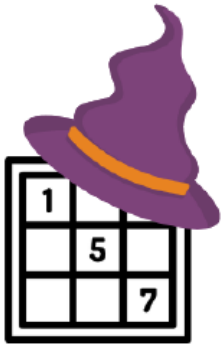
sudoQ

mickael.bobovitch
maxime.ellerbach
gabriel.toledano
noe.susset

Group: C!(Sors.c)

S3 2026

C!(Sors.c)



Contents

1 Team members

1.1 Mickael Bobovitch

I always liked to build stuff from scratch. This project is not a exception. I am a big fan of computer vision and AI, so this one of the best project i could have dreamed of. I am curious about everything that exist and always try to educate myself about new things. This is especially true for this project. I have learned so much different technics and algorithms and always try to optimize. In this project i worked on the structure, APIs, UI, unit testing and the computer vision.

1.2 Maxime Ellerbach

I have done in the past multiple computer vision and Deep Learning projects. I never really felt the need to dig up all the algorithm I used, and this is precisely the opportunity to do so ! I will mainly focus on implementing from scratch the DeepLearning part of the project, not only by meeting the requirements of the subject, but going a bit further to make use of my previous experiences. I am also really interested in the process of structuring a C project which I never had the occasion to do so. Working as a team should not be an issue as we developed a bunch of automated tests using Github actions early in the project. This avoids breaking any previously written code and to ensure a great test coverage of our code, this also comes with a great Makefile to facilitate testing individual parts of our project.

1.3 Gabriel Toledano

Now in my second year of the integrated preparatory program at EPITA, I discovered that this world attracted me when I was in high school. This is due to my discovery of cybersecurity thanks to a course offered on the internet. Indeed, like most of the skills I acquired before entering EPITA, I discovered computer science on the internet and discovered its many facets. Thus, once I entered the school, I was able to improve my computer skills thanks to the different projects proposed. The one I did in my first year of preparatory school was a game in CSharp developed in the Unity environment. The game was similar to a virtual escape game with an AI as game master. Today, I work in parallel for the courses, my cybersecurity projects, my mobile application projects and finally on a sudoku solver. Indeed, this year's project consists in receiving a picture of an unfilled sudoku and solving it using the analysis of the picture, an AI recognizing the numbers and an algorithm finding a solution to this problem. I am particularly interested in the image analysis, which is very vast, as it requires knowledge of multiple factors to be able to carry it out as well as possible. The application of filters on the image and the recognition of the grid are the parts that interest me the most.

1.4 Noé Susset

Hello my name is Noé Susset, at first this project were for me a mix between fear and excitation, because i was afraid of all the work that needed to be done but at the same time couldn't way to discover and implement all of that. I am really happy of all that work that we have done for the moment, the organisation inside the team is great and there is no problem working all together. I think we are on a good path

for this project and we expect it to be very good at the end. I really appreciate all the new things I learned during this journey, between the C, SDL, image processing and the AI, so I can't wait to finish this project.

2 Planned tasks distribution

Tasks	Mickael B.	Maxime E.	Gabriel T.	Noé S.
Project Setup	x	x		
Unit Testing	x	x	x	
Solver				x
Matrix Operations	x	x	x	
Image Preprocessing	x		x	x
Image Segmentation	x		x	
Dataset Generation	x	x	x	
Data Visualization	x			x
Convolutional Neural Network		x		
User Interface				x
Website			x	

3 Project Structure

Our project typically consists of several different components, including the following:

- The Sudoku Solver Program, which contains the logic to load, resolve and save the Sudoku.
- The Matrix Module: this part is one of the most important in this projects as the CV and NN parts heavily rely on it.
- The Computer Vision Module: there we can find all the logic and algorithms that concern computer vision.
- The Neural Network Module: this module is splitted in 2 others subparts (layers and the implementation itself). The Neural Network Module is minimal, yet powerful as it comes with a completely working CNN!
- The Testing Unit is responsible for making our code bug free, easy to read, easy to debug and make the project more modular without having breaking changes.
- The UI which assemble all pieces together!

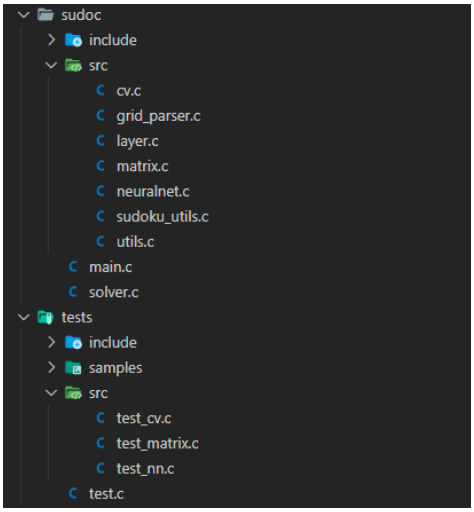


Figure 1: The structure of our project

4 Solver Module and Executable

4.1 The Process

To implement a simple sudoku solver program, we first started by creating a data structure to represent the sudoku grid. We used a two-dimensional array to store the values of each cell in the grid. Next, we implemented a function to read in a sudoku puzzle from a file and populate the grid with the initial values. This involved parsing the input file and validating the input to ensure that it was in the correct format. Once the grid was populated with the initial values, we implemented a series of algorithms to solve the puzzle. These algorithms used various techniques, such as checking for cells with only one possible value, and using constraint propagation to eliminate possibilities for other cells. We then implemented a function to display the solved sudoku puzzle in a user-friendly format. This involved formatting the output and displaying it in a grid that closely resembled the traditional sudoku layout. Finally, we tested the program using a variety of sudoku puzzles, both easy and difficult, to ensure that it was able to solve them accurately and efficiently. Overall, our sudoku solver program was able to solve most sudoku puzzles accurately and efficiently, thanks to the use of various solving algorithms and a well-designed data structure.

4.2 Parsing

To create a parser to read a text file containing sudoku data, we first started by designing the data structure that would be used to represent the sudoku grid. We decided to use a two-dimensional array to store the values of each cell in the grid. Next, we implemented a function that would read in the text file containing the sudoku data. This involved opening the file and reading in the data line by line, using string manipulation techniques to extract the relevant information from each line. Once the data was read in, we implemented another function to parse the data and populate the grid with the initial values. This involved validating the input data to ensure that it was in the correct format and populating the grid with the appropriate values. Finally, our parser was able to read and parse text files.

4.3 Solving

To solve the sudoku puzzle, we implemented a series of algorithms that used various techniques to determine the correct values for each cell in the grid. These algorithms included the following:

1. Checking for cells with only one possible value: In this technique, we would look for cells that only had a single possible value, based on the existing values in the same row, column, and 3x3 sub-grid. If a cell had only one possible value, we would fill it in and propagate the change to the other cells to eliminate that value from their possible values.

2. Using constraint propagation: This technique involved using the existing values in the grid to eliminate possibilities for other cells. For example, if a cell had a value of 9, we would eliminate the possibility of 9 for all other cells in the same row, column, and 3x3 sub-grid.

3. Backtracking: If the previous techniques were unable to determine the value for a cell, we would use backtracking to try different combinations of values until we found a solution. This involved filling in a cell with a possible value, and then propagating the change to the other cells to see if it led to a valid solution. If it did not, we would undo the change and try the next possible value.

This algorithm is very effective against a sudoku and will solve it fast.

4.4 Saving

After we solve the sudoku we need to re-create a text file with the same name of the original file and the extension *.result*, this file need to be the solved grid of the source file. So we need to do the opposite thing of the parser : transform a matrix into a text file and save it.

Below there is an example of all those functions together and the results of this.

```
noesusset@DESKTOP-65E930O:~/test$ ls
grid_test  solver
noesusset@DESKTOP-65E930O:~/test$ cat grid_test
53. .7. ...
6.. 195 ...
.98 ... .6.

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

.6. ... 28.
... 419 ..5
... .8. .79
noesusset@DESKTOP-65E930O:~/test$ ./solver grid_test
noesusset@DESKTOP-65E930O:~/test$ ls
grid_test  grid_test.result  solver
noesusset@DESKTOP-65E930O:~/test$ cat grid_test.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 2: Example of the parsing, solving and saving of a grid

4.5 Saving with digits (Extra Feature)

Once the sudoku is solved, we want to draw the digits that our AI has found on the image to help people easily. To do so, our team as a whole implemented a very complicated function based on array initialization. Meaning that we created a table of arrays with zeroes and ones. Each one represents a line to draw to form the wanted digit.

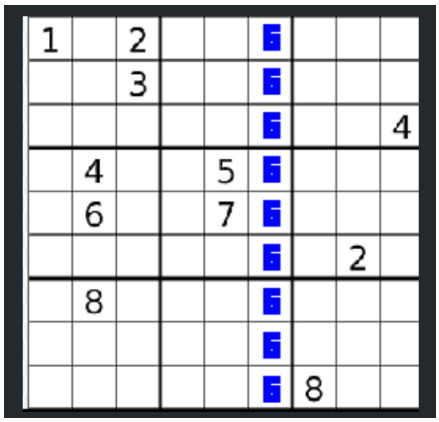


Figure 3: An output if we wanted to put 6 on one column

5 Matrix operations

Matrix are of a huge importance on this project. Indeed, between working with images and having a neural network with different kind of layers: fully connected or convolutional, a good handling and manipulation of matrices is really important. We needed to implement different types of matrix depending on the situation.

When working with regular matrices, we are using a simple 1d array in row major, this enables us to allocate only once the array. We then go through the matrix as a 2d array using macros just like a regular 2D array.
When working with images, we are using 3d Arrays. Which can be considered as an array of 2D matrices of shape: (channels, height, width). Again, it is in fact a single dimensional array.
When working with batches of images (used for the convolutional layers), we are using 4D matrices which can be considered as an array of 3D matrices. Again, it is in fact a single dimensional array.

The reason why all of our multidimensional arrays are in fact 1D arrays is because We found this way was the easiest for us to be able to copy easily the content of an array or part of an array using the memcpy function. For example, we can easily extract the content of a channel of an image to a 2D matrix. Same goes for putting images into a batch of images, a single memcpy is required.

Our Matrix4 struct which represent our 4D matrix:

