S3 Project - Letscallitaday - Final report

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

December 2022

Contents

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
- Creation of the UI

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 the UI
- Sudoku saver

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's 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's 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 check 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 along with some config file at the base of the repository. 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 bare distribution. 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.

Figure 1: Example of a page from our documentation

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 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.
Alongside the manual rotation, a more efficient automatic rotation was also implemented. (Cf. 3.1.6)

3.1.3 Manual cropping

To ease the recognition of the Sudoku, a manual cropping of the image has been implemented. This allows to crop the image around the Sudoku in order to remove any text or lines around it which may disturb the process.

3.1.4 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.

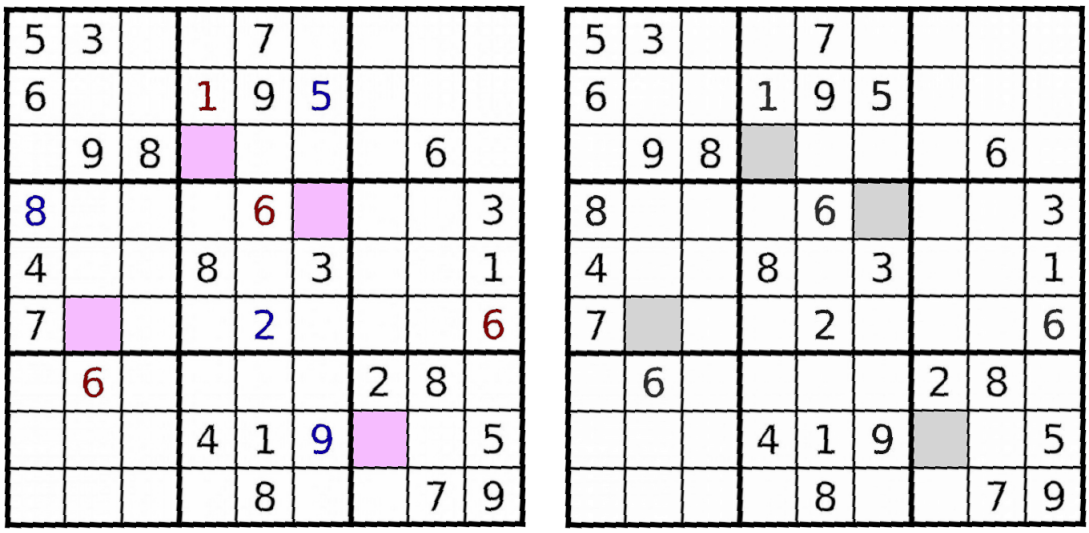


Figure 2: A gray-scaled Sudoku obtained using our program, compared to its colored counterpart

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.

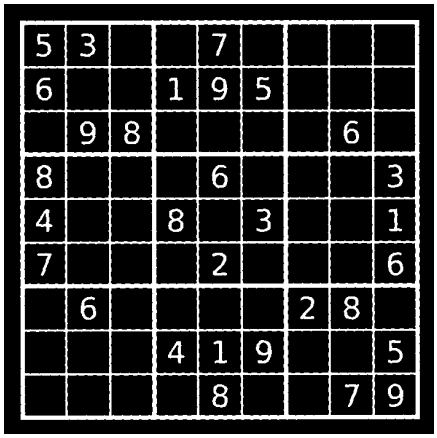


Figure 3: A black and white Sudoku obtained using our program

3.1.5 Gaussian blur filter

In order to reduce the impact of noise on the images, we implemented a Gaussian blur filter with a 5x5 Gaussian matrix.

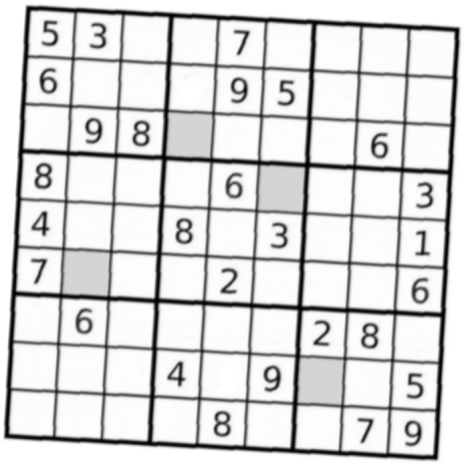


Figure 4: A gray-scaled and blurred Sudoku

3.1.6 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.

