OCR
Rapport 1

Julie Fiadino, Nicolas Dek, Roshan Jeyakumar, Oscar Chevalier

Prepa S3 – Octobre 2022

Sommaire

Introduction

Chez Julie & Co, notre but est de proposer le meilleur outil de résolution de sudoku qui soit. Notre équipe est constitué de Julie Fiadino, Nicolas Dek, Roshan Jeyakumar et Oscar Chevalier.

Répartition

Les 4 grandes parties du projet ont été réparties parmis chacun des membres. Cette répartition n'est pas non plus fixe car chacun peut aider une section en difficulté. Elle permet surtout d'associer un expert à chacune des sections qui sera capable d'en connaitre l'avancement et l'aspect technique de manière précise.

- Julie Fiadino : En charge de l'interface graphique (cheffe de projet)
- Oscar Chevalier : en charge de l'executable 'solver'
- Nicolas Dek : en charge du traitement d'image
- Roshan Jeyakumar : en charge du réseau de neurones

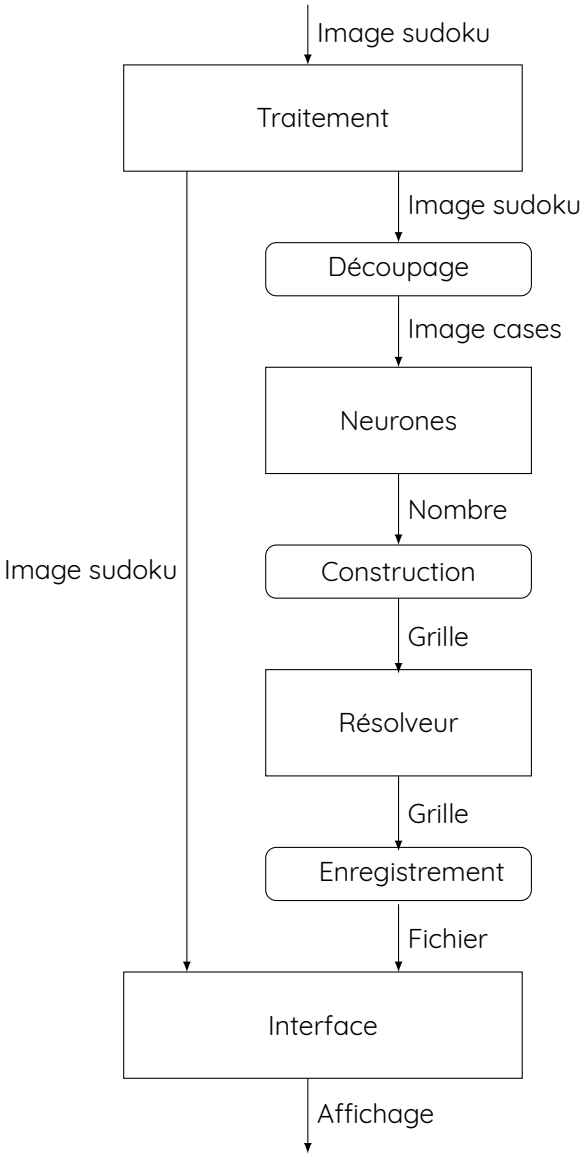
Pour cette première soutenance, certaines sections sont restées en arrière plan (notamment l'interface graphique). Voici donc la répartition du travail de chacun :

Tâche	Répartition
Résolveur (algo)	Oscar Chevalier
Résolveur (executable)	Oscar Chevalier
Traitement d'image (luminosité)	Julie Fiadino
Traitement d'image (détection de composants)	Nicolas Dek
Traitement d'image (Recadrage)	Nicolas Dek
Réseau de neurones	Roshan Jeyakumar

Architecture de projet

1.2.1 Architecture du programme

L'architecture hypothétique du programme est celle-ci :



Sur ce schéma, chaque bloc rectangulaire représente une section importante du programme. Les blocs arrondis représentent une étape de conversion.

Les flèches représentent le type de donnée entrant et sortant.

Ce diagramme correspond aux étapes que va suivre notre programme. Chacun des blocs importants étant indépendants (pour leur permettre d'être facilement modifiables) une étape de conversion est donc requise pour passer de l'un à l'autre.

