



EPITA RENNES
RAPPORT DE SOUTENANCE PROJET OCR

0x12R

0x12R
—
Sudoku Solver

0X12R TEAM

Paolo Wattebled
Xavier de Place

Arthur Guelennoc
Mathieu Pastre

10 novembre 2022

Table des matières

Introduction	3
Équipe	3
Arthur	3
Mathieu	3
Paolo	3
Xavier	4
Projet	4
Répartition des tâches	4
1 Traitement de l'image	5
1.1 Pré-traitements	6
1.1.1 Importation d'image	6
1.1.2 Redimensionnement d'image	6
1.1.3 Filtre de gris	7
1.1.4 Augmentation du contraste	8
1.1.5 Normalisation des éclairages	9
1.1.6 Flou	10
1.2 Binarisation et détection des contours	11
1.2.1 Canny Edge Detection	11
1.2.2 Gradient d'intensité	12
1.2.3 Suppression des non-maxima	13
1.2.4 Seuillage des contours	14
1.2.5 Otsu Threshold	15
1.2.6 Adaptative Threshold	17
1.3 Détection de la grille	19
1.3.1 Détection des coins	19
1.3.2 Rotation d'Image Manuelle	20
1.3.3 Transformation Homographique	20
1.4 Segmentation	21
2 Réseau de Neurones	22
2.1 XOR	22
2.2 Digit Recognition	24
2.2.1 L'architecture de notre modèle	26
2.2.2 Les bases de données - MNIST	27
2.2.3 Les bases de données - Numeric	30

3 Résolution du sudoku	32
3.1 Importation	32
3.2 Backtracking	32
4 Interface Graphique	33
Conclusion	36
Références	36

Introduction

Ce document est le second rapport de notre projet OCR. Nous allons vous détailler comment nous sommes arrivés à notre état actuel, et les problèmes rencontrés. Nous vous souhaitons une bonne lecture !

Équipe

Voici les membres de notre équipe, la TEAM 0x12R :

Arthur Guelennoc

Pour ce projet, je me suis chargé de mettre en place le réseau de neurones. J'ai toujours été intéressé par le domaine du *Deep Learning*, et j'avais eu l'occasion d'y mener quelques projets personnels. Cependant, je n'ai jamais réalisé un réseau de neurones sans l'aide des librairies extensives de **Python** avec **TensorFlow** ou **Keras**. Ainsi, ce fût une expérience originale qui m'as permis d'avoir une toute nouvelle perspective sur ce domaine d'intérêt. En utilisant le **C**, j'ai compris le fonctionnement des calculs qui me paraissaient assez abstraits lorsque j'utilisais des architectures pré-fabriquées sans me poser de questions.

Mathieu Pastre

<https://fr.overleaf.com/project/6368b34a4ee813695383d84f> Élève en SPÉ à l'EPITA, je devais m'occuper du traitement d'image. Mon objectif était de faire la rotation de l'image, manuellement et automatiquement, ainsi que de modifier sa taille. Lors de la deuxième soutenance, j'ai été chargé de l'affichage des nombres. J'ai été satisfait du travail que j'ai fourni lors de cette deuxième partie du projet. Ce projet m'a beaucoup appris.

Paolo Wattebled

Élève en SPÉ à l'EPITA, je me suis chargé au cours de ce projet du traitement d'image ainsi que la détection de la grille et sa segmentation. Ce projet, m'a apporté énormément de compétences et connaissances en programmation C. Malgré la difficulté de prise en main d'un tel langage, je me suis accommodé et appris à l'apprécier malgré son exigence.

Xavier de Place

Je suis en SPÉ R1 à EPITA Rennes cette année. Pour ce projet, je me suis occupé de l'interface graphique et j'étais en support pour le traitement d'image et pour la rédaction de ce rapport. Je n'avais jamais fait de C avant ce projet. Je suis content d'avoir pu apprendre ce langage. Il est bien différent de Python, mais je trouve que c'est un langage très puissant. J'ai appris beaucoup de choses sur les interfaces, notamment ce qu'est GTK et l'utilisation du CSS même en dehors des pages web.

Projet

Le concept de notre projet est simple. Nous créons un OCR, un *Optical Character Recognition*. C'est un programme qui prend une image en entrée, et qui en extrait le texte. Nous le faisons spécialement pour les grilles de Sudoku, il contient donc un *solveur* de Sudoku.

Répartition des tâches

Voici un tableau récapitulatif de la répartition des tâches au sein de l'équipe.

	Arthur	Mathieu	Paolo	Xavier
Traitement de l'image				
Chargement de l'image				R
Suppression des couleurs			R	
Prétraitement de l'image		R	R	
Détection de la grille			R	
Réseau de Neurones				
XOR	R			
Réseau de neurones	R			
Autres				
Interface Graphique				R
Résolveur			R	

1 Traitement de l'image

Le traitement d'image est un axe de travail majeur dans un projet de reconnaissance de caractères. Il fallait donc mettre en place des algorithmes efficaces et optimisés dans le but de détecter la grille du Sudoku ainsi que les chiffres à l'intérieur. Nous vous présenterons ce que nous avons réalisé pour cette première soutenance.

Pour illustrer les différentes étapes, l'avancement du traitement sur cette image.

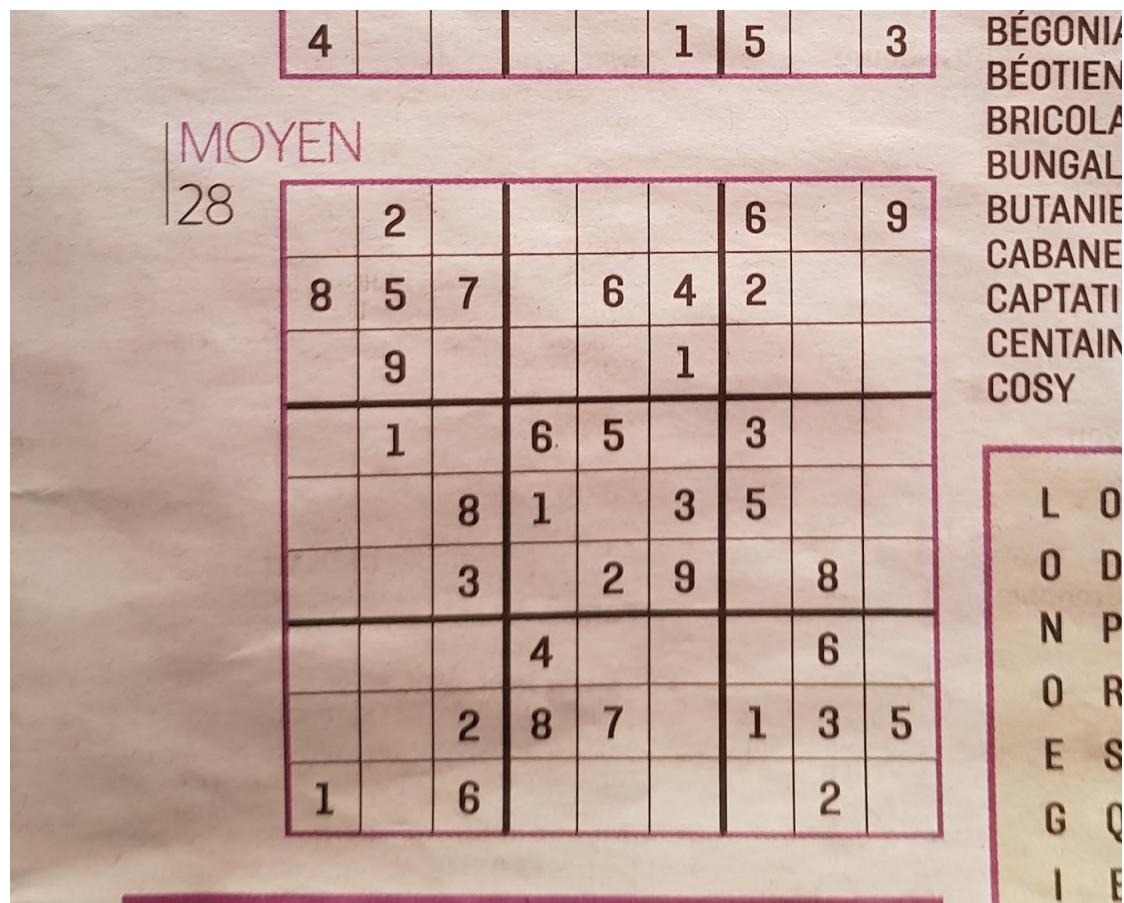


FIGURE 1 – Image originale

1.1 Pré-traitements

1.1.1 Importation d'image

Lors de notre réflexion sur l'architecture globale du projet, nous avons décidé d'utiliser nos propres structures d'images au lieu d'utiliser la librairie disponible **SDL2**. Nous avons fait ce choix parce qu'utiliser **SDL2** demande de créer en amont une architecture très peu modulable. De plus, **SDL2** est beaucoup moins optimisée et prend plus de temps à exécuter des opérations sur des images. Donc pour ce qui est du traitement d'image, nous utilisons notre propre structure. Mais pour importer/exporter des images, nous utilisons **SDL2**. Voici la structure que nous utilisons :

```

1      typedef struct Pixel {
2          unsigned int r, g, b;
3      } Pixel;
4
5      typedef struct Image {
6          unsigned int width;
7          unsigned int height;
8          Pixel **pixels;
9          char *path;
10     } Image
11

```

1.1.2 Redimensionnement d'image

Après l'importation et la création de notre structure d'image, nous réduisons les dimensions pour devoir traiter moins de données tout en ayant les mêmes résultats qu'avec la taille de base. Pour ce faire, nous utilisons l'*Interpolation du plus proche voisin*.



FIGURE 2 – Image Redimensionnée

Pour la suite des illustrations, nous allons garder la version non redimensionnée de l'image, pour mieux voir les changements.

