EPITA École d'ingénieurs en informatique

Projet OCR

Rapport Soutenance Finale

05 Décembre 2021

Grégoire VEST

Alexandre TSANG-HIN-SUN

Gauthier MAUPAS

Alexandre TEIRLYNCK

Table des matières

1 Introduction

Nous voici enfin arrivés à la soutenance finale du projet d'OCR ! Nous étions partis sur de très bonnes bases, avec un programme de XOR qui s'entrainait et fonctionnait parfaitement, un traitement d'image dont il restait quelques bugs mineurs à corriger mais qui fonctionnait tout aussi parfaitement, et un solveur de sudoku adéquat pour aller avec.

Pour cette soutenance, il nous restait donc à faire : un OCR fonctionnel, une interface graphique ainsi que les quelques parties concernant les images qu'il restait (comme la reconstruction de l'image ou le zoom des cases). Nous nous sommes réparti les tâches, mais cette répartition a rapidement changé et tout le monde s'est mis à un peu tout faire : ce qui semble logique puisque chaque partie en engendre ou est engendrée par une autre.

Les parties ont été donnée de cette manière : Alexandre Ts. a dû terminer de bien gérer tout ce qui concerne les images, Gauthier a construit l'interface graphique, et Alexandre Te. et Grégoire devaient continuer le réseau de neurone pour qu'il puisse reconnaitre les chiffres de 1 à 9.

Le réseau de neurone a été très dur a finaliser, nous avons créé plusieurs réseaux différents se basant sur des principes quelques peu différents, mais aucun ne voulait fonctionner correctement. Nous avons donc dû passer énormément de temps à chercher, chacun de notre côté ou tous ensemble, encore et encore, pour réussir à trouver un réseau qui fonctionne. Cela a été vraiment dur de terminer le projet, nous pouvons compter environ 6 réseaux différents créés, mais nous sommes désormais contents de voir l'aboutissement du projet car celui-ci est vraiment beau et intéressant à voir !

Pour le plan de cette soutenance, chaque personne va présenter son travail (donc il y aura la présentation des différents réseaux de neurones créés) depuis le début du projet, sans vraiment parler du XOR puisque ce n'est pas vraiment intéressant pour ce rapport. Nous allons évidemment présenter les problèmes que nous avons rencontrés et les aboutissements trouvés. Nous allons aussi présenter le ressenti de chaque personne sur le projet.

2 Prétraitement de l'image

2.1 Chargement de l'image

La reconnaissance optique de caractère (OCR) doit commencer par le chargement d'une image de type bitmap afin de pouvoir l'exploiter et aussi l'afficher. La bibliothèque SDL2 nous permet d'utiliser le type " SDL_Surface* " qui est un pointeur vers l'image sous forme de matrice longueur*largeur.

2.2 Suppression des couleurs

La suppression des couleurs de l'image s'effectue en deux étapes : le passage en niveaux de gris, puis la binarisation (passage en noir et blanc).

2.2.1 Niveaux de gris

Une image contient des pixels. Chaque pixel est la combinaison de 3 couleurs : le rouge, le vert et le bleu. Chaque couleur forme un octet donc la valeur décimale est comprise entre 0 et 255. Pour appliquer un niveau de gris à l'image, la méthode la plus évidente est de calculer la luminance, c'est-à-dire la somme des couleurs divisée par 3, puis de le réinjecter à chaque couleur. Néanmoins, la luminance telle quelle ne permet pas toujours de distinguer le bon niveau de gris. Ainsi, on utilise une formule mathématique beaucoup plus appropriée : $luminance = 0.3 * rouge + 0.59 * vert + 0.11 * bleu$

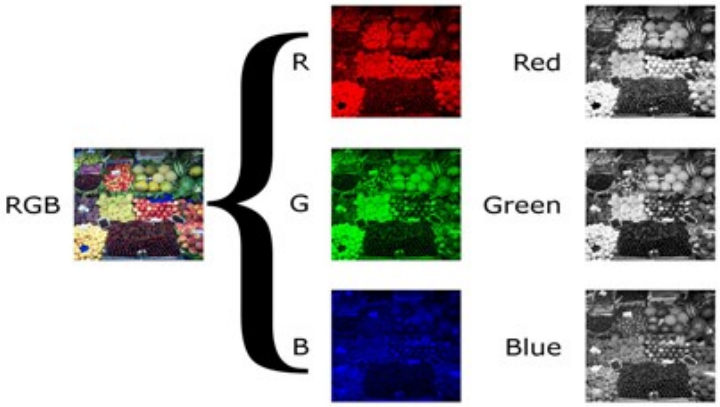


Image originale convertie en niveau de gris

2.2.2 Binarisation (Noir et blanc)

Il existe un nombre conséquent d'algorithmes de binarisation. Mais nous avons essayé d'implémenter ceux qui nous ont semblé les plus efficaces et optimisés. Par exemple, la binarisation de Otsu qui parcourt l'image et établit un histogramme, dans lequel il va chercher un seuil. Celui-ci fera la distinction claire entre l'arrière- plan de l'image et son premier plan. Ce dernier a donc pour but de souligner davantage les objets exposés en premier plan. Néanmoins Otsu ne donne pas toujours de très bons résultats. On fait donc recours à une autre technique qui est un seuil adaptable à chaque image. Il donne donc généralement de meilleurs résultats que Otsu mais est plus coûteux en compilation. L'algorithme parcourt des zones de l'image et adapte un seuil à chaque zone.

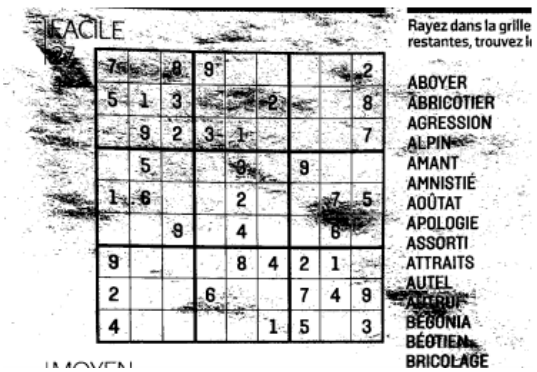
2.2.3 Méthode d'Otsu

La méthode d'Otsu est sûrement la plus intuitive. A partir d'un seuil obtenu par l'algorithme d'Otsu, si la luminance est inférieure au seuil, le pixel sera blanc sinon il sera noir. Afin de déterminer ce seuil, l'algorithme d'Otsu va utiliser un histogramme balayant tous les niveaux d'intensité (de 0 à 255). Ensuite, on parcourt tous les seuils possibles et on effectue des calculs afin de déterminer une variance pour chaque classe d'intensité. La variance interclasse est donnée par :

$$\sigma_b^2(t) = w_1(t)w_2(t)[\mu_1(t) - \mu_2(t)]^2$$

Le seuil à appliquer correspond au maximum des $\sigma 2b(t)$ L'inconvénient de cette méthode est qu'elle utilise un seuillage global. Ainsi, elle n'est pas adaptée pour les images contenant du bruit.

Pour ces dernières, on préférera utiliser une méthode de seuillage local.



Exemple d'Otsu sur cette grille de Sudoku

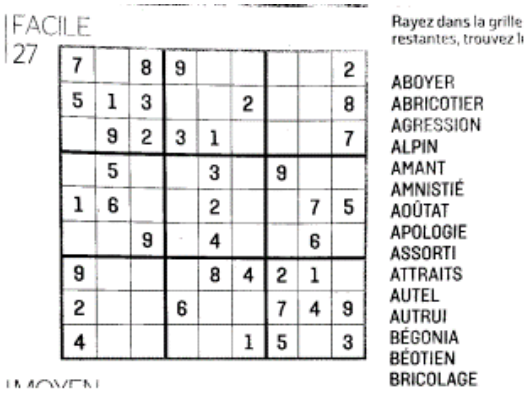
2.2.4 Méthode de l'Adaptive Thresholding

Si l'image utilisée en entrée du programme OCR contient du bruit, par exemple, une tâche de café, l'algorithme d'Otsu montrera ses limites. Après plusieurs heures de recherches, et plusieurs méthodes testées, nous obtenons de bons résultats avec l'Adaptative Thresholding (une adaptation de la méthode de Wellner). La méthode est démontrée sur ce lien : source

Cette méthode utilise une matrice d'image intégrale définie par cette formule :

$$I(x,y) = f(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1)$$

où $I(x,y)$ est la somme des termes $f(x,y)$ à gauche et au dessus du pixel (x,y) Ensuite, on compare le pixel (x,y) à la moyenne des pixels qui l'entourent. Si la valeur du pixel est à t% inférieur à la moyenne, il deviendra noir, sinon il deviendra blanc. La valeur de t est importante, si elle est trop faible, la binarisation laissera du bruit sous forme de tâches noires, et si elle est trop élevée, des pixels pourraient être blanchis par erreur.



