Rapport du projet OCR

A Pouception Creation

Hugo Baptista
Romain Doulaud
Camille Chapelle
Félix Coste

2022 - 2023

Contents

1 Introduction **4**
- **1.1 Les usages dans le secteur professionnel** 4
- **1.2 Répartition et avancement des tâches** 5

2 Le Prétraitement de l'image **6**
- **2.1 Méthode de binarisation** 6
  - **2.1.1 L'algorithme d'Otsu** 8
  - **2.1.2 Première implémentation de l'algorithme d'Otsu** 8
  - **2.1.3 Implémentation des variantes de l'algorithme d'Otsu** 10
- **2.2 Recherches et autres méthodes** 12

3 La Découpe de la grille **13**
- **3.1 Détection des lignes** 13
  - **3.1.1 Le transformé de Hough** 13
  - **3.1.2 La mise en application** 14
- **3.2 Rotation de l'image** 17
- **3.3 Filtrage des lignes** 17
- **3.4 Détection des intersections** 19
- **3.5 Découpe des cases** 20

4 La reconnaissance des chiffres **21**
- **4.1 Preuve de concept** 22
- **4.2 Les matrices** 22
- **4.3 Les maths qui se cachent derrière** 22
  - **4.3.1 Forward Propagation** 23
  - **4.3.2 Backward Propagation** 24
- **4.4 Vérifier les performances** 25
- **4.5 Le dataset d'entraînement** 26
- **4.6 Les caractéristiques de notre réseau** 26
- **4.7 Enregistrer le réseau** 27

5 La résolution du Sudoku **29**

6 L'interface Utilisateur (UI) 31
6.1 La fenêtre d'ouverture 31
6.2 La fenêtre de rotation manuelle 33
6.3 La fenêtre de binarisation 34
6.4 La fenêtre de détection de ligne 36
6.5 La fenêtre de résolution et sauvegarde 37

7 Organisation, Équipe et Expérience 38

8 Conclusion 40

1 Introduction

Ce rapport de soutenance est le résultat de plusieurs mois de travail sur l'implémentation de divers algorithmes permettant la résolution d'une grille de Sudoku à partir d'une image.

1.1 Les usages dans le secteur professionnel

Avant de plonger dan notre projet, il est important de rappeler que l'OCR est une technologie très utilisé dans le secteur professionnels. Il y a déjà de nombreuses implémentations qui existent comme par exemple:

- Google Traduction, pour traduire du texte à partir de la caméra
- Apple iOS, il est possible sur tous les iPhones de copier du texte depuis n'importe quelle image ou vidéo
- Les centres de tri La Poste, pour détecter des adresses manuscrites
- Les applications demandant une vérification d'identité, qui extrait toutes les informations à partir d'un scan de passeport ou d'une CNI.

Plus globalement, l'OCR est utilisé pour éviter à un utilisateur de retaper une quantité conséquente d'informations. Cela se traduit bien à notre projet, en effet nous n'allons pas demander à l'utilisateur de recopier les 81 cases de sa grille de sudoku. Nous allons plutôt lui permettre de prendre une photo de sa grille, qui subira divers traitement et à la fin du processus, il aura une image de la grille résolu.

1.2 Répartition et avancement des tâches

Table 1: Répartition des tâches

	Hugo	Romain	Camille	Felix
Pré-traitement de l'image				X
Découpe de la grille		X		
Reconnaissance de chiffre	X			
Résolution du Sudoku			X	
UI			X	

Table 2: Planification de l'avancement

	Soutenance 1	Soutenance finale
Binarisation	80%	100%
Découpe de la grille	80%	100%
Réseau de neurones	50%	100%
Résolution du Sudoku	100%	100%
UI	0%	100%

Comme mentionné par le tableau ci-dessus, le projet à atteint un état stable dans lequel un utilisateur peut télécharger notre application, ouvrir une image, ajuster manuellement s'il le souhaite et finalement voir la grille résolue. Alors dans chaque partie, nous allons détailler les étapes de développement, ainsi que les retours d'expériences des personnes en charge de la partie.

2 Le Prétraitement de l'image

La première étape du projet est d'effectuer un prétraitement sur l'image afin que celle-ci soit traitable pour les étapes futures.

2.1 Méthode de binarisation

La binarisation de l'image consiste à supprimer les couleurs afin de ne garder que le noir et le blanc dans une image.

Tout d'abord, la première étape est de passer l'image en grayscale, c'est-à-dire uniquement en tons de gris. Pour ce faire, nous avons utilisé le code du tp 6 d'Informatique pratique, nous apprenant à nous servir de SDL et à grayscale une image.

Cette méthode de grayscale consiste à récupérer la composante rgb de chaque pixel séparément en Uint8, et d'y appliquer la formule : $0.3 * r + 0.59 * g + 0.11 * b$ (r représentant la composante rouge, g la verte et b la bleue). Les coefficients représentent les l'impacte lumineux de chaque couleur. Cette formule nous renvoyait alors un Uint32, servant de nouvelle composante rgb au pixel. On l'appliquait alors à l'image et le grayscale était créé.

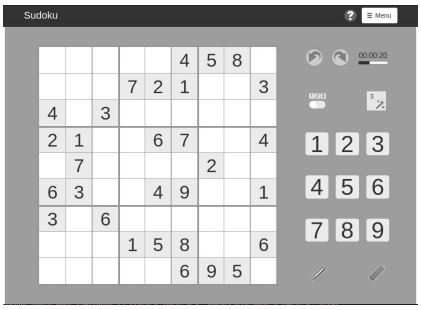


Figure 1: Résultat du grayscale sur l'image 3

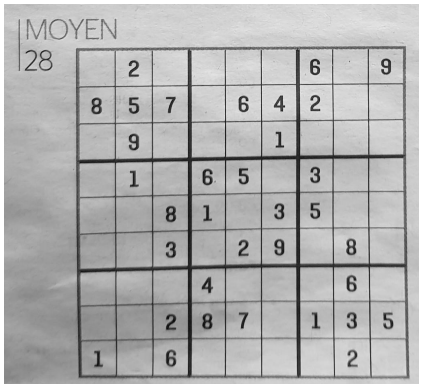


Figure 2: Résultat du grayscale sur l'image 2

Une fois le grayscale réalisé, il fallait transformer l'image en pixels noirs et blancs uniquement.

Afin de réaliser une binarisation, il nous faut une valeur seuil. Ce seuil (threshold) sera le point de passage entre un pixel blanc et un pixel noir sur l'image, la fameuse limite cherchée afin que l'image puisse rester exploitable.

Nous avons d'abord essayé de réaliser une binarisation avec la valeur moyenne des pixels, la valeur maximum étant 255, puis nous divisions par 2 afin de récupérer le premier seuil. Cette méthode s'est toutefois avérée inutile, car l'image restait complètement inexploitable.

La deuxième méthode essayée fut d'implémenter cette moyenne de couleur des pixels de manière locale. Nous divisions alors l'image en 5 sur la longueur et 5 sur la largeur et faisions la moyenne du niveau de gris du carré. Puis nous colorions chaque pixel dont la valeur de gris est inférieure à ce seuil en noir, et les autres en blanc. Faute de Segfault récurrents et de grandes difficultés à implémenter cette méthode dû au grand nombre de pixels dans les images, il a été décidé que cette méthode sera optimisée et réutilisée par la suite, afin d'obtenir un meilleur résultat sur l'algorithme d'Otsu.

Enfin, après de nombreuses recherches,nous avons trouvé 2 algorithmes de binarisation d'image : la méthode de Sauvola et celle d'Otsu. Nous avons décidé d'implémenter l'algorithme d'Otsu car cette méthode semblait plus facile.

2.1.1 L'algorithme d'Otsu

La méthode d'Otsu consiste à trouver le seuil idéal afin de séparer binariser l'image. Il faut donc commencer par récuperer les valeurs de chaque pixel d'une image grayscale, puis de construire un histogramme à partir de ces valeurs. À partir de cet histogramme, il faut trouver la valeur idéale du seuil. Cette valeur peut être trouvée lorsque la variance de l'histogramme calculée à partir de cette formule $\sigma(t) = \omega1(t)\omega2(t)(\mu1(t) - \mu2(t))$ est maximale. $\omega1$ correspond à l'espérance de toutes les valeurs, c'est-à-dire la somme de chaque valeur (de 0 à 255) multipliée par son nombre d'occurences dans l'histogramme. $\omega2$ la somme de toutes les espérances dans les valeurs de 0 à t. $\mu1$ correspond quant à elle à la valeur moyenne du cluster 1 (des valeurs de 0 à t), c'est à dire $\omega1/n1$ où n1 correspond à la somme de toutes les valeurs de l'histogramme de 0 à t. $\mu2$ correspond donc à la valeur moyenne du cluster 2 (des valeurs de t à 255), c'est-à-dire $(\omega1 - \omega2)/n2$ où n2 correspond à la somme des valeurs de l'histogramme allant de t à 255. Pour chaque valeur de t allant de 0 à 255, nous effectuons donc le calcul de cette variance, et la valeur maximale est choisie.

