OCR Rapport 1

Fait par les membres de STEL :

Ewan Schwaller, Lorenzo Taalba
Sacha Reggiani et Tristan Hette

Octobre 2022

Contents

1 Introduction

1.1 Présentation du groupe

1.1.1 Ewan SCHWALLER

Je suis intéressé par l'informatique depuis maintenant quelque années. C'est donc pour ça que j'ai décidé de rejoindre l'EPITA juste après l'obtention de mon bac. En plus de cet intérêt, j'ai pu également commencer a programmer et a coder durant mon temps libre au lycée. J'ai donc construit quelque base en C# mais aussi dans le codage de site (avec les basiques HTML, CSS et JavaScript). Ainsi je commence le projet en aillant jamais fait du C avant cette année, mais je compte bien découvrir tout ses aspects lors de ce projet !

1.1.2 Lorenzo TAALBA

Anciennement joueur de jeux vidéo à mes temps perdus, c'est dans cette discipline que j'ai découvert l'envie de coder, changer les règles de ces univers pour créer le mien. Ce passe-temps est devenu ma passion et s'est étendu à l'informatique en général. J'ai approfondi mes connaissances avec la spécialité informatique au lycée, qui m'a permis de mieux comprendre le fonctionnement de la programmation. Cependant, je n'ai jusqu'à présent touché qu'à des langages de haut niveau et donc le C est pour moi une découverte et un nouveau défi.

1.1.3 Sacha REGGIANI

Amateur de jeux vidéo depuis tout jeune, l'informatique m'intéresse depuis maintenant la 3ème, résultant d'une curiosité d'en apprendre plus sur la création de ces œuvres. j'ai suivi des tutoriels sur internet, j'ai appris en bidouillant chez moi et en faisant des recherches. J'ai ainsi appris les bases de quelques langages comme le Python et le HTML. Jusqu'ici, la programmation était un loisir fait principalement de découvertes et de pratiques dans lesquelles je me suis bien amusé. J'ai aussi fait des recherches sur l'art du game design, toujours dans la curiosité d'en apprendre sur le jeu vidéo et sa création.

1.1.4 Tristan HETTE

J'ai commencé à m'intéresser à l'informatique en 1ère, car je suis curieux et que je trouvais que l'informatique était vraiment un domaine passionnant et plus précisément le code. J'avais l'impression que rien n'était impossible avec. Je me suis donc former à quelques langages de programmations tels que le Python ou le c grâce à des ressources en lignes tels que OpenClassroom et SoloLearn. Je me suis également intéressé à d'autres aspects de l'informatique comme le développement de jeux vidéo car je suis un joueur depuis mon enfance et la Cyber Sécurité. Ce projet me permet de développer mes connaissances en C et également d'en apprendre plus sur les réseaux de neurones.

1.2 Répartition du projet

	Ewan	Lorenzo	Sacha	Tristan
Chargement de l'image				
Suppression des couleurs				
Rotation manuelle de l'image				
Détection de la grille				
Découpage de l'image				
Algorithme de résolution d'un sudoku				
Réseau de neurones				

1.3 Avancement du projet

Chargement de l'image	100%	Résolution du sudoku	100%
Suppression des couleurs	100%	Binarization des couleurs	20%
Traitement de l'image	30%	Interface	0%
Rotation manuelle de l'image	100%	Rotation automatique de l'image	100%
Détection de la grille	100%	Découpage de l'image	100%
Réseau de neurones (XOR)	100%	Réseau de neurones complet	10%
Reconnaissance des caractères	0%	Reconstruction de la grille	0%

2 Traitement de l'image

2.1 Couleur

Tout d'abord, nous avons passé l'image en noir et blanc. Pour se faire, on s'est servi du TP fait en programmation et on a appliqué la même méthode : on calcule la moyenne de chacune des valeurs RGB et on remplace sa couleur par cette valeur moyenne trouvée.