```
struct Matrix4
{
    int dim1;
    int dim2;
    int dim3;
    int dim4;
    int size;
    float *data;
};
```

Figure 4: Matrix4D struct

5.1 2D Operations

In order to easily develop functions in our project, we needed to be able to rely on basic matrix operations. We developed early in the project those operations with a complete set of tests to avoid any unexpected behaviors.
Here are the basic operations we implemented for 2D matrices:

- Element-wise addition.
- Element-wise subtraction.
- scalar addition / subtraction.
- matrix multiplication.
- matrix transpose.
- matrix map function (apply a float-¿float function element-wise).
- matrix determinant
- matrix inverse
- matrix solution

Those are really useful for the image processing and the fully connected layers of the neural network.

5.2 Example use case with Computer Graphics

Iin image processing, matrices are used to represent and manipulate digital images. This can be useful for a variety of applications, such as resizing, cropping, and rotating images, as well as applying filters and other transformations.

Matrices are often used to represent the intensity values of the pixels in an image. For example, a grayscale image can be represented by a two-dimensional matrix, where the value of each element in the matrix corresponds to the intensity of the pixel at that location in the image. This allows for efficient manipulation of the image by treating it as a mathematical object.

In addition to representing the pixel values of an image, matrices can also be used to apply mathematical operations to the image. For example, a matrix can be used to rotate an image by representing the rotation as a linear transformation, and then applying that transformation to the matrix representing the pixel values of the image.

Here is another use case:
A perspective transformation is a type of image transformation that is used to change the apparent perspective of an image. This is typically done by using a transformation matrix, which is a mathematical representation of the transformation that is to be applied to the image. In the case of a perspective transformation, the matrix encodes the mapping of points in the original image to new positions in the transformed image. This mapping allows the image to be transformed in a way

that changes its apparent perspective, for example, by making objects in the image appear to be closer or farther away. This is used in our optical character recognition to correct for distortions in the original image and to make objects in the image appear more natural to the rest of our processing.

A transition matrix is a mathematical construct that can be used to represent a linear transformation, such as a translation, rotation, or scaling. In the context of zooming and scaling a Sudoku grid image, the transition matrix can be used to map the coordinates of the corners of the grid in the original image to their corresponding coordinates in the transformed (zoomed and scaled) image.

The transition matrix, denoted by T, is a 3x3 matrix of the form:

$$T = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

where a, b, c, d, e, and f are real numbers that determine the type and amount of the linear transformation. For example, if we want to translate a point by a certain amount in the x and y directions, we would use a matrix of the form:

$$T = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}$$

where x and y represent the amounts to translate the point in the x and y directions, respectively.

To apply a transition matrix to a point in an image, we first need to represent the point as a 3x1 column vector, denoted by P, of the form:

$$P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where x and y are the coordinates of the point in the image. Then, to transform the point using the transition matrix, we simply need to compute the matrix product $T \cdot P$, which will give us the new coordinates of the point in the transformed image.

In the context of zooming and scaling a Sudoku grid image, we can use the transition matrix to map the coordinates of the corners of the grid in the original image to their corresponding coordinates in the transformed image.

To zoom and scale a Sudoku grid image, we need to use a transition matrix that encodes the desired scaling and zooming transformations. For example, if we want to scale the image by a factor of 2 in both the x and y directions, and also translate the image by a certain amount in the x and y directions, we could use a matrix of

the form:

$$T = \begin{bmatrix} 2 & 0 & x & 0 & 2 & y & 0 & 0 & 1 \end{bmatrix}$$

where x and y represent the amounts to translate the image in the x and y directions, respectively.

To apply this transformation to the Sudoku grid image, we would first need to represent the coordinates of the corners of the grid as 3x1 column vectors, and then compute the matrix product of the transition matrix and each of these column vectors to get the new coordinates of the corners in the transformed image.

For example, suppose that the original coordinates of the top-left corner of the grid are (x_1, y_1), and the coordinates of the top-right corner are (x_2, y_2). We could represent these points as 3x1 column vectors as follows:

$$P_1 = \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} \quad P_2 = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix}$$

Then, to transform these points using the transition matrix T, we would compute the matrix products $T \cdot P_1$ and $T \cdot P_2$ to get the new coordinates of the top-left and top-right corners in the transformed image.

After computing the new coordinates of all the corners of the grid in the transformed image, we can use these coordinates to draw the grid on the transformed image.

Overall, the use of matrices in image processing allows for efficient and precise manipulation of digital images.

We can use Matrices to do Perspective Warp Transformations:

This is a method for transforming an image in such a way that the resulting image has a different perspective from the original. This can be useful for correcting distortion in an image, or for creating a specific effect. The algorithm works by defining a set of four points in the original image, and then specifying how those points should be mapped to four new points in the transformed image. The algorithm then uses these mappings to calculate how the pixels in the original image should be distorted in order to create the desired perspective in the transformed image. This allows the algorithm to effectively "warp" the perspective of the image, giving it a different appearance than the original.

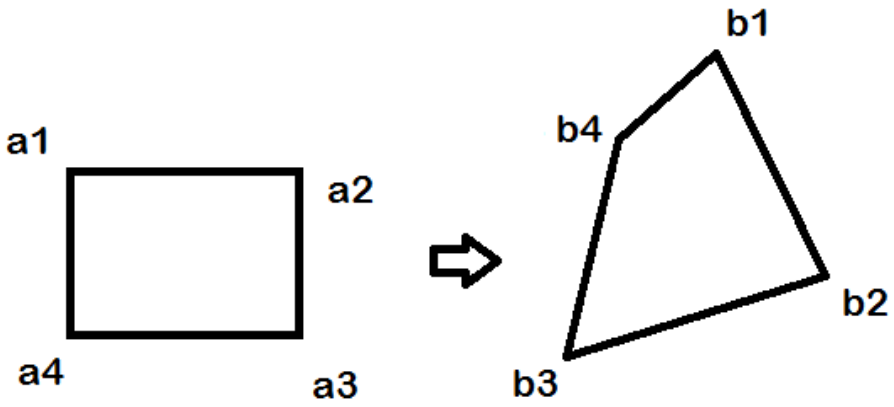


Figure 5: Matrix Perspective Transform

5.3 4D Operations

In order to facilitate the development of our convolutional layers, as weights are 4D arrays, we were required to implement similar operations as we have for 2D matrices.
Here are the implemented operations:

- Element-wise addition.
- Element-wise subtraction.
- scalar addition / subtraction.
- matrix4 convolution (convolution of a 4D matrix with a 4D weight matrix containing 2D kernels).
- matrix4 transpose. actually does the same as a regular matrix transpose, transposes the last 2 dimensions of the 4D matrix, resulting in a 2D matrix transpose.
- matrix4 convolution transpose (a 180 degrees rotation of the kernels followed by a convolution).
- matrix4 map function (apply a float-¿float function element-wise).

5.4 Others

There are others functions about matrix that were important for debugging, utils and so on. Those functions are available for both matrix 2D, 3D and 4D:

- Initiation. initiate a matrix of the size we want and either fill it with 0 or copy it from another matrix
- Deletion. Deletes the matrix one we don't need it anymore.
- Print. Prints the matrix along with its dimensions for debugging purpose.
- Equal. in order to test our functions, we had to create an element-wise equal function (this is simply comparing each element of the 1D array containing all the values along with the dimensions of the matrix). Note that two matrices can have the same 1D array values but not the same dimensions and thus not be equal.

6 Image Processing

6.1 Image loading and saving

For the Image processing we started by using the SDL library, to load the image then we convert it into a structure that would be easier for us. This structure use 4 attributes that are :

- The width (w)
- The height (h)
- The number of channel (c)
- Data (data)

This allow us to not use a Row-Major representation and modify easily the image as we want. Here is the structure :

