Projet OCR

EPITA 2ème Année Cycle Préparatoire

Rapport de Soutenance Finale

Luc Mahoux (Chef de projet)
Victor Chartraire
Garice Morin
Charles Neyrand
Décembre 2021

Table des matières

Introduction

Le projet OCR comporte trois grandes parties. La première est le traitement de l'image, pour reconnaître la grille de sudoku et extraire les cases de celle-ci. La deuxième partie consiste à créer un réseau de neurones capable de reconnaître les chiffres dans ces cases. Enfin, la dernière partie consiste à créer une interface graphique afin que l'utilisateur puisse se servir aisément de notre programme. Garice et Victor sont responsables de la gestion de l'image, Luc et Charles de la création du réseau de neurones. Pour cette dernière soutenance, nous sommes fier de pouvoir annoncer que notre OCR est fonctionnel pour les sudokus.

Tâches - Respo X / Suppléant O	Charles	Garice	Luc	Victor
Interface Graphique				X
Greyscale		X		
Binarisation		X		
Rotation de l'image				X
Rotation Automatique	O	X	O	O
Détection de la grille		X		
Découpage de la grille	O	O	O	X
Redimensionnement				X
Optimisation du Traitement		X		
Solver	X			O
Chargement de la Grille	O	O	X	O
Affichage de la Grille				X
Implémentation Réseau	X			
Création du jeu de données	X			
Propagation de l'erreur			X	
Mise à jour des poids/biais			X	
Sauvegarde/chargement du réseau			X	
Amélioration du réseau	O		X	
Entraînement du réseau	X		O	

1 Traitement d'images

Comme dit précédemment, Garice et Victor ont été en charge de la partie concernant le traitement d'images.

On peut diviser cette section en deux parties : la première concerne le traitement colorimétrique de l'image, et la deuxième portant d'avantage sur sa rotation et son découpage.

1.1 Noir et Blanc

Le passage en noir et blanc de l'image, autrement dit sa binarisation, est une étape extrêmement importante de l'analyse d'image. En effet, il est beaucoup plus simple d'analyser une image comportant deux valeurs qu'une en comprenant des milliers.

1.1.1 Greyscale

Pour arriver à ce résultat, nous passons tout d'abord l'image en nuances de gris. L'algorithme réalisant cette opération est des plus simples. Il suffit de parcourir l'image, et à chaque pixel, nous remplaçons sa valeur par la moyenne pondérée de la valeur de chacun de ses canaux (rouge, bleu et vert).

1.1.2 Binarisation

La binarisation a pour but de transformer une image en nuance de gris en une image consistant de 2 valeurs, dans notre cas le noir et le blanc.

Méthode de Sauvola

Une fois l'image traduite en nuances de gris, il faut maintenant décider pour chaque pixel s'il sera noir ou blanc.

Nous avons d'abord naïvement implémenté un algorithme de binarisation général, la méthode d'Otsu. L'algorithme calcule un seuil en fonction de la composition de l'image. Le problème de cet algorithme est qu'il a tendance à ne pas fonctionner correctement sur les zones d'ombres, effet qui est particulièrement visible sur les exemples de grilles de sudoku.

Nous nous sommes alors penché sur des algorithmes locaux. Ces algorithmes ont un fonctionnement similaire à celui de la méthode d'Otsu, mais il est appliqué à une petite partie de l'image et non l'image entière.

Quelques recherches sur Internet nous présentèrent de nombreuses méthodes, chacune ayant leurs qualités et leurs défauts. Nous avons tout d'abord implémenté une méthode qui définissait le seuil d'un pixel selon la valeur moyenne des pixels alentours. Bien que fonctionnelle, cette méthode n'était pas réellement satisfaisante, certaines images finissant avec une tache noire cachant une partie de la grille de sudoku.

Nous avons finalement retenu pour la première soutenance la méthode de Sauvola. Cette méthode calcule le seuil en fonction de la moyenne et de l'écart-type des valeurs de ses pixels voisins. Le seuil est donné par cette formule :

$$T(x,y) = m(x,y) \times [1 + k \times (\frac{s(x,y)}{R} - 1)]$$

T(x,y) est la valeur du seuil calculée

R est la valeur maximale de l'écart-type

k est un biais compris entre 0.2 et 0.5

m(x,y) est la moyenne de la valeur des pixels alentours

s(x,y) est l'écart-type des pixels alentours

Une fois le seuil déterminé, il suffit de définir la couleur du pixel à noir si sa valeur est inférieure au seuil, à blanc sinon.

Méthode de Wellner

Dans la suite du projet, nous avons dû changer de méthode, car comme dit plus haut, la méthode de Sauvola n'obtenait pas des résultats satisfaisants sur des images fortement bruitées. Nous nous sommes alors penché sur la méthode de Wellner. Son fonctionnement est assez différent de la méthode précédente.

Tout d'abord, nous commençons par calculer une nouvelle image, appelée "image intégrale" de l'image. L'image est considérée comme un tableau d'entier à deux dimensions. La valeur d'un entier en une position (x,y) nous est donnée par cette formule :

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

I(x,y) est la valeur de l'image intégrale à la position (x,y)

f(x,y) est la valeur du pixel de l'image d'origine à la position (x,y)

Une fois cette image intégrale calculée, il faut maintenant binariser l'image. Pour cela, nous calculons, en un temps constant grâce à l'image intégrale, la valeur moyenne des pixels de l'image dans une fenêtre de taille fixe. Cette valeur est obtenue grâce à cette équation :

$$\sum_{x=x1}^{x2} \sum_{y=y1}^{y2} f(x,y) = I(x2,y2) - I(x2,y1-1) - I(x1-1,y2) + I(x1-1,y1-1) \quad (1)$$

f(x,y) est la valeur moyenne des valeurs de l'image intégrale contenues dans une fenêtre de taille (x2-x1)*(y2-y1)

I(x,y) est la valeur de l'image intégrale à la position (x,y)

Ensuite, nous comparons la valeur obtenue dans l'expression (1) (que l'on appellera λ ici) à la valeur de niveau de gris de l'image originelle (que l'on appellera μ). Définissons aussi un seuil t avec $0 \leq t \leq 100$ et appelons le nombre de pixels dans la fenêtre n.
Si $n\mu < \lambda(100 - t)/100$ alors le pixel de l'image deviendra noir, blanc dans le cas contraire.

Une dernière chose reste à vérifier. En effet, le seuil t est dépendant de la quantité de bruit sur limage. Après quelques essais, nous avons remarqué qu'il suffisait de passer un première fois sur l'image avec un seuil faible (≈ 15), puis de calculer la valeur moyenne des pixels sur l'image binarisée, et si cette moyenne est inférieure à une certaine valeur de repasser sur l'image avec un seuil plus élevé (≈ 60).

Cette méthode nous permet d'avoir une image finale ne contenant presque aucun bruit pouvant déranger les algorithmes qui utiliseront l'image binarisée. Le temps d'exécution de la fonction se retrouve aussi grandement réduit, les calculs étant bien moins nombreux.

1.2 Rotation de l'image

Le principe de cette fonction est de pouvoir faire pivoter une image, par rapport à son centre, d'un angle donné par l'utilisateur. L'angle fourni par l'utilisateur correspond à une rotation dans le sens horaire.

1.2.1 Dimensionnement de l'image finale

La première étape a été de calculer la taille de l'image d'arrivée. Ne voulant aucune perte, et un minimum d'ajouts de données inutiles, nous nous devons

de définir la largeur et la hauteur de l'image pivotée. Pour se faire nous n'utilisons que des formules de trigonométries basiques pour calculer les "côtés" des angles qui nous intéressent.

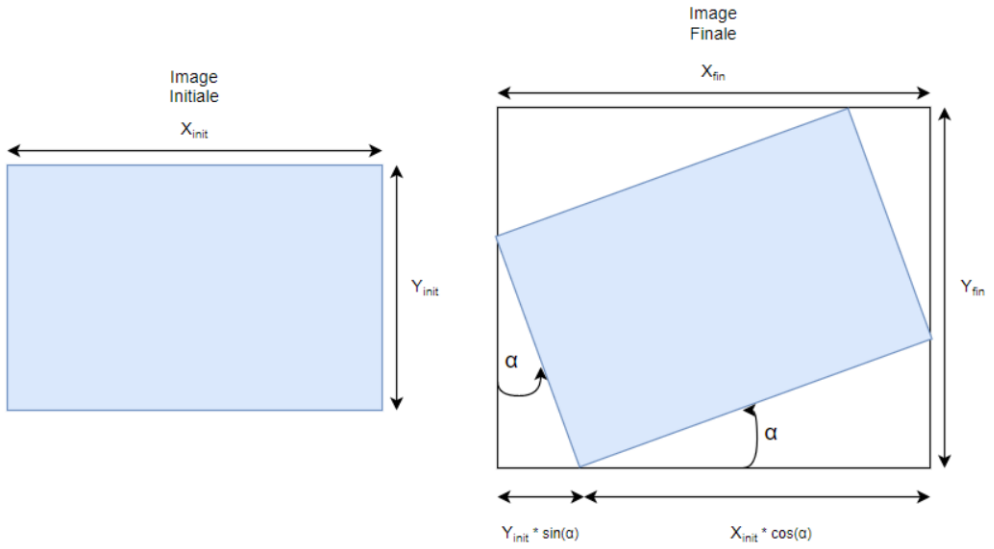


FIGURE 1 – Calcul des dimensions d'une image pivotée d'un angle α

1.2.2 Rotation et interpolation

La deuxième étape de cette fonction a été de décaler l'origine de notre image. Effectivement, d'un commun accord nous avons décidé de définir le pixel de coordonnées (0, 0) comme celui du coin haut-gauche de l'image. Cependant, la rotation devant se faire par rapport au centre de cette image, les calculs de coordonnées se retrouvent simplifiées pour la suite en les décalant.

Enfin, la dernière partie a été de pivoter l'image. La première idée était de parcourir l'image initiale, et pour chaque pixel, calculer son pixel "image" sur la sortie et laisser les autres pixels en blanc. Cependant, cela entraîne un problème majeur : étant donné que chaque pixel ne correspond qu'à un seul pixel sur l'image d'arrivée, et que cette image ne peut être que plus grande ou égale en taille, des lignes blanches dues au manque d'antécédents apparaissaient en plein milieu de l'image tournée.