L'interface est le seul bloc avec 2 entrées: l'image après traitement et le fichier après résolution, pour

obtenir une image de la grille résolue.

1.2.2 Architecture des fichiers

Le projet suit les conventions de structure d'un programme. Il contient :

- Un dossier `include`: contenant tous les fichiers .h du projet,
- Un dossier `src`: contenant tous les fichiers .c du projet,
- Un dossier `img`: avec toutes les images utilisées par le projet (réseau de neurones),
- Un dossier `tests`: avec toutes les fonctions de tests du projet,
- Un Makefile: permettant de compiler les différentes versions du code.

Pour faire écho à l'architecture du programme, le code pour chacune des sections importantes est séparé dans son propre sous dossier, que ce soit dans `include` ou dans `src`.

Progression globale

Le projet a progressé de manière régulière et dans les délais requis.

Le traitement d'image est maintenant presque complet à défaut de quelques amélioration ainsi que la rotation automatique.

Le solver est complet et fonctionnel, et la grande majorité du réseau de neurone a été faite.

Pour la prochaine soutenance, il terminer l'entrainement du réseau de neurones et assembler toutes les parties du programme ensemble, puis construire l'interface utilisateur.

Aspects Techniques

Dans cette section, nous allons aborder tous les aspects techniques de la construction du programme.

Résolveur

Le résolveur (solver) utilise la méthode classique de résolution de sudokus de retour sur trace (backtracking).

2.1.1 Algorithme de retour sur trace

Cet algorithme consiste à tester récursivement si une proposition de solution est valide ou non.
Dans le cas du sudoku, il faut créer une fonction récursive qui prend en paramètre la grille.

Dans un premier temps, cette fonction attribue à la première case vide (d'abord de gauche à droite puis de haut en bas) toutes les valeurs entre 1 et 9 (inclus) les unes après les autres.

Une fois une valeur initialisée sur la case choisie, il faut vérifier que le sudoku est toujours correct. Pour cela il existe des fonctions vérifiant que la ligne, le carré et la colonne sont corrects.

Si le sudoku est valide, on appelle cette même fonction récursive.
Si sur un appel de la fonction récursive il ne reste plus aucune case vide, alors cela signifie que le sudoku est terminé et il renvoit la valeur 1 (vrai).

Si l'algorithme n'a pas réussi à trouver de solution, cela signifie que le sudoku n'est pas résolvable, il renvoit alors la valeur 0 (faux).

2.1.2 Possible amélioration

Pour améliorer le temps de recherche du résolveur, on peut créer un tableau en 3 dimensions contenant l'ensemble des valeurs possibles sur chaque case permettant ainsi de ne pas essayer des nombres impossibles.

Si une case vide n'a qu'un nombre possible on peut donc attribuer ce nombre directement et mettre à jour toutes les cases victimes de ce changement.
Plus la grille se remplie, plus les possibilités sont faibles dans les cases vides.

Avec cette solution, on peut résoudre sans aucun ”retour sur trace” une grande partie des grilles et donc d’avoir une complexité beaucoup plus faible.

Cette amélioration est déjà partiellement programmée et sera utilisé pour la prochaine soutenance.

Traitement d’image

Le traitement d’image est constitué de plusieurs étapes, allant de l’ajustement de luminosité à la redimension de l’image.

2.2.1 Noir et blanc

Grayscale

Avant tout traitement, l’image est convertie en grayscale. Pour calculer la valeur de chaque pixel, notre algorithme prend la valeur des couleurs (r, g, b) la plus claire et l’utilise pour définir le niveau de gris.

Le but de cette méthode est de filtrer les couleurs qui peuvent être perçues comme foncées par un grayscale ’normal’ (aveec calcul de moyenne).

Par exemple, le bleu de l’image 4 va être perçu plus clair à cause de sa faible teneur en rouge.