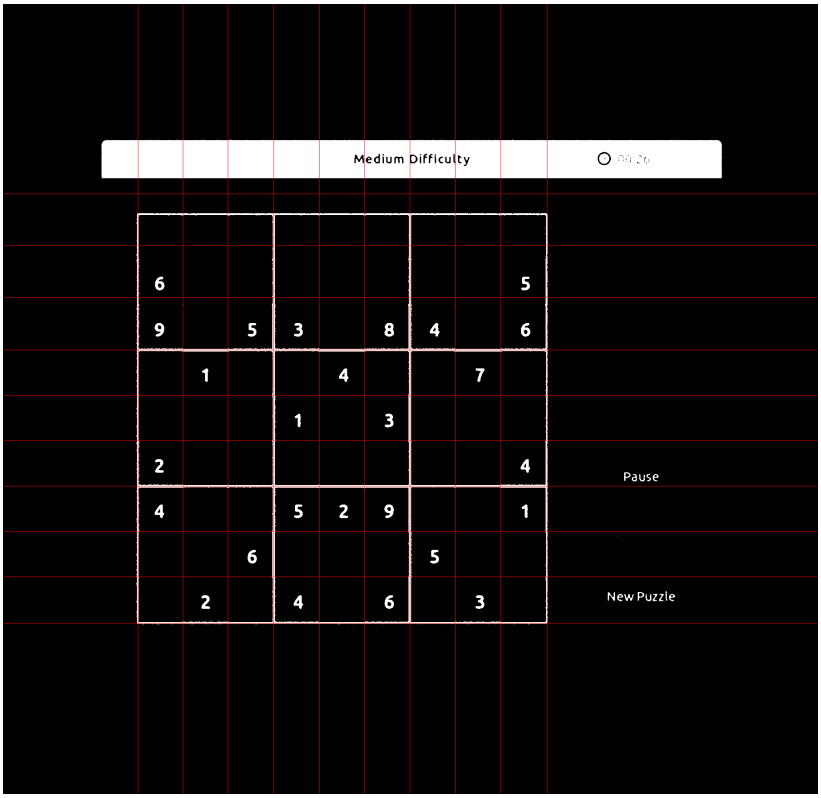


Figure 5: The lines found on the Sudoku picture after the automatic rotation

3.1.7 Automatic rotation

Once the lines are detected, we have to distinct cases:

- The first case is when the image is already well oriented. When that happens, we have nothing to do and we can continue to the next part of the algorithm

- The other case is when the image is not well oriented and has to be in order to continue the algorithm. In this case, we can find the average angle by which the image is rotated thanks to the lines we detected. In fact, even though the line detection works way better on a well oriented image, it can still detect enough lines to do this task as we can see in the Figure 4.

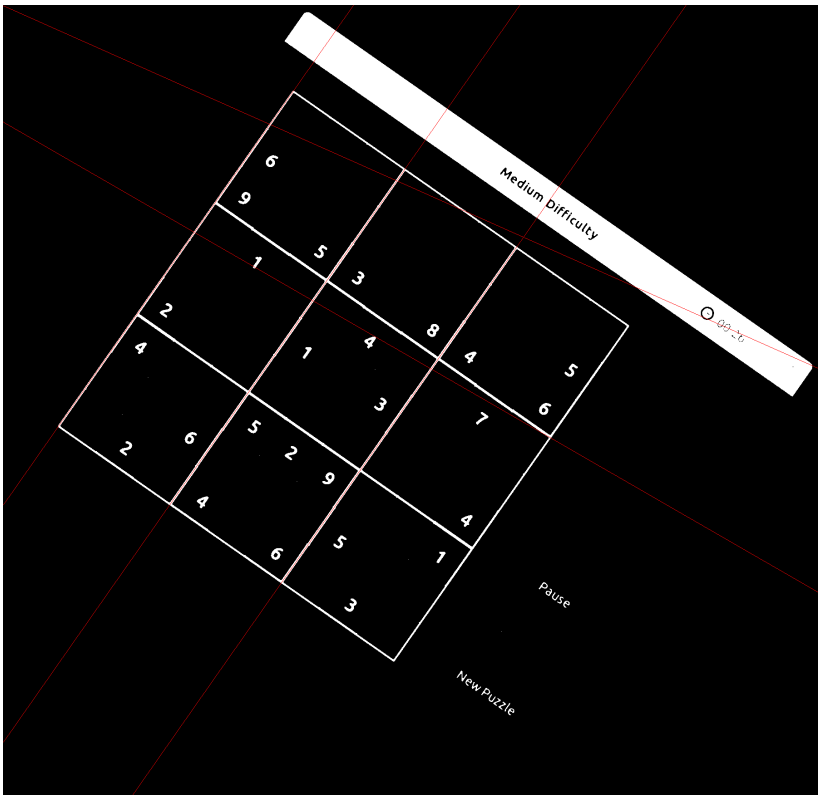


Figure 6: The lines found on the sudoku picture before the automatic rotation

3.1.8 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.

3.1.9 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)

3.1.10 Handling incomplete and weird 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.
As for the weird images, with for example some elements around the Sudoku which make the line detection difficult, we added some extra layers of verification and filtering to check which detected lines were the most likely to be the lines of the actual Sudoku.
Thanks to that, we can filter the lines that are found too far away from (or too close to) the other detected lines, the ones that have an angle which doesn't match the angle of the others, etc.

3.1.11 Bonus: implementation for hexadecimal and other types of grids

After finishing the line detection for the 9x9 grids, it was not really hard to adapt our algorithm since we implemented it well from the start. Therefore, we modified briefly our algorithm to be able to handle 9x9, 16x16, 25x25, and even more types of grids

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 Used dataset

3.3.1 Importance of a good data set

A good data set is important for machine learning because the quality and quantity of the data that a model is trained on can greatly affect its performance. A data set that is too small or has too much noise or bias can make it difficult for a model to learn effectively. On the other hand, a data set that is large and clean can allow a model to learn more accurately and make more accurate predictions on new data. Ultimately, the quality of a data set can make the difference between a machine learning model that is successful and one that is not.

