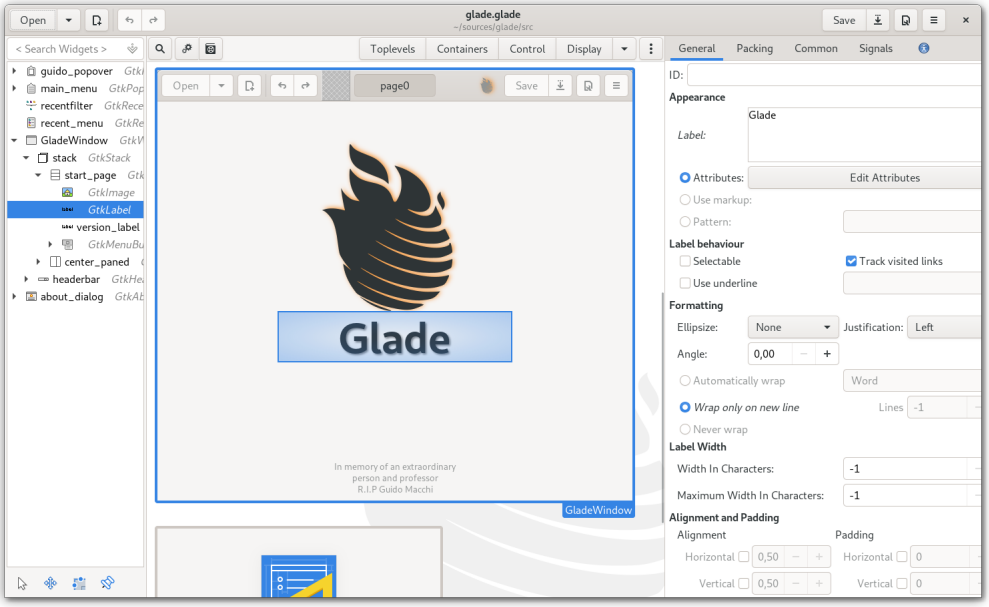


Figure 29. Glade interface

Glade is a RAD tool to enable quick easy development of user interfaces for the GTK toolkit and the GNOME desktop environment. The user interfaces designed in Glade are saved as XML, and by using the GtkBuilder GTK object these can be loaded by applications dynamically as needed. It is loaded and build in our main program, with all that set up we can update our pages, images and all the stuff we need!

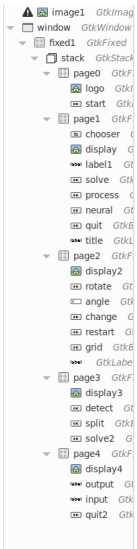


Figure 30. Our glade architecture

4.7.1 Main page

Our main page is only for aesthetic purposes, in fact it does nothing if we speak about OCR but we found it good to have one in order to make the application a bit more user friendly!

Figure 31. GUI main page

This page has only two elements, the first one is an **GtkImage*** which is just here as a container for our logo SUDOKU. This image is defined in glade directly and the source file is in the src directory of the project.

Then we set a **GtkButton*** with label "Launch the solver!", it will only redirect the user to second page of the stack.

4.7.2 Neural Network page

We decided to let the user test if the number is recognizable by the neural network. To do so we have a page where you can load a digit and then you will have the output gave by the neural network. If image is blank nothing is find!

Figure 32. A digit was found!