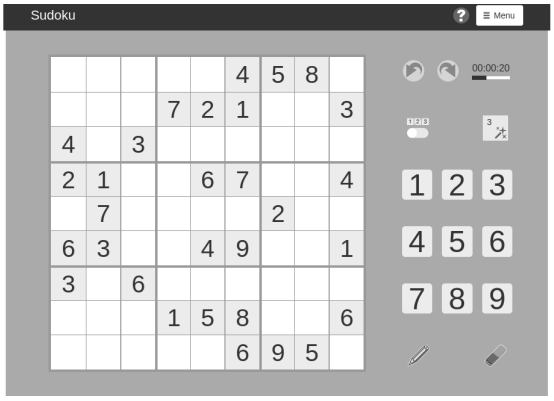


Figure 2.1: Image 4 avec grayscale ’normal’

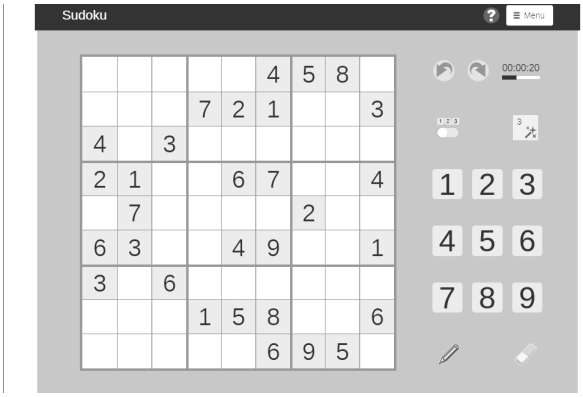


Figure 2.2: Image 4 avec grayscale maximal

Noir et Blanc

Pour la mise en noir et blanc, nous avons besoin au préalable d'une image déjà convertie en grayscale.

D'abord nous calculons la moyenne du niveau de gris dans l'ensemble des pixels de l'image ce qui va donner notre seuil.

Ensuite pour chaque pixels de notre image on regarde si il est supérieur à notre seuil, si c'est le cas on remplace notre pixel par un pixel noir et dans le cas contraire on le remplace par un pixel blanc.

2.2.2 Ajustement d'image

Rotation

Le calcul de la rotation se fait à l'aide d'une matrice de rotation.

Cette matrice permet d'obtenir les coordonnées suivantes : $x' = \sin(angle) * x + \cos(angle) * y$ et $y' = \cos(angle) * x - \sin(angle) * y$

Il suffit donc d'assigner la couleur actuelle aux nouvelles coordonnées.

Pour ajuster le niveau de luminosité de l'image, notre programme applique une 'closing operation'.

Cette opération va effectuer une dilation puis une érosion sur l'image.

Figure 2.3: Closing operation sur l'image 1

Le but de ces opérations va être d'obscurer les lignes du sudoku.

Dilation et Erosion

Ces 2 opérations sont appelées 'opérations morphologiques'.

Elles prennent toutes les 2 un élément structurel qui va permettre de déterminer la nouvelle couleur des pixels de l'image.

Dans le cas de la dilation, on place l'élément structurel au centre de chacun des pixels, et on lui assigne la couleur la plus claire contenue dans l'élément.

Pour l'érosion, on y assigne la couleur la plus sombre.

