Cliptronix

Rapport de Projet

Aergia

Paul FOURNILLON
Titouan GRAGNIC
Titouan GUESDON
Alex BLANCO

10 novembre 2022

EPITA 2026 - PROJET SPE S3

Table des matières

1 Introduction

Pour notre troisième semestre à EPITA, le projet que nous devons réaliser est un logiciel *Optical Character Recognition* (Reconnaissance Optique de Charactères en français, par la suite abrégé en OCR) capable de résoudre des grilles de sudoku quelles qu'elles soient. La reconnaissance optique de charactères peut se définir comme le processus de convertion d'une image composée de texte en texte lisible par une machine.

1.1 Présentation de Cliptronix

Ce projet est à réaliser par groupe de 4 personnes. Voici les membres du studio Cliptronix :

- Alex BLANCO
- Titouan GRAGNIC
- Titouan GUESDON
- Paul FOURNILLON

Et voici son logo :

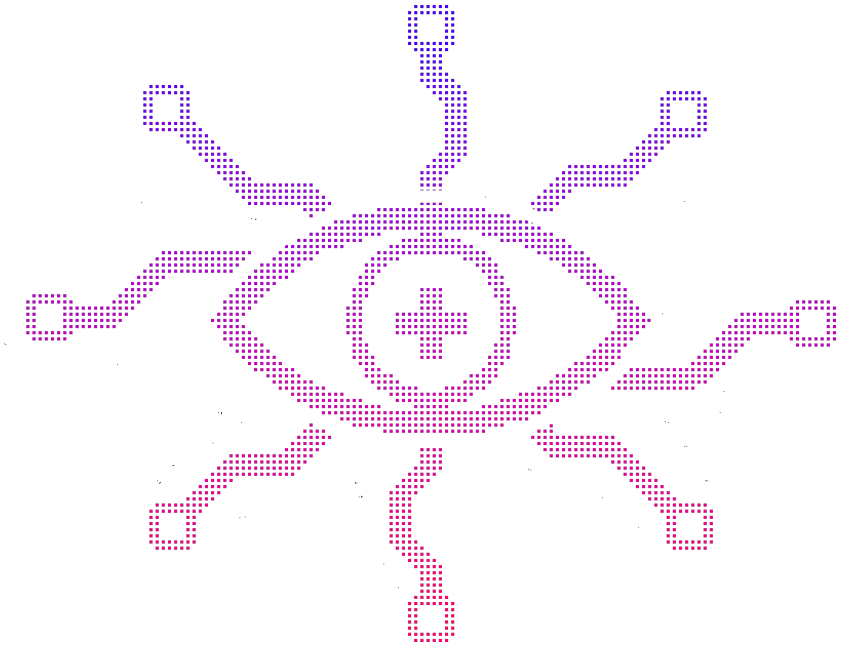

FIGURE 1 – Logo de Cliptronix

1.2 Aergia

Cliptronix se lance donc dans la réalisation d'un nouveau projet nommé : *Aergia*. Voici son logo :

FIGURE 2 – Logo d'Aergia

Pour mener à bien ce nouveau challenge, nous avons découpé le projet en 4 parties principales :

- le réseau de neurones
- le traitement d'image
- l'interface graphique
- le solveur de sudoku

Nous nous sommes chacun répartis sur les différentes tâches à réaliser afin d'avancer ces différentes parties en parallèle avant de mettre en commun notre travail.

1.3 Répartition des tâches

Voici un tableau récapitulatif de la répartition des tâches au sein de Cliptronix. Cette répartition est à titre indicatif et est susceptible d'évoluer tout au long du projet en fonction de l'avancée globale du projet et des besoins de chacun.

Répartition des tâches	**Paul**	**Titouan GR**	**Titouan GU**	**Alex**
Réseau de neurones	x			
Traitement d'images		x	x	
Solveur		x		
Interface graphique				x

TABLE 1 – Tableau représentant la répartition des différentes tâches entre les membres de Cliptronix. Légende : x : responsable.

2 Réalisations Techniques

2.1 Réseau de neurones

Afin de reconnaître les chiffres présents sur une grille de sudoku, nous devons implémenter un réseau de neurones capable d'apprendre à reconnaître les différents chiffres et lettres possiblement présents sur une image.

