



EPITA
2ÈME ANNÉE CYCLE PRÉPARATOIRE

OCR

Rapport de Soutenance 2

Auteurs

Raphaël NAHOUUM Mathis FRIGANT
Othmane ZIYAD Malo BEAUCHAMPS
Mickaël RAZZOUK

Septembre 2022 - Décembre 2022

Table des matières

1	Introduction	4
1.1	Présentation du groupe	4
1.1.1	Malo	4
1.1.2	Micka	4
1.1.3	Othmane	5
1.1.4	Raphaël	6
1.1.5	Mathis	6
2	Répartition des tâches et avancement	7
2.1	Répartition	7
2.2	Avancement	8
3	Notre avancement	9
3.1	Pré-traitement	9
3.1.1	Nuance de gris	9
3.1.2	Binarisation	10
3.1.3	Flou Gaussien	11
3.1.4	Sobel	12
3.1.5	Érosion et Dilation	13
3.1.6	Sauvegarde d'images intermédiaires	14
3.2	Réseau de neurone	15
3.2.1	Définition des objectifs et du contexte d'utilisation : . .	15

3.2.2	Préparation des données :	15
3.2.3	Choix de l'architecture du réseau de neurones :	16
3.2.4	Choisir les paramètres du modèle :	18
3.2.5	Implémentation du réseau de neurone	18
3.2.6	Problématique rencontrée	20
3.2.7	Performances finales :	22
3.2.8	Améliorations possibles :	23
3.3	Sudoku solver	23
3.3.1	I/O	23
3.3.2	Optimisation	23
3.3.3	Hexadoku Solver	24
3.3.4	Problèmes rencontrés	25
3.4	Écriture des résultats	26
3.4.1	Calculs et affichage	26
3.4.2	Adaptation	29
3.5	Save et load	31
3.6	Utils et tools : isempty et pic2list	31
3.6.1	isempty	31
3.6.2	pic2list	32
3.7	Site web	32
3.8	Interface utilisateur	33
3.8.1	Application principale	33
3.8.2	Application mobile	36
3.9	Manipulation de l'image	38
3.9.1	Détection de la grille	38
3.9.2	Détection des lignes	40
3.9.3	Rotation automatique	41
3.9.4	Recadrage automatique	41
3.9.5	Identification des cases	43

3.9.6	Implémentation finale	43
4	Conclusion	45

Introduction

1.1 Présentation du groupe

1.1.1 Malo

Je n'avais jamais fait de C avant l'EPITA, et ce projet m'a encore plus rassure, j'aime beaucoup le C. Étant un fervent adorateur de Linux (I use arch BTW), et de l'open source, le C est un langage bas niveau très important. Néanmoins, je n'avais jamais fait de C avant l'EPITA, et ce projet m'a encore plus rassuré, j'aime beaucoup le C. C'est un langage très puissant quand bien utilisé.

1.1.2 Micka

J'ai toujours touché à l'informatique en me lançant des projets personnels, je suis très intéressé par les défis techniques à relever durant ce projet, je pense qu'il me permettra de développer des nouvelles compétences, car il aborde des points plus techniques que ce à quoi je suis habitué. J'ai été responsable du réseau de neurone, ce qui m'a permis de beaucoup apprendre sur une technologie intéressante et plus généralement d'approfondir mes connaissances sur le langage C.

1.1.3 Othmane

Après une année relativement mouvementée à l'EPITA, je me retrouve aujourd'hui prêt à entamer ce projet de S3. Je m'appelle Othmane Ziyad, et j'ai dès à présent comme tâche de travailler sur ce projet. Trêve de blabla, je suis passionné par l'informatique et la programmation, mais sans grande surprise (sinon je ne serais pas à cette école). Après avoir découvert le plaisir de l'algorithmique en première avec la spécialité NSI (que j'ai bien évidemment gardée en terminale), j'ai eu la chance de développer mes compétences et prendre goût à la matière. De plus, une année d'étude à l'EPITA m'a aidé à progresser et mieux coder, ainsi que de mieux travailler en groupe. Il n'en est pas moins que j'appréhendais un peu le projet, notamment à cause du fait que je n'ai jamais touché de C avant cette année. Coder une Intelligence Artificielle de reconnaissance d'image en C me semblait hors de question. Tout de même, je me suis familiarisé avec le langage et ses techniques, et j'ai fait de mon mieux pour surmonter les difficultés que j'ai rencontré. Je sais faire preuve d'une grande autonomie et adaptation. J'ai de grandes attentes pour ce projet, car non seulement est-ce pour moi un but à atteindre et un challenge à surmonter, mais il me permettra également d'acquérir des compétences que je n'avais pas auparavant, ce qui m'aidera grandement dans le futur. Je suis convaincu qu'à l'aide de mon groupe, on réussira tous à se pousser de l'avant et créer un projet digne de ce nom.

1.1.4 Raphaël

Passionné d'informatique également, j'aime créer des projets avec des amis. J'aime également la rétro ingénierie (reverse engineering), j'ai principalement commencé avec du bytecode java depuis plusieurs années. Ceci m'a permis de comprendre beaucoup de concepts liés au développement de logiciels et au fonctionnement de certains systèmes informatiques.

1.1.5 Mathis

Je suis passionné de cybersécurité et fais partie de l'équipe HDFR. Je m'intéresse beaucoup à l'informatique et maintenant à tout ce qui est infrastructure réseau. J'ai aussi des bases en langage bas niveau : assembleur x86 et ++C. Membre de la communauté open-source depuis quelque temps, j'apporte une grande importance au respect de la vie privée.

Répartition des tâches et avancement

2.1 Répartition

Tâches	Malo Beauchamps	Mickael RAZZOUK	Othmane Ziyad	Raphael Nahoum	Mathis Frigant
Réseau de neurones		✓✓	✓	✓	
Traitemet d'image	✓✓			✓	✓
Analyse des fichiers	✓		✓✓	✓	
Séparation d'image		✓	✓		✓✓
GUI				✓✓	
Site web			✓✓		
Application	✓✓	✓		✓	✓
Écriture du résultat			✓✓		

Légende :

✓✓ = Responsable de la tâche

✓ = Travaille sur la tâche

2.2 Avancement

Tâches	Avancement
Réseau de neurones	100%
Traitement d'image	100%
Sudoku solver	100%
Séparation d'image	100%
I/O	100%
UI	100%
Site web	100%
Écriture	100%

Notre avancement

3.1 Pré-traitement

Dans le but d'avoir l'image la plus propre et nette possible, nous devons nécessairement passer par l'importante étape qu'est le pré-traitement d'image. Cela consiste à traiter l'image avant qu'elle ne soit analysée par l'algorithme de reconnaissance de chiffres. Le but de ce pré-traitement est de simplifier l'image et de rendre les chiffres plus faciles à détecter et à identifier.

3.1.1 Nuance de gris

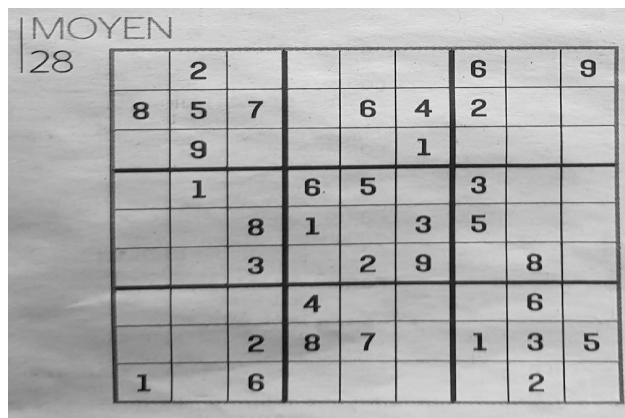
Il existe différentes techniques de pré-traitement d'image qui peuvent être utilisées pour améliorer les résultats de l'OCR. Par exemple, on peut passer d'une image en couleur à une image en nuances de gris pour simplifier les données et éliminer les bruits et les variations de luminosité. Le TP SDL2 m'a donc permis de mettre en pratique ces techniques de filtrage d'image et de les utiliser pour améliorer la reconnaissance de chiffres dans mon projet. Grâce à ce TP, j'ai pu expérimenter différents filtres et choisir les plus efficaces pour résoudre le problème de reconnaissance de chiffres. Cela m'a donné l'occasion d'approfondir mes connaissances en programmation et en traitement d'image, et j'ai beaucoup apprécié cette partie du projet. Pour cela, il faut calculer la valeur moyenne des couleurs de chaque pixel. Cette moyenne peut être calculée en utilisant les valeurs de rouge, vert et bleu de chaque pixel.

La formule utilisée est :

$$\text{average} = 0.2989 \times \text{rouge} + 0.587 \times \text{vert} + 0.114 \times \text{bleu}$$

Une fois que la valeur moyenne de chaque pixel a été calculée, il suffit de remplacer les valeurs de rouge, vert et bleu de chaque pixel par cette valeur moyenne, ce qui permet d'obtenir une image en nuances de gris.

