S3 Project Report

Sudoku OCR by Nord En Face

Made by :
Alexis Meunier
Ilan Mayeux
Augustin Hubert
Yun Lu
Aurelien Marques

Contents

1 Introduction

Our goal for this defense was to make almost everything that was asked for the second defense, with a few exceptions, such as :

- the GUI
- the digit recognition neural network
- optimizations (neural network + a bit for the solver)

We also tried to make our code very readable and quickly executable, that is why we have easily modifiable code that shows help commands and handles all the possible functions (More on those later).

We are very proud to present our progress up until now!

2 The members of Nord en Face

2.1 Alexis Meunier

I have started programming since high school and loved it ever since, but only this year have I started learning C. Even if it is more complex than other languages that I learned before such as Python, I find it much more interesting. I am enthusiastic about this project because it has deepened my knowledge of the C language, and allowed me to learn new algorithms that are very interesting and useful. I am a little disappointed in the quality and quantity of my work so far, especially compared to my peers, hence I will work more for the second defense so I can be proud of what I will have done !

2.2 Ilan Mayeux

I am wandering as Ilan Mayeux. A second-year Epita student from the A2 class. Previously from Le Bon Sauveur highschool, I learned precious knowledge during my scholarship and built a passion in computer science as a result. Neuron Networks and Image manipulations are subjects that I always tried to avoid. This project allows us to perfect our skills in C and discover new algorithms. In high school I learned about neuron network and image manipulation and I hardly understood. This round two is an opportunity for me to fight again. Yet it seems that I am once again not ready as my network implementation is forcing me to face logical bugs, the worst ones to solve. I hope I will manage to overcome all of that and proudly succeed with my team.

2.3 Augustin Hubert

Since I was young I always found myself solving puzzles or try to understand every algorithm/mechanism I encountered. Therefore I instantly loved my calculator Ti-82 stats.fr when I understood how to create programs on it. This was my first experience as a programmer on what was an assembly like language. Since then I how to code properly on python, c# and obviously as of today c. I was very enthusiastic to learn more about a low level programming language and even more so on how we are tasked to do an entire project in c. While I tried to understand what everyone was doing on this project, I focused myself on preprocessing and the grid detection. As last words I really liked working on the project so far and will try my best to finish it cleanly.

2.4 Aurélien Marques

I started programming a long time ago, back when I was in middle school (collège); Scratch was introduced to me by a friend, we made all the basic games such as platformers and mazes. I was then introduced by my friend Luc to C#, I really enjoyed it and the easily usable Unity made it even better. Thanks to classes and delegates, I felt like everything was possible. But then, an event changed my life: I learned about C, I instantly fell in love, the simplicity of the language, combined with the endless possilbities of C# showed me that anything was possible in code, it was beautiful.
When I saw that the OCR we had to do was in C, I was very happy and that's why I really enjoyed this project. My personnal favorite was the neural network, even though I didn't work on it for a long time, it was still fairly enjoyable since it is mainly maths, and I really appreciate mathematics.

A fun fact about myself : I ***really*** like pasta

2.5 Yun LU

3 Task Distribution

Tasks	Alexis	Aurélien	Augustin	Ilan	Yun
Preprocessing					
Gaussian Blur		✓			
Grayscaling			✓		
Adaptive thresholding		✓	✓		
Sobel Edge Detection			✓		
Grid Detection					
Hough Transformation	✓	✓	✓		
Line Filtering			✓		
Square Detection/Selection			✓		
Segmentation					
Image Correction	✓				
Image Crop	✓				
Image Resizing	✓			✓	
Image Rotation		✓	✓		
Image Splitting	✓	✓			
Neural Network					
Gaussian distribution	✓				
Neural Network XOR				✓	
Neural Network Digits		✓		✓	
GUI					
GUI		✓			

Table 1: Final table of role divisions

4 Our steps for the project

To make the OCR work, we aimed for the following steps:

- (Optional in the GUI) Manual Rotation
- Get the image from a file, turn it into a SDL surface
- Grayscale the image and normalize the brightness
- Blur the image a bit (removes a bit of random noise)
- Apply adaptive thresholding (get binary values for pixels)
- Invert the colors of the image if necessary (we want a white grid over a black background)
- Erode the image (get rid of isolated pixel)
- Apply sobel gradient (gives only the edges of an image)
- Apply hough transform (gets all lines on the image)
- Correct the perspective/rotation of the image according to the grid found
- Split the image into 81 sub images
- Extract the Digit part of each split image
- Feed the 81 images to the neural network 1 by 1
- Construct a new 9x9 matrix of numbers
- Feed this new matrix into the solver
- Construct a new image of the solved sudoku

5 Preprocessing

5.1 GrayScale

5.2 Normalize Brightness

5.3 Level Pixel Intensity

5.4 Gaussian Blur

The point of this blur is to smoothen the image, and to remove a little bit of spontanious noise. For example, if a pixel is white and is surrounded by many black pixels, it will become essentially black (the tresholding is done after so it will remove this white pixel).
To do this, the gaussian blur works by replacing each pixel by an average of the neighbouring pixels, each of which will have a weight attached to it, depending on how far this pixel is to the pixel we are setting.

The gaussian blur function takes 2 parameters, size and sigma; size controls the size of the kernel, for example, in the image below, size = 5. Sigma is used to control the steepness of the curve; a high sigma will increase the center weights and reduce the outer ones; while a lower sigma will spread out the weights.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

5.5 Sauvola Thresholding

At first, we used a normal thresholding, which was very basic: loop through every pixel, set it to 0 if the red value is under 127, and set it to white otherwise. This had a big problem: if the image is rather dark, but still readable, the normal thresholding would return an image which is almost only black.

To fix this issue, we turned to adaptive thresholding, a method which is similar to thresholding, but instead of using 127 as a threshold, we used the average of the neighbouring pixels, so that if most pixels are black, the threshold will be lower, and we will still be able to have lines and numbers come out of the resulting image.

5.6 Filter : Erode/Dilate

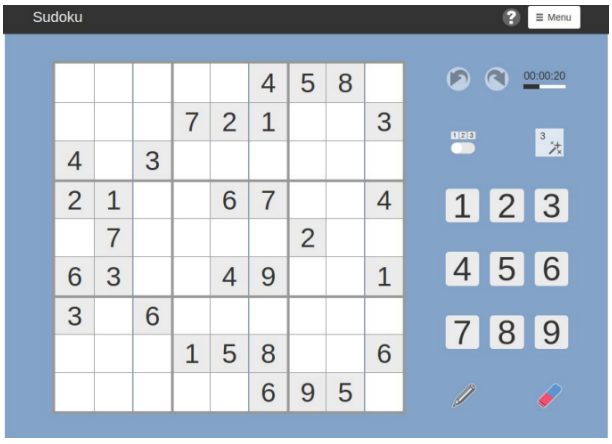
A simple erosion algorithm is used to reduce noise on an image, it uses an algorithm similar to Conway's game of life (uses neighbouring cells): if a white cell is around enough white cells, it lives, however, if it is isolated, it becomes black. This algorithm is used since adaptive thresholding may result in a few random points appearing. This will smoothen out the entire image.