calcule: **grey = 0.3*red + 0.59*green + 0.11*blue**

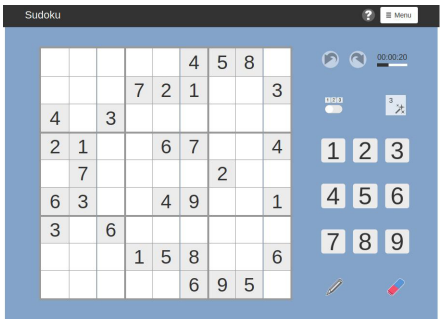

Figure 1: Avant

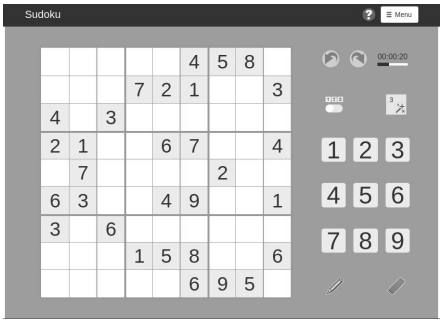

Figure 2: Après

2.2 Rotation de l'image

2.2.1 Manuelle

Pour effectuer une rotation à l'image avec un angle donné, on applique une matrice de rotation à l'image. En effet, cette méthode donne des rapides et bons résultats.

La matrice s'exprime avec les formules suivantes pour trouver les coordonnées (x', y') d'un pixel tourné, donc se trouvant sur l'image final, tourné sur un angle θ ayant comme origine le pixel situé en (x, y)

$$x' = x * cos(\theta) - y * sin(\theta)$$
$$y' = x * sin(\theta) + y * cos(\theta)$$

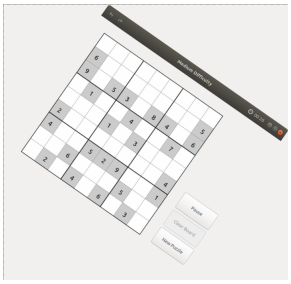

Figure 3: Avant

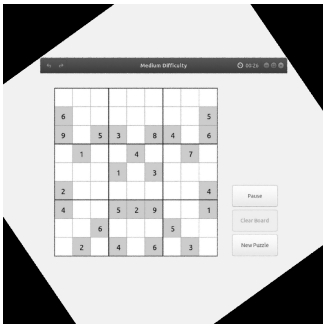

Figure 4: Après avoir effectué une rotation de 35° dans le sens anti-horaire

2.2.2 Automatique

Pour trouver l'angle de rotation a faire automatiquement, on s'est servi de l'algorithme détectant les ligne. En effet, une fois l'angle trouvé, nous avions plus qu'à appliquer la fonction dont on s'est servi pour la rotation manuelle.

Pour trouver l'angle de rotation, nous avons pris l'angle thêta le plus présent dans l'accumulateur de Hough puis appliquons sa valeur à l'image.

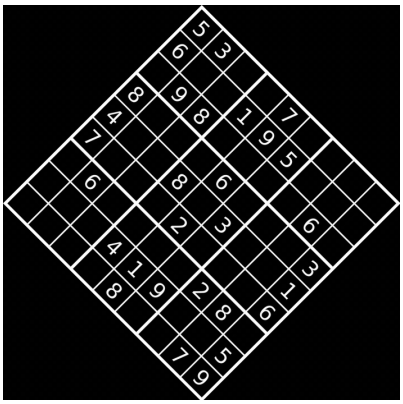

Figure 5: Avant

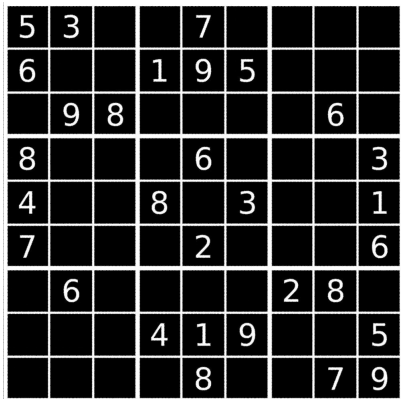

Figure 6: Après avoir effectué une rotation automatique

3 Découpage de l'image

Pour cette partie on a procédé ainsi :

- Détection de lignes
- Détection des carrées
- Détection du plus gros carré (donc la grille)
- Re-cadrage de l'image pour n'avoir plus que la grille
- Découpage de la grille

3.1 Détection de ligne

Grâce l'algorithme de Hough, ou la transformée de Hough, on a pu faire en sorte que notre projet détecte les lignes. Cet algorithme construit une courbe pour chaque pixels blancs dans son accumulateur (les edge points). Cet courbe s'ecrit de la forme :

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

avec θ allant de -90 deg a 90 deg.

Ainsi l'endroit où deux courbes se croisent dans l'accumulateur, représente potentiellement une ligne sur l'image.