2.1.2 Première implémentation de l'algorithme d'Otsu

Durant l'implémentation de l'algorithme d'Otsu, nous avons rencontré de nombreux problèmes. Le premier problème rencontré fût de comprendre et d'appliquer les maths. En effet, la documentation sur l'algorithme d'Otsu, même si elle est présente, est mal expliquée et détaillée. Nous avons passé plusieurs heures sur des dizaines de sites avant de réussir à comprendre l'algorithme comme il faut.

Une fois implémenté, nous avons rencontré de nombreux soucis dans l'algorithme, le premier étant que la valeur seuil trouvée n'était pas du tout efficace et semblait être loin du résultat attendu. En effet, l'image se retrouvait en permanence toute noire ou toute blanche. Cette erreur était dûe au mauvais cast des variables. En effet, lorsque l'on travaille sur une image de plusieurs milliers de pixels, chaque nombre derrière la virgule est important. Il fallut alors caster tous les int en float, afin d'affiner la précision. Une fois cette manoeuvre réalisée, les résultats étaient bien plus concluants mais

encore pas parfaits.

Le deuxième problème rencontré était que l'algorithme trouvait uniquement deux valeurs seuils, les deux mêmes pour chaque image : 147 et 253. Nous pensions alors d'abord qu'il s'agissait d'un dépassement de valeur. Nous avons donc essayé de recaster les variables en double ou long afin de permettre aux variables d'atteindre des valeurs plus grandes. Le problème ne venait pas de là. L'erreur était tout simplement que l'histogramme était initialisé à l'aide d'un malloc et non d'un calloc. Les valeurs étaient donc pas réinitialisées et l'histogramme était faussé.

Après la résolution de ces erreurs, l'algorithme était implémenté, mais ses résultats étaient pas optimaux.

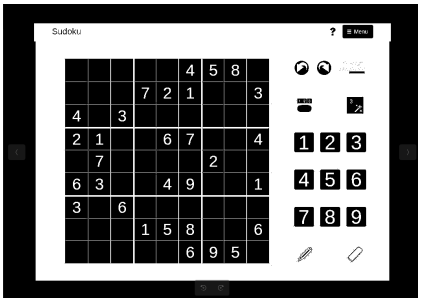


Figure 3: Résultat de la binarisation de l'image 3

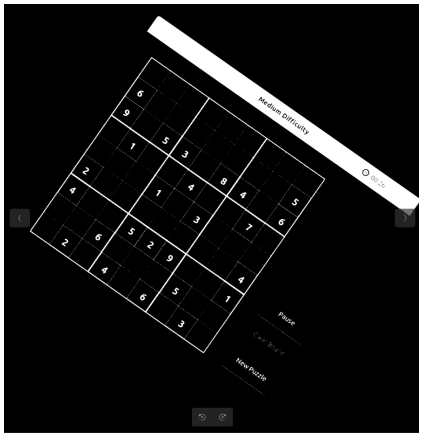


Figure 4: Résultat de la binarisation sur l'image 5

2.1.3 Implémentation des variantes de l'algorithme d'Otsu

Durant le temps passé entre la 1ère soutenance et la soutenance finale, nous sommes donc revenus à une idée infructueuse qui était d'implémenter l'algorithme d'Otsu en local afin d'améliorer ses résultats. Les premiers obstacles rencontrés furent les nombreux segmentation fault. En effet, implémenter un algorithme en version locale implique de manier des arrays et de nombreux pointeurs. En effet, il y avait de nombreux dépassements, et les calculs ne se limitaient pas aux carrés. De plus, une erreur d'inattention qui nous a bloqué pendant un bon moment était le simple fait d'inverser les composantes de hauteur et de largeur de l'image. Il y avait donc sans arrêt des segfault pour des raisons parfaitement évitables. Une fois la première version locale d'Otsu implémentée, il fallait trouver la bonne valeur de découpe (la racine du nombre de carrés à binariser) pour laquelle l'algorithme obtiendrait des résultats optimaux. Ainsi, les valeurs choisies ont été 5, 10 et 100. La valeur 1 correspond à l'algorithme d'Otsu classique, réalisé sur la globalité de l'image, et en fonction de la quantité de détails et de bruit, l'utilisateur peut choisir d'affiner la précision de l'algorithme.

L'implémentation en local d'Otsu permet de réduire le bruit ainsi que de traiter l'image sans y appliquer de filtre, et cette implémentation le réalise bien.

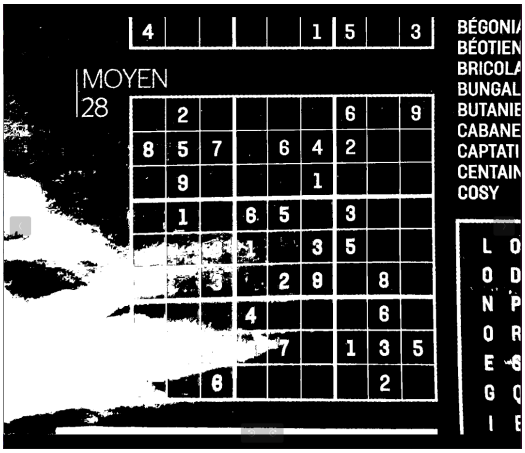


Figure 5: Algorithme d'Otsu global

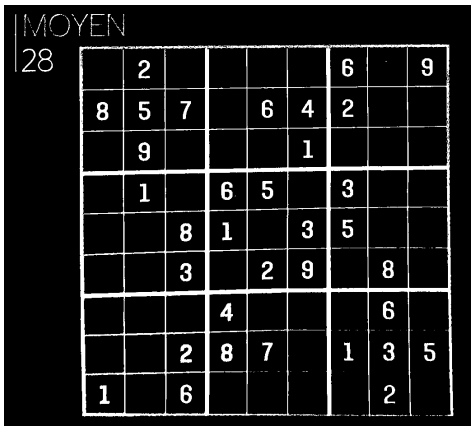


Figure 6: Algorithme d'Otsu local

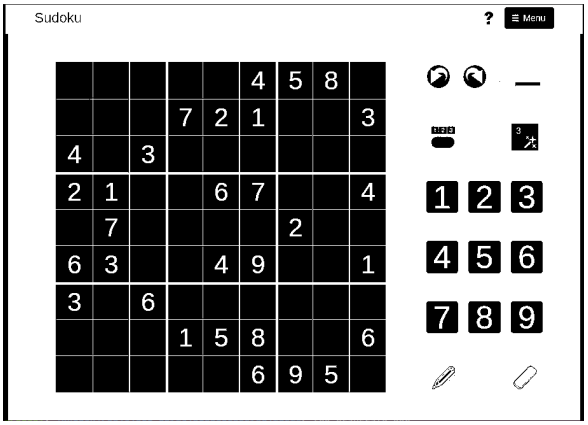


Figure 7: Binarisation finie

2.2 Recherches et autres méthodes

Lors des recherches sur les algorithmes et principe de binarisation, nous avons trouvé plusieurs méthodes alternatives à Otsu :

- Sauvola, cette algorithme consiste à trouvé des seuils locaux comme l'implémentation local d'Otsu.
- Sobel, ce filtre ne rends pas directement une image binarisé mais détecte très bien les variations de couleur ce qui rends au final une image en nuance de gris représentant toutes les contours et lignes d'une image. Cette méthode est très utilisé pour détecter les lignes sur la route et les silhouette de personnes/objets.
- Niblack, c'est aussi un algorithme de seuil mais qui est plus adapté à la binarisation d'image composé de texte

Pour notre cas d'usage l'algorithme d'Otsu et ses déclinaisons locales semblent les plus adaptés pour binarisé des images de grilles de sudoku.

Nous avons aussi tenté une première binarisation pour la détection de lignes puis une seconde passe sur les images découpé cependant cela n'as pas abouti à des résultats concluant nous n'avons donc pas poursuivi cette piste.

3 La Découpe de la grille

Une fois l'image binarisée il faut détecter la grille du sudoku afin de découper les 81 cases du jeu et les envoyer au réseau de neurones qui va pouvoir trouver les chiffres correspondants.

3.1 Détection des lignes

3.1.1 Le transformé de Hough

Afin de détecter la grille du sudoku, nous avons utilisé et réimplementé le transformé de Hough.