1.1.3 Filtre de gris

Nous réalisons un filtre de gris sur chaque pixel de l'image pour réaliser les algorithmes de traitement d'image. Cela nous permet de normaliser les couleurs, et de nous enlever un paramètre compliqué à gérer. La formule appliquée est :

$$p_{(x,y)} = p_{(x,y)}.r * 0.3 + p_{(x,y)}.g * 0.59 + p_{(x,y)}.b * 0.11$$

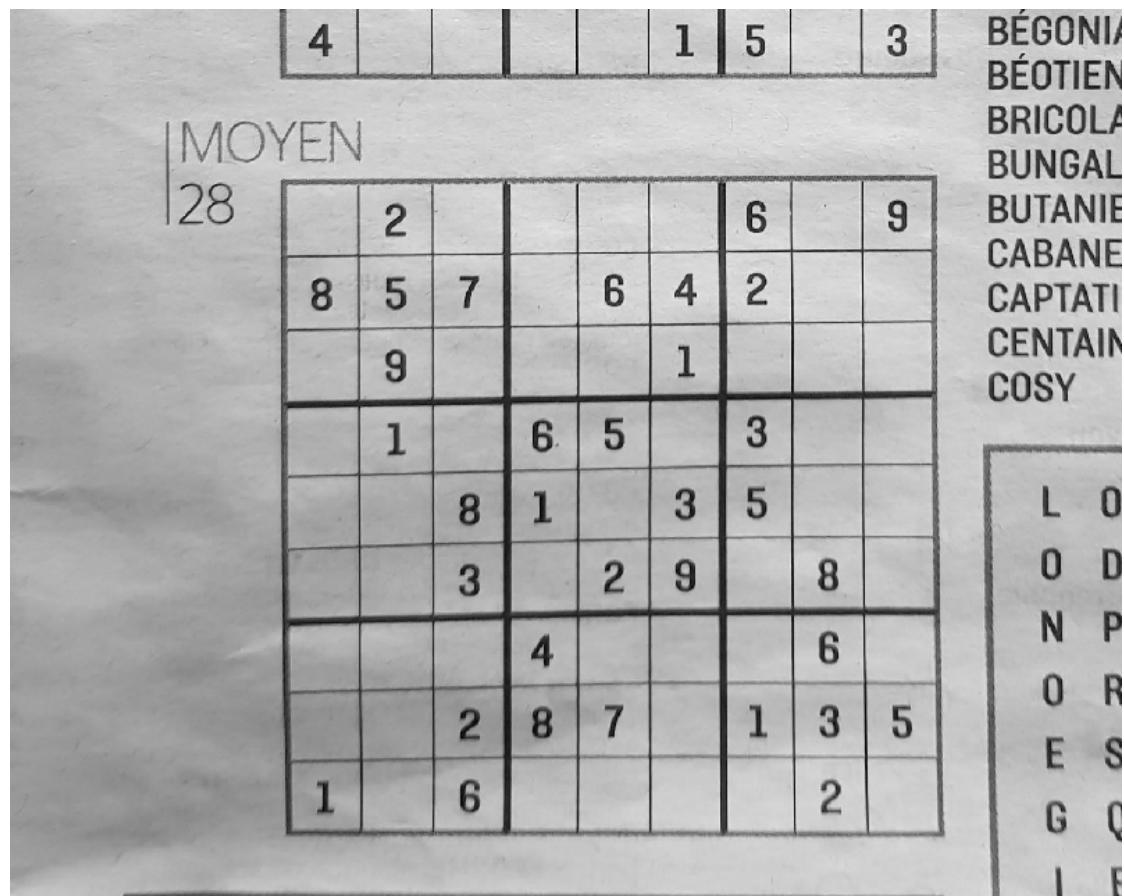


FIGURE 3 – Filtre de gris

1.1.4 Augmentation du contraste

Après avoir réalisé le filtre de gris et que toutes les valeurs R,G,B d'un pixel sont les mêmes, nous augmentons le contraste de notre image pour réduire certains bruits parasites. La logique de l'algorithme pour chaque pixel de notre image est la suivante :

$$\begin{aligned} i = 0 \rightarrow F \\ \text{si } p_{(x,y)} \in [i * (255/F), (i + 1) * (255/F)] \\ p_{(x,y)} = (i + 1) * (255/F) \end{aligned}$$

Ici, F est le facteur qui permet de faire un contraste plus ou moins élevé. Pour notre projet, il est mis à la valeur de $F = 12$.

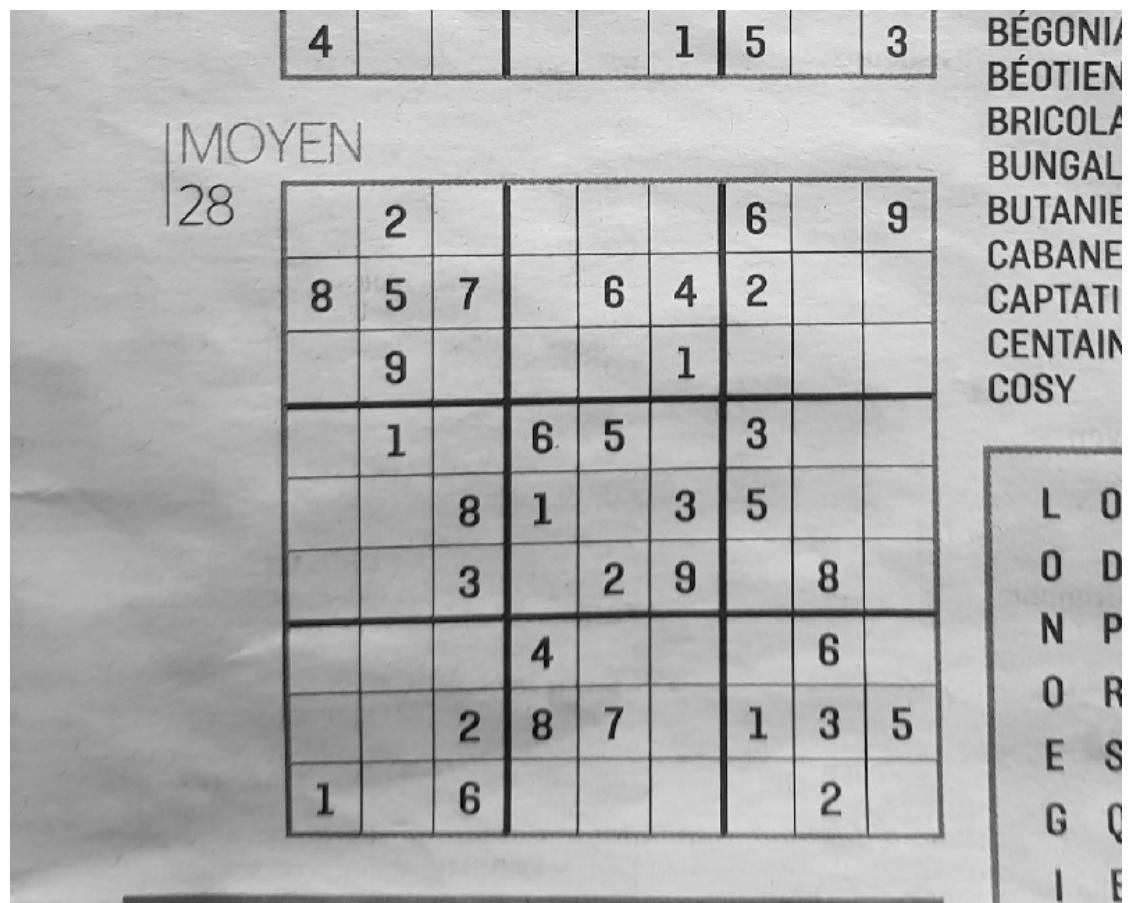


FIGURE 4 – Augmentation du contraste

1.1.5 Normalisation des éclairages

Cette technique nous a permis de réduire encore davantage certains bruits sur notre image. Ici, m est le pixel qui a la plus grande valeur.

$$p_{(x,y)} = 255 - p_{(x,y)}.r * (255/m)$$

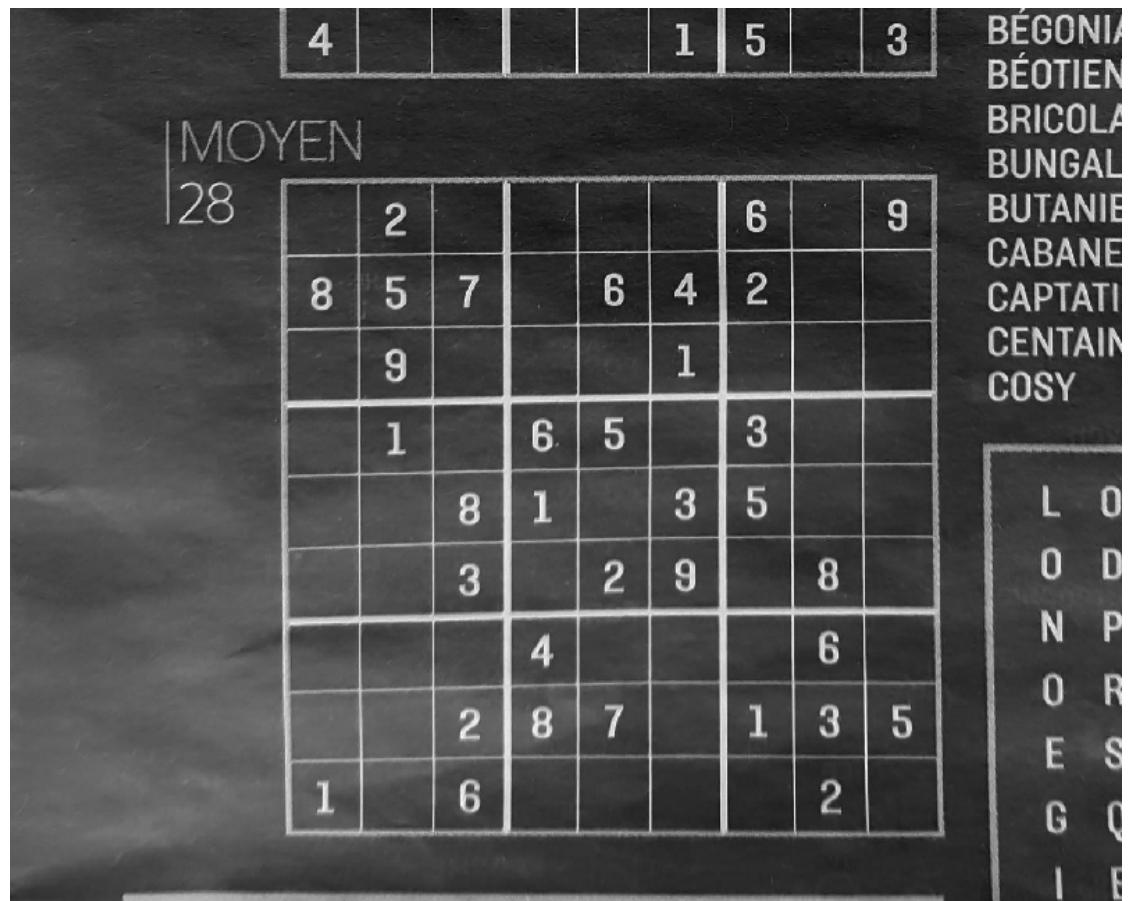


FIGURE 5 – Ajustement des éclairages

W

1.1.6 Flou

Finalement, lors du pré-traitement d'image, nous effectuons un flou Gaussien pour réduire les bruits et lisser l'image. L'intensité du flou peut être ajusté selon une variable `box radius br`. Définissons une fonction de convolution f . et un flou horizontal b_h tels que

$$b_h[i, j] = \sum_{x=j-br}^{j+br} f[i, x]/(2 \cdot br)$$

Maintenant, nous voulons calculer ce flou horizontal sur tout l'image. Nous faisons alors :

$$b_h[i, j], b_h[i, j + 1], b_h[i, j + 2], \dots$$

Mais les valeurs des voisins $b_h[i, j]$ et $b_h[i, j + 1]$ sont similaires. La seule différence réside dans une valeur la plus à gauche et une valeur la plus à droite. Donc

$$b_h[i, j + 1] = b_h[i, j] + f[i, j + r + 1] - f[i, j - r]$$

Cet algorithme de flou est bien plus performant qu'un simple filtre Gaussien réalisé avec un kernel de convolution. En effet, nous réduisons la complexité de l'algorithme de $O(n * r)$ à $6 * O(n)$ avec n le nombre de pixels dans l'image.



