

# S3 Project - Letscallitaday - First report

Romain Le Miere • Lucas Tilly • Hugo Schreiber • Emmanuel Vouillon

November 2022



# Contents

<b>1</b>	<b>The group</b>	<b>4</b>
1.1	Who we are . . . . .	4
1.1.1	Romain . . . . .	4
1.1.2	Hugo . . . . .	4
1.1.3	Lucas . . . . .	4
1.1.4	Emmanuel . . . . .	4
1.2	Task distribution . . . . .	5
1.3	Organisation . . . . .	5
<b>2</b>	<b>Workflow</b>	<b>6</b>
2.1	git . . . . .	6
2.1.1	Branches . . . . .	6
2.1.2	CI/CD . . . . .	6
2.2	Code formatting . . . . .	7
2.3	Documentation . . . . .	7
<b>3</b>	<b>Progress of the project</b>	<b>8</b>
3.1	Image processing . . . . .	8
3.1.1	Image loading . . . . .	8
3.1.2	Color removal . . . . .	8
3.1.3	Gaussian blur filter . . . . .	9
3.1.4	Manual rotation . . . . .	9

3.1.5	Lines detection . . . . .	10
3.1.6	Intersections detection . . . . .	10
3.1.7	Squares separation . . . . .	10
3.1.8	Handling incomplete pictures . . . . .	11
3.2	Networking . . . . .	11
3.2.1	Math lib . . . . .	11
3.2.2	Neural network . . . . .	11
3.3	Solving . . . . .	11
3.3.1	Backpropagation . . . . .	12
3.3.2	Saving network . . . . .	13
3.3.3	Loading Network . . . . .	13
3.4	Loading Data . . . . .	14
3.5	Grid image to text-file . . . . .	15
3.6	Sudoku solving algorithm . . . . .	15
3.7	Graphical interface . . . . .	16
<b>4</b>	<b>CLI</b>	<b>18</b>
<b>5</b>	<b>Future expectations</b>	<b>19</b>
5.1	Backpropagation . . . . .	19
5.2	UI . . . . .	19

# **1 The group**

## **1.1 Who we are**

### **1.1.1 Romain**

My name is Romain and I am 18 years old. As project master, this project taught me a lot both about gitlab and git proper workflow, and about C programming. Since I was in charge of the barebones implementation of the project, I learnt a lot about structures and object abstraction. Moreover, I learnt a lot about CI/CD using gitlab pipelines.

### **1.1.2 Hugo**

My name is Hugo and I am 18 years old. I find this project very interesting because it allows us to discover complex things like a neural network, to be implemented in a language that we know only recently, the C language. What interests me the most in this project is all the mathematics behind the learning of the neural network. That is why I have been doing the research on the different algorithms allowing the network to learn.

### **1.1.3 Lucas**

My name is Lucas, I am 19 years old and I mostly took care of the detection of the Sudoku. This project was a good opportunity to be familiar with the C language and to improve our capacity to work as a team. Overall, even though I often faced difficulties, I always managed to overcome them and finally succeeded what I had to do.

### **1.1.4 Emmanuel**

My name is Emmanuel and I find this project has been quite the mixed bag. Working with the C language has brought its fair share of difficulties but I feel like being pushed into such a project has really taught me a lot.

## 1.2 Task distribution

To ensure a steady progress we distributed the different tasks between the members of the project as follows:

Romain:

- Basic math lib implementation.
- Neural network implementation.
- Loading and saving the neural network.
- Loading csv data for training and testing.
- CLI

Lucas:

- Line detection
- Intersection detection
- Separation of each square

Hugo:

- Sudoku solving algorithm
- Converting the image to the neural network
- Maths behind a neural network

Emmanuel:

- Color removal on the image
- Application of filters
- Manual rotation of the image
- Creation of a showcase UI

## 1.3 Organisation

In order to best organize the project, we decided to use the tools available on gitlab, i.e. issue planning with milestones and labels. In fact we were able to follow the progress of each team member, which allowed us to know when to merge the git branches when it was necessary.