2.1.1 Principe de base

Un réseau de neurones est composé de plusieurs neurones appelés perceptrons, qui intéragissent entre eux. Ces neurones possèdent un biais qui est une valeur numérique et chaque liaison d'un neurone à un autre est valué par ce qu'on appelle un poids.

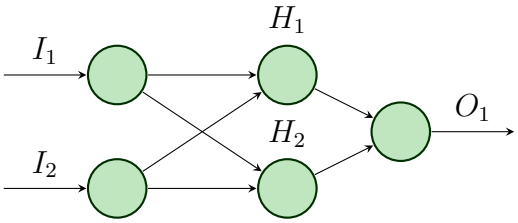

FIGURE 3 – Représentation d'un réseau de neurones avec 1 couche cachée.

L'apprentissage dans un réseau de neurones consiste à calculer les sorties de ce réseau de neurones en fonction des différents biais et poids puis à actualiser les valeurs de ces poids et biais en fonction de l'erreur calculée entre la valeur de sortie calculée et la valeur de sortie attendue.

Pour calculer les sorties des neurones, on applique une fonction mathématique appelée fonction de transfert. Dans notre cas, il s'agira de la fonction *sigmoïde*.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Ce phénomène de calcul des valeurs de sorties des neurones pour chacune des couches du réseau est appelée *propagation vers l'avant*. Les sorties d'une couche de neurones sont calculées en appliquant la fonction sigmoïde au produit du poids et de la sortie de la couche précédente de même index auquel on additionne le biais de même indice. Les équations suivantes explicitent ces calculs en utlisant des outils mathématiques différents.

$$a^{(1)} = \sigma\left(w_{1,0}a_0^{(0)} + w_{1,1}a_1^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right)$$

$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)$$

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma \left[\begin{pmatrix} w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ w_{2,0} & w_{2,1} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix} \right]$$

$$\mathrm{a}^{(1)} = \sigma\left(\mathbf{W}^{(0)} a^{(0)} + \mathbf{b}^{(0)}\right)$$

Ensuite, pour finaliser le processus d'apprentissage, il faut actualiser les valeurs des poids et des biais des neurones afin que les prochaines valeurs calculées par le réseau soient plus précises. Pour cela, on parcourt le réseau dans le sens inverse (c'est-à-dire en partant de la dernière couche) et on réalise une *propagation arrière*. Autrement dit, on calcule l'erreur entre la valeur de sortie attendue et la valeur obtenue et cette erreur nous sert à augmenter ou diminuer la valeur de nos poids et biais en fonction du coefficient d'apprentissage. On applique ce processus jusqu'à la première couche.

2.1.2 Implémentation

Pour implémenter notre réseau de neurones, nous avons découpé l'architecture en 3 parties principales. Ces 3 parties correspondent à 3 structures de données (*struct* en C).

La première structure est celle correspondant à une couche de notre réseau de neurones. Elle contient les matrices de poids, biais, erreurs, valeurs de sorties de la couche.

La deuxième est celle représentant le réseau de neurones. Cette structure regroupe toutes les couches du réseau, ce qui permet d'effectuer les opérations beaucoup plus facilement.

Enfin, la dernière structure est une structure nommée *entraînement*, qui contient les matrices de valeurs d'entrées du réseau, de valeurs de sorties.

L'utilisation de ces structures de données permet de rendre le code beaucoup plus lisibles et les opérations beaucoup plus faciles à implémenter. De plus, en structurant de la sorte notre code, il va être simple de lui faire apprendre des chiffres. En effet, il n'y aura qu'à changer l'ensemble d'entraînement et les dimensions du réseau de neurones (nombre de couches, nombre de neurones par couche) pour pouvoir commencer à l'entraîner.

2.1.3 *Proof Of Concept* - XOR

Pour montrer que notre réseau de neurones était fonctionnel, nous devions réaliser une preuve de concept en apprenant à notre réseau la fonction logique XOR.

Entrée 1	Entrée 2	Sortie
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 2 – Table de vérité de la fonction XOR.

Pour cela, nous avons du implémenté un réseau de neurones à 3 couches (composé d'une couche cachée, voir 3) car l'opération XOR ne peut pas être représentée par un modèle linéaire il nous faut donc obligatoirement plusieurs neurones même pour apprendre cette opération pourtant simple.