La solution est alors de faire le chemin inverse : nous parcourons chaque pixel de l'image finale, puis nous calculons les coordonnées de son antécédent, c'est l'interpolation. Pour finir, nous vérifions que celui-ci est bel et bien dans l'image de départ et si c'est le cas, nous le reportons, sinon on met un pixel blanc à la place.

La formule pour la rotation d'un point P de coordonnées (x, y) d'un angle α est la suivante :

$$P' = P \times M \tag{2}$$

- P' la matrice de coordonnées résultante de la rotation du point P
- M la matrice de rotation par l'angle α

La matrice de rotation M par l'angle α est :

$$M = \begin{pmatrix} \cos(\alpha) & \text{-}\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

D'où, le point P' a pour coordonnées :

$$P' = \begin{pmatrix} \text{x'} & \text{y'} \end{pmatrix} = \begin{pmatrix} \text{x} & \text{y} \end{pmatrix} \begin{pmatrix} \cos(\alpha) & \text{-}\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

$$(1) <=> \begin{cases} \text{x'} = \text{x}.\cos(\alpha) + \text{y}.\sin(\alpha) \\ \text{y'} = \text{x}.\text{-}\sin(\alpha) + \text{y}.\cos(\alpha) \end{cases} \tag{3}$$

Nous choisissons de mettre des pixels blancs dans les zones qui n'ont pas d'antécédents pour ne pas gêner les traitements qui suivent. L'ajout de pixels noirs, par exemple, aurait été considéré comme un bruit supplémentaire sur l'image tandis que le pixel blanc n'influe sur aucun calcul.

1.3 Rotation Automatique

1.3.1 Avant le découpage de la grille

Une fois la rotation de l'image fonctionnelle, il fallait encore que le programme puisse déterminer de lui-même l'orientation de l'image afin de la compenser.

Pour cela, nous n'avons pas réinventé la roue et nous avons utilisé des fonctions que nous avions déjà codé pour la première soutenance. En effet, la première idée qui nous est venue était d'utiliser la transformée de Hough, qui nous servait alors à détecter la grille, afin de trouver une orientation moyenne des lignes, qui correspondrait à une rotation de l'image. Mais il n'était pas possible d'intégrer cela n'importe comment. Une ligne oblique, appartenant ou non à la grille de sudoku, aurait pu rendre la moyenne fausse, et par effet boule de neige, rendre toute la résolution du sudoku impossible.

Nous avons alors pensé à utiliser une image ne contenant que la grille. Cela tombe bien, nous le faisions déjà pour découper la grille de l'image fournie par l'utilisateur. Ne nous restait plus qu'à effectuer une transformée de Hough sur cette image isolée. Un autre problème surgit alors, l'image était toujours tournée de 45 degrés par rapport à sa rotation d'origine. Après quelques recherches, il s'avéra que ce problème provenait de notre calcul de l'orientation moyenne des droites.

En effet, nous ne séparions pas les droites "horizontales" des "verticales". Or cela posait un problème, car la moyenne des orientations nous donnait une valeur de 45° ± l'angle d'origine. Il nous a alors fallu séparer les lignes selon leur orientation, et ne calculer la moyenne de l'orientation uniquement sur soit les lignes verticales, soit les lignes horizontales, le choix de l'une ou l'autre ne changeant pas fondamentalement le résultat.

Après quelques légers ajustements, le programme est capable de déterminer avec une précision d'environ 1° la rotation de l'image. Ce degré de différence n'importune que très peu le reste du programme.

1.3.2 Après le découpage de la grille

Cette étape intervient après que la grille ait été isolée sur une image. A ce stade, nous ne savons pas si l'image n'est pas retournée d'un multiple de 90°, les lignes ne portant aucune information à ce sujet. Nous avons alors décider de faire tourner plusieurs fois (au maximum 4) la grille de 90° en envoyant à chaque fois le résultat au réseau de neurones. Si le réseau trouve une solution à cette orientation de la grille, nous arrêtons les rotations et passons à l'affichage de la grille résolue. Dans le cas contraire, nous tournons encore l'image de 90°. Si au bout de 4 essais, soit un tour complet de l'image, le solveur ne trouve pas de solution à la grille, nous considérons la grille comme non résolvable, et nous le précisons à l'utilisateur par le biais de l'interface graphique.

Bien que cette solution ne soit pas des plus efficaces en termes de temps de traitement, elle est la plus simple à mettre en oeuvre et ne demande pas d'algorithme complexe. Bien sûr, cette solution a aussi l'inconvénient de pouvoir, si les planètes sont bien alignées, donner une bonne réponse pour une mauvaise orientation. Par exemple, une grille de départ composée uniquement de 6, de 9 et de 8 pourrait fournir une réponse malgré une image retournée de 180°. Nous aurions bien sûr préféré que ce "edgecase" n'existe pas, mais faute de temps, nous n'avons pas trouvé de solution plus optimale.

1.4 Détection de la grille

1.4.1 Pour la première soutenance

La détection de la grille est essentielle afin de fournir au réseau de neurones les bonnes données. En effet, nous voulons lui fournir les chiffres de la grille et non autre chose. A partir de cette étape, nous considérons la grille droite et correctement binarisée.

Pour détecter la grille, nous utilisons les "connected components". Cette méthode consiste à détecter des pixels adjacents de même couleur, et d'étiqueter ces groupes de pixels en "clusters". Ces "clusters" sont ensuite stockés.

Une fois tous les clusters détectés, il faut trouver le plus grand d'entre eux, qui représentera dans la plupart des cas la grille de sudoku. Pour ce faire, nous devons d'abord fusionner les "clusters" adjacents. Nous parcourons à nouveau l'image afin de compter le nombre de pixels de chaque groupe et déterminer le plus grand.

Finalement nous ne conservons que les pixels appartenant au plus grand groupe, tous les autres devenant noirs.

Une fois les données inutiles supprimées de l'image nous devons encore détecter les cases de la grille. Pour cela nous utilisons la transformée de Hough. Son déroulement est le suivant :

— On crée un espace de paramètre H(ρ, θ) où ρ et θ correspondent aux paramètres polaires d'un segment passant par l'origine d'un repère cartésien, normal à la droite.

— A chaque pixel blanc de l'image, on calcule le paramètre ρ en fonction de θ.

— A chaque θ, on augmente la valeur du point correspondant dans l'espace de paramètres H.

— Une fois l'image parcourue en entier, on parcourt l'espace de paramètres, en conservant chaque couple de paramètres (ρ,θ) dont la valeur excède un certain seuil.

— A partir des équations de la normale, il est possible de retrouver une équation de droite cartésienne.

Une fois les équations de droites obtenues, nous récupérons les coordonnées des intersections des coins de la grille, ce qui nous permet de créer une nouvelle image de la taille de la grille, ne contenant que celle-ci, afin de pouvoir en extraire les cases.

1.4.2 Pour la deuxième soutenance

1.4.2.1 Isolation de la grille A la suite de la première soutenance, nous avons revu la méthode de découpage de la grille. Bien qu'élégante en terme d'algorithme, la méthode précédemment utlisée à l'inconvénient majeur de reposer sur la détection de ligne. C'est-à-dire que si une ligne est mal détectée, la coordonnée de l'un des coins de la grille peut s'en retrouver faussée. Pour répondre à cela nous avons choisi une approche plus simple. Nous parcourons simplement l'image et gardons une trace du pixel blanc le plus à gauche, le plus à droite, le plus haut et la plus bas, afin de définir un quadrilatère contenant la grille. Si celui-ci n'est pas un carré, nous rajoutons des pixels sur le côté le plus court, afin de pouvoir avoir un découpage des cases régulier. Cette méthode ne nous était pas possible précédemment, en effet, il nous faut une image avec un minimum de bruit afin de ne pas trouver des pixels blancs loin de la grille.

1.4.2.2 Suppression des bordures Une autre amélioration que nous devions apporter était la suppression des bordures extérieures de la grille. En effet, sur certaines images, nous observions l'apparition d'une épaisse bordure, ce qui avait pour effet de créer un décalage non négligeable lors du découpage de l'image. Notre première idée pour remédier à ce problème, nous partons des coins de l'image et en nous déplaçant diagonalement, nous repérons l'endroit où se termine la bordure. Le fait de se déplacer en diagonale (rappelons que l'image fournie à la fonction est carrée) nous permet de n'utiliser qu'une seule boucle, et ainsi de réduire la complexité. Le résultat était plutôt satisfaisant, les bordures étant dans le meilleur des cas complètement retirées, ou alors très largement réduite, et ne causant plus qu'un décalage négligeable des cases lors du découpage.

Une autre solution nous est venu plus tard. Comme nous avions toujours une image ne contenant que le contour de la grille, nous avons eu l'idée de nous en servir afin de supprimer la grille sur l'image fournie par l'utilisateur. Pour cela, nous parcourons notre image isolée, et à chaque pixel blanc que nous rencontrons sur celle-ci (rappelons que l'image est blanche sur un fond noir), nous remplaçons le pixel à la même position sur l'image de l'utilisateur par un pixel blanc. De cette façon, nous sommes sûrs de supprimer à la fois les bordures les plus extérieures mais aussi les bordures intérieures. De cette façon, nous n'avons pas de pollution sur les images que nous découpons sur la grille, et le réseau de neurones n'est pas dérangé par une ligne de pixels noirs qui pourrait l'induire en erreur.

1.5 Découpage et redimensionnement

Pour le découpage, nous considérons que l'image donnée en entrée respecte les critères requis par la fonction de scindage :

- L'image est une grille de sudoku centrée et sans bords inutiles (issue de la fonction de détection de la grille)
- L'image est carrée ($hauteur = largeur$)

Cette partie renvoie une "liste" d'images, chaque élément correspondant à une case de la grille. Nous initialisons chaque élément comme une case "vide", puis nous allons recopier chaque pixel de la grille correspondant à la case voulue à l'intérieur de cet élément.

Pour le redimensionnement, de même que pour le découpage, nous considérons que l'image fournie respecte les critères de la fonction :

- La fonction de redimensionnement ne doit s'appliquer que sur les cases issues du découpage de la grille
- La case est carrée ($hauteur = largeur$)

En effet, cette fonction fournie en sortie une image qui peut être acceptée par le réseau neuronal, une image carrée de 28 × 28 pixels. Pour ce faire, nous calculons premièrement, un ratio r :

$$r = \frac{\text{côté}_{init}}{\text{côté}_{fin}}$$

Puis, pour chaque pixel de l'image finale, nous calculons sa position relative sur l'image initiale par la formule :

$$(x_{init}, y_{init}) = r \times (x_{fin}, y_{fin})$$

Pour le recopier ensuite sur l'image de sortie.