Figure 2.4: Sudoku en grayscale

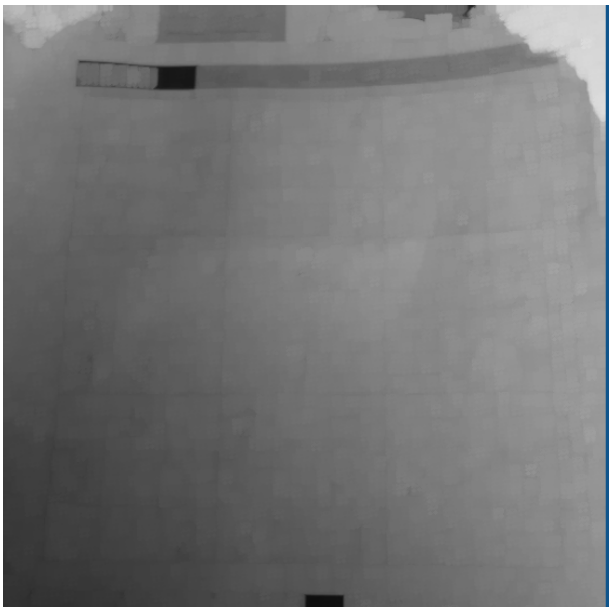


Figure 2.5: Dilation sur le sudoku

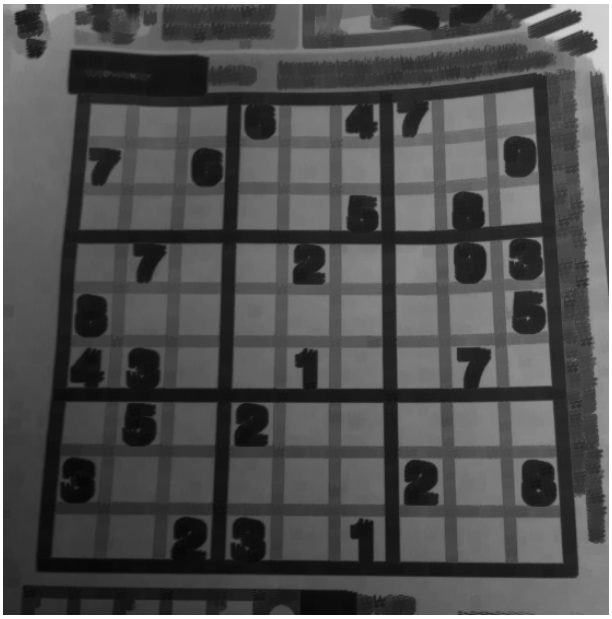


Figure 2.6: Erosion sur le sudoku

La dilation va donc permettre de faire ressortir le fond, et l'érosion les lignes noires.

Pour faire une 'closing operation', il suffit d'effectuer sur une image une dilation puis une érosion.

Ajustement

En divisant l'image de base par celle obtenue, les tons trop sombres ou trop clairs vont donc s'annuler, ne laissant que les chiffres et les lignes de la grille visible.

Figure 2.7: Sudoku après ajustement

Cette technique est efficace car elle permet d'obtenir un résultat satisfaisant pour chaque type d'image. Cela évite donc de détecter la luminosité de l'image pour choisir le type d'ajustement.

De plus, elle permet d'ajuster des images avec un éclairage pas toujours uniforme. C'est par exemple le cas de l'image 7.

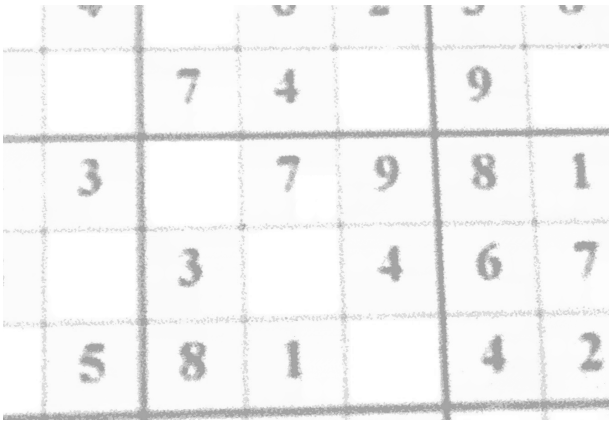


Figure 2.8: Image 7 après ajustement

Cette méthode est en revanche extrèmement couteuse, avec $width^2/100 \times height$ opérations. Les images trop grandes doivent donc être rétrécies au préalable.

2.2.3 Détection de la grille

Détection de composants

Pour la détection des composants dans une image on utilise l'algorithme Connected-component labeling, cela va nous permettre de pouvoir facilement identifier la grille du sudoku dans une image.

Pour faire fonctionner l'algorithme il nous faut une image binarisé (c'est à dire une image avec seulement 2 couleurs).

Voici le fonctionnement de l'algorithme:

On met le compteur de label a 1.

On parcours chaque pixel d'une image de gauche vers la droite et de haut en bas.

Si le pixel de gauche contient un label alors le pixel courant récupère son label.

Si le pixel de gauche n'a pas de label mais que le label de haut en a un, alors on récupère le label de celui du haut.

Sinon, si le pixel de gauche et de haut n'ont pas de label on donne au pixel courant un label et on incrémente le compteur de label.

Ensuite après avoir finis de parcourir l'image, on reparcourt l'image une deuxième fois et cette fois pour chaque pixel on remplace son label par le plus petit label de ses voisins.