Cette méthode utilise le fait qu'une droite peut être représentée de plusieurs façons différentes: la première étant la représentation la plus commune en coordonnées cartésiennes: y=ax+b (figure 8). La seconde étant en coordonnées polaires: $\rho = x * cos(\theta) + y * sin(\theta)$ (figure 9). Grâce à cela, nous pouvons ainsi créer deux repères géométriques pour représenter la même chose. Le premier est un repère cartésien avec les conventions habituelles: x en abscisse et y en ordonnée. Tandis que le second est un repère en coordonnées polaires avec en abscisse: un angle thêta (θ), et en ordonnée: une distance rhô (ρ).

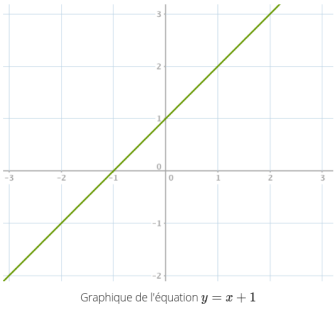


Figure 8: Droite cartésienne

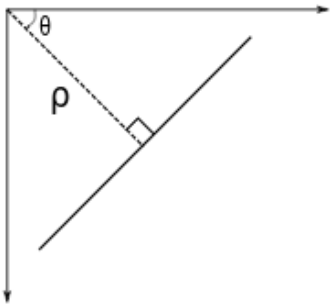


Figure 9: Droite polaire

Ces deux repères sont très liés l'un l'autre. En effet, un point dans le repère cartésien représente une courbe sinusoïdale dans le repère polaire. Réciproquement, un point dans le polaire représente une droite dans le cartésien.

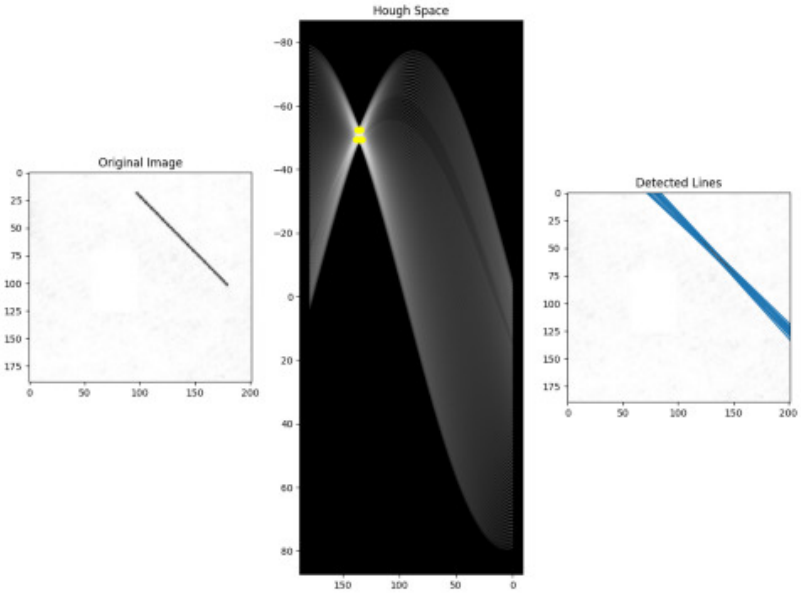


Figure 10: Transformé de Hough

3.1.2 La mise en application

La première étape pour implémenter cette technique a été de créer les différents repères. Le repère cartésien a les mêmes dimensions que celles de l'image et est représenté par l'array de pixels issu de la surface SDL reçue par la binarisation. Cependant, une des contraintes d'utiliser directement cet array est qu'il n'est qu'en 1 dimension, ainsi pour extraire les composantes x et y de chaque pixel nous avons utilisé le row-major order pour parcourir l'array (figure 11).

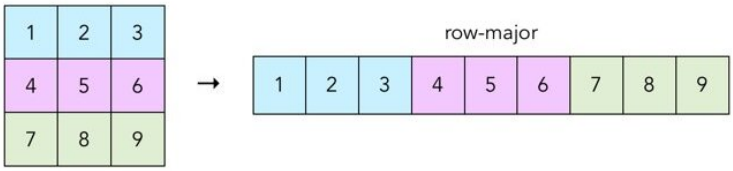


Figure 11: Transformé de Hough

Le second repère est un peu plus complexe: comme on peut le voir sur la figure 9 représentant une droite polaire, la distance rho maximal caractérisant une droite est la diagonale allant du coin haut-gauche au coin bas-droit de l'image, l'angle theta lui peut varier de 0 à 360 degrés. Contrairement au repère cartésien nous avons initialisé le repère polaire (aussi appelé accumulateur) nous-mêmes avec un simple calloc, ce qui nous renvoie un pointeur sur un array à une dimension rempli de 0. Nous avons donc encore une fois dû utiliser le row-major order pour le parcourir.

Nous avons désormais les deux repères requis pour appliquer le transformé de Hough et pouvons passer à l'application. Pour faire cela, nous parcourons la liste de pixels et calculons la moyenne des couleurs RGB du pixel afin d'en extraire son niveau de gris. Si ce dernier est supérieur à une valeur choisie nous allons le considérer comme un point important mis en avant lors de la binarisation. On va donc pouvoir tracer sa courbe correspondante, ce qui revient à ajouter 1 pour chaque point de la courbe dans l'accumulateur, puis répéter ce processus pour chaque pixel de l'array.

Une fois le parcours de l'image fini nous obtenons un repère polaire rempli. Or d'après le transformé de Hough, toutes les courbes sinusoïdales correspondantes aux points situés sur une même ligne dans le repère cartésien vont se croiser en un point dans le repère polaire. Ce point précis de coordonnées thêta et rhô va donc correspondre à une des lignes du quadrillage du sudoku. Il ne nous reste donc plus qu'à trouver les valeurs les plus grandes de l'accumulateur afin de trouver le quadrillage du sudoku et VOILÀ. Dans la théorie si l'on applique tout cela nous avons déjà fini de détecter les lignes, or la réalité et bien plus compliquée: la première grosse difficulté rencontrée

a été de trouver une valeur seuil permettant de différencier les lignes et les valeurs non désirées dans l'accumulateur. Au fil de nombreux tests nous avons trouvé que prendre 1/3 de la plus grosse ligne comme seuil était plutôt efficace pour la majorité des images. À partir de maintenant nous avons détecté toutes les lignes voir même trop mais ce n'est que temporaire car nous allons utiliser ce surplus de lignes pour vérifier si l'image a besoin de subir une rotation ou non.

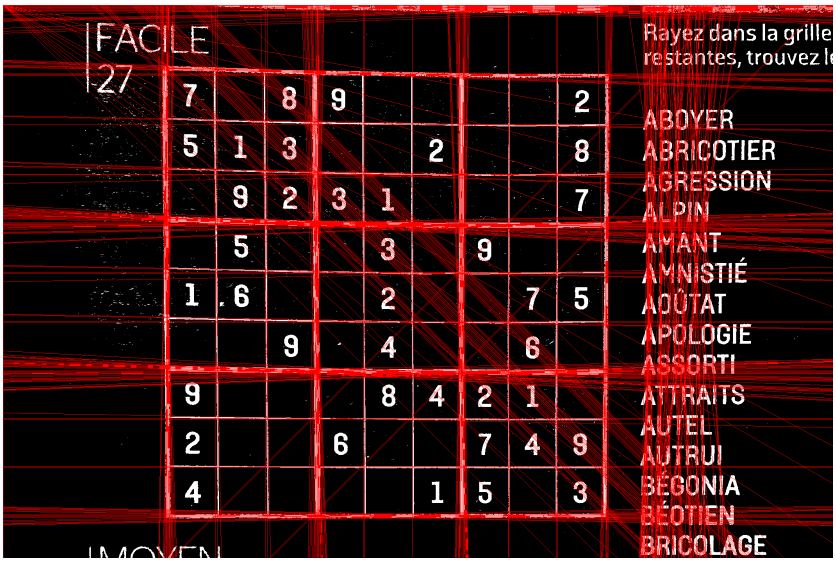


Figure 12: Toutes les lignes détectées

3.2 Rotation de l'image

Nous avons supposé qu'avant que l'image passe dans tout le processus de binarisation, l'utilisateur a redressé l'image de manière assez vulgaire. C'est à dire que la grille de sudoku a été orientée vers le haut mais n'est pas parfaitement droite (l'angle maximal avant la rotation étant de plus ou moins 45°). La fonction de rotation permet de vérifier si l'image a besoin de subir une rotation supplémentaire ou non, dans le cas positif l'image va être recalibrée de manière très précise pour avoir les chiffres orientés vers le haut. Pour faire cela, nous avons calculé la moyenne des valeurs des angles thêta de chacune des droites afin d'obtenir une valeur de rotation de l'image. Au final si nous avons tourné l'image nous devons refaire la détection des lignes mais cette fois-ci nous irons plus loin dans le processus pour isoler seulement les lignes qui nous intéressent. Si l'image n'a pas subi de rotation nous partons des lignes déjà détectées et allons filtrer les lignes parasites.