## **2 Workflow**

### **2.1 git**

For the git, we decided to host our project on epita gitlab to get the CI/CD of gitlab. Furthermore, gitlab allowed us to create issues and proper merge requests and provide us a graphical interface for the review of the project. To avoid having to push on both repositories (epita gitlab and S3 project dedicated git), we decided to setup Repository mirroring.

#### **2.1.1 Branches**

In order to keep the workflow clean, we decided to go for merge requests. Each Issue was bound to a merge request only one member could work on, and each merge request has to be reviewed by another member to ensure the code is understandable to anyone who has not worked on the project. This workflow made every member of the group responsible for more or less all the code base, increasing productivity when working with the code of others.

#### **2.1.2 CI/CD**

We decided also to add CI/CD to the project so that every branch would be checked beforehand. These three things are checked when pushing to a branch to ensure the viability of the branch :

- Linting
- Build
- Tests

First, we wanted to stick to some formatting convention for our code so it would be kept clean during all the project; thus, **clang-format** has been adopted alongwith some config file at the base of the repo. The first stage of the pipeline checks whether the code has been properly formatted by everyone, causing the pipeline to fail if the code was not properly formatted. Once the formatting of the code is checked, the code is compiled for production to see if it builds properly in production on almost baremetal distro. Finally, the compiled code is bundled into a static library that is testing using unit testing with criterion. Criterion is a unit testing library used in our project on our machines to produce test codes ran by the gitlab pipeline. It can detect memory leaks when running code from our API, and also test each features separately. As of now, only a few features are properly tested in the pipeline, but the final goal is to have all our features properly tested.

## 2.2 Code formatting

As said earlier, *clang-format* is widely used in the project to ensure coding style consistency amongst all files in the project.

## 2.3 Documentation

To ensure that every member of the group could understand all of our exposed API, we decided to write documentation for each function exposed using *doxygen* syntax. Therefore, each function, struct or type is properly documented, and man pages and html website are generated. As for all basic action in the project, the process has been automatized in the *Makefile* so that a simple command allows to generate the whole docs for the project.

## 3 Progress of the project

### 3.1 Image processing

#### 3.1.1 Image loading

Using the library SDL2, loading images was a simple task: we have made a small function which turns an image into a surface so that it can be transformed and used later on.

#### 3.1.2 Color removal

Color removal has been divided in two steps; gray-scaling and turning into black and white.

The former has been realized by applying a function on the red, green and blue values of each pixel of the image.

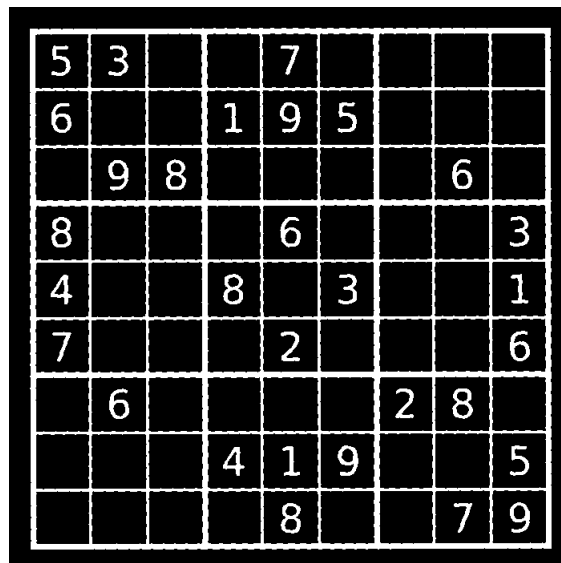
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

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

Figure 1: A grayscaled Sudoku obtained using our program, compared to its colored counterpart



The latter required more work. In order to obtain an adaptive threshold depending on the image we were processing, we started by implementing Otsu's method. Once the threshold calculated, it was child's play to turn an image into a black and white image: each pixel on the image which was above the threshold was turned white and each pixel below was turned black.



