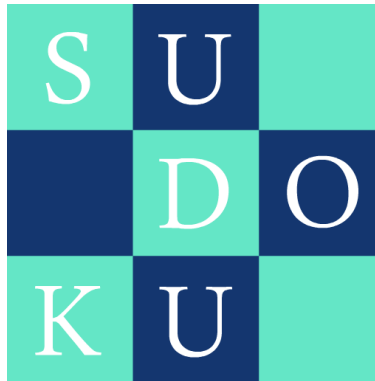


# Rapport de Soutenance finale

Décembre 2021



Nicolas Reyland  
Lilian Schall  
Théo Schandel  
Ahmed Hassayoune

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Répartition des charges</b>	<b>4</b>
<b>3</b>	<b>Interface utilisateur</b>	<b>5</b>
3.1	État d'avancement . . . . .	5
3.2	Aspects techniques . . . . .	6
<b>4</b>	<b>Traitement d'images</b>	<b>7</b>
4.1	État d'avancement . . . . .	7
4.1.1	Niveau de gris (Grayscale) . . . . .	7
4.1.2	Normalisation (Renforcement des contrastes) . . . . .	7
4.1.3	Floutage (Blur) . . . . .	7
4.1.4	Binarisation . . . . .	7
4.1.5	Rotation . . . . .	8
4.1.6	Reconstruction de la grille de sudoku . . . . .	8
4.2	Aspects techniques . . . . .	8
<b>5</b>	<b>Détection de la grille</b>	<b>13</b>
5.1	État d'avancement . . . . .	13
5.1.1	Segmentation . . . . .	14
5.1.2	Isolation de la grille . . . . .	15
5.1.3	Détection des coins de la grille . . . . .	15
5.1.4	Rotation et redressement automatique . . . . .	16
5.1.5	Découpage des cases . . . . .	17
<b>6</b>	<b>Réseaux de neurones</b>	<b>17</b>
6.1	Aspects techniques . . . . .	18
6.1.1	Description de forme et de dimension . . . . .	20
6.1.2	Génération d'un modèle . . . . .	21
6.1.3	Chargement des données . . . . .	22
6.1.4	Entraînement d'un modèle . . . . .	23
6.1.5	Fonctions d'activation et leurs dérivées . . . . .	24
6.1.6	Session d'entraînement et de test . . . . .	25
6.1.7	Fonction de coût . . . . .	25
6.1.8	Sauvegarde et Chargement d'un modèle . . . . .	26
6.1.9	Utilisation du modèle . . . . .	27
6.2	Aspects techniques - Annexe . . . . .	27
6.2.1	Conversion de cellules en objet numérique . . . . .	27
6.2.2	Logging . . . . .	29
6.2.3	Manipulation de bases de données . . . . .	30
6.2.4	Verbosity . . . . .	40
6.2.5	Entraînement générique et tests spécialisés . . . . .	40
6.2.6	Calibrage des modèles . . . . .	41
6.3	Conclusion - Réseaux de neurones . . . . .	44
6.4	Le solveur de sudoku . . . . .	46
6.4.1	Aspects techniques . . . . .	46
<b>7</b>	<b>Aspects techniques - Autres</b>	<b>46</b>
7.1	Prévention de fuite mémoire . . . . .	46
7.2	Bourne Again Test Framework . . . . .	47



# 1 Introduction

Notre groupe est composé d’Ahmed Hassayoune, Théo Schandel, Nicolas Reyland et Lilian Schall. Nous nous sommes répartis les tâches imposées par le cahier des charges de la manière suivante :

- Interface utilisateur → Ahmed Hassayoune
- Traitement d’image → Ahmed Hassayoune
- Détection de la grille → Théo Schandel
- Réseaux de neurones → Nicolas Reyland
- Réseaux de neurones → Lilian Schall

# 2 Répartition des charges

	Tâches	Nicolas.R	Lilian.S	Théo.S	Ahmed.H
Interface utilisateur	Chargement d’images				✓
	Gestion des widgets				✓
	Gestion des boutons et des entrées				✓
Traitement d’images	Niveau de gris et Normalisation (contraste)				✓
	Floutage				✓
	Binarisation				✓
	Rotation d’images				✓
	Reconstruction de la grille				✓
Détection de grille	Segmentation			✓	
	Isolation de la grille			✓	
	Detection des coins			✓	
	Rotation et redressement automatique			✓	
	Découpage des cases			✓	
Réseaux de neurones	Neurone	✓			
	Couche du modèle	✓			
	Description de forme	✓	✓		
	Modèle (réseau de neurones)	✓			
	Chargement de données		✓		
	Fonction d’activations et leurs dérivées		✓		
	Session d’entraînement et de test		✓		
	Fonction de coût		✓		
	Solver de Sudoku		✓		
	Feed Forward	✓			
	Back Propagation	✓			
	Mise à jour des poids	✓			
Annexe	Prévention de fuite mémoire	✓			
	Framework de tests unitaires	✓			

## 3 Interface utilisateur

### 3.1 État d'avancement

La bibliothèque utilisée pour créer l'interface utilisateur de ce projet est GTK 3<sup>1</sup>. L'interface graphique créée contient diverses fonctionnalités pour permettre l'utilisateur d'interagir avec le programme. Dès l'exécution du logiciel, l'utilisateur peut charger l'image à traiter à travers le bouton « Load image » qui va faire apparaître une boîte de dialogue lui permettant de naviguer à travers ses dossiers et de choisir l'image voulue. Une fois l'image choisie, elle est directement affichée dans la partie supérieure de l'interface graphique. L'utilisateur peut aussi voir les différentes étapes du traitement de l'image entrée à travers les trois boutons « Grayscale image », « Blurred image » et « Binarised image » situés dans la partie inférieure de l'interface graphique.

