Projet OCR

EPITA 2ème Année Cycle Préparatoire

Rapport de Première Soutenance

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

Octobre 2021

Table des matières

Introduction

Le projet OCR comporte deux grandes parties. La gestion 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. Garice et Victor sont responsables de la gestion de l'image, Luc et Charles de la création du réseau de neurones. Pour cette première soutenance, la transformation de l'image en niveau de gris, la binarisation, la rotation de l'image, la détection, le découpage et le redimensionnement de la grille ont été implémenté. De même un réseaux de neurone capable d'apprendre la fonction XOR à été réalisé. Nous avons aussi implémenté un algorithme de résolution de sudoku. Toutes ces étapes sont détaillés dans la suite de ce rapport.

Tâche	Charles	Garice	Luc	Victor
Greyscale		X		
Binarisation		X		
Rotation de l'image				X
Détection de la grille		X		
Découpage de la grille				X
Redimensionnement				X
Solver	X			
Implémentation Réseau	X			
Donnée d'entrainement	X			
Propagation de l'erreur			X	
Mise à jour des poids/biais			X	
Sauvegarde/chargement du réseau			X	

1 Traitement d'images

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

L'objectif pour cette première soutenance était de pouvoir fournir une image, la binariser, la pivoter d'un angle donné par l'utilisateur, détecter la grille de sudoku sur l'image, l'extraire, et pour finir, découper chaque case faisant partie de la grille et les redimensionner pour les envoyer au réseau neuronal.

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

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 cette 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.

Bien que plus performante que la méthode précédente, le résultat n'est toujours pas satisfaisant notamment sur les images ayant un faible contraste. Pour la suite du projet, nous allons implémenter une autre méthode, la méthode de Wellner.

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 pertes, 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.

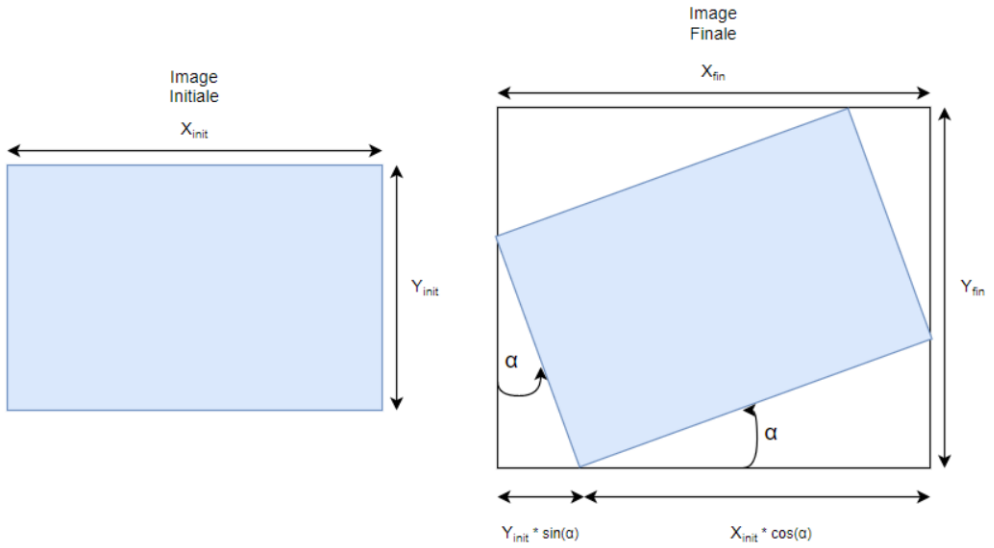

FIGURE 1 – Calcul des dimensions d'une image pivotée d'un angle α

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 :

$$P' = P \times M \tag{1}$$

- 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 :

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

$$(1) <=> \begin{cases} \text{x'} = \text{x}.\cos(\alpha) + \text{y}.\sin(\alpha) \\ \text{y'} = \text{x}.\text{-}\sin(\alpha) + \text{y}.\cos(\alpha) \end{cases} \tag{2}$$

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 Détection de la grille

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 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 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 :

$$r = \frac{côté_{init}}{côté_{fin}}$$

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.

2 Réseaux de neurones

Comme dit précédemment, Charles et Luc ont été en charge de la partie concernant les réseaux de neurones.

Notre but pour cette soutenance était de réussir à implémenter un réseau de neurones reconnaissant le "OU Exclusif", ou plus simplement "XOR". Nous avons choisi pour la structure d'utiliser trois couches de neurones dont une cachée. La couche d'entrée a deux neurones. La couche cachée en compte huit. La sortie en 1 un seul. La fonction d'activation que nous avons utilisé est la fonction sigmoïde pour toute les couches. Nous avons choisis de ne pas implémenter Softmax pour la dernière couche car notre réseau ne possède qu'un seul neurone sur celle ci. Cette fonction d'activation sera implémentée pour la soutenance finale.