Exemple de résultat

2.3 Suppression des bruits

Le bruit d'une image est le nom donné à la présence d'informations parasites qui viennent s'ajouter de façon aléatoire aux détails d'une image numérique. La principale conséquence de cela est la perte de netteté de l'image dû à l'apparition de pixel parasite nommé artefact. Comme tout logiciel permettant un traitement d'image, nous nous sommes rendu compte de la nécessité d'utiliser des filtres. Nous avons, dans un premier temps, concentré nos efforts sur un filtre permettant d'éliminer les pixels parasites (créés par la binarisation et la rotation) puis nous avons également créé un filtre de contraste pour contrebalancer le flou créé par le filtre précédent. Ainsi après avoir effectué des recherches nous avons trouvé des méthodes différentes pour éliminer le bruit : le filtre médian, le gaussien, le sobel, la méthode d'Otsu et le treshold. Bien évidemment ces différents filtres ont pour but d'éliminer les artefacts. Nous avons donc décidé d'implémenter ces types de filtres pour pouvoir choisir le plus adapté.

2.3.1 Filtre Gaussien :

La première méthode, consiste à appliquer une matrice de convolution $3 * 3$, sur chaque pixel de l'image. Chaque composante de couleur du pixel est modifiée en fonction du filtre utilisé et des pixels environnants. Cette méthode se base sur le fait que les pixels d'une image sont en interactions les uns avec les autres. On applique donc la matrice sur le pixel, en sommant les produits des composantes colorées des pixels voisins avec la valeur de la matrice correspondante. On divise ensuite le résultat par la somme des valeurs de la matrice. Voici la matrice de convolution généralement utilisée pour ce filtre :

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

2.3.2 Filtre Médian :

Cette méthode consiste à appliquer une matrice de convolution 3 * 3, sur chaque pixel de l'image, comme le précédent. A la différence qu'on utilise la matrice suivante :

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

On remplace ensuite la valeur du pixel par la valeur médiane des pixels voisins, après application du masque. Le principe de ce modèle est le suivant : On parcourt l'image pixel par pixel, en appliquant le masque sur chaque pixel et sur ses voisins. On applique la formule suivante sur chacun des pixels voisins : luminosité = 0.3 * rouge + 0.59 * vert + 0.11 * bleu. Une fois le masque parcouru, on classe les valeurs du tableau, et on prend la valeur médiane. On remplace ensuite les composantes du pixel par celles du pixel correspondant.

2.3.3 Filtre de Sobel :

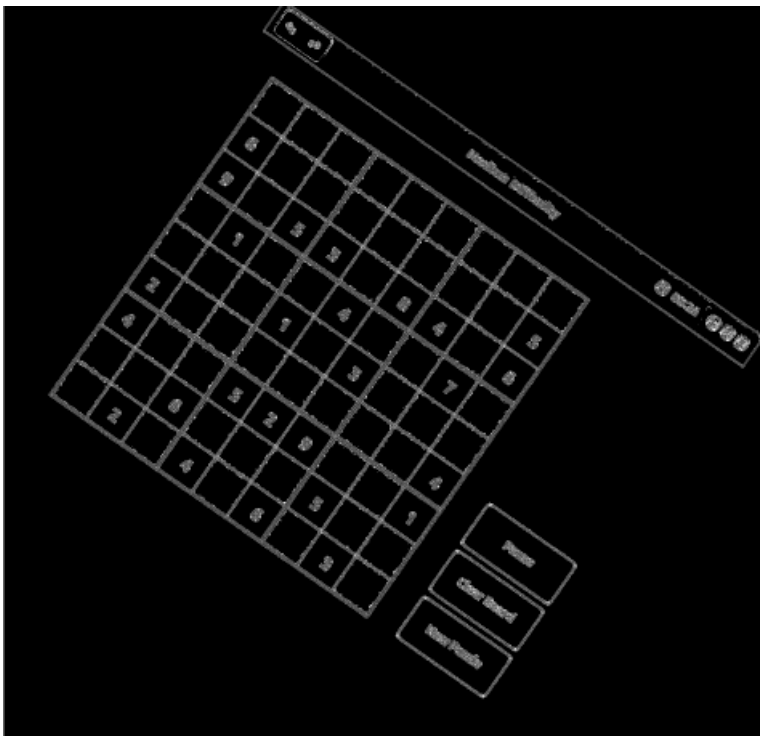
Nous utilisons également le filtre de sobel qui est un calcul du gradient de l'intensité de chaque pixel pour indiquer les variations du clair au sombre pour la détection des contours.

Le filtre de Sobel calcule le gradient de l'intensité pour chaque pixel définie par cette formule :

$$G_{x,y} = \sqrt{Gx^2 + Gy^2}$$

Pour calculer ce gradient on utilise deux matrices pour obtenir les valeurs de Gx et Gy sur une image A :

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \mathbf{A} \quad \text{et} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \mathbf{A}$$



Démonstration du Sobel

2.4 Segmentation et rotation

2.4.1 Préparation de la segmentation

Pour cette partie, le Sobel ne suffit point. Il faut donc ajouter un autre type de filtre convolutionnel comme par exemple le Canny suivi d'un Hysteris. Cela va nous permettre d'obtenir une image dont la grille de sudoku sera plus épaisse et donc plus facile pour l'algorithme (HoughTransform) de la détecter.

2.4.2 Détection de lignes et colonnes

Cela représente la partie la plus difficile de la rotation de l'image. Pour déterminer l'angle de la rotation, nous avons utilisé le procédé de la transformée de Hough. C'est une technique qui permet de reconnaître des formes dans le traitement d'images numériques. Ici, nous l'utiliserons pour détecter les lignes et colonnes de notre sudoku.

2.4.3 Détection de l'angle (pour la rotation automatique)

A l'aide de l'algorithme de Hough, nous pouvons néanmoins connaître l'angle thêta qui apparaît le plus souvent sur une image. Nous en déduisons aisément que cette occurrence fréquente d'un angle θ permet de distinguer d'autres angles. Donc nous en connaissons l'angle de la rotation de notre sudoku. Cependant cette méthode n'est pas optimisée.

2.4.4 Rotation de l'image

Le principe simplifié de la rotation est d'application la formule de rotation d'un point de coordonnée (x, y) à chaque pixel pour un angle θ donné. L'angle est donné en degrés, il est converti ensuite en radiant au début de l'algorithme.

$$x\prime = x * cos(\theta) - y * sin(\theta)$$
$$y\prime = x * sin(\theta) + y * cos(\theta)$$

Pour ne pas "écraser" de valeurs en tournant les pixels de l'image originale, il nous faut en créer une copie. Toutefois, en effectuant une rotation de l'image, elle a de fortes chances de sortir en partie du cadre. Afin de garder le visuel de l'image entière tournée, nous devons changer les dimensions de la copie. Pour se faire, on calcule les coordonnées des coins de l'image suite à leurs rotations, et, après plusieurs comparaisons entre les coordonnées (x, y) de chaque coin, on obtient les coordonnées minimum et maximum de la fenêtre souhaitée. En les soustrayant, on connaît la hauteur et la largeur de l'image post-rotation que l'on peut utiliser pour créer une nouvelle surface à la bonne taille.

Finalement, on effectue un parcourt de cette image copie. Pour chaque pixel, on calcule ses coordonnées (x', y') après rotation autour du centre $(x0, y0)$, on vérifie qu'elle sont toujours comprises dans les dimensions, puis on récupère le pixel à cette position sur la surface originale. Si les coordonnées sont en dehors de l'image alors le pixel sera noir.

$$x\prime = (x - x0) * cos(\theta) - (y - y0) * sin(\theta) + x0$$
$$y\prime = (x - x0) * sin(\theta) + (y - y0) * cos(\theta) + y0$$

Le pixel est placé, cette fois-ci sur la nouvelle image, aux coordonnées (x, y) initiales. Pour finir, cette image est retournée. Cette dernière étant alignée, elle nous donne la possibilité de la segmenter.