Voici une image de l'interface graphique :

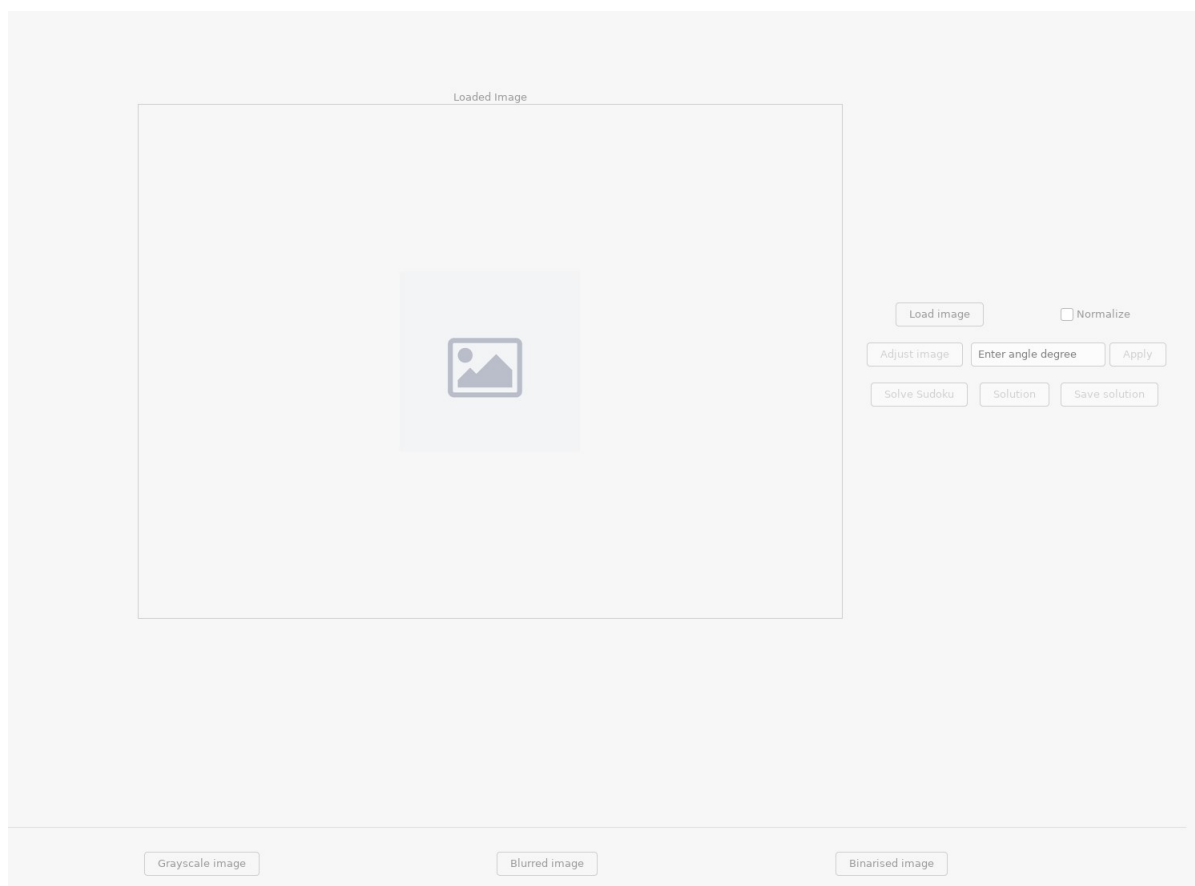


FIGURE 1 – Apparence de l'interface graphique.

---

1. GTK est un ensemble de bibliothèques logicielles permettant de réaliser des interfaces graphiques.

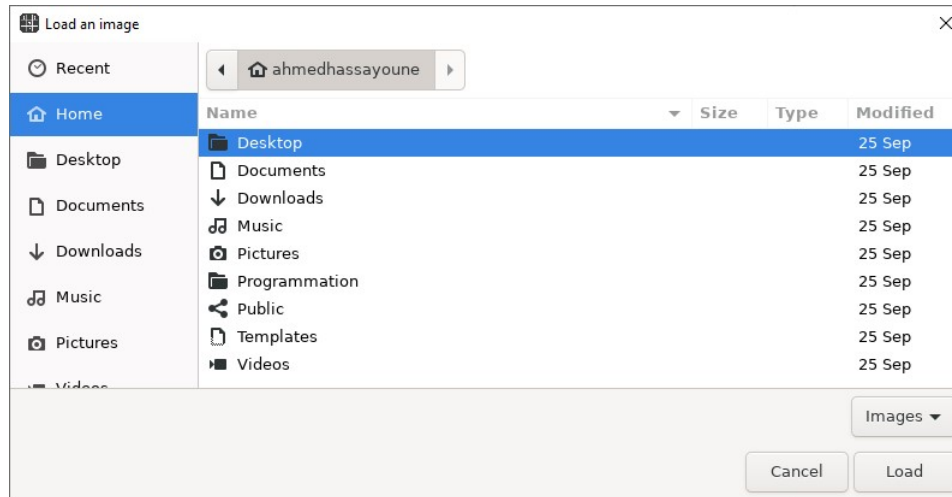


FIGURE 2 – Apparence de la boîte de dialogue.

### 3.2 Aspects techniques

L'interface graphique est composée :

- D'un champ (frame) qui affiche l'image chargée
- De neuf boutons :
  - « Load image » qui permet de charger une image ;
  - « Adjust image » qui redresse l'image (suppression de perspective) ;
  - « Apply » qui permet d'appliquer la rotation de l'image avec un angle préalablement saisi ;
  - « Solve Sudoku » qui lance tous les traitements afin de résoudre la grille de sudoku et affiche ensuite une reconstruction de celle-ci contenant les chiffres détectés depuis l'image chargée ;
  - « Solution » qui affiche une reconstruction de la grille de sudoku contenant la solution ;
  - « Save solution » qui sauvegarde la reconstruction de la grille de sudoku résolue sous le format « PNG » ;
  - « Grayscale image » qui affiche l'image entrée en niveau de gris ;
  - « Blurred image » qui affiche l'image en niveau de gris floutée par une matrice de convolution (méthode de Gauss) ;
  - « Binarised image » qui affiche une binarisation de l'image floutée avec la méthode de Sauvola.
- D'une case à cocher qui permet de normaliser l'image entrée.
- D'une entrée qui permet de saisir l'angle de rotation en degré, à appliquer à l'image chargée.

La boîte de dialogue utilisée dans l'interface graphique est de type « File/Open »<sup>2</sup>. Un filtre lui a été ajouté afin de ne détecter que les fichiers contenant l'extension .png, .jpg, .jpeg et .bmp.

<sup>2</sup>. Ce type permet seulement de sélectionner un fichier à ouvrir.

## 4 Traitement d'images

### 4.1 État d'avancement

L'ordre qui a été choisi pour le traitement d'images est le suivant : L'image est tout d'abord transformée en niveau de gris, puis un renforcement des contrastes (normalisation de l'image) est appliqué si l'utilisateur le souhaite. Par la suite, un floutage est appliqué à l'image générée et finalement celle-ci subie une binarisation.

#### 4.1.1 Niveau de gris (Grayscale)

Afin d'avoir une bonne segmentation pour pouvoir détecter la grille de sudoku, il faut supprimer les informations inutiles et redondantes de l'image. Les couleurs de l'image à traiter ne sont pas utiles dans notre cas. L'image est donc tout d'abord transformée en niveau de gris afin de supprimer les couleurs, mais aussi afin de se ramener à l'étude d'une unique valeur pour chaque pixel pour le reste du traitement total.

