EPITA 2027 – S3 2027

AléaD1-2, D1

KADOUCI Marwane
OGÉ Guillaume
POIRON Roman

NOVEMBRE 2023

Sommaire

1. Présentation des membres du groupes

Guillaume

Pour moi, ce projet est en grande partie pour apprendre de nouvelles méthodes au niveau du C, que ce soit au niveau des structures de données comme les listes chaînées ou simplement les mallocs. L'intérêt aussi sur ce projet, contrairement au projet S2, est d'utiliser beaucoup plus les maths pour certains traitements sur l'image. Enfin, en 2023, on parle beaucoup de l'IA, notamment avec Chat-GPT, et ce projet est une porte d'entrée pour comprendre comment marche un réseau de neurone et un exemple concret d'utilisation. Même si c'est un projet assez complexe à comprendre et à retranscrire en code, j'en apprends beaucoup et c'est très intéressant.

Marwane

Je suis enthousiaste à l'idée de travailler sur ce projet. Avant d'entrer à l'EPITA, je m'étais demandé comment il est possible de réaliser un programme de reconnaissance de caractère dès la deuxième année ... nous y voilà ! J'espère développer à travers ce projet ma capacité à apprendre en autonomie et à travailler en équipe. Ce projet me permettra par les approches documentaires et démarches personnelles d'apprendre comment présenter mes idées (résumé, synthèse), analyser un document (repérer des idées importantes, relier les éléments entre eux), exploiter des informations (sélectionner des informations, développer les aspects pertinents). Ces compétences sont essentielles dans le métier d'ingénieur, là est ma source de motivation.

Roman

Ce projet est le premier de ma scolarité à l'EPITA, c'est donc pour moi l'occasion de découvrir comment cela fonctionne ici et de faire un projet qui demande du temps et de l'investissement en coopération. Je trouve ce projet intéressant car il permet une application très concrète du langage C ainsi que de comprendre un peu le fonctionnement du traitement d'une image à l'aide d'une IA, ce qui dans un monde où l'IA croit en influence quotidiennement est je pense nécessaire. Ce projet est donc un pas vers une meilleur compréhension des IA. Compréhension, qui me sera je pense très utile dans le futur.

Organisation du projet :

	Marwane	Guillaume	Roman
Pré-traitement de l'image			
Grayscale			
Contrast			
Normaliser luminosité			
Réduction du bruit			
Seuil local			
Détéction des agnles avec Sobel			
Détéction de grille			
Transformation de Hough			
Rotation			
Détection des carrés			
Segmentation de l'image			
Résolution du sudoku			
XOR et réseaux de neurones			

Setup du projet et requis

Packages : Les packages autorisées pour ce projet sont :

- SDL2, pour tout ce qui et traitement de l'image
- GTK+3, pour l'interface graphique qui sera terminée d'ici la deuxième soutenance

2. Prétraitement de l'image

Une des plus grosses parties à réaliser est le prétraitement de l'image. Cela constitue plus de la moitié du projet. Dans cette partie très dense, on va utiliser une image fournie dans le cahier des charges donnée par l'EPITA.

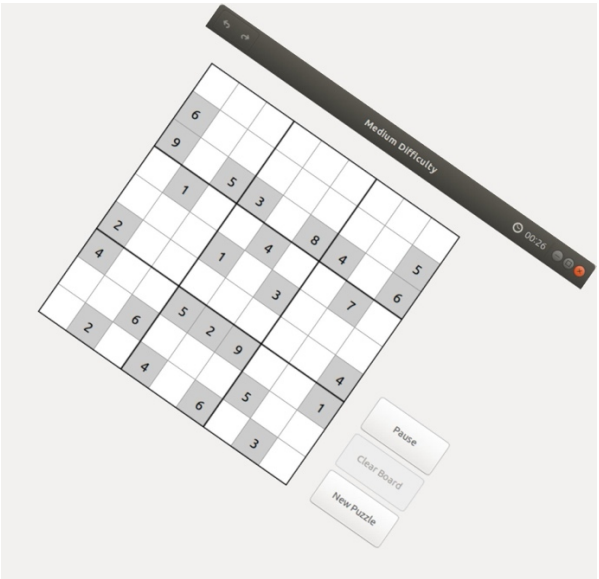


Image originale

La première étape avant de faire tout le prétraitement est évidemment de charger l'image et convertir l'image étudiée en SDL_Surface avec la bibliothèque graphique SDL2_Image. On peut convertir sur n'importe quel format d'image, que ce soit en « .jpg », « .png » ou même en « .bmp ».