Voilà le résultat obtenu :



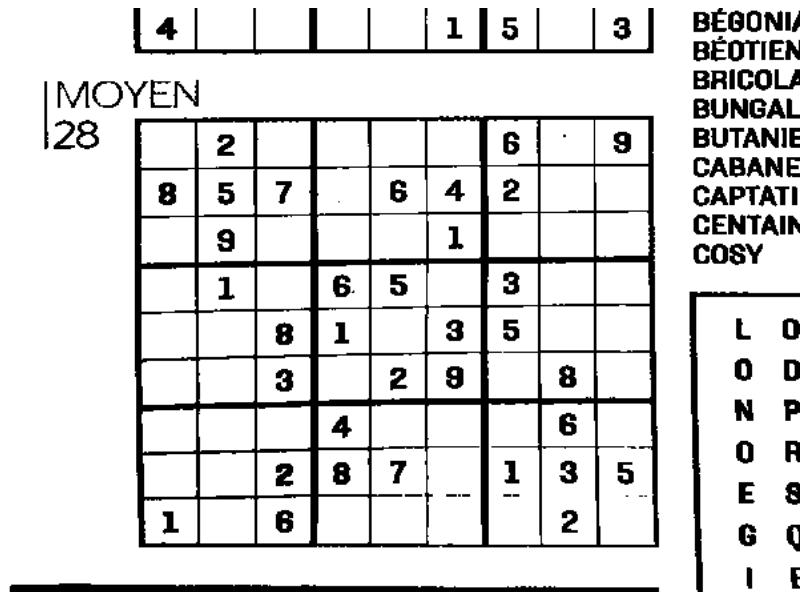
3.1.2 Binarisation

En plus de cela, nous avons implémenté l'otsu thresholding. L'Otsu Thresholding est un algorithme qui permet de passer d'une image en nuances de gris à une image en noir et blanc. Cet algorithme fonctionne en créant un histogramme des valeurs de chaque pixel dans l'image. Ensuite, il effectue un calcul pour trouver le seuil (ou threshold) optimal, c'est-à-dire la valeur qui permet de séparer les pixels en deux groupes distincts : les pixels plus clairs et les pixels plus sombres. Une fois que le seuil optimal a été calculé, l'algorithme peut binariser l'image en remplaçant chaque pixel par du blanc si sa valeur est supérieure ou égale au seuil ou du noir dans le cas contraire. Cette étape permet de simplifier l'image encore plus et de mettre en évidence les contours des chiffres qui se trouvent dans l'image.

L'Otsu thresholding est un algorithme qui calcule un seuil global, nous avons modifié cet algorithme pour qu'il découpe l'image en 9 carrés, puis calcule un

seuil moyen pour chaque carré et pour l'image dans son ensemble. Cela nous permet de traiter chaque carré indépendamment, ce qui améliore la précision de la détection des contours des chiffres.

Voici donc le résultat final après application de la binarisation.



3.1.3 Flou Gaussien

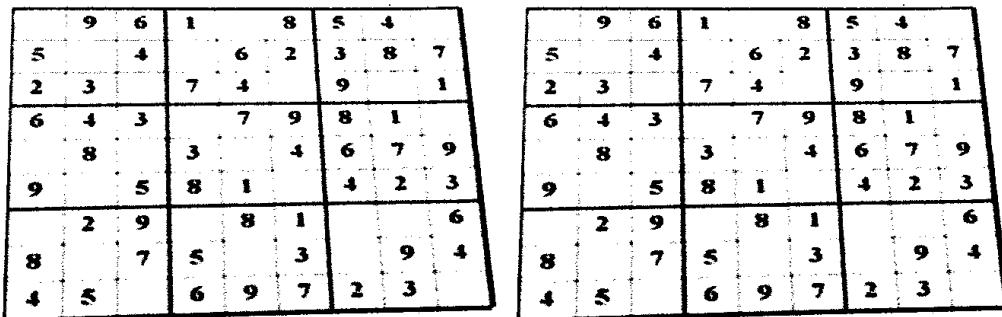
Nous avons implémenté le flou Gaussien dans notre programme. Cette technique permet de réduire le bruit dans une image en lissant les pixels. Elle utilise un masque de convolution, appelé noyau de Gauss, pour calculer une valeur moyenne des pixels voisins d'un pixel donné. Cette valeur moyenne remplace la valeur originale du pixel, ce qui permet de lisser l'image et de réduire le bruit.

Au début, nous avons rencontré quelques problèmes, dont le principal était que nous réécrivions sur les mêmes pixels que ceux que nous utilisions pour calculer les valeurs. Cela créait une boucle infinie et empêchait notre programme de fonctionner correctement. Nous avons dû trouver une solution pour résoudre ce problème, comme utiliser une copie temporaire des pixels pour effectuer les calculs. Cela nous a permis d'éviter d'écaser les données originales et

d'assurer un fonctionnement correct de notre programme, et donc éviter des images toutes bleues comme nous pouvons voir ici :



Voilà le résultat sans/avec flou gaussien :

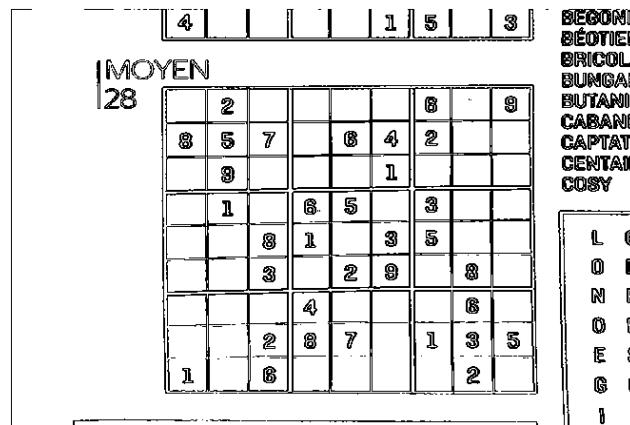


Le résultat n'est pas forcément visible à l'oeil nu, mais au niveau des pixels les programmes suivant auront un meilleur résultat.

3.1.4 Sobel

Nous avons également implémenté l'algorithme de Sobel dans notre programme. Cet algorithme permet de mettre en évidence les contours d'une image en

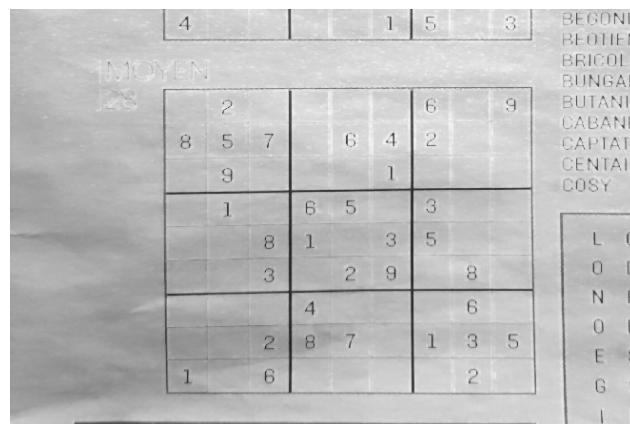
utilisant des opérations de convolution pour calculer les dérivées des fonctions d'intensité des pixels. Cependant, après avoir effectué de nombreux tests, nous n'avons pas obtenu de résultats très satisfaisants avec cet algorithme. Nous avons donc essayé d'autres méthodes pour améliorer la reconnaissance des chiffres dans l'image.



3.1.5 Érosion et Dilatation

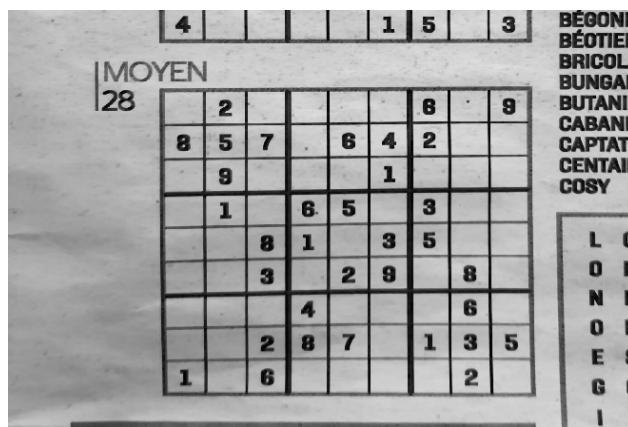
Les algorithmes de dilation et d'érosion sont des techniques couramment utilisées en traitement d'image pour améliorer la détection de contours et la reconnaissance des formes dans une image. La dilation consiste à agrandir les régions sombres d'une image en remplaçant chaque pixel par le pixel le plus sombre de ses voisins. Cette technique permet de mettre en évidence les contours des objets et de les rendre plus visibles.

Voici ce que renvoie la dilation :



L'érosion, quant à elle, consiste à rétrécir les régions sombres d'une image en remplaçant chaque pixel par le pixel le plus clair de ses voisins. Cette technique permet de réduire les bruits et les variations de luminosité dans l'image, ce qui peut améliorer la reconnaissance des formes.

De la même façon, l'érosion :



En utilisant ces deux algorithmes de manière combinée, nous pouvons obtenir de meilleurs résultats pour la détection de lignes et la reconnaissance des chiffres dans une image. Ces techniques peuvent être utiles pour différents types d'applications, comme la reconnaissance de caractères, la détection de formes ou encore l'analyse d'images médicales.

3.1.6 Sauvegarde d'images intermédiaires

Ensuite, nous sauvegardons les images binaires et en niveaux de gris pour les passer à la rotation et à la détection des grilles mais aussi pour la GUI. La rotation des images est importante car les grilles peuvent être inclinées dans l'image d'origine, ce qui peut rendre difficile la reconnaissance des chiffres. La détection des grilles permet de séparer les chiffres individuels pour les traiter séparément, ce qui améliore la précision de la reconnaissance. Ces étapes sont cruciales pour la performance de l'OCR.