6 Detection of the grid

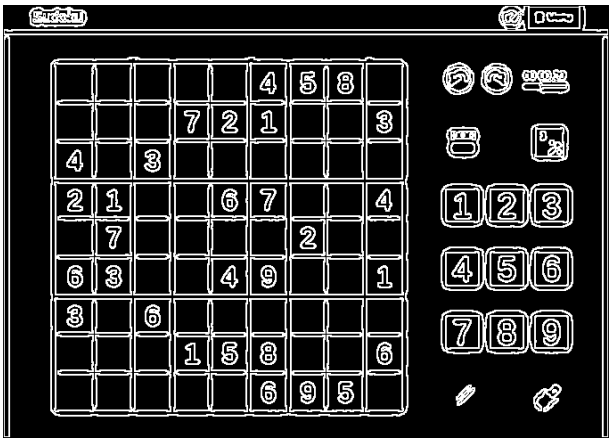
The detection of the grid was split into 3 parts, sobel, hough and the detection at the end. The goal of sobel is to find the edges of the image, hough turns those edges into lines, and then turn the lines into squares and find the best square.

6.1 Sobel Edge Detection

The sobel edge detection takes a black and white image and tries to find straight lines, this will, for example, turn this image :



into



The algorithm is similar to gaussian blur, but uses this kernel instead of a 2D gaussian curve.

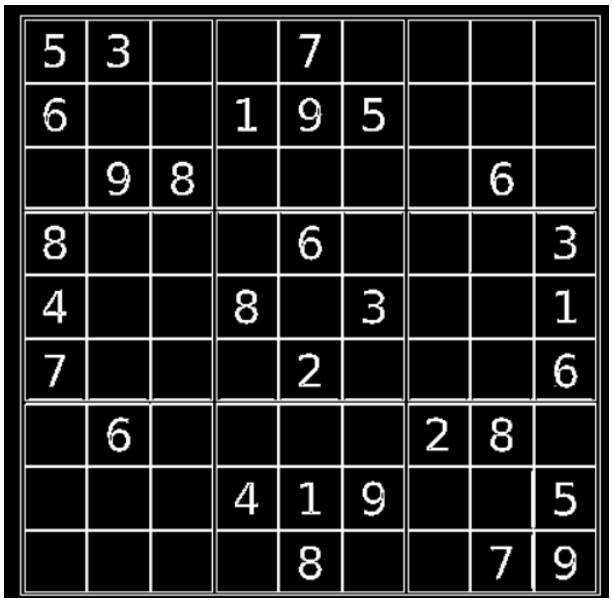
$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The algorithm behind is trying to detect high deltas along the x and y axis in the image. Then it averages the deltas found in the x and y axis and if this value is more than 127,then the pixel of high delta is set to white.

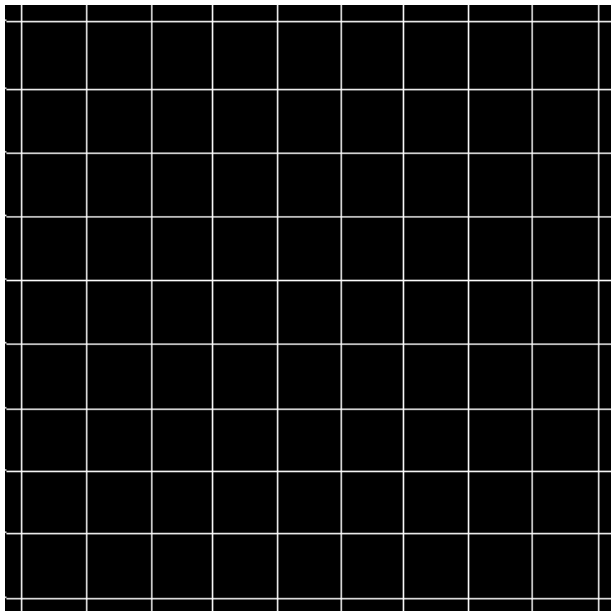
6.2 Hough Transform

Now comes the fun part of the grid detection, the hough transform (pronounced houg with a russian accent). This transformation is supposed to take an image and returns only lines that form this image, so for the sudoku, it will return a new image with only the grid.

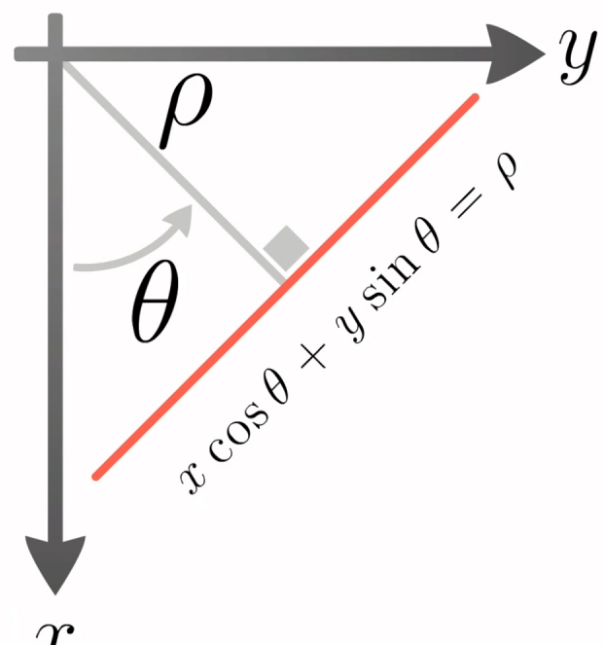
For example, it will turn an input like this :



Into this :