a. Grayscale

La première étape est de transformer l'image en noir et blanc. Cette transformation est primordiale pour pouvoir détecter tous les éléments. Pour rappel, un pixel est constitué d'une variation de 3 couleurs différentes (les couleurs primaires RGB) : Red, Blue et Green. Comme nous avons eu un TP cette année sur SDL et un exercice pour convertir une image en noir et blanc, nous avons utilisé une formule modifiée pour convertir chaque pixel. La formule va faire la « moyenne » de couleur pixels par pixels puis l'appliquer ensuite :

$$Gray = R*0.2126 + G*0.7152 + B*0.0722$$

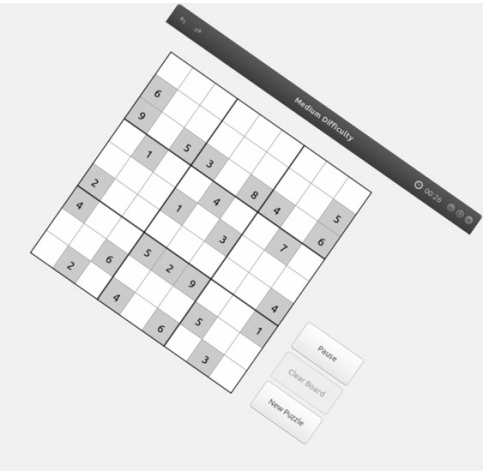


Image après l'étape du grayscale

b. Contraste

i. Ajout du contraste

Pour que notre OCR puisse donner des performances correctes et justes, il faut absolument augmenter le contraste de l'image. Cela permettra de détecter pour la suite le plus d'éléments possible. Cela se fait en parcourant chaque pixel et en comparant sa valeur désormais en niveaux de gris à la valeur la plus élevée de chaque pixel. La différence de couleur est ainsi augmentée pour que l'on puisse, par la suite, distinguer plus facilement les bords des formes comme les coins de la grille sudoku.

ii. Inversion de l'image

Il fallait absolument inverser les couleurs de l'image, pour pouvoir détecter plus facilement la grille, qui passera du noir au blanc. On applique cette formule pour chaque pixel individuellement :

$$NouvelleValeurPixel = 255 - ValeurPixel$$

c. Normaliser la luminosité

Même si le contraste a aidé à faire apparaître plus de détails sur le sudoku, on doit normaliser la luminosité de l'image. Pour faire cette normalisation, on doit chercher le pixel avec la plus grande valeur sur toute l'image et appliquer séparément sur chaque pixel cette formule :

$$NouvelleValeurPixel = 255 - ((255/MaxValeurPixel) * ValeurPixel)$$

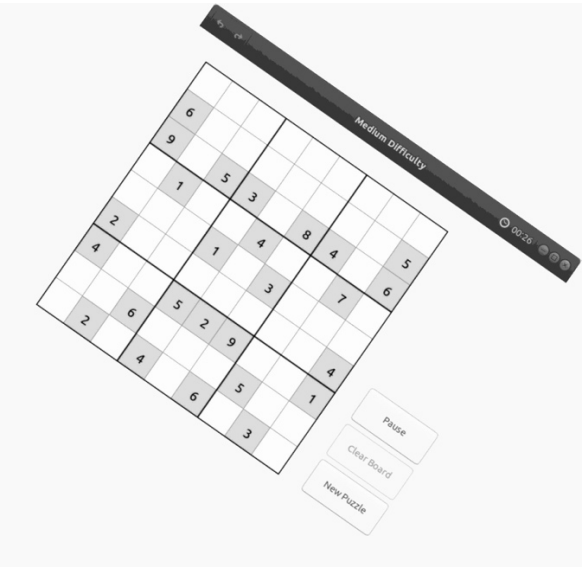


Image après l'augmentation du contraste

d. Réduction du bruit

La grande majorité des images que l'on va avoir proviendra en général d'une caméra de téléphone. Et même si de nos jours les caméras de smartphones ont de meilleure qualité graphique et que l'on pourrait même utiliser pour faire un film, il reste des imperfections toujours visibles et cela pourrait faire bloquer l'OCR pour plus tard. Nous allons donc créer un traitement pour réduire le bruit de l'image.
Pour cela, on créer une copie de l'image parce que le processus de réduction de bruit fonctionne par échantillonnage dans différents groupes de pixels. En faisant cela, on compare après l'image modifiée et l'image originale pour voir si elle a été trop dénaturée.

i. Filtre médian

La première étape avant d'appliquer une réduction de bruit est d'appliquer un filtre médian. On prend les 9 pixels voisins d'un pixel spécifique et on les met dans une array. L'array va ainsi trier les éléments et l'élément du milieu, appelé le médian, sera pris et remplacera les 9 autres pixels. Nous avons préféré utiliser ce filtre un peu exotique comparé à un filtre basique car il donnait de meilleurs résultats finaux sur la détection des angles et de la grille.