En effet, on peut voir sur l'image suivante que l'opération XOR n'est pas linéaire (ses valeurs s'obtiennent en réalisant des arcs de cercle sur le graphique). La fonction XOR peut en fait se décomposer en une opération logique de fonctions qui elles sont linéaires (XOR = OR AND NAND). Nos neurones ont donc pour objectif de respectivement apprendre les opérations OR NAND et AND.

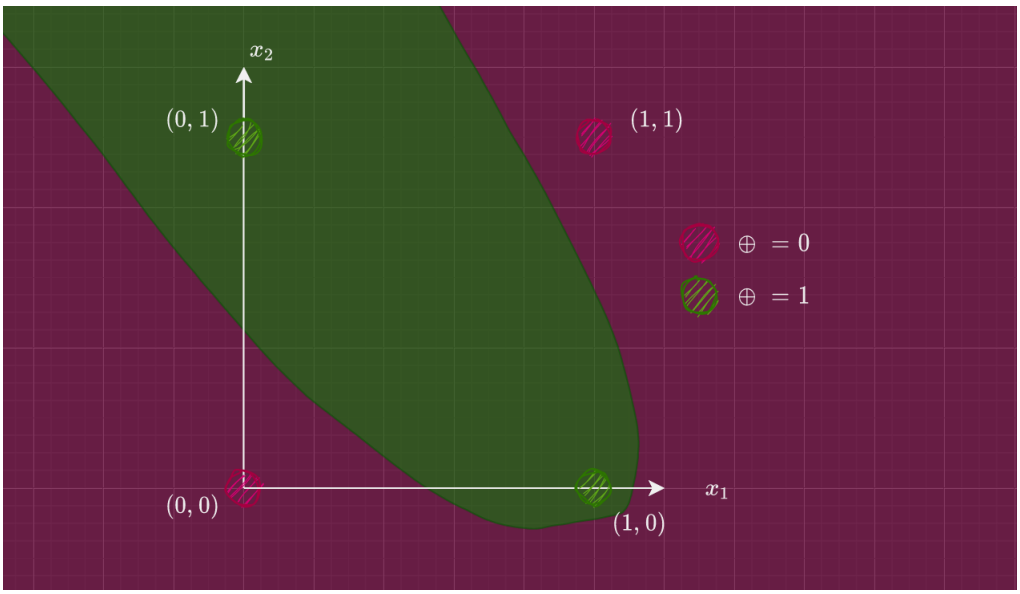

FIGURE 4 – Graphe montrant que la fonction XOR n'est pas linéaire.

Pour entraîner notre réseau, nous avons choisi d'effectuer 5000 itérations. Cette valeur est peut-être excessive mais nous ne risquions pas de sur-entraîner notre réseau étant donné l'ensemble de tests très réduit et le peu de variété dans les valeurs possibles.

2.2 Traitement d'images

2.2.1 Prétraitement

Afin de rendre notre image plus lisible pour l'ordinateur et qu'elle soit plus simple à détecter, il nous a fallu appliquer plusieurs filtres afin de trouver notre grille plus facilement.

2.2.1.1 Grayscale

Premièrement nous appliquons un filtre de niveau de gris, pour cela, nous appliquons sur chaque pixel la formule suivante :

$pixel(x, y) = 0,3 \times pixel(x, y) \times r + 0,59 \times pixel(x, y) \times g + 0,11 \times pixel(x, y) \times b$

tel que r,g,b sont la puissance des couleurs appartenant à [0,255] ;

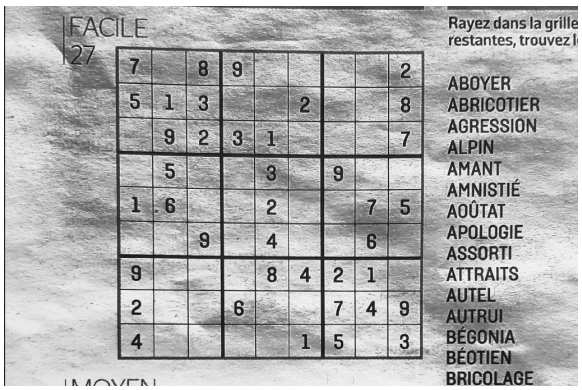

FIGURE 5 – Application du filtre grayscale.