In order for a data set to be effective for training a machine learning model, it should be representative of the real-world data that the model will be applied to. This means that the data set should include a diverse range of examples, including various types of input data and corresponding output labels. For example, if you are training a model to recognize different types of objects in images, your data set should include images of many different objects in various poses and lighting conditions.

Additionally, a good data set should be free from errors and anomalies, as these can confuse a model and prevent it from learning effectively. For example, a data set that contains incorrectly labeled examples or examples that do not follow the expected format can make it difficult for a model to learn the underlying patterns in the data.

Finally, a good data set should be balanced, meaning that it should include a similar number of examples for each class or category that the model is being trained to predict. For example, if you are training a model to classify images as either containing a cat or a dog, your data set should include a roughly equal number of images of cats and dogs. This can help to prevent a model from biasing towards one class or another.

Overall, the quality of a data set is an important consideration when developing a machine learning model. A good data set can help to ensure that the model is able to learn effectively and make accurate predictions on new data.

3.4 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 gradiant descent algorithm. As for the feedforward, each layer can be backpropageted 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.

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 Enhancing performances

In order to enhance the performances of the neural network, we had to tweak the activation functions. We decided to use the *cross-entropy* cost function. Furthermore, having the same activation function for all the layers could be a huge obstacle in the learning process. Therefore, we changed the activation processus, so that each layer has its own activation function. We then used the *leaky-relu* activation function for the hidden layers, and the *softmax* activation function for the outer layer.

Since the performances where strongly improved by these changes, we faced the problem of exploding gradients. Exploding gradients is a problem happening when the backpropagation leads to numbers that are way to big to be expressed by *64-bits* machines doubles. Therefore, the value becomes *NaN (not a number)*, which breaks the whole learning. After some research, it appeared that in order to reduce this, we had to properly set the weights of the neurons at the creation of the network, so that it could find faster the local minimum. We used *Marsaglia polar* algorithm to generate randomly distributed weights for each neurons, using the *Xavier-distribution*, which is distribution method applied to neural networks based on the number of neurons at each layer

Exploding gradients is a problem that can arise in deep learning models, which are a type of machine learning model that are composed of many layers of interconnected nodes. This problem occurs when the gradients, which are the partial derivatives of the loss function with respect to the model parameters, become very large. This can happen when the weights in the model are very large, or when the gradients are calculated using the chain rule over many layers.

If the gradients become too large, they can cause the model parameters to become extremely large as well, leading to numerical instability and poor performance. In some cases, the gradients can even become infinite, which can cause the model to fail entirely.

There are several strategies that can be used to address the problem of exploding gradients. One common approach is to use a technique called gradient clipping, which involves limiting the maximum size of the gradients. This can prevent the gradients from becoming too large and causing instability in the model.

Another approach is to use a different activation function in the model. Activation functions are mathematical operations that are applied to the output of each node in a neural network, and they can have a significant impact on the gradients in the network. Some activation functions, such as the ReLU function, can help to prevent exploding gradients by limiting the range of values that the output of each node can take on.

Finally, it is also possible to use a different optimization algorithm when training the model. Some optimization algorithms, such as RMSProp and Adam, are specifically designed to help prevent exploding gradients. Using one of these algorithms can help to ensure that the model's weights remain stable and well-behaved during training.

3.5 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.

3.6 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. ... ...
... ...

3.7 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.

3.7.1 About backtracking...

One algorithm for solving a sudoku puzzle is called backtracking. This is a type of brute force search algorithm that tries every possible option until it finds the correct solution. Here's how it works:

1. Start by filling in the known numbers in the sudoku grid.

2. Choose a cell that is empty and try filling it in with a number between 1 and 9.

3. Check if the number you chose is valid in the current sudoku grid. This means checking if the number appears in the same row, column, or 3x3 block as any other number. If the number is valid, move on to the next empty cell. If not, go back to step 2 and try a different number.

4. Repeat step 3 until all cells are filled in or you reach a point where there are no valid numbers for a given cell. If you reach a point where there are no valid numbers, this means that the previous choices you made were not correct and you need to backtrack and try a different path.

5. If you are able to fill in all the cells without having to backtrack, you have found a valid solution to the sudoku puzzle.

This algorithm can solve any sudoku puzzle, but it may be slow for very difficult puzzles. There are other more efficient algorithms for solving sudoku puzzles, but the backtracking algorithm is one of the simplest to understand and implement.