FIGURE 6 – Application du flou

1.2 Binarisation et détection des contours

Afin de détecter la grille, nous devons faire ressortir les lignes principales de notre image. Nous avons testé plusieurs algorithmes différents tels que *Canny Edge Detection*, *Threshold d’Otsu* et *Adaptative Threshold*

1.2.1 Canny Edge Detection

L’algorithme Canny se décompose en plusieurs étapes :

1. Flou Gaussien (réalisé précédemment)
2. Gradient d’intensité (Filtre de Sobel)
3. Direction des contours
4. Suppression des non-maxima
5. Seuillage des contours (Hystérésis)

1.2.2 Gradient d'intensité

L'opérateur utilise des matrices de convolution. La matrice de taille 3×3 subit une convolution avec l'image pour calculer des approximations des dérivées horizontale et verticale. Soit \mathbf{A} l'image source, \mathbf{G}_x et \mathbf{G}_y deux images qui en chaque point contiennent des approximations respectivement de la dérivée horizontale et verticale de chaque point. Ces images sont calculées comme suit :

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \mathbf{A} \quad \text{et} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \mathbf{A}$$

En chaque point, les approximations des gradients horizontaux et verticaux peuvent être combinées comme suit pour obtenir une approximation de la norme du gradient :

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

On peut également calculer la direction du gradient comme suit :

$$\Theta = \text{atan2}(\mathbf{G}_y, \mathbf{G}_x)$$

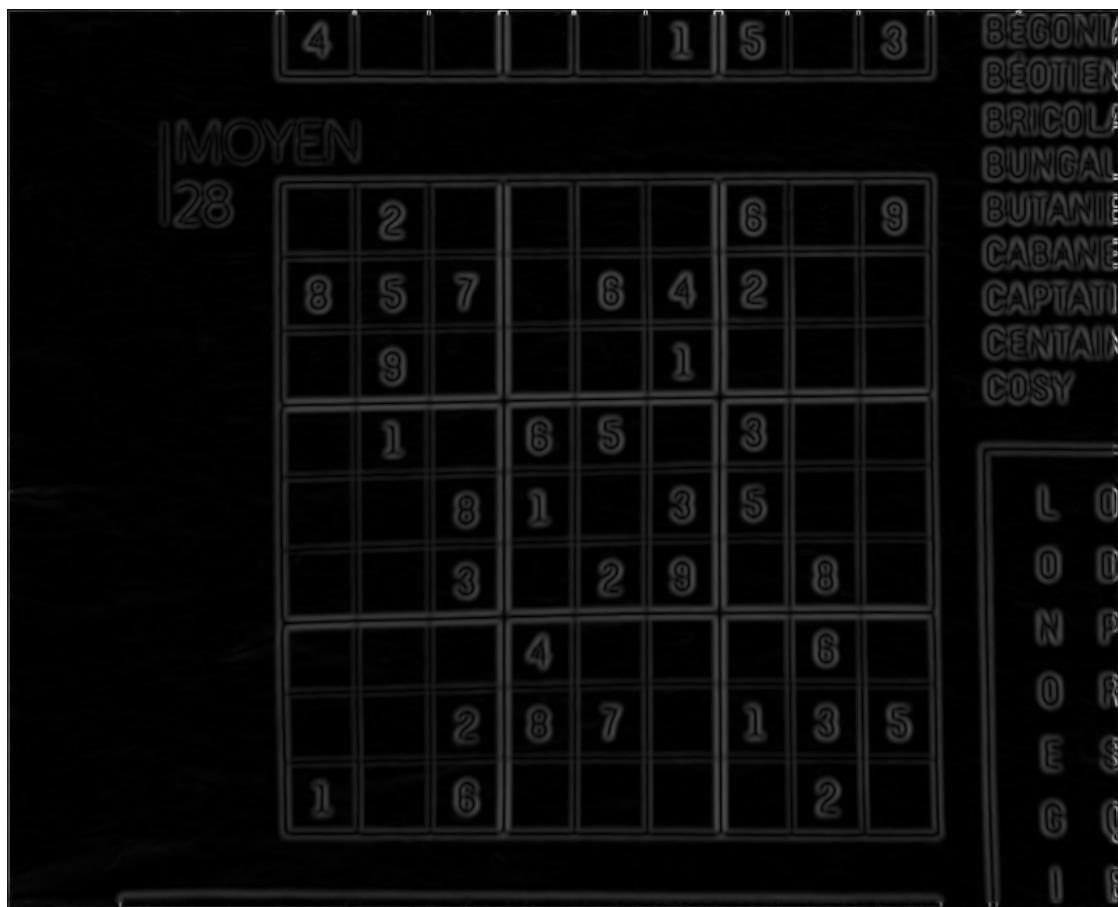


FIGURE 7 – Application du filtre Sobel

1.2.3 Suppression des non-maxima

Pour ce faire, considérons 2 points $P1$ et $P2$. En rapprochant les points $P1$ et $P2$ des pixels voisins par interpolation linéaire, il est possible de comparer les gradients. Si le gradient en (i, j) est supérieur aux gradients en $P1$ et $P2$, on le conserve. Sinon, on l'élimine. On procède ainsi pour tous les pixels bords de l'image, ainsi les bords ne seront large uniquement que d'un pixel.

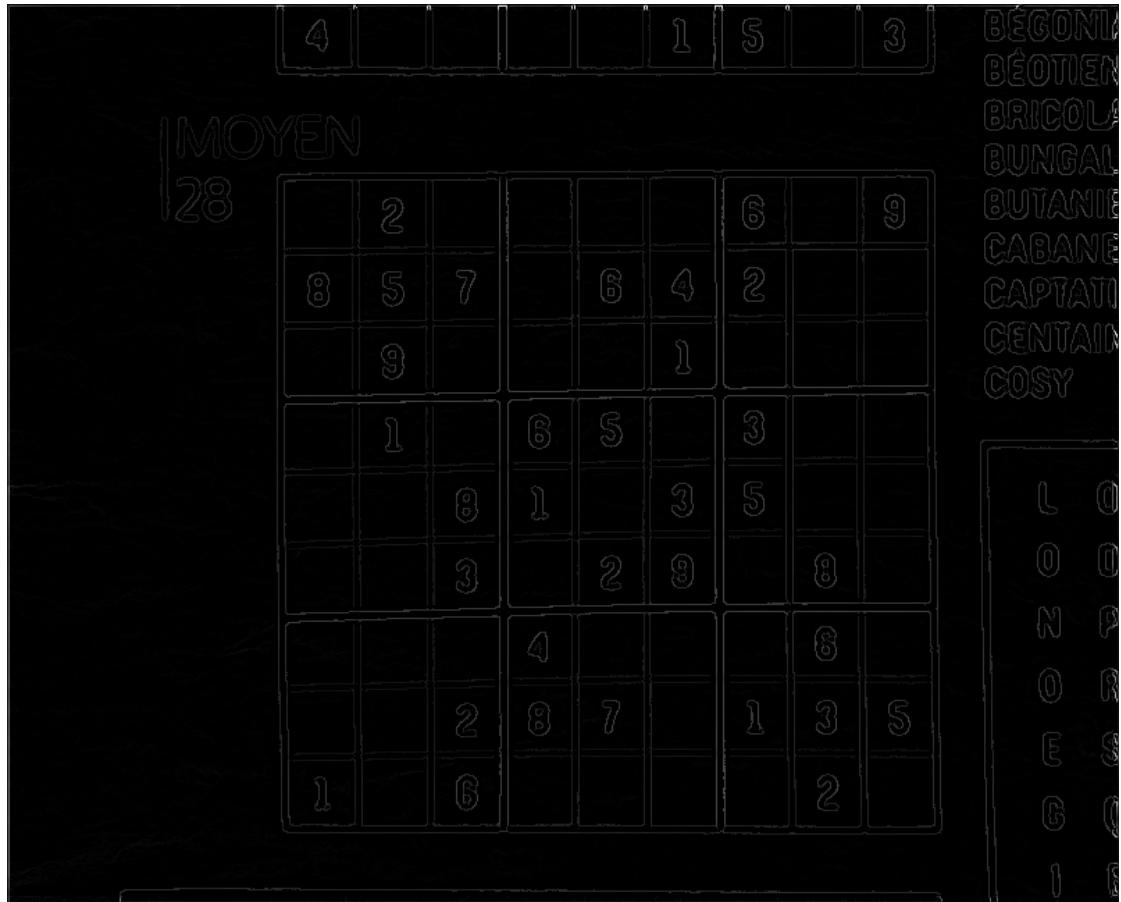


FIGURE 8 – Suppression des non-maxima

1.2.4 Seuillage des contours

La différenciation des contours sur l'image générée se fait par seuillage à hystérisis.

Cela nécessite deux seuils (un haut et un bas), qui seront comparés à l'intensité du gradient de chaque point. Le critère de décision est le suivant. Pour chaque point, si l'intensité de son gradient est :

1. Inférieur au seuil bas, le point est rejeté,
2. Supérieur au seuil haut, le point est accepté comme formant un contour,
3. Entre le seuil bas et le seuil haut, le point est accepté s'il est connecté à un point déjà accepté.

Une fois ceci réalisé, l'image obtenue est binaire avec d'une part les pixels appartenant aux contours et d'autre part, les autres.

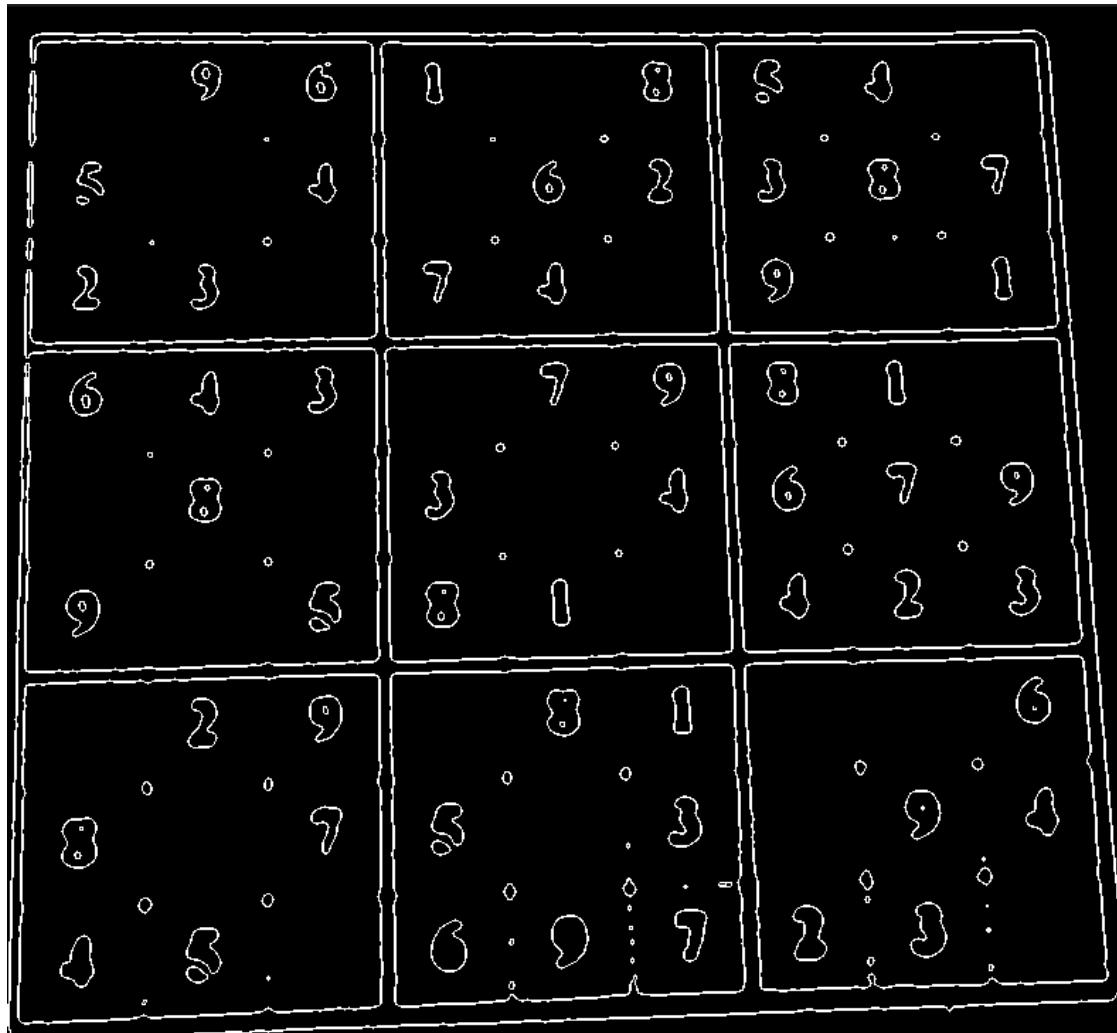


FIGURE 9 – Seuillage à hystérésis de l'image

Ainsi, Canny est un très bon algorithme pour faire ressortir les lignes de l'image mais n'est pas très adapté pour faire ressortir proprement les chiffres. De plus, pour la méthode de détection de grille, Canny n'est pas non plus adaptée étant donné que les lignes de la grilles ne sont pas continues.

