Computer Project File

SPE 2022

December 5, 2018

# Optical Character Recognition

## Final Report

NoneOfAllOfTheAbove

Arielle LEVI
Ferdinand MOM
Louis CHEVALIER
Théo LEPAGE

# Contents

# 1 Introduction

The second year student's project at EPITA consists in making an Optical Character Recognition software. Its objective is to extract text from images. This aim of this document is to present our implementation and its progress as well as the way it works. Moreover, we will address our work flow and our vision of the project.A lot of features and amelioration have been added to our project since the first defense, which permits us to present you today a fully functional OCR software. This OCR software recognizes most of the words extracted letter by letter (as long as the text font is not too weird), and offers a nice and intuitive graphical interface that can be used among others to load new pictures and save the text extracted in a file.

The project is the result of a good teamwork, where everyone was welcome to participate and share their ideas. Many plannings and group meetings helped to structure everything, link all the parts together and survive the C language together! We hope you will enjoy our work. All the members of the team wish you a good reading!

- The None Of all of the above Team

## 1.1 Object of the study and specifications

The book of specifications states that our application must read an image as input of usual standard formats and produce the extracted text. For the final version a graphical user interface is required. The general features will be image loading, image visualization, extracted text visualization, text edition and text saving. Finally, we are required to use a machine learning algorithm, to train a neural network, load and save its parameters. To improve the recognition capabilities, we are suggested to improve the preprocessing by handling rotations, noise canceling and contrast enhancement. We have to respect several guidelines for the compilation step, our code format and the version control. The most important one is to use the C99 standard.

The main steps of our algorithm are as follows:

1. Image loading, binarization (gray-scale to black/white) and preprocessing
2. Text block and lines detection and character splitting
3. Extracted characters recognition
4. Text reconstruction and GUI

## 1.2 State of the art

With the development of computer vision and artificial intelligence, Optical character recognition software became widely used. As well as every system capable of allowing a computer to understand our environment. Optical character recognition software are implemented in smart phones to scan images to documents for instance. They are also used to automate tasks in the industry or to help sightless persons.

Concerning existing solutions, only Tesseract (open source) and Adobe Acrobat Professional stand out. It is due to the fact that they are not targeting regular users but scientists or companies.

## 1.3 NoneOfAllOfTheAbove

NoneOfAllOfTheAbove is the result of the combination of two groups of last year's project: AllOfTheAbove and NoneOfTheAbove. It is composed of four members:

- Arielle LEVI (arielle.levi@epita.fr)
- Ferdinand MOM (ferdinand.mom@epita.fr)
- Louis CHEVALIER (louis.chevalier@epita.fr)
- Theo LEPAGE (theo.lepage@epita.fr) [Project leader]

The main common point between us is that we are hard-working and demanding towards our work. We have all a discipline of choice ranging from mathematics, algorithmic and programming. Our objective is to produce the most advanced OCR we are capable of. Furthermore we are all interested in carrying on the project later on.

# 2 Organization

## 2.1 Workflow and softwares

Our workflow consists in working on separate branches of the git repository. Each branch corresponds to a feature: character-recognition for the neural network and character-detection for the segmentation for instance. Before each defense we merge all branches to master. Git makes the merging process very convenient.

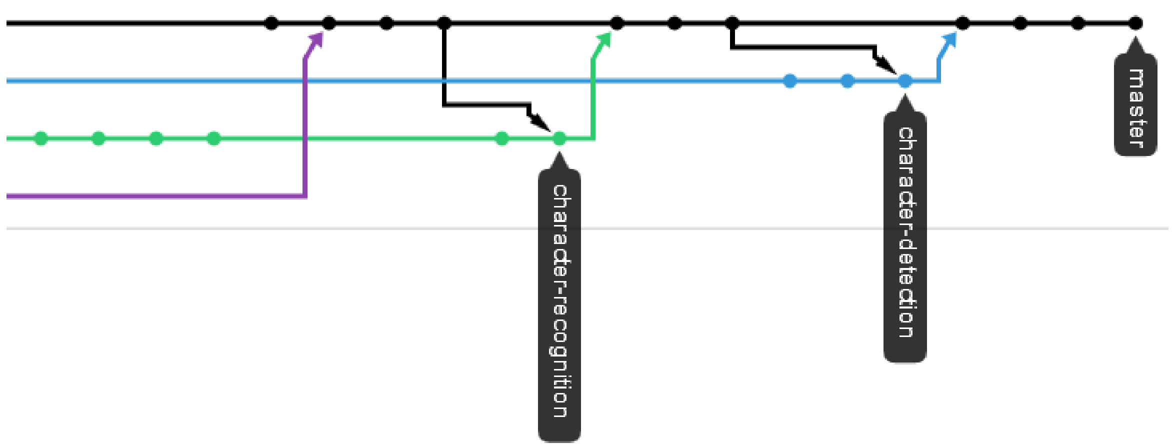
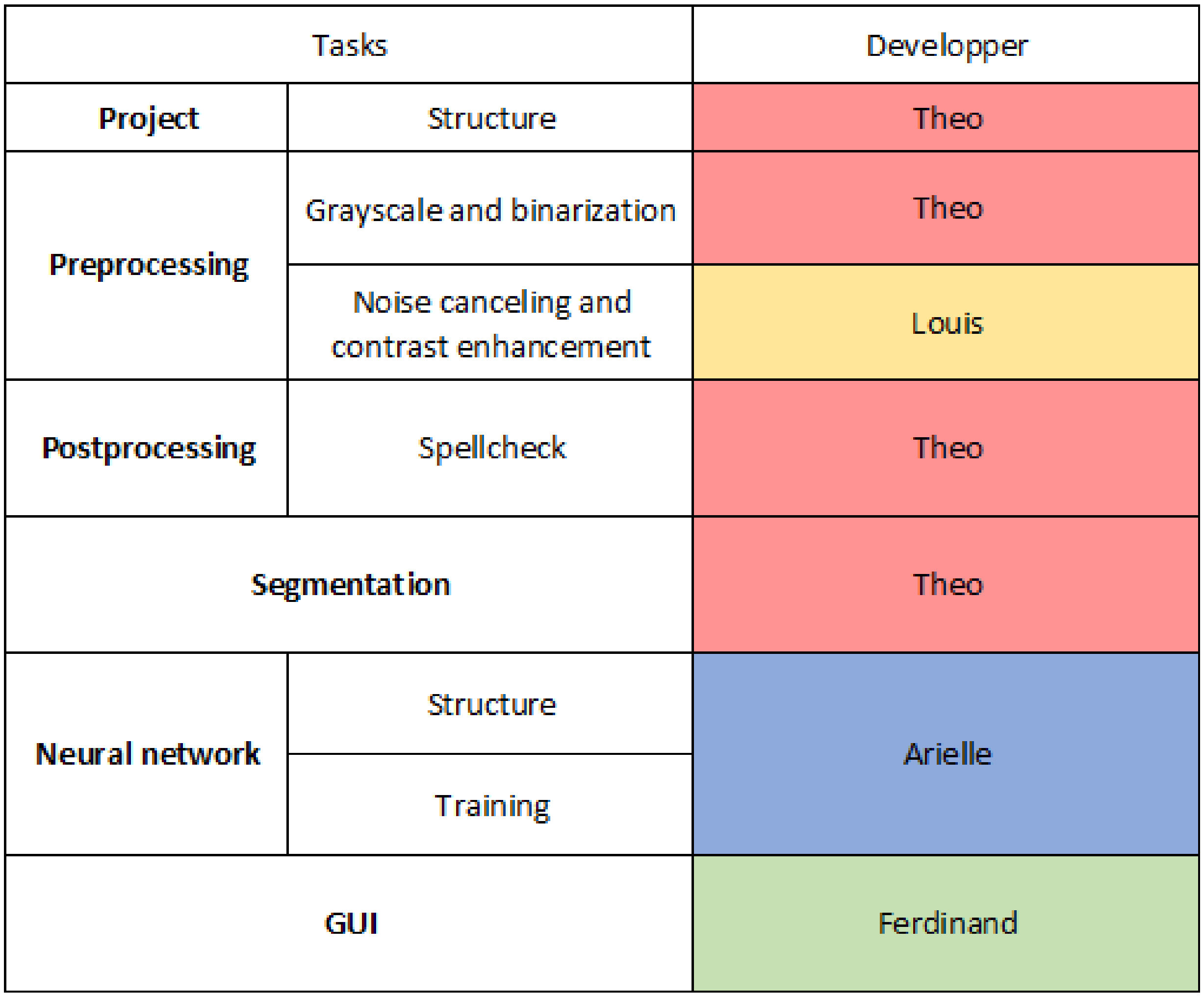


Figure 1: The git tree of our project representing the different branches.

Concerning the tools used for the development process, we rely only on Vim as a code editor (or Visual Studio Code). We use GCC to compile our code and a Makefile to make the compiling process simpler and faster by adding expected C flags and linkers to libraries. For the realization of documents and reports we use Overleaf that allows us to type Latex more efficiently. Finally all the team members work on a Linux distribution, Archlinux or a Debian based one.

## 2.2 Distribution of tasks

The distribution of tasks is very important to make the project progress faster. Each member are assigned with a set of tasks that are considered to suit his/her personal skills.

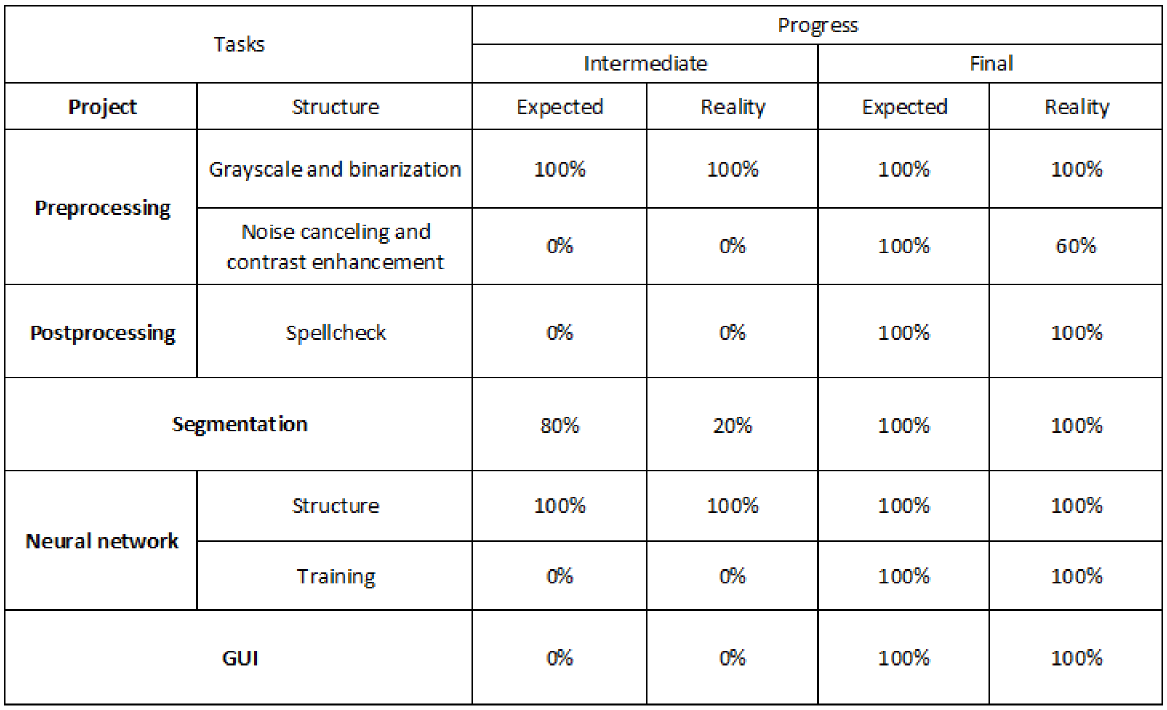


| Tasks | | Developper |
|---|---|---|
| **Project** | Structure | Theo |
| **Preprocessing** | Grayscale and binarization | Theo |
| | Noise canceling and contrast enhancement | Louis |
| **Postprocessing** | Spellcheck | Theo |
| **Segmentation** | | Theo |
| **Neural network** | Structure | Arielle |
| | Training | |
| **GUI** | | Ferdinand |

Figure 2: Distribution of tasks within developpers.