```
typedef struct
{
    int c;
    int h;
    int w;
    pixel_t *data;
} Image;
```

Figure 6: Struct of the type IMAGE

For the saving we made another function that convert our structure into a SDL image then we can use the lib to save the image as a PNG after we modify it.

6.2 Grayscale

One major conversion that is used in this project in the grayscale, indeed we don't really care about the color so changing the image into a gray one is more efficient since we get rid of useless information like the RGB. To apply this filter we update each channel of the image by the result of this equation, using the red, green and blue channel.

$$average = (r + g + b)/3$$

Then we can get only gray pixels on all the image

Figure 7: Application of the grayscale filter

6.3 Rotation

In the case that the image entered by the user is not correctly oriented, it is necessary to rectify this discrepancy in order to run the segmentation algorithm.

6.3.1 Manual Rotation

We implemented this manual rotation, that performs a rotation of the image for a given angle. We start by creating a destination matrix (which is an instance of our structure image) of the same size and depth as the source image then using this equation on each pixel we can get the coordinates of the new pixel position in the destination image :

$$x_1 = cos(\theta) * (x_0 - x_{center}) - sin(\theta) * (y_0 - y_{center}) + x_{center}$$
$$y_1 = sin(\theta) * (x_0 - x_{center}) + cos(\theta) * (y_0 - y_{center}) + y_{center}$$

Where :

1. (x_0, y_0) : coordinates of the pixel in the source image
2. (x_1, y_1) : coordinates of the pixel in the destination image
3. (x_{center}, y_{center}) : respectively the center of the width and the center of the height
4. θ : angle of rotation

If the new position of the pixel is not included into the bound of the image it will put a black pixel at the initial position.

Figure 8: Application of a rotation of -39 on the image

6.4 Gaussian blur

In image processing, a Gaussian blur is the result of blurring an image by a Gaussian function. 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. The apply a Gaussian filter to an image, we first need to compute a convolution kernel matrix. The matrix is a simple 2D Gaussian distribution and is applied to each pixel of the image. It is good to know that, the larger the kernel, the longer does is takes to process the image.

Figure 9: Application of a rotation of -39 on the image

6.5 Edge detection

The aim of the analysis of a sudoku image is to locate the grid. To do this, we will use different filters and methods to format the image as we wish in order to extract the most important information. Edge detection is the name for a set of mathematical methods which aim at identifying points in a digital image at which the image brightness changes sharply. The aim is to find the boundaries of objects within images.

6.5.1 Sobel filter

To begin with, thanks to the Gaussian blur that removed the noise, we implemented the Sobel filter. The operator calculates the gradient of the intensity of

each pixel. This indicates the direction of greatest change from light to dark, and the rate of change in that direction. The points of sudden change in brightness, presumably corresponding to edges, and the orientation of these edges are then known. In mathematical terms, the gradient of a function of two variables (in this case intensity versus image coordinates) is a vector of dimension 2 whose coordinates are the derivatives in the horizontal and vertical directions. At each point, the gradient points in the direction of greatest change in intensity, and its length represents the rate of change in that direction. The gradient in an area of constant intensity is therefore zero. At a contour, the gradient crosses the contour from the darkest to the lightest intensities.

The operator uses convolution matrices. The matrix of size 3×3 is convolved with the image to calculate approximations of the horizontal and vertical derivatives. These images are calculated as follows:

$$G_x = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]] * A$$
$$G_y = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]] * A$$

Where :

1. G_x : image which at each point contains approximations of the horizontal derivative
2. G_y : image which at each point contains approximations of the vertical derivative
3. A : the source image

At each point, the approximations of the horizontal and vertical gradients can be combined as follows to obtain an approximation of the gradient norm:

$$G = \sqrt{(G_x)^2 + (G_y)^2}$$

We then apply this operation on all pixels of the source image to get our final one.

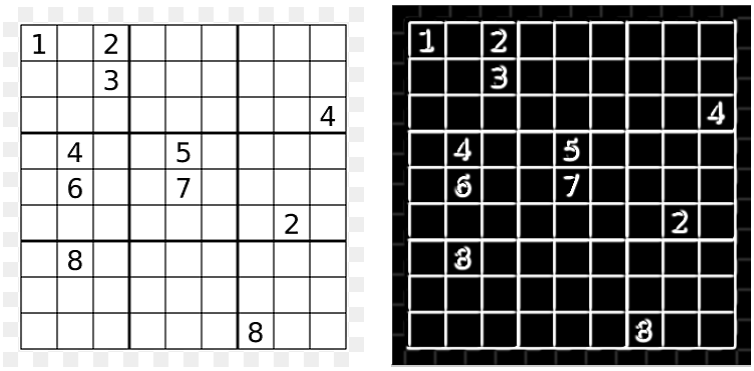


Figure 10: Application of a sobel filter on the image

6.5.2 Canny filter

Nevertheless, the sobel filter returns white outlines that are too thick and sometimes not precise enough. We therefore needed to implement a more efficient algorithm, such as the canny filter. This filter allows us to receive an image with the contours proposed by sobel but more precise and finer. Indeed, all the contours displayed will have a size of 1px, which allows an increase in the precision of the image data.

The algorithm consists of three parts. First, we apply the sobel filter. Then, we perform a non-maxima suppression. The gradient map obtained previously provides an intensity at each point of the image. A high intensity indicates a high probability of the presence of a contour. Only the points corresponding to local maxima are considered to correspond to contours, and are kept for the next step of the detection. Finally, a thresholding of the contours is applied to the map generated by the previous process. This requires two thresholds, one high and one low, which are compared to the gradient intensity of each point. The decision criterion is as follows. For each point, if its gradient intensity is below the low threshold, the point is rejected. If it is above the high threshold, the point is accepted as forming a contour. If it is between the low and high threshold, the point is accepted if it is connected to an already accepted point.

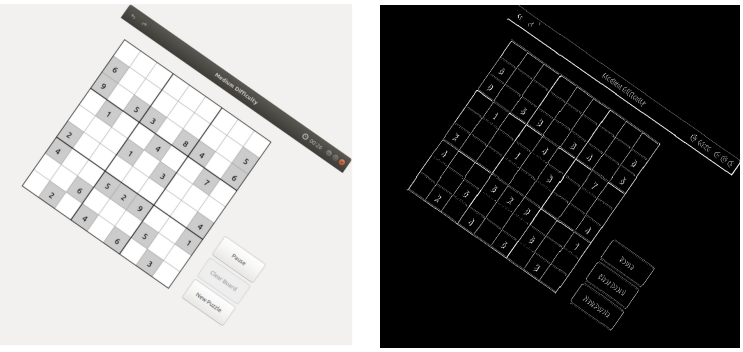


Figure 11: Application of a canny filter on the image

6.5.3 Otsu filter

OTSU method is used to perform automatic image thresholding. In the simplest form, the algorithm returns a single intensity threshold that separate pixels into two classes, foreground and background. This threshold is determined by minimizing intra-class intensity variance, or equivalently, by maximizing inter-class variance. Otsu's method is a one-dimensional discrete analog of Fisher's Discriminant Analysis, is related to Jenks optimization method, and is equivalent to a globally optimal k-means performed on the intensity histogram. The extension to multi-level thresholding was described in the original paper, and computationally efficient implementations have since been proposed.

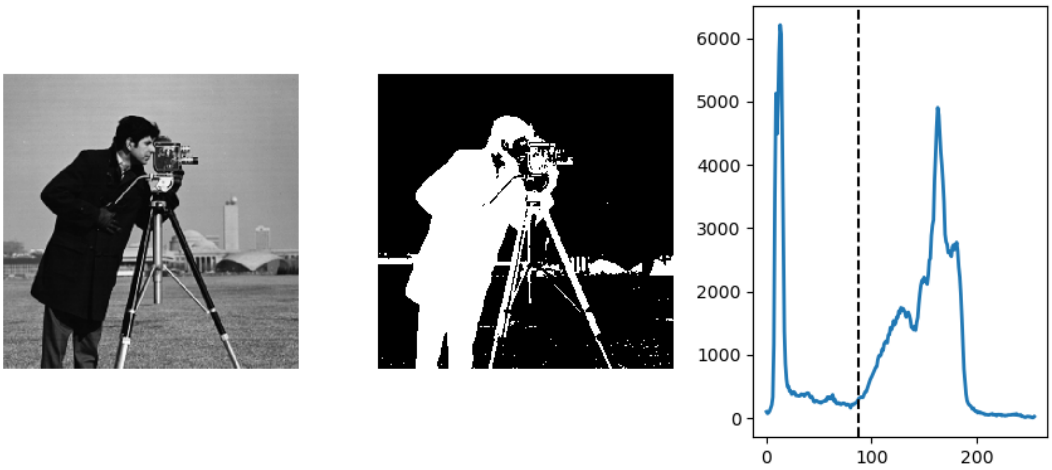


Figure 12: OTSU Technic example

Pseudo code:

Algorithm 1: Otsu implementation

Input: input_image (grayscale), overridden$_t$*hreshold*
Output: output_image
N = input_image.width x input_imnumberage.height;
threshold, var_max, sum, sumB = 0;
max_intensity = 255;
for $i \leftarrow 1, n$ **do**
  histogram[value] = 0;
end
if *num_channels(input$_i$mage)* > 1 **then**
  return *error*
end
for $i \leftarrow 0, N$ **do**
  value = input_image[i];
  histogram[value] += 1;
end
for $i \leftarrow 0, max\_intensity$ **do**
  sum += i x histogram[i];
end
for $t \leftarrow 0, max\_intensity$ **do**
  q_1+ = *histogram*[t];
  if $q_1 == 0$ **then**
  end
  $q_2 = N - q_1$;
  sumB += t x histogram[t];
  $\mu_1 = sumB/q_1$;
  $\mu_2 = (sum - sumB)/q_2$;
  $\sigma_b^2(t) = q_1(t)q_2(t)[\mu_1(t) - \mu_2(t)]^2$;
  if $\sigma_b^2(t) > var\_max$ **then**
    threshold = t;
    var_max = $\sigma_b^2(t)$;
  end
end
for $t \leftarrow 0, N$ **do**
  if *input$_i$mage*[i] > *threshold* **then**
    output_image[i] = 1;
  else
    output_image[i] = 0;
  end
end
return *output_image*

Figure 13: OTSU Technic example

6.5.4 Adaptive Threshold filter

OTSU has a very impractical disadvantage : it's works great on equally exposed images, but not on images with different light exposure which impact negatively the result by adding some staint. I improved the OTSU Method by combining an adaptive threshold filter which rely on general threshold computed by OTSU and a local threshold computed with a gaussian kernel.

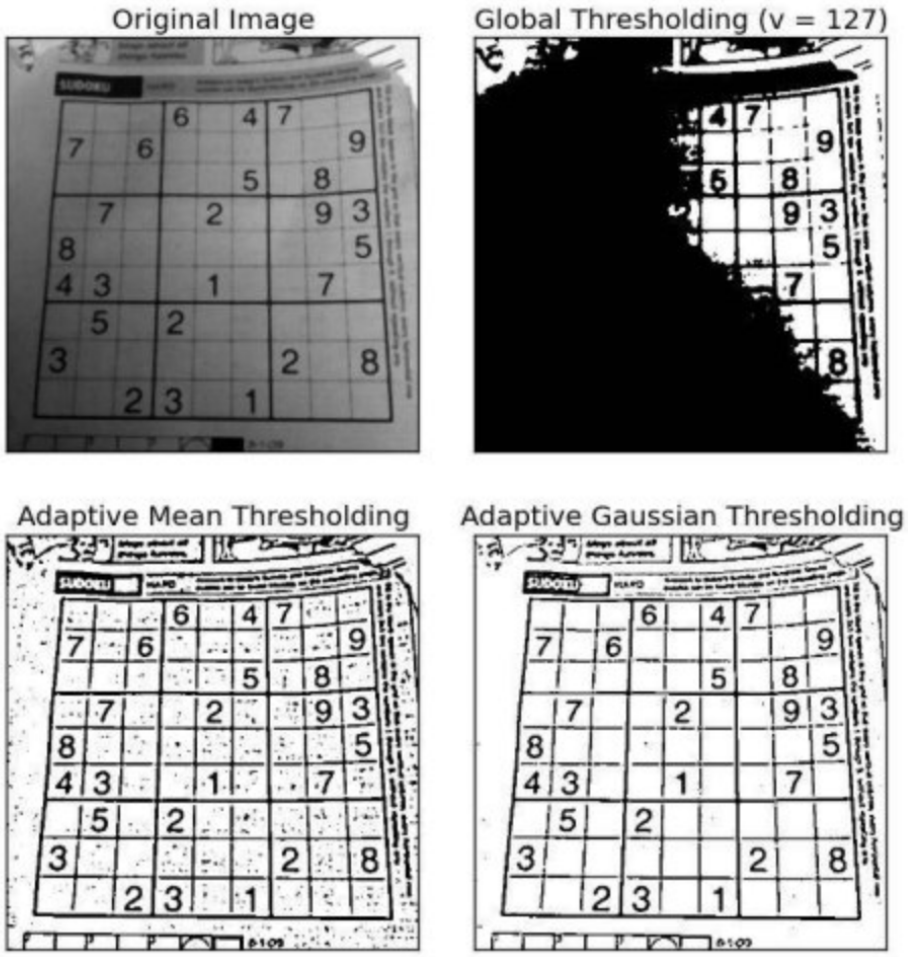


Figure 14: OTSU Technic example

6.5.5 Binary operators on filters

As the paper is white and lines and digits are black, we need to revert the colors. As we work with floating points values, a black pixel value is 0 and a white pixel is 1. We implemented some functions like NOT, AND, OR and XOR. As the neural network will work with values that have meanings, in our case zeroes or black pixels are considered as the background of the image.

6.6 Lines detection

Once we have detected the edges of the input image, we need to output the sudoku grid. To do this, we will perform line detection using different algorithms.

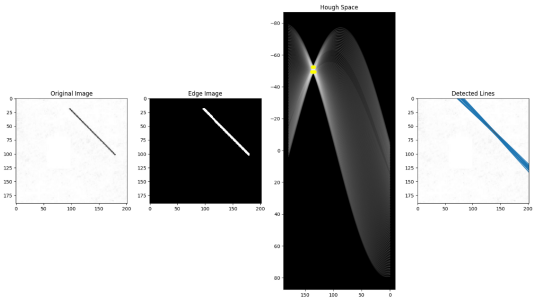


Figure 15: Accumulator represented as an image

6.6.1 Hough Transform

The Hough transform is a pattern recognition technique. We use it to recognize lines in our case. Firstly, we assumed that if lines or line segments are present in an image, they will be part of the contours present in the image. We therefore start by identifying all the contour points in this image using the previous filters. Each of the points of the contours thus identified (x, y) will then allow a projection in a plane (the transformed plane) of the polar coordinates of all the lines passing through this point. The equations of the lines passing through each of these points (x, y) are therefore represented by a pair (ρ, θ) satisfying the following equation:

$$\rho = x \cdot cos(\theta) + y \cdot sin(\theta)$$

Thus, if we take our image as an orthonormal frame of reference, its origin will be in the left-hand corner. From there, we have the angle between the x-axis line and the line we have just identified, as well as its distance from the origin. Now, we have to make sure that the point considered as a straight line is indeed a straight line. To do this, we initialise an accumulator of the size of the number of pixels in the image multiplied by all the possible angles through the line. That is, from 0 to 180°.