Ensuite, on divise l'accumulateur en plusieurs parties et on retient, dans chacune de ces parties, la valeur maximale dépassant un seuil fixé pour l'instant à 90% de la valeur maximale dans tout l'accumulateur. Ces valeurs une fois obtenu, nous donnent les lignes de notre image.

Pour les trouver, on a besoin de deux points. Pour le premier, c'est facile, il suffit de transformer l'équation $\rho = x * cos(\theta) + y * sin(\theta)$ en cartésien. On a donc pour un couple (x_1, y_1)

$$x_1 = \rho * cos(\theta)$$
$$y_1 = \rho * sin(\theta)$$

Pour l'autre points il suffit projeter le point à une distance donnée d

$$x_2 = \rho * cos(\theta) + d * (-sin(\theta))$$
$$y_2 = \rho * sin(\theta) + d * cos(\theta)$$

Ainsi, on obtient une liste dont chacune des lignes contiennent 4 valeurs décrivant deux points.

3.2 Détection des carrées

Pour la détection des carrées, on a tout d'abord calculé les équations de chaque ligne notées sous la forme :

$$\begin{cases} y = ax + b & \text{si la ligne n'est pas vertical} \\ x = a & \text{sinon} \end{cases}$$

Chacune des lignes sont stockées dans une liste pour pouvoir y accéder plus tard.

Un fois les équations obtenues, on doit trouver quatre lignes perpendiculaires entre elles. Pour savoir si deux lignes sont perpendiculaires, il y a trois possibilités :

- **Si aucune des deux lignes sont verticales :** on multiplie leur coefficients directeurs et regarde si le résultat est égale à -1.
- **Si une des deux lignes sont verticales :** on vérifie que le coefficients de la ligne non verticale est égale a 0.
- **Si les deux lignes sont verticales :** elles ne sont pas perpendiculaires

Une fois avoir trouvé quatre lignes perpendiculaires, on a bien trouvé un carré !

3.3 Détection du plus gros carré

Pour savoir quel est le plus grand, on a décidé de calculer son périmètre. Si ce dernier est plus grand que le précédant plus grand, alors, on stocke la valeur de son périmètre et les index des lignes le composant selon l'ordre dans lequel les lignes sont stockées dans la liste.

Pour le calcul du périmètre, on calcule chaque intersection entre les lignes (donc chaque coins du carrées) puis on additionne chacune de distances entre les intersections.

3.4 Re-cadrage de l'image

Le re-cadrage ne consiste "qu'à" trouver quel points se situe où (En haut à droite, en bas à gauche, etc...). Pour se faire, on regarde selon leur valeur en abscisse et en ordonnée. Par exemple, celui qui aura la plus petite abscisse et la plus petite ordonnée sera en haut a gauche, etc...

Une fois l'ordre trouvé, il suffit de calculer la nouvelle longueur de l'image (c'est-à-dire la longueur de la grille) et sa nouvelle hauteur (donc la hauteur de la grille). Enfin on applique une fonction SDL pour replacer la grille correctement puis on ajuste ses dimensions selon les valeurs trouvées précédemment.

3.5 Découpage

Une fois que toutes les étapes précédentes ont été effectuées, il est très simple de découper la grille. Comme l'image à déjà été re-dimensionnée il n'y aucune détection de grille à faire, donc on découpe l'image en 81 carrés. Pour se faire on prend la longueur de l'image (largeur ou hauteur peu importe vu que c'est un carré), et on divise cette valeur par 9 pour obtenir la taille de chaque case du sudoku. Enfin on utilise une fonction de SDL pour copier la surface désirée dans le tableau qui contiendra toutes les surfaces.

4 Sudoku Solver

Le but du Sudoku Solver, doit être de pouvoir lire un fichier représentant un sudoku, le convertir dans un programme informatique, le résoudre, puis enfin enregistrer le résultat dans un autre fichier.

Tout d'abord nous avons commencé par faire le sudoku solver, ayant déjà fait un l'année précédente il était facile de le refaire en C.

4.1 Solver brut force

Le but d'une résolution par force brute est de tester toutes les possibilités jusqu'à ce que cela fonctionne. Dans le cas d'un sudoku solver cela se traduit par tester pour chaque case vide un chiffre, vérifier si la grille est toujours une grille valide, si oui passer à la case suivante, sinon tester avec le chiffre suivant et ainsi de suite jusqu'à la dernière case.

Cette fonction est très simple à mettre en place et est très courte, mais représente un problème quand à la performance du programme, celui-ci devant tester toutes les possibilités chaque fois, même celles qui sont évidentes de ne pas fonctionner.