## 2.3 Progression

We succeeded to implement every features required for the first defense and we will strive to stay in advance to eventually add other features.



| Tasks | | Progress | | | |
|---|---|---|---|---|---|
| | | Intermediate | | Final | |
| **Project** | Structure | Expected | Reality | Expected | Reality |
| **Preprocessing** | Grayscale and binarization | 100% | 100% | 100% | 100% |
| | Noise canceling and contrast enhancement | 0% | 0% | 100% | 60% |
| **Postprocessing** | Spellcheck | 0% | 0% | 100% | 100% |
| **Segmentation** | | 80% | 20% | 100% | 100% |
| **Neural network** | Structure | 100% | 100% | 100% | 100% |
| | Training | 0% | 0% | 100% | 100% |
| **GUI** | | 0% | 0% | 100% | 100% |

Figure 3: Progress expected and achieved for each task.

### 2.3.1 Tasks done

- Image loading, color removal, noise canceling and contrast enhancement
- Blocks, lines, words and character detection and splitting (handle multi column)
- A working neural network able to recognize characters (by saving weights)
- User interface and result saving

### 2.3.2 Tasks planned for the future

- Improve preprocessing (deskew)
- Improve segmentation (handle images)
- Train neural network with more fonts
- Refactoring

# 3 Project structure

## 3.1 Structure of the project

We are very concerned of having a well structured project. From the beginning we divided the source files in different folders. The idea is to keep the main source file short and intelligible. An overview of the structure is shown below.

```
|
|__bin
|__docs
|  |__samples
|__resources
|  |__nn.data
|__src
   |__GUI
   |__NeuralNetwork
   |__Preprocessing
   |__Postprocessing
   |__Segmentation
   |__Matrix
   |__Stack
```

## 3.2 Compilation

Two executables are generated, in the bin folder, by the Makefile: *OCR* and *Tools*. Tools is based on the source file tools.c and use several parts of the application (preprocessing, neural network). It is used by the team to train the neural network or prepare the dataset. The Makefile is straightforward, it creates the bin folder, compile OCR and Tools with gcc and allows to clean these files. It is configured to show every warnings and use the C99 standard. In the future, we could consider using optimizations settings.

## 3.3 Dependencies

Our software currently depends on four libraries listed below.

- SDL2 (for debug GUI)
- SDL2 Image (for Preprocessing)
- GTK+ 3.0 (for main GUI)
- Hunspell (for Spellcheck)

# 4 Preprocessing

## 4.1 Image loading

Concerning the image loading, at first, we wanted to handle images without any external library other than libpng and libjpeg. It turns out that it was thought to get pixels' colors without SDL2's image library. Therefore we decided to rely on it mainly because it allows us to support many standard formats without having to handle this tedious task. The loading process is about creating a SDL surface from a loaded image of any format (jpeg, png, gif, tiff). Then, using SDL's methods we are able to get the red, green and blue components of a pixel at a specified location.

## 4.2 Colors to Grayscale

The first function of the preprocessing step, *LoadImageAsGrayscale*, is about loading an image and returning a matrix of its grayscale values. There exists several method to compute the grayscale value of a RGB pixel. The luminosity method follows this formula: *(0.3 * r) + (0.59 * g) + (0.11 * b)*. The average method consists in computing the average value of the three components: *(R + G + B) / 3*. We decided to use the luminosity method rather than the average one. Indeed, both methods give results that do not affect the next parts.

To be more specific, the function *LoadImageAsGrayscale* returns a dynamically allocated array of arrays of type unsigned char and of size image's height times image's width. We decided to represent images as matrix, in other words, as array of arrays and not with a single dimensional array even though it was recommended. The allocation of a two dimensional array is not tough and getting pixels with the notation: *matrix[y][x]* seems more convenient. Our choice will be explained more in details later in the neural network part.

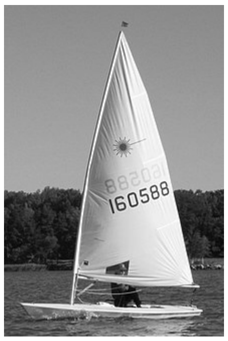


Figure 4: Example of a sailboat image converted to grayscale.

## 4.3 Noise cancelling

As an image of bad quality can lead to mistakes during the characters recognition, we decided to implement an image convolution algorithm as it allows us to apply various different filters on our picture, before the segmentation's process. Convolution is the process of adding each element of the image to its local neighbors, weighted by a kernel. A kernel is matrix, usually of size 3x3, which contains weights to apply to each neighboring pixel's value, in order to determine the value of the current pixel in the modified picture. We decided to implemented a general convolution function, to allow us to apply a wide range of filters to our image. Eventually, even if some filters had interesting effects (emboss, sharpen, sorel's filter), we kept only the 2 best filters for our goal, that are the gaussian blur and the box, or mean, blur filter.

Image noise is the random variation of brightness or color information in images, and is usually produced by the sensor and circuitry of a scanner or digital camera. To reduce it, we used the box blur filter and the gaussian blur filter, as they have slightly different effects and can be combined to get the best out of the original picture.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 5: Gaussian blur kernel

As nothing is better than an example to explain something, let us take a picture with an unclear background and apply the Gaussian kernel to it.

As you can see, the Gaussian blur filter can have a very impressive effect, as it can clear the picture's background without harming the text integrity. Indeed, even if the shape of some letters have been slightly modified by the Gaussian blur, the change is not significant and did not have any impact on the character recognition efficiency.

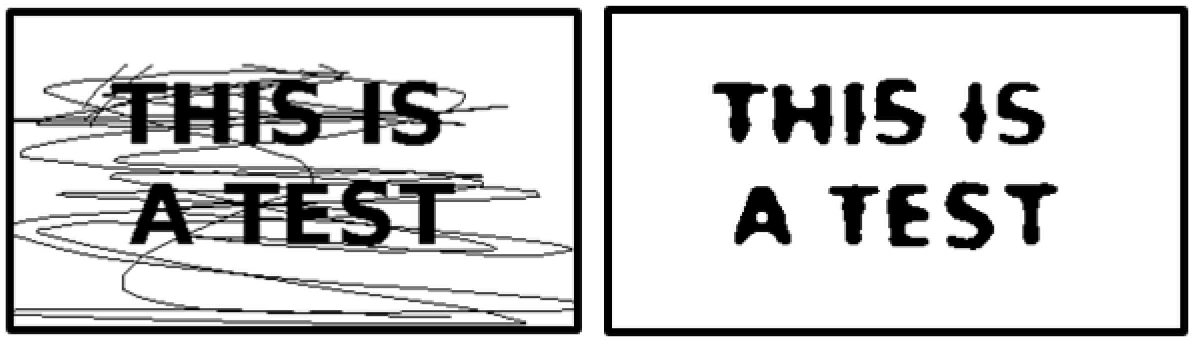


Figure 6: From left to right, before and after applying our Gaussian blur kernel implementation

## 4.4 Contrast enhancement

Another frequent issue on text pictures is contrast, as low contrast can lead to bad segmentation. A simple way to detect badly contrasted image is by studying the image histogram. An image histogram is a type of histogram that acts as a graphical representation of the color distribution in a digital image. In a gray scale image, it plots the number of pixels for each color value, from 0 to 255.

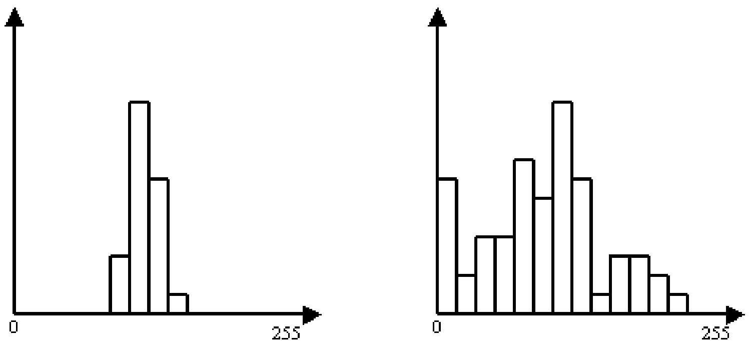


Figure 7: Bad color repartition (left histogram) means bad contrast

A narrow histogram means bad contrast, that is why we implemented a well-known mathematical treatment called dynamics expansion. The goal is to extend the range of the histogram by applying a simple formula on each pixel.

$$Pixelvalue = round\left(M \cdot \left(\frac{n - minimum}{maximum - minimum}\right)\right)$$

In this formula, *M* represents the maximum output value (255 in our case) and *round* is a function which returns the closest integer. To determine *minimum*, we need to go through the histogram from left to right and stop when we reach 1% of total pixels count. *Minimum* is then the corresponding color value. To get *maximum*, we use the same reasoning as for *minimum* but from right to left.

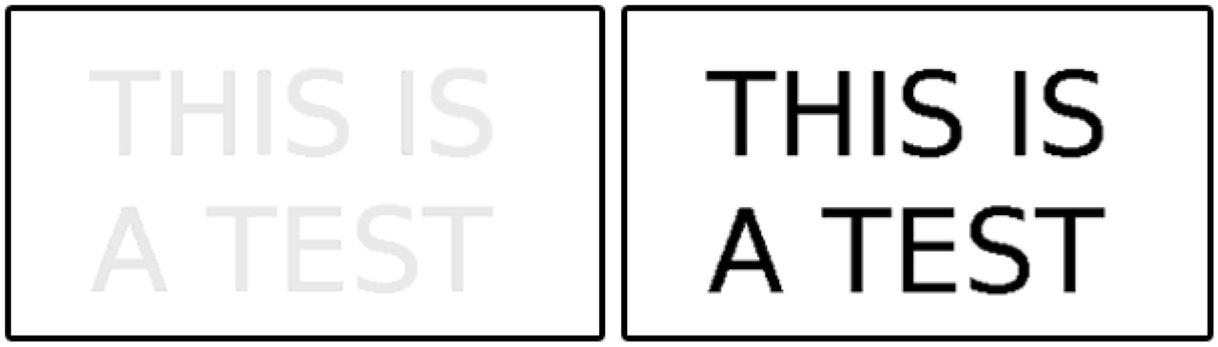


Figure 8: From left to right, before and after applying our dynamics expansion implementation

The main problem caused by the dynamics expansion technique is that when used in parallel with Otsu's binarization method it can invert black and white in the binarized image. That is why we created a simple function which detects when black and white are inverted and fix it.

## 4.5 Binarization

The second function, *BinarizeImage*, takes the grayscale matrix and returns a binarized matrix with the same structure than the previous one. To do so, we computed a threshold using Otsu's algorithm[1] which is often used for this type of application. The algorithm assumes that the image contains two classes of pixels, it then calculates the optimum threshold separating the two classes. This method gives results way better than with a fixed threshold or threshold set to the average intensity of all pixels.

[1]Otsu's method. `https://en.wikipedia.org/wiki/Otsu%27s_method`

Then we assigned each pixel either with the black value or the white value depending on its intensity compared to the previously computed threshold. As you can see with the example of a picture of a laser's sail, the numbers on the sail are emphasized. We can notice that the numbers on the other side are ignored except for the darker one. This is why enhancing the contrast of the image is necessary.

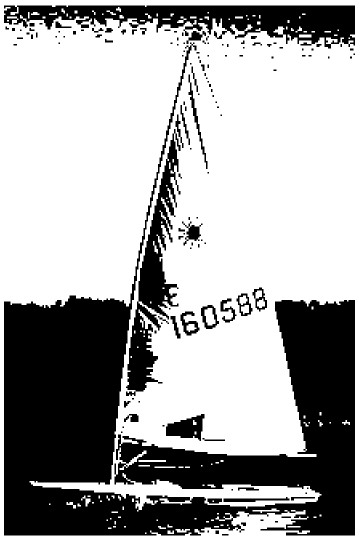


Figure 9: The same image of the boat converted to a ”binarized” matrix.

# 5 Segmentation

The segmentation step processes the output image of the preprocessing step to handle the text structure. Its purpose is to prepare a matrix representation of each character, with a specified size. Indeed, the character's matrix is the input given to the neural network.