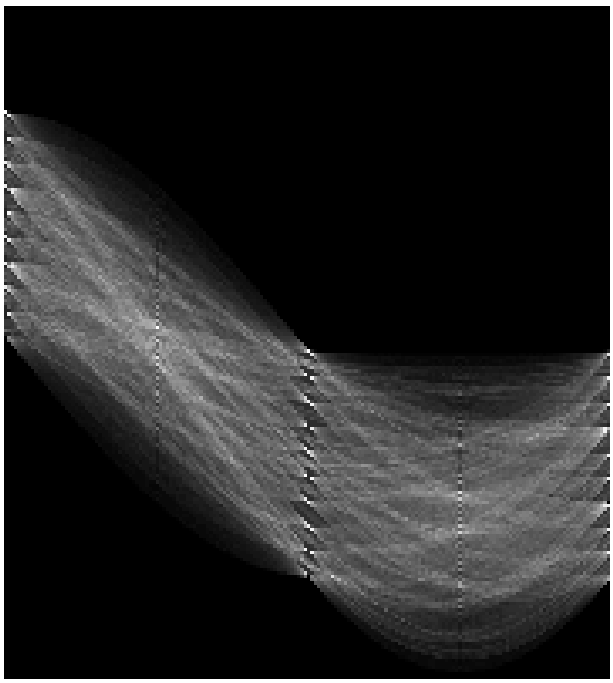
To do so, Hough uses the fact that a line, in a (0,x,y) plane, can be written as a function y = ax + b with a the slope and b the distance to the origin, which translates to b = -xa + y in a (0,a,b) plane. To find a and b, we need to use another way to express a line, which is its distance to the origin, and the angle at which it is by a perpendicular line, as can be seen in this image :



Therefore we now have a way to find each line that goes through a pixel, we then have to create an accumulator which will work as a (0,a,b) plane using a matrix. Every point of this plane represents a line in the conventional (0,x,y) plane. For example the point at coordinates (50,90) is a single line that is 50 pixels away from the origin with an angle of 90^{o}.

Now the Function will travel through every pixel of the image, and each time a pixel is white or very close to white, it will iterate using a theta from 0 to 180 (to compute every angle), and compute a corresponding rho with the formula : (x*cos(theta) + y*sin(theta)) with (x,y) the coordinates of the pixel. Doing that gives us EVERY line that goes through this point (even those we don't really care about). Then we increment the value of coordinate (rho,theta) in the accumulator.

At the end of this function, some coordinates in the accumulator have bigger values, here is the accumulator represented with every white pixel a value. The whiter the pixel the bigger the value :



This means that there are many points that have the same line going through them. Hence we found the lines we want to show, because they have the most points aligned, they will give the lines of the image as shown before.

You can think of this image as a 2D graph of a function, with the x axis being an angle going from -90° to 90°, and the y axis being the first point hit by a ray coming out from any point, the ray is coming out at an angle x (the x axis controls the angle)

To retrieve all the important lines from the accumulator we first look for its maximum, and then gather all local maximum in the accumulator higher than a threshold being a percentage of the maximum. This will leave us with all the prominents lines in the image. As a result, the Hough Transform function gives back a linked list of line defined by a rho and a theta (lengths and angles)

It is noteworthy that due to the nature of this algorithm, it relies heavily on the input image, but is also a core part of this OCR. Meaning this function first needs close to perfect preprocessing to find a correct image with clear lines and not too much noise, but it needs to be nearly perfect itself as finding wrong lines, or not finding good lines can break the Grid detection

6.3 Line Filtering

Because the list of lines given by Hough Transform may varies in quality when it is used to look for quadrilateral, the list is filtered to get rid of useless lines
The first thing is that we look for each line in the list, all the lines that are similar (in angle and length), and averages them in a 'new' line. We then deletes all the duplicates. This helps a lot afterwards when looking at the intersections of the lines.

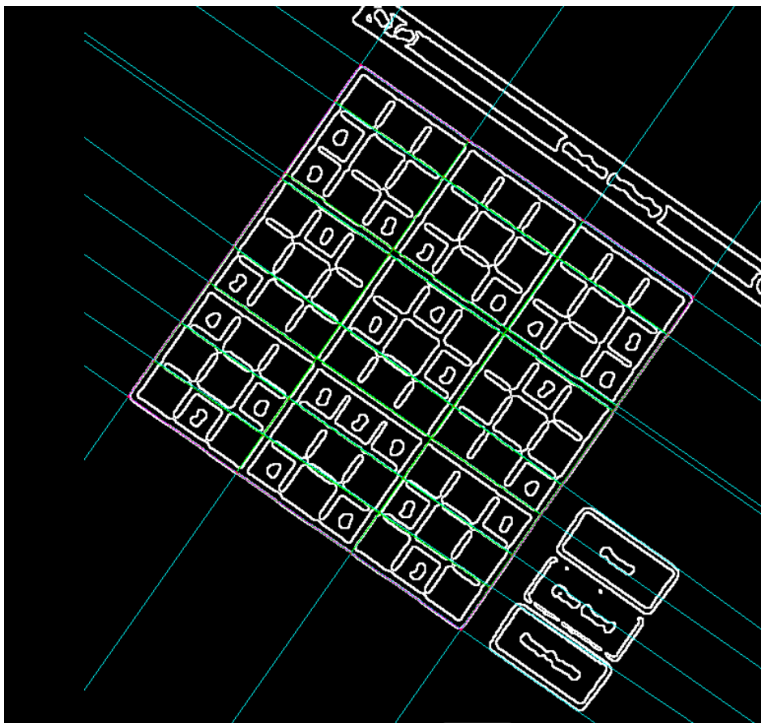
To further filter out any useless lines we make an histogram of the angle of all the lines. This histogram then recompile the number of lines per angle. We then find the maximum, such angle is then considered the angle of grid. Thus all lines that are not sufficiently parallel or perpendicular to this angle are discarded.

6.4 Grid Detection

First, to detect the grid, we had to detect every square (or shape that vaguely looks like a square) in the image, only using a line list. To do that, we used 4 nested for loops, which will check for every 4 tuple of lines (without duplicates) if they form a square. If they do, we add that square into a linked list.
Once we have a quad list, we calculate a ratio for each quad according to its:
- area, (since we are looking for the largest quad)
- "squariness" of the quad (the ratio between the its smallest and biggest side)

We then pick the square which is the largest, but at the same time it must be very close to a square.

These functions took a lot of time to write, but at the end, thanks to good visualizing scripts to help test, they work pretty well.



Here the cyan represent all the lines found, the green represent all the quads found, and the magenta is the biggest square found (not very visible but it is correct!)

7 Segmentation

Having found our grid, the hardest part has been done, but we still need a little more work to adapt the image since we have to rotate the image to make it not sideways, that way the digit recognition would be easier and correct more often. Furthermore we need to crop the wanted Quadrilateral since it is the sudoku grid we are looking for, and we only need the grid and not outside shapes. We also need a resizing algorithm for the neural network, and a perspective correction for images that have been taken sideways or at weird angles. Lastly we need to split the image in 81 squares for each squares of the sudoku to give it to the neural network so it can find the corresponding value for the solver.