Récupération des lignes

Les lignes sont récupérées en dérivant l'image en x et en y.

Cette opération s'effectue grâce à une convolution, démarche visant à passer une matrice appelée 'kernel' sur chacun des pixels et à calculer la nouvelle couleur en fonction des poids inscrits.

Pour dériver l'image, le programme utilise un kernel de sobel :

$$K_{sobel} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

La dérivée en x de l'image revient donc à faire une convolution de l'image I par le kernel. Pour la dérivée en y, on utilise la transposée du kernel.

Cette matrice permet seulement de faire des dérivées de premier degré. Or, pour plus de précision, il est préférable de passer par une dérivée de second degré. La convolution étant associative, on peut donc faire une convolution sur le kernel avec le kernel.

On obtient alors le kernel suivant :

$$K_{sobel} * K_{sobel} = \begin{pmatrix} 1 & 0 & -2 & 0 & 1 \\ 4 & 0 & -8 & 0 & 4 \\ 6 & 0 & -12 & 0 & 6 \\ 4 & 0 & -8 & 0 & 4 \\ 1 & 0 & -2 & 0 & 1 \end{pmatrix}$$

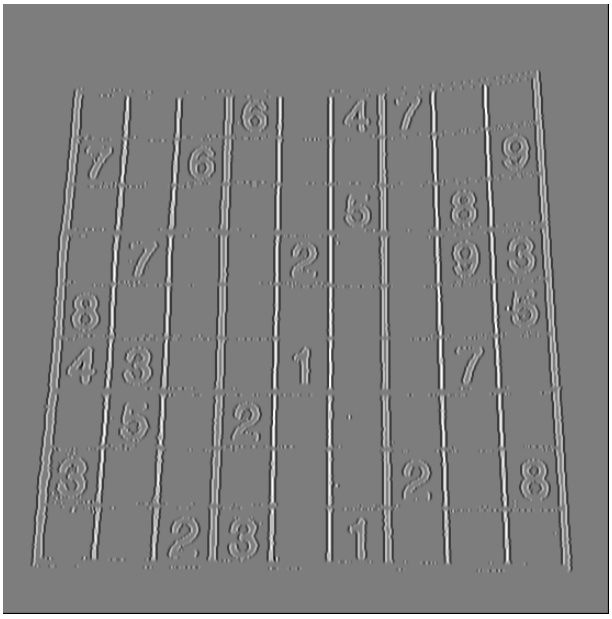


Figure 2.9: Dérivée en x du sudoku

Il suffit ensuite de binariser l'image et de lui appliquer une dilation pour rendre les lignes plus visibles.

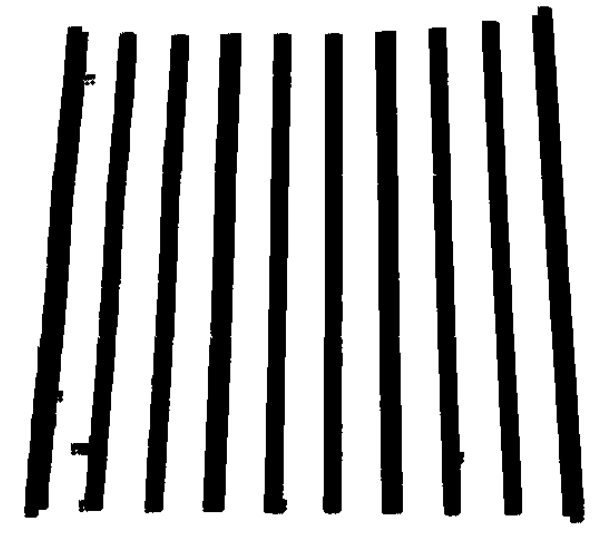


Figure 2.10: Dérivée en x après binarisation et dilation

En appliquant un 'et binaire' entre les 2 dérivées, on obtient donc les intersections des lignes c'est-à-dire tous les coins des cases de la grille.

2.2.4 Interpolation

Après traitement, le sudoku est donc isolé et enregistré dans une nouvelle image de taille fixe qui sera ensuite découpée et envoyée au réseau de neurone.