4.2 Solver pré-traitement

Pour optimiser et contrer ce problème, nous avons revu notre implémentation du sudoku et ajouté une fonction de pré-traitement. Dans notre premier solver, notre grille était représenté par une liste simple de 81 cases, les 9 lignes * les 9 colonnes, où chaque case était un Uint (soit 0 pour une case vide, ou sinon un chiffre de 1 à 9). Dans la nouvelle implémentation, la grille est une liste de liste, une liste de 81 cases contenant chacune une liste de 10 éléments, dans ces 10 éléments, le premier représente le chiffre de la case (0 pour une case vide ou un chiffre de 1 à 9), les 9 autres étant les chiffres possibles, dans le cas d'une case libre (une liste de 1 à 9 où si la valeur du chiffre n'est pas valide par rapport aux règles du sudoku, elle est remise à 0). Autrement dit, une case:

- [0,1,2,3,4,5,6,7,8,9] veut dire que la case est vide et qu'on ne possède pas assez d'informations pour en déduire des chiffres impossibles.
- [4,0,0,0,0,0,0,0,0,0] que la case à été initié à 4
- [0,1,2,0,4,0,0,0,8,9] que la case est vide et que seules les chiffres 1, 2, 4, 8 et 9 peuvent être des valeurs possibles

Initialement les cases peuvent être soit dans le cas d'une case non-vide avec une valeur non-nulle au premier index et la suite à 0, soit pour une case vide avec 0 au début et les chiffres de 1 à 9 dans le reste de la liste. Par la suite nous appliquons une fonction pour supprimer les valeurs impossibles des cases libres. Le programme liste toutes les valeurs déjà rempli sur la ligne (et respectivement sur les colonnes et les zones de 3X3) et retires ces éléments de chacune des cases libres car une ligne (de même pour les colonnes et les zones de 3X3) ne peut contenir plusieurs fois le même chiffre.

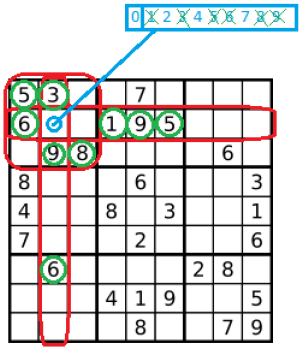

Figure 7: exemple de traitement pour une case

Grâce à cette méthode, nous évitons de tester à chaque fois des chiffres qui sont incontestablement impossibles, avec une simple boucle au tout début du programme. Cela a permis de réduire le temps de résolution de moitié par rapport au forçage brut.

Figure 8: comparaison avec la force brute

4.3 Solver optimisé

La dernière méthode rajoute par contre, le besoin d'une capacité de stockage plus grande, alors tout d'abord nous avons directement penser à changer le fait d'avoir une liste de liste de Uint qui prenait trop de place (81 * 10 * 4 bytes), par une liste de liste de char qui réduirait par 4 le besoin de stockage, un char ne prenant qu'un bytes par rapport aux 4 bytes du Uint.

Ensuite lors du pré-traitement, beaucoup de cases se retrouvent avec seulement qu'un chiffre possible, mais pour le programme la case est vide et doit à chaque récursion rechercher ce chiffre dans la suite de la liste. Pour améliorer ceci, lors du pré-traitement, si une case libre ne possède plus qu'une seule valeur possible, alors il transforme cette case libre en case remplie, permettant au programme de résolution de directement passer à la case suivante. Avec ces petites modifications, le solver a réduit son temps de résolution par 10 par rapport au solver par force brute.

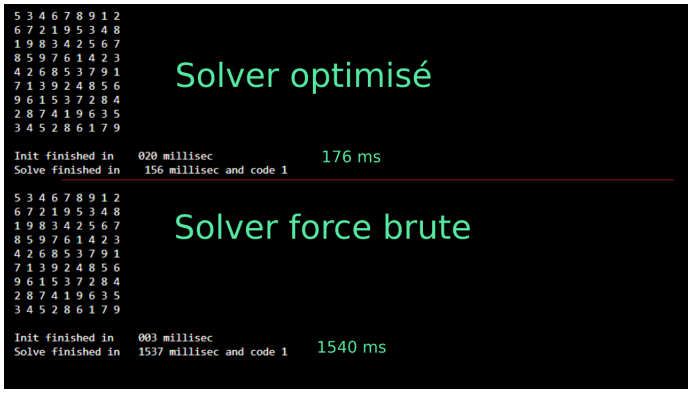

Figure 9: comparaison avec la force brute

5 Réseau de neurone

5.1 Architecture du réseau

Notre réseau de neurone a une architecture assez simple et classique. Nous disposons de plusieurs couches, la première est la couche contenant les entrées, la dernière contient le neurone de sortie et entre ces deux couches, nous avons les couches cachées qui contiennent nos neurones.