1.6 Centrage des chiffres

Une dernière optimisation que nous avons apporté au traitement d'image est une fonction ayant pour but de recentrer le chiffre présent sur les cases découpées. En effet, pour diverses raisons, comme un effet de perspective sur l'image de base, le chiffre présent peut se retrouver décalé sur l'un des côtés. Cela ne serait pas dérangeant si le réseau neuronal n'était pas entraîné sur des images centrées. Ces décalages peuvent alors causer des erreurs d'interprétation lors du passage dans le réseau de neurones. Il nous fallait donc corriger ce décalage.

Pour ce faire, nous cherchons tout d'abord le centre de celle-ci. La taille de l'image étant fixe et connue, il n'est pas compliqué de trouver le centre de l'image, une seule division par deux étant nécessaire. Pour ce qui est du centre du caractère, nous cherchons d'abord les bordures du chiffre. Nous calculons ensuite le centre grâce à de simples calculs de milieu. Soit a et b les deux extrémités d'une droite, avec $a < b$. Le centre k de cette droite est donnée par :

$$k = a + \frac{(b-a)}{2}$$

Une fois les centre de l'image et du chiffre déterminés, nous comparons leur position. Nous pouvons alors déterminer un décalage horizontal et vertical. Nous les appellerons dans la suite de ce rapport Δh et Δv. Nous créons une nouvelle image de même taille que l'image donnée en paramètre de la fonction. Ensuite pour chaque pixel de cette nouvelle image, nous regardons le pixel présent à la position (x+Δh,y+Δv) sur l'image d'origine. Si le pixel se trouve sur l'image d'origine, nous copions simplement la couleur dudit pixel. Dans le cas contraire, nous plaçons un pixel blanc.

1.7 Affichage de la Grille

L'affichage de la grille se fait sur l'interface, mais sa construction va être ici détaillée. Cette partie ne peut être atteinte que si la grille traitée actuellement possède une solution valide. En effet, si la grille possède une solution, on peut alors comparer les valeurs des cases de la grille d'entrée et de sortie. En entrée, une case vide équivaut à un 0. De ce fait, toutes les cases qui étaient des 0 ne le sont plus après passage dans l'algorithme de résolution.

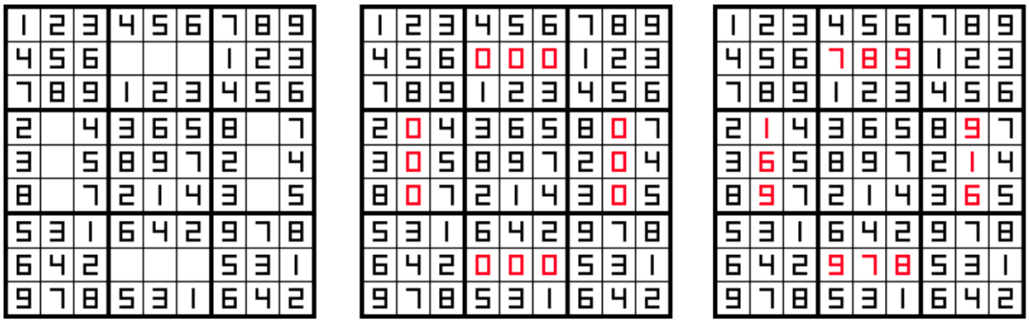


FIGURE 2 – Grilles entrante, utilisée, résultante

Cette fonction utilise des images prédéfinies et stockées dans l'arborescence du projet. A l'aide du site Pixilart, nous avons pu créer toutes ces images en

Pixel Art. Au début nous récupèrons une grille vide (seules les lignes sont représentées), puis pour chaque case, on va comparer les valeurs de la grille "utilisée" à 0 et en définir si la valeur était déjà ou non présente. Le chiffre de la grille résolue est alors ajouté dans la grille dans la couleur correspondante : Noir si le chiffre était présent à l'entrée et rouge si le chiffre à été rajouté par l'algorithme de résolution de sudoku.

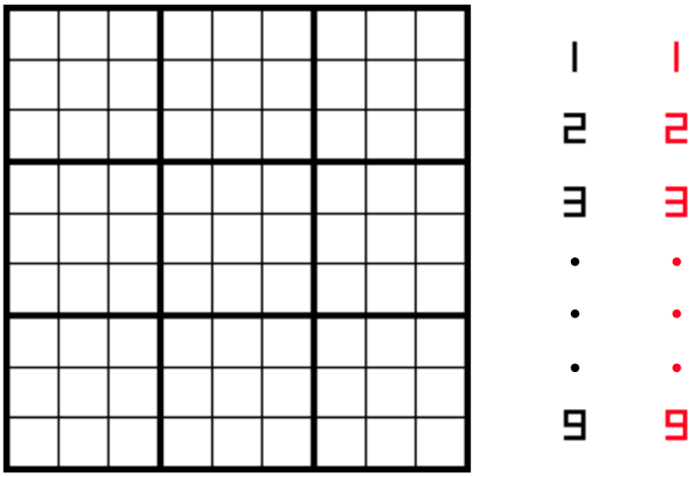


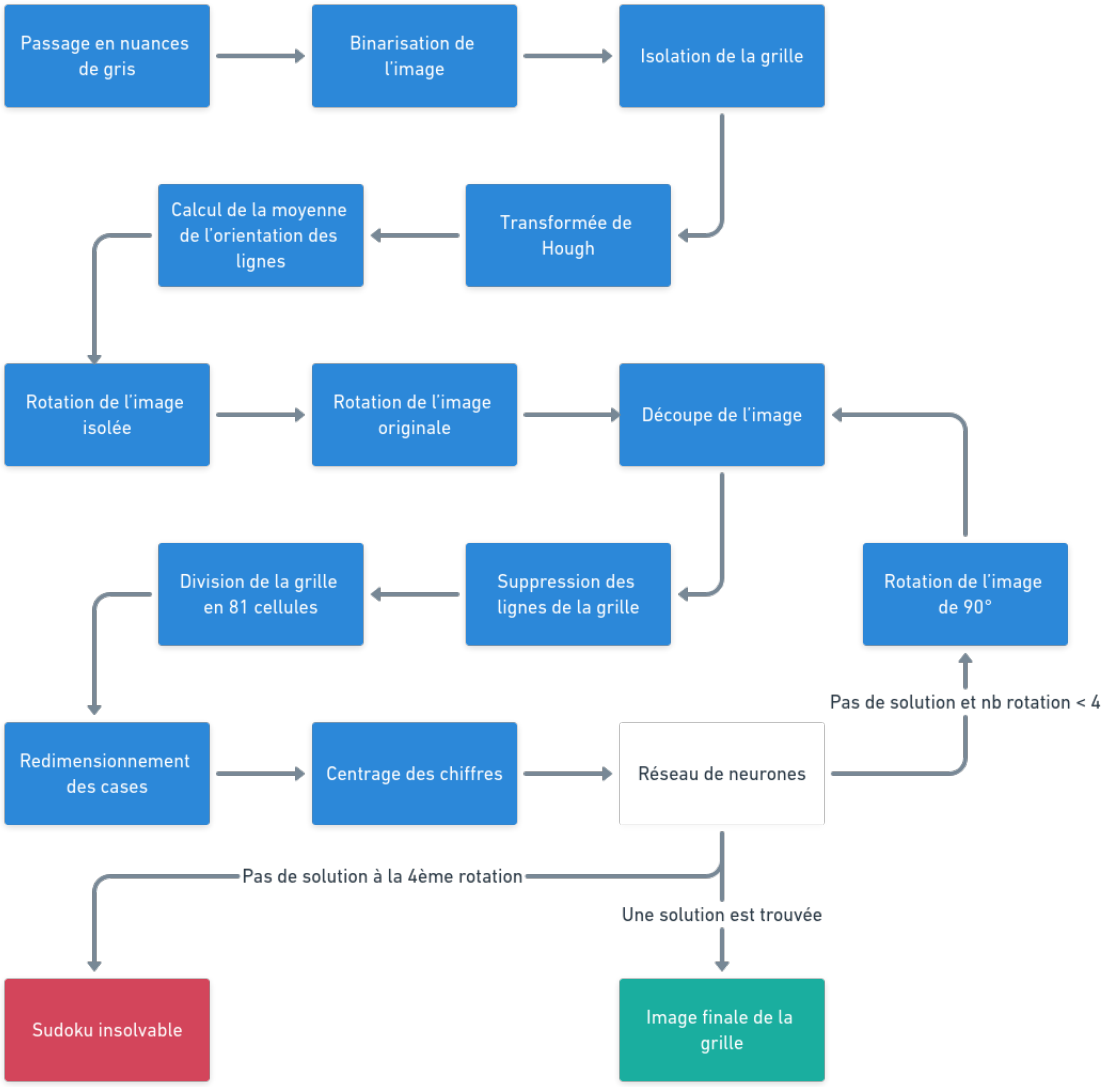
FIGURE 3 – Images prédéfinies utilisées

L'image finale est aussi enregistrée en tant que "solved.png" pour qu'elle soit accessible et pour faciliter son affichage dans l'interface.

1.8 Résumé des changements à la deuxième soutenance

Les modifications du code pour cette soutenance se sont surtout résumées à des optimisations diverses, notamment sur la binarisation de l'image, les fonctions que nous avions codées avant cette deuxième soutenance n'étant pas les plus optimales et plus important ne fonctionnant pas sur certaines images, celles-ci ressortant avec beaucoup trop de bruit pour les fonctions suivantes. Nous avons aussi du rajouter une fonction de rotation automatique dont nous expliquons le fonctionnement dans la section 1.3.1 de ce rapport. La binarisation de l'image n'est pas encore parfaite, et certains algorithmes pouvant être encore optimisé. Mais le temps limité ne nous a pas permis de mener à bout cette démarche.

1.9 Fonctionnement du traitement de l'image



Fonctionnement du processus de traitement de l'image.

2 Réseaux de neurones

Charles et Luc ont été en charge de la partie concernant les réseaux de neurones.