ii. Flou Gaussien

La seconde étape de la réduction de bruit est le filtre Gaussien. Cette technique est une variant d'une autre technique du flou. Ce processus requiert de pondérer chaque pixel autour de ses voisins. On utilise une matrice cette fois-ci pour stocker les valeurs des pixels. On multiplie donc chaque pixel de son voisin que l'on est en train de modifier par sa valeur correspondante dans cette même matrice.

$$\text{MatriceGaussienne} = ((1,2,1),(2,4,2),(1,2,1))$$

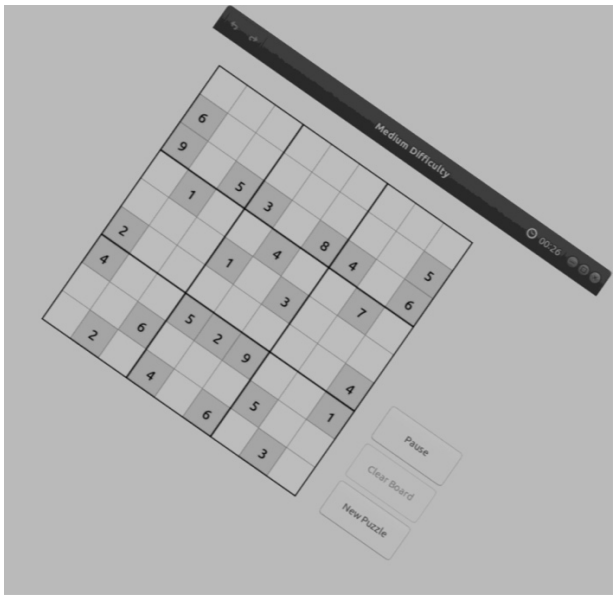


Image après la réduction de bruit

e. Seuil local

L'image doit être aplatie et nous devons compenser la disparité d'exposition. Certaines modifications que l'on a faite sur l'image depuis le début pourraient conduire à une mauvaise interprétation si ce dernier cas n'était pas pris en compte. Nous avons donc utilisé une technique de « seuillage local » au lieu d'une technique générale car certaines images peuvent avoir un niveau d'exposition différent. Pour résumer cette partie, on « binarise » l'image, ce qui signifie que le tableau de pixels en sortie contient uniquement des pixels ayant la valeur blanc ou noir. L'algorithme calcule ensuite le seuil ou la moyenne des 9 pixels autour du pixel courant. Il parcourt chaque pixel en lui attribuant la couleur blanche si le pixel actuel est au-dessus du seuil, noire sinon.

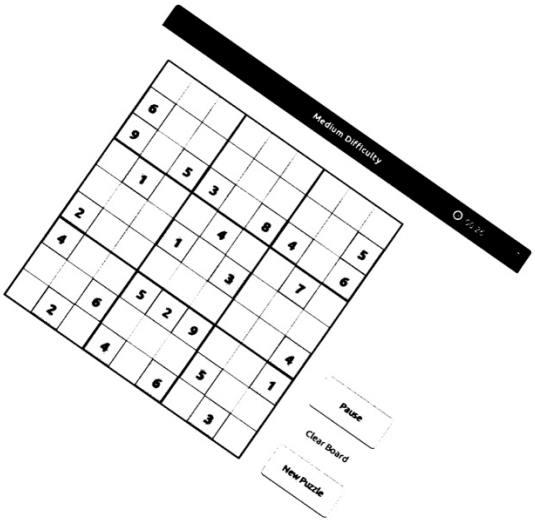


Image après la « binarisation » ou l'application du seuil local

f. Détection/Filtre de Sobel

La détection de Sobel, ou filtre de Sobel, est la dernière étape de notre prétraitement. Cette méthode permet de détecter les contours dans une image.
Pour calculer ce gradient, on utilise deux matrices pour obtenir les valeurs de G_x et G_y sur une image A tel que :

$$G_x = ((+1,0,-1),(+2,0,-2),(+1,0,-1)) \ . \ A$$
$$et \ G_y = ((+1,+2,+1),(0,0,0),(-1,-2,-1)) \ . \ A$$

Dépendant du la valeur calculée, on la compare à la valeur de gris (c'est-à-dire RGB (255/2)) et on la met en noir ou en blanc, dépendant de la valeur sur laquelle on tombe.