Et finalement, quand nous aurons toutes les cases découpées, il nous reste à dilater l'image pour une meilleure compréhension par le réseau de neurones

3.2 Réseau de neurone

3.2.1 Définition des objectifs et du contexte d'utilisation :

Avant de commencer à implémenter un réseau de neurones, il est important de déterminer clairement les objectifs poursuivis et le contexte d'utilisation. Cela permettra de choisir le type de réseau de neurones le plus adapté et de préparer les données nécessaires pour l'entraînement. Dans le cadre de cette 2 ème soutenance, nous cherchons à implémenter un réseau de neurone qui reconnaît les chiffres.

3.2.2 Préparation des données :

Avant de pouvoir entraîner un réseau de neurones, il est nécessaire de disposer de données d'entraînement, et de validation. Ces données doivent être préparées et structurées de manière à pouvoir être utilisées par le réseau de neurones. Cela implique des étapes de nettoyage et de normalisation des données.

Nous utilisons le TMNIST une version non manuscrite du MNIST. Le TMNIST est un bon jeu de données pour l'entraînement d'un réseau de neurones pour plusieurs raisons. Tout d'abord, il contient un très grand nombre d'exemples d'images de chiffres, ce qui permet de bien entraîner le réseau de neurones. En outre, les images dans le TMNIST sont toutes de taille uniforme et ont été pré-traitées pour faciliter le traitement par le réseau de neurones. Enfin, le TMNIST est largement utilisé dans la communauté de l'apprentissage automatique, ce qui facilite la comparaison des résultats obtenus avec d'autres modèles. Ensemble, ces caractéristiques font du TMNIST un jeu de données idéal pour l'entraînement d'un réseau de neurones.

Un jeu de données varié permet d'obtenir de meilleures performances, car il permet de mieux capturer la diversité des données réelles auxquelles le modèle

sera confronté lors de son utilisation. Plus le jeu de données est varié, plus il est probable que le modèle appris soit capable de généraliser correctement à des données qu'il n'a pas vues lors de l'entraînement. En revanche, si le jeu de données est peu varié, il est plus probable que le modèle soit sur-entraîné et qu'il ne soit pas capable de généraliser. En résumé, un jeu de données varié permet d'obtenir de meilleures performances en apprentissage automatique car il permet d'éviter l'overfitting et de mieux réagir face à des données nouvelles.

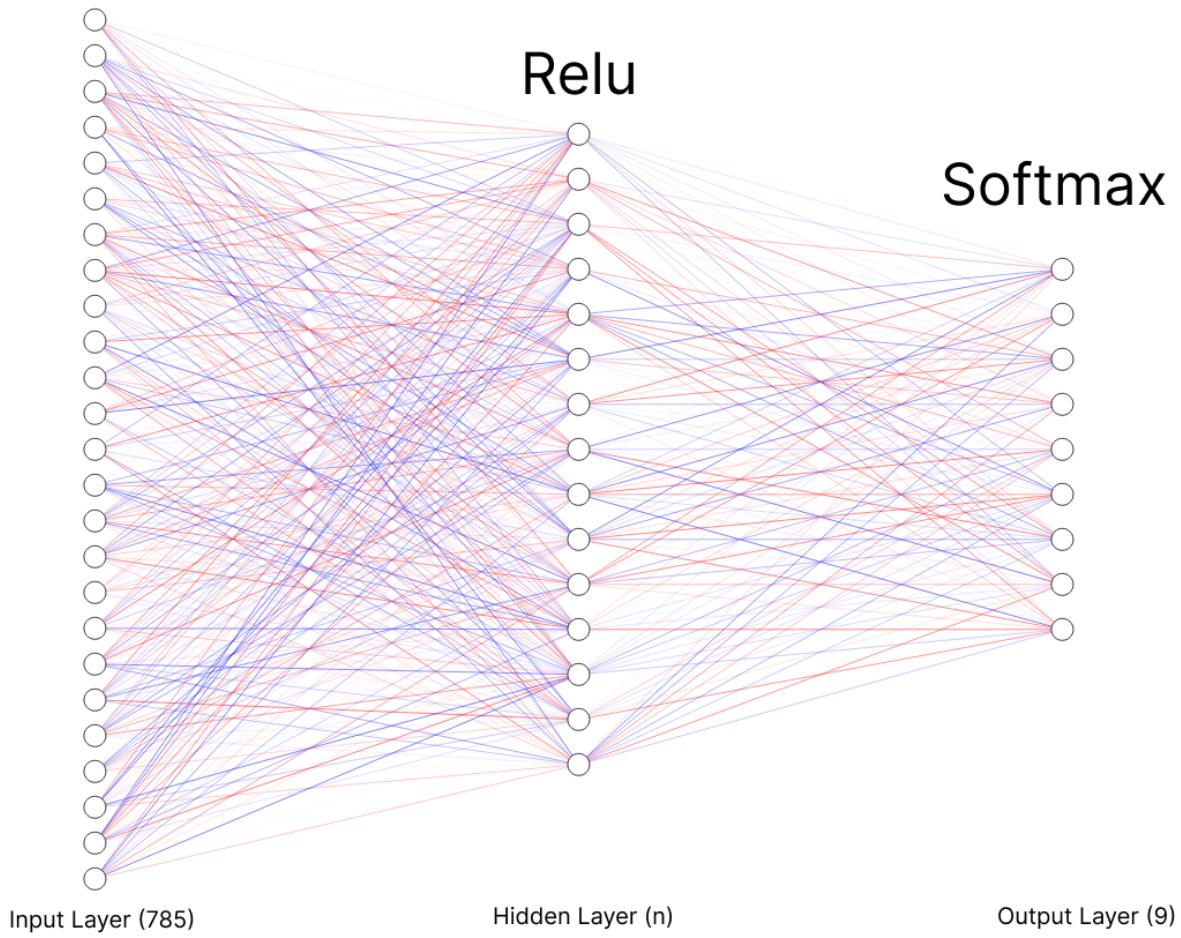
3.2.3 Choix de l'architecture du réseau de neurones :

Nous avons décidé de partir avec l'architecture suivante :

Input Layer : 784 neurones

Hidden Layer : Au choix

Output Layer : 9 neurones



Aspect final de notre réseau de neurones.

La couche d'entrée de 784 neurones est adaptée pour recevoir les données d'une image de 28 pixels sur 28 pixels, soit 784 pixels au total. La couche cachée de taille modulable permet de déterminer quelle configuration offre les meilleures performances en fonction des données. La couche de sortie de 9 neurones est adaptée pour la reconnaissance des 9 chiffres que nous souhaitons détecter.

3.2.4 Choisir les paramètres du modèle :

Le choix des paramètres d'un réseau de neurones peut avoir un impact significatif sur les performances du modèle. Les paramètres sont des valeurs qui ne sont pas apprises directement à partir des données d'entraînement, mais qui influencent la façon dont le modèle apprend et se comporte. Je parle ici du nombre de neurones par couche, des fonctions d'activation à utiliser, et le taux d'apprentissage. Trouver les bons paramètres peut nécessiter de faire des essais, des erreurs et de la recherche.

Il n'existe pas de valeurs universelles pour ces paramètres, mais les informations que nous avons pu trouver sur des forums rassemblant la communauté des data scientists nous ont guidé vers cette configuration :

- Fonction d'activation relu sur la couche cachée
- Fonction d'activation softmax sur la couche de sortie

Pour les valeurs du nombre de neurones caches et de la learning rate, nous avons commencé par des valeurs aléatoires en retenant les modèles les plus performants.

Nous avons donc retenu une vingtaine de neurones caches et une learning rate de 0.0001.

3.2.5 Implémentation du réseau de neurone

Depuis la dernière soutenance, nous avons amélioré l'implémentation de notre réseau de neurones en utilisant des structures pour simplifier les opérations effectuées lors de chaque étape de l'entraînement. Cela nous a permis de rendre notre code plus lisible et plus facile à maintenir.

Nos structures :

Nous avons 2 structs : La première struct est celle du réseau de neurone, elle contient les paramètres du réseau : - Les poids - Les biais - Les tailles de chaque

couche. La 2eme struct est la struct training qui contient l'environnement d'entraînement à savoir : - Un réseau de neurone initialiser aléatoirement - Un double* qui pointer vers le début du training set avec un char* qui en parallèle stock les résultats attendus. - Les valeurs d'activation sur toutes les layer qui sont nécessaires à la retropropagation mais pas à l'utilisation du model.

Implémentation de l'entraînement :

Pour lancer un entraînement, nous initialiser un réseau de neurone aléatoirement pour le placer dans un environnement d'entraînement ou le training set a été chargé. Pour chaque epoch, nous mixons l'ordre des éléments du training set et lançons successivement la forward et backward propagation.

Une fois le nombre d'epoch écoulé, nous calculons le f1-score sur un data set de notre choix puis les valeurs dans la struc du réseau sont sauvegardés sous la forme d'un fichier binaire afin d'être rechargé plus tard. C'est également une amélioration par rapport à la précédente soutenance où nous écrivions dans des fichiers txt bien plus lourds et long à ouvrir.

Implémentation de job :

Nous définissons un job par une requête effectué à notre réseau dans l'attente d'une réponse, hors du cadre de l'entraînement, nous attendons une réponse précise. définissons Pour ce faire, nous commençons par initialiser un réseau de neurone, mais a la différence du training, nous utilisons un modèle sous forme binaire préalablement entrainé pour obtenir les valeurs de nos poids et biais.

Nous convertissons et normalisons l'input qui est une image de 28x28 grâce à la fonction PicToList puis nous appliquons une seule forward propagation pour déterminer quel neurone de sorte a la plus haute activation. d'indice de ce neurone dans la couche de sortie est la réponse du réseau.