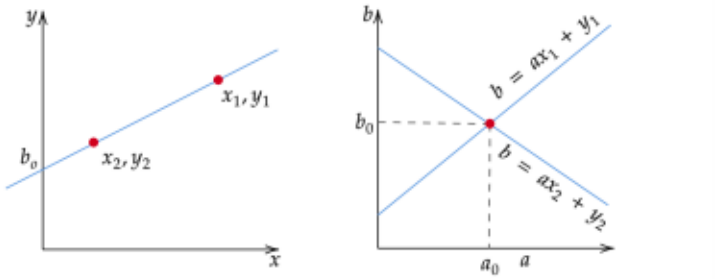
2.5 Segmentation

Nous avons développé un algorithme qui nous segmente notre grille afin de détourer toutes les petites. Son concept, même intrinsèquement mathématique, reste simple à utiliser et à comprendre. Une image est parcourue verticalement et horizontalement. Des marqueurs de couleurs sont mis dessus tout en repérant les occurrences les plus répandues de séquences grille/non-grille. Or comme les cases à segmenter sont des carrés, il est simple de comprendre que toutes les cases carrées sont détectées et segmentées.

Pour pouvoir trouver les droites continues en partant du postulat que l'image si besoin est, a déja était tourné dans le bon sens. On utilise la transformation de Hough.

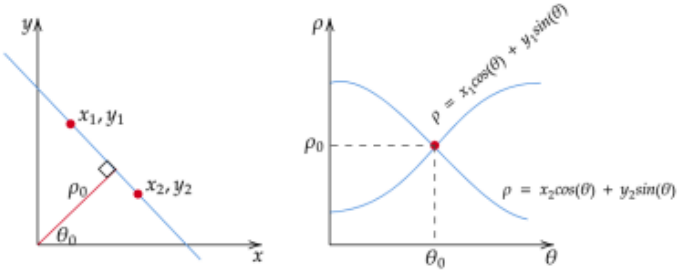
2.5.1 Transformation de Hough

Afin de découper la grille il est nécessaire de détecter et récupérer les coordonnées des lignes présentes, pour cela nous utilisons l'algorithme de Hough. Nous partirons comme image de base, la "edge map" donné par le filtre de Sobel. Pour comprendre cette transformation il faut expliquer ce qu'est l'espace de Hough. C'est un array à deux dimensions qui sur l'axe horizontal représente la pente et l'axe 9 vertical l'intersection d'une droite. Chaque edge line (ligne de point blanc sur la edge map) définit par deux point (x1, y1) et (x2, y2) vas être représenter sur l'espace de Hough par un point de coordonnée horizontal égal à la pente de la droite et de coordonnée vertical égal à b = a x1 + y1 et b = a x2 + y2, le point d'intersection.



Représentation d'une droite (gauche) en un point sur l'espace de Hough (droite) en coordonnées cartésiennes

Représenter des droite en coordonnées cartésiennes a ses limites surtout quand les lignes sont vertical, il est donc nécessaire de passer sur un repère polaire. Ainsi chaque point de l'espace de Hough auras comme coordonnées (,) avec : = x cos() + y sin()



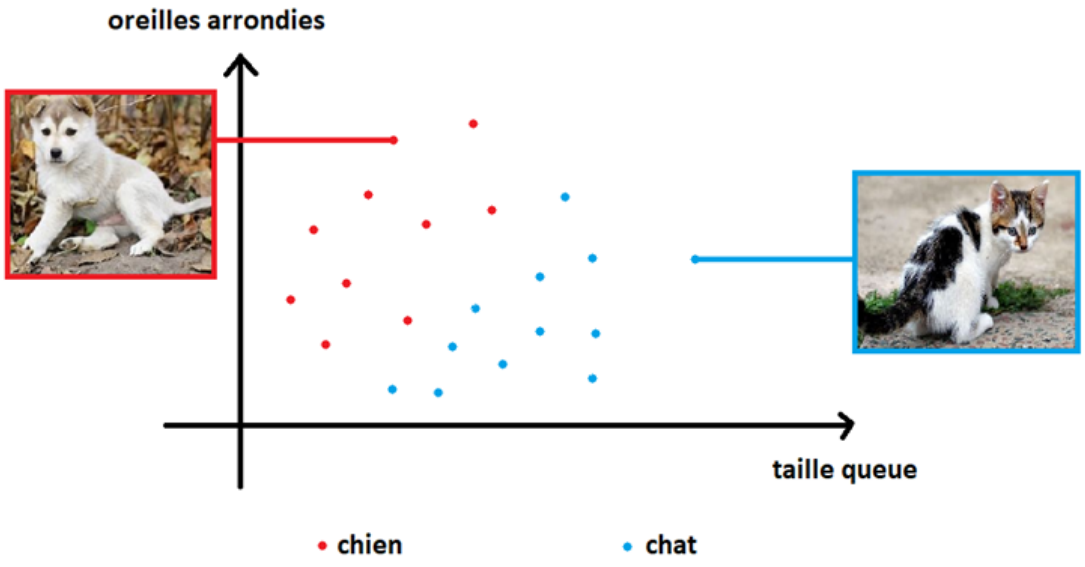
Représentation d'une droite (gauche) en un point sur l'espace de Hough (droite) en coordonnées polaires

Les edge point sont stockés sur l'espace de Hough par un principe de vote, pour chaque ligne sur la edge map, on incrémente de 1 sa valeur à la position correspondante sur l'espace de Hough. Donc plus un point de l'espace de Hough a une valeur élevé plus il y a de ligne de la edge image qui l'intercepte. Ainsi en prenant tous les points de la edge map ayant une valeur dépassant un certain seuil, on peut approximer la position des lignes de la grille de sudoku.

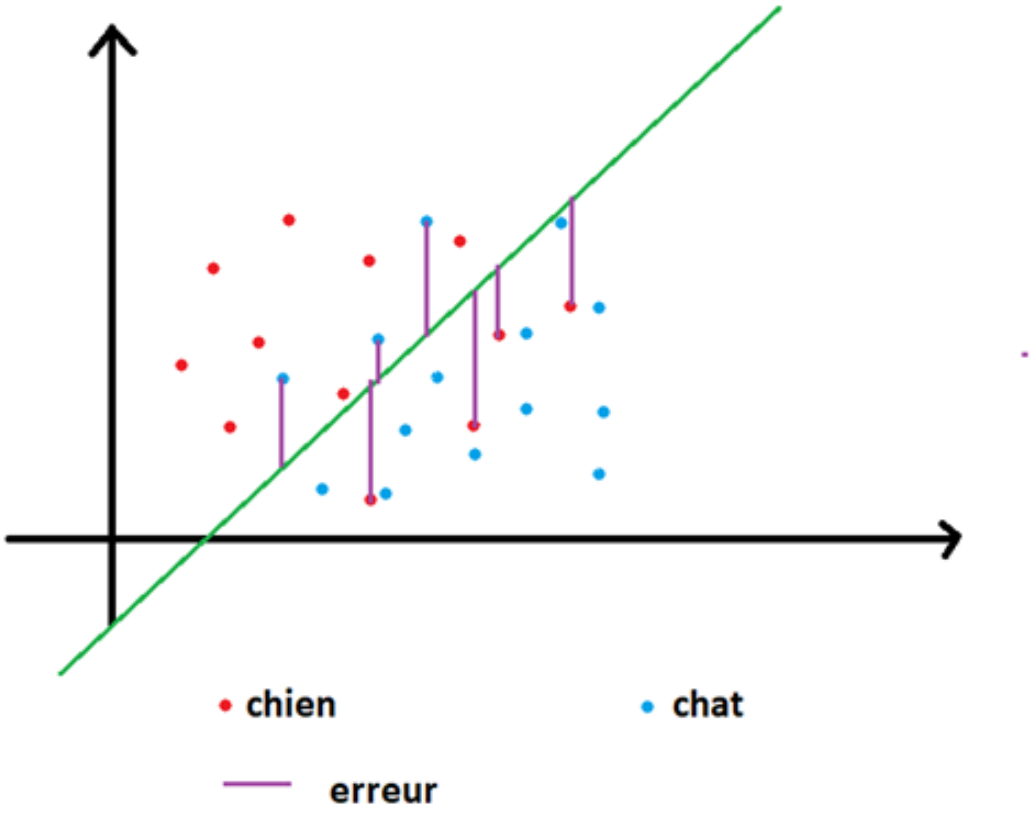
3 OCR

3.1 Théorie

Reconnaitre des neurones revient à un problème de classification. Pour illustrer ce problème, on imagine qu'on souhaite savoir si un dessin représente un chat ou un chien. On imaginera qu'on est capable de calculer la taille de la queue et de connaitre la forme des oreilles. On peut représenter plusieurs images dans un graphique.