3.3 Filtrage des lignes

Comme vu précédemment il arrive régulièrement qu'on obtienne beaucoup de droites très similaires pour une seule et même ligne de l'image originale (voir figure 12). Nous allons donc passer toutes ces droites dans une fonction de filtrage nous permettant d'en garder une seule pour chaque ligne du quadrillage. Afin de faire cela, nous avons créé une nouvelle liste qui va contenir seulement les lignes que nous souhaitons garder. Pour remplir cette dernière nous parcourons toutes les droites une à une et les comparons à celles que nous avons déjà choisies. Si nous ne trouvons pas de ligne trop similaire nous pouvons l'ajouter à la liste des droites choisies. Dans le cas contraire nous allons comparer et voir laquelle des deux droites représente mieux le sudoku afin de garder la meilleure.

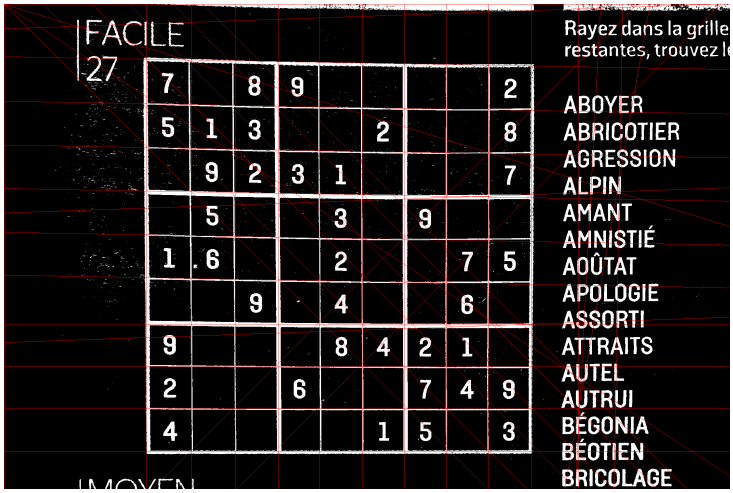


Figure 13: Suppression des doublons

Une étape supplémentaire assez simple, permettant de filtrer davantage de lignes parasites, consiste à se débarrasser des lignes qui ne sont ni verticales ni horizontales. À la fin de cette étape, nous nous retrouvons avec une liste plus exhaustive recensant seulement quelques dizaines de lignes au lieu des centaines précédentes.

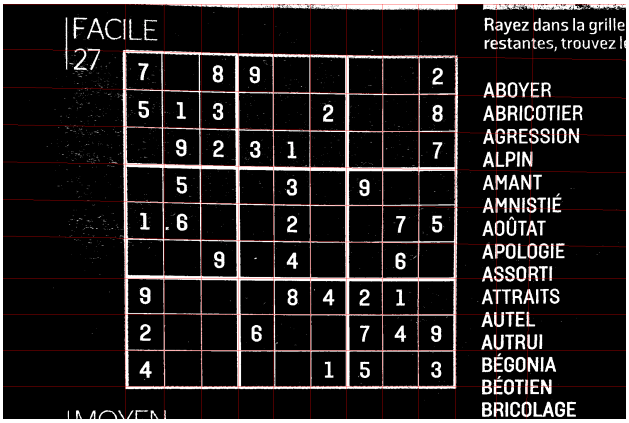


Figure 14: Suppression des droites obliques

Nous arrivons donc à un résultat proche du but, comme le montre l'image ci-dessus et la dernière étape du processus de filtration consiste à supprimer les dernières lignes en trop sur l'image pour qu'il n'en reste plus que 10 verticales et 10 horizontales. Cela a été une étape assez compliquée car elle a créé beaucoup de bugs et il a fallu l'implémenter après tout le reste sans pour autant dégrader le reste de la chaîne. La première étape a été de trouver la distance entre chaque droites, or un simple calcul de moyenne ne pouvait pas marcher car les lignes en trop risquaient de perturber le résultat. À la place nous avons créé un histogramme recensant toutes les valeurs d'écart entre les droites, puis en parcourant cette liste nous pouvons déterminer la valeur qui revenait le plus souvent et ainsi trouver la taille des cases. Il ne restait donc plus qu'à calculer l'espace entre chaque droites et retirer celles qui ne correspondaient pas à deux lignes de la grille. Ce n'est seulement qu'après toutes ces étapes que nous avons pu déterminer la grille du sudoku, or ce n'est pas fini, désormais il faut découper les cases pour les envoyer au réseau de neurones.

3.4 Détection des intersections

Maintenant que l'on a une seule droite correspondante à chaque ligne de la grille du sudoku, il nous faut trouver les intersections de celles-ci afin de pouvoir en extraire les cases. Cette partie n'était pas la plus dure mais a demandé quand même une bonne réflexion avant de trouver une solution optimale. L'idée est de trouver pour chaque droite le point d'intersection avec toutes les autres. Pour faire cela, nous avons initialisé à l'aide d'un calloc, un array rempli de 0 de la taille de notre image, comme d'habitude nous allons utiliser le row-major order pour considérer l'array comme ayant 2 dimensions. Ensuite, nous avons tracé chaque ligne dans l'array, c'est-à-dire que pour chaque point (x, y) de la droite, nous avons incrémenté de 1 la valeur à l'indice [x][y] dans la liste. Ainsi, lorsque deux droites se croisent, une valeur dans la liste aura été incrémentée deux fois ce qui nous révélera la position exacte de l'intersection de ces deux droites. Nous voilà ainsi en possession des 100 intersections et prêts à extraire les cases.

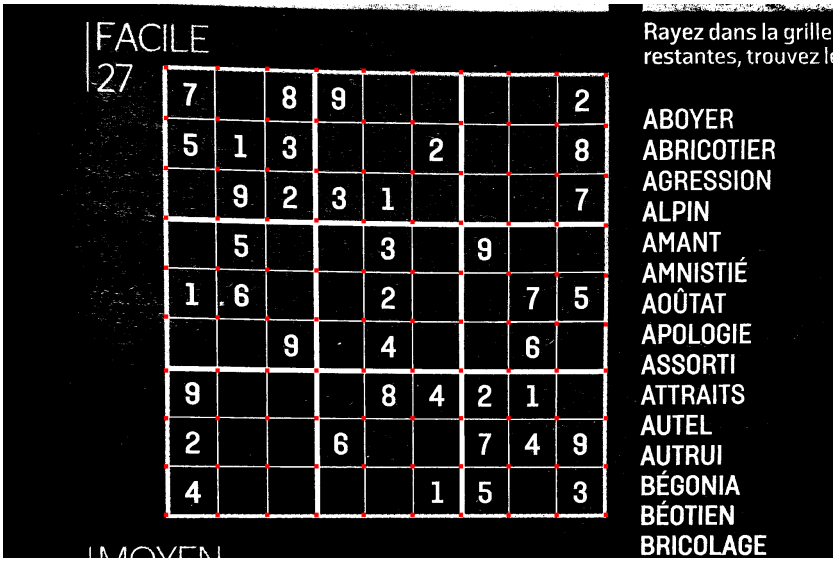


Figure 15: Intersections des droites

3.5 Découpe des cases

Découper les cases a été la dernière et la plus simple des étapes. Le processus consistait simplement à itérer à travers les 81 intersections situées en haut à gauche d'une case, récupérer ses voisines de droite et de gauche pour en déduire la hauteur et la largeur exacte de la case. Puis effectuer une copie de cette partie de l'image dans une surface SDL pour générer une nouvelle image correspondante à la case. Or une dernière petite modification s'impose car le réseau de neurones n'accepte que des images de 28*28 pixels. Il a donc fallu redimensionner la surface SDL pour la suite. Toutes ces surfaces SDL ont été stockées dans un array puis envoyées au reste de la chaîne.