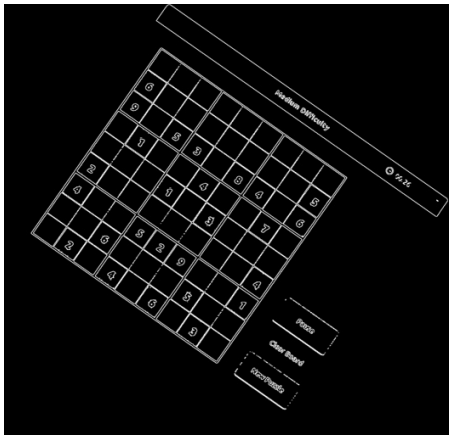


Image après la détection de Sobel

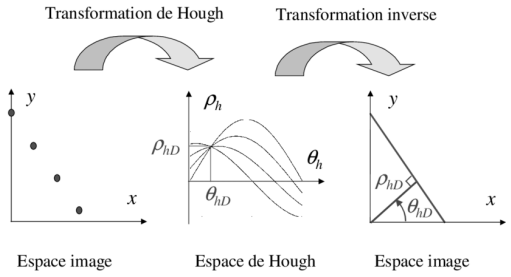
3. Détection de la grille

a. Transformation de Hough

Pour détecter la grille du sudoku, on a cherché plusieurs moyens, notamment utiliser un réseau de neurones entrainé pour cet usage (ce qui était trop complexe pour juste détecter une série de lignes), ou même juste analyser tous les pixels à l'horizontal puis à la verticale et regarder si la ligne contient un pourcentage important de pixels noirs (ce qui allait nous handicaper pour la suite, comme la rotation automatique). On a donc choisi d'utiliser la transformation de Hough.
L'algorithme de transformation sera moins efficace sans le filtre de Sobel, raison de plus pour son implémentation.
Il fonctionne en convertissant chaque point (x, y) du domaine cartésien d'une image donnée en un point polaire de coordonnées (θ, r). Chaque point correspond à une ligne dans l'image et chaque point en coordonnées polaires correspond à une ligne dans l'espace cartésien.
Pour démarrer l'algorithme, nous créons un nouveau tableau correctement dimensionné afin de stocker toutes les valeurs dans le repère polaire. Nous parcourons chaque pixel de l'image binarisée, et si le pixel est blanc, nous traçons la ligne qui lui correspond en utilisant la formule de l'image ci-dessous :

$$\textit{Espace image } (x_i, y_i) : \boldsymbol{y_i} = m\boldsymbol{x_i} + c$$
$$\textit{Espace de Hough } (m, c) : \boldsymbol{c} = -\boldsymbol{m}x_i + y_i$$

À mesure que le nombre de croisements au même point d'intersection augmente, il devient de plus en plus probable qu'une ligne se connecte.



Explication schématisée transformation de Hough

Nous effectuons ensuite une boucle dans la matrice des coordonnées polaires pour obtenir la valeur maximale, qui est en fait la plus grande quantité d'intersection de lignes. Il définira donc la ligne principale en coordonnées cartésiennes qu'il faudrait tracer sur l'image initiale.
Une fois que nous avons la valeur maximale, nous pouvons tracer et stocker toutes les lignes qui ont au moins la valeur égale à la moitié du maximum dans la matrice polaire.

Comme cette technique peut produire une très grande quantité de lignes, nous l'avons réduite en vérifiant, lors de l'ajout d'une nouvelle ligne, s'il y en a une qui est déjà placée approximativement à la même position dans la liste de toutes les autres lignes.

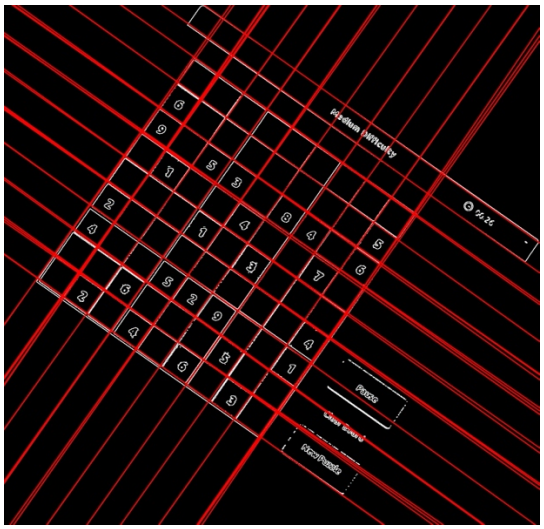


Image processus de la transformation de Hough, les lignes sont détectées

b. Rotation

Cette étape consiste à trouver l'angle de rotation. Notre algorithme permettant la rotation d'une image en mettant en entrée l'image et l'angle de rotation souhaité. Grâce à la transformation de Hough, on peut obtenir l'angle de rotation de l'image. On cherche donc l'angle maximal trouvé puis on le rentre dans la fonction de rotation. La rotation automatique n'est pas encore parfaite sur certaine image, c'est donc une des améliorations à faire sur la prochaine soutenance.

c. Détection des carrés

La détection des carrés est définie sur l'algorithme suivant :

1. Parcourir la liste des lignes pour trouver une intersection.

2. Quand on en trouve un, on boucle à nouveau pour trouver d'autres points d'intersection de la même ligne pour former un carré.