La solution de ce problème est une courbe séparant les points représentant les chiens des points représentant les chats. Ainsi pour toute nouvelle image on regardera de quelle coté elle se trouve de la courbe. Ce problème peut être résolu a l'aide d'un perceptron. Un perceptron est une structure X entrées et qui renvoie une sortie. Cette sortie doit être comprise entre 0 et 1 pour pouvoir être interprété comme une probabilité de succès. Pour atteindre cette sortie, le neurone possède des poids (autant que d'entrées) et un biais. Il somme la multiplication entre chaque entrée et son poids. Puis il vient ajouter un biais a la manière d'une fonction affine pour pouvoir créer n'importe quelle équation de droite. Ensuite il active le résultat de son calcul entre 0 et 1. On comprend aisément que l'apprentissage du perceptron réside dans la correction des poids et des biais. Pour les corriger, on calcul l'erreur du perceptron. Encore faut-il savoir qu'on a faux. Pour cela on entraine le perceptron avec des images dont on connait le résultat. L'erreur est la différence entre le résultat et la sortie attendue. On appelle la partie qui calcul la sortie, la propagation avant ou forward propagation. Et la partie qui corrige les poids et les biais, la propagation arrière ou backward propagation. Pour entrainer un neurone il faut répéter plusieurs centaines de fois un cycle propagation avant – propagation arrière.

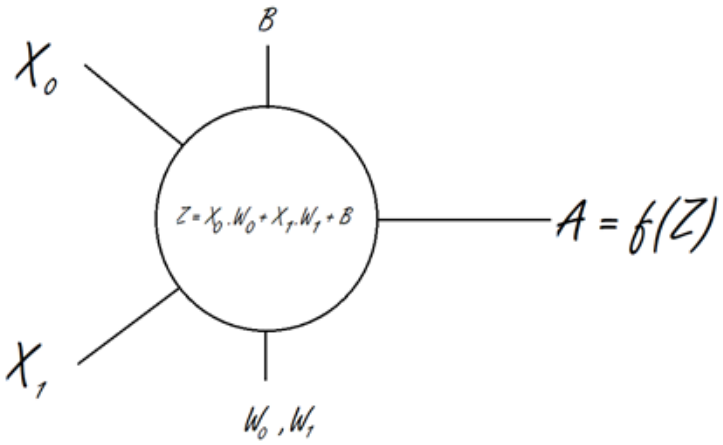


A l'aide de l'erreur et d'un paramètre appelé « pas de l'apprentissage », le perceptron est en mesure d'apprendre de ces erreurs en corrigeant ses poids selon les dérivées partielles de la fonctions erreur. Pour obtenir la correction nécessaire des poids, on dérive par rapport aux poids et par rapport au biais. Puis on corrige les poids selon les relations suivantes.

$$W = W - alpha * dW$$

$$B = B - alpha * dB$$

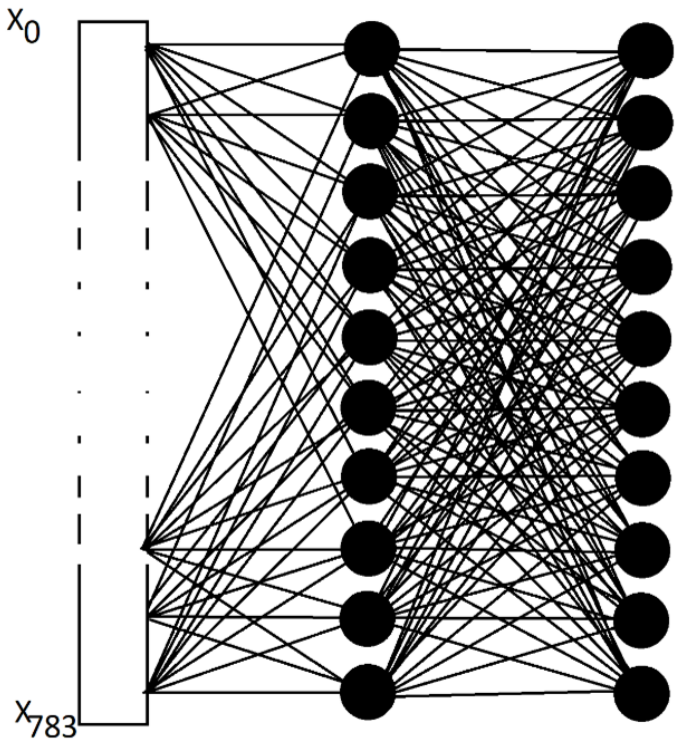
On peut résumer le modèle du perceptron avec le schéma suivant



En généralisant le problème, on se rend compte que si on souhaite reconnaitre un 3 -ème animal, une seule droite ne peut pas suffire. Il faut donc créer un 2 -ème perceptron prenant les mêmes entrées mais qui possède des poids et des biais différents. On a créé une couche de perceptron : notre premier réseau de neurones. La sortie sera la position du neurone le plus actif. Cependant il faut encore ajouter un neurone puisqu'avec on ne peut avoir que 2 sortie différente et rappelons le, on en veut 3.

Par ailleurs plus il y a d'entrée plus il est difficile pour une couche de perceptron d'obtenir un résultat précis. Pour cela nous récupérons les sorties de cette couche, pour les envoyer en tant qu'entrée dans une couche suivante. Cela permet d'augmenter la précision de la sortie finale.

Revenons a notre objectif qui est de créer un réseau de neurones capable de reconnaitre des chiffres compris entre 1 et 9. Il prend en entrée 784 pixel représentant une case du sudoku. Il est composé de 2 couches de perceptron de 10 neurones. Le but de la première couche est de reconnaitre à grossièrement de quel chiffre il s'agit. Cela réduit les possibilités pour la deuxième couche qui déduira le chiffre : par exemple si la première couche indique qu'il peut s'agir d'un 7, 9, 8, la deuxième couche annoncera qu'il s'agira d'un 9 parce que le 9 et le 8 ont une boucle dans la partie supérieure et que le 7 et le 9 n'ont pas de boucle dans la partie inferieur.



La première couche est activée a l'aide de la fonction Relu qui associe tout nombre négatif à 0. Comme il s'agit d'une étape préliminaire nous avons jugé pertinent d'utiliser une fonction nécessitant un cout faible en opération. Les réseaux de neurones plus complexes utilisent la fonction logistique pour projeter leur résultat entre 0 et 1. Pour notre deuxième couche, nous utilisons la fonction softmax. Cette fonction prend en paramètre un vecteur et modifie chaque valeur du vecteur en une probabilité. Cette fonction est intéressante car si on somme toutes les valeurs du vecteur résultat, nous obtenons 1.