```
int w = src->w;
int h = src->h;

int *accumulator = (int *)malloc(sizeof(int) * w * h * 180);
memset(accumulator, 0, sizeof(int) * w * h * 180);
```

Figure 16: Initialisation of the accumulator

The choice of the right will therefore work by voting system. Indeed, at the time of calling the function, we indicate a value of threshold to be equalled or exceeded to be considered as a right. During the course of the image, as soon as we find a valid ρ and θ, we will increment by 1 the value of the accumulator at the location $accumulator[\rho * 180 + \theta]$. This will then allow us to count the number of rows

that reach the target required by the previous threshold. We thus obtain the most important lines of the image.

6.6.2 Simplify Hough Lines

As the Hough transformation is sometimes uncertain depending on the image and the threshold values that are passed as parameters, we are obliged to merge lines that are too close to each other. Indeed, we know that in a sudoku, lines with the same θ angle cannot be almost glued. So we still proceed with a threshold value which allows us to merge together lines which are separated by a distance smaller than the threshold value, but which also have a difference of angle smaller than this one. Indeed, the values of their angle can sometimes differ from one degree.
$Ex : (89, 120) and (90, 120)$
Here, the lines will be merged, since this difference is negligible for our analysis of a sudoku grid.

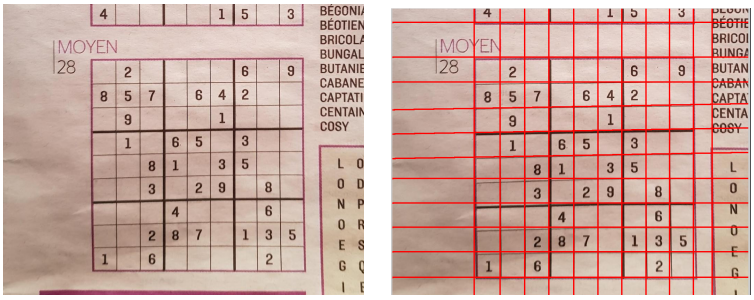


Figure 17: Application of the Hough Transformation on the image

6.6.3 Lines intersections detection

Once we have found the lines, we need to find the intersections between them to simplify our task later. The solution is to apply some fairly simple mathematics to the θ and ρ found earlier to find the crossing points. This is done by comparing the lines in pairs. We start by selecting the two lines. Then, we have to solve a system of coordinate equations to find out if the x and y match for both. This can be expressed in a 2×2 matrix. We then calculate the determinant of the matrix and if this is zero then this means that there are no solutions, so the lines have no points in common. If the determinant is different from 0 then we calculate the point of intersection (x, y) and add it to the list of points that the function returns.

```
for (int i = 0; i < nlines; i++)
{
    int rho1 = lines[i * 2];
    int theta1 = lines[i * 2 + 1];

    for (int k = i + 1; k < nlines; k++)
    {
        int rho2 = lines[k * 2];
        int theta2 = lines[k * 2 + 1];

        float a = cos(theta1 * PI / 180.0);
        float b = sin(theta1 * PI / 180.0);
        float c = cos(theta2 * PI / 180.0);
        float d = sin(theta2 * PI / 180.0);

        float det = a * d - b * c; // determinant of the matrix [[a, b], [c, d]]
        if (det == 0)
            continue;

        float x = (d * rho1 - b * rho2) / det; // intersection point
        float y = (-c * rho1 + a * rho2) / det; // intersection point

        if (x < 0 || x > w || y < 0 || y > h)
            continue;

