

EPITA

Rapport de Projet - projet S3

SUDOKU SOLVER-OCR

Membres : Hugo Schlegel, Antoine Riquet, Angelina Kuntz, Mathéo Romé

Groupe : Hanabi



Contents

Introduction	4
Mathéo Romé	4
Présentation	4
Première soutenance	4
Début du projet	4
Réseau de neurones	4
Seconde Soutenance	7
Modification de la structure	7
Entrainement	8
Détection des numéros	9
Ajustement	9
Conclusion personnelle	10
Hugo Schlegel	11
Présentation	11
Première soutenance :	11
Début du projet	11
Prémices et idées d'algorithmes	11
Algorithme de ligne de partage des eaux	11
Détection des bords et découpe des cases	13
Prochaine Soutenance	14
Seconde soutenance :	15
Amélioration et passage du watershed en itératif	15
Normalisation et nettoyage des cases	15
Impression du résultat dans une grille	16
Alimentation de la bibliothèque	17
Résolution de bugs et plomberie	17
Améliorations possibles	17
Conclusion personnelle	17
Angelina Kuntz	18
Présentation	18
Première soutenance	18
Début du projet	18
Rotation	18
Interface graphique	20
Bibliothèque	22
Prochaine Soutenance	22
Seconde soutenance	22
Bibliothèque	22
Interface	23
Rotation automatique	26
Conclusion personnelle	28

Antoine	29
Présentation	29
Première soutenance	29
Début de projet	29
Solver de Sudoku	29
Traitement d'image	30
Seconde soutenance	33
Finalisation du traitement d'image	33
Assistance et assemblage de l'ensemble du projet	35
Conclusion personnelle	36
Conclusion générale	38
Le groupe HANABI	38

Introduction

Nous sommes le groupe Hanabi, mené par Mathéo Romé et constitué des membres Hugo Schlegel, Angelina Kuntz et Antoine Riquet. La dénomination *Hanabi* vient du japonais fleur de feu qui se traduit par feu d'artifice. Notre équipe se veut d'une ambiance festive et cette appellation se tient en référence aux traditionnels festivals japonais. Nous avons décidé de garder le même groupe que l'année précédente, car nous nous entendions bien et que nous connaissions chacun notre façon de travailler. Nous avons ainsi pu facilement répartir les tâches en fonction de chacun.

Mathéo Romé

Présentation

Je me présente je suis Mathéo Romé et je me suis occupé essentiellement de la partie reconnaissance de caractère, c'est-à-dire de la création, de l'entraînement et de tout ce qu'il y a autour d'un réseau de neurones. Pour la première soutenance je m'étais occupé de faire un "proof of concept" en créant un réseau capable d'apprendre la fonction logique XOR ainsi que des fonctions permettant de sauvegarder un réseau de neurones pour l'utiliser plus tard. Pour la seconde soutenance, je m'étais occupé de modifier le réseau de neurones afin de lui faire apprendre à reconnaître un caractère dans une image ainsi que de l'entraîner en créant des fonctions créant des "bases de données" à partir d'images.

Première soutenance

Début du projet

Pour débuter ce projet, je me suis beaucoup renseigné sur le langage C, pour lequel j'étais encore novice, ainsi sur les réseaux de neurones. J'ai voulu être bien sûr de comprendre leurs fonctionnements avant de commencer à en créer un.

Réseau de neurones

Un réseau de neurones, c'est quoi ?

Pour la première soutenance, il nous fallait une preuve de concept de notre réseau de neurones. Pour cela, j'ai effectué un réseau apprenant le "ou exclusif" (ou XOR).

Un réseau de neurones est une structure composée de neurones (eh oui !) et reliée entre eux permettant de résoudre des problèmes complexes comme de la reconnaissance d'image, ce qui nous intéresse ici. Les neurones sont ordonnés par "couches" : une couche pour les différentes entrées (les informations que l'on donne à notre programme), une couche de sortie composée des neurones indiquant la réponse du réseau au problème posé, et enfin d'une ou plusieurs couches "cachées" permettant d'effectuer des transformations non-linéaires sur les valeurs d'entrées. Plus il y a des couches cachées plus le réseau pourra résoudre des problèmes complexes. Dans le cadre de notre projet, nous n'aurons besoin que d'une couche cachée.

Chaque neurone d'une couche est relié à tous les neurones de la couche suivant par des "poids" d'une valeur aléatoire à la création du réseau. Un poids peut être vu comme une flèche reliant un neurone à un autre.

Chaque neurone est également composé d'une valeur et d'un biais (sauf sur la couche d'entrée). Le biais est également déterminé aléatoirement à la création du réseau. La valeur de chaque neurone est déterminée par la somme de toutes les valeurs des neurones de la couche précédente chacune multipliée par le poids associé.

On ajoute enfin à cette somme le biais associé au neurone dont on cherche la valeur.

Un petit exemple

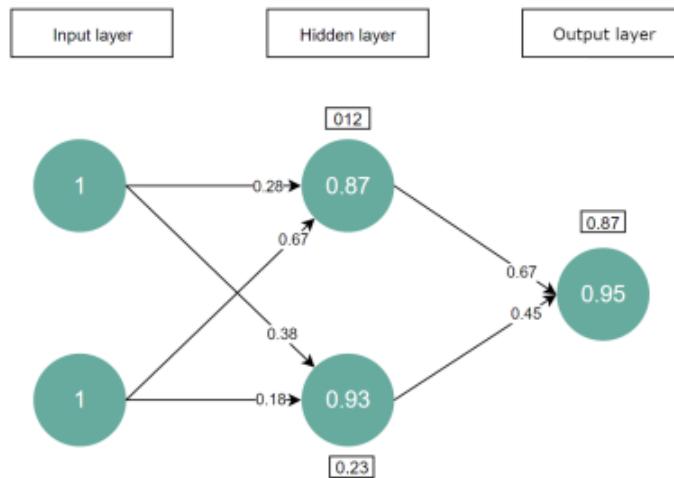


Figure 1: Exemple d'un réseau de neurones

Dans l'exemple ci-dessus, les valeurs numériques ne sont là qu'à titre d'exemple et ne sont pas cohérentes entre elles. Néanmoins, si l'on voulait calculer la valeur du premier neurone de la couche caché, on obtiendrait ce calcul : $1 \cdot 0,28 + 1 \cdot 0,67 + 0,12$.

La fonction d'activation

La prochaine étape est d'appliquer une fonction d'activation sur le réseau de neurones. Cette fonction permet "modifier" une donnée de manière non-linéaire, c'est-à-dire changer la façon dont on considère la valeur. Par exemple "50% des clients achète du chocolat" peut être considéré comme "50% des clients aiment le chocolat".

Il existe plusieurs types de fonctions d'activation chacune ayant son utilité en fonction du type de réseau que nous souhaitons implémenter.

Pour le réseau de neurones reconnaissant un XOR, j'ai utilisé la fonction d'activation sigmoïde renvoyant pour un x donné une valeur entre 0 et 1 permettant un résultat binaire, ce qui est ici recherché. Pour le réseau de neurones chargé de reconnaître un chiffre dans une image, je prévois d'utiliser la fonction softmax permettant d'avoir une probabilité donc également un résultat entre 0 et 1, mais dans le cas d'un où l'on a plusieurs neurones sur la couche de sortie, la somme de leur valeur sera 1. Il suffira donc de choisir le neurone avec la plus grande probabilité.

Ces deux fonctions ne sont pas implémentées de la même façon : pour une sigmoïde, il faut l'appliquer après chaque calcul de la valeur d'un neurone du réseau tandis que le softmax ne sera appelé qu'une fois et ne modifiera que les valeurs des neurones sur la couche de sortie.

L'apprentissage

La partie intéressante arrive maintenant : comment un ensemble de nombres sans rapport apparent entre eux peuvent-ils apprendre ?

Il y a plein de manières de faire apprendre un tel réseau, mais la solution utilisée ici se base sur le modèle de la rétropropagation du gradient ou backtracking en anglais.

La première étape, commune à beaucoup de modèles de réseau de neurones est la "propagation avant" ou feed forward en anglais. Cette étape consiste tout simplement à calculer toutes les valeurs des neurones en utilisant le calcul mentionné ci-dessus ainsi que la fonction d'activation.

La rétropropagation consiste à ajuster la valeur des poids en fonction de l'écart entre le résultat attendu et le résultat obtenu à la sortie du réseau puis d'utiliser cet écart pour calculer l'écart des valeurs des neurones de la couche précédente. On utilise ensuite ses valeurs pour modifier nos poids et nos biais, en parcourant le réseau de la sortie vers l'entrée, d'où le nom de rétropropagation.

Cette fonction va ainsi essayer de pousser le réseau dans la bonne direction pour résoudre le problème.

Cette fonction a été pour moi la plus compliquée à cause de sa difficulté de compréhension due aux nombreuses formules mathématiques impliquées dans les explications de son fonctionnement.

La fonction d'erreur

Afin de mesurer les performances du réseau, j'ai également implémenté la fonction d'erreur quadratique moyenne ou mean squared error en anglais qui va mesurer le taux d'erreur de notre réseau et nous renvoyer une valeur entre 0 et 1. Plus celle-ci est proche de 0 plus le réseau est optimal.

Le résultat final

Avec toutes ses composantes, nous allons enfin pouvoir entraîner notre réseau de neurones !

Cet entraînement se découpe en "époque" ou "epoch" en anglais qui correspond au nombre d'essais que le réseau effectue. Chacune de ces époques va tester les quatre combinaisons du XOR (0-0, 0-1, 1-0, 1-1) et ajuster les poids et les biais en conséquence. La somme des erreurs de ces quatre passages dans le réseau va également déterminer le taux d'erreur de la génération.

Ce schéma va se répéter pour un nombre de générations donné, ou dans notre cas tant que l'erreur est supérieure à 0,001.

```
=====
Epoch n.0 :
0.000000 XOR 0.000000 = 0.809272, expected : 0.000000
0.000000 XOR 1.000000 = 0.703977, expected : 1.000000
1.000000 XOR 0.000000 = 0.790082, expected : 1.000000
1.000000 XOR 1.000000 = 0.806770, expected : 0.000000
Error rate = 0.718748
=====
Epoch n.300 :
0.000000 XOR 0.000000 = 0.036557, expected : 0.000000
0.000000 XOR 1.000000 = 0.976046, expected : 1.000000
1.000000 XOR 0.000000 = 0.976057, expected : 1.000000
1.000000 XOR 1.000000 = 0.000023, expected : 0.000000
Error rate = 0.001242
=====
TRAINING COMPLETED!
Epoch needed : 364           Error rate : 0.000999
=====
```

Figure 2: Résultat avant/après l'entraînement

Une fois le taux d'erreur assez faible, j'ai créé une fonction permettant de sauvegarder les éléments les plus importants du réseau : les poids et les biais : elles sont stockées dans un fichier texte permettant plus tard au moyen d'une autre fonction de recréer le réseau sans avoir besoin de l'entraîner.

Seconde Soutenance

Pour cette soutenance, je me suis occupé de transformer notre réseau de neurones apprenant un XOR en un réseau capable d'apprendre à reconnaître une image et qui nous sera donc plus utile pour résoudre un sudoku.

Modification de la structure

J'ai tout d'abord commencé par modifier la structure même de notre réseau de neurones. En effet la structure du réseau de neurones pour un XOR était très simpliste avec seulement, je le rappelle 4 neurones d'entrée, 3 sur la couche caché et 1 seul sur la sortie. Or pour notre nouvelle version du réseau de neurones, il nous fallait autant de neurones d'entrée que de pixel sur l'image possédant que nous voulions déterminer et 9 sorties (1 neurone par possibilité de résultat en excluant le 0 qui n'est jamais présent sur une grille de sudoku).

Pour la couche d'entrée j'ai tout d'abord pensé à utiliser une matrice représentant l'image mais l'utilisation d'un tableau uni-dimensionnelle s'est finalement révélée plus intuitive dans la manipulation d'index.

Pour la couche cachée, il n'y a pas de montant fixé mais nous en reparlerons plus tard dans ce rapport.

La modification de la couche de sortie est le plus gros changement sur le réseau en général : passer d'un seul neurone à plusieurs changes à différents endroits l'algorithme de rétropagation

notamment pour la modification des poids entre la couche sortie et la couche cachée ainsi que les biais de cette dernière.

Une fois ces modifications faites, la prochaine étape dans la métamorphose du réseau était le changement de fonction d'activation dans la propagation avant.

Je pensais utiliser la fonction softmax sur la couche de sortie à la place de la fonction sigmoïde mais l'utiliser nécessitait l'implémentation de sa dérivée pour la rétropropagation et une fois la fonction implémentée les tests n'ont pas été très concluants. J'ai donc préféré conserver la fonction sigmoïde qui posait moins de difficulté et qui, au contraire, donnait des résultats plus intéressants.