3. Inspecter et s'assurer que les distances sont à peu près les mêmes tout en négligeant les carrés qui ont une distance jugée trop petite définie par un seuil précis.

4. Si un carré est détecté, nous utilisons notre structure « SudokuCell » en fournissant ses valeurs de dimensions.

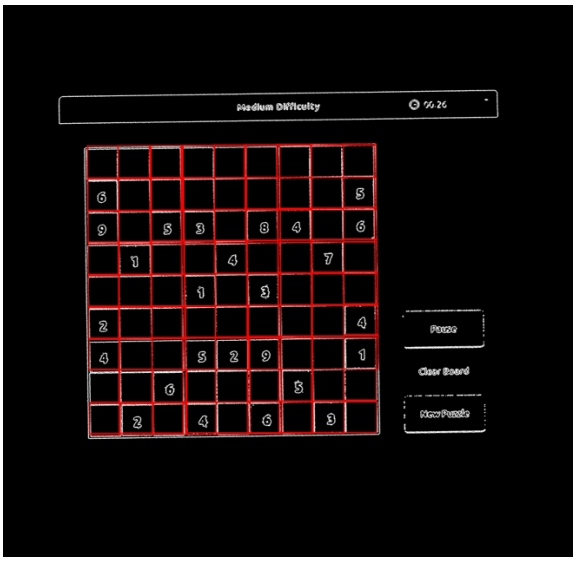
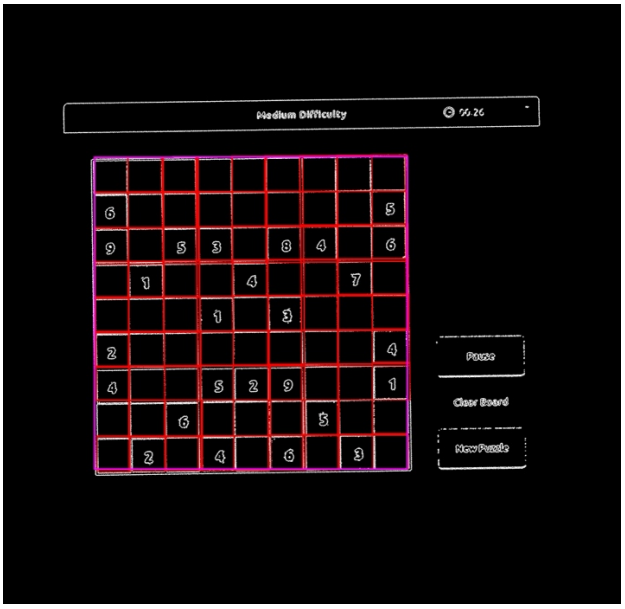


Image après la détection des carrés

Après avoir détecté chaque carré, nous devions trouver les points supérieurs gauche et inférieur droit du sudoku. Notre programme a parcouru chaque carré pour détecter celui qui avait les valeurs les plus proches de 0 pour le coin supérieur gauche. Nous avons découvert que cela fonctionnait bien, mais qu'il dépendait fortement de la qualité globale de l'image. Nous avons donc fixé un seuil de proximité avec le nouveau point pour résoudre ce problème pour l'instant mais il sera amélioré dans le futur. Le même processus a été appliqué pour déterminer le point en bas à droite du sudoku.
À ce stade, nous disposons d'un point de départ (x, y) et de deux distances prêtes pour l'algorithme de fractionnement d'image.



Détection de la grille du sudoku

d. Segmentation de l'image

Une fois que l'image initiale a été traitée et que la grille de sudoku a été détectée, nous avions besoin d'une fonction capable de séparer chaque cellule du sudoku en 81 sous-images extraites.

Ces dernières devaient être de la même taille (28x28 pixels) et créées de manière à ce qu'elles contiennent toutes un seul chiffre, si possible centré, pour faciliter la tâche du réseau de neurones chargé de détecter le chiffre en question. Ainsi, en utilisant toujours la bibliothèque SDL2_Image utilisée pour le prétraitement, nous avons commencé à créer un premier prototype de la fonction permettant une telle division des parties de l'image donnée. Cette fonction fonctionnait à partir des coordonnées d'un point dans l'image fournie en entrée au programme, ainsi que d'une valeur de longueur et de hauteur exprimée en pixels.

Lors de son exécution, le programme calculait la distance horizontale et verticale en pixels de la zone à subdiviser en petites images carrées de taille uniforme. Il stockait ensuite les dimensions de chacun des carreaux en divisant les dimensions précédemment calculées par 9 afin de séparer la zone en 81 sous-images.