        intersection[j * 2] = (int)x;
        intersection[j * 2 + 1] = (int)y;
        j++;
    }
}
```

Figure 18: Code to get all lines intersections

6.6.4 Grid boxes detection

Once the list of intersection points is obtained, we sort it according to the y's in order to have all the points from left to right and from top to bottom.From there, we start to locate the squares by finding the points of all the squares. We will stop at two points per square: the top left corner and the bottom right corner. Indeed, we can easily know the dimensions of a square thanks to these two points. To find these points we call a function that returns an array with all these values. This function takes as parameters the number of intersections found previously, the list of intersection points and a reference value to count the number of squares obtained. Indeed, here we could stop at the resolution of 9x9 sudoku but this approach (avoiding constant values) allows us to have a more modular algorithm. The algorithm starts by computing the square root of the number of intersections. Then a loop starts with the limit of the number of estimated cells:

$$nbinter - \sqrt{nbinter}$$

in the case of a 9x9 sudoku it would give 90. This seems strange at first because we expect to get 81 cells and not 90. In reality we will get 81 cells thanks to a condition we add.
If $(i+1) \mod \sqrt{nbinter} == 0$ then it means that we are at an intersection located on the right edge of the sudoku, thus a limit of the grid. This condition is true 9 times, which means for our example that we do not add cells 9 times out of 90. We obtain $90-9=81$ cells in the end. So we save an array filled with dots representing the cells of the sudoku. We can now iterate through the array we obtained to save each box in different image files. To do this, we have created a function that copies part of an image using two points passed as parameters. The function will calculate the area of the area to be copied and return a new image with the dimensions of the copied area and the expected content. So, when we want to save all the images of the boxes, we only have to send the two points representing them to this function and save the images that it returns to us.

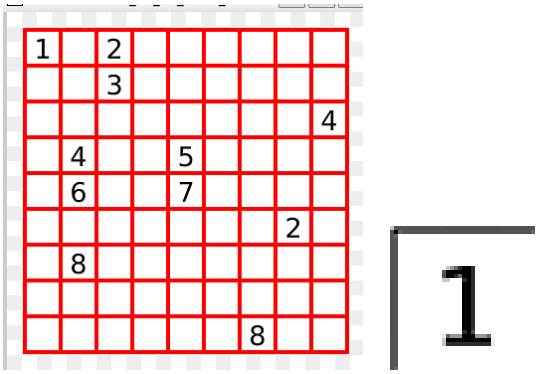


Figure 19: On the left: all boxes are drawn ; On the right: the first box saved

6.6.5 The Flood Fill Algorithm

The flood fill algorithm is a method for filling an area of a digital image with a single color. It is often used in image processing and computer graphics, and is a popular technique for making certain kinds of digital drawings. The algorithm works by starting at a given point in the image, called the seed point, and then painting the surrounding pixels with the desired color. The algorithm uses a queue to keep track of the pixels that need to be painted, and it continues to paint pixels until all of the pixels in the region have been filled. The algorithm is called "flood fill" because it works by flooding the region with the desired color, starting from the seed point and spreading outwards until the entire region is filled.

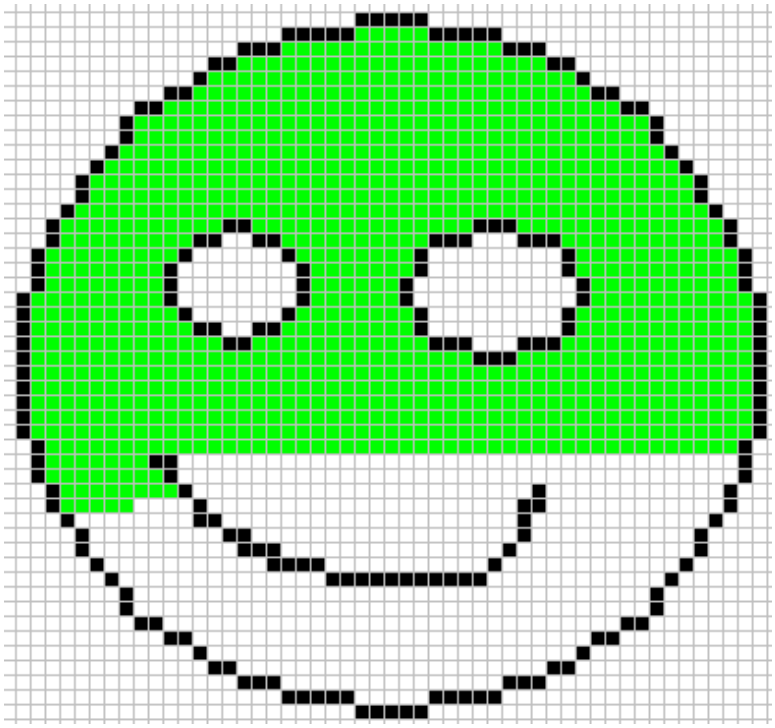


Figure 20: Food Fill Technic to get the biggest connected area

6.6.6 The Convex Hull Algorithm

The convex hull algorithm is a method for finding the convex hull of a set of points. The convex hull is the smallest convex polygon that contains all of the points in the

set. It is often used in geometry, computer graphics, and other fields where it is necessary to find the smallest enclosing shape for a set of points. The algorithm works by first sorting the points by their x-coordinates, and then constructing the convex hull by successively adding points to it. The basic idea is to start with the leftmost point, and then add the remaining points one at a time in increasing order of their x-coordinates. At each step, the algorithm checks to see if the new point forms a left turn or a right turn with the previous two points on the hull. If it forms a left turn, the point is added to the hull, but if it forms a right turn, the previous point is removed from the hull. This process continues until all of the points have been added to the hull.

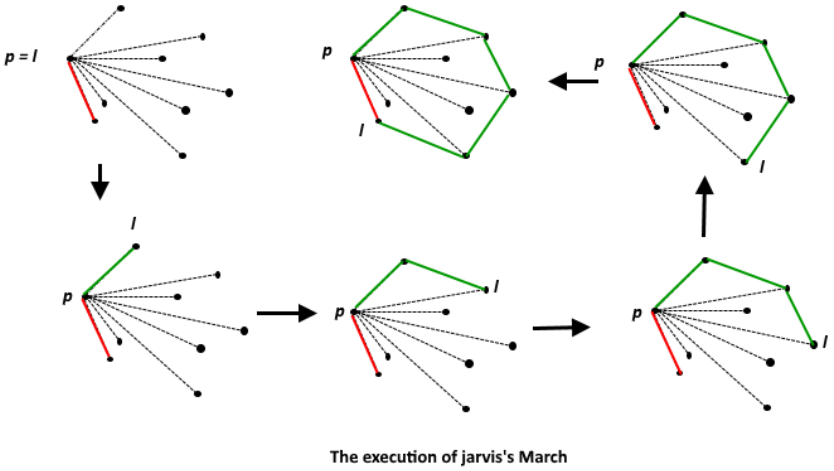


Figure 21: Convex Hull Algorithm to get the contours

7 Dataset and Sudoku grid generation

In order to train our neural network, we will use a mix between the MNIST dataset and the a custom dataset generated using a variety of fonts. The final dataset contains about 80,000 training images and 10,000 test images of both handwritten and fonts numbers. Each image is 28x28 pixels and is labeled with the correct number.

Nevertheless, we sometimes need special rotations, special fonts of digits. In order to do that, instead of browsing for hours, we have done our own dataset generator. We did it in python.
First, we make a request to the Google Fonts API. This allows us to retrieve all fonts made available by Google Inc. . Then, we choose the color of the digit font, the color of the image's background. There are others parameters such as the font size and the rotation of the image to make our OCR really powerful.

```python
def get_random_font():
  # Get a list of fonts from the Google Fonts API
  fonts = requests.get("https://www.googleapis.com/webfonts/v1/webfonts?key="+ GOOGLE_FONTS_API_KEY +"&sort=popularity").json()

  # Choose a random font from the list
  font = random.choice(fonts["items"])
  # Choose a random variant of the font
  variant = random.choice(font["variants"])

  # Download the font file
  font_url = "https://fonts.googleapis.com/css?family={}:{}".format(font["family"], variant)
  font_file = requests.get(font_url).text.split("url(")[1].split(")")[0]
  # Return the font file
  return font_file
```

Figure 22: Request to the Google Font API

Once we have chosen everything, we randomly choose a font in the ones provided by the requests. Then we choose a variant of the font randomly (light, regular, bold and more). This reduces the probability to get the same image twice. We also choose the digit between 0 and 9 randomly and because we generate a minimum of 25 000 thousand images we are sure that we will get almost as much 0 than 1 than 2 than 3 than 4 than 5 than 6 than 7 than 8 and than 9.

```
def generate_dataset(n):
  for i in range(n):
    # Generate a random font
    font = get_random_font()
    #download the font file
    font = requests.get(font).content
    #save the font file
    with open("fonts/" + str(i) + ".ttf", "wb") as f:
      f.write(font)
    # Generate a random font size
    font_size = random.randint(10, 14)
    # Generate a random digit from 0 to 9
    text = str(random.randint(0, 9))
    # Generate a random rotation
    rotation = random.randint(-180, 180)
    # Generate a position for the text depending on the font size
    position = (random.randint(10, 28 - font_size), random.randint(10, 28 - font_size))
    img = Image.new("RGB", (28, 28), color=(255, 255, 255))
    draw = ImageDraw.Draw(img)
```

Figure 23: Choosing all images parameters

Nevertheless, to draw the digit, we need to download the file of the font. To do so, we create a folder where we temporally store the font file in TTF format. Once it has been done, we go through the algorithm of the image creation. Once the image has been created, the digit has been drawn, the image has then been saved to the "out" folder, we can delete the corresponding file.
We got into some issues of files not having numbers for instance so we needed to try the files and catch errors if any in order to not stop the global process of image generation.

```
    try:
      draw.text(position, text, font=ImageFont.truetype("fonts/" + str(i) + ".ttf", font_size), fill=(0, 0, 0))
    except:
      os.remove("fonts/" + str(i) + ".ttf")
      continue
    image = img.rotate(rotation)

    image.save("out/" + str(text) + "_" + str(i) + ".png")

    os.remove("fonts/" + str(i) + ".ttf")
```

Figure 24: Try the font file

Finally, we also wanted to train our model on empty images so that it recognizes empty cells of the Sudoku grid. In order to do that, we just divided the number of wanted digits images by 10 and then saved 28 pixels by 28 pixels images with only a white background. Nevertheless, not all images that we will get will be fully white. So we have decided to add random black pixels to the image. Between 0 and 50 pixels are randomly displayed on the image.