Pour cela, le programme utilise de l'interpolation linéaire. L'image source est considérée comme une texture qui va être appliquée sur le carré que représente la nouvelle image.

Cette méthode va donc permettre d'automatiquement étirer l'image pour qu'elle rentre dans les dimensions demandées de manière uniforme.

Réseau de neurones

2.3.1 Le XOR

Pour la première soutenance, il nous est demandé de faire un réseau de neronnes pouvant apprendre à simuler le principe de la porte XOR, afin de comprendre le fonctionnement du réseau dans un cas plus simple que la reconnaissance de chiffre.

2.3.2 L'utilisation de matrices

À travers les recherches, la méthode la plus simple est d'utiliser des matrices qui représentent les "Hidden layer". Cela permet de modifier les biais et les poids en parallèle sans passer dans chaque noeud. Pour cela nous avons créer une structure "Matrix" qui permet de simuler le comportement d'une matrice en parcourant les adresses mémoire.

2.3.3 Le réseau

Le réseau possède 2 entrées, 5 neurones intermédiaires et une sortie. L'apprentissage se fait à travers 100,000 "epochs"(iteration) qui corrigent les poids et les biais des neurones à la fin d'une "epoch". À la fin de cet apprentissage, on teste les réseaux afin de vérifier les résultats

2.3.4 La reconnaissance de chiffre

La reconnaissance de chiffre se fera de la même manière mais le réseau aura chaque pixel en entree et 10 sorties. En effet, le nombre de neuronnes intermédiaires sera beaucoup plus élevés afin de pouvoir apprendre la complexité lié au résultat attendu.

Sauvegarde et chargement de fichiers

2.4.1 Sauvegarde de la grille

La sauvegarde la grille se fait dans un fichier respectant le format de fichier décrit dans le cahier des charges.
Les fichiers peuvent être sauvegardé suivant un chemin avec le nom que l'on souhaite.

2.4.2 Chargement de la grille

Le chargement de la grille se fait suivant le format de fichier décrit dans le cahier des charges. Si les caractères obtenus ne sont pas attendus, le programme s'arrête.

2.4.3 Sauvegarde des poids du réseau de neurones

La sauvegarde des poids et des biais se fait dans un fichier qui indique les dimension de la matrice puis les

Environnement de Travail

Notre travail est principalement effectué sur un dépot GitHub, qui est ensuite synchronisé avec le dépot GitLab de l'école. Cette démarche apporte de nombreux avantages.

Automatisation de tests

Chaque branche du projet est testée avant de pouvoir être fusionnée. GitHub va compiler l'executable du dossier tests et lancer le programme.

Ce programme est construit afin que chaque test non concluant renvoie une `EXIT_FAILURE`. Ainsi, en cas de code de retour non valide, la branche ne sera pas déployable.

Intégration Continue

Toutes ces opérations sont possibles grâce à l'intégration continue. Elle est programmable grâce à un fichier yaml décrivant sont comportement.

La CI effectue 3 étapes :

1. Elle compile l'executable et vérifie qu'il ne contient pas de warnings,
2. Elle lance les tests et vérifie qu'il n'y a pas d'erreurs,
3. Elle déploie le code du repo GitHub vers GitLab.

Chacune de ces étapes ne peut être effectuée si les étapes précédentes n'ont pas été validées. Ainsi, le code sur le repo GitLab est donc toujours valide et fonctionnel.

Makefile

Le Makefile est utilisé pour compiler ces différentes versions. Il comporte 5 entrées différentes :

1. Une entrée `main`, qui va compiler l'executable principal,

2. Une entrée `debug`, qui compile l'executable principal avec des options de debugging,
3. Une entrée `solver`, qui compile l'executable solver,
4. Une entrée `test`, qui compile l'executable de test (avec des options de debugging),
5. Une entrée `clean`, qui supprime tous les fichiers .o et executables.

Les options de debugging, autre que le classique `-g`, contiennent la définitions de variables telles que DEBUG. Ces variables sont utilisées dans le programme pour définir des fonctions dites de debug. Ces fonctions sont précédées de `DEBUG_` et entourées d'instructions `#if DEBUG ... #endif`. Elles ne seront donc pas compilées lors de la création de l'executable principal.