#### 4.1.2 Normalisation (Renforcement des contrastes)

Si l'image entrée possède un faible contraste alors l'utilisateur peut demander au programme d'inclure une normalisation de l'image dans la procédure de traitement. Normaliser une image consiste à appliquer une transformation de son histogramme afin d'étendre la plage de valeur de l'image à l'ensemble des valeurs disponibles. Ceci permettra d'isoler la grille de sudoku et donc de faciliter sa recherche dans l'image.

#### 4.1.3 Floutage (Blur)

Après avoir transformé notre image en niveau de gris, on lui applique un filtre passe-bas pour la rendre plus lisse et donc éviter d'avoir les effets de changement d'intensité brutal de pixels et ainsi enlever les bruits tout en gardant la majorité de l'image intacte. Pour cela, un floutage de Gauss optimisé avec une matrice de convolution à une seule dimension est appliqué horizontalement puis verticalement à l'ensemble des pixels de l'image.

#### 4.1.4 Binarisation

Pour localiser la grille de sudoku, des opérations morphologiques<sup>3</sup> seront appliquées à la suite du traitement total de l'image entrée. Ces opérations fonctionnent généralement sur des images binarisées. Ainsi, une binarisation de l'image floutée est réalisée en utilisant un seuillage local<sup>4</sup>. La méthode de Sauvola est considérée comme une bonne technique de binarisation selon les résultats publiés dans différents articles. En effet, elle n'est pas sensible aux bruits et aux variations de lumières dans l'image. Une optimisation de cette méthode a donc été utilisée pour ce projet.

---

3. La morphologie est une théorie et technique mathématique et informatique d'analyse de structures qui fournit en particulier des outils de filtrage, segmentation, quantification et modélisation d'images. [Source](#)

4. Il s'agit d'un seuil qui est propre à chaque pixel.

#### 4.1.5 Rotation

En ce qui concerne la rotation de l'image à partir d'un angle quelconque, une combinaison de la matrice de rotation (2D) et de trois matrices de transvection a été implémentée. L'utilisateur peut donc saisir un angle exprimé en degré pour appliquer une rotation de l'image chargée.

#### 4.1.6 Reconstruction de la grille de sudoku

Pour ce qui est de la reconstruction de la grille de sudoku, le résultat est directement écrit sur une image par défaut à l'aide des fonctions de la bibliothèque SDL. Afin de rendre la solution plus lisible, les chiffres nouvellement ajoutés sont écrits d'une couleur différente.

### 4.2 Aspects techniques

La formule utilisée pour transformer l'image en niveau de gris est celle de la luminance qui est basée sur la perception de l'œil :

$$P = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

Concernant le renforcement des contrastes de l'image, la normalisation de son histogramme consiste à étendre la plage de valeurs sur tout l'intervalle possible<sup>5</sup>. La formule utilisée pour la normalisation de l'image est la suivante :

$$I = 255 \times \frac{(i - m)}{(M - m)} \quad (1)$$

- $I$  représente la nouvelle valeur de l'intensité du pixel ;
- $i$  représente l'ancienne valeur de l'intensité du pixel ;
- $m$  représente l'intensité minimale de l'image ;
- $M$  représente l'intensité maximale de l'image.

Voici deux exemples de normalisation dont l'un est la dernière image donnée dans le cahier des charges :

---

5.  $Vmin$  et  $Vmax$  sont respectivement les bornes de l'intervalle possible.  $Vmin$  (resp.  $Vmax$ ) représente la valeur de l'intensité minimale (resp. maximale) de l'image.



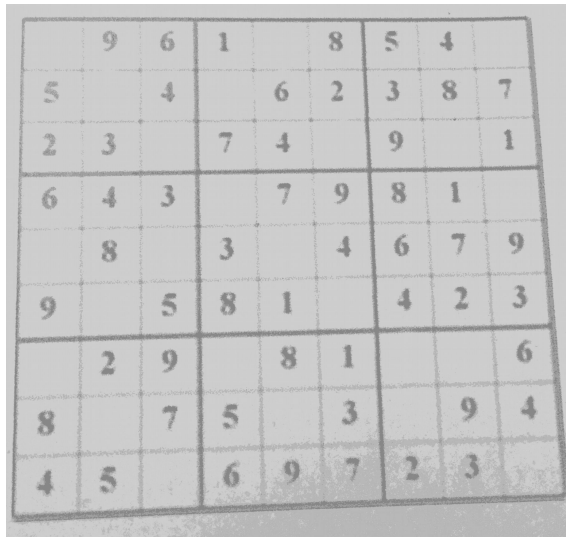


FIGURE 3 – Image en niveau de gris avec un faible contraste.

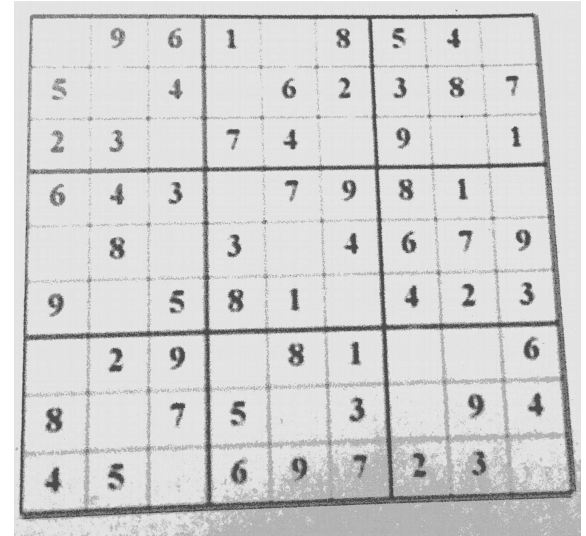


FIGURE 4 – Résultat de la normalisation sur l'image en niveau de gris.



FIGURE 5 – Image non normalisée. [Source](#)



FIGURE 6 – Résultat de la normalisation de l'image. [Source](#)

Pour le floutage de l'image, une optimisation a été ajoutée en réalisant une double convolution - horizontale puis verticale - sur chaque pixel de l'image en niveau de gris avec la matrice à une dimension suivante :

$$(0.006 \quad 0.061 \quad 0.242 \quad 0.383 \quad 0.242 \quad 0.061 \quad 0.006) \quad (2)$$

Ancienne matrice de convolution (2D) utilisée :

$$\frac{1}{273} \begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix}$$

La matrice (2) utilisée est ici une moyenne pondérée des pixels voisins qui donne plus de poids pour ceux qui sont proches du pixel étudié<sup>6</sup>.