J'ai enfin modifié l'initialisation du réseau de neurones en générant des nombres aléatoires entre -1 et 1 au lieu d'entre 0 et 1. Pour le XOR, générer un nombre dans un tel intervalle ne posait pas de problème mais pour détecter une image, cela créait des situations dans lesquelles la sigmoïde ne renvoyait que des 1 (cela était causé par l'apparition de nombre trop éloigné de l'intervalle de valeur habituelle donné à une sigmoïde)

Entrainement

Une partie importante de mon travail pour cette soutenance a été d'entrainer le réseau de neurones.

Pour cela j'ai créé une struct supplémentaire séparée de mon réseau de neurones composé de 2 tableaux à deux dimensions : un tableau contenant les images d'entraînement et un tableau contenant les images de tests. Pour remplir cette structure, j'ai écrit un algorithme cherchant dès un dossier et ses sous-dossiers donnés toutes les images, préalablement nommées "nb.bmp" avec nb correspondant au numéro représenté sur l'image. L'algorithme va ainsi ouvrir une image, parcourir les pixels un à un en mettant dans le tableau 0 ou 1 selon que le pixel est respectivement blanc ou noir puis de fermer l'image pour passer à la suivante. Le but étant d'entraîner ou de tester le réseau, il était également important de savoir quel chiffre contenait l'image afin de pouvoir savoir quand notre réseau se trompait. Pour cela j'ai réservé la première case du tableau (l'index 0) pour y placer le label. Pour obtenir ledit label, j'ai simplement eu à récupérer les 4 derniers caractères du path de l'image (car le path fini en nb.bmp).

Nous avons donc deux tableaux, chacun ayant une fonction spécifique : le tableau d'entraînement, composé de 70% du total d'images de la bibliothèque a pour but d'être utilisé à fin... d'entraînement, cela semble plutôt logique : on envoie des images dans le réseau, on regarde ce qu'il nous renvoie grâce à la propagation avant puis on modifie les poids et les biais en fonction de ce résultat grâce à la rétropropagation.

Le but est de passer plusieurs fois, de faire plusieurs époques, sur le tableau d'entraînement tout en évitant le surapprentissage où le réseau serait incapable de donner une réponse cohérente si l'image n'est pas très proche des images d'entraînement.

Pour éviter cela, on vérifie les performances du réseau de neurones sur le second tableau, indépendant du premier et composé des 30% d'images restantes. Cette fois-ci on ne s'intéresse qu'aux résultats donc une simple propagation avant est suffisante. On compare ensuite les résultats obtenus avec ceux attendus pour obtenir une idée du réseau.

Initialement j'avais structuré mon réseau de neurones pour des images en 16*16 pixels et à cet effet, Angelina avait créé une bibliothèque composée de chiffres de 1 à 9 en plein de polices différentes. Malheureusement après concertation avec Hugo et ses résultats sur l'extraction des différents numéros de la grille de sudoku en les réduisant ensuite à une image en 16*16 nous avons décidé d'abandonner l'idée pour passer à une taille supérieure.

Nous avons donc ensuite choisi la taille de 28*28 pour plusieurs raisons, tout d'abors parce que cette taille nous donnait de beaux résultats sur les numéros d'Hugo mais aussi parce que cela nous donnait accès à une toute nouvelle possibilité d'entraînement : la MNIST (Modified/Mixed National Institute of Standards) une bibliothèque de 60 000 images d'entraînement et 10 000 images de tests. Cette bibliothèque est ainsi composée d'images de chiffre manuscrit en tout genre. J'ai donc commencé à entraîner mon réseau de neurones uniquement avec cette bibliothèque. Les résultats étaient en sois très bon avec environ 90% de taux de réussite sur le set de test mais dès que l'on testait sur les images renvoyées par Hugo, ce taux s'effondrait à environ 50%. Pour pallier ce problème, Angelina a créé une deuxième version de la bibliothèque avec cette fois-ci des images en 28*28. Après de nombreux tests nous avons malheureusement décidé de retirer cette bibliothèque ainsi que la MNIST pour ne constituer qu'une bibliothèque composée de différentes images de numéros renvoyé par l'algorithme d'Hugo sur différentes grilles de sudoku trouvé sur internet. Cette solution nous a ainsi permis d'avoir de bien meilleurs résultats quant à la détection des numéros de la grille.

Détection des numéros

Pour détecter les différents symboles, j'ai créé une fonction semblable à celle qui remplit les tableaux d'entraînements et de tests mais qui va construire dans un fichier texte la grille au fur et à mesure des images. Grâce à Hugo, les images vides possèdent un marqueur (pixel de valeur 42) permettant de ne pas passer le réseau de neurones dessus mais de simplement placer un "." dans le fichier texte. Pour les images avec un numéro, on la charge tout simplement dans le réseau de neurones et on fait une propagation avant pour récupérer la réponse du réseau que l'on place ensuite dans le fichier texte.

Une fois toutes les images parcourues, la grille est prête à être résolue grâce au solveur d'Antoine.

Ajustement

La plus grosse partie du travail a été d'ajuster le réseau de neurones pour le rendre capable de reconnaître dans les images renvoyées par Hugo. Nous avons déjà parlé de différentes bibliothèques que nous avons utilisées, mais il y a bien d'autres facteurs qui rentrent en compte. Par exemple le nombre des neurones sur la couche cachée ou le taux d'apprentissage. Le nombre de neurones sur la couche cachée a pour effet de modifier le nombre de liaisons entre les couches et ainsi d'augmenter ou de réduire la quantité de calcul nécessaire pour un passage (allez ou retour) dans le réseau et donc de potentiellement augmenter la durée de l'entraînement. Après de nombreux essais nous avons choisi de mettre 150 neurones sur la couche cachée.

Pour ce qui est du taux d'apprentissage, il correspond à la "vitesse" de changement du réseau et est une valeur comprise généralement entre 0 et 1. Pour bien comprendre cette notion nous pouvons comparer notre réseau à une piste où l'on peut avancer à chaque fois d'une certaine

distance. Le but est d'atteindre un objectif en avançant en avant ou en arrière. Si à chaque "tour" on parcourt de trop grandes distances on risque de ne jamais pouvoir s'approcher de l'objectif et au contraire si la distance est trop courte, on mettrait trop de temps pour atteindre l'objectif. Pour notre réseau de neurones, nous avons choisi un taux d'apprentissage de 0.01 ce qui peut paraître peu mais qui donne néanmoins des résultats concluants.

Chaque entraînement donnera un résultat différent et donc il a parfois fallu faire de nombreux essaies afin de parvenir à un réseau de neurones apprenant de façon acceptable. En effet parfois le réseau avait des difficultés à apprendre à reconnaître un 8 ou bien confondait les 6 et les 9. Entrainer un réseau de neurones repose en partie sur l'aléatoire, mais une fois entraîné, le réseau devient complètement déterministe laissant ainsi aucune place à l'aléatoire quant à la résolution d'une même grille de sudoku deux fois de suite.

```

Got :6,6 was expected
Got :2,2 was expected
Got :3,3 was expected
Got :7,7 was expected
Got :9,9 was expected
Got :5,5 was expected
Got :7,7 was expected
Got :1,1 was expected
Got :3,3 was expected
Got :4,4 was expected
Got :8,8 was expected
Got :7,7 was expected
Got :8,8 was expected
Got :3,3 was expected
Got :4,4 was expected
Got :5,5 was expected
Got :6,6 was expected
Got :8,8 was expected
Got :2,2 was expected
Got :3,3 was expected
Got :1,1 was expected
Got :6,6 was expected
Got :7,7 was expected
Got :2,2 was expected
Got :5,5 was expected
Got :7,7 was expected
Got :3,8 was expected
Got :1,1 was expected
Got :2,2 was expected
Got :6,6 was expected
Got :1,1 was expected
Got :5,5 was expected
Got :7,7 was expected
Got :6,8 was expected
Got :2,2 was expected
Got :3,3 was expected
Got :8,8 was expected
Got :9,9 was expected
Got :2,2 was expected
Got :3,3 was expected
Got :4,4 was expected
Got :6,6 was expected
Got :9,9 was expected
Got :1,1 was expected
Got :3,3 was expected
Got :5,5 was expected
Got :1,1 was expected
Got :2,2 was expected
Got :3,3 was expected
Got :7,7 was expected
Got :8,8 was expected
Got :9,9 was expected
Got :1,1 was expected
Error : 5.128205

```

Figure 3: Exemple d'un entraînement sur notre réseau final : 5% d'erreur (la totalité des tests n'est pas présent)

Conclusion personnelle

Pour conclure sur ce projet, je l'ai trouvé très intéressant bien qu'un peut frustrant quand le réseau de neurones ne marchait pas ou donnais des résultats absurdes sans que l'on ne sache pourquoi. La construction d'un réseau de neurones a été une épreuve passionnante tant sur la partie recherche est compréhension de son fonctionnement que de sa mise en oeuvre. A part cela ce projet m'a permis de bien développer les fonctionnements de base du langage C et me rend curieux pour la suite.

Hugo Schlegel

Présentation

Je me présente : Hugo Schlegel, ancien chef de groupe pour le projet S2 j'ai pour ce projet laissé ma place à Mathéo Romé, histoire de pouvoir faire tourner le rôle au sein du groupe, celui-ci étant en réalité *pour le show*, comme le diraient nos amis les anglais.

Pour cette première soutenance, je me serai consacré à la détection de la grille de sudoku ainsi que son découpage en plusieurs petites case. Vous l'aurez donc compris ma partie aura été pleinement consacrée à l'image, à l'inverse de celle de Mathéo.

Première soutenance :

Début du projet

Commencer à travailler sur ce projet aura été assez compliqué en toute honnêteté. Sans réelle expérience avec le C, le traitement d'image et la SDL (Simple DirectMedia Layer), il était difficile de trouver où commencer. Mon travail aura donc au début, vous l'aurez compris, été principalement de la recherche : *"Comment marche le C ? Que sont les pointeurs ? Comment fonctionne la SDL ? Quels algorithmes étudier ?"*. Mais, avec un peu de motivation et de bonne fois, j'ai au final facilement pu me mettre à travailler sur le projet.

Prémices et idées d'algorithmes

Je me devais donc pour la première soutenance d'être capable de détecter la grille du sudoku dans une image en noir et blanc, prétraitée par Antoine. J'ai donc passé du temps à chercher divers algorithmes de détection de formes et l'un d'entre eux m'aura rapidement attiré : celui de la transformée de Hough, un algorithme de détection de lignes dans les images, exactement ce dont j'avais besoin. L'algorithme consiste en représenter chaque point (x, y) de l'image dans un plan à deux dimensions (p, O) tel que : $p = x * \cos(O) + y * \sin(O)$. La résultante de cet algorithme est un plan contenant plusieurs sinusoïdes correspondantes aux différents points de l'image de base. La transformée de hough veut alors que les points de ce plan de Hough où se croisent grand nombre de sinusoïdes soient en réalité des lignes dans le plan original (x, y). J'avais plutôt bien compris cet algorithme et m'étais mis à coder sans perdre plus de temps jusqu'à heurter un problème majeur, j'étais dans l'incapacité la plus totale d'exploiter les résultats ressortants de l'algorithme, je voyais bien les lignes sur le plan de Hough mais je ne savais pas comment les exploiter avec du code. Et c'est ainsi que je me suis vite détourné de cet algorithme, sans pouvoir progresser sur cette voie. C'est alors que, sur conseil d'un camarade qui faisait face aux mêmes problèmes que moi, je me suis tourné vers une version simplifiée de l'algorithme de ligne de partage des eaux (ou watershed segmentation en anglais).

Algorithme de ligne de partage des eaux

Les algorithmes de ligne de partage des eaux se veulent être des algorithmes qui *considèrent une image à niveaux de gris comme un relief topographique, dont on simule l'inondation* (d'après wikipédia). Ici le travail serait pour moi grandement simplifié, en effet, l'image étant binarisée, ses pixels ne peuvent être que noir ou blancs. Mon but avec cet algorithme était donc de pouvoir

déTECTer tous les différents éléments (sous forme d'aglomérats de pixels) présents dans mon image et ainsi de pouvoir déterminer lequel parmi eux est la grille afin de pouvoir l'isoler et continuer les traitements sur elle seule.

Mon algorithme consiste alors à parcourir l'image, pixel par pixel, jusqu'à tomber sur un pixel noir non libellé, dans quel cas j'assigne une valeur à ce pixel (je le labelle) et j'appelle sur celui-ci ma fonction de propagation (comparable au parcours profondeur d'un graphe où les sommets seraient les pixels de mon image et les liaisons des relations d'adjacence entre les pixels noirs). Cette dernière est une fonction récursive qui associe à tous les pixels adjacents qui ne sont pas déjà libellés le label du pixel initial avant d'appeler à nouveau la fonction de propagation. À la fin de tous les appels restera alors un aglomérat de pixels portant partout le même label. Précisons que la fonction de propagation renvoie aussi le nombre total de pixels de l'aglomérat qui vient d'être libellé. Je reprend alors le parcours de l'image depuis le pixel initial jusqu'à tomber sur un autre pixel noir sans label auquel j'en associerai un nouveau d'une autre valeur avant de propager à nouveau. À la fin du parcours complet de l'image, tous les éléments sont donc différenciables un à un de par leur label.