2.2.1.2 Augmentation des contrastes

Ensuite, nous appliquons un filtre d'augmentation des contrastes. Les éléments parasites vont ressortir et vont pouvoir être plus facilement détectés et supprimés. Arbitrairement nous posons une valeur à 10, pour chaque pixel on itère un index de 0 à 10, et si le pixel appartient $[index \times (255/value), (index + 1) \times (255/value)]$ alors :

$$pixel = (index + 1) \times (255/value)$$

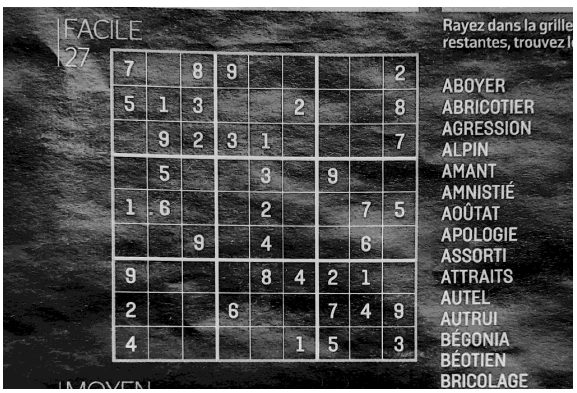

FIGURE 6 – Application du filtre d'augmentation des contrastes.

2.2.1.3 Egalisation de la luminosité

Après cela, nous normalisons l'éclairage global de l'image. Pour cela on applique sur chaque pixel la formule suivante :

$$pixel(x, y) = 255 - pixel(x, y)r \times (255/maxValue)$$

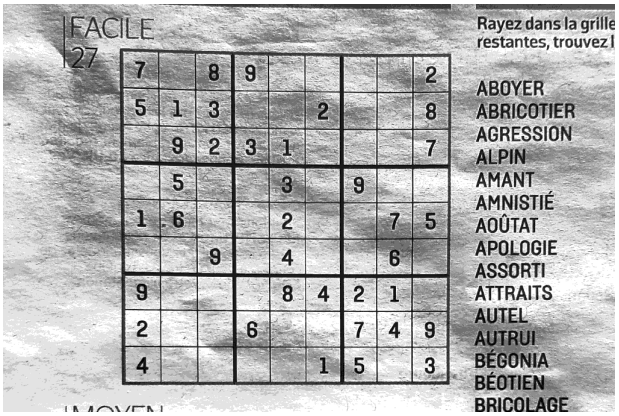

FIGURE 7 – Application du filtre d'égalisation de la luminosité.

2.2.1.4 Réduction des bruits

La prochaine étape est de réduire le bruit présent sur l'image. Pour cela, nous appliquons un filtre médian sur notre image. On récupère donc chacune des valeurs des voisins de chaque pixel grâce à une matrice. On applique la valeur médiane de la matrice au pixel courant.

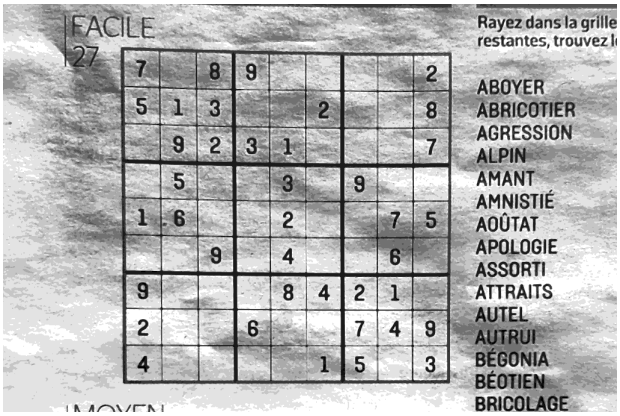

FIGURE 8 – Application du filtre de réduction du bruits.