7.1 Image Rotation

7.1.1 Manual rotation

To rotate an image, we first had to be sure of what we wanted to rotate. Since we are using SDL to handle images, we decidede to use SDL surfaces; they are fairly easy to manipulate and hold an array of pixels (Uint32's). So, we first had to understand how to represent rotations mathematically, fortunately for us, it could be found easily on the internet :

$$x' = x\cos\theta - y\sin\theta$$

$$y' = y\cos\theta + x\sin\theta$$

So, for each pixel on the original image, we added the same pixel at the new coordinates, using theta as a parameter of the function. However, this had the downside of sometimes mapping 2 different pixels to the same new pixel. On top of that, there would be blank pixels (no pixel maps to it).

To deal with all these problems, we decided to change our view on the problem: instead of searching for the "image" of each pixel on the original image, we could search the "preimage" of each pixel on the new image. Thanks to this, every pixel was filled with a pixel from the original image, appart from the out of bounds pixels, which appear on the corners of the image (but these can be filled with black easily).

We also had some issues with angles that were too big, it was because we forgot to put an absolute value for the height and width of the rotated image, leading the image to shrink after each rotation, eventually leading to a segmentation fault. The fix was easy and it has been corrected since.

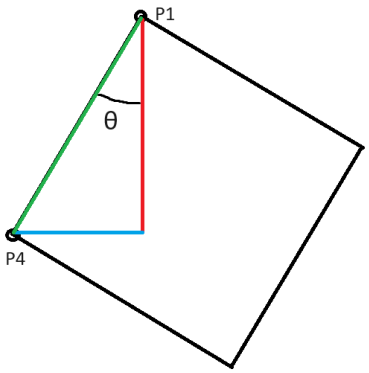
7.1.2 Automatic rotation

To rotate automatically an image, we first had the following idea, we can create a function FindSkew which would give a number (int) to every image, this number would try to represent the amount of rotation it has, the higher the number, the closer to being correct it is.

Our first idea was to make a function which loops through all pixels, and if a pixel is white (filled), it checks it's four neighbours; if it's top or bottom neighbours are also white, then we add 2 to a global counter, if the left or right pixels are white, we remove 1.

This should try to give bigger numbers if most lines are vertical on the image. This, however, is not a very robust system since a large vertical bar (which is not part of the sudoku grid) on the sides of the image can very easily create large numbers, even though the image is rotated.

In the end, with the system not being robust enough (and after another few failed attempts at rotation), we decided to try to make the rotation work with the grid we found. After the square detection we have a quadrilatral (from the struct Quad), since we have all 4 corners, and the first corner is sure to be the top left one, we can check which rotation will have the least angle (since we assumed that the rotation would be less than 45° by default thanks to the manual rotation at the start. The rotation is then straightforward using a bit of mathematics/trigonometry.



We know that $\cos(\Theta) = \frac{adjacent}{hypothenuse}$ which implies $\Theta = \arccos(\frac{adjacent}{hypothenuse}) = \arccos(\frac{|p1.y - p4.y|}{\sqrt{|p1.y - p4.y|^2 + |p1.x - p4.x|^2}})$

7.2 Resizing

7.3 Croping

7.4 Correct Perspective

7.5 Image splitting

For the spitting part of this project, we had to split an image which was assumed to be of the right scale into 81 subimages, which would then get treated individually by the neural network.

This task had us struggle quite a bit since we first tried to use malloc to create a list of matrices of pixels. Unfortunately, Alexis and Aurelien started the splitting before the lecture about malloc, so we did not really understand how/when to free, how malloc really worked, and had no idea calloc and realloc existed. Our code was not very good and had many memory leaks as a consequence.

The first version (which did not work) used a pointer to 81 pointers of Uint32. It worked by iterating from 0 to 9 for an i and a j, then creating a pointer to Uint32, and iterating over a (h = i*height; h < (i+1)*height/9; h+=width) and a (w = j*width/9; w < (j+1)*width/9; w++) and then setting the pixel of the pointer to the

$(h + w)^{th}$ pixel of the original image.

This didn't make any sense ? Well it didn't make any sense for us either, and for the computer either apparently because we got the most monstrous image know to mankind as a result. At first we tried debugging it, changing values, changing something, hoping it would miraculously give us correct images and no segmentation faults but to no avail. Well, the values are so weird, what could we hope for, we're not even sure why we did it like that. Hence, after a few weeks (after the lecture about malloc), we revisited the code to make it work (we managed to remove the use for malloc completely) without any memory leaks or issues.

After some cleaning up, the code works with two nested loops which both loop from 0 to 9 (i < 9), and for each of those, we loop through 1 ninth of the width and height, which makes it so we loop through the whole image, but segmented into 81 loops, we can then create new images and save them in a path, to which we add an index (path_01 - path_81)

7.6 Extract Digit

8 Neural Network

8.1 Theory

In order to recognize the digits written on the Sudoku, a neural network is necessary as handwritten digits are not all the same. The Sudoku contains digits from 0 to 9 and, fortunately, composed digits are not part of it. This mean it is unnecessary to try to recognise negative numbers or numbers above 9. This simplifies a lot the problem. You may wonder why not simply write a code to "guess" what digit is written? The idea makes sense. Yet, how? Recognizing digits is something natural for us, human. But how to write a code that does that? Hard coding everything? It would most probably finish into a terrifying spaghetti code that does not work. Hence, Neuron Network is a solution to these kind of problems were written a solution is out of the scope of a mathematical representation of a problem.

A Neural Network is a structure made of neurons. A neuron is a representation that holds a bias and have weights for each input it has. When given inputs, it gives out an output. We call them perceptrons.

Perceptrons alone are meaningless. It is like a brain neuron. It goes by hundreds to finally start playing a role. Hence, a neural network is composed of layers with a certain number of neurons in each. Each neuron of a layer is connected with the previous one and the next one. A neural network is travelled from left to right and, unlike the human brain, each layer is visited once. There are no loop.