1.2.5 Otsu Threshold

En vision par ordinateur et traitement d'image, la méthode d'Otsu est utilisée pour effectuer un seuillage automatique à partir de la forme de l'histogramme de l'image, ou la réduction d'une image à niveaux de gris en une image binaire. L'algorithme suppose alors que l'image à binariser ne contient que deux classes de

pixels, (c'est-à-dire le premier plan et l'arrière-plan) puis calcule le seuil optimal qui sépare ces deux classes afin que leur variance intra-classe soit minimale

Dans la méthode d'Otsu, le seuil qui minimise la variance intra-classe est recherché à partir de tous les seuillages possibles

$$\sigma_w^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t)$$

Les poids ω_i représentent la probabilité d'être dans la i ème classe, chacune étant séparée par un seuil t . Finalement, les σ_i^2 sont les variances de ces classes.

Otsu montre que minimiser la variance intra-classe revient à maximiser la variance inter-classe :

$$\sigma_b^2(t) = \sigma^2 - \sigma_w^2(t) = \omega_1(t)\omega_2(t) [\mu_1(t) - \mu_2(t)]^2$$

qui est exprimée en termes des probabilités de classe ω_i et des moyennes de classes μ_i qui à leur tour peuvent être mises à jour itérativement. Cette idée conduit à un algorithme efficace.

Voici les résultats de cet algorithme sur une image normale puis sur des images de sudoku



FIGURE 10 – Otsu threshold sur images sans bruit



FIGURE 11 – Utilisation d’Otsu sur grille de sudoku avec du bruit

Comme nous pouvons le voir, cette méthode à ses limites, ce qui fait que nous n'allons pas l'utiliser sur des images avec beaucoup de bruit. En effet, *Otsu* calcule un seuil global appliqué à l'image. Or, une image n'est pas forcément uniforme. Ce qui fait que son utilisation n'est pas adaptée pour notre utilisation.

1.2.6 Adaptative Threshold

L'Adaptative Threshold ou le seuillage adaptatif ne sélectionne pas un seuil global pour l'image mais en calcul un localement. Voici la formule appliquée à chaque pixel sachant que r est la taille de la zone traitée et c une constante pour normaliser le threshold :

$$i = 0 \rightarrow \text{Hauteur}$$

$$j = 0 \rightarrow \text{Largeur}$$

$$p[i, j] = (\sum_{k=-r}^r \sum_{l=-r}^r p[i+k, j+l]) / r^2 - c$$

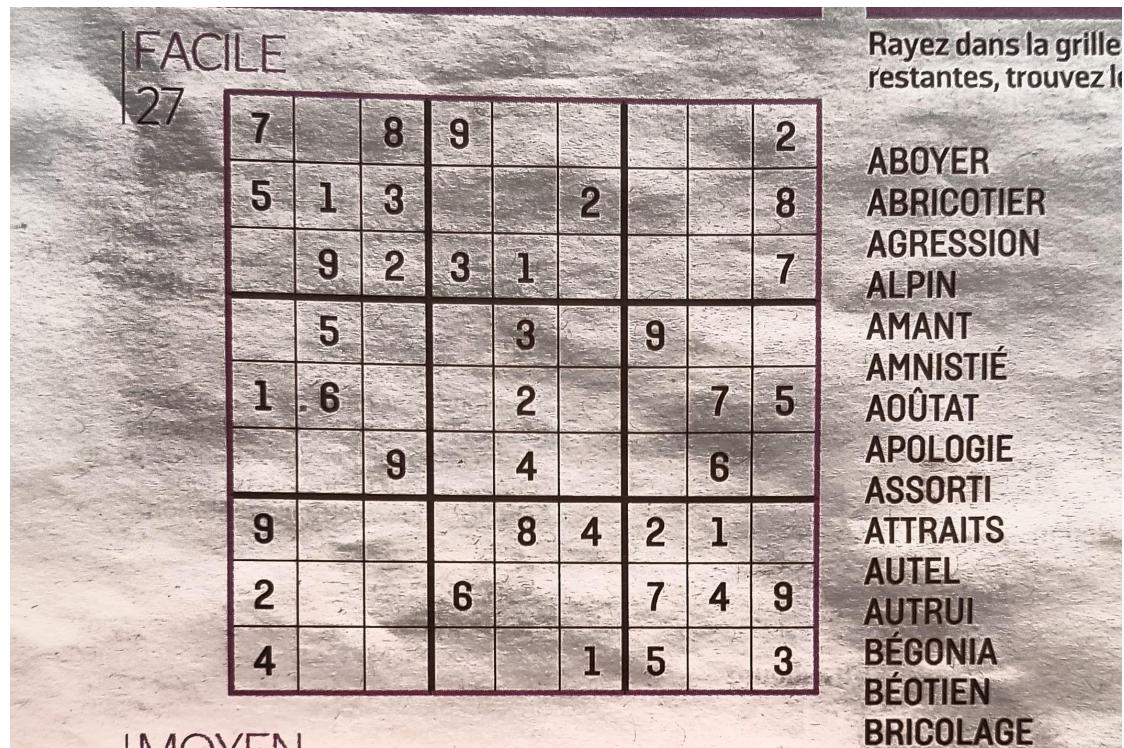


FIGURE 12 – Avant Adaptative Threshold

clearpage Voici le résultat obtenu :

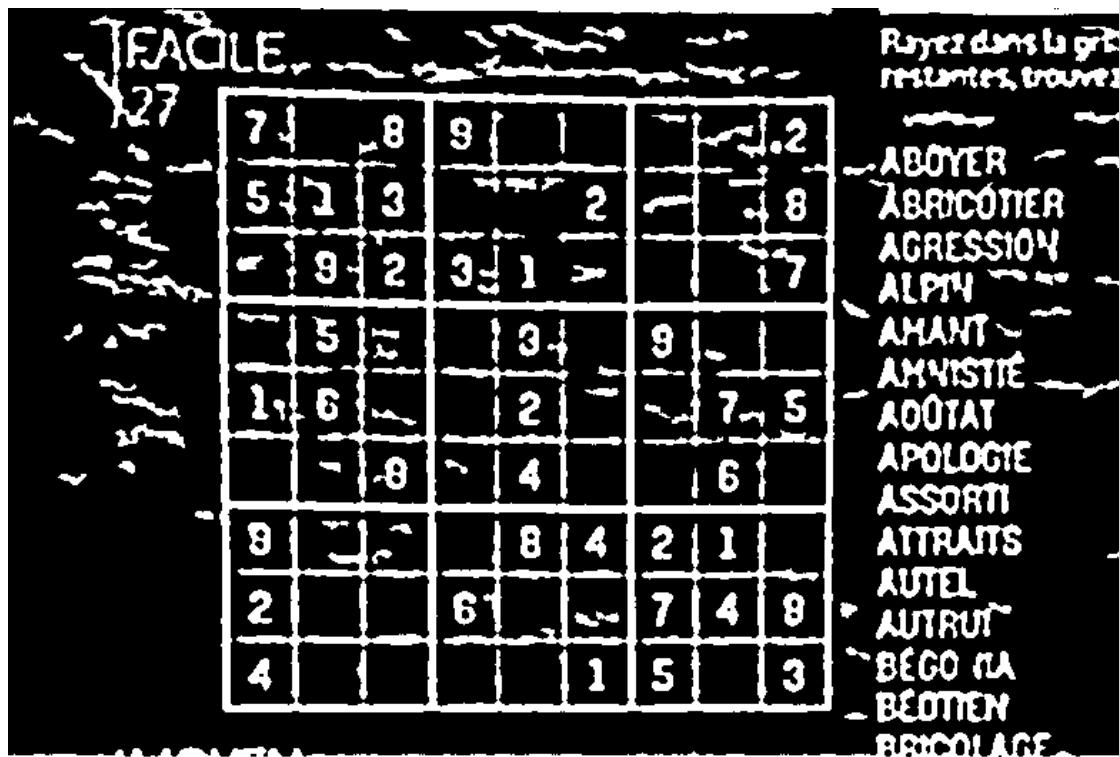


FIGURE 13 – Après Adaptative Threshold

Comme nous pouvons le voir, cette méthode se débrouille assez bien pour faire ressortir la grille et réduire les bruits. Nous allons donc garder cette méthode pour notre traitement d'image. De plus, les chiffres ressortent bien ce qui facilitera leur détection dans le réseau de neurones.

1.3 Détection de la grille

Pour réaliser la détection de la grille, nous avons essayé 2 méthodes : *Hough Transform* et *Blob detection*. D'un côté, *Hough transform* détecte les lignes sur une image. De l'autre, *Blob detection* crée et détecte le plus gros amas de pixels blancs sur l'image. Ainsi, de part nos résultats, nous avons opté pour la seconde option, étant donné que ses résultats sont plus fiables et plus faciles à manipuler. Voici en image ce que cela donne :

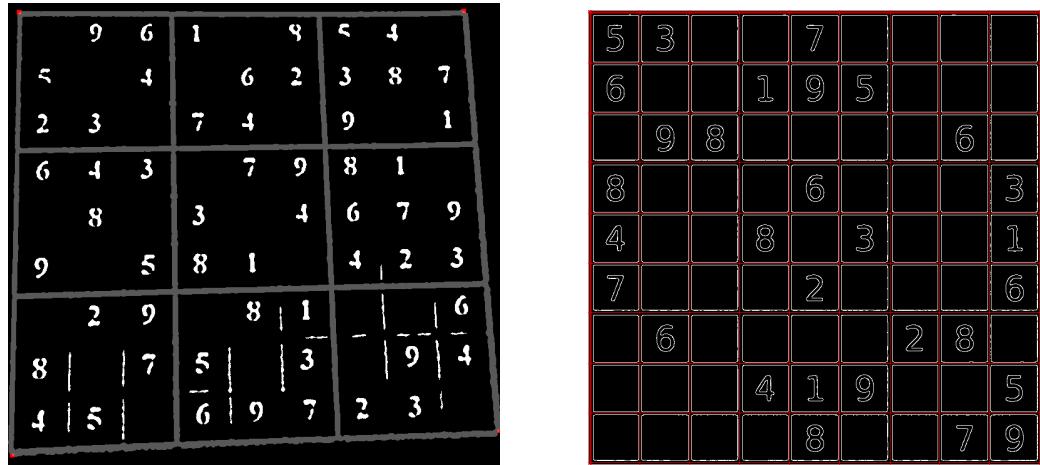


FIGURE 14 – Blob detection et Hough transform

Après avoir détecté le plus gros blob ou amas de pixel, il nous reste à détecter les coins. Grâce à ces derniers, nous allons pouvoir réaliser la transformation homographique, qui réalisera une rotation, correction de perspective et une redimension.