En ce qui concerne la binarisation, c'est cette fonction de la méthode de Sauvola qui est utilisée :

$$T = m \times \left[ 1 + k \times \left( \frac{s}{R} - 1 \right) \right]$$

- $T$  représente le seuil du pixel ;
- $m$  représente la moyenne du pixel ;
- $s$  représente l'écart type du pixel ;
- $k$  est une constante appartenant à l'intervalle  $[0.2, 0.5]$  ;
- $R$  représente la valeur maximale de l'écart type pour une image en niveau de gris.

Cette fonction nécessite le calcul de la moyenne et de l'écart type en parcourant une fenêtre de pixels voisins de chaque pixel étudié. L'implémentation classique pour le calcul de ces variables peut devenir très long si la taille de la fenêtre est de plus en plus grande. Une optimisation a donc été appliquée en utilisant le principe des images intégrales<sup>7</sup> pour réduire la complexité du calcul en un temps constant. En contre-partie, cette méthode impose une plus grande utilisation de la mémoire.

Pour ce qui est de la rotation, la formule utilisée est une combinaison de la matrice de rotation (2D) et de trois matrices de transvections.

$$\begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \quad (\text{Matrice de rotation 2D})$$

$$\begin{pmatrix} 1 & -\tan \theta/2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \sin \theta & 1 \end{pmatrix} \begin{pmatrix} 1 & -\tan \theta/2 \\ 0 & 1 \end{pmatrix} \quad (\text{Matrices de transvection 2D})$$

---

6. Le pixel central.

7. Une image intégrale est une représentation sous la forme d'une image numérique, permettant de calculer rapidement des sommes de valeurs dans des zones rectangulaires. [Source](#)

Contrairement aux matrices de transvections, le résultat de la matrice de rotation (2D) sur une image peut causer des problèmes d'aliasing pour certains angles.



FIGURE 7 – Problème d'aliasing avec la matrice de rotation 2D.  
[Source](#)



FIGURE 8 – Correction du problème d'aliasing avec les matrices de transvection 2D.  
[Source](#)

Cependant, il n'est pas possible de seulement utiliser les matrices de transvection pour la rotation puisque l'angle  $\pi$  donne une forme indéterminée dans la formule. C'est pour cela qu'une combinaison de ces deux méthodes est utilisée dans ce projet.

Finalement, pour la reconstruction de la grille, une police<sup>8</sup> a été choisie pour écrire les chiffres dans une image contenant une grille de sudoku vide. Tout d'abord, deux surfaces ont été générées pour chaque chiffre de 1 à 9 dont l'une est en noir et l'autre en vert à partir de la fonction `TTF_RenderText_Blended()` de la SDL. Ces surfaces sont ensuite copiées à l'aide de la fonction `SDL_BlitSurface()` dans l'image contenant la grille de sudoku vide à partir des informations extraites au fil du traitement.

Voici l'image par défaut utilisée pour la reconstruction d'une grille de sudoku :

---

8. C'est la police `arial_narrow_7.ttf` qui a été utilisée.

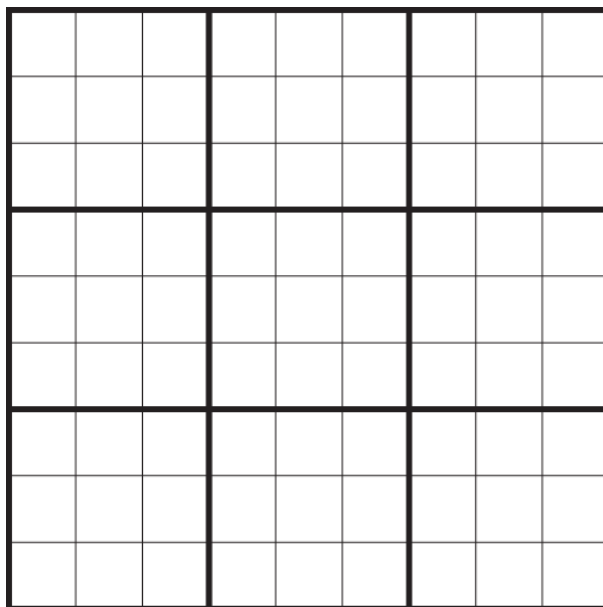


FIGURE 9 – Image utilisée par défaut pour la reconstruction d’une grille de sudoku. [Source](#)

Et voici à quoi ressemble le résultat de la reconstruction d’une grille de sudoku donnée :

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

FIGURE 10 – Reconstruction d’une grille de sudoku non résolue.

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

FIGURE 11 – Reconstruction d’une grille de sudoku résolue.

## 5 Détection de la grille

Lors de la première soutenance nous avons expliqué notre raisonnement pour détecter la grille, celui-ci était le suivant :

Premièrement nous avons fait la supposition suivante : La grille est normalement l'élément qui possède le plus de pixels dans l'image, nous avons donc segmenté l'image afin de différencier chaque groupe de pixel. Un groupe de pixel étant un ensemble de pixel où chaque pixel du groupe peut être relié à n'importe quel autre pixel de ce groupe en parcourant les pixels de l'image cote a cote sans rencontrer de pixel noir. Sur l'image ci-dessous, chaque groupe de pixel est différencié par sa couleur.



FIGURE 12 – Exemple de groupes de pixel

Puis nous avons pris l'élément possédant le plus de pixel dans l'image et déterminé celui-ci comme étant la grille de sudoku. Comme celle-ci était supposée être droite, il nous suffisait de parcourir l'image de haut en bas jusqu'à atteindre la grille pour déterminer la coordonnée supérieure de la grille, on fait de même pour les coordonnées inférieure, gauche et droite. Une fois ces coordonnées obtenues, on découpe l'image et on divise celle-ci par 9 en hauteur et 9 en largeur pour sauvegarder chacune des cases du sudoku avant qu'elles soient traitées par le réseau de neurone.

### 5.1 État d'avancement

La méthode de détection que nous utilisons pour l'application finale est similaire à celle décrite au-dessus. Mais nous avons ajouté des fonctionnalités et amélioré certaines des fonctions existantes. Voici les modifications apportées à la méthode utilisée lors de la première soutenance :

- La fonction de segmentation a été modifiée pour passer d'une implémentation réursive a une implémentation itérative. (Explications détaillés dans la partie dédiée plus bas)
- Pour avoir les coordonnées de la grille même lorsque celle-ci n'est pas droite nous utilisons la méthode de Manhattan après avoir inscrit la grille dans un carré droit

avec la méthode précédente, ainsi nous obtenons les coordonnées de chaque coin. (Plus d'explication dans la partie dédiée à la détection des coins plus bas)