3.2.6 Problématique rencontrée

Instabilité numérique

Lors du développement de ce projet, nous avons remarqué des NaN apparaître dans nos array de poids et biais après quelques recherches, nous avons pu mettre un nom sur ce phénomène : instabilité numérique.

L'instabilité numérique est un problème courant dans les modèles de réseaux de neurones, en particulier lorsque ces modèles sont entraînés sur de grandes quantités de données. L'instabilité numérique peut se manifester de différentes manières, notamment :

L'explosion des gradients, qui se produit lorsque les gradients des poids des réseaux de neurones augmentent de manière excessive. Cela peut entraîner des erreurs d'arrondi et des problèmes de convergence.

Le Vanishing gradient, qui se produit lorsque les gradients des poids des réseaux de neurones diminuent de manière excessive. Cela peut empêcher les poids des réseaux de neurones de se mettre à jour correctement et ralentir la convergence des modèles.

Les oscillations dans les poids des réseaux de neurones, qui peuvent se produire lorsque les gradients des poids oscillent de manière excessive. Cela peut entraîner une instabilité dans les performances du modèle et ralentir la convergence.

Nous avons eu la chance de compter parmi nos problèmes l'explosion des gradients et les oscillations dans les poids.

Pour éviter l'instabilité numérique dans les modèles de réseaux de neurones, il est important de bien ajuster les paramètres, tels que le taux d'apprentissage. De plus, il peut être utile d'utiliser des techniques telles que la normalisation des données pour améliorer la stabilité des modèles. Dans notre cas, la normalisation des entrées et des fonctions d'activation nous a permis de résoudre

l'explosion des gradients et diminuer la learning rate nous a permis d'échapper à l'oscillation dans les poids.

La normalisation des entrées consiste à diviser par 255 la valeur grayscale du pixel pour maintenir la valeur perçue par le réseau entre 0 et 1 afin d'éviter la divergence des valeurs.

La normalisation de la fonction softmax consiste à modifier la fonction de manière à soustraire la valeur maximum du vecteur à tous ses éléments avant d'appliquer la softmax classique.

Comparaison des modèles difficile

Il est difficile de comparer les performances de différents modèles de machine learning en raison des différences dans les données utilisées pour entraîner et évaluer les modèles et des différences dans les paramètres des modèles.

Pour comparer de manière objective les performances de différents modèles, il est important de s'assurer que les données utilisées pour entraîner et évaluer les modèles sont identiques.

Pour nous aider à faire cela, nous avons implémenté une fonction qui donne le f1-score d'un modèle donné sur un data set donné.

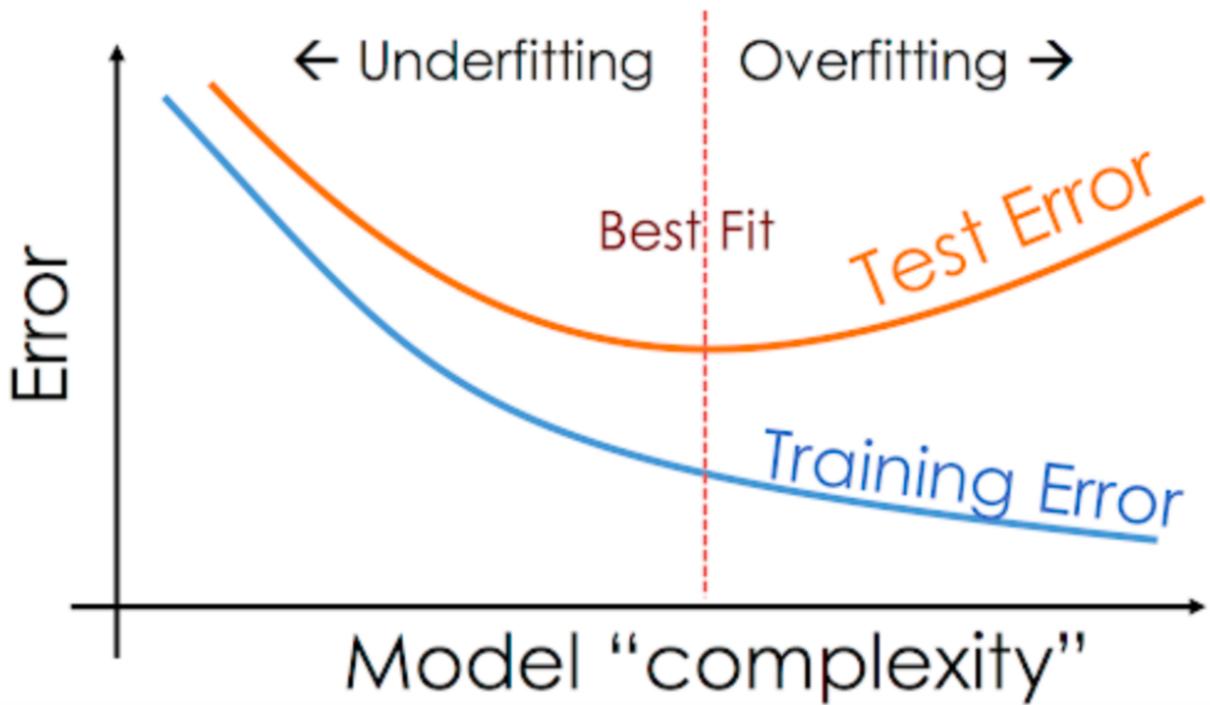
Le F1 score est un indicateur de performance couramment utilisé dans les tâches de classification. Il prend en compte à la fois la précision et le rappel d'un modèle, ce qui le rend plus adapté que l'utilisation de ces deux mesures séparément pour évaluer les performances d'un modèle de réseau de neurones.

La précision mesure le pourcentage de prédictions correctes parmi toutes les prédictions effectuées par le modèle. Le rappel, quant à lui, mesure le pourcentage d'éléments appartenant à la classe cible qui ont été correctement prédits par le modèle.

Le F1 score combine ces deux mesures en calculant la moyenne harmonique entre la précision et le rappel.

3.2.7 Performances finales :

Après entraînement et comparaison des modèles, nous avons remarqué que trop d'entraînement peut entraîner l'effondrement d'un modèle, également connu sous le nom d'overfitting. L'overfitting se produit lorsqu'un modèle est entraîné sur des données d'entraînement de manière trop approfondie, de sorte qu'il commence à capturer les bruits et les détails spécifiques des données d'entraînement plutôt que les tendances générales qui peuvent être généralisées à de nouvelles données. En conséquence, le modèle peut se comporter de manière très précise sur les données d'entraînement, mais il sera moins précis sur de nouvelles données. Ce qui n'est pas souhaitable dans notre cas.



Exemple d'overfitting vs underfitting

Comme visible sur le graphe, la difficulté est de savoir quand nous sommes sur cette ligne, nous avons essayé de nous en approcher le plus possible.

3.2.8 Améliorations possibles :

Nous avons noté des perspectives possibles d'amélioration que nous n'avons pas eu le temps où la possibilité d'implémenter, mais qu'il nous semble intéressant de mentionner. Dans un premier lieu, nous aurions pu implémenter une learning rate qui évolue au fil des epoch avec une fonction de décroissance exponentielle de manière à obtenir une correction agressive au début de l'entraînement et au fur est à mesure affinée les corrections pour éviter de déséquilibrer le réseau.

Le choix d'une meilleure loss function que celle utilisée (simple delta entre résultat obtenu et résultat attendu) aurait pu permettre au modèle de mieux s'adapter aux données et améliorer ses performances.

Enfin, la recherche d'un modèle performant a été difficile et nous ne prétendons pas avoir su exploiter tout le potentiel de notre réseau de neurone, nous aurions pu automatiser la recherche de paramètres et l'entraînement de modèle ce qui nous aurait permis d'augmenter les performances du réseau de neurone.

3.3 Sudoku solver

3.3.1 I/O

Pour résoudre un sudoku, il faut d'abord charger un fichier contenant la grille à résoudre. Le solver accepte différents types de sudoku en fonction de leur taille (9x9, 16x16, etc.). La première étape consiste à lire le fichier pour en extraire les informations nécessaires.

3.3.2 Optimisation

L'algorithme de base que nous avions implémentés était l'algorithme naïf. Nous utilisions la technique de backtracking pour résoudre le sudoku. La résolution utilisait trois fonctions principales : isboardvalid() qui vérifie si la

board passée en argument est valide et/ou solvable, issolved() qui vérifie si la board passée en argument est résolue, et solve(), le cœur du solver, qui résoud le sudoku.

En réalité, il existait déjà une optimisation pour le code, celle sur le isboardvalid(). Bien que la fonction la plus coûteuse soit le solve (évidemment), elle utilise à chaque appel isboardvalid() et issolved (qui appelle également isboardvalid()). Globalement, une optimisation sur le isboardvalid() se répercuterait sur toutes les fonctions. isboardvalid(), par nature, doit parcourir toute la board pour s'assurer que celle-ci ne soit pas invalide. La façon de gérer ce problème peut très facilement devenir coûteuse, surtout car on devrait faire plusieurs parcours pour lire les lignes, colonnes, mais aussi les carrés de 3x3.

La façon dont on a implémenté isboardvalid(), cependant, est à nos yeux la façon la plus optimale et la plus rapide de parcourir la board. On n'effectue qu'un seul parcours de la board, en définissant une liste pour les lignes, colonnes, et carrés. Dès que l'on rencontre une case qui n'est pas nulle, on la marque. Ainsi, au fur et à mesure du parcours, dès que la fonction rencontre une case qui a été rencontré auparavant, et qui est marquée, elle sait que la board n'est pas valide, et return 0 instantanément. Sinon, elle effectue son unique parcours jusqu'au bout, et return donc 1.