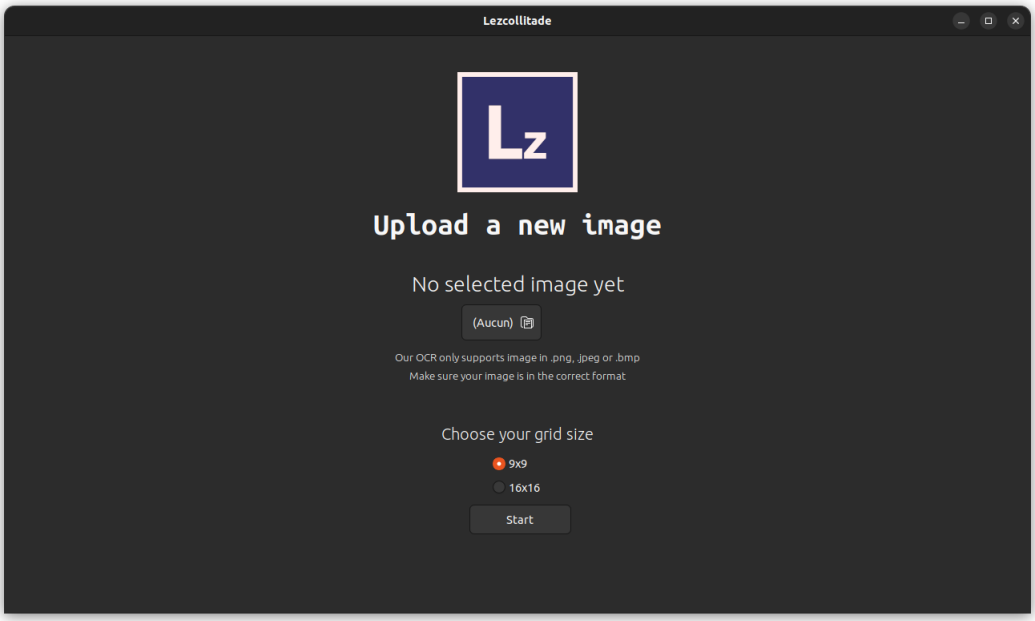
3.8 Graphical interface

In order to showcase what our project was able to do we have created a graphical interface. It was made using the library GTK3 and the tool Glade. The interface is separated into several pages, each one of them showcasing a part of our work.

3.8.1 The main page

When the program is first executed, the page which is displayed is the home page a.k.a. the upload page. It is the starting point of the program. On this page are displayed information such as the logo of the project and which image formats are accepted.

Choosing the image which will be processed is done on this page through the file chooser in the middle. This is also the page where the user selects whether the Sudoku chosen is in a 9x9 or a 16x16 format using the two accordingly named radio buttons. Finally the **Start** button at the bottom of the page starts the process and switches to the next frame if an image has been chosen.



3.8.2 The rotation page

The second page is the rotation page. It is the frame where all the necessary rotations are applied to the image in order to allow for a better recognition of the image.
On top of the page is displayed the image the user has selected with the current rotation applied to it. Below the image is a slider which allows the user to manually rotate the image as he pleases.
The **Continue** button on the bottom of the page takes into account the manual rotation the user input before applying an automatic rotation on the image and then switches to the next page. The **Continue with automatic rotation** button simply takes the initial image an applies an automatic rotation on the image before going to the next frame.
Finally the **Restart** button allows the user to go back to the main page.

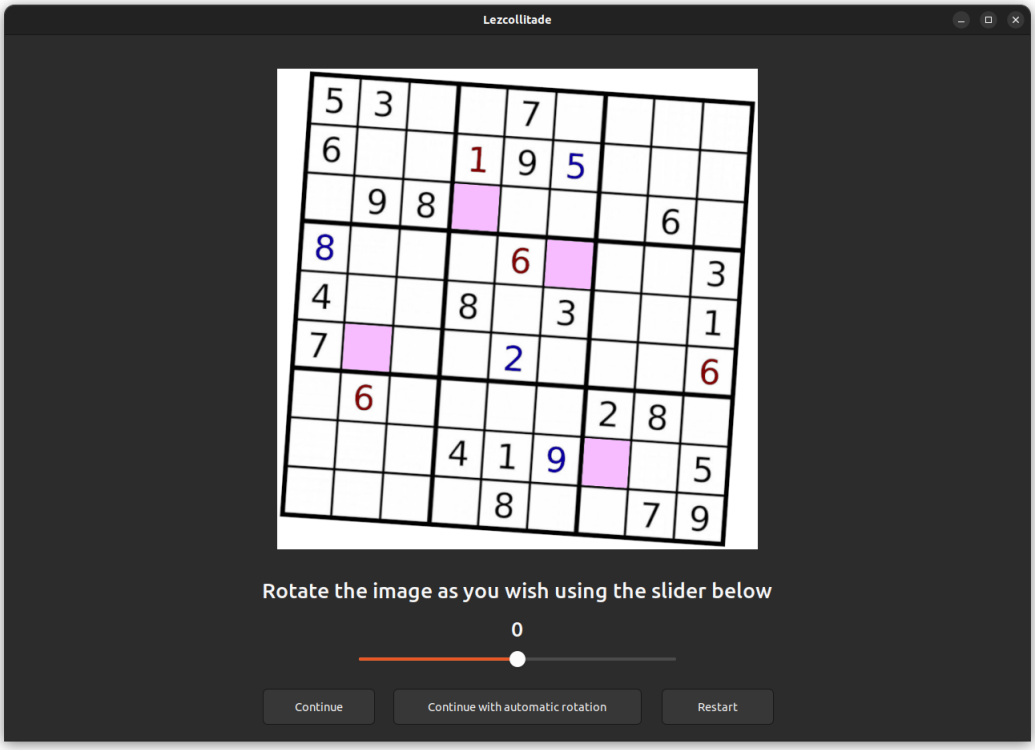
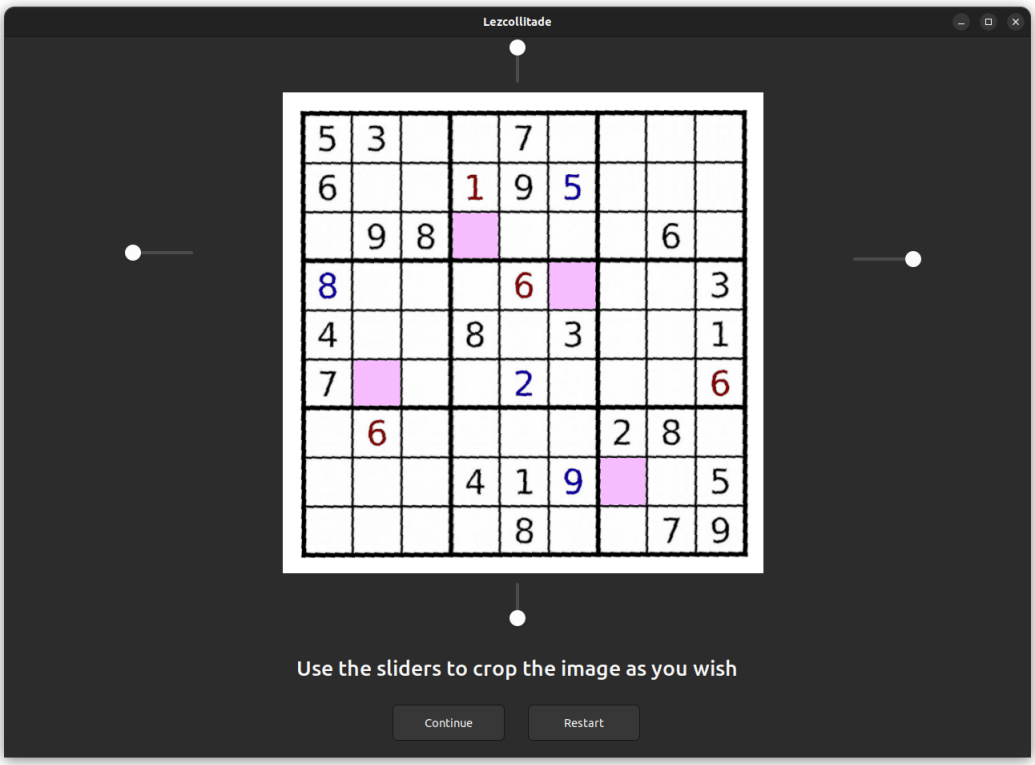