- Nous ne prenons plus l'élément qui possède le plus de pixel mais l'élément qui couvre la plus grande surface. (Surface du quadrilatère dont les coins sont les coordonnées trouvées dans la fonction de détection des coins) Utiliser la surface plutôt que le nombre de pixel permet d'éliminer les éléments monochromes avec une forte densité de pixel tels que les cadres. (Exemple avec l'image 5 du cahier des charges)
- Nous avons pu, grâce aux coordonnées obtenue, faire une rotation automatique ainsi qu'un redressement de l'image pour enlever la perspective et ainsi avoir une image parfaitement carrée pour sauvegarder les cases de la grille. (Explications sur la rotation automatique dans la partie dédiée plus bas)
- Enfin la fonction de sauvegarde des cases a été améliorée pour que les images sauvegardées soient de taille 28x28 et que le chiffre soit centré. (Explications dans la partie dédiée plus bas)

#### 5.1.1 Segmentation

Lors de la première soutenance la segmentation des différents éléments de l'image était réalisée grâce à une fonction récursive, celle-ci posait un problème sur des grandes images car le nombre d'appel récursif très important provoquait un stack overflow. Nous avons donc annoncé que pour la soutenance finale, nous aurions implémenté cette fonction itérativement. C'est ce que nous avons fait. La fonction que nous avons implémentée est celle nommée « Two pass segmentation » en anglais.

La segmentation fonctionne en faisant deux parcours de l'image, d'où le nom two pass segmentation, le premier passage se fait colonne par colonne et labelise chaque pixel blanc en fonction de ceux se trouvant au-dessus de lui et à sa gauche :

- Si aucun des deux n'est labelisé, alors on crée un nouveau label qu'on donne à ce pixel, puis on stocke ce label dans une liste
- Si un seul des deux pixels est labelisé, on donne le label du pixel labelisé au pixel sur lequel on se trouve dans le parcours
- Si les deux pixels sont labélisés, on donne le label le plus bas (parmi les deux pixels labelisés) au pixel sur lequel on se trouve dans le parcours, puis on fait pointer le label le plus grand vers le label le plus petit dans la liste (de manière que lorsque l'on veut accéder au label le plus grand, on reçoit la valeur du label le plus petit.

Le second passage permet de rassembler les labels faisant parti du même ensemble. Pour chaque pixel de l'image on accède à son label dans la liste. (Celui-ci pointera vers le plus petit label de l'élément dont il fait partie)

Note : l'implémentation réelle est un peu plus compliquée et nécessite l'utilisation de structures de données tels que des ensembles disjoints, nous n'avons pas voulu encombrer l'explication de cette fonction avec des notations supplémentaires, voilà pourquoi elle est expliquée plus simplement. Toutefois le principe ne change pas.

### 5.1.2 Isolation de la grille

L'isolation de la grille se fait suite à la segmentation, la fonction de segmentation récupère le composant de l'image qui comprend le plus de pixels et renvoie son label. On peut donc ensuite effectuer un parcours complet de l'image où on supprime tous les pixels n'ayant pas le label du composant le plus grand.

Cela permet d'améliorer la détection des chiffres car si la grille n'est plus là, alors elle ne pourra pas être détectée à la place du chiffre.

Cette partie n'a pas été modifiée depuis la dernière soutenance.

### 5.1.3 Détection des coins de la grille

La détection des coins se repose sur ce que nous avons fait lors de la première soutenance, elle se fait en trouvant le pixel de la grille étant le plus proche du coin du carré dans lequel est inscrite celle-ci. Nous utilisons ici la méthode de Manhattan sur les pixels extérieurs de la grille pour savoir lequel d'entre eux est le plus proche du coin. Ainsi nous obtenons les coordonnées  $x$  et  $y$  de tous les coins de la grille.

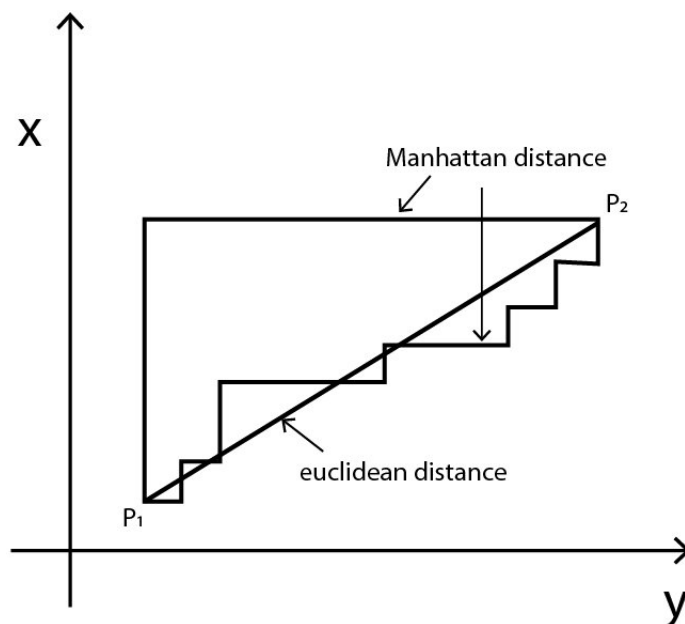


FIGURE 13 – Distance de Manhattan

#### 5.1.4 Rotation et redressement automatique

Cette partie a été entièrement ajoutée depuis la dernière soutenance, la partie du redressement utilise les coordonnées des coins trouvés dans la partie précédente pour calculer une matrice de transformation.

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2x'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

FIGURE 14 – Calcul de la matrice pour l'homographie

On essaie d'obtenir les valeurs se trouvant dans la matrice encadrée en rouge, les valeurs  $x$  et  $y$  numérotés sont les coordonnées des coins du carré de destination, l'image va être redressée pour que la grille soit de la forme de ce carré. Les valeurs  $x'$  et  $y'$  numérotés sont les coordonnées des coins de la grille sur l'image source.

Pour avoir es valeurs de la matrice encadrée en rouge on effectue des calculs sur les trois matrices de l'image, voici les calculs effectués :

On note la première matrice  $A$ , la seconde  $h$ , la dernière  $B$  et la transposée de  $A$  se note  $A_t$ .

$$h = \text{inverse}(A_t * A) * A_t * B \quad (3)$$

Une fois la matrice  $h$  obtenue, un parcours une image carré de destination que l'on vient de créer. Pour chacun de ses pixels on calcule le pixel correspondant dans l'image source et on applique ce pixel sur l'image de destination.

L'image de destination est donc une image ne contenant que la grille de sudoku, elle a été mise droite et un redressement a été appliqué.



Notre rotation automatique toutefois ne parvient pas à détecter si l'orientation de la grille est bonne. En fait, il se peut que si l'image d'entrée soit tournée à 90° dans un sens ou 180° l'image de sortie soit une image contenant la grille tournée, elle aussi, à 90° ou 180° dans le même sens. Nous aurions pu détecter ces rotations en envoyant les cases au réseau de neurone et en étudiant le résultat obtenu. Mais cette méthode est très couteuse et pas très sérieuse. Nous avons fait une hypothèse, celle-ci étant qu'un utilisateur normal ne s'amuserait pas à prendre une photo du sudoku à résoudre à l'envers ou de côté. L'image de sortie devrait donc tout le temps être droite. Nous avons toutefois décidé de laisser à l'utilisateur la possibilité de tourner l'image après le redressement de celle-ci au cas où l'image ne serait pas bien orientée.