3.3.3 Hexadoku Solver

En théorie, le hexadoku solver fonctionne quasi identiquement au sudoku solver, à quelques nuances près. En théorie. En réalité, il y a beaucoup de différences, et simplement transposer le code en changeant les bornes ne suffit pas. Il faut modifier tout le code.

La première nuance est, comme cité précédemment, les bornes. Un hexadoku est une grille de 16x16, comparé au simple 9x9 d'un sudoku. Les listes des

isboardvalid(), ainsi que les boucles, changent ainsi. Il faut parcourir toutes les cases, ainsi que les carrés additionnels, ce qui change considérablement le code.

La façon dont nous avons choisi d'implémenter le hexadoku solver garde le système de int array, plutôt que de char array. Notre code du sudoku solver normal traitant des arrays de int, il aurait été fastidieux de modifier toute la structure de nos fonctions. Ainsi, le hexadoku solver tel qu'on l'a traité est une grille de 16x16 de 16 entiers, allant de 0 à 15. Bien évidemment, 10,11,12,13,14,15 correspondent respectivement à A,B,C,D,E, et F. Ultimement, on parcourra la liste résolue finale, et créera une liste de caractères correspondante. Ainsi, nous obtiendrons un hexadoku résolu avec les bonnes valeurs.

Il y avait toutefois un imprévu. Dans un hexadoku, le 0 est une valeur. Il est possible et même nécessaire d'utiliser la valeur 0 pour résoudre un hexadoku. Or, dans le sudoku que l'on a codé jusqu'alors, le 0 n'était pas une possibilité, on l'utilisait donc pour reconnaître une case vide. Cela changeait fondamentalement le code. Nous avons essayé de remplacer les 0 par des NULL, et de gérer les cas particuliers dans le solve, mais cela revenait au même. Nous avons finalement décidé d'opter avec une solution plus facile, simplement marquer les cases vides par un -1. Il faudrait donc rajouter le cas du 0, mais outre cela, tout se comporterait de manière analogue.

Bien évidemment, il fallait adapter les fonctions issolved() et solve() également : cela va de soi.

3.3.4 Problèmes rencontrés

En ce qui en est de l'optimisation du sudoku solver classique, nous avons essayé d'optimiser la fonction solve() également. Nos recherches nous ont montré qu'il existait plusieurs façons "d'optimiser" celle-ci : Une modification

du solve qui vérifie un chiffre avant de l'essayer, une modification où l'on utilise des bitshifts, et une modification qui estime quel chiffre utiliser en fonction des précédents. Après s'être lancé sur ces pistes, et s'être heurté à quelques difficultés, nous avons remarqué une propriété non anodine : les complexités de toutes les fonctions citées précédemment sont exactement les mêmes. Toutes les méthodes de résolution de sudoku solver que l'on a trouvé sont d'ordre de complexité $O(9^{(N*N)})$.

Au vu des difficultés que l'on a rencontré lors de leur implémentation, et à la différence minime qu'elles apporteraient à notre projet, ainsi que le temps limité qu'il nous restait, nous avons pris la décision de garder le solver tel qu'il est.

Dans le cas du hexadoku solver, il ne marche tout simplement pas. Nous avons essayé de trouver le problème, mais en théorie le solver fait exactement ce qu'il devrait faire. Après avoir écrit une liste de 256 entiers à la main, nous l'avons passé en argument du hexsolve(), et le solver ne la résolvait tout simplement pas. Ce qui est étrange, car mis à part les modifications que l'on a apportées pour l'adapter au cas de l'hexadoku, le code fonctionne de manière virtuellement identique au solve normal. Le principe reste exactement le même. Au bout de quelques jours de séchage complet, nous avons décidé tout comme avant qu'il était sûrement plus judicieux d'abandonner l'idée, de plus que les autres parties n'étaient pas adaptées pour traiter un hexadoku solver. Dans le meilleur des cas, nous aurions un hexadoku solver, mais rien pour l'utiliser, ce qui n'est franchement pas très pratique.

3.4 Écriture des résultats

3.4.1 Calculs et affichage

Résoudre le sudoku dans les fichiers, c'est bien. Afficher la réponse, c'est mieux. En effet, il serait fâcheux d'avoir codé le solver si ce dernier ne renvoyait pas à

l'utilisateur la réponse de manière visuelle. Il y avait une myriade de façon de procéder, de plus que le choix revenait entièrement à nous. La façon la plus simple aurait été de créer notre propre grille, avec des dimensions prédéfinies, et de simplement renvoyer à l'utilisateur cette grille résolue. Ainsi, nous aurions assez peu calculs à faire, et la fonction pourra être codée facilement et en peu de temps.

Nous avons donc décidé de ne pas faire cela. Plutôt, nous avons opté pour l'option que nous considérions la plus compliquée : Écrire le résultat sur l'image de départ.

Cette façon se divise donc en plusieurs étapes. Premièrement, il s'agit de load l'image, la copier, et de pouvoir la modifier en écrivant dessus. Ensuite, il faut la parcourir, et dès lors que l'on sait qu'on a une case vide, on écrit le chiffre correspondant sur la case. S'il y a déjà un chiffre dans la case, alors on n'écrit rien. Finalement, il faut sauvegarder cette nouvelle image dans le fichier.

Après quelques recherches, il s'est avéré qu'il existe une façon spécifiquement pour écrire sur les images en C : SDL-ttf. La documentation trouvée en ligne n'était pas des plus claires : assez peu de sites la détaillent, et tous se ressemblent globalement. Il fallait donc procéder par tâtonnement pour bien manier l'outil (ou trial and error comme le disent si bien les Anglais). Un des premiers obstacles que l'on a rencontrés était la fonction qui écrit... qui n'écrit tout simplement pas. Nous avons testé sur une image totalement blanche, que l'on a ouvert, sur laquelle on a écrit, et qu'on a sauvegardé en tant que nouvelle image. Petit problème : l'image originale et l'image finale était totalement identiques. La fonction ne faisait tout simplement rien. Et bien que l'on ait essayé de modifier le code, voir où cela ne marchait pas, soit la fonction ne faisait rien au mieux, elle "segfaultait" au pire. Après un bon bon bon bout de temps, il s'est avéré que le problème venait du fait que l'on n'initialisait pas SDL. Après avoir ajouté cette ligne, le code marchait

exactement comme il fallait.

Nous ne sommes toutefois toujours pas sortis de l'auberge, loin de là d'ailleurs, car nous ne faisons que commencer. Nous pouvons désormais écrire sur l'image, reste à savoir où. Pour cela, la solution la plus ingénieuse est de passer en argument à notre fonction deux arrays : une array du sudoku non résolu, et une du sudoku résolu. Ainsi, en faisant la différence des deux, on peut facilement voir quelles cases sont celles que l'on a rempli, et sur lesquelles on doit donc écrire, et celles qui étaient déjà là préalablement, et donc que l'on saute.

En réalité, puisque l'argument que la fonction prend est l'image de la grille, avec quelques calculs astucieux, une grande partie du travail est déjà faite. Si l'image est celle de la grille, alors en divisant la width par 9, et la height par 9, on pourrait accéder à toutes les cases successivement. Avec une simple boucle dans une boucle, on parcourt les cases de l'image qui nous intéressent, et les traitons conséquemment. Or, si l'on combine cela avec le calcul que l'on a fait précédemment, on se rend vite compte d'une chose : on peut parcourir l'image ET les listes en utilisant le même i et j. Il faudrait pour cela parcourir l'image comme décrit, et pour chaque case où l'on se trouve, vérifier son équivalent dans la liste non résolue. Si c'était un 0, il suffirait donc d'écrire le chiffre correspondant dans la liste résolue. Si ce n'est pas un 0, on ne fait tout simplement rien.

Une fois cela fait, il suffira de sauvegarder la nouvelle image dans le dossier. Un premier test nous a donné cela :

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

3.4.2 Adaptation

On le remarque, la police n'est pas du tout adaptée. Celle-ci est largement trop petite (haha) pour la grille. En réalité, calculer la taille de la police risquait d'être difficile car cette dernière n'est en théorie pas proportionnelle à la taille de l'image.

Il serait donc en théorie impossible de calculer la taille de la police par rapport à la size de l'image avec précision. Il s'agissait donc de trouver une relation de sorte à ce que la police soit convenable dans la grille. Après plusieurs tests et plusieurs relations échouées (l'opération modulo n'a jamais été utilisée aussi fréquemment que cette journée), nous avons fini par opter pour une solution des plus simples. On prend la hauteur d'une case individuelle, et on la multiplie par 2. Cela nous donne une estimation correcte de la taille du font. Lorsque l'on a fait les test, on a obtenu les résultats suivants :

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

3.5 Save et load

En ce qui en est de la sauvegarde et du load, notamment pour les poids, on fonctionnait principalement en .txt avant. A l'aide de fprintf, nous remplissions les données qui nous intéressaient sur des fichiers textes. Nous avons décidé de changer cela, pour être plus efficaces, prendre moins d'espace, et de manière générale être plus optimisé.

Nous avons tout d'abord utilisé un système en hexadécimal, donc un simple système de write et de read. Vu que cela prenait des références, il n'était pas aussi dur que le sauvegarder en .bin. La fonction était très simple, la taille du fichier réduite, et le code s'exécutait rapidement.