Figure 7: The rotation page with an image which might need a rotation

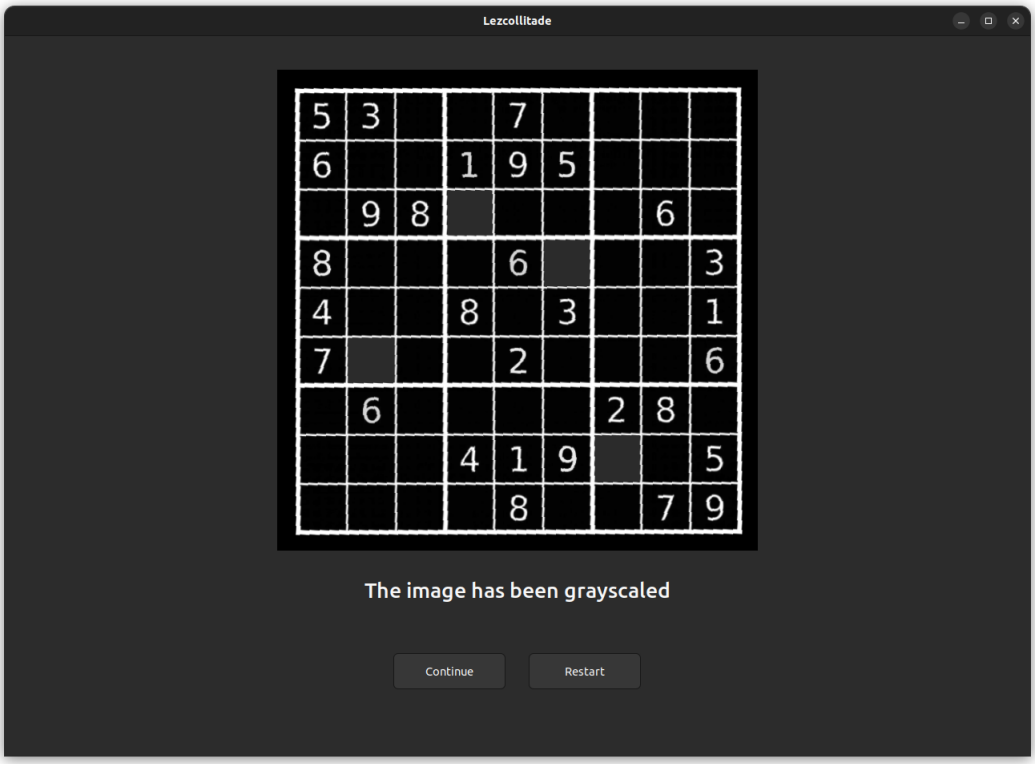
3.8.3 The cropping page

The third page is the cropping page. This page allows the user to manually crop the image in order to remove any text or lines surrounding the Sudoku which could cause the failure of the line detection.
In the middle on the screen is displayed the image which has been rotated in the previous frame.
Sliders are positioned on all four sides of the image and allow the user to crop the image as he pleases.
The **Continue** button on the bottom of the page applies the manual cropping which has been entered and switches to the next page.
Finally the **Restart** button allows the user to go back to the main page.