Notre but pour la première soutenance était de réussir à implémenter un réseau de neurones reconnaissant le "OU Exclusif", ou plus simplement "XOR". Pour la soutenance finale il était nécessaire d'implémenter la reconnaissance des chiffres, et donc nous avons revu notre implémentation, tout en ajoutant des paramètres au réseau pour de meilleurs résultats.

Cette partie est donc divisée en trois : une première afin d'expliquer notre raisonnement pour résoudre le problème du XOR, une seconde afin d'expliquer les étapes effectuées depuis la première soutenance pour la reconnaissance des chiffres et une dernière sur la sauvegarde et le chargement du réseau de neurones à partir d'un fichier.

2.1 XOR

2.1.1 Représenter un réseau de neurones

Un réseau de neurone est constitué principalement de couches de neurones. Chaque couche contient plusieurs neurones. Mais représenter une couche n'est pas si intéressant. En effet, ce qui importe ce sont les connexions, les liaisons entre les différentes couches. Il y a deux type de liaisons principalement : les poids et les biais.

Notre implémentation du réseau de neurones a donc commencé ainsi. Une struct "Neural" contenant le nombre de couche, le taux d'apprentissage et un pointeur de struct "Layer". Nous avons ensuite implémenté la struct "Layer", qui représente les connexions entre deux couches de neurones. Cette struct contenait au départ un pointeurs de float qui représentait les poids, un autre représentant les biais, et deux float qui contenait la taille de ces pointeurs. Pour pouvoir mieux gérer les pointeur, c'est à dire pour les allouer qu'une seule fois et pouvoir les libérer facilement, nous avons créé des fonction qui initialisait ces struct et d'autres qui les détruisaient.

Pour implémenter l'algorithme de Rétro-propagation nous avons ajouté plusieurs pointeurs de float au struct "Layer". Cela nous a permis de garder en mémoire les informations importantes comme la matrice d'erreurs porté par les poids.

— weights : matrice des poids de la couche

— bias : matrice des biais de la couche

— z : somme des biais et du produit matriciel des poids par les inputs

— output : application de la fonction sigmoïde à z

— error : matrice des erreur propagé lors de la rétro-propagation

— deltaB : matrice de la somme des erreurs propagé aux biais en une époque

— deltaW : matrice de la somme des erreurs propagé aux poids en une époque

2.1.2 Questionner le réseau

Une fois la structure du réseau de neurones construite, nous nous sommes penchés sur le fonctionnement proprement dit. La première fonction que nous avons dû créer est query(). Cette fonction prend deux données en entrée et elle retourne le résultat du réseau de neurones. Ces données sont comprises entre 0 et 1. Dans le cas du XOR, nous donnerons à la fonction une parmi les quatre combinaisons possible de couple de 0 et 1. Cette fonction appelle feedforward(), qui est aussi utilisée pour la rétro-propagation. Pour chacune des couches, la fonction feedforward() utilise la sortie de la couche précédente (les données entrées pour la première couche). Nous faisons le produit matriciel des poids de la couche et de cette sortie. Nous ajoutons les biais à la matrice de résultat et nous appliquons la fonction sigmoïde à tous les éléments du vecteur.

Soit a la matrice d'activation de la couche actuelle, w les poids et b les biais de la couche actuelle et i l'activation de la couche précédente :

$$a = \sigma(wi + b)$$

La matrice d'activation de la dernière couche est le résultat donné par le réseau de neurones. Le résultat est compris entre 0 et 1. Tout résultat supérieur à 0,5 est considéré comme 1, tout résultat inférieur comme 0.

2.1.3 Entraîner le réseau

Comme les poids et les biais du réseau sont initialisés aléatoirement, le résultat obtenu est aussi aléatoire. Afin de se rapprocher de l'objectif, il est donc nécessaire "d'entraîner" le réseau de neurone, afin d'améliorer les résultats obtenus.

L'optimisation de notre algorithme va être représentée par ce que l'on appelle une fonction de coût. Grossièrement, cette fonction de coût calcule la différence entre le résultat attendu et le résultat obtenu ; le but étant bien évidemment de se rapprocher le plus possible de 0. La fonction de coût choisie pour le XOR est la fonction quadratique.

Pour ce faire, nous avons utilisé l'algorithme du gradient. Celui-ci consiste à trouver le minimum d'une fonction différentiable (ici la fonction de coût).

Afin d'appliquer cet algorithme, nous avons alors besoin de calculer les erreurs (la différence entre le résultat souhaité et celui obtenu) sur chaque noeud. Il a fallu dans un premier temps calculer l'erreur obtenue à la sortie. Puis, nous avons eu à utiliser les erreurs obtenues à la sortie afin de les propager "en arrière", de la sortie vers l'entrée. C'est la fameuse fonction de rétro-propagation, ou "backpropagation", dont nous reparlerons plus tard.

Une fois l'erreur calculée, il fallait mettre à jour les biais et les poids du réseau de neurones. Nous avons alors utilisé des "époques" : nous faisons passer les tests par lots et nous récupérerons la moyenne des erreurs afin de mettre à jour définitivement les poids et les biais.

Voyons plus en détail les différentes étapes pour améliorer les résultats du réseau de neurones.

Calculer l'erreur à la sortie

La première étape, une fois le résultat obtenu pour une certaine entrée, est donc de calculer l'erreur à la sortie pour cette même entrée. Pour notre XOR, cela a été fait avec la fonction output_error() qui prend en paramètre le réseau de neurones et la liste des résultats que l'on souhaite avoir sur chaque neurones de la couche finale.

Voici différentes notations que nous allons utiliser désormais pour simplifier la rédaction et la lecture :

- L : couche de sortie
- l : couche quelconque
- δ : erreur
- δ^L : vecteur d'erreur à la couche de sortie
- y : résultat attendu
- o : résultat obtenu
- z : résultat d'un neurone avant l'application de la fonction sigmoïde

— $\sigma'(z)$: dérivée de la fonction sigmoïde appliquée à z
— Δw^l : matrice des correctifs à apporter aux poids de la layer l
— Δb^l : vecteur des correctifs à apporter aux biais de la layer l

On a alors, pour l'erreur à la sortie :

$$\delta^L = (o^L - y) \odot \sigma'(z^L)$$

Nous calculons la matrice erreur en appliquant la formule ci-dessus et la stockons donc dans l'attribut correspondant dans la couche finale du réseau de neurones.

Comme vous l'avez peut-être déjà remarqué, la quasi-totalité des calculs pour le réseau de neurones sont des opérations sur les matrices. Nous avons donc logiquement implémenté les fonctions afin de pouvoir effectuer ces opérations sans réécrire le code pour chaque fonction. Ces fonctions dans la version finale de l'OCR sont présentes dans le fichier vector_op.c. Les différentes fonctions de ce fichier sont :

— add_mat : écrit dans la matrice r le résultat de l'addition des deux matrices m1 et m2
— sub_mat : écrit dans la matrice r le résultat de la soustraction des deux matrices m1 et m2
— mul_mat : écrit dans la matrice r le résultat de la multiplication matricielle de la matrice m1 par la matrice m2. Le résultat écrase les valeurs contenues dans r avant l'opération
— mul_mat_add : écrit dans la matrice r le résultat de la multiplication matricielle de la matrice m1 par la matrice m2. Le résultat ici s'ajoute à celui déjà présent dans r avant l'opération
— transpose : écrit dans la matrice r la transposée de la matrice m
— hadamard_prod : écrit dans la matrice r le résultat du produit de hadamard entre la matrice m1 et m2
— mult_const : écrit dans la matrice r le résultat de la matrice m multipliée par un scalaire k
— apply_func : écrit dans la matrice r le résultat de l'application d'une fonction à chaque case de la matrice m

Propager l'erreur dans le réseau

C'est ici que la phase la plus fondamentale de la rétro-propagation entre en jeu. En effet, nous utilisons ici les erreurs trouvées à la layer l+1 afin de trouver la matrice layer de la couche l.

Ainsi, maintenant que l'erreur est calculée à la sortie, nous calculons les matrices erreurs de toutes les autres couches, en partant de L - 1 jusqu'à la première couche. C'est la fonction backpropagate_error() qui va propager dans tout le réseau de neurones les matrices erreurs à partir des erreurs trouvées par output_error. Cette fonction va être appelée une seule fois par entrée, comme la fonction output_error.

Voici la formule utilisée afin d'obtenir la matrice des erreurs sur la couche l :

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Cette formule a l'air complexe mais en réalité elle ne l'est pas tant que ça. Nous faisons le chemin inverse par rapport à la fonction feedforward() expliquée précédemment. Nous multiplions donc la transposée de la matrice d'erreur de la couche suivante par l'erreur trouvée à la couche suivante. On applique le produit matriciel d'Hadamard avec la matrice trouvée par la dérivée de la fonction sigmoide et nous obtenons le résultat escompté.

Si nous utilisons la transposée de la matrice de poids, on peut penser encore une fois que l'on effectue une opération qui est "inverse" à celle du feedforward. De plus, si nous n'utilisons pas la matrice transposée des poids, le produit matriciel ne peut pas avoir lieu car le nombre de colonne de la matrice de poids ne pourrait pas correspondre avec celle des erreurs.

Nous avons désormais mis à jour les erreurs de tous les neurones du réseau. Il est temps de les utiliser afin de calculer les deltaW et les deltaB dans tout le réseau. La fonction backprop() qui a pour argument le réseau de neurones à entraîner et l'input, est utilisée afin d'appeler les fonctions feedforward(), output_error() et backpropagate_error, mais aussi de calculer les deltaW et deltaB de tout le réseau. Encore une fois, ce sont ces valeurs que nous utiliserons afin d'optimiser le réseau de neurones.

Nous appliquons alors les calculs suivants afin de trouver les valeurs à ajouter à deltaW et deltaB (tous les deltaW et deltaB sont initialisés à 0 avant chaque début de traitement d'une époque) :

$$deltaW^l = deltaW^l + \delta^l.o^{l-1}$$
$$deltaB^l = deltaB^l + \delta^l$$

Nous sommes presque arrivé au moment de mettre à jour le réseau de neurones.

2.1.4 Mettre à jour les biais et les poids

Jusque là, nous avons expliqué le fonction backprop() qui va mettre à jour les deltaW et les deltaB de chaque couche du réseau en fonction d'une entrée donnée. C'est seulement ces valeurs qui nous seront utiles à la fin afin de recalculer la valeurs des poids et des biais.