Ultimement, vu que nous avions fini cette tâche avec de l'avance, nous avons décidé de les sauvegarder en binaire. C'était un peu plus complexe, car nous avons dû passer quelques paramètres en argument, notamment les longueurs des listes que l'on voulait, pour pouvoir les récupérer par la suite lors du load. Cependant, en faisant cela, on s'assurait du bon fonctionnement des fonctions, ce qui était donc notre priorité. Nous avons fini par adopter le système en .bin, bien qu'il soit équivalent à celui en hexadécimal niveau taille de fichier.

3.6 Utils et tools : isempty et pic2list

3.6.1 isempty

La fonction isempty prend en paramètre une image et détermine si elle contient un nombre avant de l'envoyer au réseau de neurones. Pour ce faire, nous mesurons la densité en pixels de couleur blanche sur l'image puis nous comparons son ratio par le nombre total de pixels. Si le nombre de pixels de couleur blanche est trop grand, nous déterminons que cette case ne contient pas de chiffre et donc qu'elle n'est utile à envoyer au réseau de neurones. Grossièrement, la fonction parcours l'image, fait une moyenne de couleur, et si celle-ci

dépasse un certain seuil, on considère que la case est vide. Sinon, elle contient un chiffre et on renvoie faux.

3.6.2 pic2list

La fonction pic2list, quant à elle, va récupérer une image pour la convertir en image exploitable par le réseau de neurones. Le réseau de neurones à besoin d'une image de taille 28 par 28 pixels et en nuances de gris, chaque pixel doit correspondre à la valeur de gris divisée par 255. Pour répondre à ces critères, chaque pixel de l'image sera mappé à sa valeur divisée par 255. C'est à dire que pour chaque pixel, on récupère sa valeur en nuances de gris et on la divise par 255 puis on la stocke dans un tableau d'int. Ce tableau sera ensuite utilisé par le réseau de neurones afin de déterminer le chiffre présent dans cette case. Cette fonction parcours donc une image, et stocke dans une liste les valeurs normalisées des pixels de sorte à ce que le réseau de neuronne puisse les analyser.

Pour mettre tout en commun, on envoie d'abord l'image de la case binarisée à la fonction isempty. Si celle-ci détermine que la case est vide, le réseau de neurones la marque comme vide. Sinon, le chiffre est testé par le réseau de neurones.

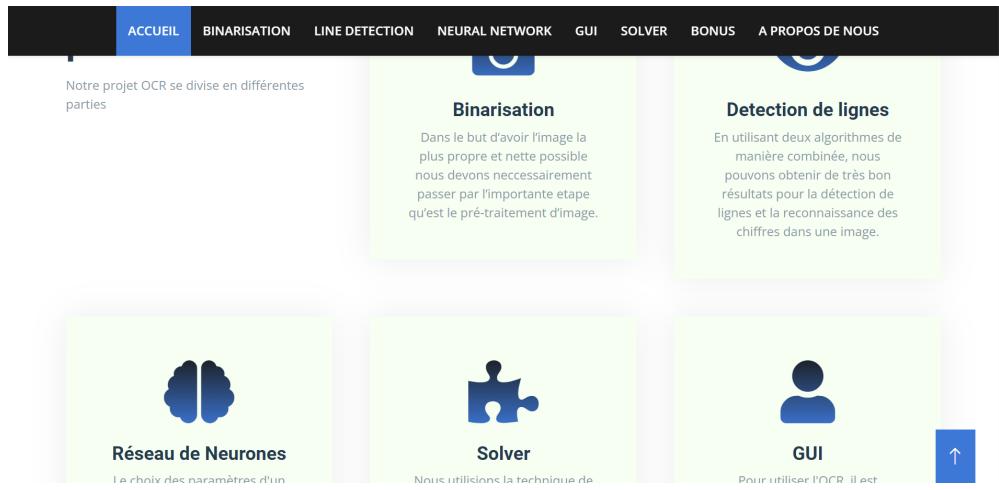
3.7 Site web

Nous avons décidé de coder un site web pour notre projet. Ce dernier contiendrait toutes les informations en relation avec notre OCR, avec des images explicatives, des détails sur son fonctionnement, les étapes que l'on a suivi, une présentation de notre groupe, etc...

Notre petit site web a été bricolé de pièces en pièces par des fichiers html, css et javascript. Il y a pas mal de petites implémentations sympathiques, notamment la sticky navbar, les boutons qui changent de couleur lorsqu'on

les survole, le bouton pour revenir au haut de la page, et les animations. On a même fait en sorte que le color theme soit le même que pour le GUI et l'application mobile.

Les icons utilisés un peu partout dans le site ont été importés de fontawesome, et donnent un peu plus de vie au site. Ce dernier restant simpliste dans sa structure, mais présentant de nombreuses fonctionnalités dynamique permettant une interaction riche avec l'utilisateur.



3.8 Interface utilisateur

3.8.1 Application principale

Pour utiliser l'OCR, il est nécessaire de créer une interface utilisateur. Cette interface doit être simple, intuitive et agréable à regarder. C'est ce que nous avons essayé de faire en créant une maquette. On a utilisé Figma, un outil puissant pour concevoir des maquettes ou créer cette interface. L'interface est en fin de compte un lien entre les différentes parties du projet, elle utilise les fonctions que chaque personne a créées et les regroupe graphiquement pour l'utilisateur.

Afin de créer l'interface utilisateur, nous avons choisi de développer notre

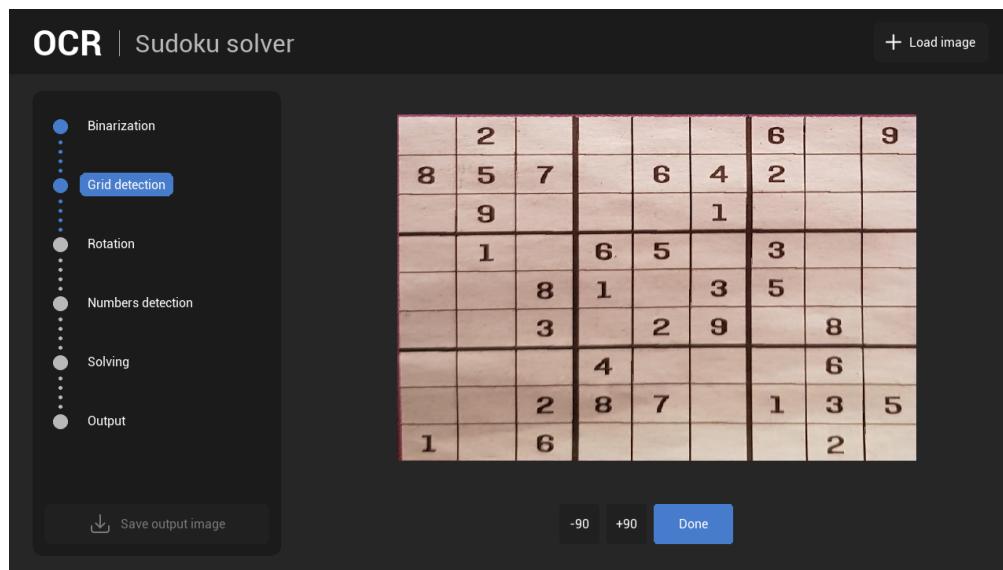
propre librairie pour avoir un maximum de contrôle et de possibilités. Cette librairie est conçue de manière à ne pas dépendre d'une implémentation spécifique. Elle permet de créer des interfaces utilisateur rapidement et facilement en utilisant un modèle dit "Immediate Mode GUI" (IMGUI). Un IMGUI est un type d'interface utilisateur graphique dans lequel les éléments de l'interface sont dessinés à chaque frame ou boucle d'événements, plutôt que de conserver en mémoire une représentation persistante des éléments de l'interface. Cela peut être comparé à une interface utilisateur traditionnelle, dans laquelle les éléments de l'interface sont dessinés une fois et conservés en mémoire. Les IMGUIs permettent une plus grande flexibilité et une rapidité de modification accrue.

Ici, nous utilisons principalement SDL. SDL, ou Simple DirectMedia Layer, est une bibliothèque logicielle libre qui permet de gérer les entrées/sorties audio et vidéo, ainsi que les fenêtres et les événements d'une application en temps réel. SDL est souvent utilisée pour créer des jeux et des applications multimédia en général. Elle est écrite en C ce qui la rend parfaite pour ce projet.

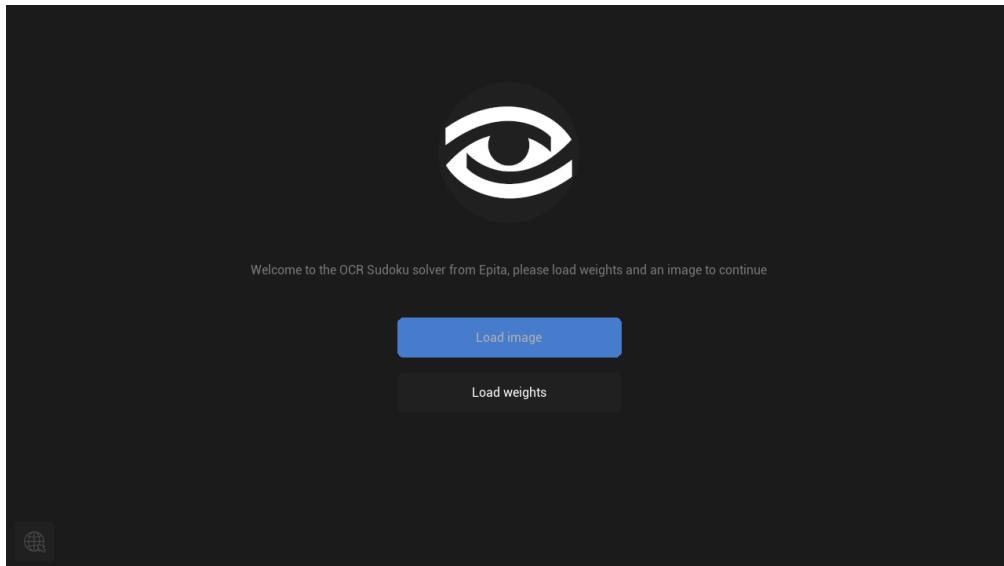
La première difficulté que nous avons rencontrée est le manque de bibliothèques pré-installées sur les machines de l'école, ce qui nous a obligé à utiliser la première version de SDL pour l'implémentation backend. Bien que ce ne soit pas un problème majeur, il serait intéressant de créer des implémentations pour d'autres backend, comme SDL 2 ou GLFW, et d'autres moteurs de rendu, comme OpenGL, afin de pouvoir utiliser des outils plus récents et plus performants. Cela pourrait être un axe d'amélioration pour le projet. SDL version 1 est assez vieux et le support de police d'écriture a donc dû être développé d'une façon non agnostique du backend.