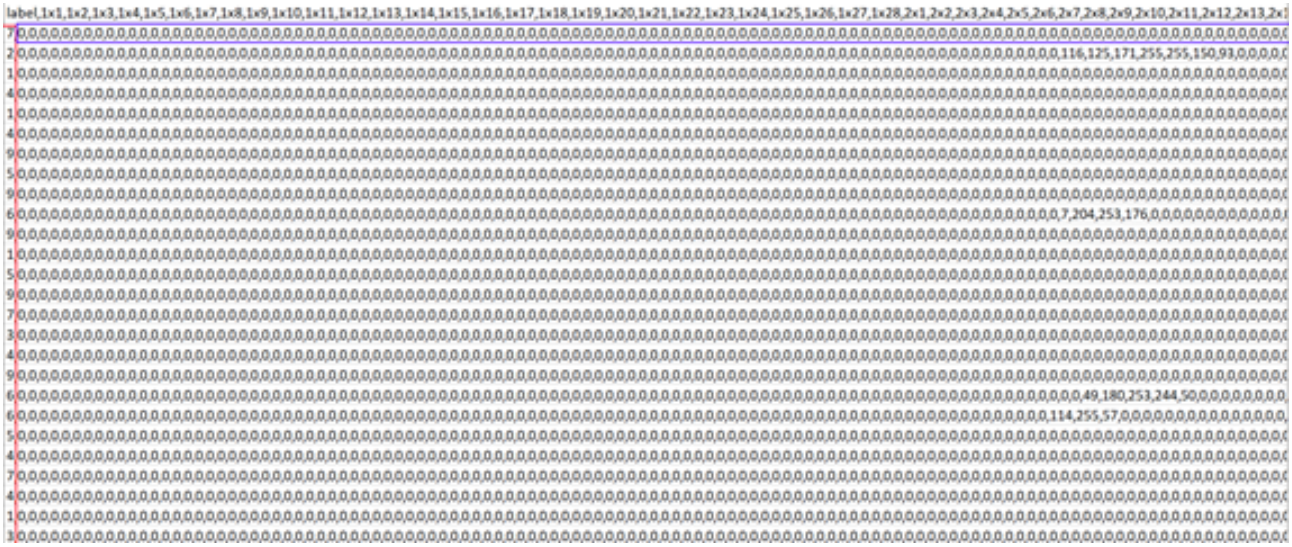
RELU(X) :
si $X < 0$ alors 0
sinon X

$$sigmoid(X) = \frac{1}{1 + e^{-x}}$$

$$softmax(L) = \frac{exp(l_i)}{sum(exp(L))}$$

3.2 Différentes méthodes

Le réseau de neurone a été pour nous la partie qui nous a pris le plus de temps et de stress. En effet chaque réseau que nous construisions ne fonctionnait pas, pourtant chacun se basait sur des façons différentes d'atteindre l'objectif. On peut distinguer 4 catégories de réseaux différents que nous avons implémentés : Partir du code du XOR, contrôler les calculs des neurones, réduire l'exhaustivité du code et repartir d'un OCR implémenté en python comme nous avions pu le faire pour le XOR. Pour toutes les méthodes nous utilisions le MNIST : 10 sorties (les nombres de 1 à 10), et évidemment 784 entrées (images de 28x28 pixels). Nous enlevions la sortie 0, donc cela nous faisait 9 sorties pour 9020 exemples. Au final, on se retrouve avec une matrice de 9020*784 pour entrainer le réseau, ce qui est particulièrement énorme (précisément 7 071 680 cases) par rapport à la matrice 2*4 du XOR. On peut donc prévoir à l'avance les problèmes de mémoire ainsi que des fonctions mal codées mais qui fonctionnaient tout de même.



Matrice du MNIST :

En bleu : les pixels d'un chiffre en image (784)

En rouge : les sorties (9020)

3.2.1 Partir du XOR

La 1e idée était donc de reprendre le code de la 1e soutenance pour agrandir la taille des entrées et des sorties. Le problème qui est apparu était de vouloir agrandir la taille des matrices. En effet, en C il est impossible de créer des tableaux de 7 000 000 de cases, nous avons donc dû trouver une solution alternative : les pointeurs. L'idée est simple : allocation de mémoire pour un pointeur, puis utilisation de celui-ci de la même manière que les tableaux du XOR. En effet, nous avions déjà codé nos fonctions de manière à utiliser des tableaux à une seule dimension, donc c'était parfait pour utiliser les pointeurs.

```
//init params
//data
double *Data = NULL;
double *Y_train = NULL;
double *X_train = NULL;
double maxvalue = 1;

double *X = NULL;
double *Y = NULL;

//weights
double *W1 = NULL;
double *W2 = NULL;

double *dW1 = NULL;
double *dW2 = NULL;

//bias
double *b1 = NULL;
double *b2 = NULL;

//gradiant
double *Z1 = NULL;
double *dZ1 = NULL;
double *dZ1bis = NULL;
double *Z2 = NULL;
double *dZ2 = NULL;
double *A1 = NULL;
double *A2 = NULL;

//cost shape
double *prediction = NULL;

//tempory shape
double *nData = NULL;
double *X_tr = NULL;
double *W2_tr = NULL;
double *A1_tr = NULL;
double *sumE = NULL;
double *one_hot_Y = NULL;
double *one_hot_Y2 = NULL;
```

Aperçu de toutes les matrices utilisées pour le réseau de neurone

On remarque qu'énormément de matrices sont utilisées. Ce système de pointeur a été une solution très efficace pour résoudre notre problème de mémoire. Mais vu la taille des pointeurs, les calculs restaient assez longs et un problème que nous attendions était donc le temps de calcul, qui rend les essais difficiles : il a fallu, pour chaque réseau, optimiser le temps d'entrainement pour tester les codes.

Malheureusement, nous n'avons pas pu finaliser de reconstruire notre réseau de neurone sur la base du XOR de la 1e soutenance car il y a eu beaucoup trop de problèmes dû à la gigantesque taille des matrices. Nous avons donc abandonné cette idée pour essayer une autre façon de faire un réseau.

3.2.2 Essai 2

Suite aux nombreux échecs dus à la manipulation de matrice de tailles conséquente, nous avons tenté de créer un réseau de neurones, en créant un neurone lui-même et en le plaçant dans une liste de neurones représentant une couche. L'objectif était d'éviter a manipuler des tableaux trop grand.

Cette implémentation promettait des tests plus ciblés sur le réseau de neurones dans le but de le debugger. Cependant nous avons vu les limites de la méthode en implémentant la propagation arrière puisqu'elle a besoin des résultats de tout le dataset fourni. Nous nous retrouvions au même problème que l'implémentation du XOR à savoir manipuler des tableaux énormes.

3.2.3 Essai 3

Comprenant que la manipulation de matrice serait obligatoire. On a essayé de refaire un code propre et structuré. Le premier objectif était de réduire considérablement le nombre de paramètre dans les fonctions et de pouvoir discerner parfaitement les opérations effectuées. Pour cela nous avons créé une structure « Network » qui stockerai directement tous les tableaux de poids, biais et sortie ainsi que leur dimension. Devoir fournir a chaque fonction les dimensions des matrices était très laborieux donc nous avons créé également une structure matrice qui stocke un tableau et ces dimen-

sions. Nous pouvions enfin renvoyer une matrice depuis les fonctions sans devoir lui fournir des pointeurs pour le résultat. Les limites de l'implémentation étaient un temps de calcul extravagant : 2 minutes pour 1 cycle. Et une erreur dont l'origine n'aura jamais était trouvé : la fonction softmax ne renvoyait que des NaN.

3.2.4 Partir d'un réseau en Python

Notre dernière solution a été de faire comme à la 1e soutenance : partir d'un code fonctionnel implémenté en python car c'est un langage que nous maitrisons beaucoup mieux. En effet, le code du XOR que nous avions pris pour la 1e soutenance avait une extension qui utilisait le MNIST pour faire un OCR. Après quelques essais, nous avons vite réussi à faire fonctionner le réseau de neurone en python, donc nous avons réessayé de prendre le code python (avec toutes les fonctions des matrices et les principes de base déjà codées pour le XOR) pour le transformer en C. Même s'il était certain que ça nous prendrait beaucoup plus de temps que d'implémenter un réseau directement en C, il était indéniable que nous devions forcément arriver à un résultat positif.

Pour cela, nous avons repris tout le code que nous avions laissé tomber de la partie « Partir du XOR », c'est-à-dire toute la structure, toutes les fonctions sur les matrices ainsi que la méthode d'entrainement. Et, grâce au code python, corriger les erreurs une par une, en s'assurant bien que chaque ligne renvoie le bon résultat.

Premièrement, nous avons remarqué beaucoup d'erreurs de dimensions. En effet, le problème du XOR est que la matrice entrée*sortie est une matrice carrée (2*2), il est donc très facile de confondre les deux (l'entrée et la sortie) et c'est ce que nous avions fait à plusieurs endroits du code.