1.3.1 Détection des coins

Pour l'orientation automatique et la transformation homomorphe, il est nécessaire de détecter les coins de la grille. À noter qu'une détection exacte n'est pas nécessaire, une approximation est suffisante.

Nous allons utiliser la détection du plus grand blob pour détecter les coins. Il est possible de détecter les coins en utilisant la propriété suivante : Soit ul , ur , ll et lr respectivement les coins supérieur gauche, supérieur droit, inférieur gauche et inférieur droit de la grille.

- ul est le point (x, y) où la somme $x + y$ est la plus faible
- lr est le point (x, y) où la somme $x + y$ est la plus forte
- ll est le point (x, y) où la différence $x - y$ est la plus faible

- ur est le point (x, y) où la différence $x - y$ est la plus forte
- Il est possible de comprendre cette propriété en remarquant que :
- Le coin en haut à gauche aux coordonnées $(0, 0)$ a pour somme $0 + 0$
 - Le coin en bas à droite aux coordonnées (w, h) a la somme $w + h$ maximale
 - Le coin en bas à gauche aux coordonnées $(0, h)$ a la différence $0 - h$ la plus faible
 - Le coin en haut à droite aux coordonnées $(w, 0)$ a la différence $w - 0$ la plus forte

Tout les autres points ont des coordonnées entre ces extrêmes, et ces propriétés restent vraies (approximativement) au fur et à mesure que l'on se rapproche des 4 coins de la grille.

1.3.2 Rotation d'Image Manuelle

Pour appliquer correctement le réseau de neurones, il a fallu s'assurer que l'image soit bien orientée. Pour cela, nous avons fait un premier prototype de rotation. Celui-ci prenait comme valeur l'image que nous voulions traiter ainsi que l'angle à manipuler. Il a ensuite fallu lire les valeurs de chaque pixel de notre image source et faire une rotation centrale. Nous appliquons ensuite la formules suivantes pour trouver x' et y' les deux nouvelles coordonnées du point en question :

$$\begin{aligned} x' &= (x - x_O) * \cos(\alpha) + (y - y_O) * \sin(\alpha) + x_O \\ y' &= (x - x_O) * \cos(\alpha) + (y - y_O) * \sin(\alpha) + y_O \end{aligned}$$

Ici, les points x_0 et y_0 sont les coordonnées du point central de l'image, que nous retrouvons en fonction de la taille de celle-ci. Et x et y les coordonnées de notre pixel actuel. Si Hugo tu lis ce passage, nous te devons une part de gâteau au bananes (délice traditionnel australien) ! Nous récupérons ensuite les valeurs RGB du pixel (x,y) pour les mettre dans le pixel (x',y') .

1.3.3 Transformation Homographique

Le but de la transformation homographique est de faire une rotation de l'image tout en modifiant sa taille. Cette fonction lit en entrée les coordonnées de quatre points, ce seront celles des coins du sudoku. Nous nous sommes servi pour cela de la théorie homographique qui se base sur cette équation :

Soient $(h, a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2) \in R^9$

$$\begin{bmatrix} h \times x' \\ h \times y' \\ h \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \implies \begin{bmatrix} h \times x' \\ h \times y' \\ h \end{bmatrix} = \begin{bmatrix} a_1 \times x + a_2 \times y + a_3 \\ b_1 \times x + b_2 \times y + b_3 \\ c_1 \times x + c_2 \times y + 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_1 \times x + a_2 \times y + a_3 \\ c_1 \times x + c_2 \times y + 1 \\ b_1 \times x + b_2 \times y + b_3 \\ c_1 \times x + c_2 \times y + 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x \times x' & -y \times x' \\ 0 & 0 & 0 & x & y & 1 & -x \times y' & -y \times y' \end{bmatrix} \times \vec{d}$$

où $\vec{d} = [a_1 \ a_2 \ a_3 \ b_1 \ b_2 \ b_3 \ c_1 \ c_2]$

Ces calculs permettent de trouver la coordonnée logique que devrait avoir chaque pixel de coordonnée (x,y). Faisons de même pour nos 3 autres pixels, le tout dans une seule matrice :

$$\vec{e} = \mathbf{K} \times \vec{d} \Rightarrow \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1 \times x'_1 & -y_1 \times x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1 \times y'_1 & -y_1 \times y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2 \times x'_2 & -y_2 \times x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2 \times y'_2 & -y_2 \times y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3 \times x'_3 & -y_3 \times x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3 \times y'_3 & -y_3 \times y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4 \times x'_4 & -y_4 \times x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4 \times y'_4 & -y_4 \times y'_4 \end{bmatrix} \times \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \\ b_3 \\ c_1 \\ c_2 \end{bmatrix}$$

Pour trouver les valeurs de d il nous suffit de résoudre l'équation suivante :

$$\vec{d} = \mathbf{K}^{-1} \times \vec{e}$$

Nous pouvons alors garder ces constantes pour trouver les nouvelles coordonnées de chaque pixels de l'image. Les pixels qui débordent de la taille demandée en paramètre ne sont pas pris en compte.

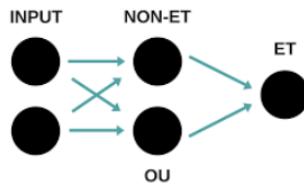
1.4 Segmentation

Après la transformation homographique, la grille est bien orientée et bien découpée. Il est donc possible de découper l'image en $9 \times 9 = 81$ carrés en partitionnant l'image. En effet, lors de la transformation homographique, nous redimensionnons l'image à un multiple de 9 pour faire en sorte que tous les carrés soient de même dimension. Ces carrés contiendront, ou non, les chiffres. Il est donc nécessaire de détecter si ces cases sont vides, ou si elles ont un chiffre. Pour se faire, nous réalisons la même logique que pour détecter la grille. Nous cherchons les blobs et gardons le plus grand, qui sera le chiffre envoyé dans le réseau de neurones.

2 Réseau de Neurones

2.1 XOR

Avant de commencer à travailler sur le réseau de neurones final, il fallait réaliser une porte logique XOR en guise de preuve de concept. Ceci fut assez simple à mettre en place. Composé de 3 couches, le réseau passe l'information initiale à la couche cachée puis, ce dernier, à la couche finale.



A chaque passage, les valeurs des neurones impliqués sont modifiées par les poids, de sorte :

$$z_i^{(l+1)} = \sum_{j=1}^n w_{ij}^{(l)} x_i + b_i^{(l)} \quad (1)$$

avec :

- l l'indice de la couche
- i l'indice de neurone référant à la couche $l + 1$
- j l'indice de neurone référant à la couche l
- n le nombre de neurones dans la couche à l'indice l

Nous utilisons la même fonction d'activation pour normaliser les valeurs de la couche cachée entre 0 et 1, ainsi que celle de la couche finale- la fonction **sigmoid** :

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (2)$$

et sa dérivée :

$$\frac{\partial}{\partial z} \sigma(z) = \sigma(z)(1 - \sigma(z)) \quad (3)$$

Enfin, avec notre valeur finale obtenue, nous appliquons la rétropropagation afin d'obtenir un résultat qui se rapproche un peu plus de la valeur désirée, en ajustant les poids :

$$\delta_j^{(l)} = \sum_{i=1}^n (w_{ij}^{(l)} + \delta_j^{(l+1)}) f'(z_j^{(l)}) \quad (4)$$

avec :

- l l'indice de la couche
- i l'indice de neurone référant à la couche $l + 1$

- j l'indice de neurone référant à la couche l
- n le nombre de neurones dans la couche à l'indice l

Toutefois, notre mise en oeuvre laissais à désirer. Avec un réseau de neurones aussi simple, nous nous ne sommes pas posés la question d'utiliser des structures, nous appuyant plutôt sur des tableaux.

```

24 // Param initialization
25
26 double train_input[8] = {0, 0, 0, 1, 1, 0, 1, 1}; // Training input
27 double train_key[4] = {0, 1, 1, 0}; // Training input results
28 double test_input[8] = {0, 0, 0, 1, 1, 0, 1, 1}; // Testing input
29 double test_key[4] = {0, 1, 1, 0}; // Testing input results
30
31 double L2[2]; // Hidden layer neurons
32 double L3[1]; // Output layer neuron
33
34 double L1_to_L2_weights[4]; // Input to Hidden layer weights
35 double L2_to_L3_weights[2]; // Hidden to Output layer weights
36
37 double learning_rate = 0.5; // Neural network LR
38
39 double *pointer = NULL; // Pointer for training or testing
40
41 // Chain rule variables
42
43 double derivative_L3[1]; // Output layer derivative
44 double derivative_L2_to_L3[1]; // Hidden to output layer value derivative
45
46 double derivative_L2_to_L3_weights[2]; // Hidden to output layer weights derivatives
47 double L2_suggested_weight_changes[2];
48
49 double derivative_L2[2]; // Hidden layer derivatives
50 double derivative_L1_to_L2[2]; // Input to hidden layer values derivatives
51
52 double derivative_L1_to_L2_weights[4]; // Input to hidden layer weights derivatives
53 double L1_suggested_weight_changes[4];

```

FIGURE 15 – Structure "hard-codée" du XOR- oui, c'est moche

Étant donné que les valeurs d'entrée et la structure étaient assez rudimentaires, notre réseau mettait très peu de temps à compiler et nous affichais le résultat désiré dans 100% des cas.

```
antithetical@Chunchunmaru:~/EPITA/OCR/neural_network/xor$ ./a.out
Output      Desired output
0.000000  0.000000
1.000000  1.000000
1.000000  1.000000
0.000000  0.000000

Input layer weights
0.972751
9.011277
0.972751
9.011372

Hidden layer weights
-61.300918
49.214821
```

FIGURE 16 – Sortie du XOR

2.2 Digit Recognition

Le réseau de neurones que nous utilisons pour identifier correctement les chiffres sur le sudoku a donc suivi le même principe. L’implémentation, cependant, fut totalement différente. En utilisant des structures pour mettre en place le réseau, les couches et les neurones, nous pouvions facilement manipuler nos paramètres d’entrée, nommés "hyperparamètres".

```
void train(unsigned int nb_hidden, unsigned int nb_neurons, double learning_rate);
```

FIGURE 17 – Les hyperparamètres en entrée de la fonction "train"

Cependant, certains détails ont effectivement changé : nous utilisons dorénavant la fonction d’activation `ReLU`, ainsi que la fonction `sigmoid`, seulement pour la dernière couche de neurones. La fonction `ReLU` nous a permis d’avoir des meilleures performances, ce qui fut essentiel, étant donné l’envergure du réseau.