```
for i in range(n//10):
    # save white images
    img = Image.new("RGB", (28, 28), color=(255, 255, 255))
    img.save("out/" + "blank_" + str(i) + ".png")

```

Figure 25: Generating white images.

To further test our neural network and image processing algorithms, we implemented, in python, a generator of sudoku grids. The generator is random and generates a grid with one or more solutions (it is not really important that the solution is unique).

Once the sudoku generated, we are generating an image of this grid. First we draw the grid on the image using OpenCV function. Then we add the numbers in the grid using random fonts making it more challenging. And finally we apply some transformation to the image to make it more realistic. The transformation are random and are applied to the image. The transformation are: 3D rotations, translation, scaling, noising, blur, adding unwanted lines. On top of that, we can add different background textures making it more challenging. Here is an example of a generated grid using our tool:

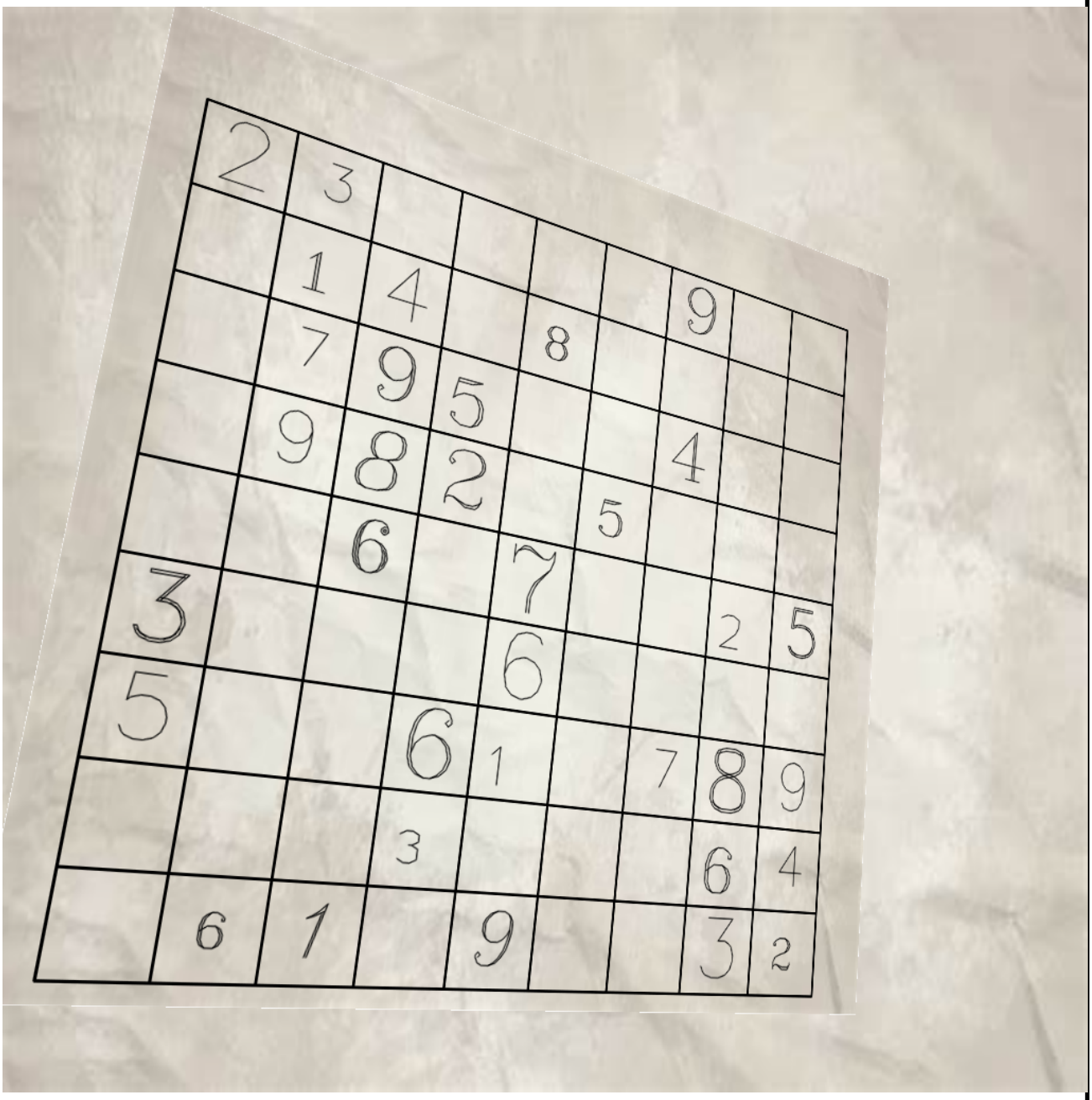


Figure 26: generated sudoku grid image

8 Neural Network

Our main goal on this part is to implement a Convolutional Neural Network from scratch, to be able to achieve this goal, we designed this part carefully to be able to easily modify our neural network architecture.

The Neural Network structure accepts a variety of parameters. The main parameter of the structure is two arrays of layers: one for the convolutional layers, and an other one for the fully connected layers. Each of those layers have an independent set of parameters, we can tweak:

- the number of neurons (fully connected layers) — filters (convolutional layers)
- the activation function along with its derivative

For simplicity purpose, we will consider our neural networks to be sequential, meaning each layer input is the output of the previous one in the list.

8.1 Activations functions

The activation functions are necessary to prevent linearity in the model. For that reason, they are non-linear. Here are some of the most used activation functions along with what they are commonly used for

8.1.1 ReLu

ReLu (Rectified Linear Unit) is one of the most used activation function in deeplearning, it is fast, no heavy computation and simply consists of cutting off every values under 0.
$Relu(z) = max(0, z)$

8.1.2 Leaky ReLu

Leaky ReLu (Leaky Rectified Linear Unit) is a modified version of ReLu. One of the biggest problem of ReLu is the fact that the gradient is vanishing when the activation is negative. To overcome this issue, Leaky ReLu has a small slope for negative values instead of a flat slope, allowing for a better backpropagation of the gradient.
$LeakyReLU(z) = max(0, x) + negative_slope * min(0, x)$

8.1.3 Sigmoid

The Sigmoid activation function is used when we need activation to be between 0 and 1, it is mostly used in multi-label binary classifications, allowing multiple labels to be detected or not at the same time. For example, to detect whether the is a Horse and an Apple in the same image independently from each other. Meaning we could have: Horse and Apple, Horse but no Apple, no Horse and an Apple and no Horse and no Apple.
$\sigma(z) = \frac{1}{1+e^{-z}}$

8.1.4 Tanh

We can see the Tanh activation function as a Sigmoid function but outputting values between -1 and 1.
$tanh(z) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1-e^{-2x}}{1+e^{-2x}}$

8.1.5 Softmax

Softmax is used in multi-label categorical classifications, it does not allow multiple class to be detected at the same time. Meaning we could have either: Horse and no Apple and no Horse and Apple.
It is not suited to a case where there are no Horse and no Apple.
$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$
Note that with big weights, there is a risk that the exponentiation of the softmax activation causes a NaN overflow, to fix this problem, we are normalizing by the

maximum value. This has the effect to shift the values that we exponentiate to a $[-inf, 0]$ interval.

8.2 Fully connected layers

This layer is the most basic one, as its name indicates, its neurons are fully connected to the previous ones, meaning there is a weight for every neuron on the previous layer for every neuron on the current layer. The size of the Weight matrix is thus of shape (num_input, num_neurons). Along with the weights, a bias is added for every output neurons, it is a vector of shape: (num_neurons). With our implementation of matrices, the bias vector is in fact a 2D matrix of shape (num_neurons, 1) or (1, num_neurons). Which are equivalent matrices in row major order.
The formula for the activation or each neurons in a fully connected layer is thus:
$z = \sum_{j=1}^{num\_input} x_j * w_j + b$
z is the neuron activation, x_j the activation of the j-th input and w_j the weight of the link between the current neuron and the j-th input

Note that A forward pass to this layer is simply the matrix multiplication between the input activations and the transpose of the weight matrix. To what we need to add the bias and then apply an activation function to the obtained matrix. We then obtain a Z matrix of shape: (batch_size, num_neurons).