Pour rendre l'algorithme plus visuel et plus facile à comprendre j'ai écrit une fonction qui colore chaque aglomérat d'un couleur différente via un bref calcul appliqué au label de chaque pixel. Une fois l'image initiale parcourue et chaque pixel coloré, voici l'image sortante :

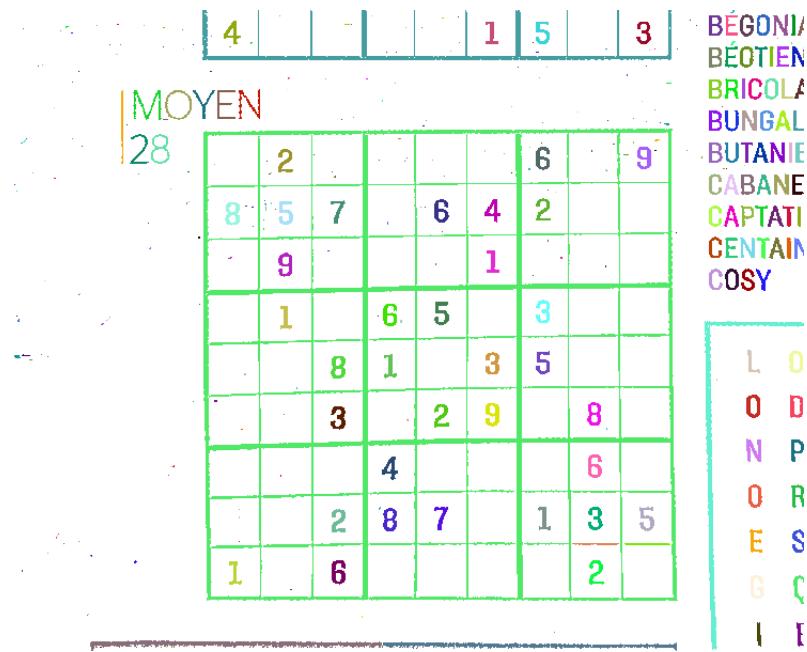


Figure 4: Image colorée après segmentation

On voit alors bien tous les aglomérats colorés différemment, la grille, les chiffres et tous les autres éléments arborants une couleur différente.

Il me suffit maintenant de ne garder parmi tous ces éléments que la grille, ce qui est plutôt tâche aisée, la fonction de propagation me renvoyant à chaque fois qu'elle labelle un élément la

taille de celui-ci en pixels. L'algorithme garde alors constamment en mémoire le label du plus gros agglomérat. La suite de l'algorithme part alors du principe que la grille est le plus gros élément de l'image. Pour ce qui est de garder uniquement la grille, je parcours une nouvelle fois l'image, en remplaçant tous les pixels n'ayant pas le même label que celui du plus gros agglomérat (la grille) par des pixels blancs. Une fois le parcours terminé, l'image sortante n'est plus que constituée de la grille :

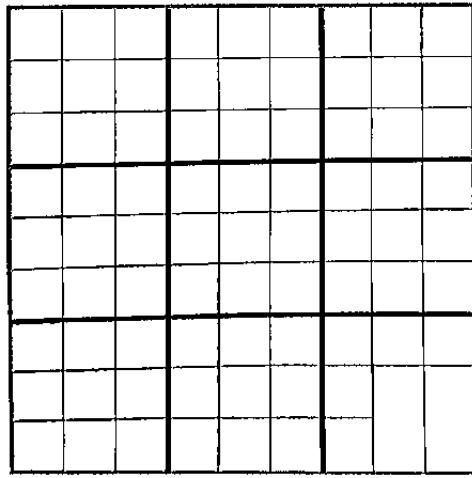


Figure 5: Image une fois tous les éléments autres que la grille retirés

Détection des bords et découpe des cases

La grille maintenant isolée la prochaine étape fut de la découper en ses 81 cases, qui pourront dans le futur être rentrées dans le réseau de neurones de Mathéo afin d'en détecter les nombres. Mais avant de commencer la découpe, il me faut d'abord obtenir les coordonnées exactes de la grille dans l'image. Pour ça j'applique alors sur l'image ne comportant que la grille mon algorithme de détection des bords. Celui projette depuis chacun des bords de l'image, et ce en direction du bord opposé, ce que l'on pourrait assimiler à un rayon. Si ce dit rayon entre en contact avec un pixel noir qui appartiendra donc forcément à la grille, celle-ci étant le seul élément restant dans l'image, il incrémentera de 1 l'élément i correspondant à la coordonnée y du pixel, si le bord en cours de détection est le haut ou le bas, ou sa coordonnées en x , si le bord en détection est le gauche ou le droit, d'un histogramme associé au bord détecté. Ainsi l'algorithme rempli 4 histogrammes associés respectivement aux bords gauche, droite, haut et bas de la grille. Je parcours ensuite ces histogrammes pour trouver le plus grand élément dont la place correspond donc à la coordonnée x ou y moyenne du bord, qui me serviront à déterminer la longueur des bords et ainsi pouvoir appliquer mon algorithme de découpe.

Une fois les coordonnées des bords en poche, il est donc assez facile de déterminer la

longueur de ces derniers. La longueur des bords horizontaux sera équivalente à la coordonnée x du bord droit moins celle du bord gauche et la longueur des bords verticaux équivalente à la coordonnée y du bord haut à laquelle on soustrait celle du bord bas. Il me suffit maintenant simplement de parcourir la grille case par case, chaque case ayant pour bords horizontaux et verticaux ceux de la grille divisés par 9, en commençant du coin haut gauche de la grille, ayant pour coordonnée x celle du bord gauche de la grille et pour coordonnée y celle du bord haut. Avant de parcourir une case je crée alors une nouvelle image ayant pour dimensions celles d'une case lambda de la grille. Je recopie alors dans cette image les pixels de la case que je parcours actuellement. Une fois la case parcourue et l'image remplie j'enregistre celle ci sous le nom de $tile_{i,j}$, avec i et j désignant la position de la case dans la grille. Une fois toutes les cases parcourues je me retrouve alors avec 81 images correspondant chacune à une case de la grille.

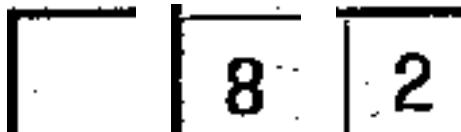


Figure 6: Exemples de cases obtenues après découpage

Toutes les cases maintenant sauvegardées, elles pourront être rentrées dans le réseau de neurones de Mathéo afin de détecter le nombre contenu dans chacune tout en gardant leur position dans la grille grâce au nom du fichier.

Prochaine Soutenance

Pour la prochaine soutenance, j'ai une nouvelle fois été amené à travailler sur la détection de la grille. En effet mon algorithme de propagation étant récursif, des images en trop haute qualité, dont le nombre de pixels est donc très élevé, pourraient faire planter celui-ci, qui n'avait pour l'instant pas de, la fonction de black and white d'Antoine réduisant à l'époque le format des images. J'ai donc été amené à optimiser mon algorithme pour que le bon fonctionnement de celui-ci ne soit pas dépendant du format de l'image fournie. Aussi, comme vous pouvez le voir sur les images des cases ci-dessus, certains morceaux de la grille restent visibles, j'ai donc fait en sorte de retirer la grille avant de découper l'image, afin de faciliter le travail au réseau de neurones. Il me fallait également passer ces cases au format 16x16, conventionnel au réseau. Aussi, je davais commencer à travailler sur un algorithme permettant d'imprimer des numéros dans une image, afin de pouvoir à l'avenir remplir la grille du sudoku résolu. Pour finir, ou devrais-je dire avant de commencer à travailler sur le reste, je me devais d'organiser mes fichiers dans le projet, ceux-ci étant à l'époque approximativement aussi rangés que ma chambre, c'est à dire ne sont pas rangés du tout.

Seconde soutenance :

Amélioration et passage du watershed en itératif

Le principal défaut de mon watershed était le fait que celui-ci, à l'inverse d'un algorithme classique de ce type, était implémenté en récursif et pouvait avoir du mal à tenir le coup sur les images de qualité élevée (comme l'image n°5 dans notre cas). Ma première tâche aura donc été d'implémenter une structure queue dans mon projet, chose que j'ai faite en m'a aidant de structures que j'ai pu trouver sur le web, de telle sorte à parvenir à une implémentation qui me convenait en fusionnant, modelant et modifiant certains exemples. En réalité, cette structure est plutôt une sorte de tableau camouflé en queue, un tableau sur lequel nous ne pouvons uniquement effectuer les opérations propres aux queues si vous préférez. Maintenant la queue implémentée, il ne s'agissait plus que de mettre à jour mon watershed pour qu'il utilise cette dernière. Heureusement je n'ai pas eu à tout refaire, ayant pris soin précédemment de faire marcher ma fonction de propagation de façon indépendante par rapport aux autres, j'ai donc simplement eu à remplacer ma fonction actuelle par un parcours largeur de mon image et non profondeur.

Normalisation et nettoyage des cases

Comme dit précédemment, pour assurer le bon fonctionnement du réseau de neurones, je me devais de passer les cases en 16 par 16, afin d'avoir un format commun à toutes. J'ai donc, sur la base d'une fonction précédemment écrite par Antoine fait en sorte de resize chacune des cases sortant de mon algorithme de découpe. Cependant, après avoir effectué quelques tests, nous nous sommes vite rendus compte que le format 16 par 16 n'était pas réellement approprié, nous faisant perdre trop de détails sur les chiffres, rendant par exemplaires difficile au réseau de neurones de différencier les 9 des 6 ou des 8; raison pour laquelle nous avons opté pour un format 28 par 28, qui semblait bien plus efficace. Mais un problème persistait, une fois les cases au bon format, il restait toujours dedans des impuretés et les chiffres n'étaient pas nécessairement centrés.

Je ne me suis donc pas contenté de juste changer le format de l'image, moi et Antoine avons passé un temps considérable à améliorer notre traitement afin de ressortir des chiffres les plus lisibles que possible. Antoine lui aura travaillé de telle façon à enlever le plus de grain possible de ses images en noir et blanc et afin de lisser les traits à l'aide d'une fonction de dilatation et moi j'aurais fait en sorte de normaliser les cases sortantes encore une fois dans le but de faciliter l'entraînement et la détection par le réseau de neurones.

J'aurais alors commencé par nettoyer les cases de leurs impuretés, pour cela je réutilise mon watershed sur chacune des cases afin de ne garder dans celle-ci uniquement le plus gros élément qui sera donc logiquement soit une simple tâche, soit un chiffre. Je repasse ensuite à nouveau sur toutes les cases afin de compter le nombre de pixels noirs qu'elles contiennent et si ce nombre est inférieur à un certain seuil : $(0.16 * \text{hauteur} * 0.16 * \text{largeur})$, je considère que la case ne contient pas de chiffre et la remplie alors de pixels, tous de la même couleur, avec leur composante rouge égale à 42, ce qui permet à Matheo de savoir en regardant un seul pixel de l'image (celui en coordonnées 0x et 0y) si celle-ci contient un chiffre ou non. Dans le cas où le

nombre de pixels noirs serait supérieur à ce seuil, je cherche la position de la zone contenant le chiffre, en fonction de sa hauteur et sa largeur dans la case initiale, je l'extract en créant une nouvelle image avec la même taille que cette dite zone, copie le chiffre dedans et la passe au format 10 par 16, que je copie ensuite au centre d'une la case blanche en 28 par 28. Toutes les cases sont alors normalisées, soit complètement vides et avec la composante rouge de chaque pixel égale à 42 (la blague était bien trop aguicheuse pour ne pas la faire), ou contenant un chiffre au format 10 par 16 au centre.

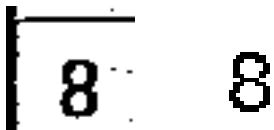


Figure 7: Exemples de cases avant et après normalisation

Impression du résultat dans une grille

Une fois les cases passées à travers le réseau de Matheo et le sudoku résolu par le solver d'Antoine, il nous fallait afficher le résultat à l'écran. Je me suis donc occupé d'écrire une fonction qui récupère le fichier contenant le résultat du sudoku et l'imprime dans une grille. Pour ça je possède un dossier contenant dedans une grille vide et une image de chacun des chiffres à imprimer dans la grille résolue (soit une image pour chaque chiffre allant de 1 à 9). Je parcours alors le fichier de la grille résolue en parallèle du fichier contenant l'état de la grille avant résolution, lorsque je tombe sur un chiffre dans le fichier pré-résolution, je le print en noir à la place correspondante dans la grille vide (en copiant dans la grille l'image du nombre), tandis que lorsque je tombe sur un "." dans le fichier pré-résolution, signifiant que le chiffre n'était pas présent dans la grille originale, je print en rouge dans la grille le nombre présent à la même place que le "." dans la grille post-résolution. L'image résultante est donc une grille de sudoku résolue mettant en évidence les chiffres insérés par notre algorithme.