Deuxièmement, certaines parties du code étaient complètement fausses sans influer pour autant le XOR : par exemple la partie du code sur le on_hot_Y, nous n'arrivions pas à savoir précisement ce que cette partie était censée effectuer, mas vuq eu ça fonctionnait, nous ne nous étions pas posé la question. On s'est rendu compte alors que la matrice était transposée par rapport à ce que l'on recherchait, donc encore une fois : problème de dimension, mais en plus problème de code.

Au final, l'aboutissement de ce réseau de neurone a été très long et difficile, mais vu le nombre de recherches que nous avions faites, ainsi que le nombre de réseaux que nous avons créé, il était à la fois primordial et évident pour nous que nous allions réussir ce réseau !

4 Solveur

4.1 Utilisation

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

Grille de Sudoku

Après avoir reconnu les différents chiffres du sudoku d'entrée il faut le résoudre. Nous avons donc fait un solver de sudoku qui peux résoudre tout type de sudoku de taille 9*9. Notre solveur prend en paramètre un fichier texte. Nous avons donc créé une fonction qui appelle notre solveur avec le path d'un fichier. Le fichier d'entrée possède certaines caractéristiques :

— Le fichier comporte 11 lignes dont 9 lignes de 11 caractères et 2 lignes vides

— Les cases vides de la grille de sudoku sont représentées par des points

— Chaque carrés du sudoku sont séparé par des espaces.

```
..7 .15 .8.
81. .67 ..9
2.. ... ..6

... ..4 ..1
..1 9.. 52.
.8. ... ...

.5. ..8 6.3
.3. 5.. ...
..9 ... ...
```

Format du fichier avec un sudoku

Une fois que le solveur de sudoku a finit de résoudre notre grid d'entrée, il enregistre un nouveau fichier avec le même nom que le fichier rentré en paramètre mais avec l'ajout de l'extension .result afin de pouvoir le distinguer le fichier résolu du fichier non résolu. En suite nous avons la fonction qui permet de reconstruire l'image qui récupèrera le fichier .result afin de reconstruire la grille résolue sur l'image du sudoku de base.

4.2 Fonctionnement

Pour réaliser le solveur de sudoku nous avons utilisé la technique du backtracking (retour sur trace). Cette méthode n'est pas la méthode la plus optimisé pour résoudre un sudoku mais c'est la méthode que nous connaissions.

La méthode de backtracking consiste à tester récursivement si une solution valide peut être trouvée depuis la dernière affectation. S'il n'y a pas de solution alors, la méthode revient en arrière pour essayer un autre chemin de solutions.

Notre solveur possède 3 fonction de check :

— Une fonction qui vérifie que le chiffre que l'on veut insérer dans la case vide n'est pas déjà présent sur la ligne
— Une fonction qui vérifie que le chiffre que l'on ajoute n'est pas déjà présent dans la colonne.
— Une fonction qui vérifie que le chiffre n'est pas présent dans le carré ou ce situe l'emplacement dans lequel on veut insérer un chiffre.

Ces trois fonctions renvoient un booléen. Nous avons ensuite notre fonction principale de backtracking qui parcourt toute la matrice représentant le sudoku, la fonction va donc tester tous les différents chiffres jusqu'à trouver un chemin valide qui remplit totalement la grille. À chaque case la fonction va tester si le chiffre que nous voulons insérer n'est pas présent sur la ligne, la colonne, ou le carré. S'il y a déjà un chiffre sur la case à tester alors on passe à la case suivante.

Le programme du solver prend en paramètre un fichier avec un certain format. Afin de lire le fichier nous avons créé une fonction qui parse notre fichier puis qui en-

registre toutes les valeurs du sudoku dans une matrice de 9*9 en remplaçant les " . " par des 0.

Enfin une fois que la fonction de backtracking a résolu le sudoku et qu'elle a modifié la matrice représentant le sudoku. Une fonction écrit le résultat du sudoku dans un fichier avec le même nom que le fichier de départ mais avec l'extension .result en plus. Le fichier avec les résultats garde le même format que le fichier entré en paramètre.

5 Reconstruction de l'image

5.1 Vérification des cases vides

Les grilles du sudoku sont constituées de cases contenant un chiffre ou de cases vides. Il est important de détecter si la case est vide ou non pour la reconnaissance du chiffre par le réseau de neurones. Pour ce faire nous calculons dans chaque case en niveau de gris l'écart-type de la valeur de tous les pixels contenus dans la case. Si cette valeur est en dessous d'un certain seuil, la case est considérée comme vide et une case blanche est retournée au système.

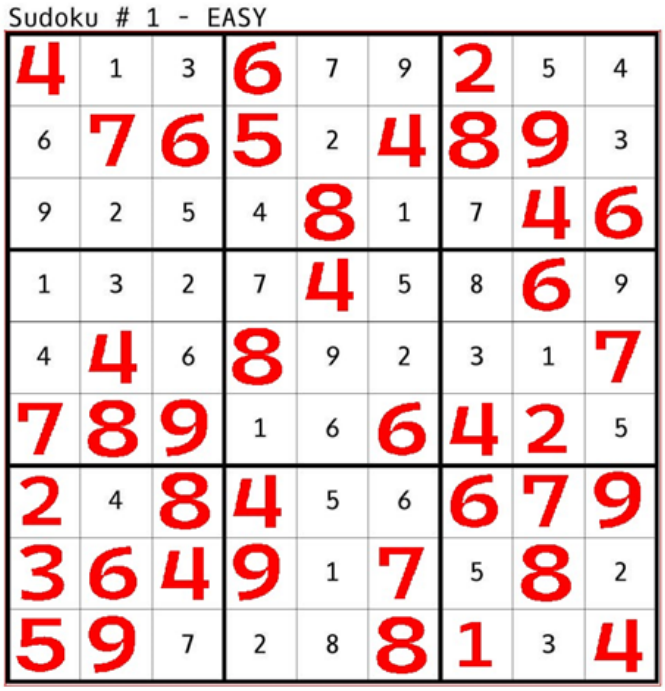
Notre algorithme est simple dans son implémentation :

— Créer un fichier texte avec que des

— Pour chaque case de la segmentation vérifier si c est une case blanche sinon on fait la prédiction du chiffre

5.2 Afficher sur la grille la solution du Solveur

Après avoir lancé le solveur le fichier texte est rempli en entier, ensuite on lance une comparaison entre les 2 fichiers textes et les différences entre des "." et un code ascii qui correspond à un chiffre entre 1 et 9 alors on remplace la case blanche par le chiffre correspondant à la bonne occurrence de la case découpée et sauvegardé dans l'ordre par la segmentation. A la fin tout ce qui est a été sortie en case segmenté on reconstruit par dessus toutes les images segmentées dont celle modifié.



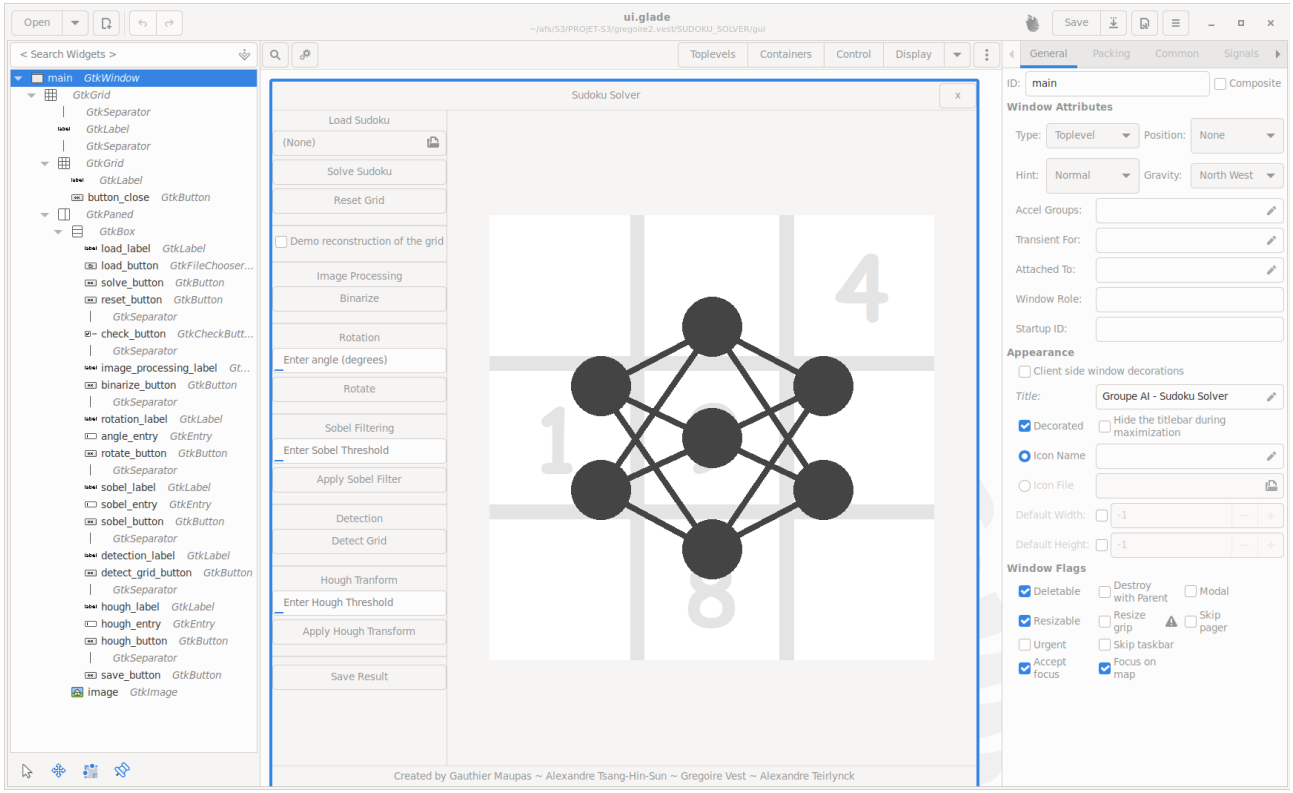
Reconstruction graphique de la solution du Sudoku