### 5.1.5 Découpage des cases

Le découpage des cases garde le même fonctionnement que la première version mais des améliorations y ont été apportées. Premièrement les images à envoyer au réseau de neurone doivent être en format 28x28. Pour ce faire il suffit de rétrécir l'image avant de l'envoyer, mais il faut également que le chiffre détecté soit centré et toujours de la même taille.

Pour faire cela nous avons utilisé la fonction de propagation utilisée à la première soutenance pour trouver les chiffres dans la case, on fait partir cette propagation au centre de la case ce qui a pour effet d'isoler le chiffre des autres éléments parasites tels que le bruit. Puis on trouve les coordonnées supérieure, inférieure, gauche et droite qui encadrent le chiffre. (Comme on a fait pour la grille)

On peut donc ensuite découper le chiffre que l'on sauvegarde en attendant qu'il soit exploité par le réseau de neurone.

## 6 Réseaux de neurones

Nous avons décidé de créer une bibliothèque nous permettant de générer, charger, sauvegarder, entraîner et tester de manière tout à fait générique des réseaux de neurones. Cette bibliothèque doit également être simple et agréable d'utilisation.

Les améliorations qui ont été faites par rapport à la première soutenance sont assez subtiles. En plus de l'ajout de fonctionnalités, de résolutions de bugs et d'améliorations de code, nous nous sommes beaucoup focalisés sur l'aspect ergonomique de notre librairie, que ce soit par rapport à l'expérience de l'utilisateur ou la nôtre, dans son développement.

Voici un ensemble de règles et conventions qui ont été mises en place afin d'avoir une librairie agréable à utiliser et à implémenter :

- On utilise le *snake\_case* pour les variables, le *pascalCase* pour les fonctions et le *CamelCase* pour les *struct* et *enum*. Les constantes sont en *FULL\_CAPS*.
- Toutes les fonctions dites *publiques* commencent avec **nn\_**
- Toutes les fonctions dites *privées* commencent avec **\_nn\_**
- Les *struct* et *enum* ont un *typedef* associé

- Tout header a un fichier c associé, avec toutes les définitions dans ce header et les implémentations dans le fichier c
- On n'utilise pas *malloc*, *calloc*, *realloc* ou *free*, mais les implémentations *mem\_malloc*, *mem\_calloc*, *mem\_realloc* et *mem\_free* (cf. Prévention de fuite mémoire)
- On ne fait pas d'*include* relatif avec un *".."* dans le chemin d'accès. Dans ce cas, on *include* en partant du fichier racine du code source "src"

## 6.1 Aspects techniques

Un de nos objectifs étant la simplicité d'usage, nous voulons que l'utilisateur ait besoin d'utiliser le moins de fonctions possible. Voici la liste des fonctions et de types 'publics' de notre bibliothèque :

### Types publics

- `nn_Model` (struct)
- `nn_Data` (struct)
- `nn_DataTuple` (struct)
- `nn_DataSet` (struct)
- `nn_Session` (struct)
- `nn_ShapeDescription` (struct)
- `activation` (enum)
- `losses` (enum)
- `optimizer` (enum)

### Fonctions générales

- `nn_initRandom`
- `nn_initMemoryTracking`

```
1 // file: src/nn/utils/misc/randomness.h
2 void nn_initRandom(void);
3
4 // file: src/nn/nnutils/mem-management.h
5 void nn_initMemoryTracking(void);
```

### Génération d'un réseau de neurones

- `nn_createModel`
- `nn_freeModel`

```
1 nn_Model* nn_createModel(size_t num_layers, nn_ShapeDescription model_architecture[],
    activation activations[], losses loss, optimizer optimizer);
2 void nn_freeModel(nn_Model* model);
```

### Chargement, sauvegarde et utilisation d'un modèle

- `nn_loadModel`
- `save`
- `use`<sup>9</sup>

---

9. Note au lecteur : lorsque nous explicitons la définition d'une fonction qui doit être appelée depuis un struct, le *vrai* nom de cette dernière n'est probablement pas celui qui vous est indiqué dans ce document. Par exemple, il n'y a pas de fonction *use* en tant que telle dans notre code-base.

```

1 // file: src/nn/model/load_model.h
2 nn_Model* nn_loadModel(char* path);
3
4 // file: src/nn/model/save_model.h
5 void save(nn_Model* model, char* path);
6 // utilisation : model->save(model, path);
7
8 // file: src/nn/model/use_model.h
9 double* use(nn_Model* model, double* input);
10 // utilisation : model->use(model, input);

```

On peut aussi sauvegarder les poids et l'architecture de manière indépendante.

### Chargement d'un jeu de données

- nn\_loadDataSet
- nn\_loadTestOnlyDataSet
- nn\_loadSingleDataInputOutput
- nn\_createDataSet

```

1 // file: src/nn/data/init_data.h
2 nn_DataSet* nn_loadDataSet(char* data_dir_path, nn_ShapeDescription* shape, bool
    verb_mode);
3 nn_DataSet* nn_loadTestOnlyDataSet(char* data_dir_path, nn_ShapeDescription* description,
    bool verb_mode);
4 nn_Data* nn_loadSingleDataInputOutput(char* input_path, char* output_path,
    nn_ShapeDescription* shape, bool verb_mode);
5
6 // file: src/nn/utils/structs/dataset.h
7 nn_DataSet* nn_createDataSet(nn_Data* train_data, nn_Data* test_data);

```

### Session d'entraînement et de test d'un modèle

- nn\_createSession
- nn\_freeSession
- train
- test
- train\_one\_hot
- test\_one\_hot

```

1 // file: src/nn/session/session.h
2 nn_Session* nn_createSession(nn_DataSet* dataset, unsigned int nb_epochs, double
    loss_threshold, bool, stop_on_loss_threshold_reached, bool verbose, double
    learning_rate, const char* loss_log_file, const char*, right_log_file);
3 void nn_freeSession(nn_Session* session);
4
5 // file: src/nn/session/session.c
6 void train(struct nn_Session* session, nn_Model* model);
7 // utilisation: session->train(session, model);
8 void test(struct nn_Session* session, nn_Model* model);
9 // utilisation: session->test(session, model);
10 void train_one_hot(struct nn_Session* session, nn_Model* model);
11 // utilisation: session->train_one_hot(session, model);
12 void test_one_hot(struct nn_Session* session, nn_Model* model);
13 // utilisation: session->test_one_hot(session, model);

```

Voici un exemple d'utilisation :

```

1 nn_Model* model1 = loadModel("models/saved-model-1/");
2 nn_Model* model2 = loadModel("models/saved-model-2/");
3 nn_DataSet* dataset = nn_LoadDataSet("data/");
4 nn_Session* session = createSession(dataset, 20000, 0.0001, false, false, 0.15, "avg-loss
    .log", "avg-right.log");
5 session->train(session, model1);
6 session->test(session, model1);
7 session->train_one_hot(session, model2);

```

```
8 session->test_one_hot(session, model2);
```

Les fonctions *test\_one\_hot* et *train\_one\_hot* sont des fonctions qui sont spécialisées dans le testing et l'entraînement de réseaux en *one\_hot*. Ils utilisent en interne les mêmes sous-fonctions *test* et *train*. Il y a juste quelques statistiques qui sont données en plus.