## 5.1 Data structure

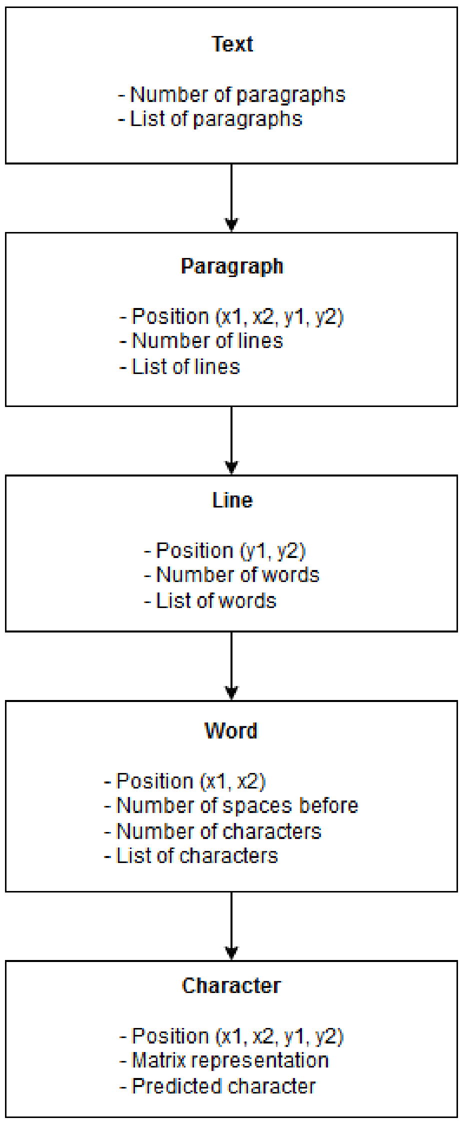


Figure 10: Recursive organization used to register elements of the segmentation.

The above diagram represents the way we store information about the segmentation. Each rectangle is an element with a name and a list of attributes. You can see this representation as a tree, the root node being Text. Text can have multiple children considered as ”paragraphs”, each paragraph can have multiple children considered as ”lines”... Each type of object has attributes defining it. You can notice that lines do not have a *x1* and *x2*, because we consider that they have the same width as their parent (a paragraph). This organization allows us also to simplify the process when creating the output text while handling spaces and line breaks. We tried as best as we can to reproduce an object-oriented way of programing while coding with structures in C.

## 5.2 Detecting paragraphs

### 5.2.1 The principle

The first element handled by the segmentation step are blocks of texts, that we call paragraphs. To detect paragraphs we use the Run Length Smoothing Algorithm[2] (RLSA) method. It consists in linking together neighboring black areas that are separated by less than a specified number of pixels. With an appropriate choice of this number, the linked areas will be regions of a common data type.

For example, with a step of 4 pixels, the sequence x is mapped into y as follows:
x : 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0
y : 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1

### 5.2.2 Our implementation to find paragraphs

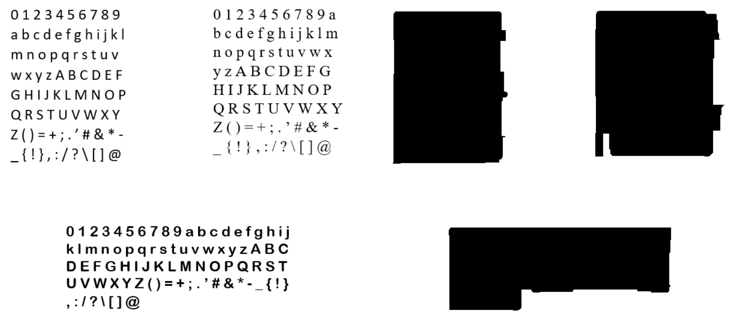


Figure 11: The original picture and its RLSA's output on the right.

[2]Rais, Martin & Goussies, Norberto & Mejail, Marta. (2011). Using Adaptive Run Length Smoothing Algorithm for Accurate Text Localization in Images. https://www.researchgate.net/publication/220843271_Using_Adaptive_Run_Length_Smoothing_Algorithm_for_Accurate_Text_Localization_in_Images

In our case, we tried several values both for horizontal and vertical pass and *100 pixels* and *50 pixels* works for many images. We may consider later multiplying these values by a ratio to make them depend on the image size.

### 5.2.3 Computing paragraphs' bounds

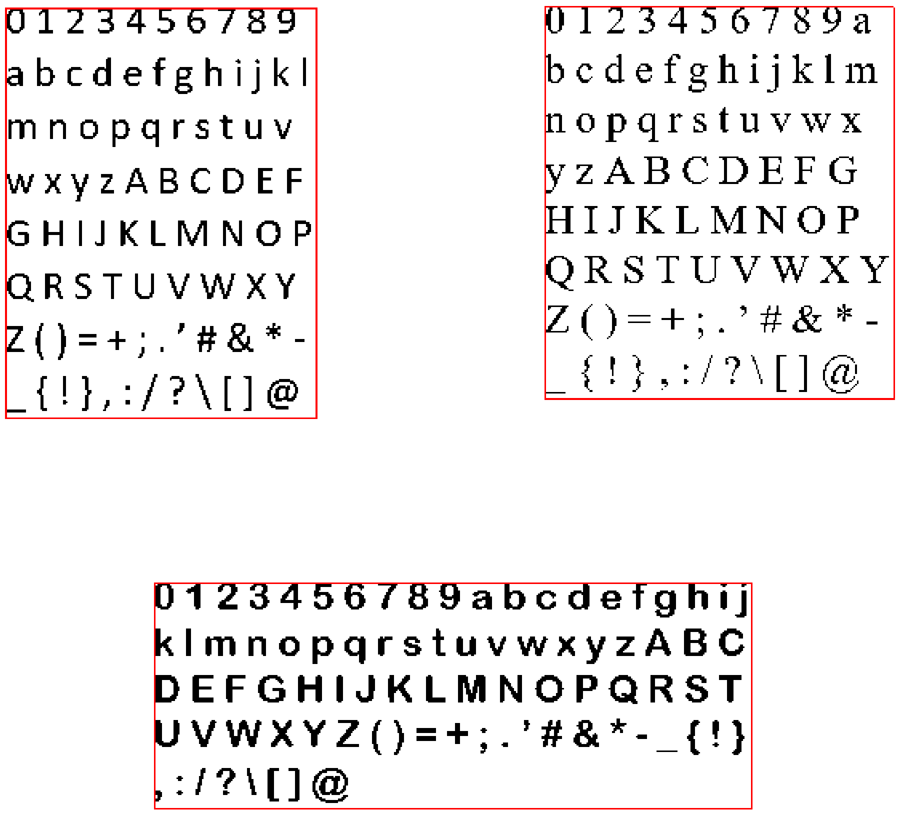


Figure 12: The result of our algorithm which find bounds of each paragraph.

From the output of the RLSA we need to find the position and the size of each paragraph. As they are represented as black pixels blocks, we need to go through each of them and find for each one: *xMin*, *xMax*, *yMin* and *yMax*. Using a recursive algorithm is not a convenient solution because we will exceed the maximum recursive calls number. For instance, on my machine it is set to 8192, because the algorithm would be called four times for each pixel (for each direction in the 2D matrix), a block of 2048 black pixels would be the maximum size handled. Therefore we decided to implement a new data structure to do this step in a iterative way: Stacks.

For each black pixel found in the matrix given by the RLSA, we run this iterative algorithm that compute the bounds of the current block and replace its pixels by white pixels. At the end, we free this matrix that is now entirely white and we have a list of Paragraphs variables *(refer to Figure 8)*. It is noteworthy that we need to run this algorithm two times. Indeed, before registering paragraphs we need to know the number of blocks in order to allocate a list with the correct size.

## 5.3 Detecting lines

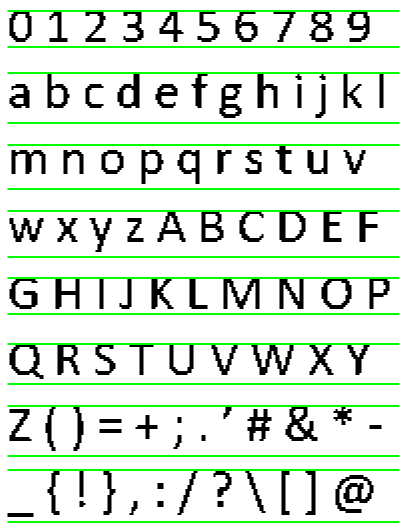
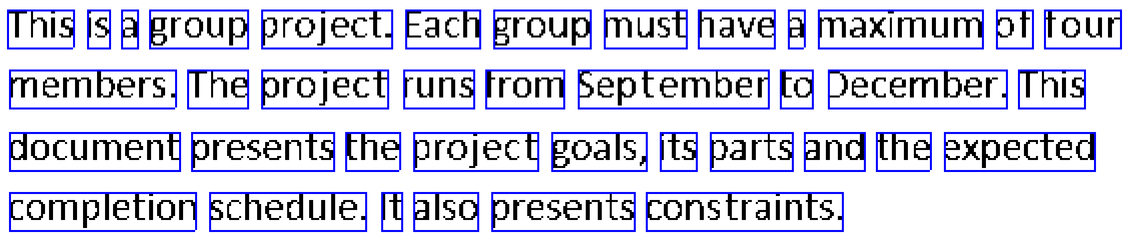


Figure 13: The result of our algorithm which find bounds of each line.

We assume that the input image does not contain noise and is correctly rotated until we implement a de-skew feature. Therefore, the line detection algorithm is very simple. Nevertheless, like for paragraphs, we need to run it two times to know the number of lines before registering them. The algorithm is called on each paragraph, in the area delimited by the paragraph's positions computed before. For each vertical line of the matrix, if there exists a black pixel we determine if it represents the top or the bottom position of the line. We implemented a "tolerance gap" so that characters, such as underscores, placed under the baseline are taken into account.

## 5.4 Detecting words



This is a group project. Each group must have a maximum of four members. The project runs from September to December. This document presents the project goals, its parts and the expected completion schedule. It also presents constraints.

Figure 14: The result of our algorithm which find bounds of each word.

The purpose of words is to handle spaces between characters in the text. It also allows us to use a spellcheck, which would be impossible without knowing in which word a character belongs. We suppose that there is a gap of at least 1 pixel between each character. To separate gaps between characters and gaps between words we first compute the average width of all characters on a line. For each white space, if it is greater than the average width of characters divided by two, we consider it as a space between two words. As for the lines, the algorithm is rather simple. Nevertheless, we need to run it two times to know the number of words before registering them.

## 5.5 Detecting characters

### 5.5.1 The algorithm

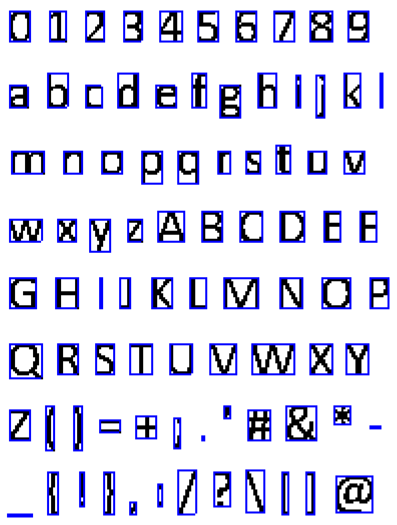


Figure 15: The result of our algorithm which find bounds of each character.

The last step of the segmentation is to detect characters inside of every words and extract its matrix representation. Furthermore, it has to transform each character's matrix to a square matrix of a specific size to match neural network's input. Because we assume that characters are not adjacent (they have at least 1 pixel of white space between them), we extract black shape isolated by a white space vertically. We could have used a Connected-component labeling[3] but our method allows us to handle letters that have multiple disconnected shapes. For instance, letters i and j and even letters with accents.

### 5.5.2 Matrix normalization

To get the matrix representation of the character we follow these steps:

1. We get the sub matrix directly around the characters using its position (x1, x2, y1, y2). The resulted matrix have *(y2 - y1)* lines and *(x2 - x1)* columns.
2. We put this matrix in a square matrix of size *x*x*, where x is a multiple of 24. In our case we use 24 because the neural network was trained using a 24x24 data set.
3. Finally, if *x* is greater than 24 we scale down it using a simple algorithm we made. It basically assigns for each pixel the average value of the sum of a set of pixels in the larger matrix.

[3] Connected-component labeling. `https://en.wikipedia.org/wiki/Connected-component_labeling`

# 6 Neural Network

For this part, we were ask to build a Neural Network that were able to recognize an exclusive or (XOR) first, and then use what we learn to obtain a program that recognizes characters. After our many researches, we found out that our Neural Network had to be represented in a certain structure, in order to be able to give us a result corresponding to the input we feed it with. If the results were not satisfying, we simply had to teach him what the good answer was, in such a way that the next prediction would get closer to the expected result. The concept of what a neural network is and how it works is very easy to grasp because it is possible intuitively to make parallels with some child or animal that needs to learn something, but the maths involved to create one are very hard to understand and to debug, that is what makes this part rather complicated. The fact that everything had to be in the C language was also disturbing since, as people used to object oriented languages, immediate reflexes to create an object ”neural network” for example had to be modified, in order to find a new way to implement this part, a new way to work.