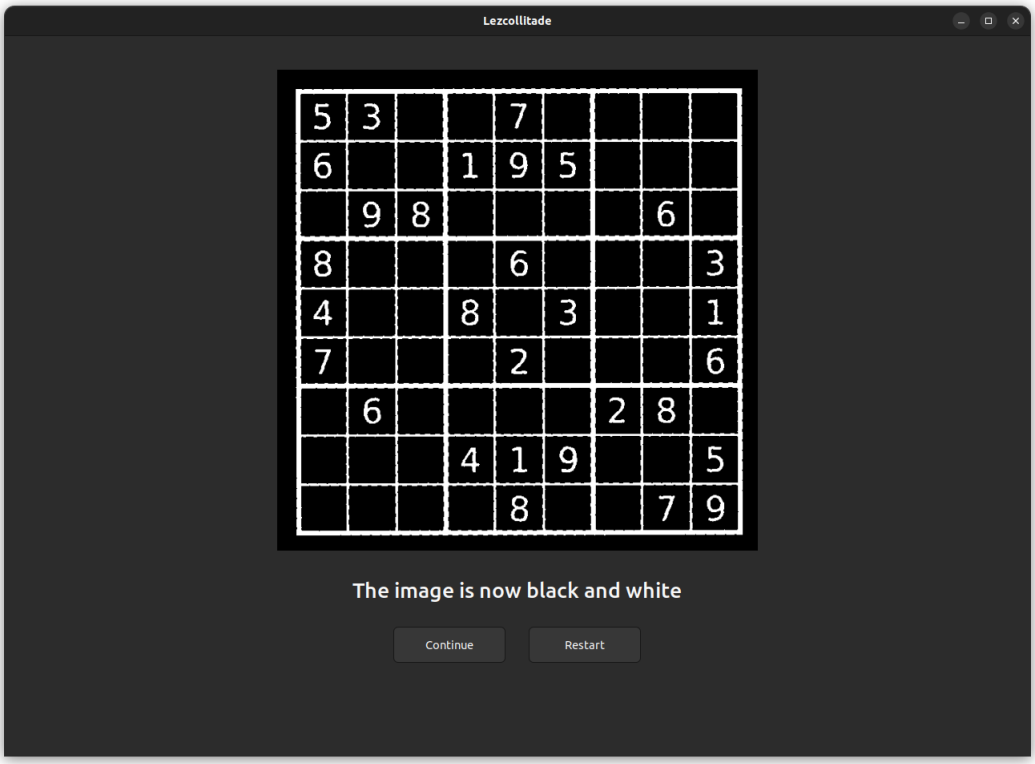
3.8.4 The gray-scaling page

The fourth page of the graphical interface is the gray-scaling page. It is a straight-forward page which displays the image selected by the user but with an inverted gray-scaling filter applied to it.
The **Continue** button on the bottom of the page switches to the next page. Finally the **Restart** button allows the user to go back to the main page.



3.8.5 The binarization page

The fifth page of the graphical interface is the binarization page. It is a straight-forward page which displays the image selected by the user but in black and white.
The **Continue** button on the bottom of the page switches to the next page. Finally the **Restart** button allows the user to go back to the main page.



3.8.6 The splitting page

The sixth page of the graphical interface is the splitting page. If the splitting succeeded, the frame shows all the cells of the Sudoku separated in the middle of the screen. If the splitting failed, nothing is displayed except a message informing the user that the splitting failed.
The **Continue** button on the bottom of the page switches to the next page. Finally the **Restart** button allows the user to go back to the main page.