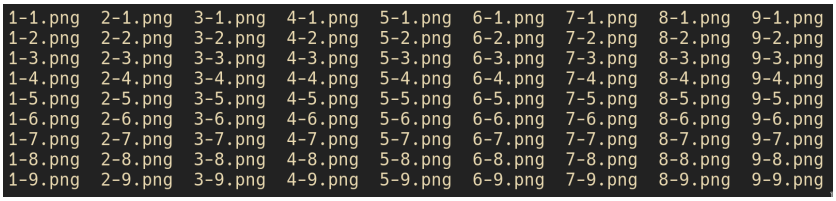


Figure 16: Toutes les cases découpées

4 La reconnaissance des chiffres

La reconnaissance de chiffres permet de convertir l'image en une grille de texte que le solver peut lire et résoudre. Notre objectif est de convertir une liste de pixels en un nombre de 1 à 9, mais aussi de détecter les cases vides que l'on a décidé de représenter par des 0.

Le problème que résout un réseau de neurones, est la régression de problèmes non linéaires. Prenons l'image ci-dessous:

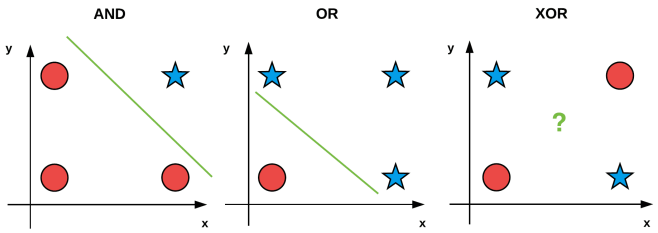


Figure 17: AND et OR linéairement séparable, XOR problème non linéaire

On remarque que les deux premières images peuvent être séparées par une simple ligne, cependant quand les données deviennent plus complexes, nous avons besoin de rajouter plus de paramètres pour pouvoir classifier nos données.

Pour revenir à notre problème qui est la reconnaissance de chiffres, si la disposition des pixels ne changeait pas, nous pourrions écrire des règles plutôt simples. Toutefois, chaque image de chiffre a sa particularité, c'est là que les neurones interviennent. De par la forte connexité d'une couche à l'autre, nous pouvons entraîner un réseau à détecter les caractéristiques qui composent un certain chiffre et donc d'être très performant s'il est entraîné avec un grand jeu de données variées.

4.1 Preuve de concept

Avant de se lancer directement dans la reconnaissance de caractères, nous avons écrit notre premier réseau de neurones pour qu'il puisse résoudre le problème du XOR. C'est un problème connu pour s'entraîner à créer des réseaux de neurones en gardant les données simples, ce qui facilite grandement l'apprentissage de cette technologie qui nécessite surtout de comprendre les maths qui se cachent derrière.

4.2 Les matrices

En effet, les réseaux de neurones ne sont que des équations mathématiques mais avec quelques milliers de paramètres. Alors pour ne pas avoir à mélanger maths et code, nous avons créé une librairie permettant la gestion de matrice. Cette dernière permet évidemment de créer des matrices mais aussi d'effectuer diverses opérations (i.e. addition, multiplication, multiplication par un scalaire, transposé, ...).

Un avantage caché de la librairie est l'optimisation des performances. Nous implémentons toutes nos matrices dans des tableaux à une dimension ce qui permet d'être plus rapide dans les calculs et donc de permettre au réseau d'apprendre plus vite.

4.3 Les maths qui se cachent derrière

Pour décomposer le problème en plusieurs étapes, la conception d'un réseau de neurones implique d'implémenter les parties suivantes :

- La Forward Propagation
- La Backward Propagation
- Les optimisations et les réglages

Mais avant tout, notons:

- L[0] la première couche et L[i] la couche i
- W[i] (resp. B[i]) les poids (resp. biais) associés à la couche i

4.3.1 Forward Propagation

La propagation vers l'avant se résume à faire une prédiction. On donne en entrée une image et on peut lire le résultat sur la dernière couche, L[2].

Dans un premier temps, il faut encoder une image en 28 par 28 dans une matrice 784x1, ou plus simplement, on lit l'image de gauche à droite en commençant par le pixel 0,0 (en haut à gauche) et on les met tous à la suite. Alors, le pixel de coordonnées x,y se retrouve à l'index 'y * 28 + x'. En plus de ce changement de dimensions, nous divisons la valeur noir et blanc des pixels par 255 pour travailler avec des valeurs entre 0 et 1. Cela permet de garder de petites valeurs à travers le réseau car nous utilisons la fonction exponentielle à certains endroits.

Le calcul d'une couche à une autre est assez simple et se résume en 3 étapes :

1. Multiplication matricielle entre W[i] et L[i-1] pour obtenir Z[i], la matrice avec les valeurs non activée
2. Ajout des biais avec Z[i] += B[i]
3. Activation des valeurs en utilisant soit la fonction ReLU, soit la fonction SoftMax, tel que L[i] = ReLU(Z[i])

Comme nous pouvons le voir, les calculs dans ce sens sont relativement simples, nous noterons aussi que la valeur Z[i] est conservée pour la partie Backward Propagation.

4.3.2 Backward Propagation

Ce sens de propagation intervient uniquement lors de l'entraînement, il permet de calculer à chaque fois les erreurs commises par le réseau et de calculer tous les ajustements nécessaires aux poids ainsi qu'aux biais. Ces 2 dernières valeurs étant les seules que l'on peut ajuster pour améliorer le réseau.

Cette partie est bien plus complexe et nécessite aussi d'avoir fait une propagation avant, en amont. Pour simplifier l'exemple, nous allons admettre que le réseau en question est constitué de 3 couches (entrée L[0], caché L[1], sortie L[2]).

1. Calcul d'erreur, c'est la différence dL[2] = L[2] - Y, où Y est le résultat que l'on attendait et dL[2] les erreurs pour L[2]
2. On obtient dW[2], les ajustements à faire à W[2] en multipliant dL[2] et la matrice transposée de L[1]
3. Or pour obtenir dW[1], il faut tout d'abord calculer la dérivé de la fonction d'activation utilisée, ici, ReLU. Une fois la dérivé obtenue, on peut la multiplier à la matrice transposée de W[2] et on obtient dL[1]. Finalement dW[1] = dL[1] x transposé de L[0]
4. Les ajustements des biais sont simplement dB[1] = dL[1] et dB[2] = dL[2]
5. Avant d'appliquer tous ces changements, on multiplie tous les ajustements par le learning rate, qui permet au réseau d'apprendre progressivement.

4.4 Vérifier les performances

Quand on développe un réseau de neurones, il faut vérifier ses performances, nous en avons sélectionné deux :

- Le coût du Dataset d'entraînement
- Le coût du Dataset de validation

Quand on lance un entraînement, on met de côté 10% de ce dernier pour contrôler les performances du réseau sur des données qu'il n'a jamais vu. Imaginons que le coût d'entraînement diminue tandis que le coût de validation stagne voire augmente, nous passons en sur-apprentissage, cela veut dire que le réseau a trop appris et qu'il essaie d'être excellent sur ses images d'entraînements et ne généralise pas.

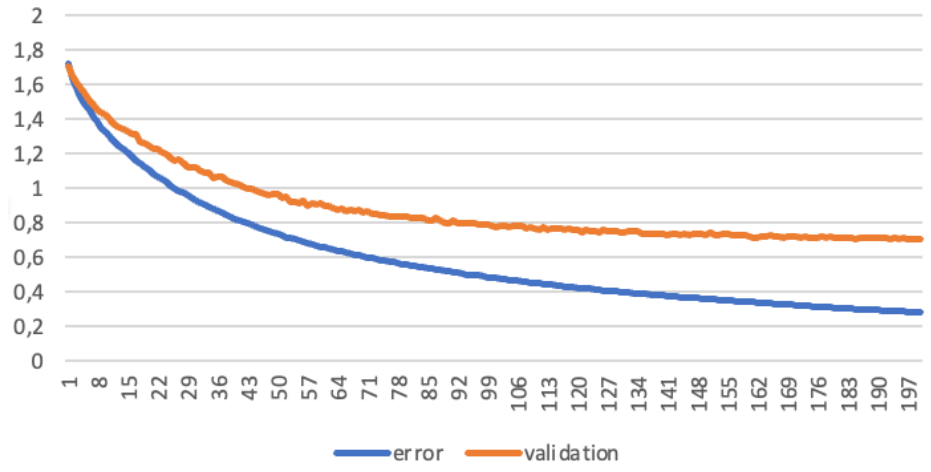


Figure 18: Donnée extraite d'un de nos réseau

4.5 Le dataset d'entraînement

Pour entraîner un réseau il faut des images, beaucoup d'images (quelques milliers). Il y a beaucoup d'options disponible sur Internet comme le MNIST qui est un dataset constitué d'une dizaine de milliers de chiffre manuscrit. Or nous travaillons avec des chiffres tapé à l'ordinateur ce qui est très différent pour un réseau de neurones.