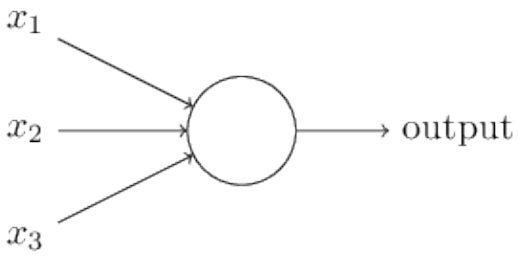


Figure 1: A perceptron with three inputs

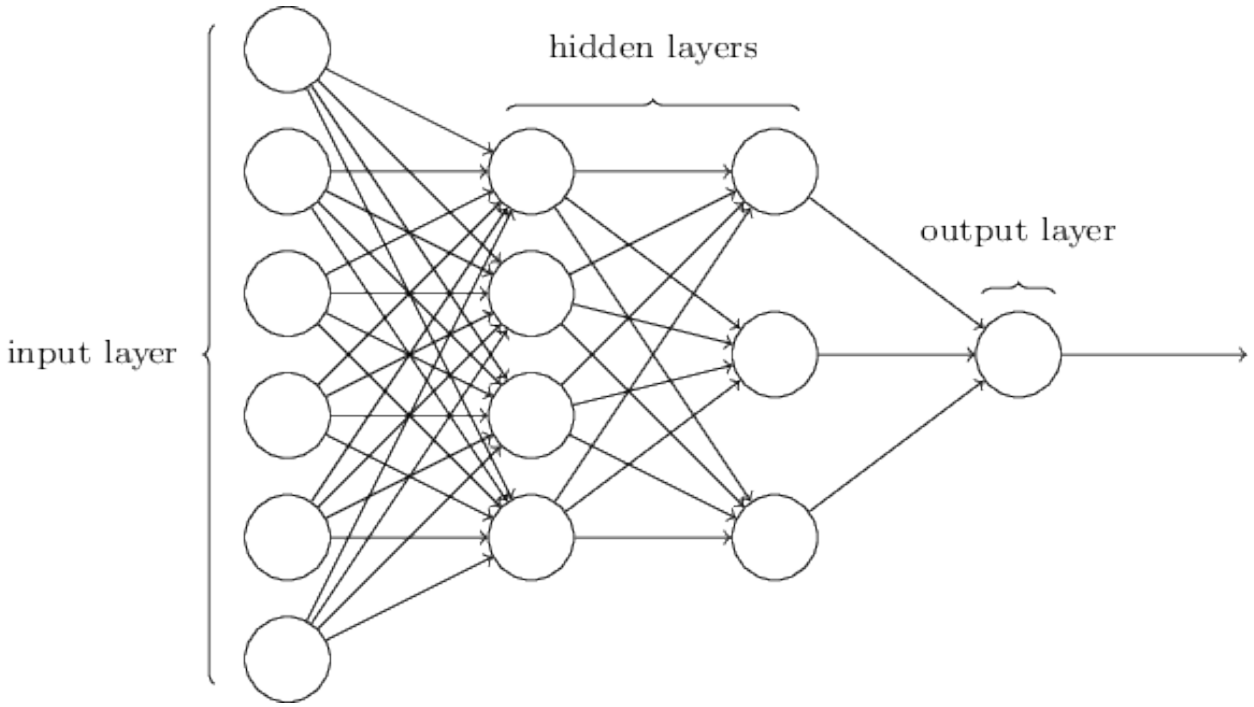


Figure 2: A neuron network with four layers 6, 4, 3, 1

The input layer is what we want to feed to our network. In our case, the image to analyze It has no weights nor biases. The output layer is the answer, what digit it thinks it is. The hidden layers are intermediate layers that are here just to give more output possibilities, hence sometimes better results. Each neuron compute its output by adding the product of the input neuron and its corresponding weights and after adding the bias.

The difficult part is to attribute each neuron the perfect weights and bias so that, in the end, it gives the expected output. This process is called the training part. We will feed the network with many inputs and compare the output with what we expected. Then, modify a little each neuron so that it gets closer to the value it should have had at each step. This process is called the backpropagation algorithm. The learning rate is what decide how much the value changes to get closer to what it should be.

In our case, this training data will be the MNSIT database. A huge database with 60 000 images containing handwritten digits images and their corresponding value and 10 000 test images to test our neuron network at each step. The images are in 28 by 28 pixels.

Since the database is using a format of 28x28, we will try to do the same to keep the consistency of our training data and test data. Hence, the pre-processing part will have to handout a resized digit image in 28x28, which means that the input layer will be made of 784 neurons. One neuron for each pixels of the image, with the value being between 0 and 1, representing the black and white degree. You may think that the output will

be made of 1 neuron. However, we expect 10 different digits. We would have 4 neurons and play with the binary to decide which digit is up. But the best solution remains having 10 different neurons, and the index of the maximum output being the represented digits.

8.2 Implementing the Network

Implementing the network is a difficult task. We are starting from 0, only with a simple math library provided by the standard C library. To simplify and fasten the calculations, we will be working with matrices. Each layer will have a matrix of weights and biases. If m is the number of neurons in the layer l and n the number of neurons in the previous layer, $l-1$, then the matrix of weights of the layer l is of size $m \times n$ and a matrix of biases of size $m \times 1$.

Therefore, we need to implement many functions to simplify the usage of these matrices such as matrix multiplications, additions, transpose, etc. The first implementation chosen was simply a heap-allocated array of size m of heap-allocated array of size n to represent the matrix where a value could simply be accessed with [i][j], with $0 \leq i \leq m$ and $0 \leq j \leq n$. However, it was difficult to work with as we had to not make any mistakes with the sizes of the matrix for the operations. This is were comes the upgrade of the matrix. A structure that holds both the matrix and its size.

```
typedef struct Matrix
{
    double **matrix;
    size_t m;
    size_t n;
} Matrix;
```

This structure holds all the necessary data and could then easily be used. The other functions were simply upgrade with a version 2 that calls the version 1. For example:

```
/**
  * Calculate matrix product
  * If A is an m * n matrix
  *    B a n * p matrix
  * then AB is a m * p matrix
  * AB is allocated in the memory
  * 1d array as 2d
  */
double **dot_2d(double **a, size_t m, size_t n, double **b, size_t p);
Matrix *dot_2d2(Matrix *a, Matrix *b);
```

It is easy to see that the second version is easier to use than the first one, 2 parameters versus 5. The same was done for the other methods which then lead to a stronger base.

With a strong helper file, we could hence start writing the network code. At first, the structures used were the following:

```
typedef struct NN_Neuron
{

    // Number of input connections to this neuron from the previous Layer
    // Also determine the number of weights
    size_t num_inputs;

    // Used to basculate the output
    double bias;

    // The array of weight from w_1 to w_n to determine the output value of the
    // neuron
    double *weights;
```

```
} NN_Neuron;

typedef struct NN_Layer
{
    size_t num_neurons;

    // array of pointers of neurons
    NN_Neuron **neurons;
} NN_Layer;

typedef struct NN_Network
{
    size_t num_layers;

    // array of pointers of layers
    NN_Layer **layers;
} NN_Network;
```

This was the one used until reaching the back-propagation algorithm part. It was starting to be difficult to handle them and the code was messy or useless transformations were made. It was however very easy to manipulate and at least, very clear. We changed and later on, moved to the one used in the book *Neural Networks and Deep Learning* by Michael Nielsen.

```
typedef struct NN_Network
{
    size_t num_layers;

    // contains the number of neurons in the respective layers
    size_t *sizes;

    // A bias for each neuron in each layer in the network
    // a matrice [layer, neuron, singleton]. Singleton is a list of length 1
    // (for matrix multiplication purpose)
    Matrix **biases2;
    // A weight for each neuron in each layer in the network
    // a matrice [layer, neuron, weights]
    Matrix **weights2;
} NN_Network;
```

This implementation uses matrices a lot. biases2 and weights2 are both arrays of matrices. Since the input layer has none of them, it is an array of size $num\_layer - 1$. Many methods were created to manipulate this structure, such as creating it and freeing it. To create a network, you simply need to specify how many layers there are and an array that contains the number of neurons in each, input layer included. There are two main functions that have to be retained: the Feedforward and the Scalar Gradient Descent (also called SGD).