Pour chacun des carrés créés, nous avons alloué une certaine portion de la zone générée à partir de l'image initiale fournie et défini un rectangle de taille correspondante à l'aide de la structure SDL_Rect. Ce rectangle était ensuite placé aux coordonnées de la tuile en cours de traitement sur l'image initiale et exporté vers un nouveau fichier au format BMP couvrant la zone du rectangle.

Les images étaient ainsi générées en progressant ligne par ligne dans la grille de sudoku au fur et à mesure que chaque cellule était partitionnée : nous avons donc nommé les fichiers selon les indices dans la grille de la cellule en cours de traitement. Cette

convention de nommage sera particulièrement utile plus tard lorsque nous devrons mettre chacune des valeurs identifiées par le réseau de neurones dans un fichier texte pour être traitées par notre solveur de sudoku.
Cette première version, assez simple à mettre en œuvre, a ensuite été légèrement améliorée afin de garantir l'uniformité des dimensions de chacune des sous-images générées. Cela nous permettrait de changer la résolution de chacune des images pour un carré plutôt plus petit afin de limiter le nombre de liens dans notre futur réseau de détection de nombres neuronaux. En particulier, nous envisagions de redimensionner chaque image à une résolution de 28x28 pixels, limitant ainsi le réseau de neurones à seulement 784 liens par nœud ou neurone au lieu de bien trop de connexions sans cette précaution. Une fois cette fonction terminée, il était relativement simple de l'intégrer après le traitement de l'image et de la grille de sudoku : elle est directement appelée dès que la détection de l'origine de cette dernière est déterminée.

5. Résolution du Sudoku

Pour la réalisation du solver, l'implémentation de l'algorithme était libre. Autrement dit, nous avions la latitude de réaliser l'algorithme de notre choix, que ce soit un algorithme de résolution basé sur les réseaux de neurones ou une méthode récursive. Après réflexion, nous avons opté pour la méthode la plus simple, déjà mise en œuvre par les élèves lors des TP du S2, à savoir une méthode récurrente.

L'idée derrière cet algorithme est simple. Le programme sélectionne un nombre entre 1 et 9, l'insère dans la case libre correspondant à l'index actuel de l'algorithme et effectue une série de tests sur la ligne, la colonne et la matrice 9x9 associée à cet index. Si tous ces tests sont concluants, le nombre est valide, et le programme poursuit récursivement avec une autre case. Cet algorithme gère efficacement les situations où un nombre a été mal positionné initialement, en raison d'un manque d'information. L'approche récursive permet de rectifier ces erreurs. Après avoir transposé l'algorithme du C en C++, nous avons constaté que le programme ne fonctionnait pas correctement. Le problème venait du fait que le système de matrice de type grid[SIZE][SIZE] n'acceptait pas les arguments sous la forme grid[x][y], mais plutôt grid[y][x]. Après avoir inversé les variables, tout fonctionnait comme prévu.

Initialement, nous pensions que concevoir le reader serait une tâche aisée, étant donné notre expérience précédente avec la création de filestreams pour lire les données depuis un document. Ce fut une erreur de jugement.

Le programme accepte en entrée le chemin d'accès d'un fichier et la matrice à remplir. Il lit chaque caractère un à un tout en déplaçant deux curseurs pour naviguer dans la

matrice. Si un caractère n'est pas un chiffre ou un point, il est ignoré. Si c'est un point, un zéro est inséré dans la matrice à l'index correspondant. Nous avons choisi cette approche plutôt que d'utiliser une matrice de caractères pour éviter les conversions de type lors de la résolution.

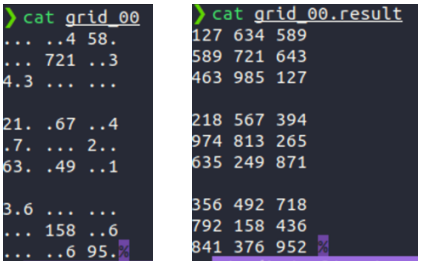


Image avant et après la résolution du sudoku grid00 donné dans le cahier des charges

L'outil d'écriture dans un fichier était, de loin, le plus simple à mettre en œuvre, car il a été développé après que tous les autres outils associés au solver étaient opérationnels. Son fonctionnement est « straightforward » : il lit la matrice et écrit son contenu dans un fichier, ajoutant des caractères spécifiques et des espaces selon certaines conditions.

Lors de notre première tentative, nous avons utilisé une sous-fonction renvoyant directement la grille. Cette approche consommait trop de mémoire car chaque matrice était recréée. À la fin de l'exécution, cinq matrices de taille 9x9 étaient stockées en mémoire. Des experts nous ont suggéré d'utiliser des types void et de ne créer qu'une seule matrice, ce qui a optimisé la performance.