Alors nous avons pris la décision de générer nos propres images. Cette partie, ne faisant pas partie du projet même, à était réalisé en Python car c'était plus simple et rapide à développer. Ce mini logiciel nous à permis de générer toute sorte d'images, tourné aléatoirement, déplacé du centre, avec du bruit et même de police différentes.

Or comme vous le verrez dans la dernière partie de ce rapport ce dataset n'aura pas su remplacer les vraies images qui sont binarisés et coupés par notre code.

Pour générer ces données, nous utilisons un fichier CSV, sur lequel on imprime toutes les valeurs durant l'entraînement, alors on peut voir quel problème survient quand le réseau ne veut pas apprendre. Ces données permettent aussi de tester des combinaisons d'hyperparamètres qui sont présentées dans la partie suivante.

4.6 Les caractéristiques de notre réseau

Notre réseau permet de reconnaître des chiffres de 1 à 9 et des cases vides dans des images de 28 pixels par 28. Cela fait un total de 784 neurones sur la première couche et 10 neurones pour la dernière couche.

Pour les couches cachées, nous avons fait le choix d'en avoir qu'une seule composée de 30 neurones. Ce chiffre n'a pas été choisi arbitrairement mais après beaucoup d'essais de combinaisons d'hyperparamètres.

Parmi ces paramètres nous en avons 3 autres :

- Le learning rate, c'est la vitesse à laquelle le réseau va faire des changements sur ses paramètres internes. Pour nous il est de 0.001
- Le decay va contrôler la vitesse de décroissance du learning rate. Car si le réseau est très mauvais à ses débuts et nécessite un haut learning rate, ce n'est plus pareil lorsque qu'il nécessite de faire de léger changement. Alors pour pallier ce problème, le learning rate décroit à chaque itération. Nous l'avons mis à 0.000001, ce qui peut paraître peu mais il est important de noter que le réseau effectue 5000 itérations par secondes.
- Le momentum. Chaque poids des paramètres internes du réseau va globalement bouger dans une seule et même direction. Pour accélérer l'entraînement, nous calculons la vitesse à laquelle le poids bouge et nous pouvons accentuer son mouvement. Il est de 0.5.

Pour ce qui est des fonctions d'activations sur les couches du réseau, nous utilisons la fonction ReLU (voir fig.) sur la couche cachée et la fonction SoftMax pour la dernière, car elle permet d'avoir un résultat sous la forme de probabilité pour chaque résultat possible (voir fig.).

4.7 Enregistrer le réseau

Après avoir entraîné notre modèle pendant quelque temps, il faut un moyen de l'enregistrer. L'option intuitive et la plus simple serait d'écrire du texte dans un fichier, cependant prenons un exemple, "0.47843934473843" en texte prendrait 16 octets alors qu'un double en C ne prends que 8 octets. Comme nous utilisons des modèles avec une quantité conséquente de poids,

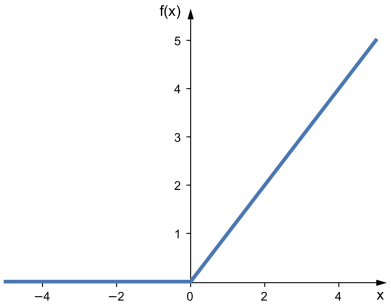


Figure 19: Fonction ReLU

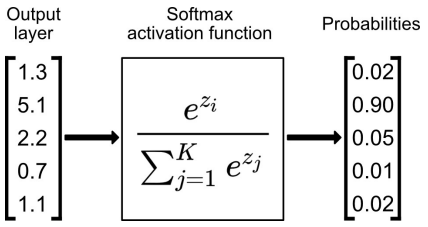


Figure 20: Fonction SoftMax

nous avons créé une seconde librairie qui permet la création de buffer. C'est-à-dire que toutes les données sont écrites à la suite directement en binaire.

Voici un graphique pour visualiser comment une matrice serait sérialisée dans le buffer :

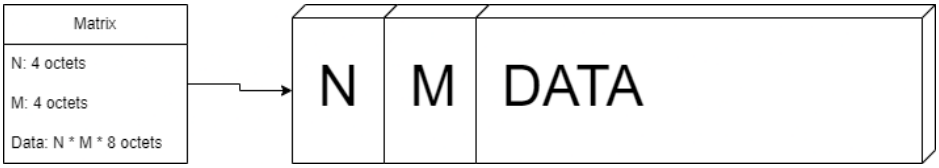


Figure 21: Une matrice dans le buffer

Quand on va rencontrer une matrice, on sait qu'il faut d'abord lire n puis m qui font chacun 4 octets. Après avoir récupéré ces 2 valeurs, on peut calculer la taille de data en faisant N * M * 8 et on peut désormais copier tous ces octets dans notre programme.

Ce système de buffer nous permet d'avoir des fichiers de sauvegarde beaucoup moins lourd, mais c'était avant tout un mini projet pour explorer plus en détail la (dé)sérialisation de données.

5 La résolution du Sudoku

Pour résoudre le sudoku, nous avons utilisé une méthode connue qui s'appelle le backtracking, déjà vu l'année dernière lors d'un des TP de programmation, où on devait faire un solveur de sudoku. Le principe est très simple: il faut que notre algorithme arrive à placer les nombres de 1 à 9 sur la ligne, la colonne et le carré de 3 par 3. Pour ça, en utilisant le backtracking, il faut que sur chaque case vide, après avoir vérifié qu'on pouvait bien y placer un nombre, on y place un nombre puis on rappelle récursivement la fonction jusqu'à une des deux possibilités arrive : soit on ne trouve pas de solution et on repart en arrière, soit on a trouvé une solution au sudoku et on renvoie la grille résolue.

Pour nous faciliter les choses, l'algorithme a été découpé en 3 partie : une qui récupère le ficher de la grille et qui le transforme en array d'integers, une autre qui résout la grille et une dernière qui enregistre la grille dans un fichier .result.

La partie qui a posé le plus de problèmes n'était pas le solver mais comment transformer le fichier en array, à cause de la manipulation d'arrays. Au début tout allait bien, on crée un pointeur sur le fichier et on enregistre chaque ligne dans un array en deux dimensions. Avant de faire le solveur, l'idée de renvoyer un array de charactères paraissait être le meilleure, car il ne fallait pas changer de type, mais après nous avons dû changer et renvoyer un array de nombres. L'étape d'après est d'enlever les espaces et de mettre les nombres dans un array final que nous envoyons au solver, seulement nous avons dû faire face au choix de la dimension de l'array qui n'était pas la bonne au début et qui renvoyait un stack smashing. Le problème a été réglé en testant plusieurs valeurs jusqu'à comprendre lesquelles fonctionnaient et pourquoi. Seulement, par habitude, on considérait qu'une ligne finissait par un charactère nul sauf que l'algorithme renvoyait des valeurs qui n'avaient rien à voir avec la grille, il a fallu beaucoup d'incompréhension et de temps pour réaliser que c'était un retour à la ligne.

Pour la partie du solveur, nous nous sommes renseignés sur internet pour faire un algorithme optimisé basé sur le backtracking, ainsi que le tp de l'année dernière. Comme expliqué précédemment, le but est de tester sur une case vide, ici les points ont étés changés en 0, les valeurs de 1 à 9, et pour chaque test, rappeler récursivement la fonction sur la grille avec la valeur qu'on veut tester. Si à la fin aucune solution a été trouvée, on passe à la valeur suivante.

Pour finir cette partie, il fallait enregistrer la grille solution dans un fichier, sans oublier de remettre des espaces entre les blocs de 3. Dans le dossier git, on peut y trouver des grilles de sudoku pour tester le solveur, certaines sont faciles et d'autres sont difficiles

```
kam@ubuntu:~/SPE/ocr-epita/solver$ ./solver ../DataSample/solver/grid001
kam@ubuntu:~/SPE/ocr-epita/solver$ ls
Makefile  result.txt  solver  solver.c  solver.h  solver.o
kam@ubuntu:~/SPE/ocr-epita/solver$ cat result.txt
127 634 589
589 721 643
463 985 127

218 567 394
974 813 265
635 249 871

356 492 718
792 158 436
841 376 952
```

Figure 22: Résultat final

6 L'interface Utilisateur (UI)

La partie de création de l'interface a été réalisée avec gtk et glade comme annoncé dans le rapport précédent. Afin de vous présenter l'interface réalisée nous allons vous présenter chaque fenêtre dans l'ordre d'apparition lors de l'utilisation.

Pour la partie ui, l'aide trouvée sur internet se résume à des tutoriels Youtube, ou de la documentation glade et gtk, ainsi que du tp d'IP qu'on a réalisé cette année.