6 Interface graphique

6.1 Création

Après avoir réalisé toutes les parties de notre projet, les traitements sur l'image, la reconnaissance de caractères, le solveur et la reconstruction de l'image il nous fallait une interface graphique qui nous permette de regrouper toutes ces actions afin de résoudre le sudoku. Afin de réaliser cette interface graphique nous avons utilisé la version 3 de la librairie GTK c'est la librairie qui nous a été proposé pour le projet.

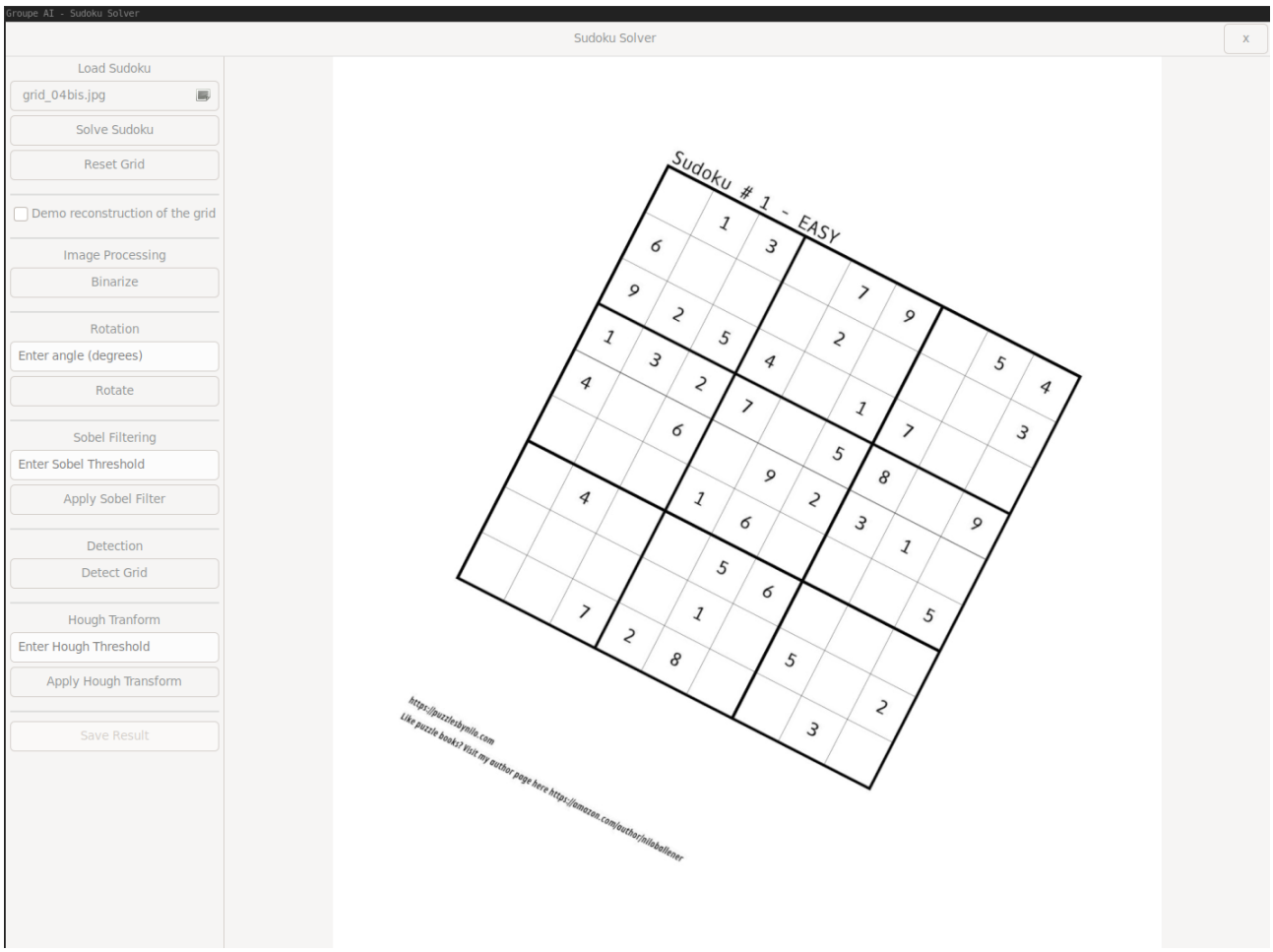
Pour pouvoir styliser notre interface graphique avec des éléments de GTK nous avons utilisé le logiciel Glade. Glade est un logiciel de conception d'interface graphique GTK. Il permet de construire une interface graphique visuellement. Glade enregistre les interfaces graphiques en générant des fichiers XML qui vont être par la suite build par notre fichier principal de l'interface graphique. Et grâce à la librairie GTK nous pouvons relier les boutons créés sur glade a des fonctions, pour cela nous utilisons des callback function.



Logiciel Glade

6.2 Utilisation

Pour notre interface graphique nous avons fait le choix de ne pas mettre qu'un bouton solve pour résoudre notre sudoku. Nous avons décider de mettre d'autres boutons afin de montrer étape pas étape les traitements sur l'image qui peuvent être effectué par notre solveur.



Notre interface graphique avec une grid

Pour lancer notre interface graphique il faut compiler le projet dans le dossier gui/ puis lancer l'interface avec l'exécutable ./ui.

Tout d'abord afin de pouvoir utiliser notre interface graphique il faut charger une image. Une fois qu'une image est chargée sur l'interface elle est affichée grâce à un GtkImage à droite de notre interface. Ensuite nous pouvons utiliser tous les boutons de traitement sauf le bouton save car il faut avoir résolu le sudoku à l'aide du bouton solve avant de pouvoir enregistrer le résultat.

Nous avons différents boutons :

— Le bouton solve qui permet d'effectuer tous les traitements sur l'image et de résoudre notre sudoku.

— Le bouton reset grid qui permet de reset l'image afficher sur l'interface graphique pour revenir à l'original.

— Le bouton Binarize qui lance la binarisation sur l'image charger sur l'interface.