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

Figure 2: A black and white Sudoku obtained using our program

### 3.1.3 Gaussian blur filter

In order to reduce noise in the images, we implemented a Gaussian blur filter.

### 3.1.4 Manual rotation

To ease the recognition of the different parts of the Sudoku, we implemented a manual rotation of the image. This allows us to rotate any image around its center by a given rotation.

### **3.1.5 Lines detection**

Now that we had a good picture with which we could work with, we had to start using it to detect the lines separating each square of the Sudoku's grid. To detect those lines, we used the Hough Transform method which seemed to be the most efficient one and which suited the best to our goal. During the process of using this method, we had to think of a way to have a good threshold so that we would only detect lines which were real lines. To take care of that issue, we used an adaptive threshold set on number of detected lines. this way, we would detect the x most prominent lines. However, due to the thickness of the lines in the pictures, way too many lines were detected in a single big line. Therefore, we had to first reduce this number by merging the lines to reduce their thickness to one pixel for each line.

### **3.1.6 Intersections detection**

To detect the intersections between the lines of the Sudoku that we stored and have them in the right order for the incoming separation of each square, we first separated our lines in 2 different lists: one for the horizontal ones and a second one for the vertical lines. With those lines sorted by their distance from the origin, we finally had what we needed to detect the four intersections of each square.

### **3.1.7 Squares separation**

Having all the sorted intersection allowed us to split all the squares one by one. However, this much is not enough, we also had to apply some transformations to each of them to make them optimal for the neural network. Therefore, we delete the lines of the Sudoku if there are some in our picture. Then, we "zoom in" so that the number we have takes the maximum space provided. Finally, we reduce the quality of our picture so that it fits the 28x28 condition for the neural network to work (because of the set we used to train it)

### **3.1.8 Handling incomplete pictures**

Even though some Sudokus were simple enough to handle, some of them had some lines not thick enough to be detected. Therefore, when we have enough lines already, we fill out the remaining ones to proceed to the end of the program.

## **3.2 Networking**

### **3.2.1 Math lib**

To properly apply calculations, a math lib had to be implemented for matrices, amongst other to keep the neural network code as clean as possible. Thus a matrix object has been implemented along with basic operations like addition, scalar multiplication and matrix multiplication.

### **3.2.2 Neural network**

As well as the Math lib, the neural network had to be properly implemented to allow it to be easily debugged considering the complexity of the neural network structure. We decided to implement each neuron separately, each of them containing the values of the weights for the previous layer in a row vector, and the bias for the neuron allowing each neuron to calculate its result independently. A layer struct has also been implemented containing a list of neurons in the given layer, and the length of this list along with the activation function struct. This layer can be computed on its own based on the column vector of inputs, and returns a column vector containing the outputs of each neuron. Finally, the neuron struct contains the list of layers, except the input layer, the number of inputs, and the number of layers.

## **3.3 Solving**

After the neural network in itself was built, the back propagation algorithm could be implemented in the code. As well as the neural network or the math lib, we wanted this code to be highly readable, therefore, the activation function mechanism was refactored so that each layer would contain its activation function along with the derivative of its activation function for the backpropagation, and a unique name to identify the function.

The backpropagation has been implemented using the gradient descent algorithm. As for the feedforward, each layer can be backpropagated independently from its predecessor. Since this algorithm requires to have the result of each layer, we had to save each result in a structure linked to the network when doing the feedforward to keep the data for the backpropagation.

### 3.3.1 Backpropagation

- FeedForward

In order to train the neural network the first step is to pass each image from the training database into the network in order to get all the outputs of each image. We have chosen the mnist database, which offers 50 000 images of handwritten numbers.