Bien que nous ayons rencontré des difficultés en utilisant cette librairie, elle

nous a permis de faire beaucoup de choses intéressantes, comme un thème unique et moderne, ainsi que des animations pour améliorer l'expérience. Nous croyons que l'interface est très importante pour l'interaction avec l'utilisateur, c'est pourquoi nous avons créé quelque chose de simple et intuitif, tout en restant fonctionnel. Nous sommes satisfaits du résultat final et espérons que les utilisateurs apprécieront notre travail. Chaque animation est rendue fluide en utilisant le principe de l'interpolation linéaire. L'interpolation linéaire est une technique utilisée pour estimer la valeur d'une variable en un point donné en utilisant les valeurs connues de cette variable en deux points différents. Cette technique consiste à créer une ligne droite entre les deux points connus et à utiliser cette ligne pour prédire la valeur de la variable en un point donné sur cette ligne. La maquette a joué un rôle important dans notre processus de développement en nous donnant un objectif à atteindre et en nous permettant de tester différentes approches pour y parvenir. Grâce à la maquette, nous avons pu vérifier que notre interface ressemble le plus possible à ce que nous avions imaginé au départ. Cela nous a aidé à rester concentré et à éviter de nous écarter de notre vision initiale. La maquette a donc été un outil précieux pour nous guider dans notre travail et nous aider à atteindre notre objectif final.



L'interface d'accueil est l'endroit où vous pouvez commencer à travailler avec votre réseau de neurones. Elle vous permet de charger des images pour et de charger des poids pré-entraînés pour initialiser le réseau de neurones. Cela vous permet de démarrer rapidement et facilement votre travail avec le réseau de neurones.

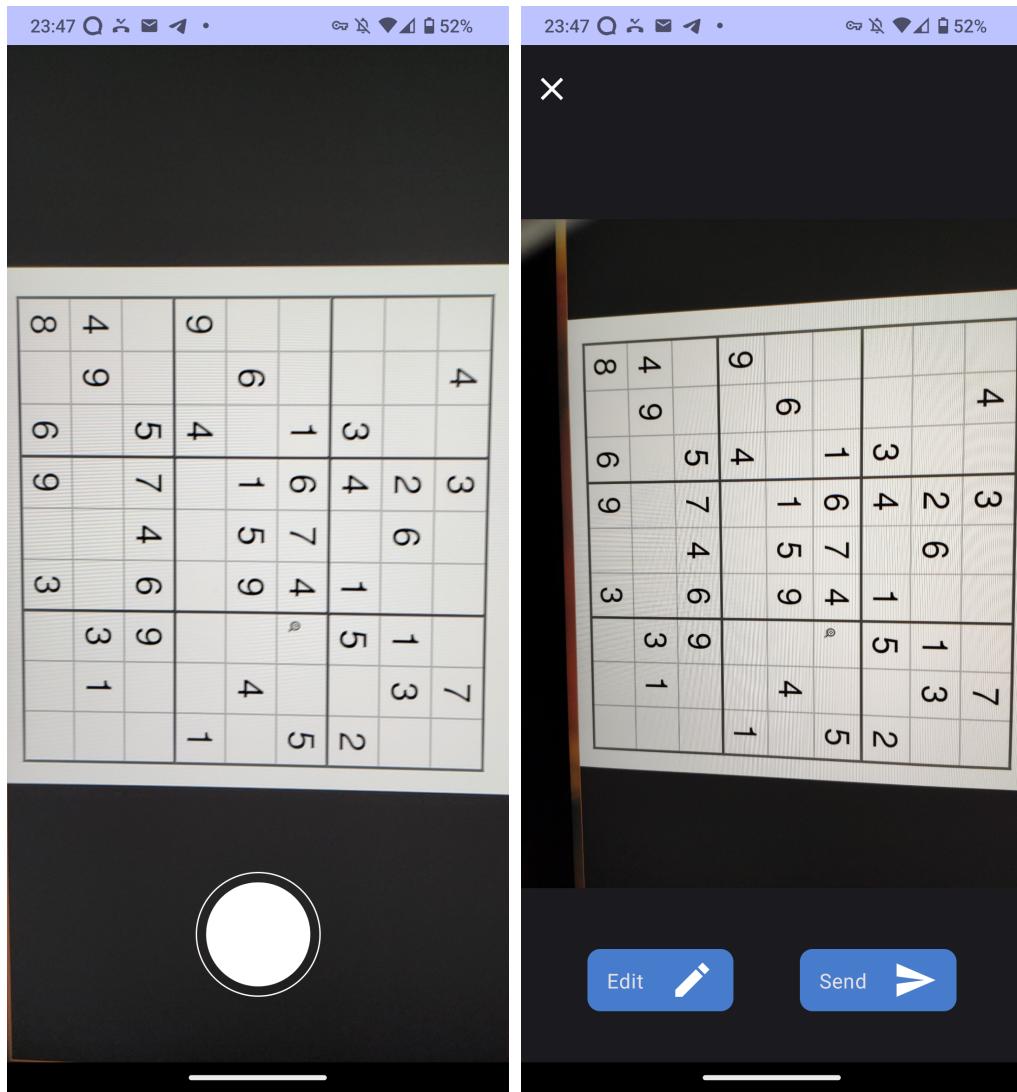


3.8.2 Application mobile

Le développement de cette application Android a été effectué en utilisant le language Kotlin et le toolkit jetpack compose pour simplifier le processus. Cela a permis de créer une application fluide et facile à utiliser, qui offre des fonctionnalités utiles pour les utilisateurs. Grâce à l'utilisation de la caméra intégrée dans les appareils Android, les utilisateurs peuvent facilement prendre des photos et les traiter directement depuis l'application. Cette application a été conçue pour répondre aux besoins des utilisateurs qui cherchent à résoudre directement un sudoku sans passer par un ordinateur.

La connexion entre l'application et le serveur se fait via un serveur socket écrit en C et un client en Java sur l'application Android. Le traitement de l'image se fait côté serveur et les résultats sont renvoyés sur le téléphone. Cela peut être pratique pour prendre une photo et résoudre instantanément le sudoku. Pour

améliorer l'application, il serait intéressant de faire le traitement entièrement en local, mais cela impliquerait de porter le réseau de neurones et le reste du code dans un autre langage ou de l'exécuter de manière native. Nous pensons cependant que cela reste un bonus intéressant pour les utilisateurs.



Nous pouvons voir à droite l'interface pour prendre une photo ainsi qu'à gauche de quoi éditer ou envoyer la photo au serveur. La photo reçue du serveur s'affichera ensuite à la place de la photo actuelle.

3.9 Manipulation de l'image

3.9.1 Détection de la grille

Afin de détecter la grille pour pouvoir résoudre le sudoku, Nous avons changé d'implémentation par rapport à celle utilisée lors de la première soutenance. En effet, lors de la première soutenance nous dépections les lignes sur la grille afin de déterminer l'orientation de la grille. Une fois cette orientation déterminée, nous pouvions faire tourner la grille. Ensuite il nous fallait jouer avec le nombre de lignes, et leurs positions pour déterminer l'emplacement de la grille. Une fois la grille détectée, on pouvait découper chacune des cases. Maintenant, nous avons complètement changé d'approche. Nous détectons toujours les lignes sur l'image. Mais une fois ces dernières détectées, nous déterminons les points d'intersection entre celles ci, pour déterminer quel type de coin de grille cette intersection peut être, si c'est un coin de grille ou bien on confirme que c'est un morceau d'arête. Une fois que nous avons déterminer quel type de coin est-ce :

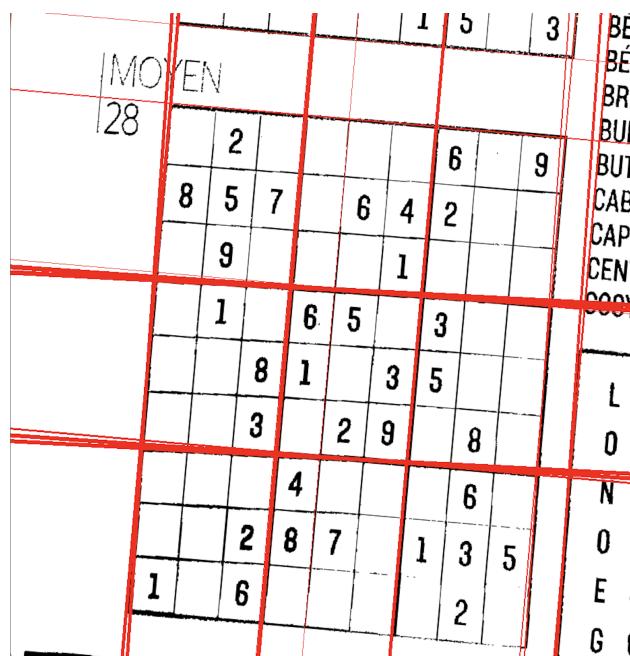
- haut gauche
- haut droit
- bas gauche
- bas droit

Nous avons du déterminer une centaines de cas différents dépendants de l'orientation des différentes lignes. En effet, pour détecter les lignes nous étudions la position des lignes par rapport aux angles. Cependant il y a de nombreux cas de figures possibles. Étant donné que nous travaillons avec des droites, les points d'intersection peuvent se retrouver à plusieurs endroits : sur une arête, sur un coin de la grille, entre deux case sur une arrête ou au milieu de la grille entre 4 cases. Nous avons donc créer plus de 200 cas différents en fonction d'où la présence de pixels de couleur noir était la plus importante.