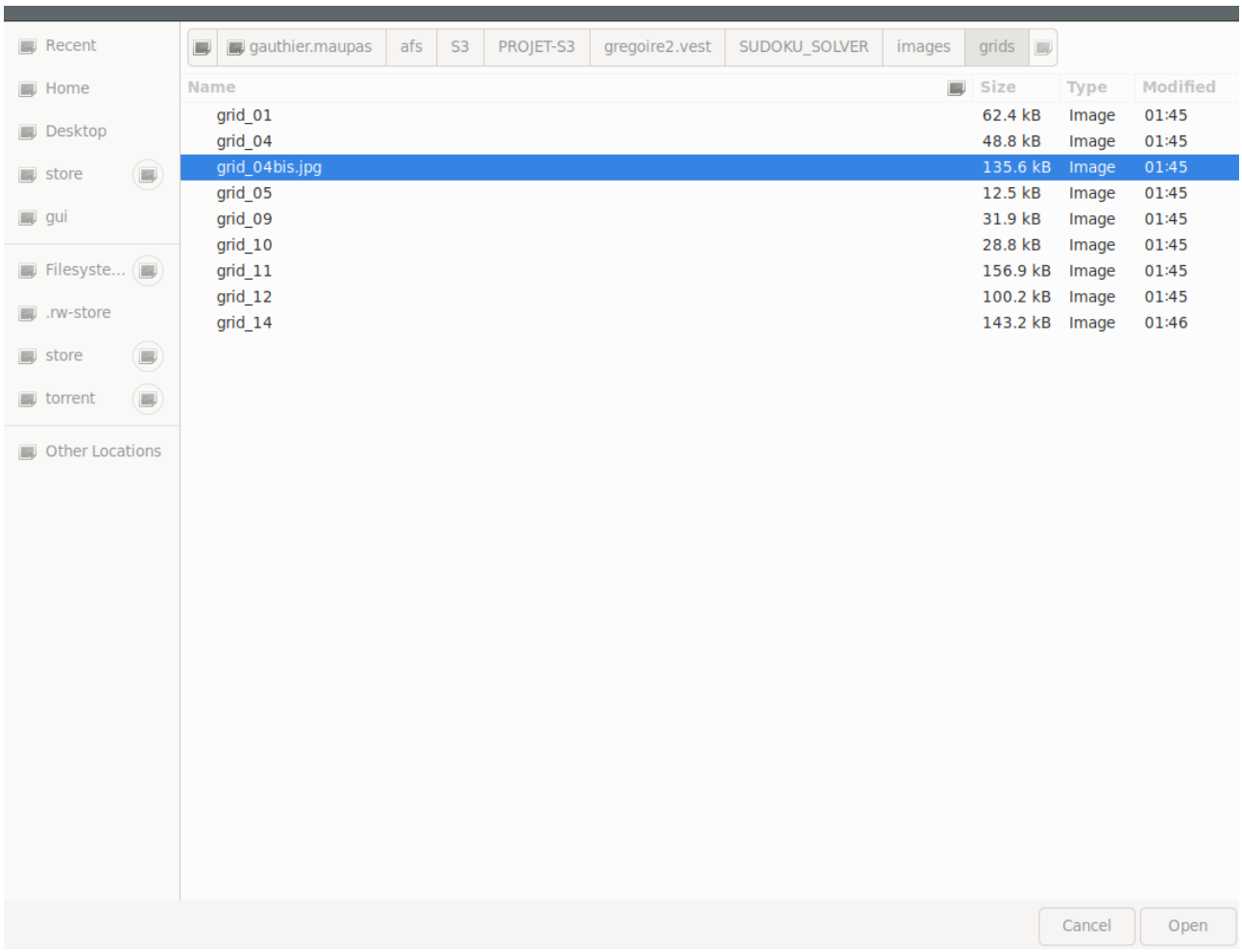
— Le bouton rotation, pour la rotation nous pouvons effectuer une rotation manuel en inscrivant dans l'input un degré de rotation. Pour effectuer une rotation automatique il faut laisser le champ vide.

— Le bouton sobel permet d'appliquer le filtre solbel sur l'image.

— Le bouton detect grid permet de détecter la grande grille de notre sudoku.

— Le bouton hough transform permet de tracer les droites intermédiaires pour la segmentation

— Le bouton save qui permet d'enregistrer la grille résolue. Le bouton save permet d'enregistrer le grille résolue mais il ne permet pas à l'utilisateur de choisir le nom et l'endroit ou enregistrer le fichier. Il est automatiquement renommé result.bmp et enregistrer dans le dossier tmp_images/.



Selection d'une grid

6.3 Difficultés

Le logiciel Glade n'a pas été facile à comprendre, avec le logiciel glade notre plus grande difficultée à été de placer nos éléments de façon correct dans l'interface et de comprendre l'utilisation des centenaires parents avec des grids.

Nous avons aussi rencontré des difficultés à rassembler tout le projet sur l'interface et de relier les boutons à leur fonctions car toutes les fonctions n'ont pas été codé pour une implémentation facile sur l'interface graphique.

7 Ressenti

7.1 Gauthier Maupas

Ce projet OCR est mon deuxième projet de groupe à EPITA et j'ai pu remarquer qu'avec mon groupe nous savions déjà tous comment travailler ensemble afin de ne pas créer des conflits git par exemple. Ce projet est un bon projet qui nous permet de travailler sur différentes utilisation du langage C comme un réseau de neurones, des algorithmes de résolution de sudoku ou faire une interface graphique. J'ai apprécié travailler sur ce projet mais pour moi trouver le temps après les cours afin de pouvoir avancer le projet a été difficile. Je pense qu'avec plus de temps nous aurions pu arriver à un meilleur aboutissement du projet. Je trouve aussi que ce qui a rendu ce projet difficile c'est de découvrir le langage C en même temps que de faire le projet. Mais je suis content de ce que nous avons réussis à faire avec notre groupe.

7.2 Alexandre Tsang-Hin-Sun

La vision par ordinateur est quelque chose qui me fascine et je m'y était déjà intéressé pour des projets personnels. Alors ce projet d'OCR est un challenge qui vaut les nuits sans sommeils pour parvenir au résultat espéré. La plus grande difficulté pour cette deuxième échéance était de bien tout formaliser, regrouper le travail de chacun avec l'interface graphique. Comme j'avais déjà expérimenté le traitement d'image auparavant je me suis occupé de toute la partie image incluant opération sur les images et segmentation. Ainsi que la reconstruction du solver a implémenter et a affiché sur l'image non résolu rentrer sur l'interface graphique en fonction des chiffres reconnus par le réseau de neurones.

7.3 Grégoire Vest

Pour moi, ce projet a été particulièrement compliqué, car c'est un sujet que j'ai du mal à comprendre et que ne m'intéresse pas plus que ça. J'ai cherché un peu ce que je voulais faire mais au final, vu que le réseau de neurone était la partie où tout le monde galérait, je me suis penché dessus et je suis l'heureux détenteur d'un réseau fonctionnel. Le projet n'est pas totalement abouti mais je dirais que ça vient surtout d'un manque de motivation après l'implémentation de l'OCR avec le traitement d'image qui nous a empêché de continuer. Personnellement j'étais bien trop fatigué après avoir réussi le réseau de neurone pour continuer à me pencher sur tout une autre partie du projet.

Au final, même si ça ne fonctionne pas super bien, je suis très content d'avoir pu aller jusqu'à ce point-là. Pour moi, nous avons réussi à faire de belles choses, et tout fonctionne bien quand on prend chaque partie indépendamment, c'est simplement un tout petit manque de coordination qui fait que le tout en fonctionne, ce qui, pour moi, reste quelque chose de minimal car j'ai appris pas mal de choses durant ce projet !

7.4 Alexandre Teirlynck

Je suis forcément déçu d'avoir échoué aussi proche du but , le projet était très long ces dernières semaines et éprouvantes pour les nerf. Le C n'est pas un langage très amical et très instinctif. Je me suis beaucoup énervé en voyant mes codes s'interrompre sans en avoir la raison comme en python ou en C. Le réseau de neurones était la partie que je souhaitais réaliser et je reste un peu dégouté de m'y être casser les dents dessus. Cependant d'un point de vue globale, je reste très satisfait sur la façon dont le groupe à travailler ensemble.

8 Conclusion

Malgré un résultat final qui ne résout pas un sudoku, Nous sommes tout de même contents de ce qui a été produit. Toutes les parties du solver de sudoku fonctionnent, du traitement de l'image à un réseau de neurones, tout le monde a effectué son travail. Le regret principal du groupe est de ne pas avoir testé le réseau de neurone sur des images dont le caractère n'occupait pas tout l'espace. Pour autant le projet est achevé l'interface graphique est complète et permet de comprendre tous les rouages mécaniques derrière la simple apparition de chiffres sur une grille de sudoku.