5. Réseaux de neurones et XOR

a. Structure

Avant de commencer à travailler sur notre réseau de neurones, il nous a fallu définir sa structure. Nous définirons notre réseau de neurones par un liste contenant le nombre de neurones de chaque couche du réseau, une liste de matrices représentant les poids associés à chaque couche du réseau ainsi qu'une liste de vecteurs (représentés sous forme de matrices ayant une colonne) représentant les biais associés à chaque neurone du réseau. La première couche du réseau de neurones n'a pas de poids ou de biais associé. Tout cela implique d'avoir défini au préalable la structure d'une matrice ainsi que d'avoir implémenté les opérations sur les matrices qui nous seront utiles lors du traitement des informations par cette dernière. Nous représenterons ainsi une matrice par son nombre de lignes, son nombre de colonne ainsi qu'une liste des données qui la composent.

Ainsi, les neurones sont implicitement définis par le réseau de neurones auquel ils appartiennent. En effet, un neurone est défini par ses liens avec les autres neurones, les

poids qu'ont les entrées dans ce neurone, le biais appliqué à la somme pondérée des entrées du neurone ainsi que par la fonction d'activation du neurone qui est dans notre réseau identique pour tous les neurones.

Dans le cas de notre réseau de neurones, nous avons choisi d'utiliser comme fonction d'activation des neurones la fonction sigmoïde. Après nous être documentés sur les fonctions d'activations que nous pourrions appliquer à notre réseau, nous avons décidé de choisir celle-ci puisqu'elle permet d'étaler de manière assez uniforme les valeurs entre 0 et 1, tout en permettant de percevoir une différence lorsque l'on modifie les paramètres des neurones, nous facilitant ainsi son apprentissage. Elle possède aussi l'avantage d'être facilement dérivable, ce qui nous aidera lorsque l'on va implémenter le programme d'entrainement.

Le réseau de neurones prendra comme entrée un vecteur représenté sous forme de matrice et donnera en sortie un autre vecteur représenté lui aussi sous forme de matrice. Ce dernier sera composé de valeurs comprises entre 0 et 1 que l'on évaluera comme probabilités (une valeur de sortie supérieure à 0.5 sera interprétée comme une réponse positive, celles strictement inférieures à 0.5 seront interprétées comme réponse négative).

Dans le cadre de la porte XOR, nous avons choisi de créer un réseau de 3 couches. La première couche (la couche d'entrée) est composée de deux neurones. Ils constituent l'information entrée dans le réseau de neurone. La dernière couche (la couche de sortie) est constituée d'un seul neurone. Dans le cas du réseau de neurones de reconnaissant la porte XOR, la matrice de sortie ne sera constituée que d'une valeur représentant la probabilité que l'entrée appliquée à la porte XOR nous donne 1. Nous avons pour la couche intermédiaire choisi de mettre 5 neurones. Ce choix est arbitraire. Par ailleurs, nous avons constaté qu'à nombre égal de neurones, un réseau ayant peu de couches de beaucoup de neurones est plus efficace qu'un réseau ayant plusieurs couches de moins de neurones, d'où notre choix.

b. Principe de fonctionnement

i. Initialisation

Nous initialisons un réseau de neurones à partir d'une liste contenant les nombres de neurones par couche de neurones souhaités par l'utilisateur. Ensuite, nous créons les listes de matrices pour les poids et les biais. Les dimensions des matrices sont initialisées en fonction de la largeur de chaque couche. Nous avons choisi d'initialiser tous les biais à 0 et d'initialiser les poids autour de 0 avec l'initialisation de Xavier (ou initialisation de Glorot). Nous avons fait ce choix car la pratique montre qu'elle permet d'une certaine manière de mieux démarrer la phase d'entrainement.

ii. Propagation de l'information

À présent, définissons comment notre réseau va à partir d'un vecteur donné en entrée, nous fournir le vecteur réponse lui correspondant. Nous allons appliquer à ce vecteur une technique appelée "feedforward". Elle consiste en la propagation de l'information au travers de réseau par couche de neurones. L'information initiale est l'entrée du réseau de neurone. Pour chaque neurone de la couche suivante, on applique à la somme pondérée des neurones d'entrée (somme des multiplications de chaque sortie de neurone par le poids lui correspondant dans le neurone suivant puis ajout du biais du neurone suivant) la fonction d'activation du neurone (ici sigmoïde). En appliquant l'opération à tous les neurones de la couche, on obtient un vecteur, sur lequel on va de nouveau répéter l'opération avec la couche de neurones suivante. Le résultat obtenu par l'opération appliquée sur la dernière couche de neurones nous donne la sortie du réseau de neurones. Étant donné que nous travaillons avec des matrices, l'opération se fait d'un coup pour tous les neurones d'une couche. La complexité est semblable mais cela rend son implémentation bien plus simple.