Comme dit dans l'introduction de la partie "Entraîner le réseau", nous utilisons des époques afin d'entraîner le réseau de neurones.

Pour pouvoir entraîner un réseau de neurones, il faut propager l'erreur créée à partir de toutes les données d'entraînement. Une fois que le réseau s'est entraîné sur toutes les données, une epoch ou époque a été réalisé. Ce qui est important c'est que le réseau de neurones ne s'entraîne pas sur les exemples toujours dans le même ordre. La fonction shuffle() se charge de faire s'entraîner le réseaux sur les exemples trié aléatoirement. La fonction epoch() se charge donc d'appeler pour chaque données d'entraînement backprop().

Jusque là, nous avons cumulés les deltaW et les deltaB de chaque appel de backpropagation. C'est ici, dans la fonction epoch() que ces valeurs vont se montrer utiles.

En effet, comme dit plus tôt, nous souhaitons modifier les valeurs des biais et des poids en effectuant une moyenne sur des calculs effectués avec plusieurs entrées. Ces deltaW et deltaB ont accumulés les modifications que nous aurions dû apporter si nous ne voulions pas mettre à jour les valeurs par époque. Nous n'avons plus qu'à diviser les valeurs de deltaW et deltaB par le nombre d'entrées de l'époque, afin d'obtenir la moyenne. Nous allons multiplier la valeur obtenue par un learningRate η, inférieur à 1, qui va légèrement freiner les modifications apportées aux poids et aux biais. Nous obtenons donc les deux formules différentes, avec m le nombre d'éléments dans l'époque :

$$w^l_{new} = w^l_{old} - \frac{\eta}{m} \sum deltaW^l$$
$$b^l_{new} = b^l_{old} - \frac{\eta}{m} \sum deltaB^l$$

Voilà ! Il ne reste plus qu'à répéter cette opération un certain nombre de fois afin que le réseau de neurones reconnaisse la fonction XOR !

2.2 La reconnaissance des chiffres

Afin de permettre au réseau de neurones de reconnaître les chiffres, il était nécessaire de pousser le concept du réseau de neurones plus loin.

Tout d'abord, l'entraînement du réseau de neurones nécessitait alors l'utilisation d'un set d'images, représentant les différents chiffres, en tant que données d'entraînement. Nous avons de même ajouté des paramètres à notre réseau de neurones afin d'améliorer ses performances.

2.2.1 Revisite du code

Entre les deux soutenances, nous avons entièrement revu la structure et l'implémentation du réseau de neurones. Alors que nous n'avions qu'un seul fichier, nous avons désormais divisé le réseau en un certain nombre de fichiers, afin de mieux s'y retrouver. Nous avons pour la version finale les fichiers suivants, avec leur fichier ".h" correspondant :

— dataset.c : contient les fonctions nécessaires au chargement des données d'entraînement

— network.c : contient les structs du réseau de neurones ainsi que les fonctions directement liées

— test.c : contient les fonctions utiles au test du réseau de neurones

— ocr.c : contient les fonctions qui permettent d'utiliser le réseau de neurones et de l'entraîner

— save_setup.c : contient les fonctions pour charger et sauvegarder le réseau de neurones

— recover_grid.c : contient les fonctions qui permettent de transformer les images retournées par le pré-traitement de l'image afin de les convertir en des données utilisable par le réseau

— main_ocr.c : contient la fonction utilisée par le programme final, utilisant un réseau de neurones chargé depuis un fichier pour récupérer la grille des valeurs à partir d'images.

Ainsi ont été changées de nombreuses fonctions. Les fonctions output_error() et backpropagation_error() ont été supprimées et c'est désormais backprop() qui contient tout le processus de la rétro-propagation, d'une manière plus optimisée. De même, la fonction output() a été supprimée et désormais feedforward effectue les opérations de cette fonction plus rapidement. Une nouvelle fonction update_weights() a été créée pour simplement effectuer la mise à jour des biais et des poids. Elle est toujours présente dans la fonction epoch() mais la création de la fonction permet une lecture plus simple.

2.2.2 Le jeu de données

Une fois la structure du code entièrement revue, il a été nécessaire de trouver le jeu de donnée pour entraîner le réseau. Nous avons essayé deux jeux de donées : le MNIST et le TMNIST. En l'état, le réseaux s'entraîne sur le jeux de données du TMNIST.

Parser le MNIST

La première approche a été d'utiliser le jeu de donnée du MNIST. En effet, les ressources que nous avons utilisé pour apprendre à coder le réseaux utilisait souvent l'exemple de la reconnaissance de caractère manuscrit et donc utilisait le MNSIT. Le MNIST, pour Mixed National Institute of Standards and Technology, est un jeu de 60000 données d'entraînement et 10000 de test. Ce sont des chiffres écrit à la main et déjà étiqueté. C'est sur le site de M. Yann Lecun que nous avons trouvé ce jeu de données. Il était sous la forme de quatre fichiers idx-ubyte, un format pensé pour être facilement parsable. Ces quatre fichiers sont :

— train-images-idx3-uyte : le fichier contenant les images d'entraînement.

— train-labels-idx1-ubyte : le fichier contenant les étiquettes associées à chaque image

— tk10-images-idx3-ubyte : le fichier contenant les images de test

— tk10-labels-idx1-ubyte : le fichier contenant les étiquettes de test

Chacun de ces fichiers est organisé de façon similaire. La première ligne est toujous le "nombre magique". c'est un nombre, différend pour les étiquettes et

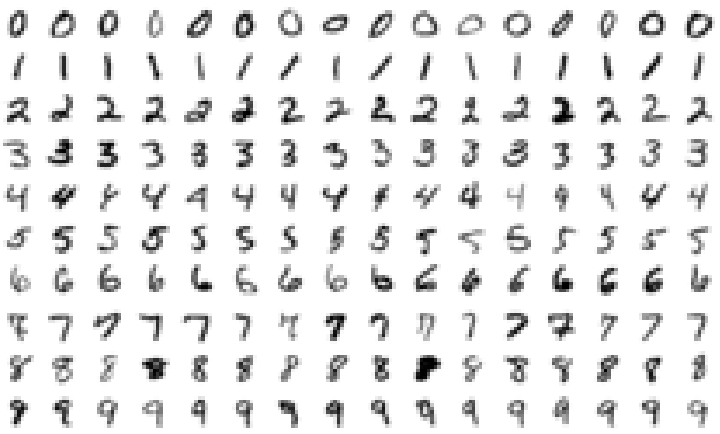


FIGURE 4 – Exemple de données du MNIST

les images, stocké sur la première ligne qui permet de vérifier que l'utilisateur lit correctement le fichier.

Ensuite la structure du fichier diffère selon que l'on lise un fichier d'étiquette ou un fichier d'image. Pour ce qui est d'un fichier d'étiquette il n'y a qu'une ligne d'information supplémentaire : le nombre d'étiquettes. Quant aux fichiers d'images il y a, en plus du nombres d'images, le nombres de colonnes et le nombre de lignes.

Une information très importante est avec combien de bits sont stocké ces informations. En effet ces informations sont stockées très précisement, d'où l'utilisation d'uint8_t et d'uint32t. Nous devions être sur que les nombres stoqué sur 32 et 8 bit prennent bien 32 et 8 bit en mémoire dans notre programme. Pour parser le fichier nous avons utilisés des structs. Conjointement avec une utilisation de fread(), le fichiers n'a pas été dur à parser. Nous nous retrouvions avec une structure de données, un Dataset, écho au Input_Data de la première soutenance, qui contenait un pointeur d'Image, un pointeur d'uint8_t pour les labels, en somme quelque chose d'utilisable

Entraîner le réseau avec le MNIST

Une fois cette struct "Dataset" créée, il a fallut entraîner le réseau de neurones. Nous avions a la dernière soutenance réalisé un réseau capable de reconnaître le "XOR". La structure de ce réseau était un peu particulière, 2 neurones d'entrées, 8 dans la couche caché et 1 en sortie. Nous nous étions trompé dans la fonction qui mettait à jour les poids et c'est pour cela que nous avions besoins d'autant de neurones sur la couche caché, le réseaux s'entraînait mal.

En reprenant notre code nous avons réglé cette erreur.

L'entraînement est divisé en "époque", une époque étant achevé lorsque l'algorithme de descente de gradient a été appliqué sur chacune des données d'entraînement. Pour plus d'efficacité les données d'entraînement sont divisé en groupe de 10, et les poids du réseaux mis-à-jour après avoir appliqué l'algorithme sur chacun des membres du groupe ou "batch". Les batchs sont formés aléatoirement avant chaque époque.

Ainsi chaque entraînement consiste en plusieurs époque, qui chacune améliore le réseau en mettant-à-jour les poids. Il faut toutefois mesurer la précision du réseaux de neurones. Pour cela nous utilisons les fichiers de test fourni avec le MNIST. Ce sont des fichiers qui contiennent des images et des labels qui ne sont pas présent dans les données d'entraînement pour vérifier la généralisation du réseaux à d'autre exemples. Il y a 10000 images de test. Après chaque époque le réseau est interrogé sur chacune de ces images pour voir s'il les reconnaît. Si c'est le cas on ajoute 1 à un compteur. On peut ainsi avoir après chaque époque le pourcentage de succès du réseau et ainsi modifier la structure de celui ci en fonction des résultats.

Un résultat correct pour un entraînement rapide est de choisir de créer le réseaux avec une couche d'entrée de 784 pour chacun des pixels, une couche caché de 30 neurones et une couche de sortie de 10 pour chacun des chiffres de 0 à 9. Après 3 époque d'entraînement on atteint une précision de 95%. Pour obtenir une précision de 97% nous avons finalement opté pour une couche caché de 128 et 7 époque d'entraînement.

Nous avons donc obtenu un réseau de neurones qui fonctionne... pour les nombres écrit à la main. Le jeu de données n'était pas adapté pour les nombres écrit par ordinateur. Il a fallut en trouver un.

Parser le TMNIST

Le jeu de donnée que nous avons décidé de choisir est le TMNIST, un jeu de donné inspiré du MNIST mais avec des chiffes écrit à la machine, provenant de 2990 polices d'écritures différentes. Le format des du fichier TMNIST est différend de ceux du MNIST, c'est un fichier csv, plus long à parser que les fichiers idx. Nous avons supprimer la première ligne du fichier qui ne contenait pas d'informations importante ainsi que la première colonnes qui contenait le nom de la police pour faciliter le parsing du fichier. On peut ainsi créer une struct Dataset qui permet de créer le réseau de neurones.