Dans tout le fichier ui.c, les variables des composants sont mises en global car on trouvait ça plus pratique, afin de les utiliser dans n'importe quelle fonction sans se demander à quel composant elles appartiennent. Notre interface est organisée de façon à ce que chaque étape se fasse sur une fenêtre différente. Cette décision a été faite afin de facilité l'organisation des fenêtres, dans le code et dans la création des widgets.

6.1 La fenêtre d'ouverture

Ce fut la première fenêtre que nous avons faite et ce fut celle qui a servi de test, afin de se familiariser avec gtk et glade. Le problème avec cette fenêtre, c'est que c'était le début et même si on avait fait un schéma de ce qu'on voulait avoir, j'étais perdue quant au type de composant à utiliser, comment le faire et à quoi ça devait ressembler. C'est donc une fenêtre qui a subi bcp de modifications.

En premier lieu, il fallait créer la fenêtre et grâce à glade, il suffit de prendre une GtkWindow dans la liste de composants qu'ils proposent. Mais une fenêtre toute seule n'est utilisable que si on ne veut mettre qu'un seul composant dedans. Pour y placer plusieurs composants, il faut utiliser des GtkContainers. Il en existe différents types, comme des grilles(GtkGrid) ou celui utilisé en tp (GtkPaned) qui sépare la surface en 2. Celui qui est le plus pratique à mon goût est le GtkFixed car il n'impose aucune contrainte, c'est juste un cadrillage sur la fenêtre et c'est celui qui a été utilisé pour toutes les fenêtres crées.

Tout d'abord, la partie la plus simple est placer le titre, qui est sous la forme de GtkLabel. L'utilisation de glade pour ça est très pratique, car il suffit de rentrer dans un champ ce qu'on veut que le label affiche et dans un autre champ, indiquer les coordonnées où le titre sera affiché. On peut régler les attributs tels que la police et la taille de texte directement dans glade. Au niveau du code, il n'y a rien a faire avec le titre, il est possible de faire apparaître et disparaître un GtkLabel avec du texte mais on ne s'est pas servi de cette fonctionnalité.

Ensuite la partie la plus importante de cette fenêtre est le GtkFileChooser. Au début, ce n'était pas ce type de composant qu'on utilisait mais un GtkEntry, qui est une barre de saisie afin de récupérer du texte et on devait y écrire le chemin vers le fichier. J'avais placé ce composant parce que je ne savais pas qu'il existait un GtkFileChosser, qui est bien plus pratique et bien plus esthétique.

Quand le ficher est sélectionné, un signal est émis et grâce à une fonction, on peut récupérer le nom du fichier. Le principe d'interface en utilisant glade et gtk repose sur les évènements et pour changer de fenêtre, il nous fallait un évènement. Pour ce faire nous avons placer des boutons "NEXT" sur chaque fenêtre afin de récupérer le signal pour passer à l'étape suivante. Et c'est comme ça qu'on passe à la deuxième fenêtre.

6.2 La fenêtre de rotation manuelle

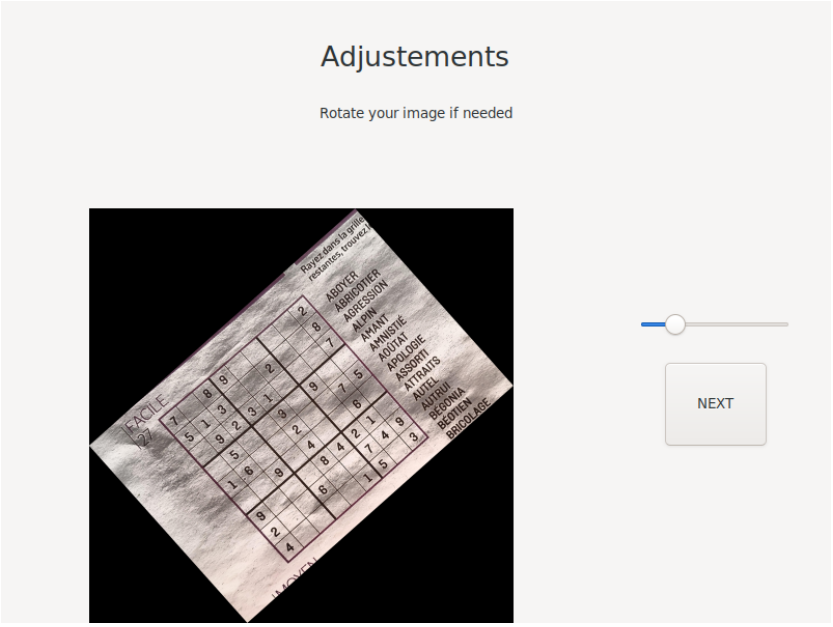
Pour cette fenêtre, nous nous sommes dit que ce serait intéressant que l'utilisateur puisse faire une rotation de l'image lui-même. L'organisation de la fenêtre et des suivantes est telle qu'on affiche l'image au milieu de celle-ci et on implémente des fonctionnalités à côté.

Du coup, pour faire ça, nous avons posé une GtkImage au centre, qui servira à afficher l'image tournée. Pour réaliser ça, nous travaillerons dans le code avec des SDL_Surface, car c'est ce qui est utilisé dans les parties de découpage, binarisation et réseau de neurones . Seulement, il n'existe pas de fonctions qui transforme une SDL_Surface en GtkImage donc nous avons dû faire la nôtre, qui a été très utile.

La personne en charge de la détection de ligne avait déjà fait une fonction qui tournait l'image donc tout ce qu'il restait à faire c'était de relier les composants à la fonction, mais il nous manquait de quoi récupérer la valeur de rotation. Pour augmenter ou diminuer l'angle de rotation, nous avons hésiter entre un GtkEntry et un GtkScale mais nous avons pensé de que le GtkScale faisait plus stylé. Cette fonction a tout le nécessaire pour faire la rotation de l'image et renvoie la SDL_Surface associée à la rotation.

Comme nous travaillons avec des surfaces et pas des images, c'est ce que nous gardons de chaque étape et ce que nous envoyons aux fonctions de traitement de l'image. Il a fallut modifier légèrement ces fonctions afin de les adapter pour l'ui. Bien sûr, il existe des instructions sur la fenêtre afin de guider l'utilisateur. La suite est accessible avec le bouton next et redirige vers la fenêtre de binarisation.

Ce qui nous à permis de faire cette page est la réactivité et la vitesse de l'algorithme de rotation, en effet si un appel à cette fonction prenait 100ms l'expérience pour l'utilisateur ne serait pas agréable.

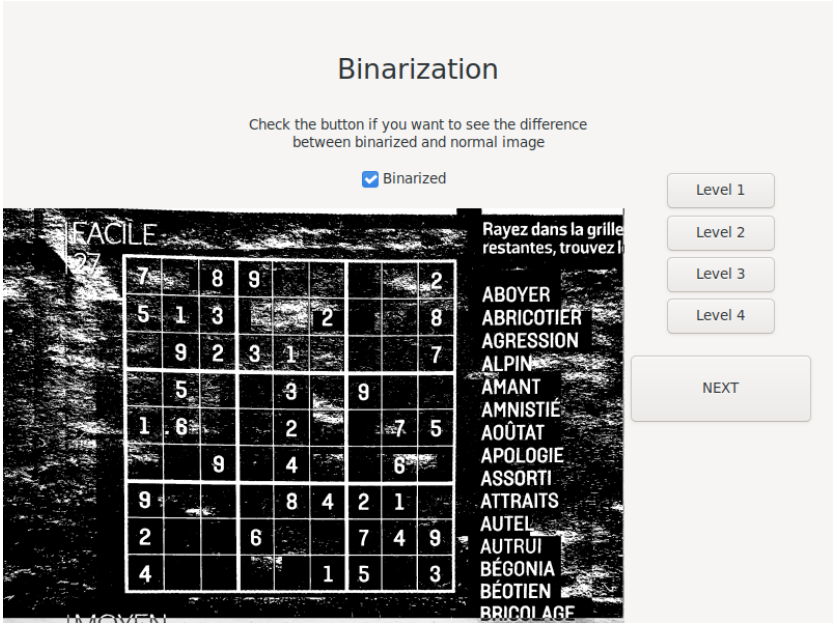


6.3 La fenêtre de binarisation

Pour la partie binarisation, nous nous sommes dit que ce serait une bonne idée de faire en sorte que l'utilisateur puisse voir la différence entre l'image binarisée et l'image normale, ainsi de pouvoir choisir de niveau de précision de l'algorithme de binarisation.

Pour ce faire, nous avons implémenter un GtkCheckButton, qui change de surface entre celle binarisée et celle normale en fonction de s'il est actif, sachant qu'il envoie un signal dès que son état est changé, donc il est simple d'implémenter ça. De plus, les boutons sur le côté servent à choisir de niveau de précision de l'algorithme de binarisation. Comme expliqué dans la partie binarisation, plus le niveau de précision est élevé, meilleure la binarisation sera.