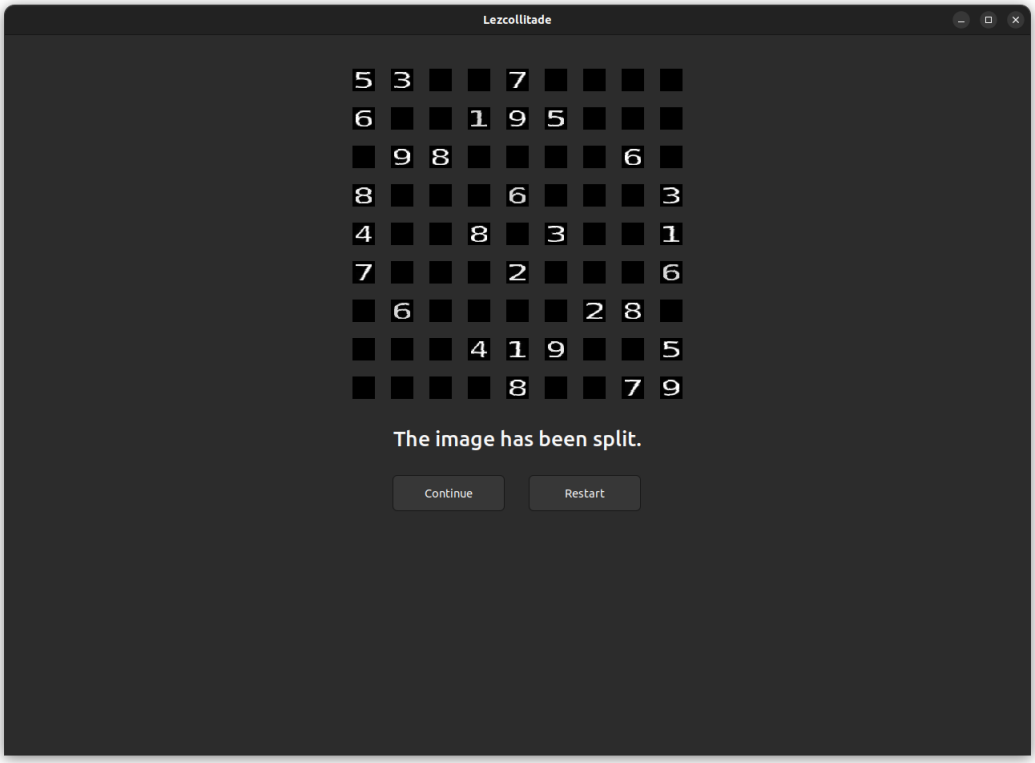


Figure 8: A successful split image

3.8.7 The results page

The seventh page of the graphical interface is the results page. It is the frame which shows the results of the neural network and allows the user to correct any mistakes.
On the right is an automatically rotated version of the initial image which can be used as reference by the user. On the left are the results of our neural network applied on the Sudoku cells. Those are placed in entries which can be modified by the user in order to add missing numbers or amend any inaccuracy.
The **Validate** button on the bottom of the page allows the user to validate the results of the neural network or their correction. If the Sudoku is not solvable then a popup appears to inform the user that an issue appeared and lets him correct the results.
Finally the **Restart** button allows the user to go back to the main page.

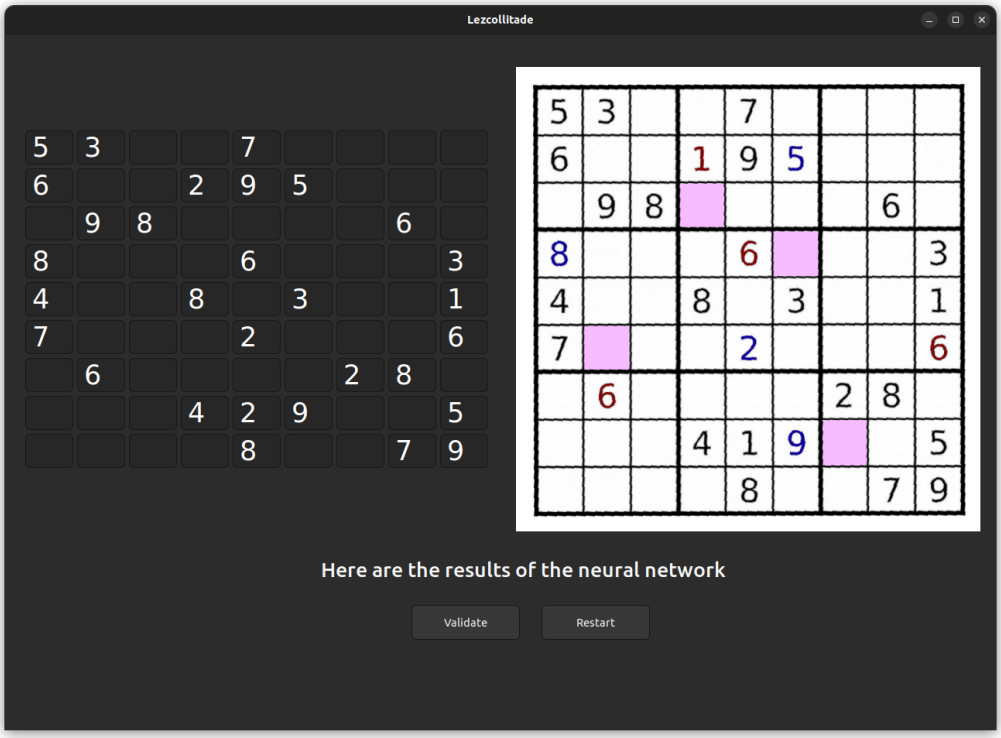


Figure 9: A successful split image

3.8.8 The final page

The eighth page of the graphical interface is the final page. When the results of the neural network are validated, the solver then solves the Sudoku and an image of the solved Sudoku is shown on the screen. On this image the original numbers are written in black and the new numbers in green. The displayed image can be found at the root of the project, saved as **solved.png**.
The **Quit** button on the bottom of the page closes the graphical interface. Finally the **Restart** button allows the user to go back to the main page.