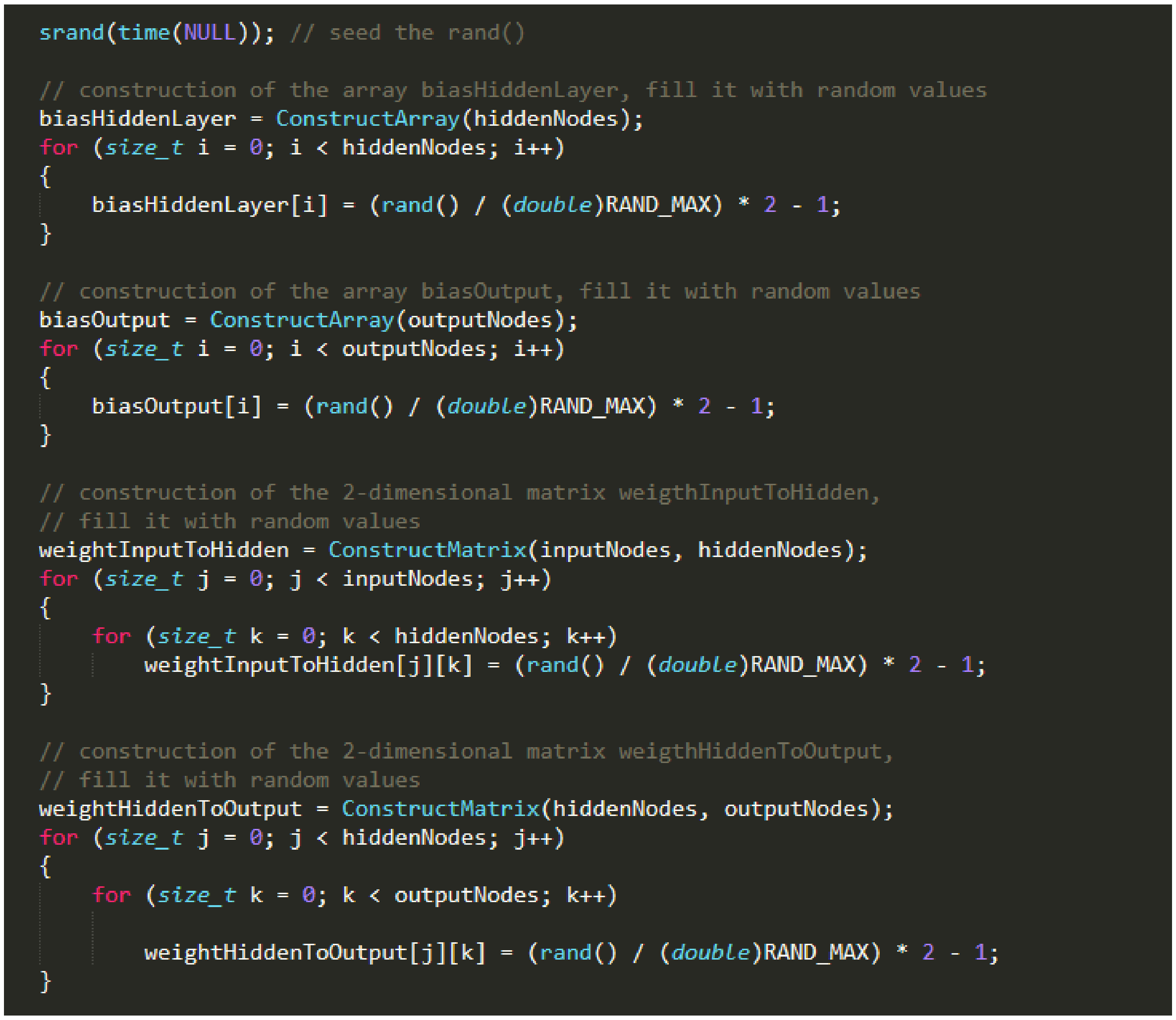
Since character recognition is the one of the main keys of our OCR, we decided to make it the clearest possible. In order to do that, we ask ourselves one question: *what does our Neural Network needs to do?*. The response was easy, it had to be able to predict the outputs we wanted according to some inputs and learn from us as stipulated above, but something was missing, before all of that we had to create from scratch our Neural Network by giving it some properties and values to rely on, to give it an identity (which explains why it was so difficult not to think about object oriented solutions). Eventually we decided to create three functions : **Start()**, **Predict()** and **Train()**.

## 6.1 Structure

The Start() function is responsible for the creation of the neural network’s structure. We had some trouble deciding how we should create it, first because we were really used to languages which are object oriented (python and C#), so the idea to create a class came pretty instantly and was hard to get rid off. This is probably why so much groups worked with structures though that provides a sort of organization. We needed something that permits us to access the values of the different weights and bias from anywhere in our functions and to possibly change them. So despite of the many warnings of our teachers last year and forums on internet we went with global values. But worry not! We made global values only when it was not possible in another way. and made them as secured as possible.

We had three $size\_t$ global values, **numberInputNodes**, **numberHiddenNodes** and **numberOutputNodes**, which obviously represent the number of neurons in each layer (as suggested in the book of specifications we decided to work with only one layer. We also have two 1-dimensional arrays, one representing the bias for each neuron in the hidden layer and the other all the bias for each neuron in the output layer, called **biasHiddenLayer** and **biasOutput**. Finally we have two 2-dimensional arrays **weightInputToHidden** and **weightHiddenToOutput** that stores the weight from the input layer to the hidden layer (respectively from the hidden layer to the output layer) in a matrix of size *numberInputNodes x numberHiddenNodes* (respectively *numberHiddenNodes x numberOutputNodes*). All the arrays and multidimensional arrays that we have, are constructed by two auxiliary functions: **ConstructMatrix()** and **ConstructArray()** (as in the C language it is a little hard to mix one and two dimensional arrays), and are filled with random numbers of the type $double$,between -1 and 1 in the function Start(). We wanted to randomize directly the matrices and arrays in the auxiliary functions but the fact that the function **rand()** can only give pseudo random numbers, made it impracticable. In fact the rand() function simply called will always give the same series of numbers since it takes the values in a large list of integers from the index 0. This is why we need to *seed the rand*, that is, to ask the function rand() to start enumerating the list from a certain index. This index can be given by the function **Time()** that returns the number of seconds that occurred since the first of January of the year 1970. The problem was that by calling Start() we were calling many times the function to construct a matrix or an array during the same second which resulted in seeding each time the rand() function with the same index. By consequences, the seed was always the same and so was the numbers in each matrices and arrays.

We decided to choose long names for our variables even though they are kind of ugly in order to make things clearer for people reading our code and for us.



```
srand(time(NULL)); // seed the rand()

// construction of the array biasHiddenLayer, fill it with random values
biasHiddenLayer = ConstructArray(hiddenNodes);
for (size_t i = 0; i < hiddenNodes; i++)
{
    biasHiddenLayer[i] = (rand() / (double)RAND_MAX) * 2 - 1;
}

// construction of the array biasOutput, fill it with random values
biasOutput = ConstructArray(outputNodes);
for (size_t i = 0; i < outputNodes; i++)
{
    biasOutput[i] = (rand() / (double)RAND_MAX) * 2 - 1;
}

// construction of the 2-dimensional matrix weigthInputToHidden,
// fill it with random values
weightInputToHidden = ConstructMatrix(inputNodes, hiddenNodes);
for (size_t j = 0; j < inputNodes; j++)
{
    for (size_t k = 0; k < hiddenNodes; k++)
        weightInputToHidden[j][k] = (rand() / (double)RAND_MAX) * 2 - 1;
}

// construction of the 2-dimensional matrix weigthHiddenToOutput,
// fill it with random values
weightHiddenToOutput = ConstructMatrix(hiddenNodes, outputNodes);
for (size_t j = 0; j < hiddenNodes; j++)
{
    for (size_t k = 0; k < outputNodes; k++)

        weightHiddenToOutput[j][k] = (rand() / (double)RAND_MAX) * 2 - 1;
}
```

We wanted to work with dynamical arrays (and matrices) so that we did not have to worry about their sizes until the time of the compilation. But on the contrary of Python for example, lengths of arrays have to be constant. This is why we had to use the dynamic allocation of the memory, thanks to the function **malloc()** in order to save for our values a certain amount of memory space we want, and after, the function **free()** that literally free the memory space we kept for our arrays with the previous function. For the matrices, we used a double pointers, a pointer to a list of pointers, which gave us the liberty to allocate as much memory as wanted for them. This is one of the most important feature of our XOR Neural Network, because by doing this, we made it completely scalable, which means that we could create a network with as much as neurons that we want (as far as our memory allows us to). This structure is totally malleable, it has been extremely easy to adapt the xor to something much more bigger, needed for the character recognition.

Notice that we used the type $size\_t$ instead of any other like $int$ or $long$ for variables: $numberInputNodes$, $numberHiddenNodes$ and $numberOutputNodes$ because this type was designed to represent the size of an object and thus guaranteed to hold any array index. Plus, we used several time the function **sizeof()** that returns a $size\_t$. After that, trying to cast it into an $integer$ was creating unsafe casts. The only alternative was to use $long$ but since we were only dealing with sizes we decided to keep everything into $size\_t$.

## 6.2 Prediction

**Predict()** is the function that returns an array of outputs from a given array off inputs (Feedforward). It takes for argument the $inputs$ array. Since we had to design a XOR gate for the first defense, our first neural network had 2 inputs and 2 outputs. We decided to had 2 outputs even if we could have done it only with one, because we know that the real problem, the one we need to solve in order to complete the OCR (which means to recognize the characters), could not be resolve only with one output. Between the input and output layers, we decided to structure our neural network in order to have one hidden layer (as suggested in the book of specification), composed of the number of node we choose (the number do not need to be known before the compilation as well as the number of input nodes and the number of output nodes), while calling Start() beforehand. We started, just before training our network for the first time, to implement a second layer to challenge ourselves and to make our network more effective. Unfortunately, we found out at this time some problems related to our database, and we prefered to solve it instead of adding features not required. At the end of the project, the network was working very well and not enough time was left if we wanted to train another with a second layer, so we never came back to its creation.

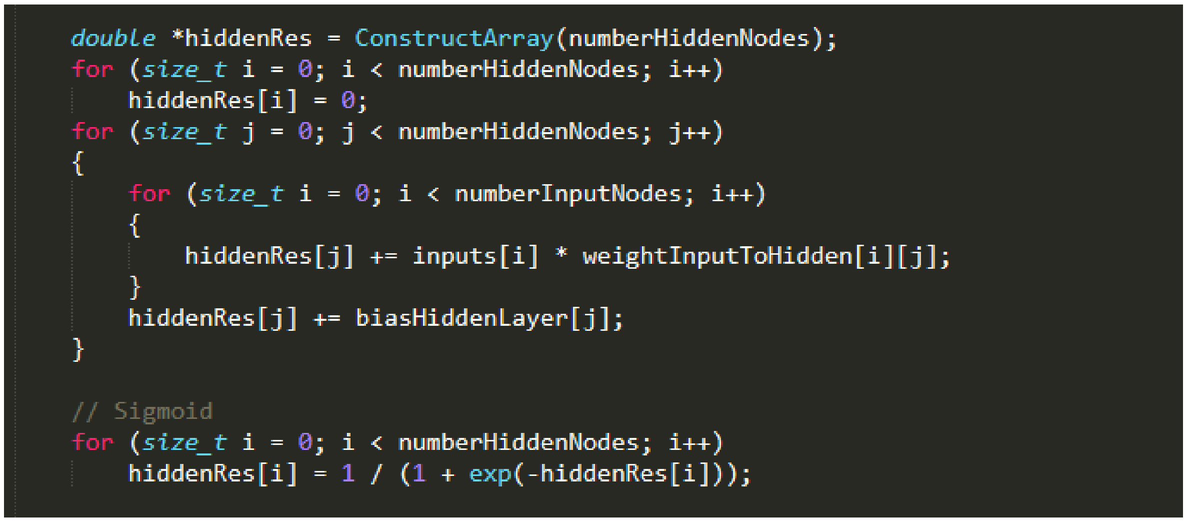
In order to get to the output layer, we have to first go to the hidden layer. To do so, we need to compute the weighted sum between the inputs and the weights of those two layers to which we add the corresponding bias to each hidden node, and then we apply to each element of the resulting array the activation function: sigmoïd. After that we realize the exact same steps to go from the Hidden layer to the output one.