8.3 Convolutional layers

This layer is not in the requirements of the project, but we consider it as a challenge for us to further understand another type of neural networks.

A convolution is an operation that applies a kernel (filter) to a signal to detect features in it. In our case, we use two dimensional convolution. The signal is the image composed of pixels (usually normalized to [1-0]). The main idea of having convolutions in a neural network is to analyze this signal (image) and encode it into a smaller signal with higher depth dimension.

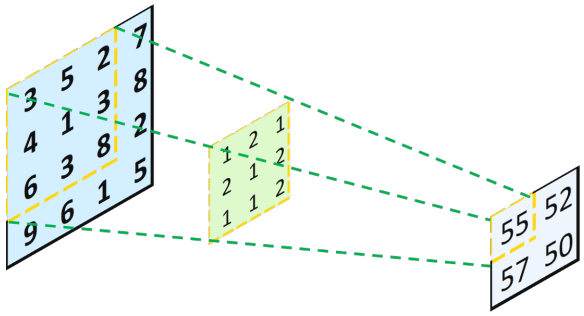


Figure 27: Convolution

Here you can see the application of a 3x3 convolution kernel on a 4x4 signal, the resulting of the application of this filter is a new signal of size 2x2. The resulting signal is smaller than the input signal, but why is that? Simply because the kernel here is only applied on the valid areas of the input signal, so the outside border of the signal is lost. To prevent that, we can add zero values around the input filters

so that the output size matches the input size:

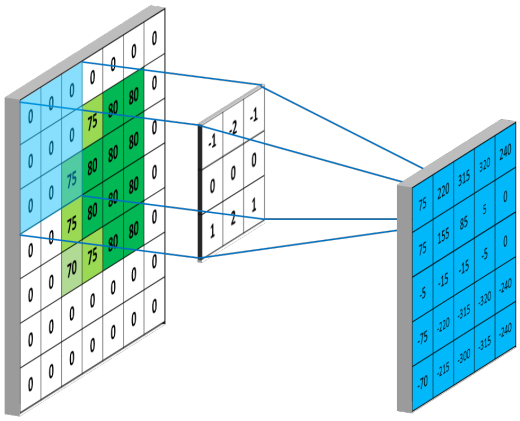


Figure 28: zeros padding

Now, as said previously, the main idea of having convolutions is to reduce the size of our image, this can be done by introducing strides to our convolution layers. The amount of movement between applications of the kernel of the input signal is referred as the stride. On the above illustrations, we had a stride of 1 meaning at each step, the kernel moved by 1 pixel. On the illustration below, a 3x3 kernel is applied to a 5x5 signal with strides set to 2 without padding. This results in a 2x2 output signal:

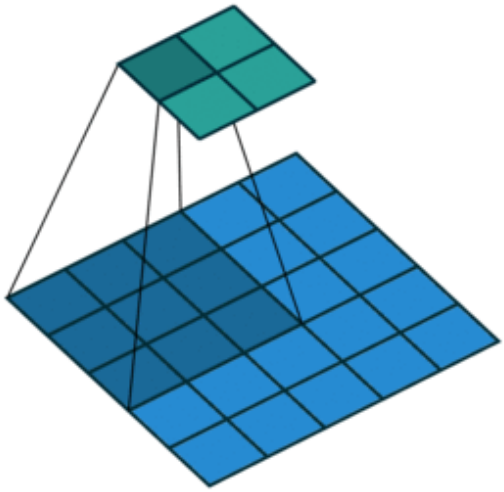


Figure 29: strides

Each filter detects simple features from the previous signal but deepest the Convolutional Neural Network is, the more complex the detected features are. Here is

an example where we are trying to detect the face of a horse in an image: The first convolution layer will detect simple features such as edges on the image. Those can correspond to the shape of the horse as well as the shape of other stuff in the image. The second will have in input the already detected edges, from those edges it could detect sets of edges looking like features coming from a horse: ears, eyes. The third one will from those features detect even more complex features and so on. After the convolutional layers, we are left with something called the latent space of our image. It is the encoded, simplified form of our image where only the key features are left. From those locally detected features, we can then have a look at the big picture by flattening the 2D signals into a 1D vector to be fed into fully connected layers and then answering the question "Is there a horse in the image?" or "Is this image containing a 1, 2 or 3?".

This layer takes as input a 4d matrix. Usually, the first input is an array of image. Meaning a 4d matrix of shape: (sample size, height, width, channels), channels being the number of channels in the image e.g. RGB -¿ 3 and Gray -¿ 1. An example of the shape of an input matrix would be for example: (16, 28, 28, 1) for a batch (or sample) of 16 gray images of size 28 by 28.

8.4 Loss functions

The loss function is used to determine the error between the output Y' of the neural network and the expected output Y (the label). It is a metric to express the distance between the output and the label. The goal of the neural network is to minimize the loss and thus get closer to the result. This is done through the backward pass.

8.4.1 MAE

MAE (Mean Absolute Error) is defined by the following:
$MAE = \sum_{i=1}^{m} |yp_i - y_i|$
with m the number of samples, yp_i the prediction for the i-th sample and y_i the label for the i-th sample.

8.4.2 MSE

MSE (Mean Squared Error) is defined by the following:
$MSE = \sum_{i=1}^{m} (yp_i - y_i)^2$ with m the number of samples, yp_i the prediction for the i-th sample and y_i the label for the i-th sample.

8.4.3 Cross Entropy

Cross Entropy is mostly used to measure the performance of a classification model with output values between 0 and 1: it is the case for sigmoid and softmax activation functions. It is defined by the following:
$CE = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$ with M the number of classes, y the binary indicator (either 0 or 1) and p the predicted probability of class c.

8.5 Forward pass

The forward pass consists of processing layer by layer the activations. A forward pass to a fully connected layer is simply the matrix multiplication between the input

activations and the transpose of the weight matrix. To what we need to add the bias and then apply an activation function to the obtained matrix. We then obtain a Z matrix of shape: (sample size, num neurons).

Here is a forward pass to a simple neural network. We can note that the sample size is conserved through the forward pass and lets us forward multiple sample at a time.

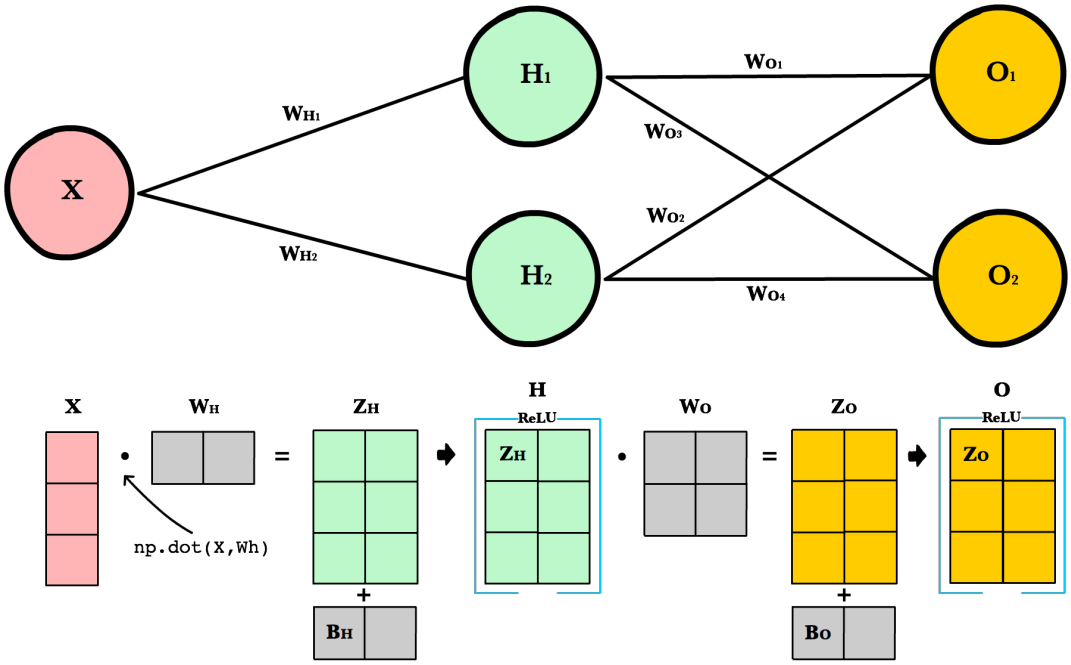


Figure 30: Forward pass