2.2.1.5 Application du flou

Pour flouter notre image, on se sert du filtre Moyen. Ce filtre réduit les détails mineurs de notre image et brouille notre image et en particulier les contours des objets la composant. Pour appliquer ce filtre, on récupère chacun des voisins du pixel courant qu'on va multiplier par son équivalent dans une matrice de Pascal, la somme du tout sera la nouvelle valeur de notre pixel.

$$\text{filre de Pascal} : \frac{1}{16} \times \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

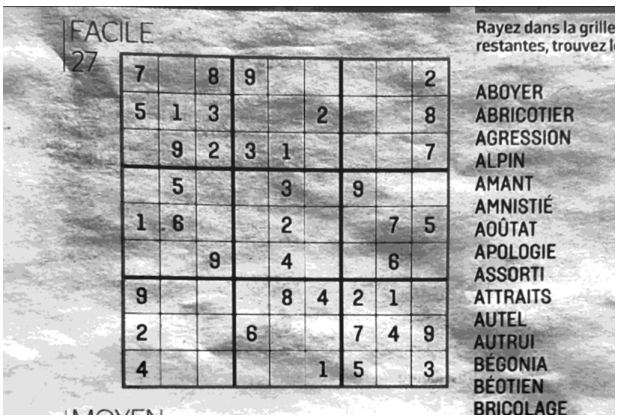

FIGURE 9 – Application du flou.

2.2.1.6 Binarisation de l'image

L'étape finale de notre prétraitement sur la couleur de notre image est la binarisation. Pour cela nous utilisons la technique du seuil adaptatif. On sépare l'image en plusieurs sous-images, on va calculer un seuil pour chaque image et on va déterminer la nouvelle valeur du pixel courant, soit blanc, soit noir.

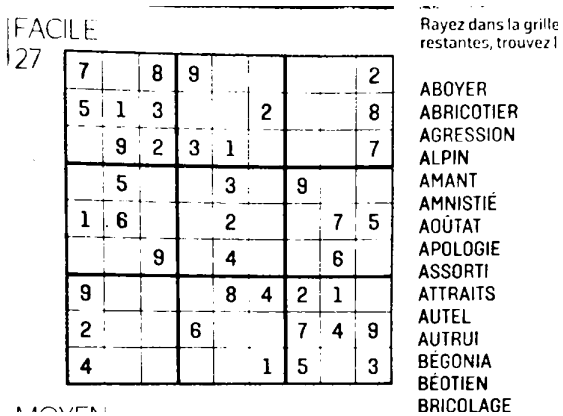

FIGURE 10 – Application de la binarisation.

2.2.2 Rotation de l'image

Pour faciliter notre détection de grille, on effectue une rotation de notre image, rendant les points importants de notre grille, comme ses coins, plus facilement trouvables.

2.2.2.1 Détection des bords

Pour détecter les bords de notre image, nous appliquons le filtre de Sobel qui va colorier tous les bords en noir, nous avons décidé d'inverser les couleurs de l'image en même temps pour optimiser nos étapes.

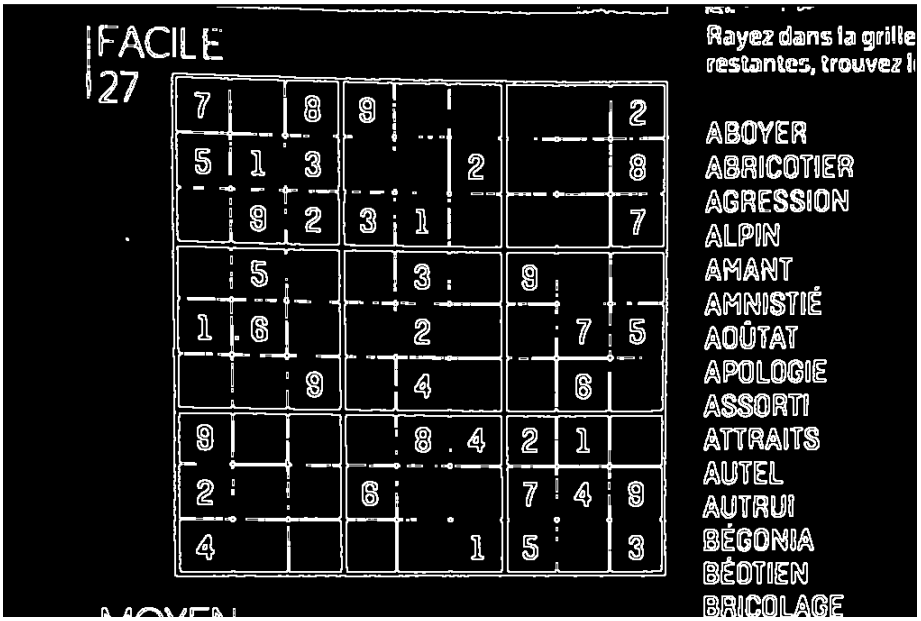

FIGURE 11 – Application du filtre de Sobel.