We tried using the softmax function as our activation function for the last transition, and we managed to do it in **Predict()**, it worked pretty well that is why we let the commented code in the NeuralNetwork.c associated with our project. However, we had, for the next function (**Train()**), to use the derivative of softmax, and despite of all the researches that we did, and even of the contribution of our maths teacher (thank you Mr Goron!), we were not able to find it or its proper formula. Now, let's dive right in the mathematics! Let's denotes $hiddenRes$, the 1-dimensional array that will contain all the results of the input to hidden layer process for each hidden neuron. Then we have the following formula:

$$hiddenRes[i] = \sigma(\sum_{j=0}^{n}(inputs[j] * weightInputToHidden[j][i]) + biasHiddenLayer[i])$$

1. with our activation function: the sigmoïd function $\sigma(x) = \frac{1}{1+e^{-x}}$.
2. with n = numberInputNodes
3. with **weightInputToHidden**, the matrix of weight between the two first layers.
4. with **inputs**, the array of inputs values.
5. with **biasHiddenLayer**, the matrix of bias between the first two layers.

Here is how we implemented this formula:



```
double *hiddenRes = ConstructArray(numberHiddenNodes);
for (size_t i = 0; i < numberHiddenNodes; i++)
    hiddenRes[i] = 0;
for (size_t j = 0; j < numberHiddenNodes; j++)
{
    for (size_t i = 0; i < numberInputNodes; i++)
    {
        hiddenRes[j] += inputs[i] * weightInputToHidden[i][j];
    }
    hiddenRes[j] += biasHiddenLayer[j];
}

// Sigmoid
for (size_t i = 0; i < numberHiddenNodes; i++)
    hiddenRes[i] = 1 / (1 + exp(-hiddenRes[i]));
```

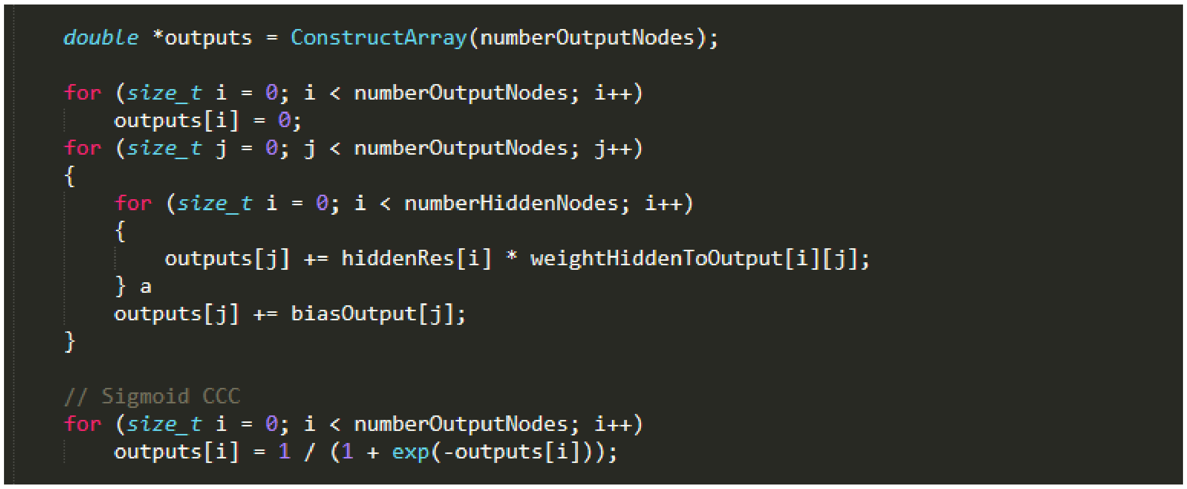
We first created the 1-dimensional array $hiddenRes$. Since in C, there is no guarantee that all the values are 0s when we initialize our array, we have to fill it by ourselves in prevention. We could have use the $Calloc$ function but if we did, we had to put it in the $ConstructArray()$ function which was called every time we needed a new array, and most of our arrays do not need to initialize everything to 0, so we though that it would e more efficient like this. Then we decided to first compute the weighted sum and add after the second for loop the bias. After this, we apply the sigmoïd $\sigma$ function on it. We tried to create a map function that takes in parameter a function and an array/matrix, but we could not find a way to take a function as a parameter of an other one in the C language.

This is the same process to go from the hidden layer to the output layer, except that this time we are not using the same values as before. Let's denote $outputs$, the 1-dimensional array that will contain all the results of this transition for each output neuron that we have. Thus we have :

$$outputs[i] = \sigma(\sum_{j=0}^{n}(hiddenRes[j] * weightHiddenToOutput[j][i]) + biasOutput[i])$$

1. with our activation function: the sigmoïd function $\sigma(x) = \frac{1}{1+e^{-x}}$.
2. with n = numberHiddenNodes
3. with **weightHiddenToOuput**, the matrix of weight between the second and the last layers.
4. with **hiddenRes**, the array of the values we got after the first transition.
5. with **biasOutput**, the matrix of bias between the second and the last layers.

Here is how we implemented this formula:



```
double *outputs = ConstructArray(numberOutputNodes);

for (size_t i = 0; i < numberOutputNodes; i++)
    outputs[i] = 0;
for (size_t j = 0; j < numberOutputNodes; j++)
{
    for (size_t i = 0; i < numberHiddenNodes; i++)
    {
        outputs[j] += hiddenRes[i] * weightHiddenToOutput[i][j];
    } a
    outputs[j] += biasOutput[j];
}

// Sigmoid CCC
for (size_t i = 0; i < numberOutputNodes; i++)
    outputs[i] = 1 / (1 + exp(-outputs[i]));
```

This is quite the same code as before except that this time we are dealing with $hiddenRes$, $weightHiddenToOutput$ and $biasOuput$. Notice that the use of global variables are very useful since we can access and change them from any scope.

Finally (after freeing the array hiddenRes), the function Predict() will return the array **outputs** that contains sort of the probabilities to return each neuron of the output layer. We just need to compare each node and find out which one has the best result (the highest probability) and return it.

## 6.3 Training

For the training part, we used the principle of $Backpropagation$. Since the function **Train()** is of type *Void*, there is nothing to return. It also takes 2 arguments, the 1-dimensional array $inputs$ and the 1-dimensional array $targets$ (array of expected result).

When we implemented our final neural network, we decided first to work with square images of 16*16 pixels, so our network had 256 input nodes, but after a while we though that we were lacking a little bit of precision so we settled together to work with square images of 24*24 pixels, so we have now 576 inputs nodes. We knew it would take a very long time to train but we decided to go for it anyway because we wanted to create the best possible OCR possible that we can. In fact it literally doubled the time dedicated to the training part, but since our neural network was **completely malleable and scalable**, adapting it took only the time to change the parameters in our main function in main.c, so ten seconds approximately. This is also why we could permit ourselves to change the size of the images we were working with so easily. This explains why we had a lot of time to train.

As we knew that we were working with a project in English, we decided to work only with letter without any accent and to work with some special characters. At the end we could recognize 85 characters which are :

```
Set_Of_Recognized_Char = {’0’, ’1’, ’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’,

’9’, ’A’, ’B’, ’C’, ’D’, ’E’, ’F’ , ’G’, ’H’, ’I’, ’J’, ’K’, ’L’, ’M’,

’N’, ’O’, ’P’, ’Q’, ’R’, ’S’, ’T’, ’U’, ’V’, ’W’, ’X’, ’Y’, ’Z’, ’a’,

’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’j’, ’k’, ’l’, ’m’, ’n’, ’o’,

’p’, ’q’, ’r’, ’s’, ’t’, ’u’, ’v’, ’w’, ’x’, ’y’, ’z’, ’"’ , ’(’, ’)’,

’=’ , ’+’ , ’-’ , ’;’ , ’.’ , ’,’ , ’?’, ’!’, ’/’, ’\’,’{’, ’}’, ’[’,

’]’, ’@’, ’’’ ,  ’*’ , ’&’, ’_’ , ’:’, ’#’}
```

Thus, we have 85 output nodes.

For the hidden layer we put a random number of hidden neuron, that we knew very big, because it is better to have to much of them than not enough. We settled the number to 85*7.

The training function, the one modifying all the weights of the neural network named **Train()** work as so:

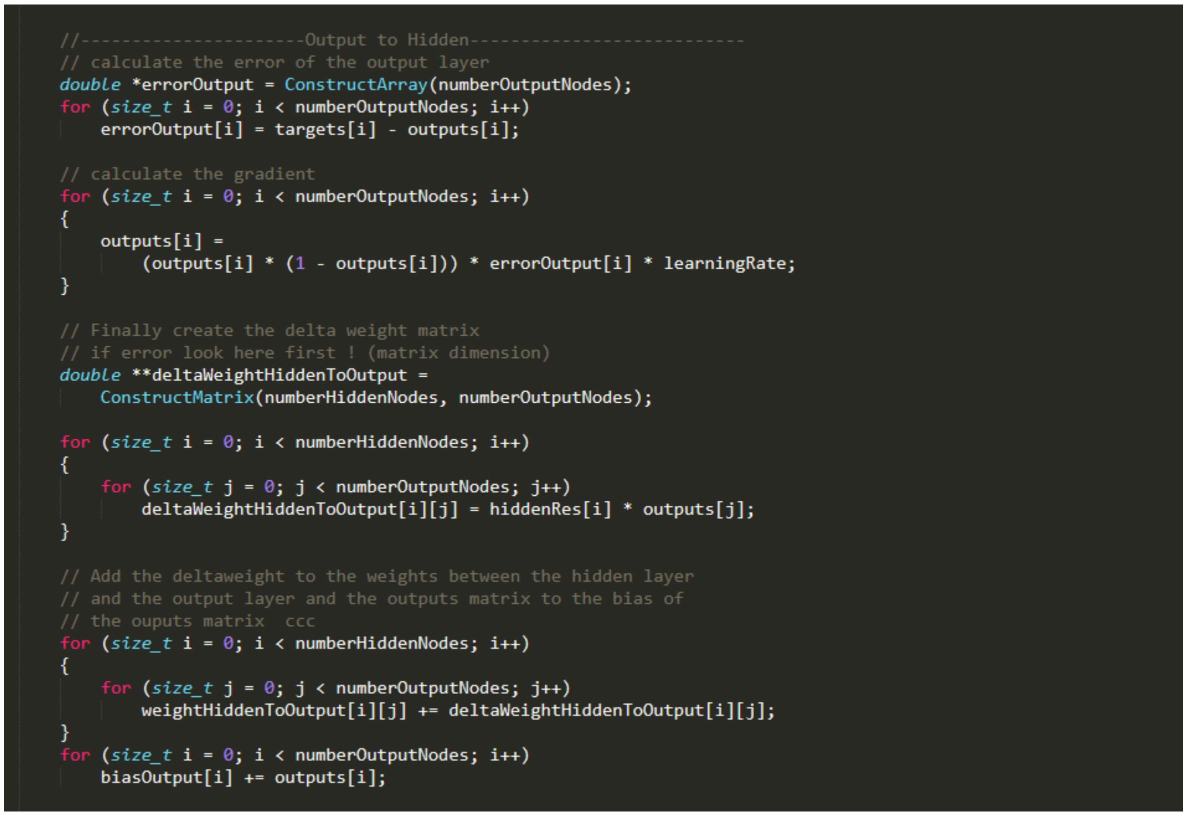
First, we compute again the values of $hiddenRes$ and $outputs$ arrays found during the function Predict. In other words we apply **Predict()**. For the sake of performances, we decided to copy and paste **Predict()** into the scope of the function Train() instead of only calling it and get the result, because we are going to need the intermediate step of **Predict()**, $hiddenRes$ and since this function returns only the results array, we would have need to compute it one more time than needed. The training of the neural network being already very time consuming.

After that, we compute the $errorOuputs$, a 1-dimensional array of size numberOutputNodes that contains the result of the matrix substraction of **targets** and **outputs**. Which means the difference between what we wanted and what we got.