The forward propagation for a convolution layer is pretty similar, but instead of a basic matrix multiplication, we apply a convolution on the input 4d matrix. it results in an other 4d matrix.

8.6 Backward pass

The backward pass consists of back-propagating the error through the neural network in order to correct the weight and biases of every parameters

8.7 XOR neural network architecture

For the architecture of our XOR neural network, we kept it simple. We used the following architecture with a softmax layer at the end, meaning we are having 2 outputs: one for the "0" result and one for the "1" result. To get the final result, we are just picking the index with the maximum value. As an example, if the neural network outputs: [[0.1, 0.9]], the maximum value corresponds to the output "1".

Layer name	input	num neurons	activation
FC Layer1	(None, 2)	32	Leaky Relu
FC Layer2	(None, 32)	32	Leaky Relu
FC Layer3 (out)	(None, 32)	2	Softmax

8.8 Weights and Biases Parser and loader

In order to save our model's weights after training, we did write a simple parser. It saves all the weights and biases contained in a layer, under the name or the index of the layer. Here is the format of the weight file: "dim1 dim2 \n w1 w2 ... wn \n b1 b2 ... bn".
In order to load the weights for the whole network, we just need to init the layer first, then call a simple function that iterates loads separately all the weights for every layers in the neural network.

8.9 Convolutional Neural Network for number recognition

For the architecture of our Convolutional neural network, we wanted something light as well as accurate. We used the following architecture with a sotmax layer at the end.

Layer name	input	parameters	activation
Conv1	(None, 1, 28, 28)	f=8, k=(3,3), s=2	Leaky Relu
Flatten	(None, 8, 14, 14)		Leaky Relu
FC Layer1	(None, 1568)	n=128	Leaky Relu
FC Layer2	(None, 128)	n=64	Leaky Relu
FC Layer3 (out)	(None, 64)	10	Softmax

We only used one layer of convolution with strides equals to two and a kernel of 3 by 3. The output of the convolutional neural network is a matrix with reduced size but increased depth. we went from having an image with depth=1, height=28, width=28 to a matrix with depth=8, height=14, width=14. Then we apply a flatten layer right after the convolution transforming our 4D matrix into a 2D matrix. This is important as our fully connected layers use 2D matrices. We then can proceed to two normal fully connected layers that will solve our NN

9 UI

To design the user interface we used a widget toolkit called GTK. The design of our user interface changed a lot over the course of the last part of the projet, with new changes and features being included and my growing understanding and knowledge of GTK.

9.1 Final design

The final design we settled on is the one shown just after. It is separated in three main areas: the options area on the left, I wonder if someone will actually read this sentence, if someone does well congratz you found one of the easter egg of the report (maybe its the only one good luck) the input area in the middle left, the output area in the middle right.

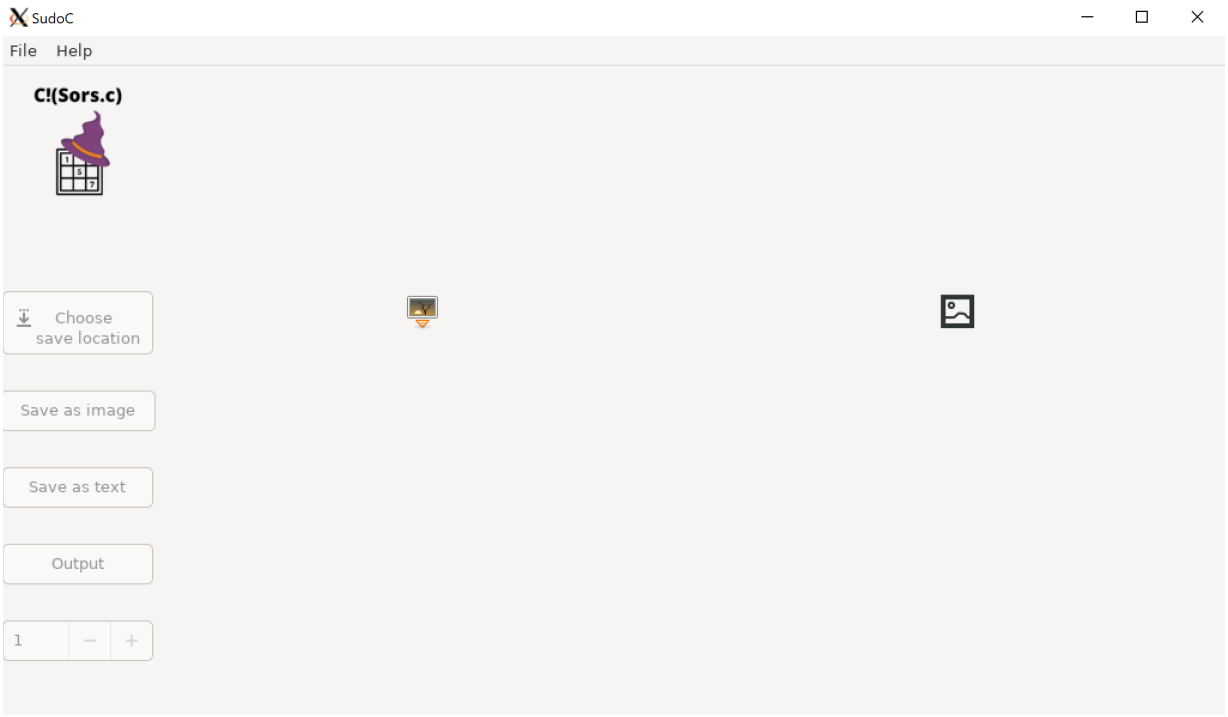


Figure 31: Final design

9.2 Input Area

The input area is comprised of an image slot, by clicking on the image slot the user is prompted to open an image file throught a simple file explorer, then the image is displayed and the button to show the output of the processing. A filter is applied so only the files that are images are displayed on the file explorer.

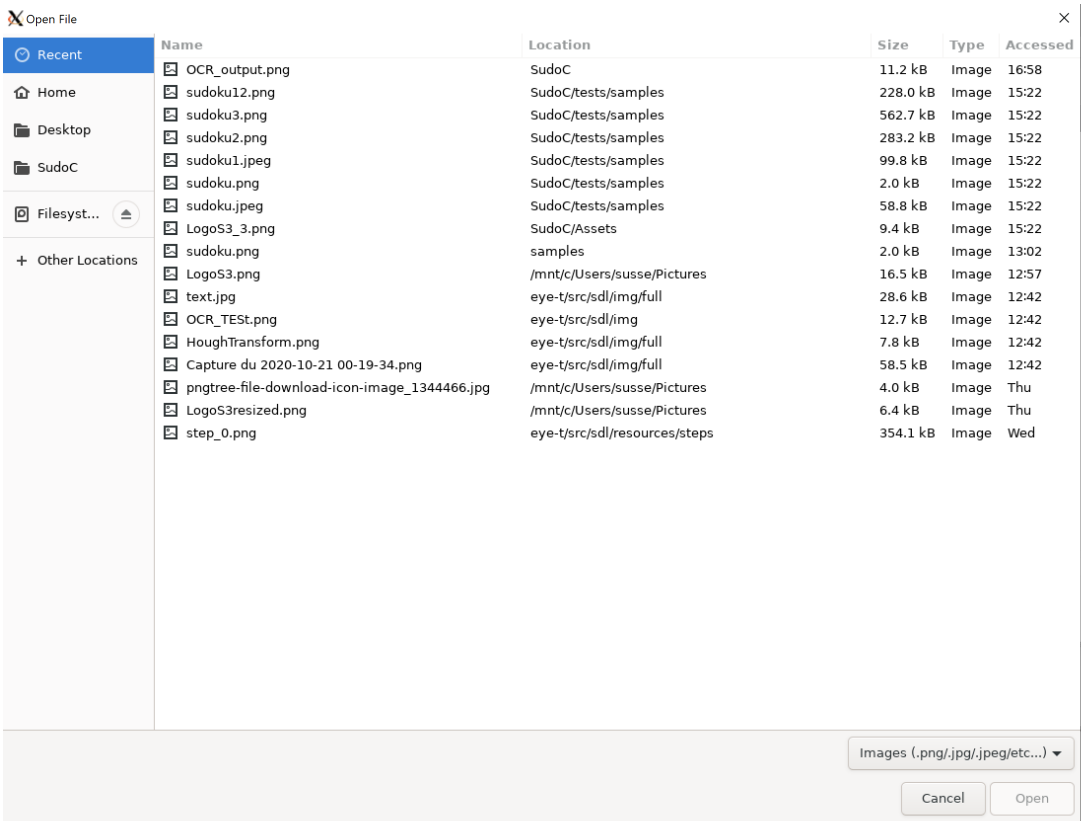


Figure 32: File explorer

9.3 Options Area

The options area holds the different button that allow the user to do a bunch of cool things such as choosing the save destination and name of the file, save the current output as an image, save the result of the sudoku as text or even go thought each steps of the pre-processing so you can see everything that the app do to the image.

9.4 Output Area

The output area allow the user to show the result of the processing of his grid. So there is all the steps that the user can save or visualize and at the end the final one, The users cant save any file until he choose a path. This is how it looks like on an image.

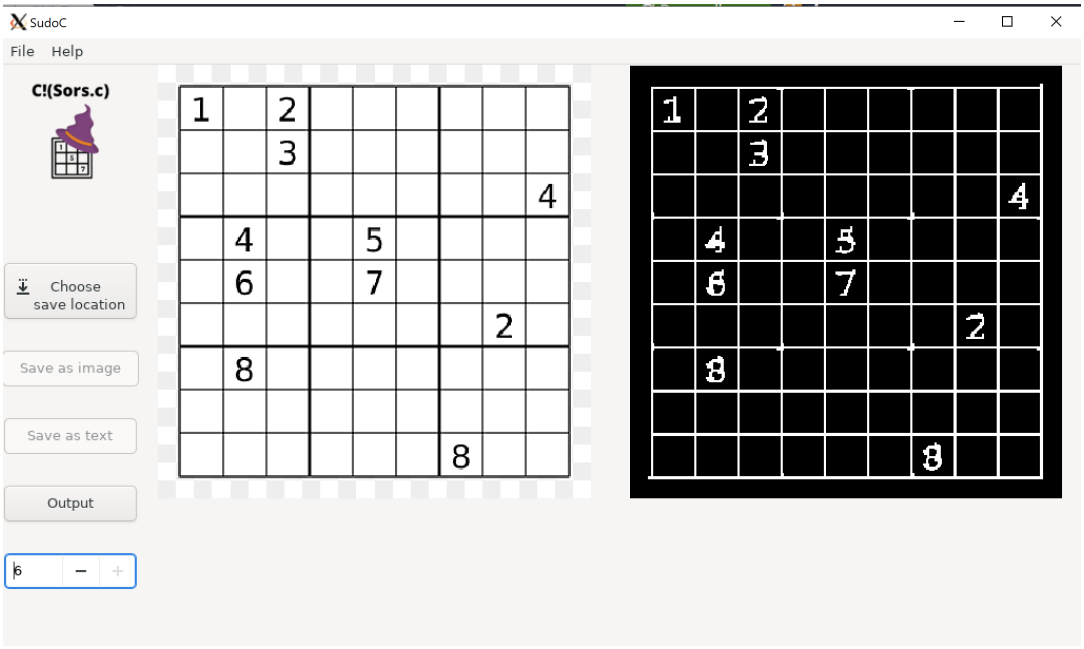


Figure 33: Output Image

9.5 Menu Bar

We also did a little menu that we think really add something to our project such as a menu. So the open button allow the user to choose an input image and quit just quit the window.

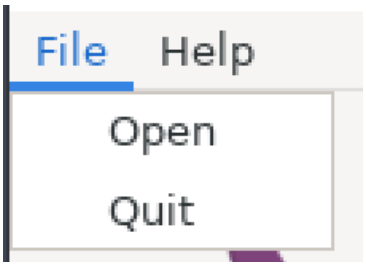


Figure 34: Tool Bar

There is also the help button that open another window with some information on our group. I think that this features is also really important and usefull for such a project that is this sudoku solver.

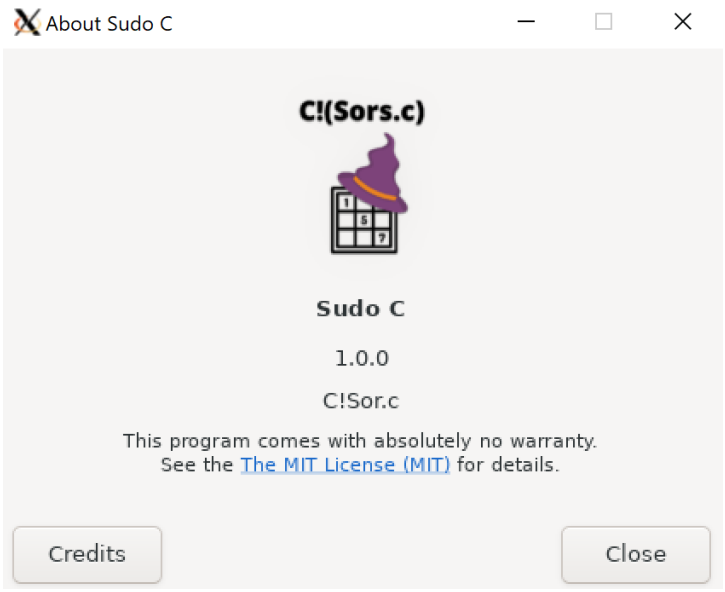


Figure 35: About us

10 Conclusion

Thus, we managed to reach all the necessary steps and sometimes more for this defense thanks to our perfect team cohesion. We are already looking forward to the future and expect to successfully complete this project in the next few weeks as well as add an interface for users wanting to use our magic tool.

List of Figures