The feedforward is what will calculate the output of the network. It takes a network and an input matrix that will act as the input layer. As an output, it will give back a matrix. In our case, the input will be a matrix of size 784×1 and the output 10×1. Each neuron will determine its output based on the inputs, weights and bias until reaching the end. You could then determine what digit is it by applying the "argmax" function to the output matrix, which gives back the index of the maximum in the matrix.

The Scalar Gradient Descent is the function called to train the network. It takes as parameter the given network, the training data and its size, the number of epochs, the batch size, the learning rate and the test data and its size. The training data is what was mentioned in 6.1 with the MNIST database of size 60 000. Same for the test data, the 10 000 samples. The number of epochs is the number of time the whole data-set is tested. For example, 200 epochs would be 200×60000 samples. The mini batch size indicates how many images we test at a time. Having the results of the delta for the 60 000 samples in the memory would be too heavy. Hence we

split them into mini batches and since we are dealing with an average, we can handle treating them in smaller portions. The learning rate was mentioned above. It describes how much each sample is allowed to change the weights and biases. SGD is calling other functions, such as the back-propagation algorithm we talked about in 6.1.

8.3 Saving and importing the Network

Having a functional neuron network is a great thing. Yet, it takes time to produce. A lot of time. Too much time for user-friendliness. And is not always giving out the best results. Hence, we would rather ship the best network we produced with the app instead of creating one at every start. This mean we need to save and load our network into files, more precisely, writing the NN_Network structure.

First, we have to think about what kind of file we want and its goal. In our case, loading it. We won't modify it from the file as it may break everything apart or lead to unexpected behaviours. This mean that binary file is the best choice. It is unreadable and discourage its direct modification. Moreover it takes less space than writing into a readable format, useful as it can takes a lot of space with a complex network.

Secondly, in which order should we write the fields so that it is easier to read it? In fact, the answer is easily answerable but still required to be thought. The *num_layers* field would be the first to be written as everyone depends on it. The *sizes* only depends on *num_layers* so it would be written afterwards. Then, since both *weights* and *biases* depends only on *num_layers* and *sizes*, the order does not matter. We hence are written the fields in the following order: *num_layers*, *sizes*, *weights*, *biases*. To handle the matrices, we first write the sizes (rows then columns) then followed by the values in a row-wise order.

Finally, to conclude the writing process, we need to choose an identifier, a signature to our binary file and to directly inform the user that the file given is wrong and not in our format. This is where our magic number plays a part. We chose 8629. It is magic! Written beforehand the network, during the reading process, if we don't read it correctly, then we know for sure that the whole file is wrong.

In order to read our network, we simply have to do the inverse operation. First read the magic number, then the number of layers followed by the number of neurons in each layer. With that, we can allocate enough memory for the rest of the network. Continuing by reading the list of weights matrices and then the list of biases matrices. We can then close the file and return the freshly allocated neuron network.

8.4 Reading the MNIST database

The MNIST database isn't written in a regular format. You may even think at first it would simply be a zip file of the 70 000 examples and tests combined images. Yet, as you expect, it would have taken a LOT of space. There are a lot of unnecessary things in our example that image format carry. The MNSIT database is hence composed of 4 files.

- train-images-idx3-ubyte: training set images
- train-labels-idx1-ubyte: training set labels
- t10k-images-idx3-ubyte: test set images
- t10k-labels-idx1-ubyte: test set labels

Each image has its corresponding label. The labels file are written in the following way:

Listing 1: labels-idx1-ubyte

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  60000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label
```

The labels values are 0 to 9.

Listing 2: images-idx3-ubyte format

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).

From these descriptions, available on `http://yann.lecun.com/exdb/mnist/`, we can determine how to read these binary files. If we want to read the 32-bit integers, we have to be careful because the bytes are written in the Most Significant bits (MSB) format while the C language is working with Least Significant Bit (LSB) format. This mean that when dealing with these numbers, we have to change the order of the bytes to get coherent values.

Thus, the difficult part is done, we simply have to read the images and their corresponding labels one by one. We can read 784 bytes and put it into the matrix since we know the dimensions. In fact, this format for the matrices is identical to the one used in the saving/loading process.

8.5 XOR Network

The XOR network is a basic test to determine whether or not our current model is capable of learning. The XOR function $\oplus$ takes two inputs and gives one output. There are 4 different cases for this function:

- $0 \oplus 0 = 0$
- $0 \oplus 1 = 1$
- $1 \oplus 0 = 1$
- $1 \oplus 1 = 0$

So these four cases will be our training data. This case is different from the next one as we are working with a finite number of cases, which is easier. Since its finite, our training data is the whole set of possibilities, and our test data the same since we only have four cases.

Now the goal is to design our network. We know that the XOR function takes two inputs and so will our input layer. Concerning the output layer, similarly to the digit network, it is natural to think that it would only have 1 output neuron. However, for the same reason as the digit network, having two outputs and taking the position of the maximum as the result is the easiest way to make it learn efficiently. We will also have a single hidden layer of two neurons.

Therefore, the network will look like this:

If the output element 0 has the highest value, then the result of the operation is 0. The same way for the 1, the result would be 1.

Listing 3: Teaching the network to learn the XOR function with 300 epochs.