It is relevant to compute the output error because it appears in the gradient formula. The gradient descent can be illustrated as the quickest direction for your function to decrease toward a local minimum of your cost function. We want to find this minimum in order to minimize the error of the prediction. Let's a 2-dimensional array $deltaWeightHiddenToOutput$ be the result of the matrix product between hiddenRes and the gradient descent for weight from hidden to output. It corresponds to the following formula:

$$deltaWeightHiddenToOutput = hiddenRes \times (errorOutput * \sigma'(outputs) * learningRate)$$

1. with the derivative of the sigmoïd function $\sigma'(x) = \sigma(x) * (1 - \sigma(x))$.
2. with the constant value $learningRate$.
3. $\times$ represents the usual matrix product
4. * represents the Hadamard matrix product.



```
//----------------------Output to Hidden--------------------------
// calculate the error of the output layer
double *errorOutput = ConstructArray(numberOutputNodes);
for (size_t i = 0; i < numberOutputNodes; i++)
    errorOutput[i] = targets[i] - outputs[i];

// calculate the gradient
for (size_t i = 0; i < numberOutputNodes; i++)
{
    outputs[i] =
        (outputs[i] * (1 - outputs[i])) * errorOutput[i] * learningRate;
}

// Finally create the delta weight matrix
// if error look here first ! (matrix dimension)
double **deltaWeightHiddenToOutput =
    ConstructMatrix(numberHiddenNodes, numberOutputNodes);

for (size_t i = 0; i < numberHiddenNodes; i++)
{
    for (size_t j = 0; j < numberOutputNodes; j++)
        deltaWeightHiddenToOutput[i][j] = hiddenRes[i] * outputs[j];
}

// Add the deltaweight to the weights between the hidden layer
// and the output layer and the outputs matrix to the bias of
// the ouputs matrix  ccc
for (size_t i = 0; i < numberHiddenNodes; i++)
{
    for (size_t j = 0; j < numberOutputNodes; j++)
        weightHiddenToOutput[i][j] += deltaWeightHiddenToOutput[i][j];
}
for (size_t i = 0; i < numberOutputNodes; i++)
    biasOutput[i] += outputs[i];
```

Here is a snippet of code of the backpropagation from the output layer to the hidden layer. For the computation of the gradient descent, instead of creating a new array and compute everything in it, we first compute the derivative of $\sigma$ which respect to the array $outputs$ and multiply (Hadamard product) it by $errorOutput$ and $learningRate$. We are allowed to do that because errorOutput is of the same size of outputs.

The $learningRate$ is a parameter that controls how much we are adjusting the weights of our network when modifications are required. If the learning rate is too small, gradient descent will takes time to converge toward the local minimum it tends to. However, if it is too large, the gradient descent can overshoot the minimum and may even never give us the local minimum, missing it each time. This is a value that the programmer sets in the scope of **Train()** (for example, we decided to put it at 0.1 at first).

After this small aside, after computing a part of the gradient descent. We need to multiply what we have computed for $outputs$ with $hiddenRes$. This will be filled in a 2-dimensional array called $deltaWeightHiddenToOuput$.

Afterwards, each value of $deltaWeightHiddenToOuput$ is added to the corresponding one in the 2-dimensional array $weightHiddenToOutput$. For $biasOutput$, we will only need to add the gradient descent to it, because according to the Chain Rule, the partial derivative of the weighted sum (in example $W * a + b$) which respect to the bias $b$ is equal to 1. Thus we have the following formula:

$$gradientDescent_{outputToHidden} = \sigma'(outputs) * errorOutput * learningRate$$

$*$ being the Hadamard matrix product

We have now back-propagated from the output layer to the hidden layer and modified the values of weights and biases. The next process is similar for the hidden layer to the input layer except that computations to find the hidden error are quite different. Indeed, it needs the value of $errorOutput$. $errorHidden$ is equal to the matrix product of errorOutput and the 2D array $weightHiddenToOutput$ .

$$(Eq_1) : errorHidden = errorOutput \times weightHiddenToOuput$$

$\times$ being the usual matrix product

After computing hidden error, we can now compute the gradient descent for weight from input to hidden . This is the same formula as before, except that we differentiate respectively to values of $hiddenRes$ and multiply it by values of $errorHidden$ and $inputs$. Of course, we don't forget to multiply by the learning rate. The gradient descent for weight from input to hidden will be denoted as $deltaWeightInputToHidden$. Thus we have :

$$(Eq_2) : deltaWeightInputToHidden = inputs \times (\sigma'(hiddenRes) * errorHidden * learningRate)$$

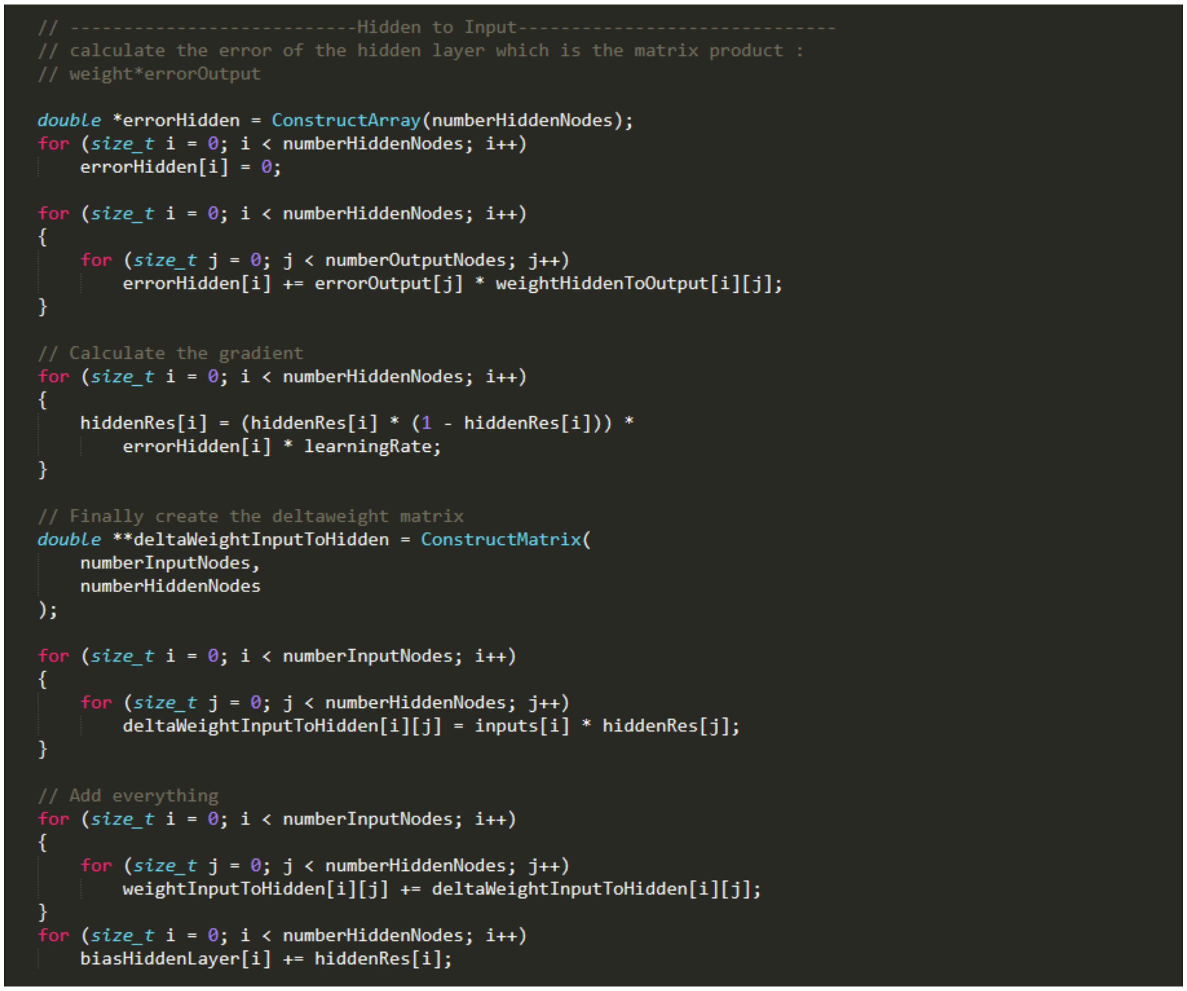
1. with the array of inputs values, $inputs$.
2. $\times$ corresponds to the usual matrix product
3. $*$ corresponds to the Hadamard matrix product
4. with the derivative of the sigmoïd function $\sigma'(x) = \sigma(x) * (1 - \sigma(x))$.
5. with the constant value $learningRate$ .

We then add $deltaWeightInputToHidden$ to $weightInputToHidden$ and Apply the same process as before on $biasHiddenLayer$ because of the same reason as $biasOutput$ one. Hence :

$$gradientDescent_{hiddenToInput} = \sigma'(hiddenRes) * errorHidden * learningRate$$

$*$ being the Hadamard matrix product.

Here is our way to implement what we have previously said:



```
// ----------------------------Hidden to Input------------------------------
// calculate the error of the hidden layer which is the matrix product :
// weight*errorOutput

double *errorHidden = ConstructArray(numberHiddenNodes);
for (size_t i = 0; i < numberHiddenNodes; i++)
    errorHidden[i] = 0;

for (size_t i = 0; i < numberHiddenNodes; i++)
{
    for (size_t j = 0; j < numberOutputNodes; j++)
        errorHidden[i] += errorOutput[j] * weightHiddenToOutput[i][j];
}

// Calculate the gradient
for (size_t i = 0; i < numberHiddenNodes; i++)
{
    hiddenRes[i] = (hiddenRes[i] * (1 - hiddenRes[i])) *
        errorHidden[i] * learningRate;
}

// Finally create the deltaweight matrix
double **deltaWeightInputToHidden = ConstructMatrix(
    numberInputNodes,
    numberHiddenNodes
);

for (size_t i = 0; i < numberInputNodes; i++)
{
    for (size_t j = 0; j < numberHiddenNodes; j++)
        deltaWeightInputToHidden[i][j] = inputs[i] * hiddenRes[j];
}

// Add everything
for (size_t i = 0; i < numberInputNodes; i++)
{
    for (size_t j = 0; j < numberHiddenNodes; j++)
        weightInputToHidden[i][j] += deltaWeightInputToHidden[i][j];
}
for (size_t i = 0; i < numberHiddenNodes; i++)
    biasHiddenLayer[i] += hiddenRes[i];
```

We are doing the same things than the backpropagation from the output layer to the hidden layer except that we are dealing with different variables. We first create the 1-dimensional array $errorHidden$ and fills it with 0's. We then compute his value as previously shown at $(Eq_1)$. We then compute the gradient descent. Instead of using an intermediate array, we prefer to store everything in $hiddenRes$ because it will be easier to update $biasHiddenLayer$. We then use the $(Eq_2)$ which correspond in the code at the part where we initialize the 2-dimensional matrix $deltaWeightInputToHidden$ and fill it with each value of the array $inputs$ multiplied by value of $hiddenRes$. The only thing left to do is to add each values of $deltaWeightInputToHidden$ to $weightInputToHidden$ The $biasHiddenLayer$ will only need to be updated thanks to $hiddenRes$.

We have now a function that can ”train” the network by applying modifications on each weight and bias of each layers. By calling many times this function, the weights and bias will converge towards the values that give the less error possible for a prediction.

The training part requires a certain amount of data to feed the neural network with. This is called a dataset. Since we decided to train our neural network in order to make it capable of recognizing 85 characters, our dataset was composed of 85 folders, each one dedicated to one char, containing a lot of images. Each folder had a name between 0 and 84, and each images were named between 0 and the number of the image in the folder - 1. Naming like this the dataset, enabled us to generate in a very easy way the path corresponding to each image. All the images had the shape of a square of 24*24 pixels (16*16 at the beginning), white with the character right in the middle of the picture. The main problem with the dataset was to find elements that would perfectly correspond to what would be given to the neural network later on by the segmentation. To create the dataset, we first searched for existing ones, but we were not able to find some having at the same time characters (from the alphabet and numbers) and special characters (characters of punctuation). Eventually we decided to use the dataset named 74k since it was very complete (containing 1016 images by characters), but the problem was that it did not contains special chars. We then tried to create a word document and copy all the characters we wanted in as many fonts as we wanted and use our segmentation to recover every character one by one in a square image of the size wanted by a simple algorithm of resize. Using a modulo, the function, not only were creating the new images, but it was also renaming it and putting it in the right folder (done thanks to a modulo 85).

Here some examples extracted from our dataset :

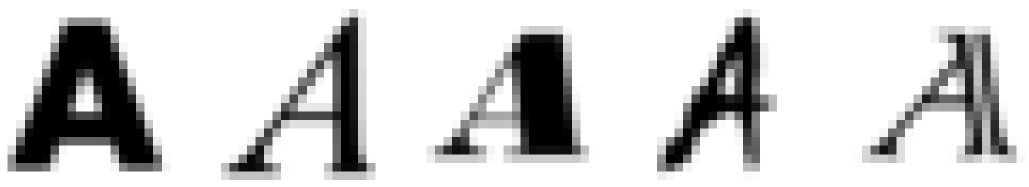


Figure 16: Samples of the letter A

The dataset gave us a really hard time, we had to do it many times (maybe five or six times), because each try was either to complete (at the beginning with the 1016 images by characters) or with not enough pictures, or with characters too different from one to another.

Finally we decided to create more than one dataset for our project, one that can be used with 16*16 images, another one with 24*24 images, one without the special characters... and so on. This unable us to train differently our neural network, in such a way that if we need only to recognize letters, we can use to structure the neural network, the file with the weights we got after training with the only-letter dataset.