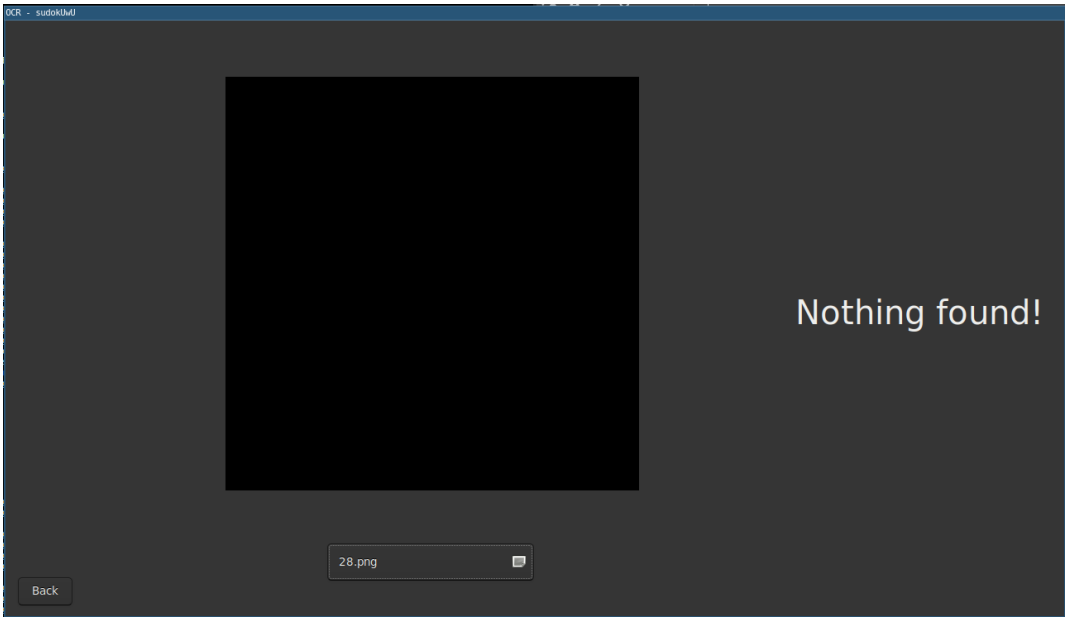


Figure 33. No digit recognizable

4.7.3 Load page

This page is where real things are starting to appear, we here use real features of glade and GTK. The second screen (load page) is displayed as follow:

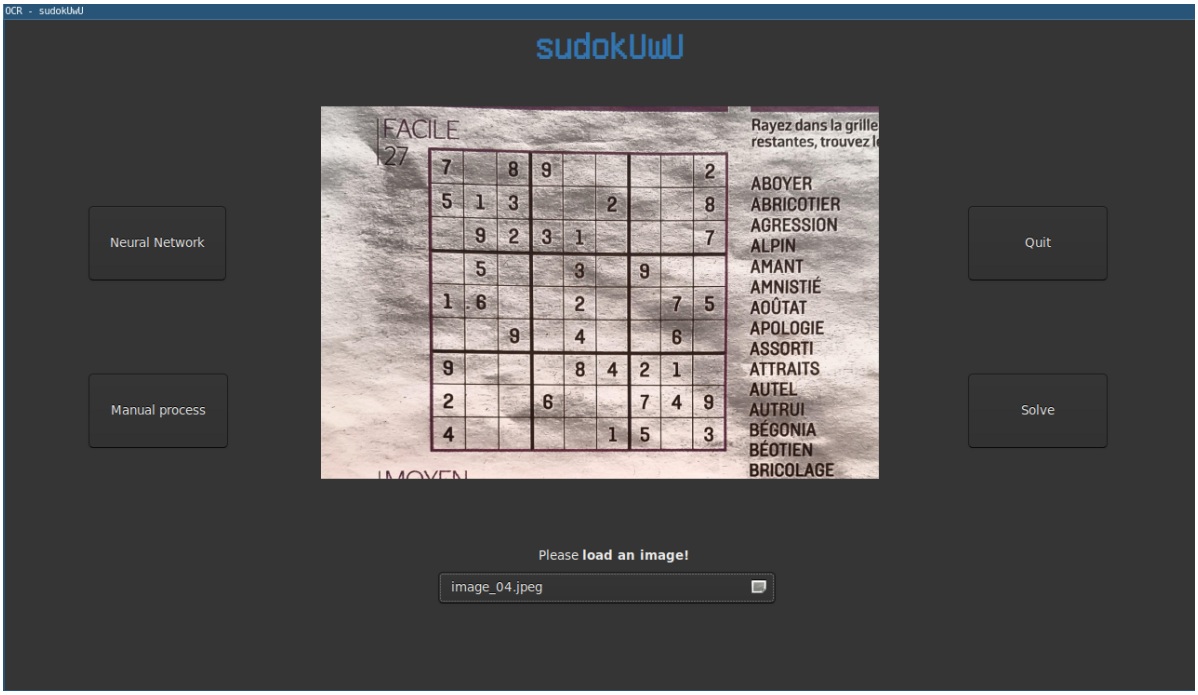


Figure 34. Load page

We can see a bunch of buttons and displaying areas, they are all useful. First we have the **GtkImage*** in the center that display the current image you want to use, by default it is empty. It will only show when one file is selected. You can change the image every time you want. To make it fit in the window we needed to re-scale it.