```
========Epoch 0/299, 75.00% (3 / 4)=========
...
========Epoch 26/299, 50.00% (2 / 4)=========
========Epoch 27/299, 50.00% (2 / 4)=========
========Epoch 28/299, 25.00% (1 / 4)=========
========Epoch 29/299, 75.00% (3 / 4)=========
...
========Epoch 180/299, 75.00% (3 / 4)=========
```

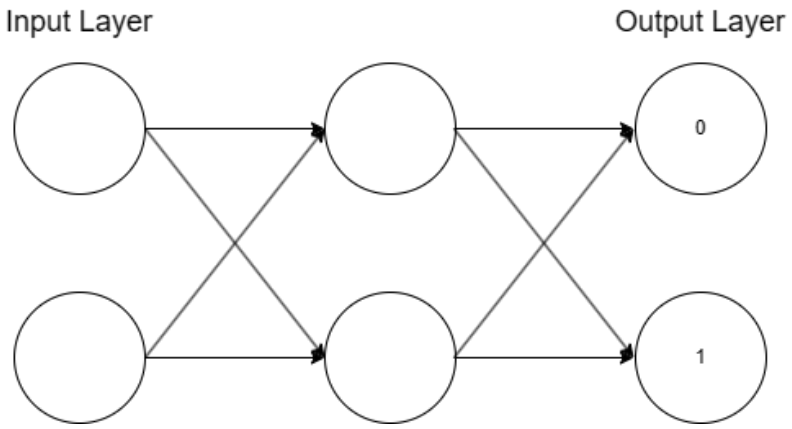


Figure 3: The chosen XOR network layout: 2 2 2