### 6.1.1 Description de forme et de dimension

Afin de pouvoir décrire des données qui sont souvent de dimension inconnue, nous avons créé un *struct* appelé *nn\_ShapeDescription*. Ce type nous permet de décrire la forme qu'on nos couches de réseaux de neurones, la forme des données et bien d'autres variables. Cette structure a pourtant une définition très simpliste :

```
1 // file: src/nn/utils/structs/shape_description.h
2 struct nn_ShapeDescription {
3     size_t dims;
4     size_t x;
5     size_t y;
6     size_t z;
7     size_t range; // x * y * z
8 };
```

On peut voir que nous nous sommes limités à un nombre de dimensions fini. Sachant que nous n'aurons jamais, dans le cadre de ce projet, besoin de données qui sont ordonnées sur plus de trois dimensions, nous nous sommes arrêtés à *x*, *y* et *z*.

Voici les fonctions utilisables pour créer un *nn\_ShapeDescription* :

- *nn\_createShapeDescription*
- *nn\_emptyShapeDescription*
- *nn\_create1DShapeDescription*
- *nn\_create2DShapeDescription*

Voici leurs déclarations :

```
1 // file: src/nn/utils/structs/shape_description
2 nn_ShapeDescription nn_createShapeDescription(size_t x, size_t y, size_t z);
3 nn_ShapeDescription nn_create1DShapeDescription(size_t x);
4 nn_ShapeDescription nn_create2DShapeDescription(size_t x, size_t y);
5 nn_ShapeDescription nn_emptyShapeDescription(void);
```

Il vaut mieux utiliser les fonctions pour créer une description de forme, car il y a des calculs qui sont faits en fond et qui ne sont pas connus de l'utilisateur.

Avec tout ce qu'on a expliqué auparavant, nous savons que lorsqu'on doit itérer à travers les éléments d'un tableau qui est accompagné par une variable *nn\_ShapeDescription*, on peut faire :

```
1 // tableau: int array[]
2 // forme: nn_ShapeDescription shape
3 size_t slice_range = shape.y * shape.x
4 for (size_t z = 0; z < shape.z; z++) {
5     for (size_t y = 0; y < shape.y; y++) {
6         for (size_t x = 0; x < shape.x; x++) {
7             int element = tableau[
8                 z * slice_range
9                 + y * layer_shape.x
10                + x
11            ];
12            // element <- tableau[z][y][x]
13        }
14    }
```

```

15 }
16 // On sait également que le nombre d'éléments dans le tableau vaut :
17 size_t num_elements = shape.range; // shape.range = shape.x * shape.y * shape.z

```

### 6.1.2 Génération d'un modèle

Afin de générer un modèle (réseau de neurone), il faut créer les couches du réseau, ainsi que les neurones (nœuds) dans ces couches.

Le *struct* décrivant un modèle est le *nn\_Model*. Celui décrivant une couche est le *nn\_Layer*, et le nœud est *nn\_Node*. Voici leurs déclarations :

```

1 // file: src/nn/model/model.h
2 struct nn_Model {
3     size_t num_layers;
4     nn_Layer** layers;
5     losses loss;
6     optimizer optimizer;
7     void (*printModelLayers)(struct nn_Model* model);
8     void (*printModelLayersValues)(struct nn_Model* model);
9     void (*printModelArchitecture)(struct nn_Model* model);
10    double* (*use)(struct nn_Model* model, double*);
11    void (*save)(struct nn_Model* model, char*);
12 } nn_Model;
13
14 // file: src/nn/model/layer/layer.h
15 struct nn_Layer {
16     nn_ShapeDescription shape;
17     activation activation;
18     size_t num_nodes;
19     nn_Node** nodes;
20 };
21
22 // file: src/nn/model/layer/node/node.h
23 struct nn_Node {
24     // weights
25     size_t num_weights;
26     double* weights;
27     double* d_weights;
28     // bias
29     double bias;
30     double d_bias;
31     // values
32     double value;
33     double d_value;
34     // raw values
35     double raw_value;
36     double d_raw_value;
37 };

```

Il est important de noter que nous avons des *typedefs* pour tous ces structs, et que nous avons des fonctions pour *free* tous ces objets du *heap*. L'utilisateur n'aura qu'à appeler une fonction appelée *nn\_freeModel* pour libérer tout l'espace mémoire utilisé par le modèle, que ce soit directement ou indirectement <sup>10</sup>.

Comme vous pouvez vous l'imaginer, la majorité de ces fonctions ne sont pas appelées directement par l'utilisateur de notre bibliothèque, mais de manière indirecte.

10. Par l'espace mémoire *indirect*, on entend les couches, les nœuds, les poids, etc. Tous ces objets sont à la base d'allocations différentes que celle du modèle de base.

### 6.1.3 Chargement des données

Le chargement des données est structuré de la manière suivante : Plusieurs structs ont été définis pour décrire de la manière la plus fidèle possible le jeu de données dont l'utilisateur dispose au réseau de neurones. Le struct Data est un struct comportant un champ Data\_collection.

```
1 // file: src/nn/data/data.h
2 typedef struct nn_Data
3 {
4     nn_DataCollection data_collection;
5     nn_DataTuple (*splitTrainTest) (struct nn_Data, int);
6     void (*printData)(struct nn_Data);
7 } nn_Data;
```

Ce champ est une liste chaînée de tuples nommé nn\_InOutTuple, où le premier membre est une suite d'entrées dans le réseau de neurones, et le second une suite de sorties attendues lors du bon fonctionnement du réseau de neurones. Les deux membres sont tous les deux des références à un struct nommé nn\_Sample, qui possède la suite de valeurs, ainsi qu'une description de forme de couche, qui décrit précisément la dimension de la suite de valeurs que contient le Sample.