We had to use **GdkPixbuf** and use one of the method that allows us to scale the image with a precise width and height. We also decided to keep the aspect ratio of the image.

To continue with the display things we use a **GtkLabel*** to give our Load page a title. The title is only the name of the group "sudokUwU".

For the technical part it was a bit more complicated since we had to find everything by ourselves, such as how to load an image. On the picture above we have 5 buttons, 4 are normal buttons that wait to be clicked and the one at the bottom is a **GtkFileChooserButton***.

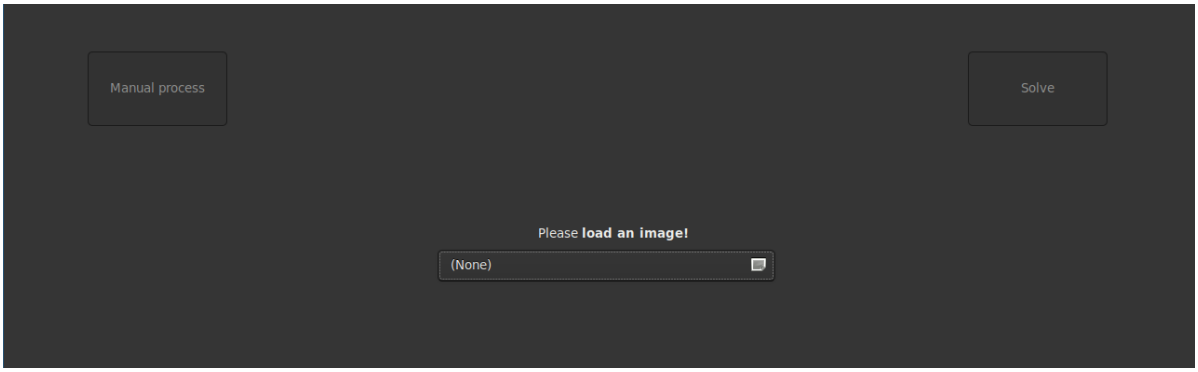


Figure 35. Buttons are not sensitive

This is how you will find the buttons when you first load this page. You can see that "Manual process" and "Solve" are set no **not sensitive** at first because we do not want to process nothing.

Then the process is as follow:

- Load an image in the GtkFileChooserButton and cast to GtkFileChooser in order to get the path of the loaded image. We now have the filename, it remains two choices regarding the solver part:
- Manual process: you will see every step of the process to solve the Sudoku (including the filters and the square-detection, etc).
- Solve: it will display directly the final result skipping the transitional steps.

Then we have the neural network button, it will redirect the user to a page where he can try if his digit his recognize or not.

And the last button is the "Quit" one, it says what it says... By clicking on it you will close the application. It uses the **gtk_widget_destroy**, and we cast the main window of the interface to a **GTK_WIDGET**.

4.7.4 Filters page

The filter's page is quite simple as well, it display the processed image in the **GtkImage*** area when we first go to this page.