Notre démarche s'est divisé en trois parties : comment implémenter le réseau proprement dit, comment le questionner et comment l'entraîner.

2.1 Représenter un réseau de neurones

Un réseau de neurone est constitué de deux composantes principales. Une liste de couches et un taux d'apprentissage. 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 liaison 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'autre qui les détruisait.

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'erreur 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

2.2 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 du créer est "Query". Cette fonction prend deux données en entrée et elle écrit le résultat. Ces données sont comprises entre 0 et 1. Dans le cas de du XOR, nous donnerons a la fonction une parmi les quatre combinaisons possible de couple de 0 et 1. Cette fonction appelle "FeedForward", qui est aussi utilisé 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 a la matrice de résultat et nous appliquons la fonction sigmoïde à tout 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.

2.3 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 représenté 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.

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 rétropropagation, 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 "epochs" : 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.

2.3.1 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. Dans notre programme, cela va être fait avec la fonction OutputError 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 sigmoide
— $\sigma'(z)$: dérivée de la fonction sigmoide 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 neurone 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.

2.3.2 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 trouvée à la layer l+1 afin de trouvée 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 "BackpropagateError" qui va propager dans tout le réseau de neurones les matrices erreurs à partie des erreurs trouvées par OutputError. Cette fonction va être appelée une seule fois par entrée, comme la fonction OutputError.

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 "Backpropagation" qui a pour argument le réseau de neurone à entraîner et l'input, est utilisée afin d'appeler les fonctions FeedForward, OutputError et BackpropagateError, 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 epoch) :

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

Nous sommes presque arrivé au moment de mettre à jour le réseau de neurones. Il reste encore quelques lignes à lire !

2.3.3 Mettre à jour les biais et les poids

Jusque là, nous avons expliqué le fonction Backpropagation 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 epochs afin d'entraîner le réseau de neurone.

Pour pouvoir entraîner un réseau de neurones il faut propager l'erreur créé à partir de toutes les données d'entraînement. Une fois que le réseau s'est entraîné sur toute 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 la fonction backpropagation.

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 "epoch". Nous n'avons plus qu'a diviser les valeurs de deltaW et deltaB par le nombre d'entrées de l'epoch, 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'epoch :

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

Voilà ! Il ne reste plus qu'à effectuer cette opération plusieurs milliers de fois afin d'obtenir des résultats satisfaisant. Les résultats vont petit à petit tendre vers les résultats espérés. Il est bien évidemment possible d'améliorer les résultats et faire en sorte de les obtenir plus rapidement, mais nous verrons cela pour la soutenance finale.

2.4 Charger et sauvegarder les valeurs d'un réseau

Si nous ne voulons pas perdre la configuration (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 SaveSetup et ReadSetup entrent en jeu.

SaveSetup va sauvegarder la configuration d'un réseau de neurones dans un fichier. Celui ci sera du format suivant :

— Nombres espacés sur la première ligne représentant le nombre de neurones par couche
— Une ligne contenant des nombres à virgules avec une précision de 6 chiffres après la virgule, représentant les poids d'une couche
— une ligne contenant des nombres à virgules avec une précision de 6 chiffres après la virgule, représentant les biais d'une couche
— Répéter les deux dernières lignes décrites pour chaque couche du réseau.

```
[epita@archlinux NeuralNetwork]$ cat neuralafter
2 8 1
3.5018 -0.4409 7.8874 2.8898 2.5869 -2.9398 2.6746 1.6388 -13.7094 -0.0966 -7.0968 2.4333 1.6620 -12.6158 2.3795 1.8630
0.4540 -1.3983 -1.8521 -3.9354 -2.7758 -5.4589 5.7884 -3.8243
-5.1880 4.4168 7.1904 -4.3438 -0.0890 -4.7925 -2.5433 -2.6405
1.9605
```

Figure 2 – Contenu d'un fichier contenant les données d'un réseau de neurones après entraînement

ReadSetup n'est pas encore finalisée, mais sera rapidement implémentée après la soutenance. Son principe sera de parser un fichier du même format que décrit ci-dessus et de retourner un réseau de neurones correspondant.

3 Solver de Sudoku

3.1 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.

3.2 Sauvegarde

Une fois la solution du sudoku obtenue, nous devons la sauvegarder dans un fichier pour 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.

4 Programme pour la soutenance finale

D'ici la prochaine soutenance nous comptons améliorer le traitement du noir et blanc. L'objectif de la prochaine soutenace est d'avoir un produit fini, ce qui implique la création d'un réseau de neurones capable de reconnaître des chiffres. Nous pensons utiliser la base de données MNIST pour entraîner notre réseau de neurones. Tout le traitement de la grille et la création de l'interface graphique sont aussi des priorité.

5 Ressenti Personnel

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'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

6 Conclusion

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 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. Disponible à l'adresse : https://towardsdatascience.com/lines-detection-with-hough-transform-84020b3b1549

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