Une fois cela fait nous pouvions, grâce aux angles des droites qui se coupent en ce point déterminer une position possible du point. Par exemple si on a des pixels noirs de part et d'autre du point avec une fréquence à peu près uniforme, nous pouvons prévoir que ce point est une arête. Par contre, si nous avons un forte concentration de pixels de couleur noirs en haut, et à gauche, mais pas des autres cotés, nous pouvons prévoir que ce point doit possiblement être un coin. Nous pouvons alors l'ajouter à une liste que nous filtrerons. Pour filtrer les points, plusieurs cas de figures s'offre à nous en fonction du nombre de points dont on est sûr. En effet, si seulement un point satisfait les pré-requis d'un coin bas gauche, on est sur que cet angle sera le coin bas gauche. On va ensuite parcourir les point haut gauche et bas droit et vérifier ceux qui ont une droite d'intersection en commun avec ce point. En effet, si ces deux points ont une droit d'intersection en commun, ils peuvent être deux angles de la grille. Si nous avons deux angles connus, ceci est bien plus simple, il nous suffit de parcourir ensemble les points opposés à ceux-ci. Si nous avons par exemple les deux points gauches, nous parcourons simultanément les deux points de droite. Dès que ces deux points satisfont le fait qu'ils aient des droites en commun, on peut déterminer les 4 points de la grille et passer à la suite. Si nous avons 3 points en commun, il nous suffit juste de prendre les deux points opposés, et de regarder leurs droites qui ne sont pas en commun avec le troisième point, puis de parcourir la liste des points satisfaisant les critères du point manquant et de récupérer les point qui passent par se droites. Nous faisons des calculs pour trouver parmi tous les coins notamment les faux positifs comme les coins d'autres grilles légèrement présentes sur l'image. Une fois les quatre coins de la grille trouvés, nous appliquons une modification de perspective afin de redresser la grille dans le cas où elle était orientée de travers. Une fois cela fait, nous pouvons récupérer les 81 cases du sudoku.

3.9.2 Détection des lignes

Pour détecter les lignes, nous avons utilisé la méthode du transformé de Hough : on exprime les coordonnées d'un point en coordonnées sphériques. on peut ensuite déterminer toutes les lignes qui passent par ce point. À l'aide d'un histogramme on peut déterminer quelles lignes sont les plus présentes et donc celle qui devraient représenter la grille. Notre première implémentation de la détection de lignes ne fut pas concluante. Les lignes détectées n'étaient pas tournées dans le bon sens. Une fois ce problème résolu, nous avons pu afficher les lignes verticales mais nous avions toujours un souci sur les lignes horizontales. Nous avons réussi à régler le souci et avons pu afficher correctement les lignes sur l'image :



La détection de lignes n'a pas beaucoup changé quant à son avancée à la première soutenance, en effet, le procédé utilisé fonctionne bien et répond à nos besoins. Cependant nous l'avons adapté à notre nouvelle implémentation de la détection de la grille.

3.9.3 Rotation automatique

Lors de la première soutenance, nous avions pu calculer le delta moyen de différence d'angle avec l'angle 90° . Nous avons ainsi pu effectuer une rotation sur l'image de l'angle trouvé et ainsi la remettre droite. Nous avions d'abord voulu effectuer une rotation sur les lignes que nous avions trouvées avec la première détection de lignes, mais le résultat n'était vraiment pas concluant. Pour pallier ceci, nous effectuons deux passes de détection de lignes sur l'image : une première pour détecter l'orientation, puis une seconde pour détecter, pour de vrai, les lignes sur l'image. Cette partie a été abandonnée au profit d'un changement de perspective qui nous permet de recadrer l'image autour de la grille.

3.9.4 Recadrage automatique

Nous utilisons un changement de base afin de recadrer l'image finale autour de la grille. Pour ce faire, nous utilisons une matrice de passage qui nous permet à partir de 3 points, qui déterminent les nouveaux angles, obtenir une image centrée autour de ces trois points.

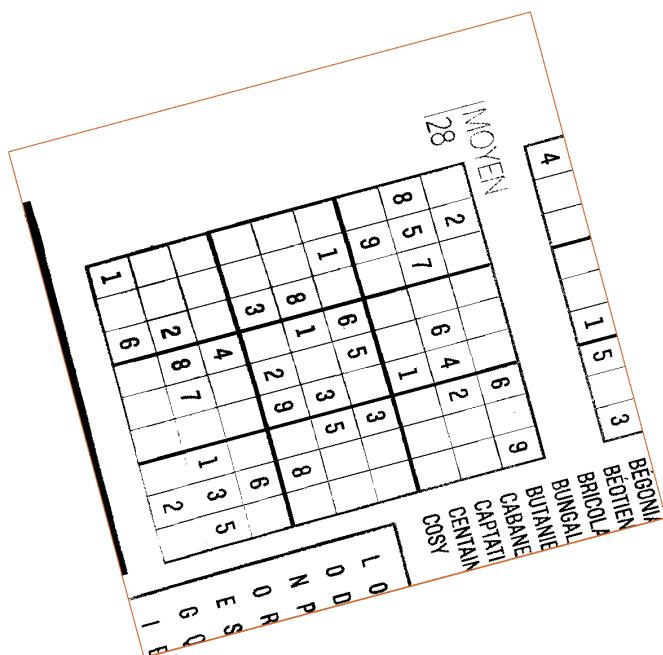


Image de départ

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

Image après recadrage automatique autour de la grille

Ceci nous permet d'avoir une image qui est droite et facile à lire pour le réseau de neurones puisqu'il a été entraîné sur des images de forme carrées.

3.9.5 Identification des cases

Pour identifier les cases et les séparer, nous déterminions les points d’intersection entre les lignes verticales et horizontales, puis nous calculions la distance moyenne en x et en y de ces points pour éliminer les points en trop et ne garder que les essentiels. Mais suite à notre changement de méthode pour détecter la grille. Nous avons ensuite essayé d’utiliser l’algorithme de détection des coins d’Harris, mais malgré de nombreux essais d’implémentation différentes, nous n’avons pas réussi à avoir des résultats concluants. Nous avons donc à nouveau changer d’implémentation. Nous parcourons alors les différents points d’intersection et vérifions s’ils sont des possibles coins ou non. Étant donné que grâce au recadrage automatique, nous n’avons plus besoin de déterminer les points d’intersections et pouvons découper l’image en 81 carrés de taille égale. Pour vérifier si l’image est vide, nous utilisons une version binarisée de l’image pour ne traiter que des pixels de couleurs blancs ou noirs. Mais le réseau de neurones avait besoin d’image en nuances de gris pour avoir plus de précision. Nous avons alors du changer notre méthode.

3.9.6 Implémentation finale

Suite aux nombreux rebondissements que nous avons eu, il nous a fallu changer de méthode à plusieurs reprises. Nous faisons donc une détection de ligne, avec l’implémentation expliquée ci-dessus, sur une image binarisée. Nous allons ensuite, sur cette même image binarisée, détecter les angles possibles de la grille. Une fois les angles détectés, il nous faut les trier pour déterminer lesquels sont des angles corrects, et lesquels n’en sont pas. Une fois que ces angles sont déterminés, nous appliquons le changement de base, à la fois sur l’image binarisée, mais aussi sur l’image en nuances de gris correspondant à l’image binarisée. Nous avons alors 2 grilles de sudoku, une en noir et blanc, et une en nuances de gris. Nous allons ensuite découper ces grilles en 81 carrés chacun. Nous passons ensuite les petits carrés dans une fonction qui va permettre

d'éliminer une grosse partie des bords de couleur noir restés sur les bords des carrés découpés. Pour ce faire, nous repérons cet espace sur les carrés de l'image binarisée puis nous reportons ce décalage sur l'image en nuances de gris. C'est cette dernière qui sera envoyée au réseau de neurones.

Conclusion

En conclusion, ce projet a été très bénéfique pour nous en tant que groupe. Nous avons acquis de nouvelles compétences et connaissances, et avons développé de nouvelles approches pour résoudre des problèmes complexes. Cependant, ce projet a aussi été extrêmement exigeant en termes de temps et d'énergie, ce qui a parfois affecté notre vie personnelle et nos heures de sommeil.

Malgré ces difficultés, nous avons su rester unis en tant que groupe et avons travaillé ensemble de manière efficace. Chacun a apporté sa contribution, en aidant les autres à résoudre les problèmes rencontrés et en partageant ses connaissances et son expertise. Grâce à notre travail d'équipe, nous avons réussi à atteindre les objectifs que nous nous étions fixés et sommes fiers de ce que nous avons accompli ensemble.