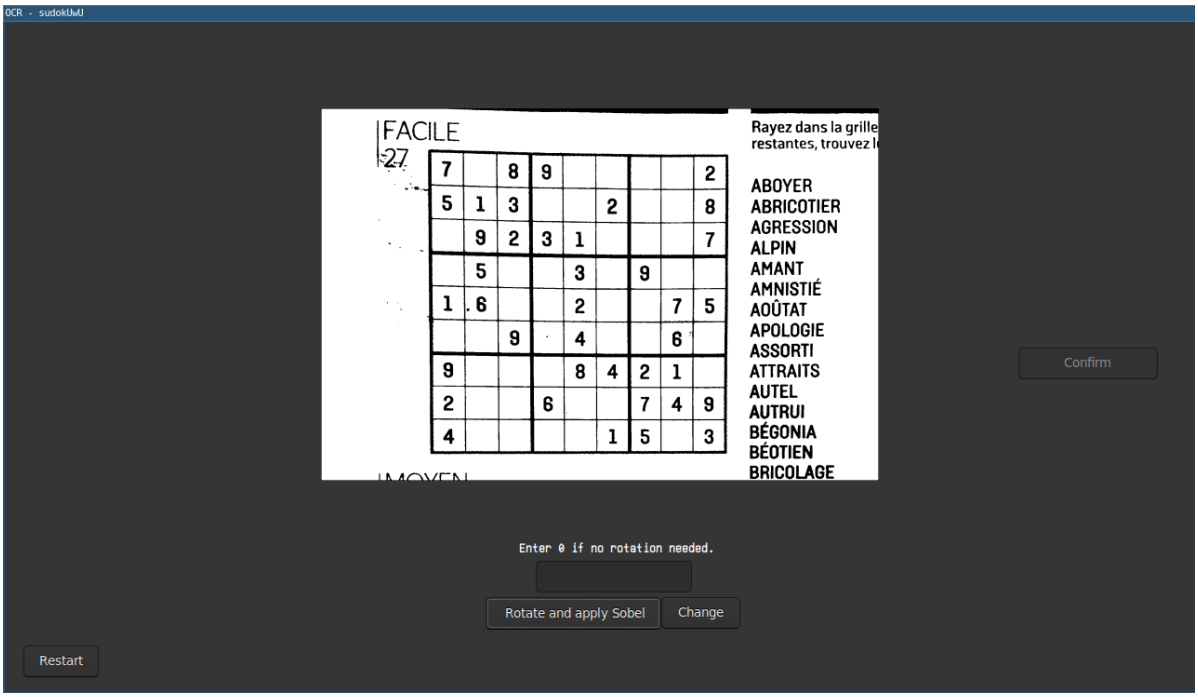


Figure 36. Filters page

But we have the possibility in this section to use one of our pre-processing function which rotate the surface. We let the choice to the user to rotate with the angle he wants. In order to get the degree we created a **GtkEntry*** field which is linked to the ManualRotate function. If by any chance, the user got a wrong angle, he still can reset the image by clicking on the button "Change". It will delete the rotated image and it will let the possibility to the user to rotate with a different angle.

Then when you have rotated the image (you need to enter 0 if you do not want to rotate) you can confirm the angle of rotation and click and "Confirm".

If the user want to change the image he is currently using he can press "Restart" and he will be sent to the first page where he can load an image.

4.7.5 Square detection page

The page where is shown the most important part of the OCR process...

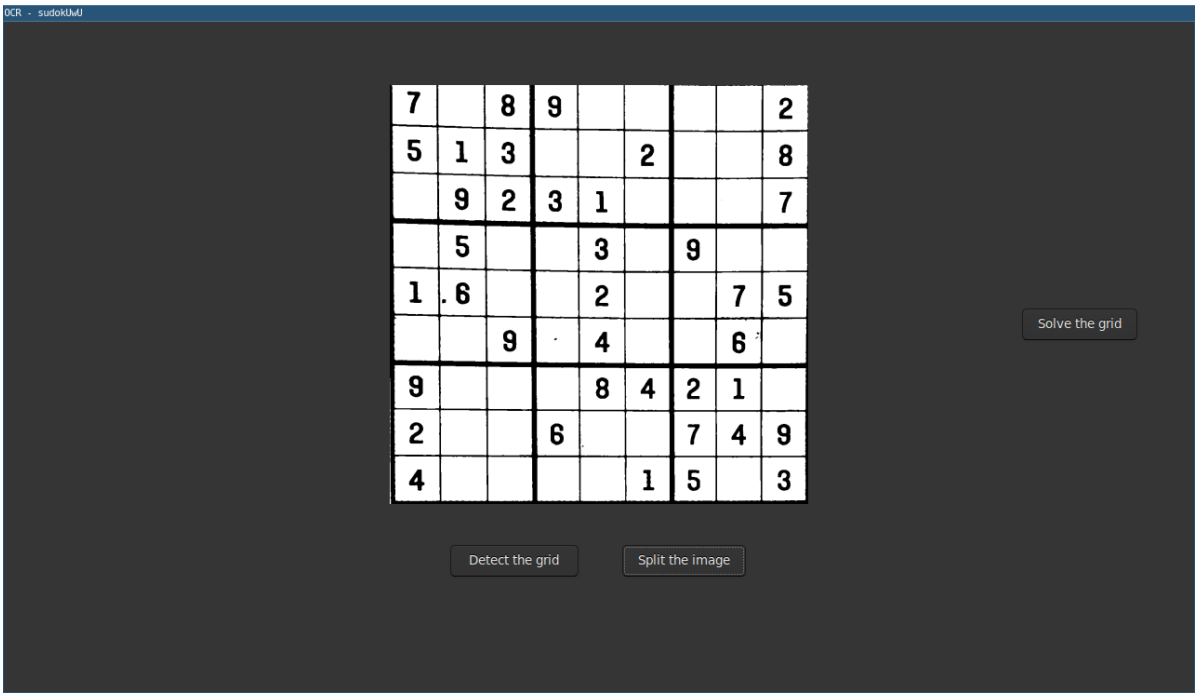


Figure 37. Square detection page

As always the current image process is shown on the screen in the GtkImage*. Only three buttons are present on this page. When first reaching this section only the button "Detect the grid" is clickable. We force the user to follow different steps in order not to make the program crash.