- Backpropagation itself

After recovering each output, we calculate the distance it has with the expected output. To do this we use what is called a cost function, currently we use the MSE (main squared error):

$$\sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$\hat{y}_i$  being the expected output of the neuron

The cost function is a key point during the learning process because it allows to know how well the network has learned. Indeed, the closer the value of the cost function is to 0, the more efficient the network is. So the goal of backpropagation is to reduce this value as much as possible.

We will then calculate the gradient of this value with respect to each weight and each bias of the network. The gradient is the partial derivative of the cost function with respect to the weight, in other words, we calculate the impact of the weight or the bias on the total error. If the weight or bias helps to reduce the value then we increase its value, if the weight or bias increases the total error, then we decrease it.

So we repeat this process on each image of the training base *epoch* times. The *epoch* number is the number of training generations, the bigger the number, the more the network will learn.

### 3.3.2 Saving network

When the neural network has been trained, it has to be saved somewhere to be used afterwards without training it all the way again. First of all, a format for the saving had to be chosen to keep things clear. We decided to write headers for each structure in the network. First, the file starts with the number of layers in the network (except the input layer), and the number of inputs comma-separated. Then, each layer was described by a header containing the number of neurons and the name of the activation function for this layer. Then, the next  $n$  lines contain the comma-separated list of weights, and the bias as last element. This architecture was chosen because it was easily understandable both by humans and by computers. With the architecture set, we only had to save the actual neural network in a file as specified by this file formatting.

### 3.3.3 Loading Network

Once the format set, we had to implement a parser for this file. The specifications for this parser were quite simple, yet representing a challenge since adding flexibility added more characters to handle for the parser. We then set a list of allowed characters at given places.

- Spaces between commas.
- Trailing newlines after each line.
- Comments with a given character ('#' in our case).

With all these added difficulties, the parser could be implemented. The implemented parser uses buffered reader with predefined length buffer and *fgets* method to parse each line properly and efficiently.

### 3.4 Loading Data

To avoid having to hardcode the training and test data which were quite redundant for the *xor* and that would have been literally impossible for the hand written numbers, we decided to write a data loader. For the data, we knew both the format of the data, and the length of each line. We decided to implement a *CSV* loader. Each line contained comma-separated with first the index of the expected fired neuron (or -1 if none), and for each following values, an int from 0 to 255 representing the intensity at which the input neuron should be fired. This loader presented many difficulties :

- The number of input data was unknown.
- There can be some metadata in the file (for CSV format for example).

In order to solve the first point, two implementations of the loader were made. The first one being the most efficient one only allocating memory once, but requiring to specify the number of lines max to be read. The second being the most flexible reading each line and reallocating the memory at each new parsable line. Although very costful for the system, it is by far the easiest way to use the parser for the end used. To solve the second point, the solution was to add a parameter specifying at which line should the parser start parsing the file. As for the neural network loader, we implemented a buffered reader, increasing the performance of the calls (and the code complexity), and reducing the number of syscalls to read a file. The loader was originally designed for a specific input set but is now function for any input set formatted as specified above, allowing us to dynamically train a neural network without changing the code at all.

### 3.5 Grid image to text-file

After having recovered the 81 images corresponding to the cells of the sudoku, we developed an algorithm allowing to pass each image in the network, to recover the figure and finally to place it at the good index in the grid. To do this each image is numbered from 0 to 80 indicating its position in the grid. At the end of the algorithm, we obtain a grid in a text file, where the X represent the digits, with the following shape :

```
... ..  
... X..  
... ..X.
```

```
.X. ...  
... .X. ...  
... ..
```

```
... ..X..  
.X. ...  
... ..
```

### 3.6 Sudoku solving algorithm

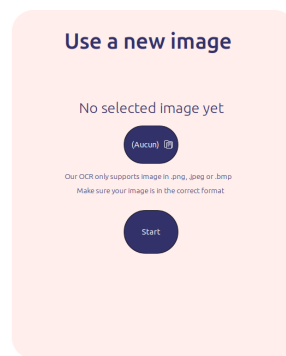
In order to solve the sudoku grid, we start by retrieving the text file containing the grid. We then retrieve each number and place it in a grid represented by a two dimensional array. We then apply a first verification to know if the initial grid is solvable or not. If it is, we use a backtracking algorithm that tests for each empty cell a new number between 1 and 9, if the number is valid, it moves to the next cell, otherwise it returns to the previous cell and changes the number.