127 634 589	127 634 589
589 721 643	589 721 643
463 985 127	463 985 127
218 567 394	218 567 394
974 813 265	974 813 265
635 249 871	635 249 871
356 492 718	356 492 718
792 158 436	792 158 436
841 376 952	841 376 952

Figure 8: Fichiers avant et après résolution

1	2	7	6	3	4	5	8	9
5	8	9	7	2	1	6	4	3
4	6	3	9	8	5	1	2	7
2	1	8	5	6	7	3	9	4
9	7	4	8	1	3	2	6	5
6	3	5	2	4	9	8	7	1
3	5	6	4	9	2	7	1	8
7	9	2	1	5	8	4	3	6
8	4	1	3	7	6	9	5	2

Figure 9: Image sortant de mon algorithme

Alimentation de la bibliothèque

Pour ce qui est de la bibliothèque d'images, j'aurais participé au même titre que mes camarades à alimenter celle-ci. En utilisant les images fournies, d'autres trouvées sur internet ou encore que j'ai prises moi-même en photo et en les passant toutes dans mon algo, j'ai fait un dossier pour chaque image en y ajoutant un chiffre de chaque avec les cases que j'obtenais. J'aurais ainsi ajouté plus d'une quinzaine de dossiers. La bibliothèque a beaucoup changé au fil de notre projet afin de contenir en permanence des chiffres le plus proche possible de ce qui à terme sera nourris au réseau de neurones de Mathéo.

Résolution de bugs et plomberie

Le plus gros problème rencontré pour moi lors de cette seconde soutenance aura été les leaks de mémoire. En effet, bien que normalement faciles à régler : on trouve la source de celui-ci et on le fix. Le problème ici c'est que, travaillant sous MacOS, les leaks sont gérés par celui-ci, ils n'empêchent en aucun cas le bon fonctionnement de mon programme ou son exécution. En quoi est-ce un problème vous me direz ? Eh bien imaginez travailler plusieurs heures voire plusieurs jours sur une fonction, heureux vous poussez votre travail et vous recevez un message à 3h du matin disant "*Hugo rien ne marche viens*". C'est un peu embêtant je vous l'accorde. C'est pourquoi avec l'aide d'Antoine qui testait en parallèle chez lui, nous avons réussi à mettre fin à toutes ces fuites. Plus de peur que de mal mais surtout pas mal de temps perdu.

Améliorations possibles

Je suis plutôt satisfait de ma partie mais il reste encore certaines choses que je peux améliorer pour faciliter son fonctionnement sur certaines images. Implémenter une fonction d'homographie pourrait permettre à mon algorithme de marcher sur des images dont l'angle et la prise de vue, qui pour l'instant rendent mon algorithme inutilisable à partir d'un certain seuil, bien plus consistant, me permettant de les redresser.

Conclusion personnelle

J'aurais beaucoup aimé travailler sur ce projet, il aura été très intéressant et m'aura permis à la fois de découvrir un nouveau langage, le C, et de travailler sur du traitement d'image, quelque chose de complètement nouveau pour moi. Les quelques bugs que j'aurais rencontrés sans connaître leur origine ou problèmes liés à l'OS m'auront parfois un peu découragé mais moi et tous mes camarades en aurons finalement vu le bout.

Angelina Kuntz

Présentation

Je me présente, Angelina Kuntz, simple membre non impliqué dans la lutte au pouvoir pour le titre de chef de groupe (à ne pas prendre au sérieux). Pour ce début de projet, je me suis surtout occupée du support de l'interface graphique, de la bibliothèque d'image ayant pour objectif de nourrir l'IA de Mathéo et enfin un petit coup de pouce à Antoine dans son traitement d'image.

Première soutenance

Début du projet

Tout ce travail a débuté avec beaucoup de documentation sur plusieurs plateformes et vidéos explicatives des nouveaux supports que je devais utiliser à commencé par SDL.

La bibliothèque d'image des nombres ne demandait aucune recherche, juste du temps et de la patience pour la création des images une par une sur le logiciel d'édition d'image GIMP.

Le plus grand défi était d'apprendre à utiliser GTK avec glade qui est une bibliothèque permettant de créer des interfaces graphiques. La difficulté est qu'il y a peu de documentation et s'il y en a, elle est souvent ancienne et parfois obsolète. Curieusement, le plus souvent datait de 2013.

Rotation

L'aventure de ce projet commence avec la rotation d'image. Comme tous le reste du groupe, un début garni de documentation sur la bibliothèque SDL. Après recherche, j'en suis arrivée à la conclusion qu'il y avait deux méthodes pour faire la rotation, la première en utilisant une matrice de rotation pour chacun des pixels de l'image, l'autre était d'utiliser directement la méthode "rotzoomSurface()". Après réflexion, je me dis qu'utiliser une méthode mathématique semblait plus facile plutôt que d'apprendre à utiliser toutes les petites spécifications de SDL. La matrice de rotation se présente ainsi :

$$R(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \text{ (rotation d'angle } \theta \text{)}$$

Figure 10: Matrice de rotation

Il est à noter que lorsque que l'on applique la matrice à chacun des pixels l'un après l'autre, il y a un défaut dans la rotation et il y a apparition de plein de petits pixels blancs sur toute la surface. Je suis partie dans plus de recherche pour régler ce souci, la solution trouvée fut au final de simplement appliquer non pas la matrice de rotation, mais l'inverse de la matrice de rotation, c'est-à-dire sa transposée en faisant la recherche non pas sur les pixels de l'image d'origine à retourner, mais sur les pixels de l'image de destination pour chercher quel pixel doit se positionner à cet endroit.

Problème réglé, à partir de là, il y avait un nouveau défi, celui de garder toute l'image dans le cadre. En effet, lorsque une image est tournée, celle-ci prend plus de place et si la surface de l'image n'est pas modifiée, les extrémités sortent et sont perdues. Il s'en suivit de nombreuses recherches de repositionnement et de changement de la taille de l'image selon plusieurs paramètres.

Au final, aidé par Antoine qui était, si nous pouvons l'appeler ainsi, le leader du traitement de l'image du groupe, nous sommes repartis du début, retour à la méthode rotzoom. Dorénavant, l'image est chargée avec "SDL_Init(SDL_INIT_VIDEO)" et permet ainsi "rotzoom" et la resize de la taille de l'écran avec "SDL_SetVideoMode" où nous pouvons passer en argument la width et la height souhaité. D'une pierre, deux coups, le problème de la rotation et de celle que celle-ci reste entièrement dans la surface sont réglés.

Pour exécuter le programme, il faut se placer dans le dossier "Rotation", faire un make et écrire "./rotate x y" avec x = path_de_l'image et y = angle_de_rotation. Voici un exemple avec le sujetlionel3.jpg et un angle de 30° avec la commande "./rotate ../../Images/sujetlionel3.jpg 30".

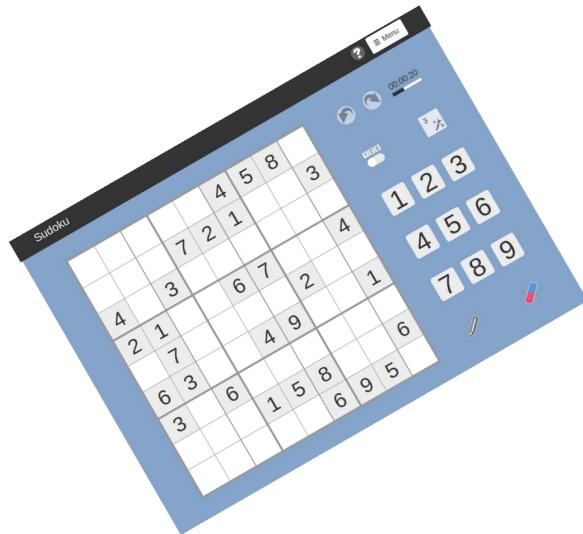


Figure 11: Rotation du sujetlionel3.jpg avec 30°

Un des gros points positifs de notre groupe, je trouve, est que nous sommes à l'écoute les uns des autres et sommes toujours prêts à donner un conseil ou un coup de pouce à l'autre, même de discuter de ce qui ne va pas et d'accepter les critiques des uns et des autres pour améliorer notre travail.

Interface graphique

Maintenant, passons à l'interface graphique. Celle-ci devra, à la fin du projet, pouvoir gérer et résoudre le sudoku en appelant tous le reste des autres fonctions. Pour commencer, je me suis documenté tant bien que mal à partir de vidéo datant de 2013 expliquant la manipulation de glade et de GTK.

J'y ai reconnu un Unity3D et le projet de l'année dernière lointain dans sa manipulation. En effet, il y a glade, qui est surtout visuel qui permet de créer des objets, une grid sur laquelle on peut accrocher d'autres éléments, des contrôleurs ce qui permet l'intercation avec l'utilisateur tel que des boutons et des éléments d'affichage tel que des labels ou des images. Évidemment, il y a beaucoup plus d'outils, mais j'ai préféré me concentrer sur ce qui était le plus utile en premier lieu, et si le temps me le permet plus tard, d'optimiser le tout avec d'autres fonctionnalités disponible. Pour le parallèle avec Unity3D, dans les deux cas, les objets peuvent avoir des attributs définis dans le code et qui sont utilisable dans le code et dans l'interface graphique lui-même (Exemple : `OnColliderEntry()` et `on_button_clicked()`). L'interface se présente ainsi :



Figure 12: Interface

Nous pouvons donc voir un certain nombre de boutons sur la partie supérieure de l'interface, et sur là partir inférieur juste de l'information pour l'utilisateur. Pour la partie inférieure, nous y retrouvons le logo de notre groupe, une fleur rouge en référence à Hanabi, le nom de notre groupe, qui veut dire feu d'artifice et se traduit en "fleur de feu" littéralement. Ensuite le nom du groupe et les noms de tous les membres. Pour rendre le visuel plus agréable, il y a quelques petits ajouts en référence à la période où nous sommes.

À présent les objets interactifs. Pour cette première soutenance, les boutons ne sont pas encore reliés aux différentes fonctions du projet, ils n'exécutent pas les programmes du traitement d'image ni de résolution de sudoku, ils ne s'occupent que de l'affichage des résultats de ses fonctions. En effet, pour les faire fonctionner, il faut au préalable avoir compilé et exécuté les main des autres fonctions qui se chargent de save les résultats générés à des endroits spécifiques sous des noms spécifiques. Cependant, certaine fonctionnalité ont déjà été entamé même si elles ne sont pas encore opérationnelle.

Par exemple, nous pouvons déjà chercher un fichier à partir du file chooser et une fois sélection-

ner, celui-ci s'affichera dans la barre du file chooser. Le bouton display, cependant, affiche l'image sauvegardée à partir de la fonction exécuté au préalable. Pour la prochaine soutenance, je prévois de relié le display au file chooser pour que celui-ci affiche bien l'image sélectionnée par l'utilisateur.



Figure 13: Sélection du sujetlionel3.jpg dans le file chooser

Pour ce qui est de la rotation, j'ai voulu quelque chose de plus original que juste un champ où l'utilisateur tape un nombre et y ait inclus un bouton spin où l'utilisateur peut évidemment écrire, mais également ajuster avec les boutons "+" et "-". Si on appuie sur le bouton de rotation à côté, il n'y a pas encore la rotation qui s'effectue, car les boutons ne sont pas encore reliés aux fonctions, mais l'angle saisi par l'utilisateur est récupéré et est affiché au niveau du label info en dessous du sudoku.



Figure 14: Angle et affichage info

Maintenant, passons aux boutons d'affichage à droite. Chaque bouton s'occupe de changer le widget container image du logo pour afficher l'image correspondante. En effet, chaque image a été sauvegardée sous la forme de "grayscale.bmp", "grid.bmp", etc. Et la fonction "on_button_clicked()" s'occupe de changer l'affichage avec gtk. Reprenons l'exemple du sujetlionel3.jpg et l'aperçu des boutons "Grayscale" et "Black and White".

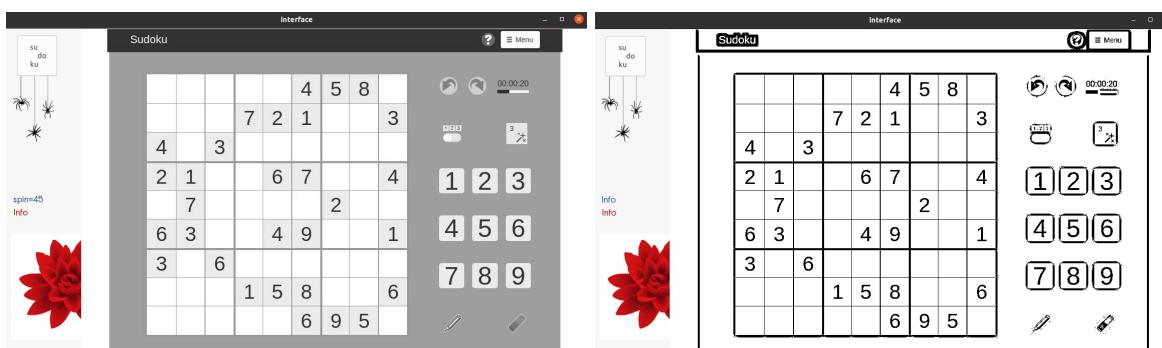


Figure 15: Bouton Grayscale et Black and White

J'ai longtemps hésité à garder la taille des sudokus réduits pour que l'utilisateur puisse toujours utiliser les autres boutons à droite, mais après réflexion une grille de sudoku très réduite est illisible et ait opté pour une autre solution. La taille des sudokus reste telle qu'elles sont et en plus il y a le bouton "su do ku" en haut à gauche qui réaffiche l'interface de base avec le logo.