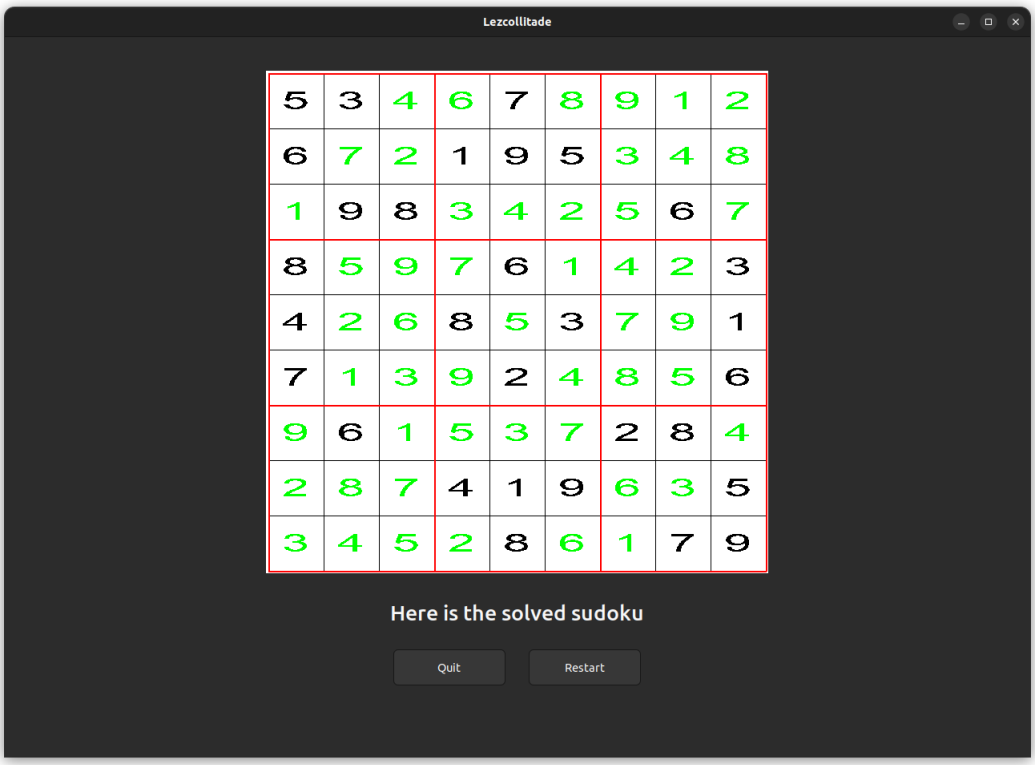


Figure 10: A successful split image

4 CLI

4.1 About the CLI

To gather some of these *features* together for testing, 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* of *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 ususally the header in a csv file) and whose output layer activation function is the soft max function.

Along with 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.

4.2 Sub-commands

Train

- Loading network.

  `--generate <string>`

Generates a neural network based on string. The string should be in the form of "„,...,".
Example: ***`Lezcollitade train --generate "2,2,1"`*** generates a network with 2 inputs, 1 hidden layer of 2 neurons, and 1 output neuron.

`--input-network|-i <file>`

Loads a network from a file. If the file doesn't exist, an error will be raised.
NB: Either ***`--generate`*** *or* ***`--input-network`*** *are required.*

`--output-activation <sigmoid|softmax> (default: sigmoid)`

The default activation function for the output layer.

`--activation <sigmoid|softmax> (default: sigmoid)`

The default activation function for the hidden layers.
NB: The activation functions are saved within the network, thus only valid when ***`--generate`*** *is used. Otherwise, the functions will be loaded from the network file.*

- Output.

  `--output-network|-o <file>`

Once the training is done, outputs the network to given file, and creates it if the file does not exist.

- Data.

  `--data <file> (required)`

Loads data from a file based on the format. If the file does not exist, an error is raised.

  `--format <csv|bin> (required)`

The format of the data.

The `csv` format should be written as each line containing an entry, and containing as first value the index of the neuron expected to be fired (-1 if none), and the value of each input from 0 to 255 included.

Example: xor training data

```
-1,0,0
-1,1,1
0,1,0
0,0,1
```

The bin format is as of now not functional...

  `--length <int> (default: -1)`

The number of line to be read in the file. If not specified (or -1), all the file up to the end will be loaded.

  `--start <int> (default: 0)`

The number of lines the loader should start at. If none specified, lines will be loaded from line 0.

- Training parameters.

  `--iterations <int>`

The number of iterations to run the data set on the network. If none specified, the training will go on until `sigstop` is raised.

`--rate <float> (default: 0.1)`

The learning rate. If none specified, the default learning rate will apply. *Warning: Using an oversized learning rate can cause double overflow thus causing the neural network to contain* ***`nan`*** *values.*

> *Example:* `Lezcollitade train --generate "784,30,10" --output-network "./output.net" --data "./mnist_train_data.csv" --format csv --rate 0.01 --start 1 --output-activation softmax --iterations 100`

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 ususally the header in a csv file) and whose output layer activaiton function is the `soft max` function.

> *Example:* `Lezcollitade train --input-network "./output.net" --output-network "./output.net" --data "./mnist_train_data.csv" --format csv --start 1 --iterations 10`

Loads the neural network in "./output.net", trains it 10 times over all the lines of `./mnist_train_data.csv` except the first line, and saves it back to "./output.net".

Test

`--generate <string>`

`--input-network|-i <file>`

See Train

`--data <file>`

`--format <csv|bin>`

See Train

`--verbose|-v` Indicates whether each tested input should be displayed along with its result in the command line.

Example: `Lezcollitade 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.

Convert

`--input-network|-i <file>` The input network to test on.

`--grid <dir>` The folder to the images of the grid.

`--data|-d` The path to the single image to be tested.

`--mode <single|multi>` Whether this tests with the `grid` folder or the `data` image path.

5 Conclusion

5.1 Conclusion

To conclude, this project is the result of dozens of hours of teamwork. As a group, we learned countless things about image manipulation, artificial neural networks, graphical interfaces and workflow. Thanks to a good organisation we have managed to fulfill all of our goals and deliver a final product we can be proud of.