Once the grid is solved, we write it in a text file with the extension .result . This file will then be reused to display the solved grid.

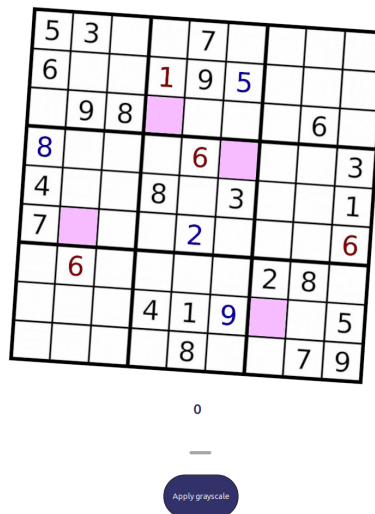
### 3.7 Graphical interface

In order to showcase some of our advancement, we decided to create a graphical interface. To create it we used the library GTK and the tool Glade. The interface is separated into several pages, each one showcasing a part of our work.

The first page allows a user to upload a Sudoku from its computer. Once the image is selected, it is copied in the project files so that it can be processed later on.



The second page shows our manual rotation. Using a slider, the user can rotate the image however he pleases but preferably so that it is upright to allow an easier line detection.





The third page shows our grayscaleing as it displays a gray-scaled version of the rotated image.

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

Image has been grayscaled

Turn into binary

The fourth page displays the black and white version of the rotated image.

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

Image has been turned into binary

Split the image

Finally, the fifth page displays the image cut down into individual squares.

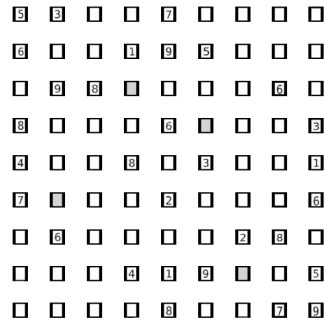


Image has been split



## 4 CLI

To gather all these *features* together, a proper CLI had to be implemented. We decided to implement a basic CLI with named parameters with subcommands. For example, if no parameters is specified, the UI is summoned. If a grid is passed as the only parameter, the grid is parsed and the solved by the solver. If any of *train* or *test* commands are specified, all following parameters are parsed by a CLI parser allowing to pass parameters to *network training function* of *network testing function*. As for the rest of the codebase, we wanted the CLI to be relatively usable for anyone.

Example:

```
./solver train --generate "784,30,10" --output-network "./output.net" --data "./mnist_train_data.csv" --format csv --start 1 --iterations 10 --output-activation softmax
```

Creates a new neural network with 784 inputs, 30 neurons in the hidden layer, and 10 outputs, saved in `./output.net` and trained 100 times on the data of `./mnist_train_data`, except the first line (which is usually the header in a csv file) and whose output layer activation function is the soft max function.

Alongwith this *train* command, the performances of a neural network can be tested using the *test* command. This commands tests the provided neural network with the provided data.

Example:

```
./solver test -input-network "./output.net" -data "./mnist_test_data.csv"  
-format csv -start 1
```

Tests the performances of the network contained in `./output.net` over all the lines of `./mnist_test_data.csv` except the first line.

## 5 Future expectations

### 5.1 Backpropagation

Concerning backpropagation, we will implement more performant activation and error functions, notably the cross-entropy functions and the leaky-relu function as activation for the hidden layers.

### 5.2 UI

The will for the final defense is to have a neural network visualizer and the ability to input any value to test the network by hand if needed. Furthermore, we would like to enhance the UI using proper CSS and link all parts of our project together properly in the same UI, thus implementing the CLI features in the UI also.