$$Relu(z) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases} \quad (5)$$

et sa dérivée :

$$Relu'(z) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases} \quad (6)$$

La performance ainsi que les résultats du réseau varient ; cela en fonction du nombres de couches cachées, du nombre de neurones par couche cachée mais aussi en fonction du taux d’apprentissage. Ce dernier influence la capacité d’apprentissage du réseau : un taux d’apprentissage trop élevé ou trop bas s’avère néfaste

pour le taux de réussite du réseau. Ainsi, ce paramètre doit être minutieusement choisi.

```
antithetical@Chunchunmaru:~/EPITA/OCR/ocr-0x12R$ ./main --mode TRAIN --learning-rate 1
Do you wish to test the existing weights? [Y/N] (default: N)
N
Success rate: (641 / 6298) → 10%.
antithetical@Chunchunmaru:~/EPITA/OCR/ocr-0x12R$ ./main --mode TRAIN --learning-rate 0.1
Do you wish to test the existing weights? [Y/N] (default: N)
N
Success rate: (5444 / 6298) → 86%.
antithetical@Chunchunmaru:~/EPITA/OCR/ocr-0x12R$ ./main --mode TRAIN --learning-rate 0.01
Do you wish to test the existing weights? [Y/N] (default: N)
N
Success rate: (2414 / 6298) → 38%.
```

FIGURE 18 – Grande variation du taux de succès

Toutefois, l'efficacité du taux d'apprentissage dépend également des autres hyperparamètres. A cette fin, nous avons créé un fichier Python qui compile le réseau de neurones, tout en ajustant les hyperparamètres, afin d'obtenir le meilleur résultat possible. Cela nécessitait donc de sauvegarder les hyperparamètres ainsi que les poids, qui régissent la valeur de sortie du réseau, dans un fichier texte.

```
1145
```
_0.559389(0.350138(0.134346(0.778815(0.004964(0.402230(0.588140(0.783333(0.156353(0.595653(0.812717(0.715986(0.482235(0.602732(0.307374(0.558767(0.771258(0.197575(0.453689(0.487360(0.506202(0.772679(0.9746
_0.202170(0.207870(0.560843(0.188454(0.176789(0.084949(0.892732(0.773722(0.215873(0.042010(0.091198(0.309486(0.659010(0.541209(0.751089(0.083316(0.657953(0.19654(0.603390(0.438144(0.538925(0.543822(0.0239
_0.161300(0.1805659(0.793876(0.159615(0.610720(0.727246(0.816757(0.702747(0.116782(0.626357(0.821349(0.604962(0.347436(0.770771(0.866989(0.467094(0.994511(0.412336(0.710208(0.811271(0.840502(0.99530(0.2478
_0.275008(0.1748457(0.184455(0.514226(0.552645(0.466445(0.212514(0.050804(0.227862(0.791051(0.251340(0.177500(0.05261(0.640189(0.505257(0.225980(0.816892(0.610081(0.313623(0.506962(0.422988(0.20171(0.9718
_0.099708(0.145951(0.092701(0.409210(0.61540(0.808633(0.105924(0.24040(0.18990(0.747054(0.513459(0.303542(0.44974(0.579389(0.327110(0.343528(0.290145(0.330721(0.102193(0.444380(0.131210(0.703518(0.8559
_0.725012(0.130541(0.802452(0.100860(0.304156(0.587733(0.804931(0.212781(0.14124(0.49330(0.023465(0.308233(0.377897(0.41349(0.01958(0.160173(0.534812(0.70399(0.51324(0.13789(0.747125(0.779733(0.505687(0.2134
_0.713084(0.226322(0.842420(0.212778(0.344550(0.141124(0.132382(0.010846(0.510481(0.098681(0.223378(0.465418(0.36497(0.818579(0.140269(0.871785(0.591574(0.47177(0.801663(0.065738(0.112955(0.402283(0.5266
_0.471188(0.064905(0.306340(0.598464(0.397328(0.700407(0.430510(0.095144(0.552048(0.087081(0.113307(0.513658(0.516591(0.291188(0.725133(0.826331(0.806651(0.237159(0.904484(0.777571(0.840303(0.054081(0.4955
_0.498839(0.15903(0.185789(0.204546(0.19976(0.386205(0.435869(0.548152(0.474140(0.92386(0.589120(0.415628(0.020389(0.50480(0.360764(0.362410(0.43578(0.582157(0.19330(0.104910(0.049310(0.0501
_0.150547(0.046651(0.78771(0.018519(0.845462(0.652181(0.846891(0.47393(0.601879(0.875644(0.74024(0.003116(0.059224(0.004939(0.097018(0.564691(0.897704(0.27125(0.559337(0.301850(0.2589
_0.348976(0.114059(0.312971(0.422792(0.92014(0.375291(0.035643(0.43322(0.191910(0.254930(0.087351(0.43710(0.7994(0.73949(0.334352(0.423659(0.851677(0.612854(0.356236(0.036081(0.270721(0.7070
_0.375923(0.027921(0.361522(0.77513(0.552901(0.87412(0.59262(0.236492(0.8895110(0.54910(0.27027(0.048969(0.672031(0.281926(0.175794(0.056914(0.276479(0.340489(0.641905(0.18479(0.53089(0.2031
_0.786577(0.583864(0.079156(0.446252(0.617842(0.110927(0.922886(0.805197(0.129967(0.723250(0.84383(0.16646(0.34979(0.10789(0.432724(0.88458(0.048152(0.493858(0.004661(0.12445(0.253748(0.45462(0.6128
_0.907292(0.581651(0.23377(0.23109(0.442424(0.595413(0.98957(0.53464(0.27730(0.672431(0.39993(0.47434(0.52764(0.445194(0.432079(0.75661(0.85479(0.457279(0.71394(0.49417(0.2340(0.7029
_0.598118(0.05630(0.120887(0.64474(0.792501(0.469440(0.162260(0.089746(0.81246(0.304657(0.304657(0.433199(0.43265(0.160220(0.13226(0.32456(0.294616(0.160220(0.13383(0.104956(0.270839(0.113730(0.2778
_0.811097(0.290757(0.427634(0.32438(0.67970(0.52749(0.550682(0.49797(0.50206(0.570253(0.075231(0.852749(0.51379(0.51456(0.295852(0.502007(0.502007(0.445223(0.471084(0.246504(0.5116
_0.076950(0.445203(0.905460(0.844455(0.822294(0.742499(0.463393(0.43432(0.37439(0.511451(0.97738(0.686719(0.76334(0.42057(0.435575(0.305275(0.091099(0.460399(0.32331(0.225924(0.435258(0.0874
_0.32209(0.447123(0.30335(0.107239(0.602294(0.19705(0.111186(0.05971(0.88991(0.301283(0.93546(0.02674(0.776731(0.07366(0.29130(0.124125(0.720391(0.845107(0.25303(0.6195
_0.815565(0.140657(0.28832(0.73394(0.432447(0.584711(0.89754(0.55736(0.51649(0.76213(0.23638(0.9233(0.59157(0.15130(0.085592(0.423037(0.71518(0.3133(0.6154
_0.845860(0.880649(0.26030(0.458759(0.212367(0.30556(0.58559(0.15080(0.772038(0.650532(0.05057(0.45842(0.179736(0.408723(0.583817(0.32032(0.329359(0.176951(0.723850(0.51036(0.113188(0.2882
_0.376657(0.59013(0.494623(0.32467(0.07980(0.291283(0.577637(0.548631(0.44613(0.27745(0.30546(0.269862(0.27945(0.32564(0.37744(0.374578(0.89751(0.94178(0.320081(0.381121(0.51274(0.329564(0.50981(0.4902
_0.046451(0.16531(0.84672(0.79834(0.07980(0.291283(0.57524(0.53749(0.407093(0.33941(0.88975(0.88793(0.508118(0.47984(0.97831(0.25583(0.18918(0.84847(0.7918
```

FIGURE 19 – Fichier poids

Ceci fut implémenté, ainsi que les outils nécessaires pour charger ces données d'un fichier dans un nouveau réseau de neurones. De sorte, le fichier Python pouvait donc sauvegarder la meilleure itération du réseau de neurones pour que nous puissions le tester et comparer a posteriori.

```

1 current_time,nb_hidden,nb_neurons,learning_rate,success_percent
2 "09/11/2022, 00:17:10",1,20,0.05,86
3 "09/11/2022, 00:17:16",1,20,0.06,87
4 "09/11/2022, 00:17:22",1,20,0.07,88
5 "09/11/2022, 00:17:28",1,20,0.08,89
6 "09/11/2022, 00:17:33",1,20,0.09,89
7 "09/11/2022, 00:17:39",1,20,0.1,89
8 "09/11/2022, 00:17:45",1,20,0.11,90
9 "09/11/2022, 00:17:51",1,20,0.12,89
10 "09/11/2022, 00:17:57",1,20,0.13,90
11 "09/11/2022, 00:18:05",1,20,0.14,90
12 "09/11/2022, 00:18:11",1,20,0.15,90
13 "09/11/2022, 00:18:17",1,20,0.16,90
14 "09/11/2022, 00:18:23",1,20,0.17,91
15 "09/11/2022, 00:18:29",1,20,0.18,90
16 "09/11/2022, 00:18:35",1,20,0.19,91
17 "09/11/2022, 00:18:40",1,20,0.2,90
18 "09/11/2022, 00:18:46",1,20,0.21,91
19 "09/11/2022, 00:18:52",1,20,0.22,91
20 "09/11/2022, 00:18:58",1,20,0.23,92
21 "09/11/2022, 00:19:04",1,20,0.24,92
22 "09/11/2022, 00:19:10",1,20,0.25,91
23 "09/11/2022, 00:19:16",1,20,0.26,91
24 "09/11/2022, 00:19:22",1,20,0.27,92
25 "09/11/2022, 00:19:28",1,20,0.28,91
26 "09/11/2022, 00:19:34",1,20,0.29,91
27 "09/11/2022, 00:19:40",1,20,0.3,92
28 "09/11/2022, 00:19:46",1,20,0.31,92
29 "09/11/2022, 00:19:52",1,20,0.32,92
30 "09/11/2022, 00:19:58",1,20,0.33,92
31 "09/11/2022, 00:20:04",1,20,0.34,92
32 "09/11/2022, 00:20:10",1,20,0.35,92
33 "09/11/2022, 00:20:16",1,20,0.36,92
34 "09/11/2022, 00:20:22",1,20,0.37,92
35 "09/11/2022, 00:20:28",1,20,0.38,92
36 "09/11/2022, 00:20:34",1,20,0.39,91
37 "09/11/2022, 00:20:40",1,20,0.4,91
38 "09/11/2022, 00:20:46",1,20,0.41,92
39 "09/11/2022, 00:20:51",1,20,0.42,93
40 "09/11/2022, 00:20:57",1,20,0.43,92
41 "09/11/2022, 00:21:04",1,20,0.44,92
42 "09/11/2022, 00:21:10",1,20,0.45,92

```

FIGURE 20 – Fichier CSV

### 2.2.1 L'architecture de notre modèle

Tout d'abord, nous avons commencé par mettre en place la structure du réseau. Certains des paramètres de cette structure étaient prédéfinis : le réseau possède au moins toujours 2 couches : une couche d'entrée et une de sortie. Toutefois, il reste quand même possible de modifier la taille des données d'entrée et de sortie. De plus, le nombre de couches cachées, c'est-à-dire les couches se situant entre la couche d'entrée et de sortie, est modifiable, ainsi que le nombre de neurones que ces couches contiennent :