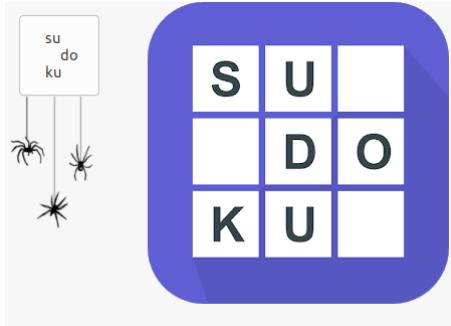


Figure 16: Bouton de retour "su do ku"

Bibliothèque

Enfin, la bibliothèque qui servira à nourrir l'IA. Au début, ce n'était que des tests avec des tailles d'image différentes. Après discussion avec Mathéo, nous sommes arrivés à la conclusion qu'il fallait que les images soient des 16x16 pixels fond blanc, chiffre en noir et en format bmp. À partir du logiciel gimp, j'ai donc créé une image par chiffre de 1 à 9 (il n'y a pas de zéro dans les sudokus) en police d'écriture différente et les ai rangés dans un dossier intitulé par le nom de la police d'écriture. Pour cette première soutenance, il n'y a que 9 dossiers.

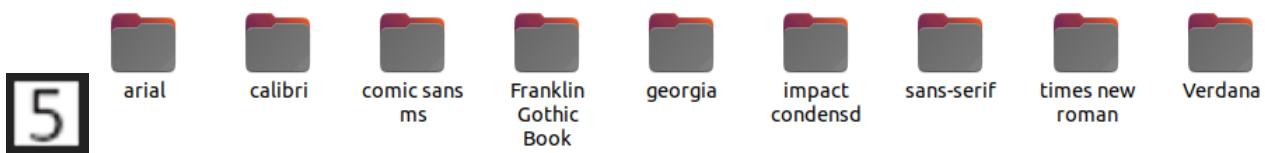


Figure 17: Exemple de l'image calibri/5.bmp et Bibliothèque d'image

Prochaine Soutenance

Pour la seconde soutenance, je m'occuperai d'enrichir la bibliothèque avec plus de polices d'écriture. Je me chargerai également de rendre l'interface graphique parfaite. Je prévois de faire en sorte que lorsque l'utilisateur sélectionne une image, c'est celle-ci qui sera affiché avec le bouton display. Lorsque l'utilisateur aura appuyé sur le bouton avec la flèche tournante, l'angle sera calculé par une fonction et il y aura toutes les autres fonctions qui seront lancées: la rotation, le grayscale, le black and white, la résolution de grille, etc. Toutes les fonctions de lancé auront save l'image du résultat et les boutons à droite auront la même mission, celle d'afficher les résultats. Une fois fait, j'optimiserai le reste de l'interface pour qu'elle soit le plus agréable à utiliser.

Seconde soutenance

Bibliothèque

Pour ce qui est de la bibliothèque, j'avançais petit à petit sur cette première version de la bibliothèque d'image de 16*16 jusqu'au moment où, retournement de situation, il fallait la refaire, mais en 28*28 cette fois-ci ! Ni une ni deux, je recommence et je la refais en 28*28 avec un total de 30 polices d'écriture différentes toujours pas le même procédé pour créer les images à partir de GIMP.

Finalement, cette bibliothèque ne sera pas utilisée et une troisième bibliothèque 3.0 est créée, mais cette fois-ci à partir de nombre issu de plein de sudokus différents.



Figure 18: Bibliothèque 2.0 en taille 28*28

Interface

À présent le plus gros morceau, l'interface. L'interface étant le noyau du projet, car c'est celui qui relie chaque partie entre elles et permet de les fusionner. Il fallait un makefile fonctionnel pour que tout fonctionne en un seul make. Pour se faire nous avons avec Antoine, consacré pas mal de notre temps à tester plein de variantes et d'écriture différente de Makefile pour qu'il fasse compiler et l'interface et le traitement d'image. Après avoir trouvé une solution avec mon acolyte, j'ai pu reprendre la même structure et ajouter les parties de Mathéo et Hugo.

Maintenant, que tout est relié, passons à l'interface lui-même. Voici tout d'abord un aperçu de celui-ci.

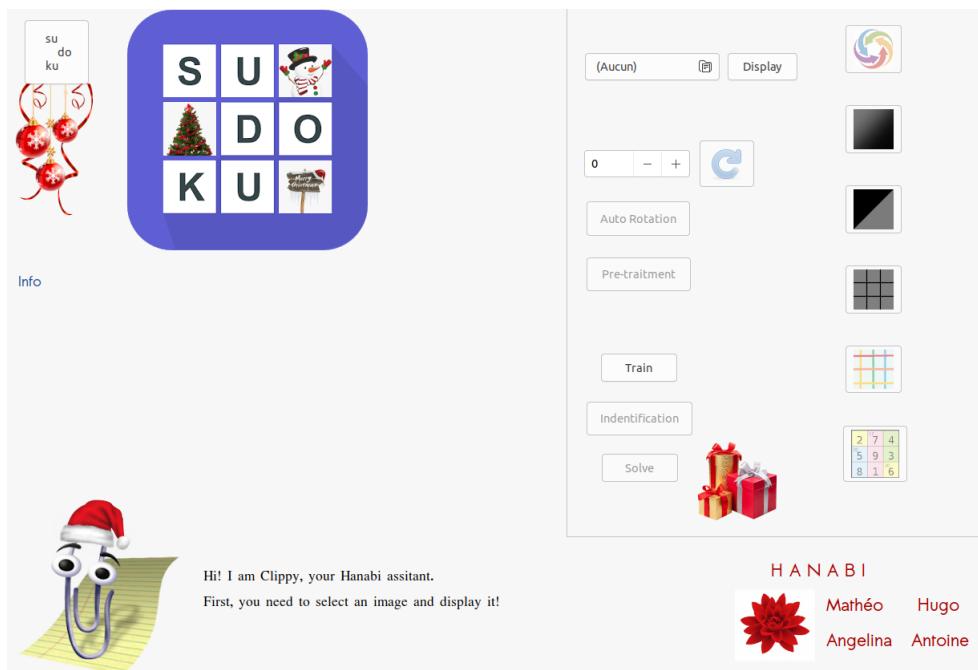


Figure 19: Interface final

Pour se retrouver parmi tous les boutons présents sur la partie droite, voici une petite légende qui explique leur utilité. Comme vous pouvez le voir sur l'image précédente, tous les boutons ne seront jamais disponibles en même temps, en effet, il ne sera pas possible de résoudre le sudoku sans avoir identifié les nombres, il ne sera pas possible d'identifier sans avoir fait de traitement, il ne sera pas possible de faire de traitement sans avoir sélectionné d'image au préalable et ainsi de suite. Pour la légende, tous les boutons sont activés pour une meilleure visibilité.

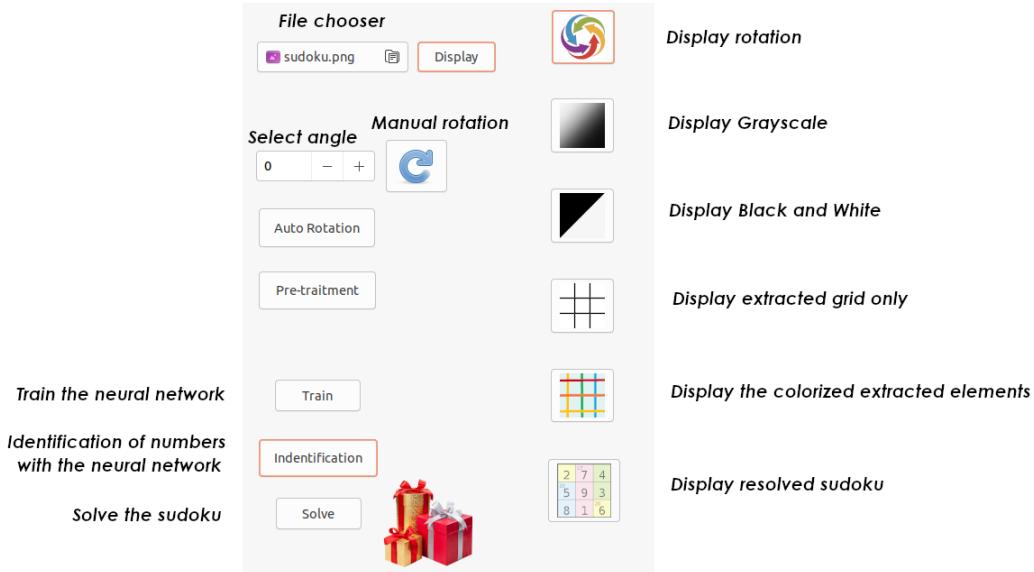


Figure 20: Légende de tous les boutons

Comme souhaité pour la première soutenance, tout le projet fonctionne grâce à l'interface qui fait appel aux différentes fonctions créées par tout le groupe et les fusionne. Comme annoncé au début, il y a un certain ordre dans lequel l'utilisateur peut cliquer sur les boutons, au lancement de l'interface, il n'y a que le "File chooser", son "Display" et le bouton "Train" qui sont disponibles. Lorsque l'utilisateur display une image qu'il a choisi, le label "info" est actualisé et affiche le path de l'image sélectionné et donc celle qu'il va traiter. Il y a également le bouton de "Manual rotation" de "Auto Rotation" et de "Pre-traitement" qui se déverrouille. Si l'utilisateur décide de faire une rotation, n'importe laquelle, l'image rotate va s'afficher directement et "Display Rotation" se déverrouille. Si l'utilisateur voudrait à présent traiter l'image rotate il doit retourner dans le file chooser et choisir la nouvelle image créée pour actualiser le path de l'image sur laquelle nous travaillons.

À présent le "Pre-Traitement", bouton qui lancera toutes les fonctions de traitements d'image, d'extraction, déverrouille tous les boutons "Display" sur la droite excepté la "Display rotate" et le "Display resolved sudoku". "Identification" se déverrouille également à ce moment-là. Lorsque l'utilisateur lance une identification, s'il n'y a pas de réseau de neurones déjà entraîné, un message indiquant qu'il faut cliquer sur "Train" pour en entraîner un s'affiche. Dans le cas contraire, le réseau de neurones est lancé et celui-ci écrira les nombres reconnus dans un fichier .txt. Ensuite, le bouton "Solve Sudoku" se déverrouille et celui-ci résoudra le .txt avec un algorithme récursif et ensuite remplira l'image d'origine avec les nombres correspondants dans les cases correspondantes. À la fin du traitement, "Display resolved sudoku" sera accessible et celui-ci affichera l'image finale dans l'interface. À savoir que le bouton "File chooser" ne se désactive jamais, si l'utilisateur change d'avis, il peut choisir une autre image et lorsqu'il l'aurait display, tout le cycle d'accessibilité des

boutons se réinitialisera comme au lancement de l'interface.

Toutes ces informations sont très indigestes et un utilisateur peu à l'aise pourrait très vite s'y perdre si l'un des créateurs n'est pas à côté pour lui expliquer ! Pour ce faire, nous accueillons un cinquième membre dans notre équipe, Clippy notre assistant ! Il guidera l'utilisateur en lui donnant des informations tout le long du processus. Les messages de Clippy s'actualisent en même temps que des boutons se déverrouillent, voici tous les conseils que notre assistant peut donner. À noter que la première phrase qu'il dit au lancement est visible sur la "Figure 18: Interface final".

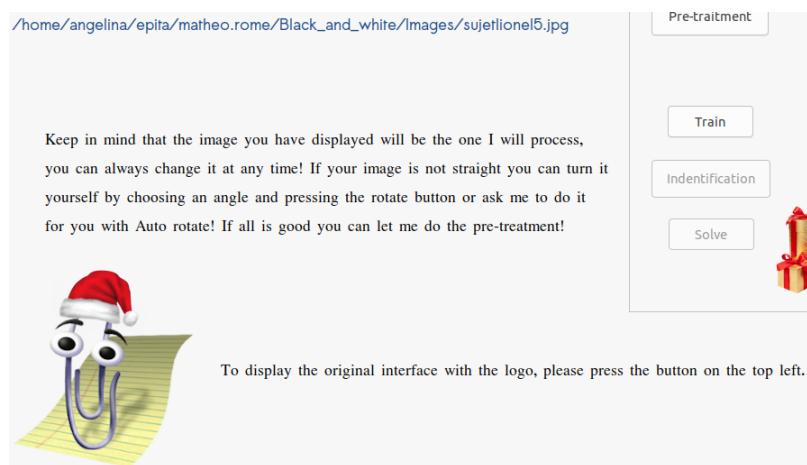


Figure 21: Clippy après la sélection d'une image.