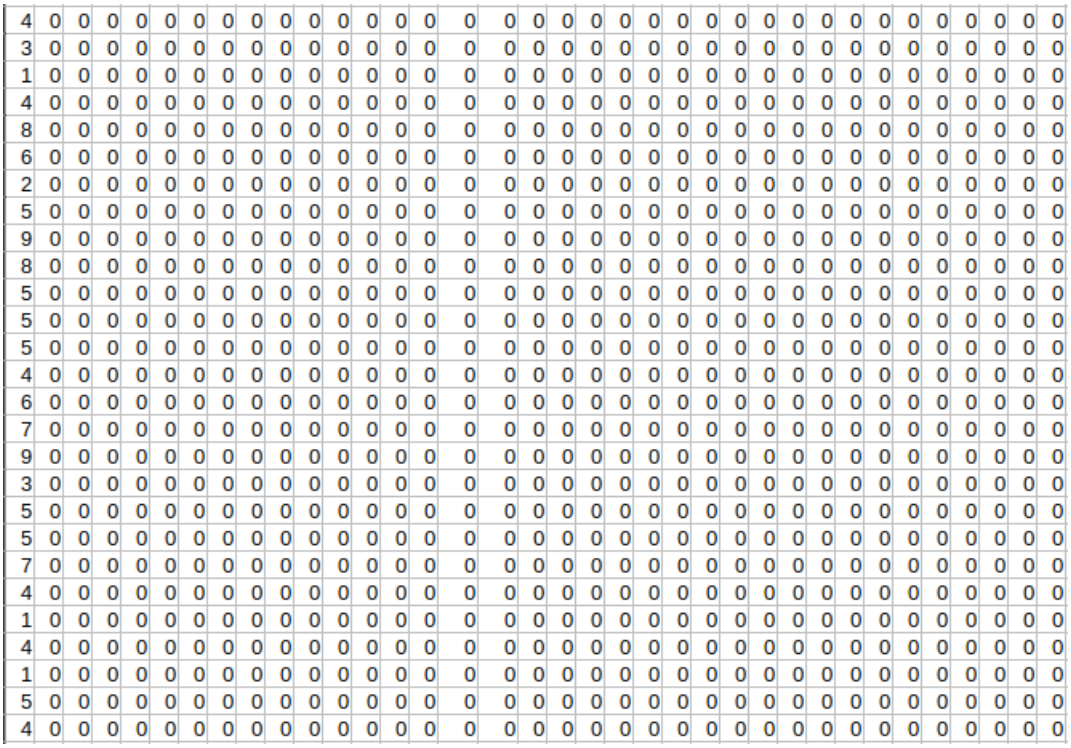


FIGURE 5 – Exemple de données du TMNIST

Utiliser le TMNIST

Pour pouvoir correctement mesurer l'efficacité d'un réseau de neuronnes il faut disposer d'un jeu de données de test, différends du jeu d'entraînement. Toutefois le TMNIST contient moins de données d'entraînement que le MNISt, nous avons donc décider de ne pas séparer le jeu en deux et ainsi perdre plusieurs précieuse données d'entraînement. Nous savions que le réseau était capable d'apprendre à reconnaître des caractères, et nous avons décidé de tester le réseaux directement avec les images de sudoku.

Utiliser le TMNIST comme jeu de données s'est révélé être la solution pour pouvoir reconnaitre les charactères des sudoku, bien sur il fut que les images soit centrées pour que le réseaux puisse reconnaître les caractères mais ce problèmes a été résolus dans la partie du traitement de l'image.

2.2.3 La fonction de coût "entropie croisée"

Nous avons rapidement évoqué la fonction de coût quadratique utilisée pour la première soutenance. Celle-ci nous permettait de calculer les nouvelles

valeurs des poids et des biais. Cependant elle comportait un défaut : elle apprenait rapidement lorsque l'erreur était moyenne ou faible, en revanche elle apprenait relativement lentement lorsque l'erreur était grande.

C'est donc ici que la fonction d'entropie croisée entre en jeu. Cette fonction a une courbe plus proche du fonctionnement d'un cerveau humain que la fonction quadratique. En effet, un individu humain a tendance à corriger ses erreurs plus rapidement lorsque celles-ci sont flagrantes. Au contraire, plus l'erreur est petite, plus il est difficile de la corriger.

Cette fonction va exactement agir ainsi. Désormais, les changements sur le réseau de neurones sont donc plus conséquents lorsque les erreurs sont grandes, ce qui paraît plus logique et est plus efficace.

2.2.4 La fonction Softmax

Softmax est une fonction d'activation, à l'instar de la fonction sigmoïde. Voici sa formule :

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Elle s'applique uniquement à la dernière couche de neurones, la couche de sortie. On aurait donc comme résultat l'exponentielle de z divisée par la somme des exponentielles de tous les z de la couche finale. Nous pouvons rapidement comprendre l'utilité de l'utilisation d'une telle fonction : nous manipulons ici des exponentielles et donc le résultat est forcément positif. De plus, on peut remarquer que cette formule calcule une probabilité : la somme de tous les résultats en couche finale après application de la fonction softmax donne tout simplement 1. On pourrait donc rapidement voir quels sont les probabilités que voit l'ordinateur selon les différents résultats.

Nous avons donc la fonction softmax() dans le fichier ocr.c qui marche comme fonction d'activation. Cependant nous avons décidé de ne pas complètement implémenter la fonction softmax dans notre OCR, et ce, pour une seule raison : sa dérivation. Il est nécessaire pour l'utilisation de cette fonction de la dérivée, lors du processus de dérivation, mais la dérivée d'une telle fonction est très complexe à effectuer : nous avons une dérivée de la forme u/v avec u une exponentielle et v une somme d'exponentielle. Nous restons donc avec la fonction sigmoïde, qui fonctionne très bien.

2.2.5 La régularisation L2

Un problème connu qui intervient avec l'entraînement du réseau est le sur-apprentissage.

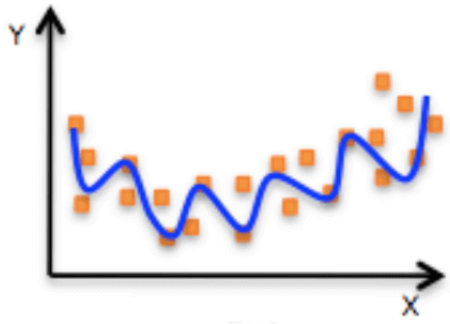


FIGURE 6 – Un exemple de sur-apprentissage

A force d'entraîner le réseau avec un jeu de données, le problème de sur-apprentissage peut intervenir. Celui-ci correspond au fait que les résultats collent trop au jeu d'entraînement. Ainsi, pour les données réelles qui ne ressemblent pas exactement à ceux du jeu d'entraînement, le réseau peut ne pas réussir à prédire le résultat. On préfère donc avoir un modèle plus simple qu'un modèle complexe, que l'on peut facilement reconnaître à la courbe comme sur le graphique ci-dessus.

La régularisation, comme nous, n'aime pas les modèles complexes, elle va donc nous aider à construire une courbe plus "lisse" et plus généralisée au problème auquel nous sommes confrontés. Il existe plusieurs méthodes de régularisation, parmi les plus connues, il y a "l'early stopping", qui consiste à arrêter l'apprentissage dès que le sur-apprentissage commence. Une autre est la régularisation L1, qui vise à "lisser" la courbe des résultats. Ce que nous avons implémentés est la régularisation L2, qui est une variante de la L1, plus simple à calculer et aussi généralement plus efficace. La régularisation n'influe pas la mise à jour des biais, mais celle des poids :

$$w \to \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m} \sum_x deltaW_x$$

Avec :

— η : learning_rate

— m : nombre de données du mini-batch

— n : nombre de de données dans le training set

— λ : facteur de régularisation

La formule ne change pas énormément à celle utilisée jusque là. En effet, ce qui change est l'apparition du facteur $\left(1 - \frac{\eta\lambda}{n}\right)$ qui est appliqué au poids avant de lui soustraire la somme des deltaW. Ce qui va changer l'efficacité de la régularisation va être le facteur λ. Plus ce facteur est élevé, plus la courbe est lisse, et inversement. On peut donc penser que pour combattre le sur-apprentissage il faut mettre une valeur très élevé mais c'est un piège : on peut ainsi obtenir du sous-entraînement et notre modèle deviendra finalement moins précis. Il faut alors trouver la bonne valeur pour la régularisation afin d'améliorer les résultats de notre réseau de neurones.

2.2.6 Initialisation des poids

L'initialisation des poids n'est désormais plus seulement régie par la fonction random_gaussian(). En effet, afin d'améliorer les performances du réseau, mieux initialiser les valeurs au départ est une piste. C'est ainsi que désormais nous initialisons nos valeurs :

$$w = random\_gaussian\left(0, \frac{1}{\sqrt{n_{input}}}\right)$$

Nous initialisons tout simplement en appliquant le random_gaussian entre 0 et l'inverse de la racine carrée du nombre total de poids du neurone en question. Cela permet d'initialiser les poids avec une meilleure approximation du résultat final. Nous obtenons ainsi des résultats plus rapidement, mais aussi une amélioration globale de la précision après l'entraînement.

2.2.7 Choix des valeurs des paramètres

Il y a tout plein de valeurs à décider pour le réseau de neurones. Parmi elles : le nombre de neurones de la couche cachée, le taux d'apprentissage, le facteur de régularisation, le nombre d'époques et la taille des batchs...

Pour chacune de ces valeurs, il nous a fallu effectuer des tests et en tirer des conclusions. Nous avons effectué les tests méthodiquement, en ne modifiant qu'un seul paramètre à la fois. Ainsi nous nous sommes rendu compte que le premier paramètre à fixer était le nombre de neurones de la couche cachée,

que nous avons fixée à 128. Moins de neurones rendait le réseau moins précis mais plus ne le rendait plus précis que de manière négligeable en comparaison avec le temps perdu. En effet, les paramètres comme le nombre de neurones et le nombre d'époques modifient la durée de l'entraînement.