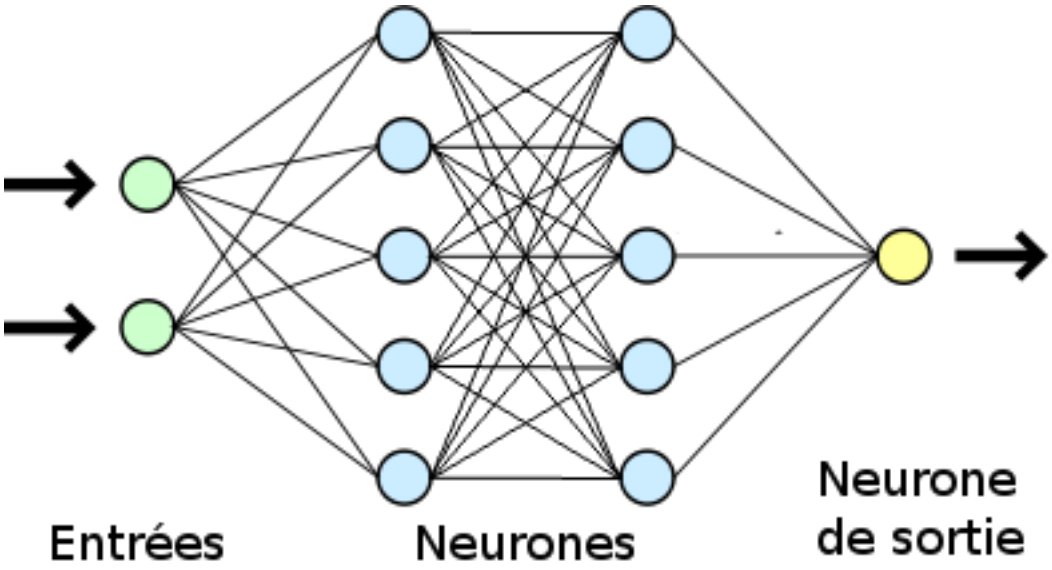

Figure 10: Architecture de notre réseau

5.2 Fonctionnement du réseau de neurone

Le fonctionnement de notre réseau de neurone est réparti en différentes parties.

Tout d'abord, nous initialisons les coefficients (poids) de notre réseau avec des valeurs aléatoires.

Ensuite nous avons une partie front-propagation qui permet de prédire à notre réseau une valeur de sortie grâce à la fonction d'activation sigmoïde (voir ci-dessous).

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

Figure 11: Fonction sigmoïde

Et enfin nous avons la back-propagation qui elle permet à notre réseau de corriger ses erreurs grâce à la dérivée de la fonction Sigmoïde (voir ci-dessous).

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Figure 12: Dérivée de la fonction sigmoïde

5.3 Preuve XOR

Pour le problème XOR, nous utilisons dans notre réseau de neurone 2 entrées, 1 couche cachées comprenant 2 neurones et 1 sortie (voir ci-dessous). Les nombres utilisés sont choisis par rapport à la table XOR (voir ci-dessous). Après de nombreuses itération, la sortie attendu dois converger vers 0 si le résultat attendu est 0 et dois converger vers 1 si le résultat attendu est 1.

X_1	X_2	$Y = X_1$ XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0

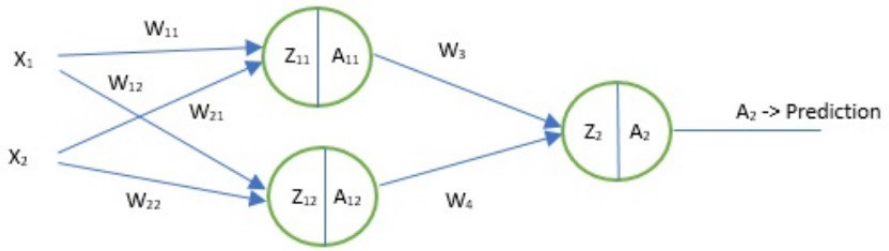

Figure 13: Table XOR et le réseau de neurone