```
========Epoch 181/299, 75.00% (3 / 4)=========
========Epoch 182/299, 75.00% (3 / 4)=========
========Epoch 183/299, 100.00% (4 / 4)=========
```

8.6 Handwritten Digits Recognition Network

The Handwritten digits recognition network is the key to detect what digits are written on the given images. As mentioned in the theory and implementation part, we are using a 784 30 10 network and the training data given by the MNIST dataset.

Since the creation of a network is easy and that the MNSIT reader is done, to try out the current implementation of our network we simply have to put all the pieces together. However, we are facing a few issues. The first one is that the network speed was sacrificed for readability and debugging purposes. The second one is that, after starting the training and waiting 30 minutes for 30 epochs, the result reached a maximum of 70% of good answers over the 10 000 tests. This result is way too low for our final product as it means we still have over 3000 images that were not recognised correctly. The goal is to reach above 95% for the next defense but this means, trying to find out where the calculation are not going according to our plan.

8.7 The network... reborn

After many many hours of work by Ilan, I (Aurelien) decided to take his place as the main person on the neural network and I tried to fix the last issues with the code. Unfortunately, after a lot of work, I was not able to find the mistake, so I restarted the entire neural network. To keep it simple since I was very much limited in time, I only worked with the first chapter from the neural network website which was advised.

The initial phase of coding a neural network was akin to assembling the pieces of a complex puzzle. I started with defining the architecture, deciding on the number of layers, and selecting activation functions. However, it wasn't long before I realized that the success of my neural network was mostly due to the training data.

One of the foremost challenges I encountered was sourcing and preparing the training dataset. Neural networks work with diverse and representative data that encapsulates many scenarios (here images) the model is expected to encounter. Obtaining a dataset that could be used was fairly annoying. At first, I did not want to use the MNIST database since it seemed quite annoying to decode (it is not just 60000 images in a zip unfortunately). I decided to use another data set, composed of "only" 3000 images, and was annoyed to find that my neural network started at 20% accuracy to reach a mere 70% after 30 epochs, the reason for the low number was that the program did not have enough test cases to be accurate, except when given the EXACT image that it had seen in the data generation part. To make the network more robust, I also added noise to

every image, so it would not train on similar images every time, and it would also limit the risk of the network giving flawed results when tested on an actual sudoku, since the sudokus would not be perfectly treated.

Once the dataset was in place, the next problem arrived when I had to code the training phase. Finding good parameters (ETA/learning rate, mini batch size, etc), as finding the right balance between too much of one and too little of the other proved to be very complex. The model's sensitivity to the initial weights and biases added an extra layer of complexity, requiring many iterations and tweaking.

Debugging the neural network also presented its own set of challenges. Identifying the root cause of errors, whether they were due to vanishing gradients, exploding gradients, or many other flaws, most of which I thought were impossible, demanded a deep understanding of neural networks, but I did not have nearly enough time for that during a normal day, so it ended up taking me most of my time for a few weeks. But at the end of the day it was worth it!

At the end of the day, the code very much resembles Ilan's, but a few matrix operations were made simpler, and many structs were used to create an almost python like code.
There a still a few differences, mostly in naming conventions (Network instead of NN_Neural_Network, column matrices instead of row matrices, etc...)
To conclude this part, I would say that I am very proud of my work, the result (90+% on the 1st epoch) is very satisfying and works fairly well for a first neural network. This network was also made possible by the many bases Ilan had set up previously.

9 Sudoku solver

9.1 Parsing

The parser is fairly straightforwards, we loop through a string (obtained by getting the contents of the file given as parameter), and fill a jagged array with values from the text.

The unparsing part was also easy : we create a char array of the appropriate length, and loop through our 2D array, while filling to string correctly (adding new lines, dots, and spaces).

9.2 Solver

The solver was the first part of the project that was completed, we had 2 algorithms for the solver, a first which would eliminate many possibilities off each cell, it is very optimized and efficient, but cannot solve every sudoku. For more complex sudokus, a backtracking algorithm was put into place.
The first algorithm is a collapse function, it goes through all cells and if the function has only one possibility, it sets it as the cell, and a cell is set, it calls the check function on all cells in the line/row/box.

The backtracking algorithm checks using recursion every possibility for every cell; if there is a contradiction, it leaves the recursion and removes the possibility.

Since backtracking is very expensive, we tried to optimize the code as much as we could, so the data was stored into a matrix of 9 by 9 of shorts, with each short representing everything we needed for both algorithms. The shorts were as follows (using an example here) :

Read from left to right: 00 1 0101 011010101

The first 2 bits are ignored, then the next bit is 0 if the cell is solved (only 1 possibility). The next 4 bits represent the number of possibilities (here 5 possibilities). The next 9 bits are 1 if in this cell, the bit index (from right to left, started by 1) is a valid possibility.

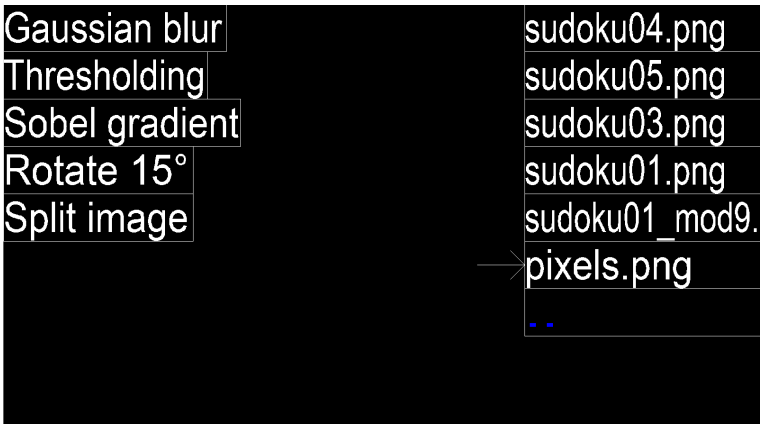
So here, the cell is not solved, and it can be of 5 possibilities : 1, 3, 5, 7 and 8.
In the case were the third most significant bit is 0 the short will only be just a digit of the sudoku.

This implementation permits us to keep track of which possibility has been already been searched and helps the performance for the resolution of the grid.

10 Graphical User Interface

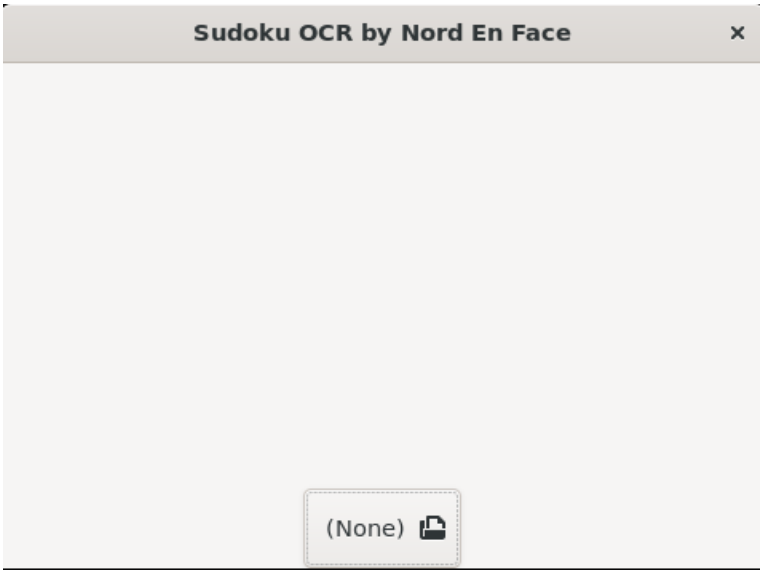
For the 1st defense, we had a very simple GUI, it was not practical and was ugly. We created it in SDL since that's what we had seen in the TP at the start of the year, and once the GUI was almost done, Ilan enlightened us with GTK, a library used for creating GUI's. Since the job was almost done, we continued working a bit on the GUI so it was usable, even though it was ugly.

This GUI had the main functions we tested on the left , and on the right it has a file explorer. This works by having on the left a list of buttons (a struct), which has an x and y positions, and a function pointer, which gets called when there is a click on top of it. On the right, the buttons are created procedurally.

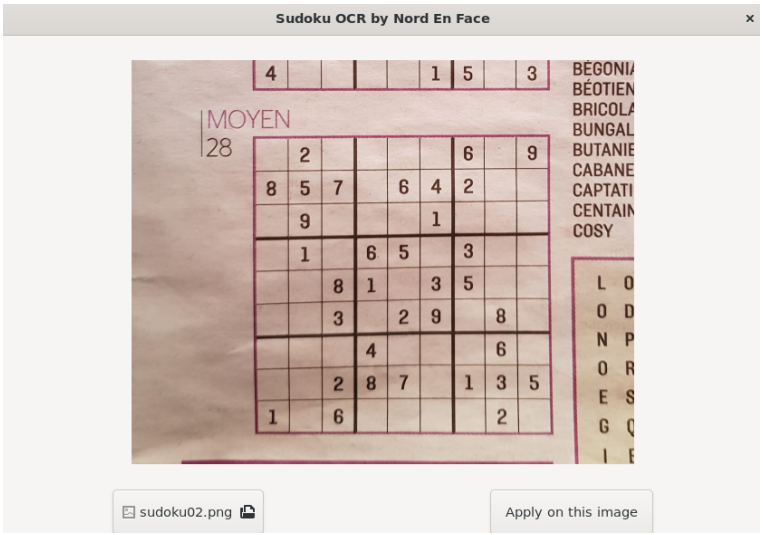


Almost immediately after the first defense, we had the GTK TP, which helped immensely to get a better understanding of GTK and glade. So we started working on the new and improved GUI. There were a few issues at first, we had to design a good looking, easy to use GUI.

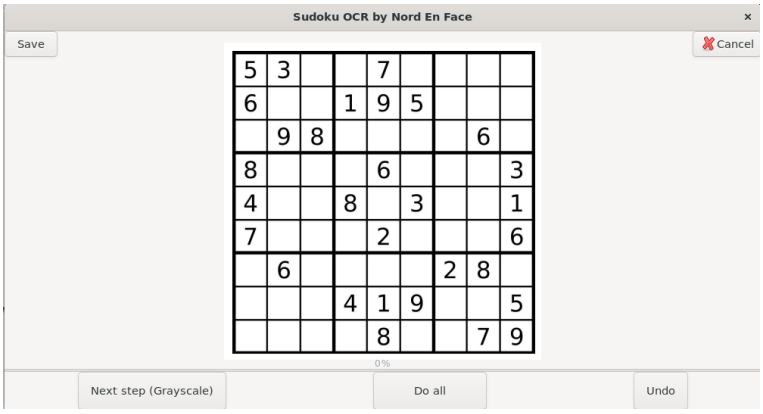
Then, we had to make the code work. We have a system similar to that of the GTK TP : there is a struct which is used to store all the gui elements and this struct is being passed around many functions.
At the end, this is the state of the GUI.



This first window is used to select a file, if a file is chosen, a second button will appear to go to the next window and the image is shown in large.



In this second window, the image is shown in the middle, and a few buttons can be seen : next step, do all, undo, cancel and save. Most of them are self explanatory but I think it's interesting to see how they work : There is a variable which represents the index of the step we are at, if we click on the next step, it saves the current image in a temp file and calculates the next image. The goal of saving many temp files are for the undo button. We don't want to recalculate everything every time we undo.



11 General Progress

Tasks	Percentage
Preprocessing	
Gaussian Blur	100%
Grayscaling	100%
Adaptive thresholding	100%
Sobel Edge Detection	100%
Grid Detection	
Hough Transformation	100%
Line Filtering	100%
Square Detection/Selection	100%
Segmentation	
Image Resizing	100%
Image Rotation	100%
Image Splitting	100%
Image Correction	100%
Neural Network	
Gaussian distribution	100%
Neural Network XOR	100%
Neural Network Digits	100%
GUI	
GUI	100%

Table 2: First Table of role divisions

We are very happy to say that we managed to finish every single step required for this project.

12 Conclusion

To conclude this report, we are very proud of our work, and we plan to continue working on the project even when in the S4 abroad (potentially to make a better neural network, implement hexadoku, etc). We are happy to have been able to work like a group, we shared tasks, gave parts of tasks to group members who were more able on certain parts, etc.