Après de multiples tests, nous avons fixé le learning_rate à 0.2 et le facteur de régularisation à 20. Le nombre d'époque est 7, car au delà, l'entrainement n'améliore plus le réseau et la taille des mini-batchs est de 10.

2.3 Charger et sauvegarder les valeurs d'un réseau

Si nous ne voulons pas perdre la configuration (nombre de couches, nombre de neurones par couche, biais et poids) d'un réseau de neurone afin de le réutiliser, il est nécessaire de la sauvegarder dans un fichier et de pouvoir la charger depuis un fichier. C'est ici que les fonctions save_setup() et read_setup() entrent en jeu.

save_setup() sauvegarde la configuration d'un réseau de neurones dans un fichier. Celui ci est écrit dans le format suivant :

- La première ligne contient la variable nbLayer
- La seconde ligne contient le nombre d'input pour le réseau de neurones.
- Pour nbLayer lignes, on écrit le nombre d'output de chaque layer
- Pour tous les poids puis tous les biais de chaque couche, on écrit ligne par ligne les valeurs

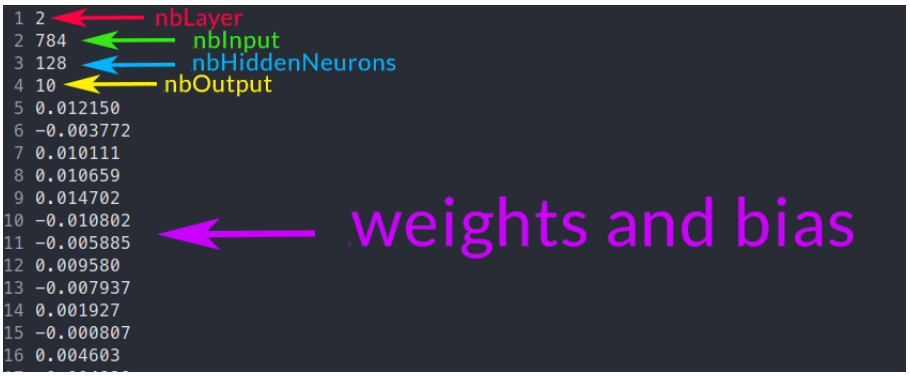


FIGURE 7 – Fichier des données d'un réseau de neurones après entraînement

La fonction read_setup() charge un réseau de neurones à partir d'un fichier dans le format utilisé par la fonciton save_setup() et le retourne. Nous avons remarqué, qu'étonnament les configurations d'un réseau de neurones

ne marche que sur une machine en particulier. Si l'on sauvegarde depuis un PC A un réseau de neurones et que l'on le charge sur ce même PC, alors le réseau de neurones fonctionne parfaitement. En revanche, en chargeant ce même réseau de neurones sur un PC B, les résultats sont biaisés et faux. Nous n'arrivons pas vraiment à expliquer ce phénomène, même si nous pensons que cela viens de la différence de composants entre deux ordinateurs.

3 Reconstruction de la grille

C'est ici que l'on fait le lien entre les deux premières parties : le traitement de l'image et le réseau de neurones. En effet, il est nécessaire d'utiliser les images prédécoupées par le pré-traitement afin de l'utiliser pour obtenir la grille du sudoku.

Nous avons alors la fonction main_ocr, présente dans le fichier éponyme qui prend en argument un pointeur de SDL_Surface, un struct de la librairie SDL, qui représente les 81 cases du sudoku.

Cette fonction reconstruit un réseau de neurones enregistré dans un fichier avec read_setup() puis lancer la fonction recover_grid() expliquée ci-dessous.

3.1 Le fichier recover_grid.c

Ce fichier contient 3 fonctions, dont une qui est utilisée par main_ocr. La première est recover_grid(), qui retourne la liste des chiffres obtenu à partir des SDL_Surface représentant les cases du sudoku. Il suffit de récupérer les pixels du SDL_Surface, de les mettre dans un pointeur de float d'une taille 784, puis de l'envoyer dans le réseau de neurones avec la fonction query(). Mais ici apparaît un problème : en effet, comment différencier les cases vides des cases non-vides ? Le réseau de neurones ne reconnaît pas les cases vides

Ici est utilisée la fonction is_blank() retournant un int fonctionnant comme un booléen : on prend les pixels situés au milieu de la case, si on détecte 5 pixels noirs ou moins alors la case est vide et on met un '0' dans le pointeur à l'index correspondant. Pourquoi tester 5 pixels ou moins ? Tout simplement car la binarisation n'étant pas parfaite, il peut rester du bruit au centre del'image. Il ne faut donc pas s'attendre à un centre parfaitement blanc.

4 Solver de Sudoku

4.1 Algorithme de résolution

Pour résoudre un sudoku, nous avons utilisé l'algorithme de backtracking. Nous remplissons la grille au fur et a mesure, lorsqu'on ne peut pas ajouter un nombre on revient en arrière. L'algorithme n'a pas été compliqué à mettre en place. Le plus dur a été de lire le fichier donné en entré ligne par ligne.

4.2 Sauvegarde

Une fois la solution du sudoku obtenue, nous devons la sauvegarder dans un fichier pour une utilisation future. Cette fonction modifie le nom du fichier ouvert en ajoutant, juste avant l'extension, la chaîne de caractères ".result", crée un nouveau fichier de ce nom, l'ouvre en mode écriture, y transcris la grille de sudoku résolue et ferme le fichier.

5 Interface Graphique

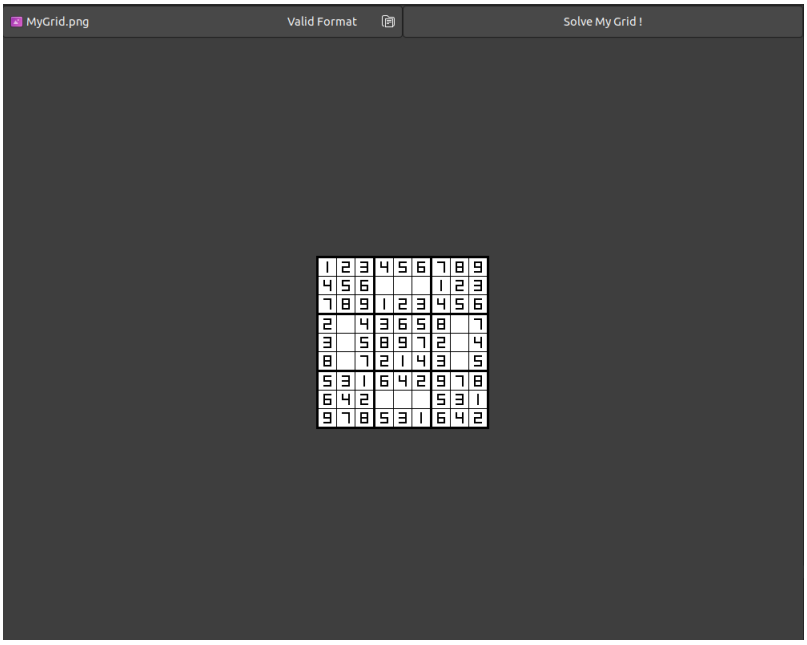


FIGURE 8 – Interface vue par l'utilisateur

L'interface graphique est la seule partie en contact avec l'utilisateur. Pour la construire, nous avons décidé d'utiliser Glade. Glade est un outil interactif

permettant de concevoir facilement une interface graphique Gtk. Cette application simplifie la création du visuel pour nous permettre de ne pas perdre de temps à tout créer. Nous avons fait le choix de garder une interface simple et épurée pour que l'utilisateur ne s'y perde pas.

Notre interface, consiste d'un FileChooser (élément permettant à l'utilisateur de choisir un fichier dans son arborescence), une zone de texte disposée sur le FileChooser, indiquant à l'utilisateur les informations qui lui sont utiles (Mauvais fichier choisis, Format valide, Traitement en cours, et autres secrets), un bouton "Solve My Grid !" permettant le lancement de la suite du projet sur le fichier choisi. De plus, une prévisualisation des fichiers choisis est aussi disponible. Cette interface marche comme décrit ci-dessous :

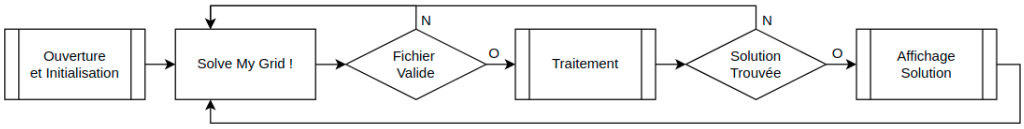


FIGURE 9 – Interface vue par l'utilisateur

L'interface suit un fonctionnement très précis : Si l'utilisateur ne choisi initialement pas de fichier ou choisi un mauvais fichier la zone de texte à côté du FileChooser affichera un message dirigé à l'utilisateur lui indiquant que le fichier n'est pas valide. Si l'utilisateur choisi de quand même d'envoyer un fichier qui n'est pas valide (non choisi ou différent d'une image possédant l'extension ".png" ou ".jpg") via le bouton "Solve My Grid !", la zone de texte affichera alors un message plus marqué pour que l'utilisateur choisissent un fichier valide. Le bouton "Solve My Grid !" n'appellera la suite du code que si le fichier choisi en entrée est une image valide. Dans ce cas, la zone de texte affichera un message indiquant que le traitement est en cours. Si, à la fin du traitement, une solution est disponible, elle sera affichée à la place de l'image que l'utilisateur avait fourni. Cependant, si l'image n'a pas de solution, alors la zone de texte sera de nouveau actualisée pour prévenir l'utilisateur de ce résultat.

6 Fonctionnement global du programme

Finalement, voici les différentes étapes par lesquelles passent notre programme afin de résoudre un sudoku à partir d'une image. Tout d'abord, l'utilisateur choisi dans l'interface une image d'un sudoku qu'il souhaite résoudre.

Cette image passe ensuite par diverses étapes lors du pré-traitement de l'image. Nous obtenons alors les 81 cases, découpées et binarisées. C'est alors que l'on va faire passer en entrée les 784 cases de chaque case, les unes après les autres, dans le réseau de neurones. Les cases blanches sont reconnues par la fonction is_blank(). Le réseau de neurones sera chargé à partir d'un fichier de configuration. Nous obtenons alors l'ensemble de la grille de sudoku grâce à la reconnaissance des chiffres.