2.2.2.2 Transformée de Hough

Afin de détecter l'angle de la grille nous avons utilisé la méthode de la transformée de Hough. Celle-ci consiste à parcourir tous les pixels de l'image puis à incrémenter toutes les lignes qui passent par celui-ci dans une matrice représentant toutes les lignes. Nous utilisons la représentation polaire des lignes de la forme :

$$\rho = x \times cos(\theta) + y \times sin(\theta)$$

Avec $\theta \in [0, 360]$ et $\rho \in]0; \sqrt{w^2 + h^2}[$, h et w sont les dimensions de l'image. Ensuite nous pouvons récupérer toutes les lignes avec une valeur supérieure au seuil.

Nous simplifions le stockage de nos lignes en les convertissant de polaire en cartésien, pour cela nous gardons deux couples de coordonnées de points tel que :

$$y_1 = \rho \times sin(\theta) \text{ et } y_2 = k \times w + \rho \times sin(\theta) \text{ avec } k = -cos(\theta)/sin(\theta)$$
$$x_1 = 0 \text{ et } x_2 = w$$
et si $|y_1| > w \times h$ et $|y_2| > w \times h$ alors :
$$x_1 = \rho \times cos(\theta) \text{ et } x_2 = k \times h + \rho \times cos(\theta) \text{ avec } k = -sin(\theta)/cos(\theta)$$
$$y_1 = 0 \text{ et } y_2 = h$$

FIGURE 12 – Affichage des lignes avec une valeur supérieure au seuil.

Grâce à cette représentation de ligne nous pouvons les simplifier en faisant une moyenne des lignes à peu près égale, deux lignes sont à peu près égales si :

$$\text{pour } i \in [\![1\ ;\ 2]\!],\ |l1.y_i - l2.y_i| < s \text{ et } |l1.x_i - l2.x_i| < s$$

tel que est notre seuil et l1 et l2 deux lignes.

FIGURE 13 – Affichage des lignes simplifié.

Avec nos lignes restantes nous calculons leurs angles par rapport à l'axe de l'abscisse et gardons le plus récurrent pour la rotation.

2.2.2.3 Application de la rotation

Pour appliquer la rotation sur notre image, on crée une nouvelle surface faisant la diagonale de notre image, afin que durant notre traitement un minimum de pixels soit perdu. Pour cela, on calcule la moitié de la longueur des x de notre image, la moitié de la longueur des y et enfin la moitié de la longueur de la diagonale. Respectivement midx, midy et middle.

Ainsi pour chaque pixel(x,y) de notre image de base on lui trouve une nouvelle place dans la nouvelle image, avec les formules suivantes :

$$x = (x - midx) \times cos(\alpha) + (y - midy) \times sin(\alpha) + middle$$
$$y = -(x - midx) \times sin(\alpha) + (y - midy) \times cos(\alpha) + middle$$

2.2.3 Détection de la grille

2.2.3.1 Le blob détection

Pour trouver notre grille on utilise le blob détection. C'est un outil permettant la détection automatique d'un blob, soit le plus grand objet simple de notre image. Pour cela on va profiter de notre image binarisée avec ses bords qui ressortent. Pour chaque pixel blanc on détecte tous ses voisins de même couleur, on compte tous ses pixels de même couleur. L'élément contenant le plus de pixel est donc théoriquement notre grille. Evidemment on met un seuil minimum de pixels, si on passe en dessous de ce seuil on retourne toute l'image. Une fois le blob maximal détecté, on colorie en noir tous ceux qui ne sont pas atteignables à partir d'un pixel blanc du blob. On obtient donc le plus petit carré possible contenant notre élément considéré comme une grille.