```

typedef struct Network
{
 unsigned int size_input;
 unsigned int size_output;
 unsigned int nb_layers;
 unsigned int nb_hidden_neurons;
 Layer* layers;
} Network;

```

FIGURE 21 – Structure du réseau de neurones

Ensuite, le réseau initialise les couches. Le nombre de neurones de la couche d'entrée et le nombre de neurones de la couche de sortie sont définis par l'utilisateur en initialisant le réseau. Le reste des couches partagent le même nombre de neurones, également définies par l'utilisateur :

```
typedef struct Layer
{
 unsigned int nb_neurons;
 Neuron* neurons;
} Layer;
```

FIGURE 22 – Structure d'une couche

Enfin, chaque couche initialise ses neurones. Le nombre de poids dépend du nombre de neurones dans la couche précédente : chaque neurone possède autant des poids qu'il y a de neurones dans la couche précédente. Leur valeur est initialisée aléatoirement entre 0 et 1 lorsque le réseau est lancé.

De plus, chaque neurone possède un "delta", qui indique au réseau si le poids de ce neurone doit être augmenté afin d'avoir plus d'importance :

```
typedef struct Neuron
{
 unsigned int nb_weights;
 double* weights;
 double value;
 double delta;
} Neuron;
```

FIGURE 23 – Structure d'un neurone

### 2.2.2 Les bases de données - MNIST

Pour un réseau de neurones, une base de donnée est une collection de données (des images dans le cas de l'OCR), qui permettent d'entraîner le réseau avant d'effectuer les tests. Ces images doivent être très proches des images réelles, notamment vis à vis du format, afin d'avoir de résultats satisfaisant.



FIGURE 24 – Exemples de chiffres de MNIST

La première base de données que nous avions utilisé était celle du *MNIST*, regroupant plus de 60,000 images de chiffres (en format 28x28) manuscrites. Après avoir ajusté les hyperparamètres, le taux de réussite est monté jusqu'à 95%.

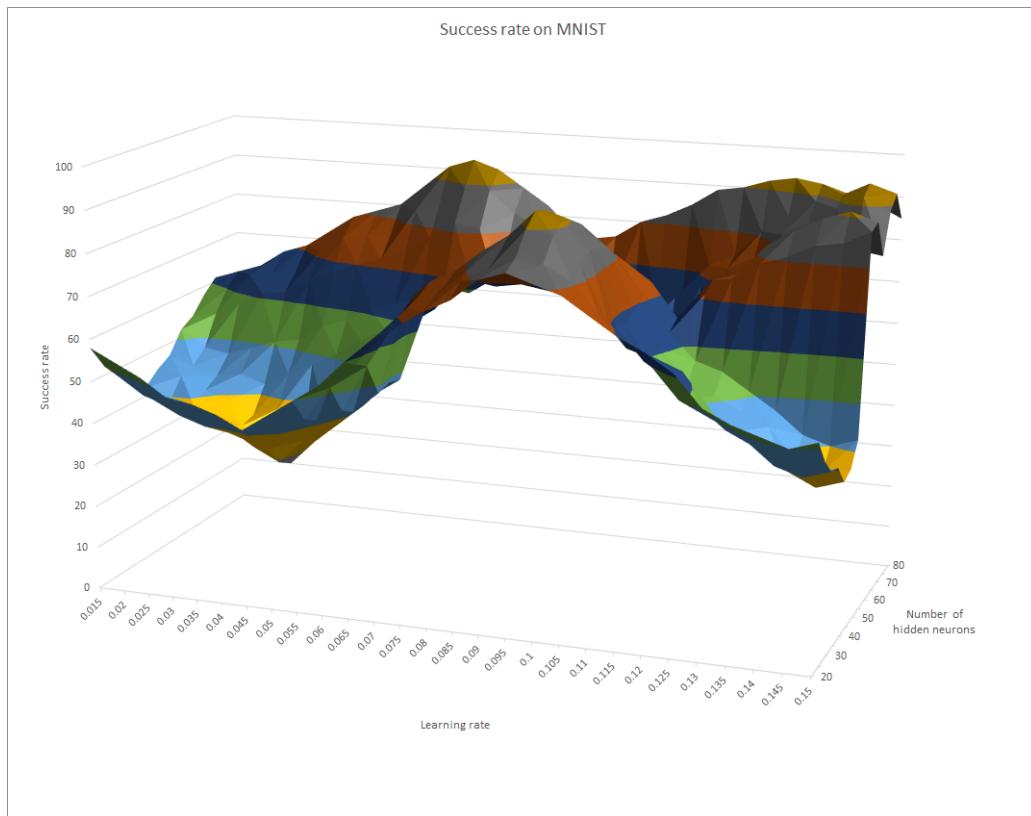


FIGURE 25 – Taux de réussite sur MNIST avec les hyperparamètres donnés

Cependant, *MNIST* présente de nombreuses limites : la première- l'inclusion du zéro. Le chiffre zéro n'est pas présent sur une grille de sudoku. Or, ce dernier représente un dixième des chiffres dans la base de donnée- une partie assez conséquente. D'autant plus que cette partie était complètement inutilisée, ainsi entraînant des calculs en plus n'ayant aucune utilité. *MNIST* est formaté sous la forme d'un fichier *idx3-ubyte*, compact et pratique. Toutefois, ce type de fichier est difficilement modifiable- supprimer le zéro de la base de donnée s'aurait avéré une tâche pénible.

Le deuxième problème survenais des cases vides. En effet, un réseau de neurones est largement capable de détecter des cases vides (même si du bruit y est présent). Cela implique alors qu'il faut des images vides dans la base de donnée que nous utilisons pour l'entraîner- or, *MNIST* ne contient pas de images vides. Il était donc absolument nécessaire de modifier la base de donnée, quelque-chose que nous étions extrêmement réticents à faire.

Enfin, le dernier problème était celui de la précision. Malgré les résultats très satisfaisant du notre réseau de neurones sur *MNIST*, les résultats sur des images réelles extraites de sudoku était peu concluants. En moyenne, le taux de réussite

maximal était assez médiocres, autour des 50%. La raison de cette baisse en performance était évidente : *MNIST* utilise des chiffres manuscrits, cependant, les grilles de sudoku contiennent des chiffres *digitales*.

### 2.2.3 Les bases de données - Numeric

Après de nombreuses recherches, nous avons choisis d'utiliser la base de donnée *Numeric*. Elle contient environ 4000 images d'un ensemble de données numériques. Chaque image est de dimension 28x28 et est en *greyscale*. Cette base de donnée a été créé pour la classification des chiffres du sudoku, c'est pourquoi il présente une image vide pour les zéros.

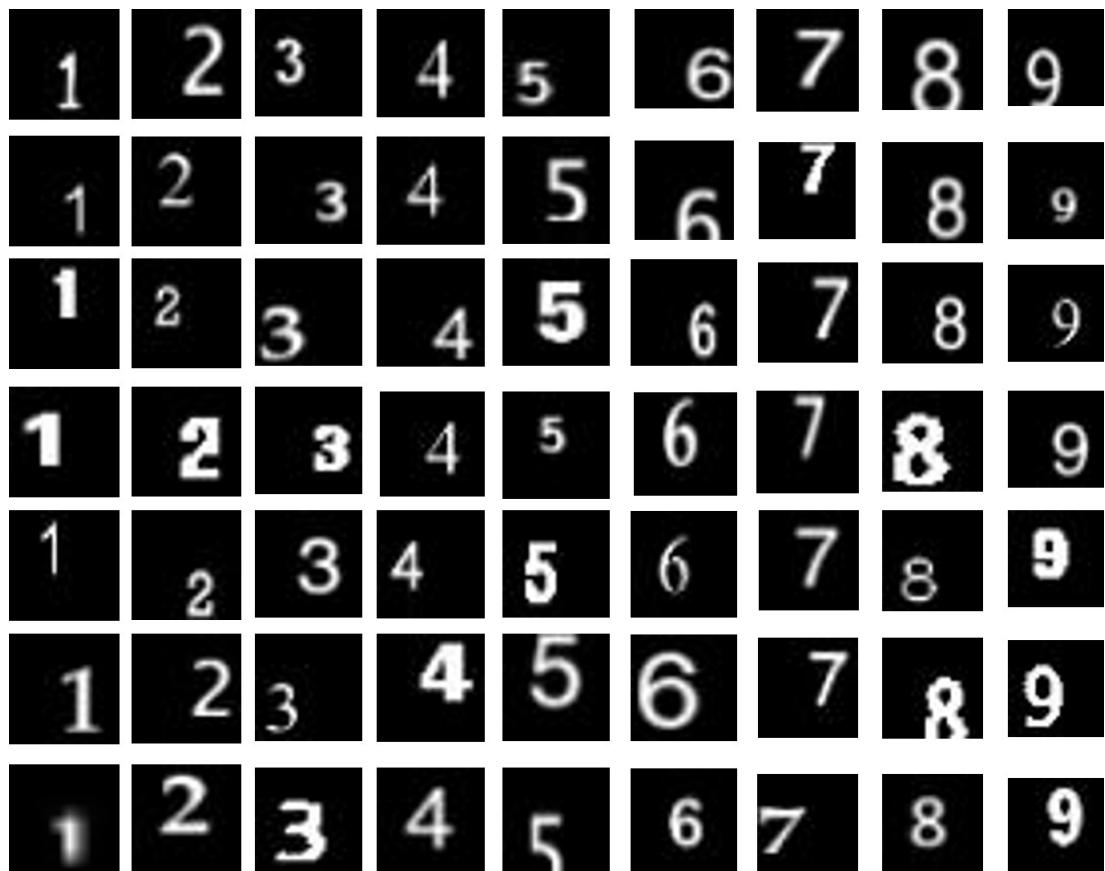


FIGURE 26 – Exemples de chiffres de Numeric

Malgré un taux de réussite maximal moins élevé sur cette base de donnée-92%, ce dernier restait stable lorsqu'il était utilisé sur des images réelles. Ainsi, nous obtenons de résultats très satisfaisant et nous avons fait le choix d'utiliser *Numeric* comme base de donnée définitive.

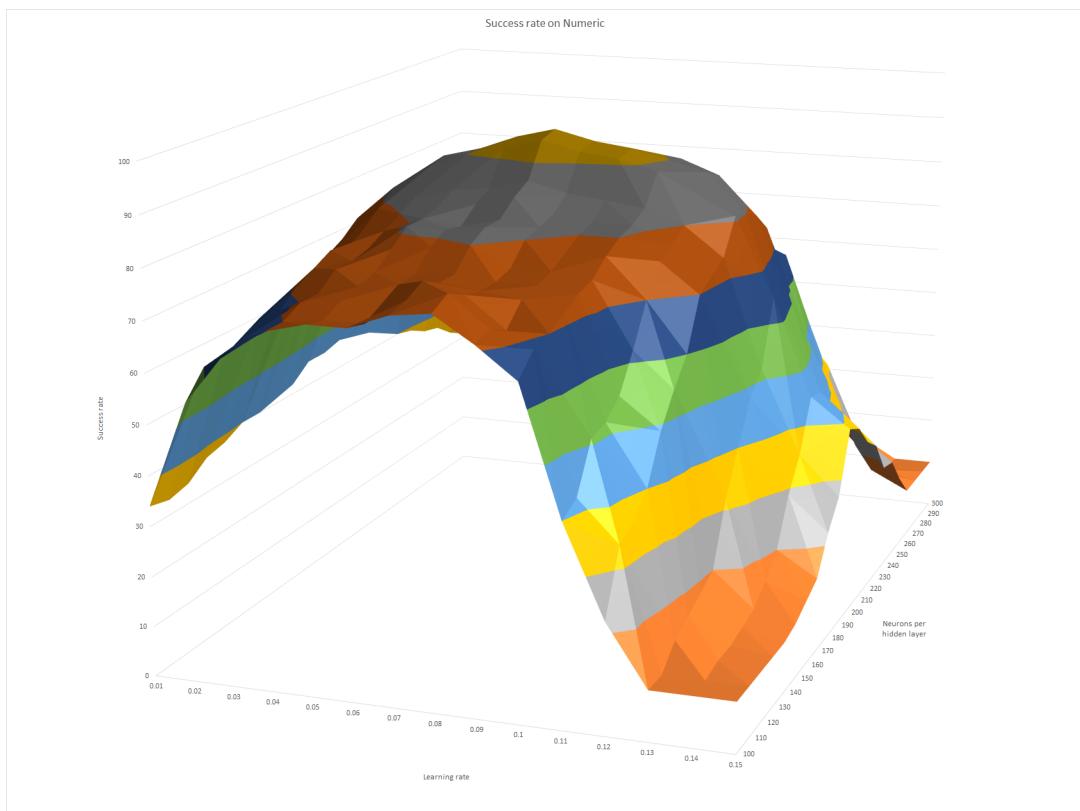


FIGURE 27 – Taux de réussite sur Numeric avec les hyperparamètres donnés

### 3 Résolution du sudoku

#### 3.1 Importation

En ce qui concerne la résolution du sudoku, nous devons respecter un format de fichier pour simplifier la lecture et l'écriture de la grille. Ainsi, nous lisons le fichier et créons une matrice d'entier de 9 par 9 pour l'envoyer dans notre algorithme de résolution, le backtracking ou retour sur trace.