Enfin, le solver entre et jeu et résout la grille de sudoku. L'orientation de la rotation automatique pouvant être mauvaise, si le premier essai de résolution ne marche pas, alors nous effectuons une rotation de 90° sur l'image et recommençons, jusqu'à trouver une solution, ou avoir fait les 4 orientations possible.

Une fois la solution trouvée, la grille de solution est assemblée et est affichée sur l'interface graphique !

7 Ressenti Personnel

7.1 Première partie du projet

7.1.1 Le ressenti des membres de l'équipe

Ce début de projet fut à mon avis, un bon départ. Un apprentissage approfondi du langage C et de ses librairies tel que SDL c'est retrouvé plus intéressant que prévu. Le passage d'un nombre d'heures non négligeable devant mon ordinateur fut très plaisant bien qu'exténuant de temps à autre. De plus, même si je n'ai pas participé à la création du réseau neuronal, les quelques sources transférées par Luc et Charles furent tout de même très intéressantes, bien que des fois compliquées à comprendre. J'espère que la partie graphique qu'est la suite de ce projet sera tout aussi intéressante que la précédente.

Victor Chartraire

Le début de ce projet a été pour moi plus compliqué que prévu. Tout d'abord pour l'utilisation de la libraire SDL et des pixels d'une image. De plus, l'implémentation d'une fonction de binarisation fut bien plus longue et frustrante que ce que je pensais. Prendre plusieurs heures à tenter d'implémenter une méthode de binarisation pour s'apercevoir que celle-ci n'est pas fonctionnelle est assez insatisfaisant. En dehors de ces désagréments, ce projet permet de découvrir plein d'aspects intéressant de la reconnaissance d'image, telle que le transformée de Hough ou l'algorithme d'Otsu. J'attends avec hâte mais aussi appréhension le commencement de la partie graphique du projet.

Garice Morin

La création du réseau de neurones a présenté son lot de difficulté. Il a fallut comprendre toute la documentation sur le sujet qui, étant en anglais, avait un certain nombre de notation qui ne m'était pas familière. La manipulation de pointeur en C est une chose délicate quand on se ne sait pas se qu'on fait. J'ai une fois alloué 10 000 fois le même pointeur parce que je n'avais pas pensé à le passer en paramètre à une fonction... Mais outre ces petits inconvénients, le C reste un langage très intéressant à utiliser. En somme cette première partie du projet m'aura permis de d'acquérir un peu d'aisance sur tout les outils nécessaire à la

bonne complétion du réseau final, qui sera, je l'espère, un peu plus optimisé que celui ci...

Charles Neyrand

Ce début de projet a été très intéressant pour ma part. En effet, il nous a fallu, en relativement peu de temps, faire des recherches sur quelque chose que nous ne connaissions pas, et l'appliquer avec un langage que nous ne connaissions pas non plus. Le C est un langage très différent de tous les autres langages étudiés jusqu'à aujourd'hui. C'est maintenant que l'on peut comprendre de quoi on parle quand il est question de langage bas niveau. De nombreux problèmes auxquels nous ne nous étions jamais confrontés ont fait leur apparition, nous pouvons penser aux fameux "Segfault" et aux problèmes liés à la gestion de la mémoire. Cependant , nous nous sommes adaptés à ce nouveau environnement de travail et c'est une nouvelle approche de la programmation que nous découvrons à travers ce projet. De plus, le concept de réseau de neurone a été très intéressant à comprendre et à mettre en pratique. Voir le XOR marcher de façon (presque) magique a été très satisfaisant !

Luc Mahoux

7.1.2 Conclusion de la première partie du projet

Le début de ce projet eut pour nous son lot de difficultés mais aussi et surtout fut extrêmement intéressant, que ce soit par les méthodes, algorithmes, concepts qu'il nous a appris ou bien par le langage en lui-même. Nous avons réussi à terminer le nécessaire pour la soutenance en temps et en heure, et le reste du projet s'annonce tout aussi passionnant que ne l'était cette première partie.

7.2 Deuxième partie du projet

7.2.1 Le ressenti des membres de l'équipe

Pour cette deuxième partie de projet, l'ambiance fut beaucoup plus pressante, dû au différents midterms, oraux et TPs. L'organisation de mon temps fut très importante et certains compromis furent décidés. Cependant, les différentes parties réussies de ce projet furent intéressantes. De plus, la communication ayant été bien faite dès le début, la mise en commun de code s'emboîtait presque toujours parfaitement et était toujours sans problèmes majeurs. Mais repasser sur mon code et trouver des solutions encore plus efficaces et surtout plus simples que précédemment était des plus satisfaisant.

En terme d'apprentissage, la création de l'interface graphique via l'utilisation de Glade et de la librairie gtk m'a permis de mieux comprendre certains TPs fournis sans beaucoup d'explication. Avec quelques connaissances supplémentaires à mon actif, le debogage des fonctions fut plus aisé que pendant la première partie de ce projet.

En général, je suis content du travail que j'ai pu fournir durant ce semestre court mais rempli et j'espère pouvoir continuer à coder des projets tout aussi intéressants.

Victor Chartraire

La deuxième partie du projet fut bien différente de la première. Tout d'abord, le temps à notre disposition était plus court, et combiner OCR, DM d'algo et TP d'IP ce révéla parfois un casse-tête. Cela dit, l'apparition, bien qu'un peu tardive d'une fonction de binarisation de bonne qualité m'apporta beaucoup de satisfaction personnelle. La fonction de rotation automatique fut un petit peu récalcitrante notamment à cause d'une fonction de séparation de lignes verticales et horizontales qui ne fonctionnait pas correctement. Mais la fierté de voir notre programme détecter de lui même la rotation d'une image et la corriger compense complètement ces mauvais points.

Quant à mon ressenti sur le projet en général, je dirais que celui-ci m'a beaucoup apporté en connaissance sur le C, les concepts d'imagerie et sur la résolution des "Memory leak" et autres "heap-buffer-overflow".

Garice Morin

Cette seconde partie du projet Ocr a été à la fois très fatiguante et très satisfaisante. Nous avions commis plusieurs erreur dans l'implémentation du réseau de neurones pour la première soutenances, et réparer ces erreurs à pris du temps. Toutefois quelle satisfaction de voir nos efforts récompensé et l'ocr fonctionné. Ce projet m'aura permis de m'améliorer grandement dans l'utilisations de plusieurs concepts de programmation, de l'utilisations de makefile à l'allocation dynamique de la mémoire. L'utilisation d'outil comme valgrind a été aussi ytrès enrichissante.Je ne désepère pas un jour de rendre un dépot git propre et bien organisé. De plus quitte à ne pas savoir comment éviter les conflits de fusion de git, je sais mieux comment les résoudre maintenant. En somme ce projet restera un bon souvenir.

Charles Neyrand

Le projet OCR a été très enrichissant pour chacun des membres de notre groupe. Personnellement je comprends désormais le fonctionnement des réseaux de neurones et même s'il reste encore énormément à apprendre dans ce domaine, je suis satisfait d'en avoir appris les bases et je vais sûrement me plonger plus dans ce monde afin d'en apprendre davantage. Il n'y avait rien de plus satisfaisant que de voir notre réseau reconnaître les nombres parfaitement ! De plus, le projet m'a permit de me familiariser avec l'éditeur vim, le langage C, l'allocation de mémoire et même quelques concepts du prétraitement de l'image de Garice et de Victor. J'ai beaucoup apprécié ce projet, plus technique que les précédents, et j'ai hâte de me relancer dans d'autres du même genre.

Luc Mahoux

8 Conclusion du projet

Ce projet fut pour nous un défi très intéressant à relever. D'un point de vue des connaissances d'abord. Lorsque nous commencions l'écriture du programme et de ses fonctions, nous ne connaissions pas ou seulement très peu le langage C. Et pour ne rien arranger, celui-ci est très différent de tous les langages que nous avions vu jusqu'ici à l'EPITA. Il nous a donc fallu apprendre sur le tas. Les erreurs telles que les *segmentation fault* ou *memory*

leak nous donnèrent du fil à retordre, notamment au début du projet. Malgré ces difficultés, nous sommes parvenus à rendre une partie de projet répondant au cahier des charges à la première soutenance.

Arriva alors la deuxième partie du projet qui s'avéra plus compliqué que la première. Non pas plus compliqué en termes de quantité de travail, mais plutôt d'organisation de celui-ci. La deadline étant beaucoup plus proche que la précédente, le rythme de travail était bien supérieur à la première partie du projet. Nous devions modifier le réseau neuronal afin que celui-ci puisse reconnaître des nombres, trouver un moyen de corriger la rotation d'une image automatiquement et créer une interface graphique. Après de nombreuses heures de travail, de jour comme de nuit, nous avons réussi à livrer un OCR fonctionnel pour cette soutenance finale.

Nous sommes tous fiers du travail que nous avons accompli, en temps que groupe, mais aussi personnellement. Nous avons pu découvrir quelques domaines intéressant de la programmation tels que le traitement d'image, les réseaux neuronaux et les interfaces graphiques avec GTK.

9 Bibliographie

— Outils pour l'écriture des mathématiques en LaTeX. *Zeste de savoir* [en ligne]. (Mis à jour le 01 Août 2019). [Consulté le 25 Octobre 2021]. Disponible à l'adresse : https://zestedesavoir.com/tutoriels/409/outils-pour-lecriture-des-mathematiques-en-latex/

— Make Your Own Neural Network - *Tariq Rashid* [2016]

— Backpropagation calculus | Deep learning - *3 Blue 1 brown* [novembre 2017]. Disponible à l'adresse : https://youtu.be/tIeHLnjs5U8?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

— Neural Networks and Deep Learning [en ligne] [Consulté en Octobre 2021]. Disponible à l'adresse : http://neuralnetworksanddeeplearning.com/index.html

— Documentation de la librairie SDL. Disponible à l'adresse : https://wiki.libsdl.org/

— Lines Detection with Hough Transform. [en ligne] Disponible à l'adresse : https://towardsdatascience.com/lines-detection-with-hough-transform-84020b3

— A New Local Adaptive Thresholding Technique in Binarization. [en ligne] Disponible à l'adresse : https://arxiv.org/ftp/arxiv/papers/1201/1201.5227.pdf

— Adaptive Thresholding Using the Integral Image. [en ligne] Disponible à l'adresse : https://people.scs.carleton.ca/~roth/iit-publications-iti/docs/gerh-50002.pdf