iii. Entraînement

Pour entrainer un neurone, il faut pouvoir déterminer le degré d'erreurs qu'il fait. Mesurer le pourcentage de réussite ne nous permet pas d'affiner les paramètres du réseau de manière efficace. En effet, si le réseau donne toujours une réponse juste mais que cette réponse est issue d'une probabilité limite (par exemple 0.51 pour une réponse positive), il va être difficile d'affiner les paramètres. Ainsi, nous allons utiliser la fonction coût quadratique, faisant la différence entre le vecteur de sortie obtenue et le vecteur de sortie attendu. Elle nous permet d'affiner les probabilités de façon à minimiser toute ambigüité (une réponse positive sera après entrainement très proche de 1). Il en existe d'autre mais celle-ci fonctionne bien avec notre modèle.

Ensuite, pour mesurer la différence d'erreur causée entre deux poids différents sur un neurone interne, il nous faut pouvoir remonter la différence d'erreur en sortie jusqu'au neurone en question. Nous utiliserons donc la technique de "backpropagation". Les formules nous ayant permis d'implémenter cet algorithme nous viennent d'études scientifique que vous trouverez en annexe. On minimise ensuite l'erreur causée par le paramètre en faisant une descente de gradient.

Ainsi, il nous suffit désormais de répéter l'opération de mise à jour des paramètres plusieurs fois sur un ensemble de données de référence. Étant donné que la fonction XOR n'a que 4 cas possible, nous avons directement implémenté la fonction d'entrainement de

sorte que pour chaque époque d'entrainement, elle entraine le réseau sur chacun des 4 cas possibles. Plus tard, il nous faudra implémenter une fonction capable d'extraire des données de références à partir de fichiers car les possibilités d'images représentant des chiffres ne sont non pas infinies mais très grandes.

c. Gestion de fichiers

i. Problématique

Un problème se pose avec ce que l'on a énoncé jusqu'à présent : il faut initialiser et entrainer un réseau de neurones à chaque fois que l'on souhaite l'utiliser pour analyser des données. En effet, même après avoir entrainé un réseau plusieurs fois, les informations qu'il contient disparaissent à la fin de l'exécution du programme. Dans le cas du réseau de neurones de la porte XOR, ça ne coute qu'une ou deux secondes et ça n'est pas dramatique. En revanche, dans le cas de la reconnaissance des chiffres, initialiser le réseau et l'entrainer sur plusieurs dizaines de milliers d'images risque d'être assez problématique.

ii. Solution

Pour Résoudre ce problème, nous avons réalisé un gestionnaire de fichier. Celui-ci nous permettra d'écrire les informations caractéristiques d'un réseau de neurones dans un fichier, d'initialiser un réseau de neurones dans un fichier, d'utiliser un réseau de neurones dont les informations sont comprises dans un fichier ainsi que d'entrainer un réseau de neurones contenu dans un fichier et de mettre à jour les informations qui y sont inscrites. Pour ce faire, Nous allons écrire toutes les informations du réseau dans un fichier binaire. Étant donné que nous travaillons avec des structures (et même des listes de structures), il nous a fallu utiliser la méthode de sérialisation des données pour les écrire dans un fichier binaire et celle de la désérialisation pour les lire. Ainsi, les entraînements que nous faisons sur le réseau de neurones ne sont pas fait en vain.

6. Conclusion

En conclusion, les objectifs fixés par le cahier des charges donné ont tous été atteints, voire même dépassés dans certains domaines. En effet, la partie traitement d'image est entièrement terminée et inclut bien sûr la capacité de charger une image et d'en retirer les couleurs. La rotation manuelle est également possible grâce à une commande définie à partir de notre fichier exécutable qui sera montrée lors de la présentation. De plus, la grille de sudoku est entièrement détectée et la séparation des 81 cellules en plusieurs sous-images est achevée. Notre fichier exécutable final peut également résoudre une grille de sudoku donnée en entrée par un fichier texte et conforme aux spécifications indiquées dans le cahier des charges. Enfin, un réseau neuronal capable d'effectuer l'opération "ou

exclusif" sur des mots binaires a été mis en place avec succès. La sauvegarde et le chargement des poids et biais de ce dernier ont déjà été réalisés et fonctionnent comme prévu.
Pour la prochaine soutenance, il nous restera à créer l'interface graphique, qui ne sera pas une mince affaire, et le réseau de neurones pour identifier les chiffres. Et si on en a la possibilité, nous créerons un site internet faisant la promotion de notre projet.