In order to get an efficient training, we decided to train more than expected, since we had some time left, to be sure that all the weights had converge toward their right value. To do so, we created a script **Tool.c** (which is in the folder resources), and created a function enabling ourselves to train as many as time as we wanted. Basically the function creates the neural network (calls the Start() function), load the right weights (if there is some to load) from the file nn.data. After that, we use the function rand() two times, once to obtain a number between 0 and 84 and the other depending on how many images we had in each folder of our dataset. As said before, the path to the right images was easy to retrieve from here, and we just had to train with the image we got. This permitted to train with random images the neural network.

Some other functions needed also to be created, as the one that generate an input according to some images or the one retrieving the right output. We launch our function in order to make one million loop (so to launch the function Train() one million time) and then created a bash script that launched the exec Tool thirty time. The neural network then trained for thirty millions times (which took more than 20 hours straight). To ensure good results, we also reduced the learning rate the more we trained.

Then to test our result we just wrote some texts, passed it through the segmentation, and call the function Predict() as many time as there were characters in our text. The only thing left was to compare the obtained results with the ones expected.

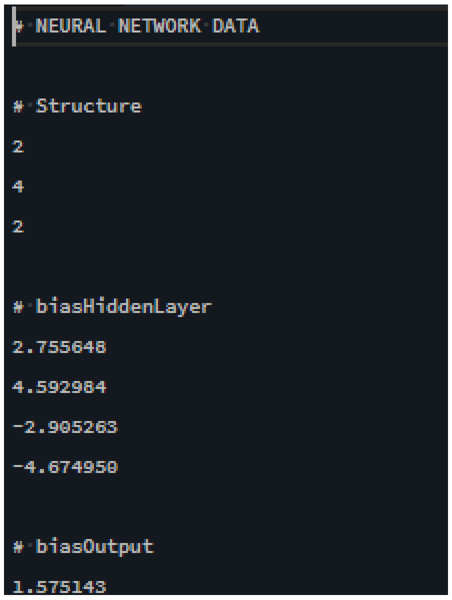
As said before, the results after training with 16*16 pixels images were satisfying, but the ones after the training 24*24 images were excellent and all the errors made by the neural network were very understandable (such as mixing o, O and D or l I and ]...). The spell check also improve sometimes the results given by our neural network, but unfortunately the English dictionary being not available on the school computers, we were not able to show the results in the last defense.

## 6.4 Saving and loading

In order for us to be able to train only once the neural network and rely on the same trained networks later on, we implemented a simple load and save feature.

The algorithm is saving, in a file, the structure of the neural network (number of input nodes, number of hidden nodes and number of output node), the bias of the hidden layer, the bias of the output layer, the ”input to hidden” weights and finally the ”hidden to output” weights. While loading values from the specified file, only floating numbers are considered. It allows us to put comments in the file to make debugging and tweaking convenient. We may consider, later on, to use an already existing format like JSON to make our software scalable.

An example of the beginning of the file, created after training for XOR:



```
# NEURAL NETWORK DATA

# Structure
2
4
2

# biasHiddenLayer
2.755648
4.592984
-2.905263
-4.674950

# biasOutput
1.575143
```

# 7 Postprocessing

## 7.1 Spellcheck

In order to improve the resulted text, we added a postprocessing step to the project. Detecting spelling mistakes for each words allows to reduce greatly errors from the neural network. Concerning the technical aspect, we decided to rely on Hunspell, a library based on MySpell. The advantage of Hunspell is its compatiblity with UTF-8 dictionaries. Moreover, it is used by many important open source projects: OpenOffice, Thunderbird, Chronium... For our application and for the moment we load only english dictionaries.

Hunspell allows us to use two functions: Spell and Suggest. The first one checks if a given word is correct and the second one returns a list of words depending on a given word. For each word, we first check if it is misspelled. If it is the case we replace it with the first one given by suggest. In the future it would be interesting to pick the one that has the most letters in common. Indeed, the objective is not to correct spelling mistakes but to prevent words from having only one or two letters not recognized by the neural network.

The above table emphasizes on the significance of the spellcheck step. Many words have an error only on one letter, the postprocessing step provides an excellent solution.

| **Before spellcheck** | **After spellcheck** |
|---|---|
| hsve | have |
| SpecTfications | Specifications |
| maxTmum | maximum |
| purumeters | parameters |
| neuraI | neural |
| eJition | edition |

# 8 GUI

## 8.1 Main GUI

For the last defense, we had to implement a graphical user interface (GUI). Since we had issues with Glade, we were not able to use it. That is why our GUI is created from scratch by only using GTK. This choice was also with the willing to fully understand how a graphical interface is built.

Here is a sketch:

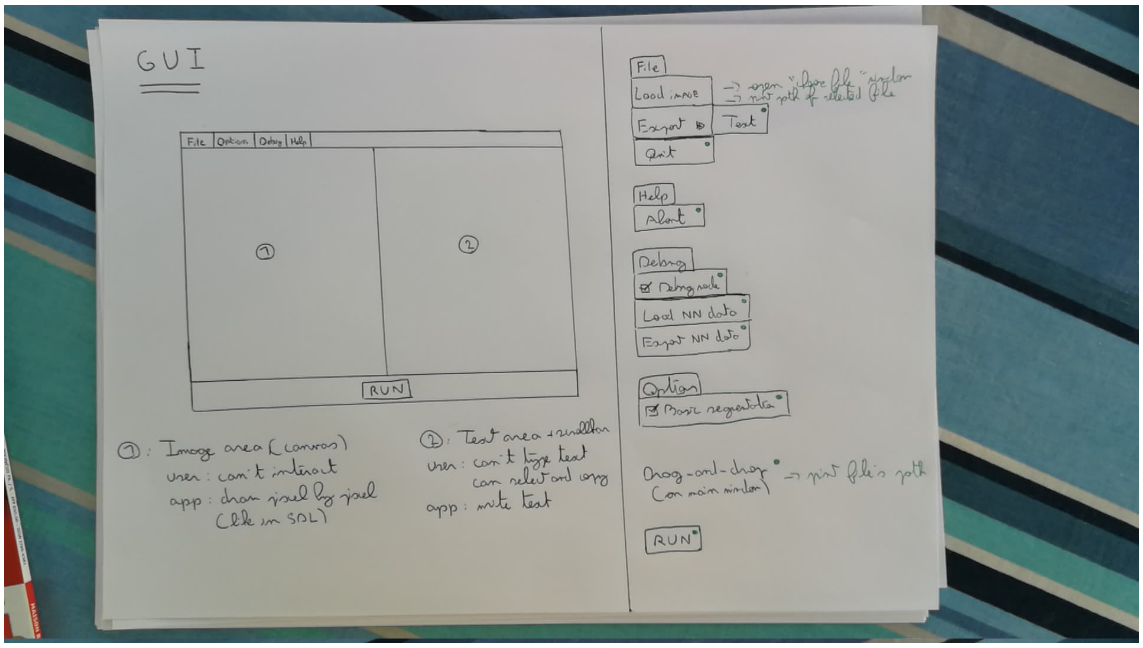


Figure 17: Sketch of GUI

Since the documentation is most of the time in GTK 2, most of the functions are then deprecated which complicated the task. However, we succeed to build a graphical interface that fits with our expectations even if some features like the ”drag and drop” were put aside due to a lack of time.

The final result looks like this:

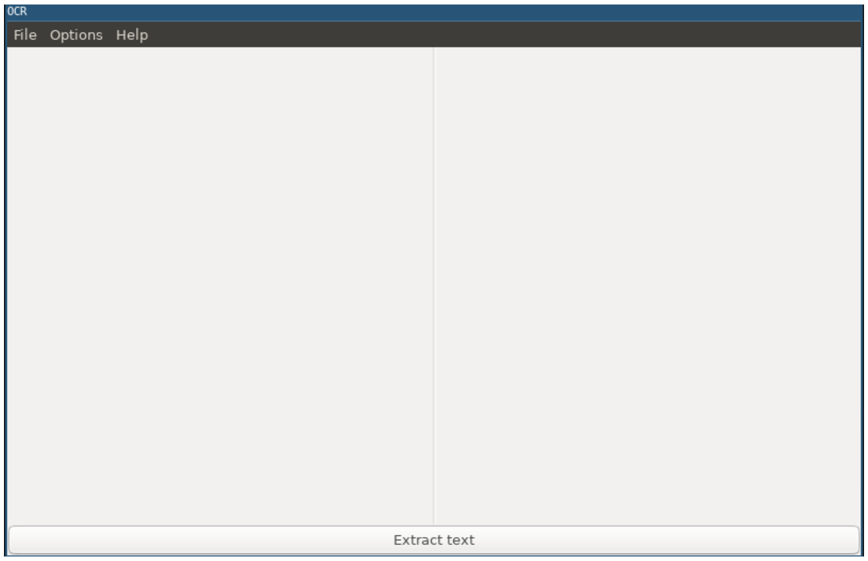
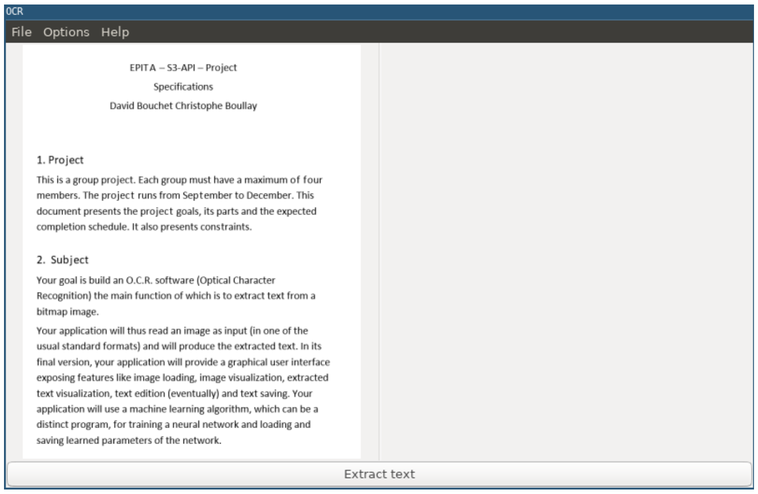


Figure 18: Implemented GUI

First, we have a customized menu bar. The **File** menu contains *Load Image*, *Export* and *Quit*. *Load Image* enable us to load a image in the left part only in the following format: jpg, jpeg, png and bmp.
Here is how it looks.



EPITA – S3-API – Project

Specifications

David Bouchet Christophe Boullay

1. Project

This is a group project. Each group must have a maximum of four members. The project runs from September to December. This document presents the project goals, its parts and the expected completion schedule. It also presents constraints.

2. Subject

Your goal is build an O.C.R. software (Optical Character Recognition) the main function of which is to extract text from a bitmap image.

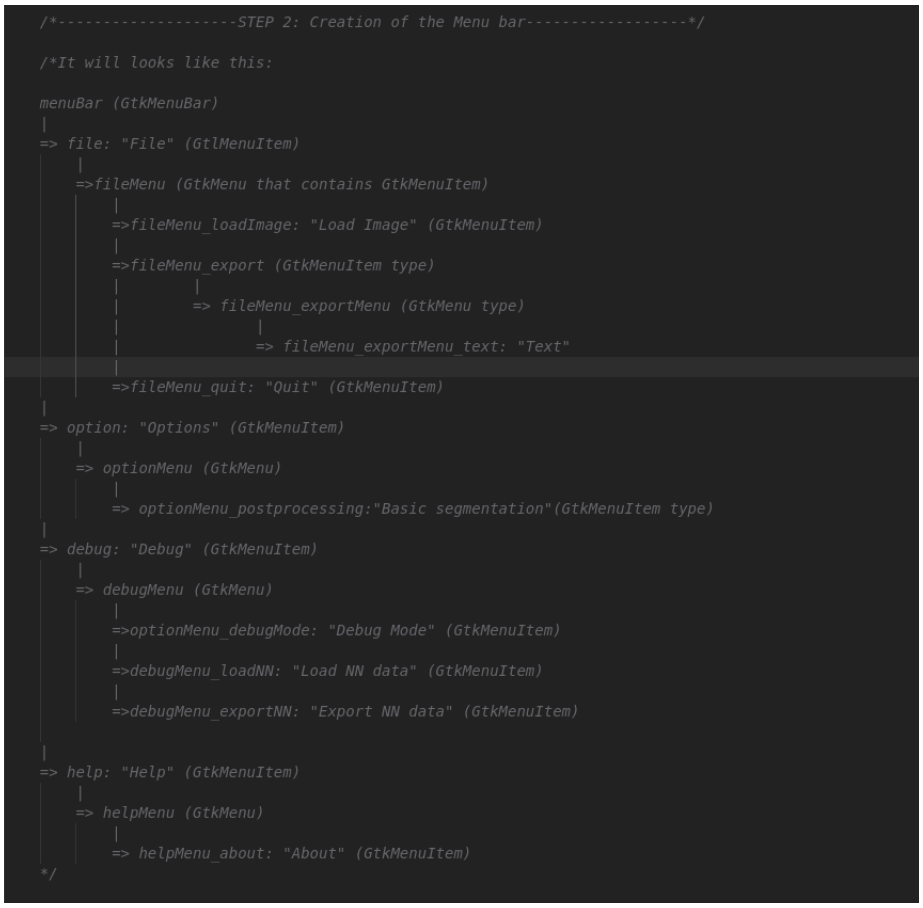
Your application will thus read an image as input (in one of the usual standard formats) and will produce the extracted text. In its final version, your application will provide a graphical user interface exposing features like image loading, image visualization, extracted text visualization, text edition (eventually) and text saving. Your application will use a machine learning algorithm, which can be a distinct program, for training a neural network and loading and saving learned parameters of the network.