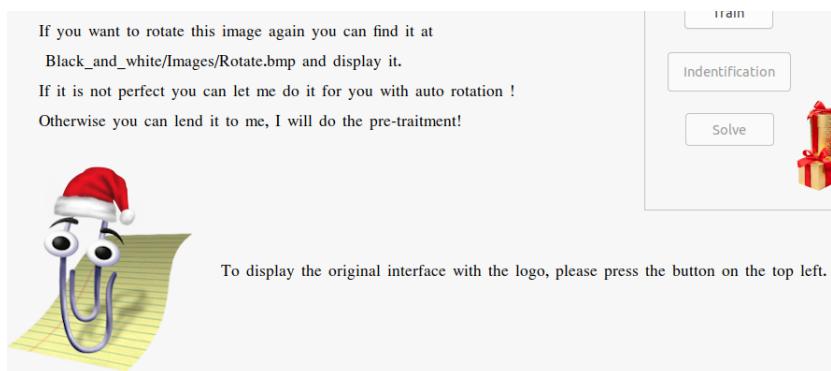


Figure 22: Clippy après une rotation.

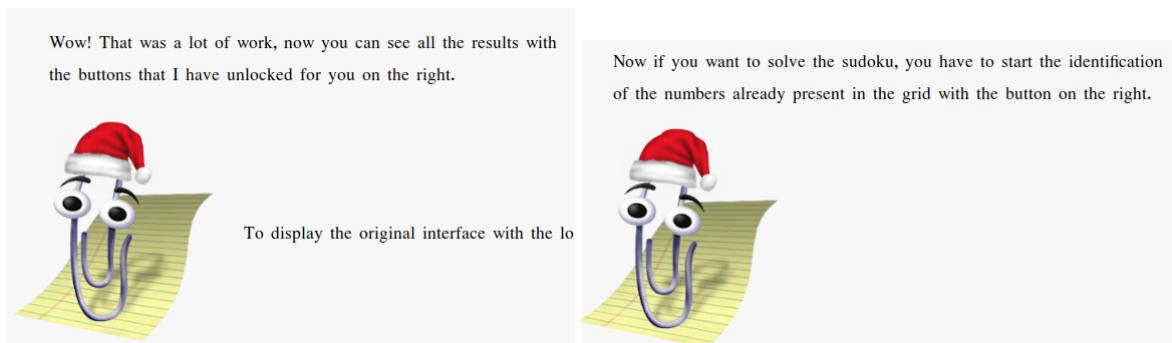


Figure 23: Clippy après un pré-traitement.

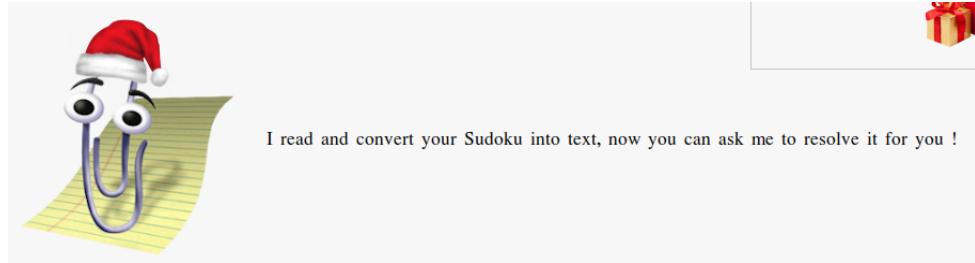


Figure 24: Clippy après une identification.

The screenshot shows a Microsoft Office ribbon interface with the 'Solve' tab selected. On the left, there's a sidebar with icons for 'su do ku' (Sudoku), a Christmas tree, and a folder labeled '/home/angelina/'. The main area displays a 9x9 Sudoku grid with numbers in various colors (red, green, blue, orange, purple, yellow). To the right of the grid is a vertical toolbar with buttons for 'sujetlionel.jpg', 'Auto Rotation', 're-treatment', 'Train', 'Identification', and 'Solve'. Below the grid, a message from Clippy reads: 'Wow we made it! After all this way together, it was a great adventure. If you want to start over you can choose a new image with the file chooser.'

Figure 25: Clippy après "Display resolved sudoku".

Rotation automatique

De retour pour une partie traitement d'image, cette fois-ci, nous nous attaquons à la rotation automatique. Pour la faire, j'ai tout de suite pensé à réutiliser de la trigonométrie comme pour la première rotation (bien que la méthode avec l'inverse de la transposée de matrice n'est pas aboutie). Lorsque le processus d'auto-rotation est lancé, il y a en réalité plusieurs algorithmes qui se lancent les uns après les autres. En premier, un pré-traitement pour "nettoyer" l'image et grossir les pixels pour épaisser les traits de la grille de sudoku. Ensuite, à partir de la nouvelle image en noir et blanc sauvegardé, on lance la fonction "extract_grid" qui en temps normal sert à extraire tous les nombres du sudoku, mais dans ce cas précis il n'y a que la grille qui est extraite, car de toute manière, l'image n'est pas droite et il y a impossibilité d'en extraire les nombres.

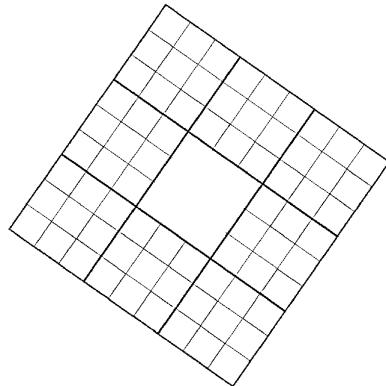


Figure 26: extract_grid_only.bmp

Enfin ! Maintenant nous voilà avec une image avec juste la grille du sudoku en noir sur fond blanc. Maintenant, nous pouvons passer au calcul de l'angle avec lequel il faudrait faire tourner l'image. Pour ce faire, à partir de la ligne supérieur horizontale de la grille du sudoku, on la parcourt une première fois jusqu'au coin gauche et on garde ses coordonnées x_1 et y_1 . Ensuite, on continue le long de la même ligne, mais dans le sens opposé pour chercher le coin droit de la ligne. Lorsque c'est le dernier pixel noir, on garde ses coordonnées x_2 et y_2 . Ainsi, nous avons les coordonnées des deux extrémités de la ligne supérieure horizontale du sudoku. Pour trouver un angle à partir de deux points, rien de plus simple qu'un peu de trigonométrie. Pour calculer l'angle, on applique la formule : $\tan(\alpha) = (y_2 - y_1) / (x_2 - x_1)$.

Parfait ! Nous avons notre angle, cependant faire la rotation sur l'image actuelle ne servirait pas à grand chose, ce n'est qu'une simple grille. À partir de l'angle calculé, nous pouvons appeler la fonction "Rotate" manuelle qui prend en argument l'image et l'angle de l'utilisateur avec la différence que c'est la première image qu'on voulait rotate et l'angle calculé précédemment. Et voilà le travail ! La rotation automatique peut avoir quelques degrés d'erreur du à la qualité de l'image, si jamais la grille est trop fine le parcours de celle-ci ne se fera pas jusqu'aux extrémités et les deux points pour calculer l'angle seraient trop rapprochés ce qui serait moins précis, et cela, se refléterait sur le résultat.

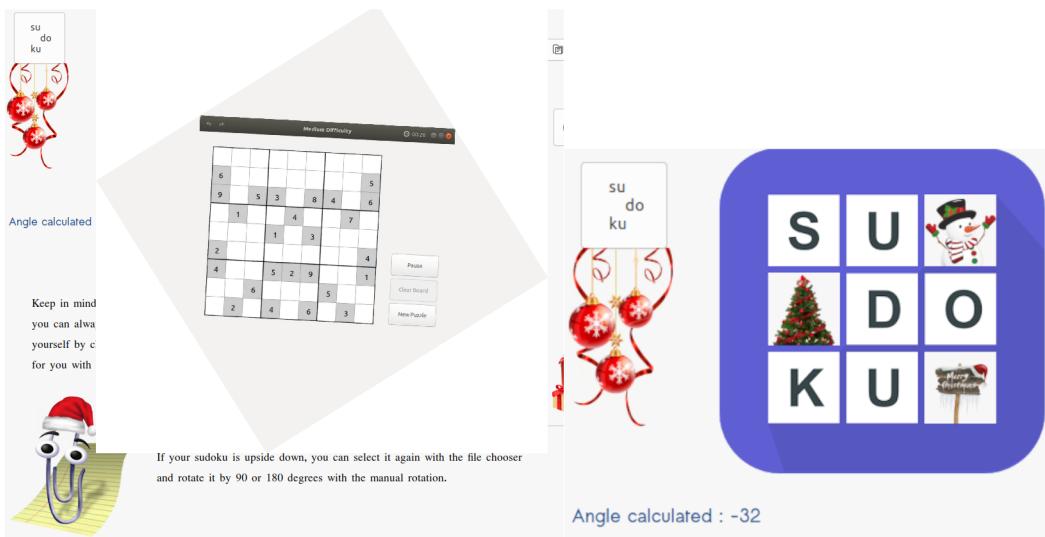


Figure 27: Auto Rotation et information sur l'angle dans l'interface

Pour revenir rapidement à la rotation manuelle, il est désormais possible de le faire directement à partir de l'interface à partir du bouton spin en choisissant un angle. Dans le cas où nous aurions une grille de sudoku qui aurait la tête à l'envers, il faudrait faire une rotation manuelle de 180 degrés avant ou après la rotation automatique pour qu'elle soit dans le bon sens et bien droite.

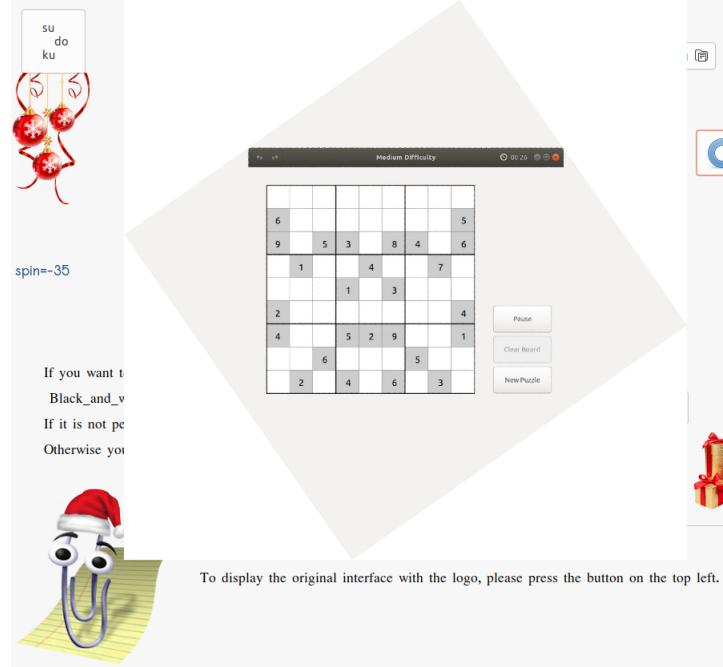


Figure 28: Rotation manuelle et information sur l'angle donné

Conclusion personnelle

En conclusion, ce projet m'aura permis d'apprendre à utiliser les bibliothèques de SDL, de celles qui vont avec, de GTK ainsi que l'utilisation de l'application de glade. J'ai également pu en apprendre plus sur les makefiles même si je ne pense pas encore avoir saisi totalement leur fonctionnement. Il y a également l'importance de la répartition du travail et de son organisation, qu'il est primordial de toujours communiquer avec ses coéquipiers sur nos doutes, nos difficultés. Surtout, ne jamais oublier de féliciter et d'encourager ses mates, car il ne faut pas perdre de vue notre objectif et le fait que nous sommes tous dans la même situation. Il faut affronter les problèmes en étant un groupe uni tous ensemble et ne pas rejeter la faute sur les autres et les laisser se débrouiller tout en les accusant. Je peux dire avec certitude que nous avons une très bonne cohésion et une bonne communication tous les quatre.

Antoine

Présentation

Je me présente, je m'appelle Antoine RIQUET et dans ce projet, je me suis principalement occupé de la partie programmation du solver, mais également de la partie traitement d'image. Cette partie étant reliée à la découpe de l'image et à une partie de la rotation, j'ai beaucoup travaillé avec Hugo et Angelina.

Première soutenance

Début de projet

Tout d'abord durant les débuts de projet, j'avais pour objectif de réaliser le solver du Sudoku en ligne de commande. Pour cela, j'ai dû regarder de nombreux tutos en C sur internet pour me familiariser avec le langage, car les notions que nous avions abordées durant les deux premières semaines de cours ne suffisaient pas à la compréhension des arrays, pointeurs, listes etc.

Ainsi, j'ai dû dans un premier temps comprendre la compilation du C en ligne de commande, chose qui était nouvelle pour moi, et mes premiers pas avec gcc furent assez laborieux étant donné que je ne comprenais pas exactement comment celui-ci fonctionnait.