```

... .4 58.
... 721 ..3
4.3
21. .67 ..4
.7. ... 2..
63. .49 ..1
3.6
... 158 ..6
... ..6 95.

```

FIGURE 28 – Format à respecter

#### 3.2 Backtracking

Le backtracking est un algorithme récursif qui parcourt toutes les valeurs possible des cases jusqu'à trouver si le sudoku est résolu. Les cas d'arrêts sont soit nous trouvons que le sudoku est résolu. Soit qu'il présente une erreur. Dans ce cas, nous retournons un appel en arrière. Cette image permet d'illustrer les appels récursif réalisé par cet algorithme jusqu'à trouver une solution au sudoku.

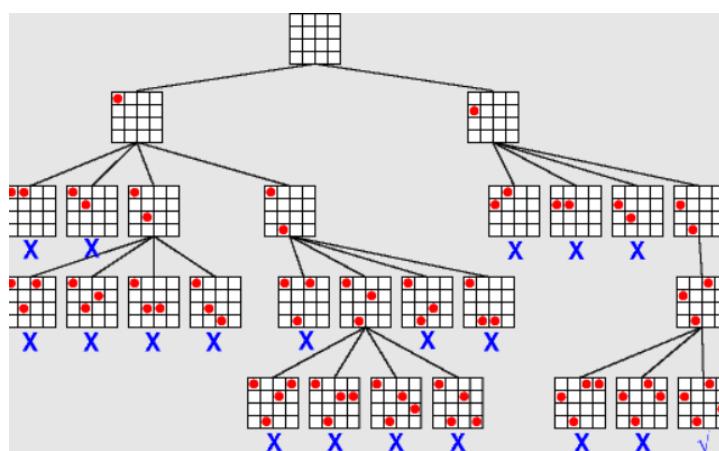
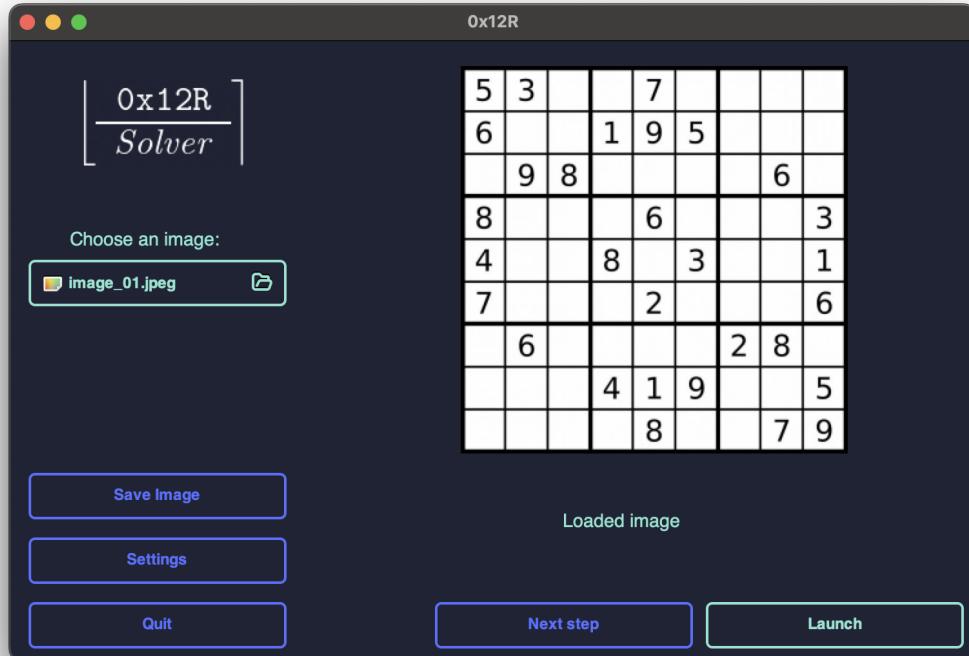


FIGURE 29 – Illustration du backtracking

## 4 Interface Graphique

Pour créer l'interface graphique, nous avons choisi le package [CTK3](#). Il est gratuit et il est la base de beaucoup d'applications Linux se basant sur l'environnement de bureau *GNOME* comme [GTK3](#). Cela implique qu'il soit déjà installé sur la plus part des systèmes Linux, même s'ils ne sont pas sous environnement *GNOME*. Nous avons utilisé le logiciel *Glade* pour générer un fichier de configuration de notre interface. Nous avons ensuite créé une feuille de style *CSS*, qui contient les couleurs et toutes les caractéristiques visuelles de notre interface. Nous nous retrouvons alors dans une configuration type web. D'un côté, notre fichier [.xml](#) généré par *Glade*, contenant chaque élément de notre interface de façon sérialisée, et de l'autre notre fichier [.css](#) constitué de propriétés graphiques pour chacun des éléments. Avec ces deux fichiers, nous avons codé une sorte de "moteur graphique" qui permet de parser notre fichier de configuration, d'en afficher les différents éléments, puis d'y appliquer le style défini. Pour notre design, nous avons décidé de faire quelque chose de simple et d'intuitif. Nous avons réduit le nombre de boutons au strict minimum, et ajouté une zone qui permet d'afficher l'état d'avancement du programme. Nous avons ajouté une fonction permettant de changer une case si jamais la segmentation ne trouve pas le bon chiffre. Cette case apparaît au moment de la dernière étape. Nous avons aussi une fenêtre pour choisir les paramètres du réseau de neurones. Ce dernier est réentraîné une fois que cette fenêtre est fermée. Pour gérer l'affichage de nos images, nous avons décidé de passer par un tableau de pointeurs de fonctions. Nous avons normalisé une dizaine de fonctions *launcher* permettant d'appliquer les filtres à l'image. À chaque fois que le bouton NEXT est clické, nous passons à la prochaine dans le tableau. Elle modifie l'image en place, et cette image est affichée.

Voici une capture d'écran de notre interface :



Pour que l'utilisateur puisse vérifier que le sudoku interprété par l'ordinateur ne contient pas d'erreurs. Nous affichons une image avant sa résolution. Dans un premier temps il peut voir la grille du sudoku avec les chiffres détectés affichés en marron. Puis il peut modifier les nombres qu'il souhaite afin que le sudoku soit bon en cas d'erreurs de l'intelligence artificielle. Ceux-ci seront affichés en vert afin de s'assurer que la modification a été faite au bon endroit. Une fois ces modifications apportées, le sudoku solver est lancé et affiche les numéros ajoutés d'une couleur bleuté. Les couleurs choisies ont été sélectionné de façon à respecter la charte graphique de l'interface. Voici quelques images pour vous montrer le résultat :

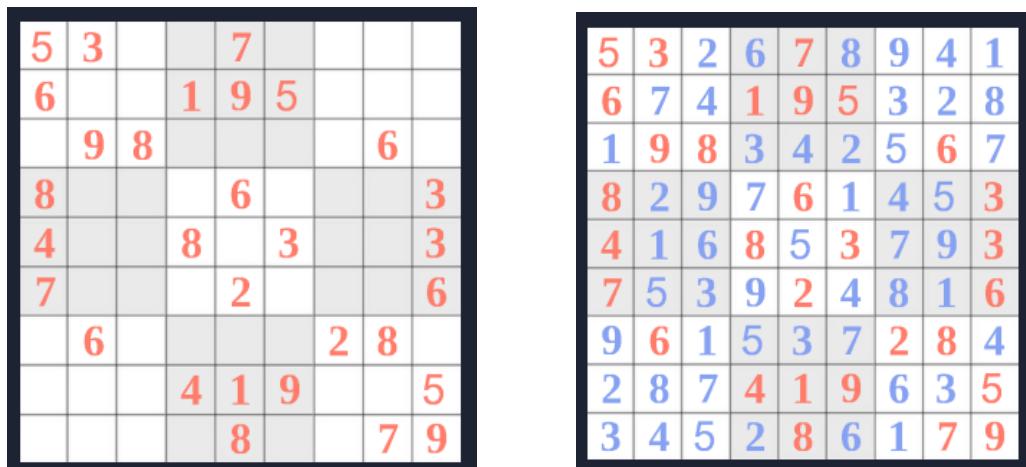


FIGURE 30 – Résolution de la grille affichée sur l'UI

## Conclusion

En conclusion, nous sommes très contents du travail effectué lors de ce semestre. Nous avons pu mettre en place le traitement complet de l'image, un réseau de neurones capable de reconnaître des chiffres digitales (et manuscrits!). Nous avons enfin pu créer une interface graphique qui permet d'utiliser notre OCR. En travaillant ensemble, notre projet s'est concrétisé et a dépassé toutes nos attentes. L'entraide et l'écoute, ainsi que la coopérations, furent les mots d'ordre de ce groupe. Ce fut donc également une expérience enrichissante, non seulement d'un point de vue technique, mais aussi humainement. Nous sommes extrêmement fiers de la qualité de notre projet et du sérieux que nous y avons apporté. Nous espérons que vous allez être tout aussi impressionnés que nous, lorsque le sudoku solver a fonctionné correctement pour la première fois. Rappelez-vous ; 12 !

## Références

### Packages

- [maths.h](#)
- [SDL2](#) (<https://www.libsdl.org/>)
- [SDL2\\_Image](#) (<https://www.libsdl.org/>)
- [GTK3](#) (<https://www.gtk.org/>)

### Outils

- [Vim](#) (<https://www.vim.org/>)
- [gcc](#) (<https://gcc.gnu.org/>)
- [Visual Studio Code](#) (<https://code.visualstudio.com/>)
- [Glade](#) (<https://glade.gnome.org/>)

### Documentation

- [stackoverflow.com](#)
- <https://magpi.raspberrypi.com/books/c-gui-programming>
- <https://youtube.com/channel/UC5oHS9h-prWeBrBNzXm8rYA>