Figure 19:

Notice that the image fits perfectly in the box. If we decide to scale down the box, the image will do so. The **File** menu also contains *Export* which enable us to export in a text file the result after applying the algorithm on it. The last feature *Quit* enables us to close the GUI. Notice that the shortcut ”CTRL + q” implemented by us, also enables us to close the window.

The **Options** menu contains 3 buttons that you can activate : *Enable debug mode*, *Enable noise cancelling and contrast enhancement* and *Enable postprocessing (spell check)*

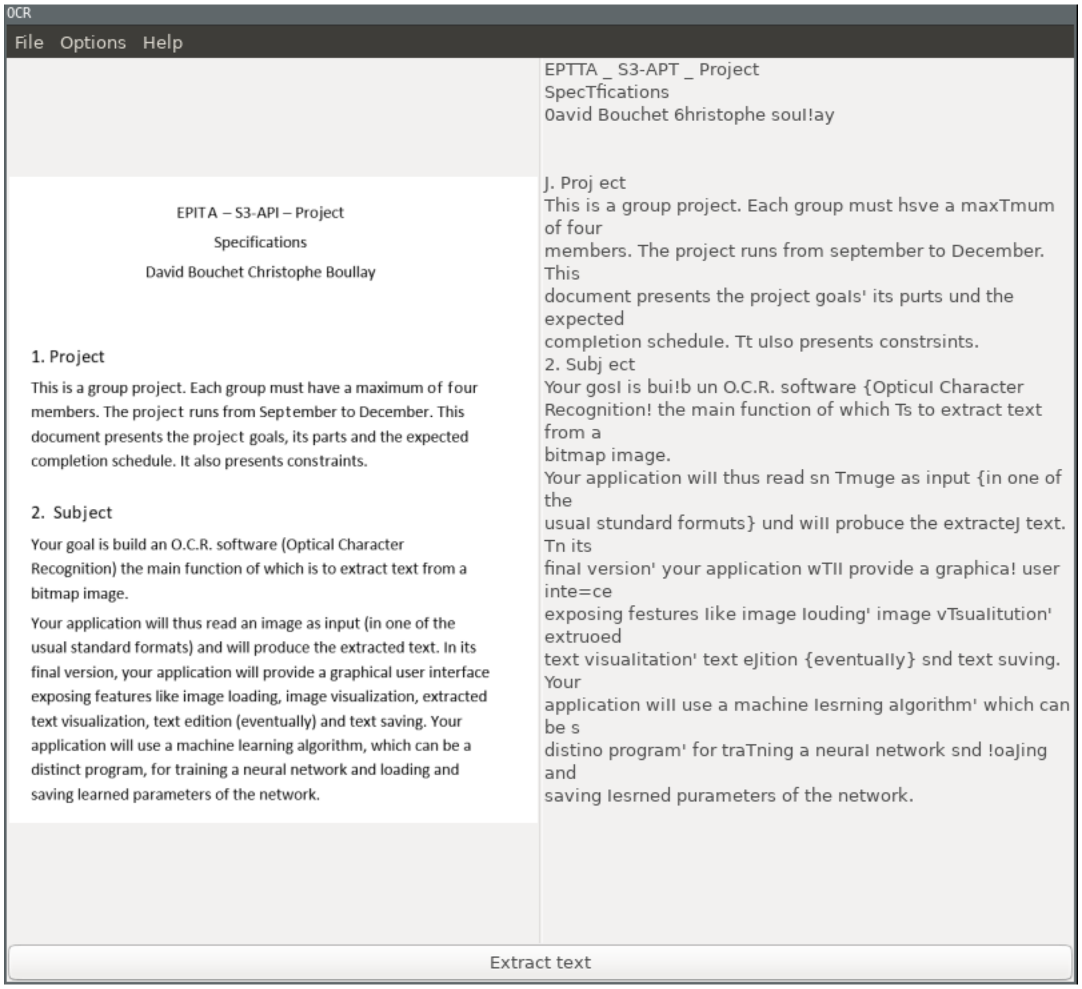
The last feature is **Help** that contains *About*. This one is clickable and redirect us to our Github page.
Here is the pseudocode of the menu bar:



```
/*-------------------STEP 2: Creation of the Menu bar-----------------*/

/*It will looks like this:

menuBar (GtkMenuBar)
|
=> file: "File" (GtlMenuItem)
    |
    =>fileMenu (GtkMenu that contains GtkMenuItem)
        |
        =>fileMenu_loadImage: "Load Image" (GtkMenuItem)
        |
        =>fileMenu_export (GtkMenuItem type)
        |        |
        |        => fileMenu_exportMenu (GtkMenu type)
        |               |
        |               => fileMenu_exportMenu_text: "Text"
        |
        =>fileMenu_quit: "Quit" (GtkMenuItem)
|
=> option: "Options" (GtkMenuItem)
    |
    => optionMenu (GtkMenu)
        |
        => optionMenu_postprocessing:"Basic segmentation"(GtkMenuItem type)
|
=> debug: "Debug" (GtkMenuItem)
    |
    => debugMenu (GtkMenu)
        |
        =>optionMenu_debugMode: "Debug Mode" (GtkMenuItem)
        |
        =>debugMenu_loadNN: "Load NN data" (GtkMenuItem)
        |
        =>debugMenu_exportNN: "Export NN data" (GtkMenuItem)

|
=> help: "Help" (GtkMenuItem)
    |
    => helpMenu (GtkMenu)
        |
        => helpMenu_about: "About" (GtkMenuItem)
*/
```

Figure 20: Pseudocode of the menu bar

The last feature is **Extract Text** which enable us to run the neural network and output the result in the right screen of the GUI.
Here is the result:



EPITA – S3-API – Project
Specifications
David Bouchet Christophe Boullay

1. Project

This is a group project. Each group must have a maximum of four members. The project runs from September to December. This document presents the project goals, its parts and the expected completion schedule. It also presents constraints.

2. Subject

Your goal is build an O.C.R. software (Optical Character Recognition) the main function of which is to extract text from a bitmap image.

Your application will thus read an image as input (in one of the usual standard formats) and will produce the extracted text. In its final version, your application will provide a graphical user interface exposing features like image loading, image visualization, extracted text visualization, text edition (eventually) and text saving. Your application will use a machine learning algorithm, which can be a distinct program, for training a neural network and loading and saving learned parameters of the network.

```
EPTTA _ S3-APT _ Project
SpecTfications
0avid Bouchet 6hristophe soul!ay


J. Proj ect
This is a group project. Each group must hsve a maxTmum
of four
members. The project runs from september to December.
This
document presents the project goaIs' its purts und the
expected
compIetion scheduIe. Tt uIso presents constrsints.
2. Subj ect
Your gosI is bui!b un O.C.R. software {OpticuI Character
Recognition! the main function of which Ts to extract text
from a
bitmap image.
Your appIication wiII thus read sn Tmuge as input {in one of
the
usuaI stundard formuts} und wiII probuce the extracteJ text.
Tn its
finaI version' your appIication wTII provide a graphica! user
inte=ce
exposing festures Iike image Iouding' image vTsuaIitution'
extruoed
text visuaIitation' text eJition {eventuaIIy} snd text suving.
Your
appIication wiII use a machine Iesrning aIgorithm' which can
be s
distino program' for traTning a neuraI network snd !oaJing
and
saving Iesrned purameters of the network.
```

Figure 21: GUI after clicking on "Extract Text" button

We had a lot of problems implementing the GUI. First, since boxes do not have a fixed size and depending on the element we are putting in it, the size of each box can change which is not a good thing. Also, we have the fact that the window does not update, causing little display bug. One way was to move the mouse out of the GUI zone so that this one can update. Since this solution isn't a clean one, we decided to find a way to fix that properly. A very annoying bug was the one enabling us to display the result of the neural network at the right box of the GUI. It wasn't displaying correctly since we discover that after initializing gtk, this one take the current language of your machine causing trouble when it comes to the displaying.

So we have to call *gtk_disable_setlocale* before so that it chooses the default language. This problem was very difficult to find since we were on the tat the initialization of gtk transformed our matrices of type double to a float one.

## 8.2 Debug GUI

We also made a very simple Graphical User Interface (GUI) that allows us to see the different steps of the main algorithm. It is made using the SDL2 library and is created at the end of the program with the function StartDemoGUI. It was very helpful during debugging. This function receives information from the preprocessing and the segmentation steps described before. The program waits for a key to be pressed and then show the steps in this order:

1. Grayscale image ;
2. Binarized image (black/white) ;
3. Result of the RLSA algoritm (detecting paragraphs) ;
4. Paragraphs and lines bounds ;
5. Words and characters bounds.

# 9 Conclusion

## 9.1 Our thoughts

So far, it has been a pleasure to work on this OCR. Even more because we are very interested in the project and it is a start for us to work with a low level programming language. Finally, this project allows us to put into practice all knowledge acquired in lectures, tutorials and practicals.

Furthermore, it is significantly improving our personal skills and this four months project will be a second concrete project for us. Hence it will be very rewarding. This is the opportunity to work on these skills:

- Project organization:
  - Project management and planning
  - Team working
  - LaTex for reports
  - Git for versionning
- Technical knowledge:
  - C language, Makefile
  - Machine learning with neural networks
  - Learning how to learn

## 9.2 Personal thoughts

”This project is awesome! I really liked the fact that we depend on nothing else but our code (nothing like unity for example) and a few libraries. I really loved the last project, the game, but here we are more free, we can create our own functions, if we have bugs we know that it is our fault and that with a little motivation we can fix them. I am thrilled that some other teammates want to continue this project after the last defense because I would love to go on as well and learn more about machine learning but also learn about other parts such as the GUI or the character detection. I am very proud of what we did in this project, I think it is very completed and the results are way more than satisfying.”

Arielle LÉVI

”During this project, I was assigned to the character-recognition part involving the development of the neural network and also to the GUI part. Since I wasn’t used to C language and find it troublesome, we decided to only code in one computer, especially Arielle’s since she has a better grasp on the language and rather focus myself to debug the code at the end, with knowledge I have gathered. I feel like the project is a good opportunity for me to learn since we are only left with documentation which is not that bad.

Ferdinand MOM

”This project was a really good thing. It was interesting, made us to interest ourselves about various different subjects but more than anything it was a great way to discover the dynamics of a group project. Indeed, a group project in IT is not just about code. It is about people. And I have been a very bad teammate. For the first defense, I did not manage to get the segmentation working in time and it earned us 13,5 out of 20, where we could have a way better mark as our XOR was working. For the second defense, I had to implement deskew and I did not do it. I think they learned not to trust me and I learned that I was not reliable. At the beginning of the project, I asked Theo to join the team because I was sure it was my chance to prove that I was not who they thought. Well I failed, and as I do not want to work with bad teammates in the future, I have work to do on myself, that is a sure thing.”

Louis CHEVALIER

”It has been a real pleasure to work on this project from the beginning with a simple prototype to a complete application. Making a software capable of recognizing characters in a image was very challenging. I felt involved in the project as soon as we received the book of specifications. I will surely continue improving the code and adding more features. Indeed, after three months working on this software, I think we can make something even more smart and autonomous and implement it in our daily life.”

Théo LEPAGE

## 9.3 For the future

Several members of the team are looking forward to continue the development of our Optical Character Recognition software. We have already organized our future work by written down every improvements to implement. From now on our main objective will be to re-factor the code to make each part an independent module. Finally, we will improve recognition capabilities by tweaking and enhancing the different algorithms. The idea is to keep the source code available on GitHub to eventually work with other passionate.

Thank you for having read us.

The None Of All Of The Above Team