Solver de Sudoku

Après m'être peu à peu familiarisé avec le langage et le système de compilation, je me suis lancé dans la recopie d'une partie d'un TP de l'année dernière en C# qui portait également sur la réalisation d'un solver de Sudoku. Cependant, j'ai dû l'améliorer et faire quelques modifications étant donné que le langage n'était pas le même et que le rendu final attendu pour celui-ci n'était pas du tout la même chose que ce que nous avions fait l'année précédente. Concernant le rendu final, voici une capture d'écran ci-dessous de mon terminal qui résume l'implémentation de mon solver en C. J'ai également ajouté un fichier « Makefile » permettant de compiler mon solver plus rapidement et facilement.

Sur cette capture d'écran, nous pouvons observer que dans mon dossier Solver se trouvent 3 fichiers. Le premier est une grille de sudoku en .txt, dans ce format, les cases vides sont représentées par des points. Lorsque nous effectuons la commande cat, la grille de sudoku en .txt s'affiche bien dans notre terminal. Nous pouvons également remarquer deux autres éléments dans ce dossier : le solver, qui est la compilation avec mon makefile du fichier C, sudoku.c. En dernier, nous retrouvons le "Makefile" utilisé pour la compilation du solver. Ensuite, lorsque nous lançons l'exécutable sur la grille de sudoku, puis que nous relançons la commande "ls", nous observons qu'un nouveau fichier est apparu. Le .txt grid01result.txt qui est en fait le fichier qui contient la résolution du sudoku de la précédente grille, et si nous affichons avec "cat" le fichier, nous pouvons voir la solution du sudoku. Bien évidemment, si nous lançons le solver avec plusieurs grilles ou avec des arguments faussés, celui-ci affichera un message d'erreur.

```

B&W/Solver> ls
grid_01.txt  Makefile  Sudoku.c
B&W/Solver> make
gcc -fsanitize=address -Wall -Wextra -std=c99 -O0 -g -MMD Sudoku.c -o Solver
B&W/Solver> ls
grid_01.txt  Makefile  Solver  Solver.d  Sudoku.c
B&W/Solver> ./Solver grid_01.txt grid_01.txt
Erreur : Nombre d'arguments invalide
B&W/Solver> ./Solver grid_01.txt
B&W/Solver> ls
grid_01.result.txt  grid_01.txt  Makefile  Solver  Solver.d  Sudoku.c
B&W/Solver> cat grid_01.txt
... .4 58.
... 721 ...3
4.3 ... ...
21. .67 ...4
.7. ... 2..
63. .49 ...1
3.6 ... ...
... 158 ...6
... .6 95.
B&W/Solver> cat grid_01.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
B&W/Solver> make clean
rm -f *.d Solver grid_01.result.txt
B&W/Solver> ls
grid_01.txt  Makefile  Sudoku.c
B&W/Solver>

```

Figure 29: Solver en ligne de commande

Traitement d'image

Après avoir implémenté le solver de sudoku, je me suis intéressé à la partie traitement d'image du projet. Pour celui-ci, je me suis tout d'abord renseigné sur les bibliothèques qui étaient autorisées et que nous pouvions utiliser pour afficher les images. L'une des principales bibliothèques réalisant cela en C est SDL. Cependant, n'ayant aucune notion de SDL, j'ai dû me renseigner également sur cette bibliothèque, comment l'utiliser, afficher des images, récupérer des pixels... Ce fut donc une grosse partie de recherche et de test, avant que ma première image ne s'affiche avec SDL. Bien évidemment, par la suite, nous avions un TP de programmation qui portait sur SDL, et j'ai pu réutiliser certaines fonctions de ce TP afin d'améliorer mon code et ma compréhension de la bibliothèque.

Après avoir pu afficher une image grâce à SDL, je me suis intéressé aux modifications que je pouvais apporter à celle-ci et comment je pouvais les réaliser. Dans un premier temps, je me suis intéressé à la suppression des couleurs en mettant mon image en nuances de gris. L'une des manières la plus simple de réaliser celle-ci est d'additionner les composantes d'un pixel et de calculer leur moyenne en faisant $((R+V+B) / 3)$. Cependant, après avoir recherché quelles valeurs seraient les meilleures pour mettre une image en nuances de gris, j'ai trouvé un document qui expliquait que si nous voulions mettre une image en nuances de gris, puis ensuite la binariser (la mettre en noir et blanc) il fallait privilégier la composante verte du pixel, ensuite le rouge et négliger fortement la composante bleue. Ma formule pour la nuances de gris après recherche et test fut donc : $(0.2125 * r + 0.7154 * g + 0.0721 * b)$.

Ci-dessous le résultat de la formule pour mettre une image en nuances de gris.

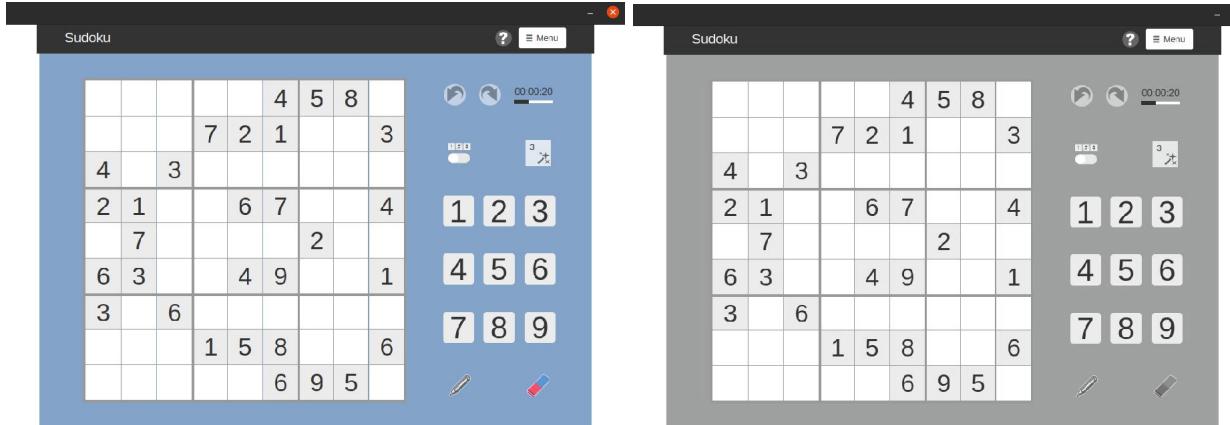


Figure 30: L'image normale et l'image en nuances de gris

L'image en nuances de gris étant faite, il m'a fallu me renseigner sur la binarisation d'une image (mettre une image en noir et blanc). Le principe de la binarisation est simple, une fois que notre image est en nuances de gris, nous la reparcourons et si notre pixel a une valeur supérieure à notre seuil (valeur comprise entre 0 et 255 défini au préalable), les composantes du pixel prendront la valeur 255, sinon elles prendront la valeur 0.

Cependant, malgré un principe simple, la binarisation se révèle très difficile. En effet, étant donné que les images ont toutes un seuil bien différent en fonction de la luminance, du contraste et bien d'autres paramètres, un seuil peut être valable sur une image, mais pas sur une autre. Il faudrait donc trouver une fonction qui calcule le seuil pour l'image qu'on entre en paramètre, puis effectuer la comparaison pour la binariser avec celui-ci.

Pour cela, j'ai donc essayé de nombreuses techniques durant une période de 1 mois à partir de la semaine de THLR, jusqu'à la première soutenance. Je vais vous expliquer certaines de ces méthodes que j'ai essayé, puis celle que j'ai finalement adoptée. Cependant, il faut savoir que je travaille toujours sur le prétraitement et que la méthode risque de subir quelques variations entre cette soutenance, et la soutenance finale de décembre.

La première méthode que j'ai essayé est une méthode assez naïve qui consistait à prendre comme seuil la moyenne de tous les pixels de mon image puis ensuite d'effectuer la binarisation. Bien évidemment, cette méthode s'est révélée peu efficace et j'ai donc essayé des variantes, en incrémentant ou décrémentant la moyenne obtenue par une valeur ou un pourcentage de celle-ci. L'ensemble de ces tests n'étant pas concluant, je me suis lancé dans des recherches sur les différents traitements que nous pouvions administrer à notre image afin qu'elle puisse être binarisée correctement.

Beaucoup de méthodes étaient basées sur la luminance et le contraste d'image avec des formules plus ou moins mathématiques qui permettaient d'accentuer certaines couleurs dans notre image, mais encore une fois, ces méthodes ne suffisaient pas pour obtenir un bon rendu d'image. Je tiens également à notifier qu'essayer l'ensemble de ces techniques était extrêmement chronophage et la plupart du temps, cela représentait une énorme perte de temps d'essayer de coder ces techniques pendant une soirée pour au final observer que le résultat n'était pas satisfaisant.

Par la suite, j'ai également testé des techniques qui permettaient de supprimer le grain de l'image, comme le flou Gaussien. Cette méthode consiste à effectuer la moyenne de tous les pixels qui entourent notre pixel courant puis de remplacer sa valeur par cette moyenne. Là aussi, cette méthode ne suffisait pas. Cependant, cette technique reste présente dans mon code final sans pour autant que je l'utilise, car je pense qu'elle me sera sans doute utile lorsque je ferai d'autres tests avec d'autres méthodes de binarisation.

Pour finir, j'ai également testé des méthodes de binarisation connues comme la méthode de Sauvola, Nick, mais également d'Otsu. Cependant, celles-ci ne m'apportaient pas un résultat concluant pour les images fournies dans le cahier des charges, et principalement pour les images 4 et 6 qui sont totalement opposées dans le type de traitement qu'il faut leur faire. En effet, l'une est sous-exposée et l'autre est surexposée.

Enfin, la méthode que je présente pour cette soutenance reste simple et à améliorer pour les images 4 et 6, mais présente de très bons résultats pour les 4 autres images du cahier des charges, mais également pour toutes les autres images que j'ai pu essayer avec cette méthode. Tout d'abord, j'effectue sur cette image une augmentation du contraste, ce qui me permet d'éclaircir l'image sans pour autant la dénaturer. Puis ensuite, je parcours toute l'image, et pour tous les pixels rencontrés, je prends comme seuil la moyenne locale de ce pixel avec une matrice carrée 13 par 13, le tout multiplié par 0.93. Je compare ensuite le pixel à son seuil, et s'il est supérieur, il prendra la valeur 255 sinon il prendra la valeur 0. Cependant, comme je l'ai dit, cette méthode est temporaire et je compte bien essayer de l'améliorer pour tenter d'obtenir de meilleurs résultats sur certaines images.

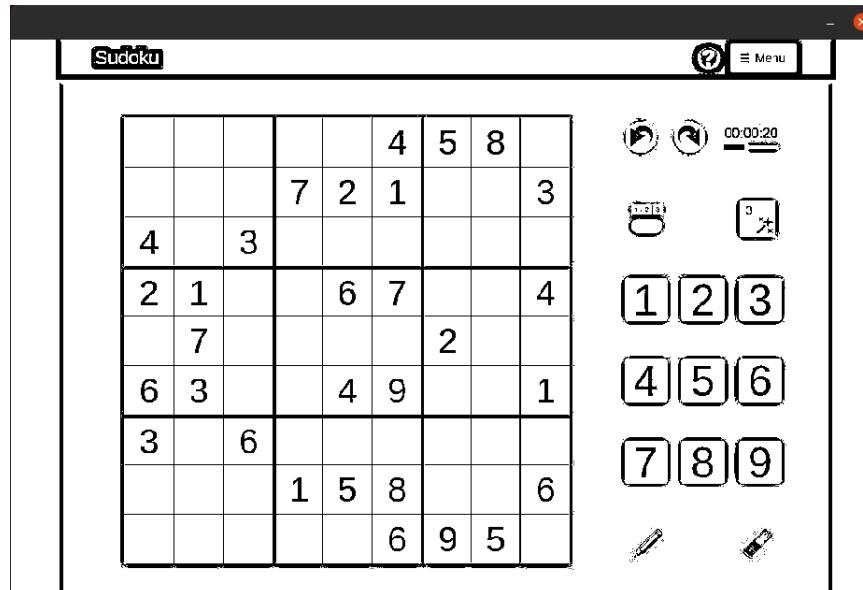


Figure 31: Résultat de la binarisation sur l'image

Pour finir sur le prétraitement d'image, j'ai créé un fichier main.c qui regroupe l'ensemble des autres fichiers et lors de la compilation et l'exécution, prend le path d'une image et sauvegarde l'image de base, l'image en nuances gris, mais également l'image binarisée dans le même dossier que l'executable afin que ces images puissent être affichées avec l'interface d'Angelina.