L'algorithme va découper l'image en plusieurs parties et va binariser ces parties indépendamment des autres. Pour certaines images qui ont une mauvaise qualité, elles nécessiteront un niveau de précision plus élevé, et inversement pour les images de bonne qualité. Il y a donc un bouton par niveau. La suite de l'interface est accessible en appuyant sur le bouton NEXT.



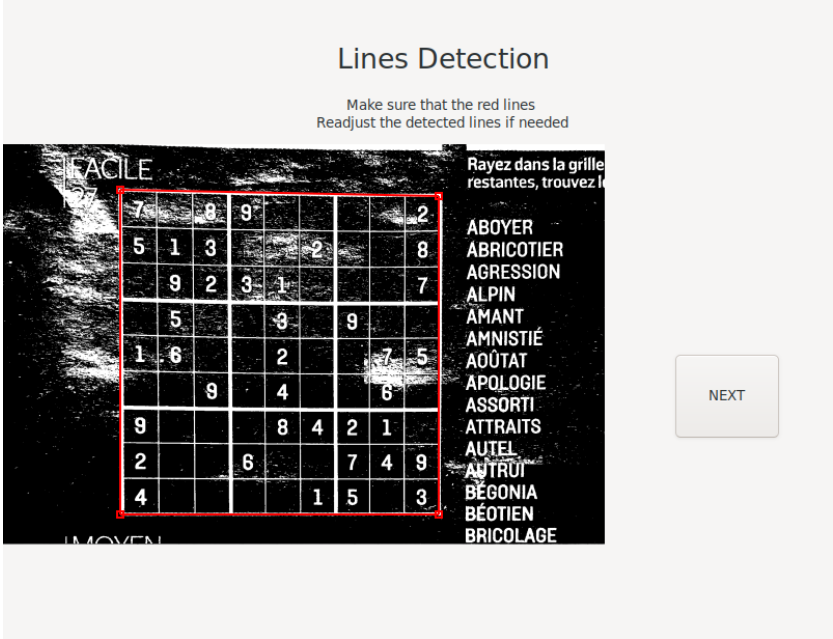
6.4 La fenêtre de détection de ligne

Cette page permet de visualiser le résultat de la détection de ligne. Pour ne pas surcharger la page, nous avons pris la décision d'afficher seulement les 4 coins de la grille du sudoku. Pour faire cela nous avons superposé une zone de dessin sur une image, la zone de dessin sert à afficher 4 points rouges correspondants aux 4 coins détectés, tandis que l'image est simplement l'image binarisée. À la fin du processus de détection nous nous somme contentés de garder les 4 intersections qui nous intéressent pour les afficher. Or l'image que l'on affiche dans l'interface utilisateur n'a pas les mêmes dimensions que l'image originelle. Il a donc fallu rétrécir l'image par une certaine valeur puis faire de même avec les coordonnées des intersections pour que celles-ci soient correctement placées sur l'image. En plus des 4 points nous avons aussi tracé des lignes les reliant pour mieux les visualiser.

Un autre avantage de cette page est qu'il est possible de replacer les points si besoin est. En effet, la détection de lignes n'étant pas parfaites, il arrive que les 4 intersections soient mal placées et afin de s'assurer que le reste de l'OCR puisse tout de même marcher, une vérification manuelle en plus était la bienvenue.

Pour réaliser cette fonctionnalité nous avons utilisé une GtkDrawingArea, qui est une zone dans l'interface où il est possible de capter certaines action de l'utilisateur. Dans notre cas nous nous sommes intéresser aux cliques et aux mouvements de la souris. Il est ainsi possible de cliquer à proximité d'un des points et de le déplacer comme bon nous semble. La partie technique de cette fonctionnalité a été de trouvé les différentes fonctions de la librairie GTK nous permettant de faire ce que nous souhaitions. La documentation de cette librairie est très complète mais aussi très très dense et trouver notre bonheur parmi cette océan de pages et de fonctions n'était vraiment pas simple.

Nous avons tout de même réussi à trouver et faire quelque chose qui marche avec comme seule contrainte, que la fonction que nous avons utilisé est dépréciée par la librairie GTK3, et crée donc des warnings à chaque compilation. Cependant la page produit l'effet désiré donc nous avons pris la décision de garder cette fonction.



6.5 La fenêtre de résolution et sauvegarde

Pour la dernière fenêtre, il s'y passe deux parties. La première démarre quand le bouton next de la fenêtre de rotation est cliqué, le programme va envoyer les cases découpées au réseau de neurones afin qu'il les transforme en un array de chiffres qui représente la grille de sudoku. Bien que cette partie soit très importante, nous n'avons pas trouvé de moyens de la montrer. La deuxième est qu'après l'appel au réseau de neurones, le programme va faire appel au solveur afin de résoudre cette grille.

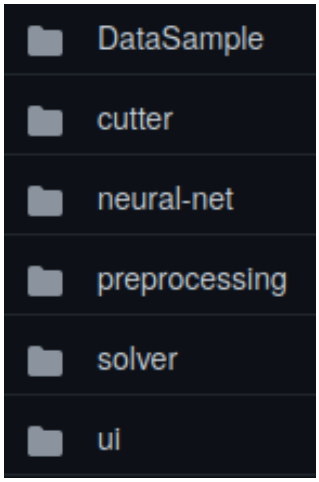
Pour l'affichage, nous avons toujours une GtkImage qui va afficher l'image de la grille résolue. Le problème avec cette partie est que je ne savais pas comment faire pour placer des images de chiffres sur une image de grille vide. Au début, je pensais faire un tableau sur la fenêtre, placer 81 éléments GtkImage et y placer les bons chiffres avec la grille que renvoie le solveur. Bien sûr cette idée est très couteuse en temps, car elle est longue à réaliser et elle ne permet pas de faire une image pour la sauvegarder ensuite.

Nous avons donc du chercher une autre solution. C'est en me renseignant sur les SDL_Surface que je suis tombé sur une fonction qui fait exactement ce que nous cherchions à faire. C'est la fonction BlitSurface qui permet de copier une surface sur une autre, en ne sélectionnant qu'un SDL_Rect qui délimite l'endroit où on colle la surface. Pour implémenter cette technique, il nous fallait juste des images des chiffres allant de 1 à 9, afin de pouvoir les convertir en surface et les coller sur la grille vide (elles se trouvent dans le fichier solver).

L'utilisateur va pouvoir sauvegarder cette image en appuyant sur le bouton Save File de l'interface en tant qu'image appelée solved_sudoku.png. L'ajout du bouton Return Home va être utile à l'utilisateur s'il souhaite revenir à la page de départ afin d'utiliser l'interface sur une autre image

7 Organisation, Équipe et Expérience

Dès la création du groupe, nous avons mis en place un Git et une procédure pour travailler. Chaque partie est isolée dans son dossier et une seule personne travaille sur une branche.



Cette organisation nous a grandement facilité la tâche lors des multiples merges qui ont eu lieu. Le grand merge de fin de projet n'aura pas été victime de multiples conflits complexes, comme nous avions eu lors du Projet de S2. Cependant, il a été victime de manque de tests d'intégration.

En effet, même si toute l'équipe a très bien testé son code individuellement, nous n'avions pratiquement jamais effectué de tests entre les parties. Pour prendre un exemple, le réseau de neurones a été testé sur un très grand nombre d'images avec des chiffres noirs sur fond blanc et sans ligne sur les côtés. Or, les images qui sortent du découpage ont des chiffres blanc sur fond noir et des lignes de sudoku de temps à autre. Ce problème de manque de tests d'intégrations dans des gros projets peuvent être très coûteux dans le monde professionnel. En tout cas, cela est sûr nous avons tous appris qu'il faut plus tester entre les parties.

L'équipe a globalement beaucoup appris, car personne n'avait touché au C avant le projet ni à la théorie et au fonctionnement d'algorithmes complexes comme le traitement d'image ou la détection de ligne.

8 Conclusion

Pour conclure, chaque membre de l'équipe à tenu ses deadlines, permettant une mise en commun rapide et efficace. Désormais, le projet ne prend plus la forme de ligne de commande dans le terminal mais d'une réelle interface, dans laquelle on peut ouvrir une image de sudoku et observer toutes les étapes du traitement de l'image jusqu'à la résolution de la grille.

Vous pourrez retrouver l'intégralité de notre travail ainsi que quelques images à l'URL suivante: https://github.com/kr4xkan/ocr-epita

Pouce-ment 👍