2.2.3.2 Correction de la perspective

Pour corriger la perspective de notre image afin de faciliter la détection des nombres et la séparation des cases on utilise la transformation homographique.
On cherche les quatre coins de notre grille, pour cela on fait une distance de Manhattan pour chaque pixel blanc de notre image avec les coins de notre image. Les pixels les plus proches de ces coins, sont les coins de notre grille que l'on va transposer aux quatre coins de notre image afin de recadrer et corriger la perspective globale de notre grille. On a en paramètre P construit à partir des quatre coins de notre grille et on cherche h, P*h = (0,0,0,0,0,0,0,1) donc h = P-1.(0,0,0,0,0,0,0,1) Résoudre ce système nous permettra de trouver la nouvelle place de tous nos pixels et rendre notre image plus lisible.

$$P \cdot H = \begin{pmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x'_1 & y_1x'_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y'_1 & y_1y'_1 & y'_1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x'_2 & y_2x'_2 & x'_2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y'_2 & y_2y'_2 & y'_2 \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3x'_3 & y_3x'_3 & x'_3 \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3y'_3 & y_3y'_3 & y'_3 \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4x'_4 & y_4x'_4 & x'_4 \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4y'_4 & y_4y'_4 & y'_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

2.2.4 Découpage des cases et traitement

2.2.4.1 Découpage

Afin de lire le contenu de la grille de sudoku, il est nécessaire de la découper en cases individuelles. Pour cela nous parcourons la grille case par case et copions le contenu sur une image dont la surface fait la taille d'une case de la grille. Cette image est ensuite enregistrée dans un dossier output. Cela nous donne donc 81 fichiers. Nous devrons par la suite trier ces images pour garder uniquement les cases pleines.

2.2.4.2 Traitement des cases

Une fois cette étape accomplie nous devrons traiter les cases individuellement pour faciliter la lecture par le réseau de neurones.

2.3 Solveur de sudokus

La partie du solveur de sudokus est simple à implémenter mais essentielle au bon fonctionnement de l'application. Pour cela nous avons implémenté l'algorithme de backtracking et implémenté la gestion de fichiers texte pour suivre le format souhaité.

2.4 Interface graphique

2.4.1 Creation de l'interface

Pour l'interface graphique nous avons décidé d'utiliser la blibliothèque *gtk*. Cette dernière permet de faire la totalité des fonctionnalités dont nous avons besoin.

2.4.2 Fonctionnement

Pour l'instant l'interface se présente sous la forme d'une fenêtre simple avec le logo d'Aergia, une zone avec des boutons, 2 en l'état, et une zone de traitement avec un emplacement permettant d'afficher une image et un slider non utilisé. La fenêtre est dynamique et son contenu garde des proportions adaptées.
Le bouton load permet d'ouvrir l'explorateur de fichiers et de séléctionner une image qui apparaît alors à l'emplacement prévu.

FIGURE 14 – interface actuelle.

3 Conclusion

3.1 Bilan sur la première partie de développement

Nous sommes déjà à un stade avancé du projet car nous possédons toutes les fonctionnalités essentielles. En effet, il ne reste que très peu de tâches à effectuer concernant la partie traitement d'images. Les principales actions restantes sont des optimisations, et des réécritures de certaines fonctions.

Pour ce qui est du réseau de neurones, nous possédons un réseau fonctionnel, capable d'apprendre l'opération logique XOR. Les principaux éléments restants à effectuer sont la création d'un ensemble d'images d'entrainement de qualités afin que nous puissions entraîner notre réseau à reconnaître des chiffres. Il faut également effectuer plusieurs tests afin de trouver la combinaison optimale de couches et de neurones par couche pour notre réseau.

Le solveur de sudoku est quant à lui entièrement terminé et ne nécessite aucune modification.

Pour ce qui est des interfaces graphiques, il faut désormais faire le lien entre les différentes fonctions implémentées de manières isolées et l'interface graphique de telle sorte que l'on puisse utiliser toutes les fonctions implémentées, simplement en cliquant sur l'interface graphique.

La mise en relation des différentes parties du projet va d'ailleurs être la partie la plus délicate. Nous risquons de rencontrer des difficultés en voulant joindre des fonctions qui n'ont pas les mêmes paramètres et/ou type de retour.

3.2 Objectifs pour la prochaine soutenance

Nous souhaitons ajouter une structure de liste chainée pour optimiser certaines de nos listes qui n'ont pas besoin d'une implémentation statique. Nous devons également traiter les cases afin de pouvoir les utiliser dans le réseau de neurones. Et toutes nos différentes parties devront êtres liées afin de fonctionner entièrement sur l'interface graphique.

Le réseau de neurones devra fonctionner parfaitement et être capable de reconnaître tous les chiffres avec une très grande certitude. Enfin, nous souhaiterions réussir à résoudre des sudokus en héxadécimal.