Seconde soutenance

Finalisation du traitement d'image

Tout d'abord, mon objectif pour cette soutenance était de fournir à Hugo un traitement d'image correct afin qu'il puisse détecter et découper correctement l'image prétraitée du Sudoku. Avec le travail que j'avais déjà fait en amont avec la précédente soutenance, j'obtenais une image assez bien traitée, mais il restait cependant beaucoup de bruits sur l'image, et lorsque nous découpons les chiffres, il subsistait avec, des résidus de ce bruit. Je me suis donc replongé dans l'amélioration du contraste de l'image en testant plusieurs méthodes afin d'assombrir ou d'éclaircir l'image en fonction de sa luminosité. Les premières méthodes que j'ai testé restaient assez simplistes, avec la création d'un seuil en fonction de la luminosité moyenne de l'image. En-dessous de celle-ci, j'ajoutais à tous les pixels de l'image une constante afin d'augmenter son contraste, puis au-dessus de celle-ci, je soustrayais une constante afin d'en diminuer sa luminosité, cela permettait aux images de faibles luminosités, de devenir plus lumineuses, et aux sur-exposées, de diminuer l'exposition. L'ensemble de ces méthodes basiques ne me donnaient aucun résultat concluant lorsque je testais différentes images. Il me fallait donc une méthode plus drastique mais tout autant générale afin de traiter au mieux toutes les images. En effectuant quelques recherches, j'ai trouvé une fonction avec un coefficient qui permettait d'augmenter plus nettement le contraste, et j'ai donc décidé de l'implémenter. Vous pouvez voir ci-dessous un exemple de l'image en nuance de gris avec la fonction de contraste et sans la fonction de contraste.



Figure 32: De gauche à droite : l'image non-contrastée/l'image contrastée

Par la suite, Angelina et Hugo m'ont fait part de leur demande d'avoir des bords ainsi que des chiffres plus épais afin de pouvoir mieux détecter les bords de l'image pour faire la rotation, mais également afin d'avoir des chiffres plus nets lorsque nous découpons l'image. Je me suis donc renseigné sur les différentes manières d'augmenter la densité des pixels noirs dans l'image. Premièrement j'ai fait une fonction qui augmentait la taille d'un pixel noir. En effet, dès que ma fonction croisait un pixel noir dans l'image, il effectuait une dilatation matricielle 3×3 et les pixels autour de celui-ci devenaient noirs. Pour une représentation visuelle de cette matrice, vous pouvez observer l'image ci-dessous.

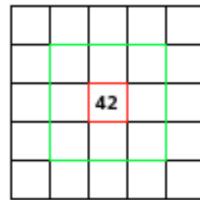


Figure 33: Matrice 3 par 3 pour la dilatation d'un pixel

Cependant, cette méthode dénaturait beaucoup l'image et cela faisait de gros amas carrés de pixels noirs, et après d'autres recherches, certaines personnes utilisaient, au lieu d'une matrice 3 par 3, une matrice en croix afin de mieux dilater les pixels d'une image. Ainsi avec cette méthode, uniquement les pixels à gauche, à droite, en haut et en bas deviennent noirs autour du pixel noir que nous sommes en train de traiter. Avec cette méthode, j'ai obtenu de très bons résultats (voir les images des chiffres sans et avec épaisseur ci-dessous) et nous avons donc décidé de conserver cette méthode pour l'épaisseur des pixels sur notre image.



Figure 34: De gauche à droite - Un 6 dilaté et non dilaté - la matrice en croix de dilatation

Assistance et assemblage de l'ensemble du projet

Par la suite, après avoir fini la partie traitement d'image, je me suis intéressé à la partie découpage et affichage dans l'interface. Tout d'abord, lorsque nous affichions une image dans l'interface qu'Angelina avait réalisée, nous avions un problème assez récurrent avec les tailles des images. En effet, celles-ci étaient trop grandes et dépassaient la taille de notre écran, et ainsi nous ne pouvions pas voir l'image en entier. Angelina m'a donc fait part de sa demande de réaliser une fonction qui redimensionnait une image afin qu'elle puisse être affichée correctement à l'écran. Dans le même intervalle, Hugo avait également besoin de cette fonction afin de donner à Mathéo et à son réseau de neurones les images des chiffres en 28 par 28 pixels. Cependant, j'avais déjà essayé de faire une telle fonction pour mes besoins personnels à la première soutenance. Je l'ai donc reprise et améliorée pour pouvoir l'implémenter dans le projet afin que mes camarades puissent l'utiliser en fonction de leurs différents besoins.

Après avoir fait cela, je me suis intéressé à la partie réseaux de neurones de Mathéo afin de savoir comment celle-ci marchait et si je pouvais essayer de l'aider. A ce moment là, la fonction que Hugo avait implémentée pour découper l'image fonctionnait parfaitement, cependant les chiffres qu'elle renvoyait au travers des tiles n'étaient jamais de la même dimension que ceux d'autres images. Ainsi, après discussion avec Mathéo, j'ai réfléchis avec Hugo à propos d'une méthode pour que sa fonction renvoie à chaque fois, et peu importe l'image, la même taille de chiffre dans les tiles. Après avoir proposé l'une de mes idées à Hugo, il l'a implémenté parfaitement tout seul et grâce à lui nous avions des tiles de chiffres parfaites pour chaque image avec la même dimension de chiffre peu importe celle-ci. Malgré tout, le réseau de neurones de Mathéo ne semblait pas être fonctionnel pour chacune des images. A ce moment là du projet, Mathéo utilisait une bibliothèque nommée MNIST afin d'entrainer son réseau de neurones. Etant donné que nous cherchions une formule afin que le réseau marche assez bien globalement, j'ai essayé de mon côté de retirer MNIST et de mettre à la place d'autres images de chiffres que ressortait notre algorithme à partir d'images d'internet. Après ces tests, je me suis rendu compte que le réseau marchait beaucoup mieux lorsque nous utilisions une bibliothèque faite à partir d'image comme celle-ci au lieu d'utiliser MNIST. J'ai fait part à Mathéo de ces résultats et à partir de ce moment là, après avoir suivi mon conseil et enlevé MNIST de l'entraînement de notre réseau de neurones, Mathéo a continué tout seul la partie entraînement du réseau de neurones jusqu'à avoir obtenu le réseau que nous vous présentons aujourd'hui.

Enfin, après avoir fait le tour des parties de chacun, il nous restait à assembler notre projet. Ainsi, avec Angelina, nous avons passé une grande soirée qui avait pour objectif de faire le makefile qui permettait de compiler tout notre projet. Au terme de cette soirée, nous avions notre projet qui compilait avec un make file qui permettait d'assembler toutes nos parties.

Conclusion personnelle

Pour finir, je dirais que grâce à ce projet, cela m'a permis d'en apprendre beaucoup sur le traitement d'image et les différentes méthodes de binarisation mais également de traitement global, comme la dilatation, la modification du contraste ou encore les différents flous applicables à une image. De plus, étant donné que j'avais beaucoup travaillé durant les premières semaines, j'ai pu finir légèrement plus tôt que les autres ma partie, et j'ai ainsi pu m'adapter à leurs besoins et travailler sur certaines fonctions qui leur ont permis de faciliter leur travail. De plus cela m'a également permis de faire le tour des parties et d'aider à la réalisation ou encore de donner quelques idées pour le travail de mes camarades. J'ai également beaucoup appris au niveau des make files et bien d'autres notions car j'ai dû passer certaines soirées à régler beaucoup de problèmes en liaison avec la mise en commun de tout le projet avec principalement Angelina. D'autre part, je me suis rendu compte de l'importance d'avoir un avis extérieur concernant son travail afin de l'améliorer et d'avoir de nouvelles idées d'implémentation. Enfin je vous laisse découvrir ci-dessous les différentes étapes d'un traitement complet d'une image que j'ai fait durant ce projet.

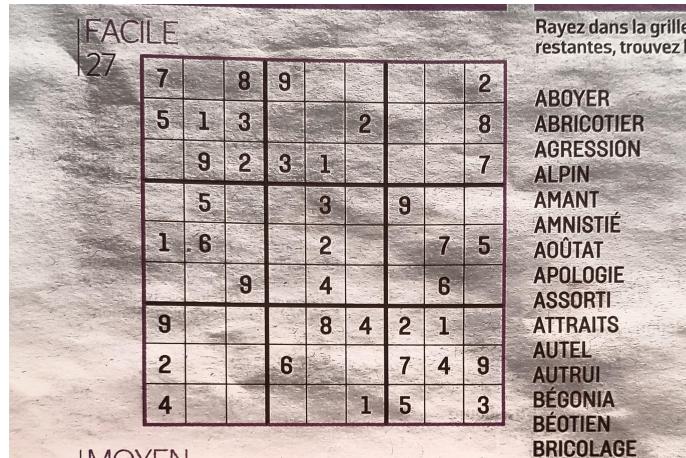


Figure 35: L'image normale

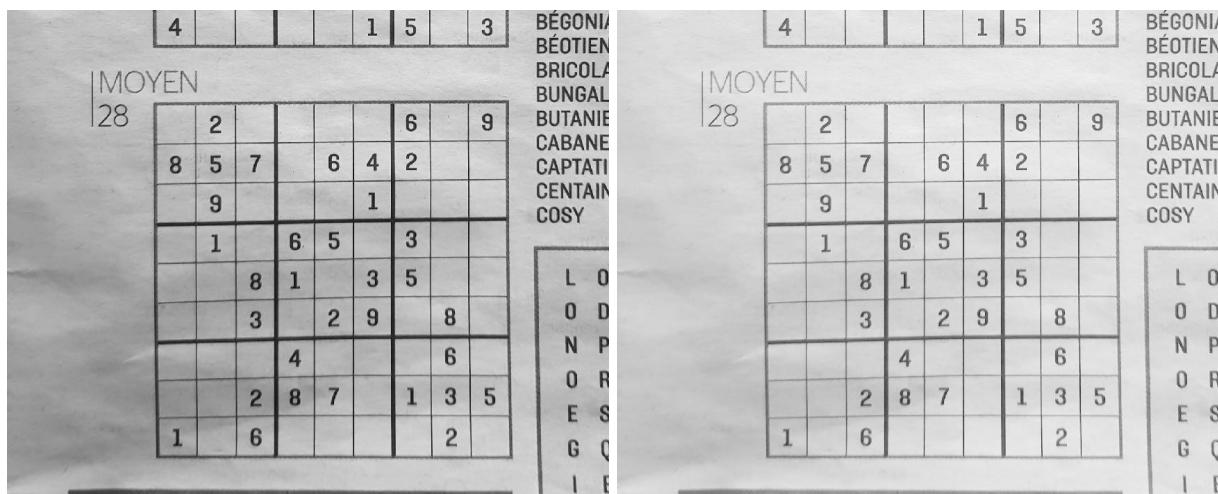


Figure 36: De gauche à droite : l'image non-contrastée/l'image contrastée

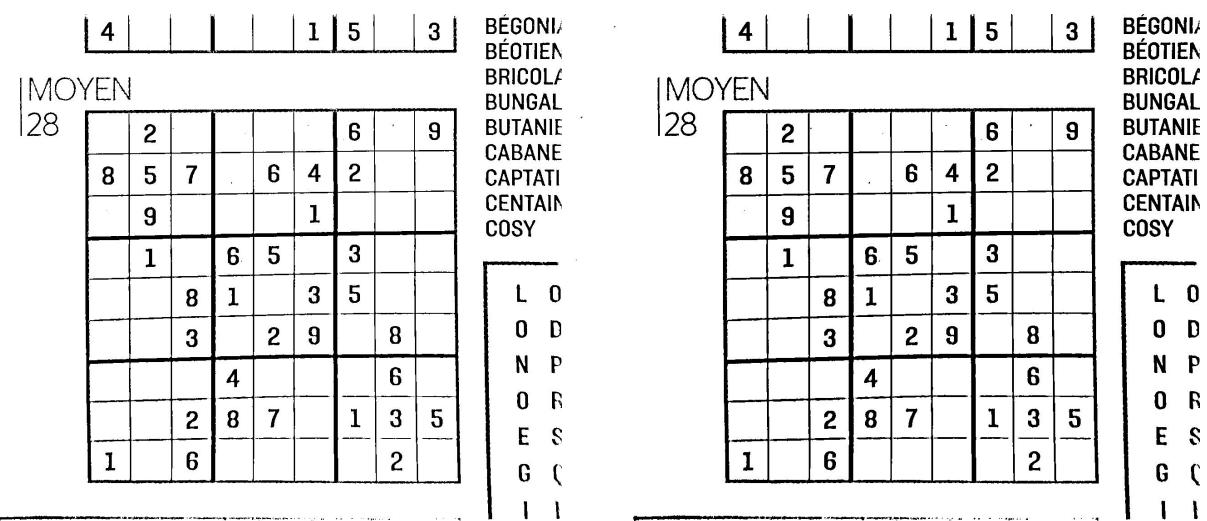


Figure 37: De gauche à droite : l'image non-dilatée/l'image dilatée

Conclusion générale

Ce projet aura donc pour nous tous été un formidable moyen de nous initier au C et à de nouveaux domaines et algorithmes, tous plus exotiques les uns les autres. Notre application est fonctionnelle et même si certaines améliorations restent possibles, nous obtenons des résultats plus que satisfaisants sur les images qui nous ont été fournies et bien d'autres encore. Au delà de l'aspect technique, il fut plaisant de pouvoir nous retrouver tous avec le même groupe que l'année dernière et de pouvoir à nouveau tous travailler ensemble, étant tous habitués au rythme des autres.

Merci d'avoir pris le temps de nous suivre lors de ce projet,

Le groupe HANABI