The first button will detect the 4 corners of the Sudoku grid using Hough Transform lines intersection. It will then stock then in an array and will determine the square.

Then you have to click on the "Split the image" button, it will cut the Sudoku square in 81 smaller images. It will also clean them to remove the border and process each images to have them reversed. To finish the button "Solve the grid" which will solve the grid and display the output on the next page!

4.7.6 Output page

The output page is the simplest one, you only have one **GtkImage*** that display the final resolved Sudoku. A "Restart" button that will redirect the user to the loading page and a "Quit" button to close the application.

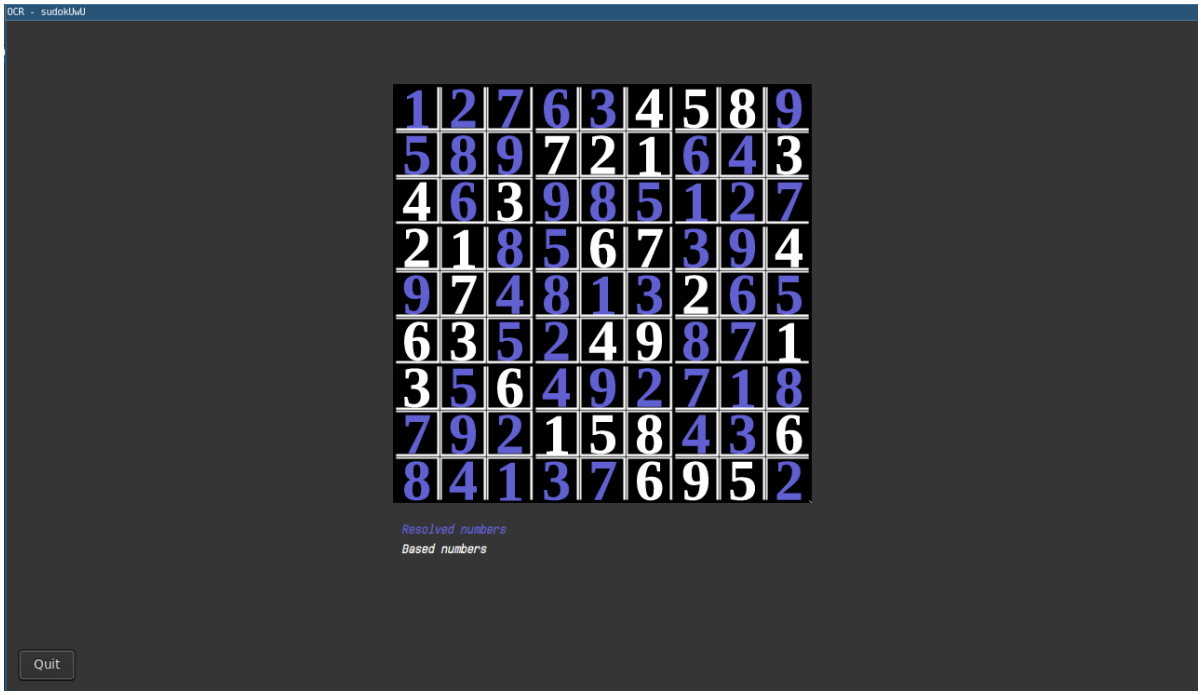


Figure 38. Output page

5 Conclusion

OCR project is now over and we are proud of what we have done so far. It was not easy, we struggled a lot on different parts but we managed to do everything we wanted. Despite the fact that we lost a comrade during the last part we succeed in reallocating parts. For the OCR we have done:

- Neural network that recognize digits (digits that are in the Sudoku).
- Complete post-processing.
- The final grid (the resolved Sudoku) not as a string printed in a terminal but as an image.
- The save of the result.
- A graphical interface to use all the features listed above.

The application is easy to use. We now feel happy that this project is over and it is with enthusiasm that we will harder for the next projects!