Le struct Data possède aussi une fonction de séparation du jeu de données en deux sous-jeux de données suivant un pourcentage renseigné en argument par l'utilisateur.

Pour initialiser un jeu de donnée avec le struct Data, l'utilisateur devra utiliser la fonction nn\_DataLoadRaw.

```
1 // file: src/nn/data/init_data.h
2
3 nn_Data nn_DataLoadRaw(char* input_path, char* output_path, bool verb_mode);
```

Cette fonction va lire deux fichiers renseignés par l'utilisateur, l'un décrivant le jeu de données d'entrées, l'autre le jeu de données de sorties, puis va construire la liste de tuples d'entrées et de sorties définissant le jeu de données brut représenté par le struct nn\_Data.

De ces structures définies ci-dessus, l'utilisateur ne pourra utiliser que le struct nn\_Data, qui représente le jeu de données brut dont il dispose.

Cependant, pour le bon fonctionnement de notre réseau de neurones, il nous faut deux jeux de données, l'un pour la phase d'entraînement, l'autre pour la phase de test.

Vient alors le struct nn\_Dataset.

L'utilisateur va tout d'abord découper en deux son jeu de données bruts en renseignant un pourcentage qu'il renseignera, puis renseignera les deux sous-jeux dans la fonction suivante :

```
1 // file: src/nn/utils/structs/dataset.h
2
3 nn_DataSet _nn_createDataSet(nn_Data trainData, nn_Data testData);
```

Ce struct sera par la suite utilisé dans l'initialisation d'un autre struct nommé nn\_Session dont nous parlerons plus en détails plus tard.

Un exemple final de chargement de données serait donc :

```
1 nn_Data data;  
2  
3 data = nn_DataLoadRaw(  
4     input_path,  
5     output_path,  
6     false  
7 );  
8 nn_DataTuple data_tuple = data.splitTrainTest(data, 0.3); // 70% train, 30% test  
9 nn_DataSet set = nn_createDataSet(data_tuple.data1, data_tuple.data2);
```

#### 6.1.4 Entraînement d'un modèle

L'implémentation du *feed-forward*, de la *back-propagation* et de la mise à jour de la valeur des poids ne s'est pas fait sans galère, mais ce sont des fonctions bien connues. Nous n'avons pas réinventé la roue et avons donc juste adapté ces fonctionnalités à la structure de nos variables.

Nous avons cependant séparé les fonctions de *back propagation* et de mise à jour des poids, ce qui n'est pas forcément le cas de toutes les implémentations. Au contraire, nous avons rassemblé en une seule fonction les étapes du *feed input* et de la *forward propagation*, ne voyant pas d'intérêt à séparer ces deux étapes.

Voici les déclarations de nos fonctions :

```
1 // file: src/nn/utils/session/evaluate.h  
2 void _nn_feedForward(nn_Model* model, double* input);  
3 void _nn_backPropagation(nn_Model* model, double* desired_output);  
4 void _nn_updateWeights(nn_Model* model, double learning_rate);
```

Pour rappel, la fonction d'entraînement, dite *feed forward* est définie par :

$$z_i^{(l+1)} = \sum_{j=1}^n w_{ij}^{(l)} x_i + b_i^{(l)}$$

avec :

- $l$  l'indice de la couche
- $i$  l'indice de neurone référant à la couche  $l + 1$
- $j$  l'indice de neurone référant à la couche  $l$
- $n$  le nombre de neurones dans la couche à l'indice  $l$

Et voici la fonction de calcul d'erreur des poids, dite *back propagation* :

$$\delta_j^{(l)} = \sum_{i=1}^n (w_{ij}^{(l)} \delta_j^{(l+1)}) f'(z_j^{(l)})$$

avec :

- $l$  l'indice de la couche
- $i$  l'indice de neurone référant à la couche  $l + 1$
- $j$  l'indice de neurone référant à la couche  $l$
- $n$  le nombre de neurones dans la couche à l'indice  $l$

Il suffit donc de sauvegarder toutes les erreurs  $\delta_j^{(l)}$  (on le fait dans le champ *d\_weights* du struct *nn\_Node*), puis de mettre à jour les poids via la formule suivante :

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial}{\partial w_{ij}^{(l)}} J(w, b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(w, b)$$

#### 6.1.5 Fonctions d'activation et leurs dérivées

Afin que le réseau de neurones puisse correctement fonctionner, que les valeurs de chaque nœud soient modifiées, nous avons mis en place plusieurs fonctions d'activation que nous pouvons choisir pour chaque couche du réseau de neurones.

Nous avons dans un premier temps fait la fonction sigmoid :

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \quad (4)$$

Puis la fonction Relu :

$$Relu(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \quad (5)$$

Puis la fonction Softmax :

$$Softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \quad (6)$$

De plus, nous avons aussi mis en place des dérivés de certaines de ces fonctions, comme par exemple dSigmoid :

$$dSigmoid(x) = sigmoid(x)(1 - sigmoid(x)) \quad (7)$$

dRelu :

$$dRelu(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \quad (8)$$

et dSoftmax :



$$dSoftmax(x_i) = \frac{\exp(x_i) * \sum_{j \neq i} \exp(x_j)}{(\sum_j \exp(x_j))^2} \quad (9)$$

#### 6.1.6 Session d'entraînement et de test

Pour pouvoir opérer les fonctions d'évaluations (propagation avant et arrières) et ainsi faire évoluer le réseau de neurones, nous avons conçu un *struct* nommé `nn_Session`.

Son but : permettre à l'utilisateur l'opération des fonctions d'évaluations sur un nombre d'époques données. Si l'utilisateur le souhaite, il peut renseigner un nombre définissant le taux d'erreur maximal voulu et permettre ainsi à la session de se terminer lorsque l'on atteint un taux d'erreur en deçà du taux renseigné par l'utilisateur. Il faut aussi renseigner le dataset que l'on aura construit au préalable avant la construction de la session.

En plus de tous ces paramètres, il est possible de donner le chemin vers deux fichiers de log. La moyenne des coûts du modèle sera sauvegardé de manière périodique dans le premier fichier donné. Le deuxième fichier quant à lui aura comme contenu l'évolution du pourcentage de prédictions justes. Ces deux fichiers sont mis à jour en même temps, ce qui permet une meilleure interprétation des données après l'entraînement.<sup>11</sup>

L'utilisateur va ensuite pouvoir entraîner son réseau de neurones à l'aide de la fonction `train` (qui opère la propagation avant ET arrière pour la correction des poids et biais du réseau), puis tester son réseau et comparer les résultats attendus avec les résultats obtenus (en opérant uniquement une propagation avant).

#### 6.1.7 Fonction de coût

Pour calculer le taux d'erreur présent sur la suite de valeur de la dernière couche du réseau de neurones, l'utilisateur peut mentionner lors de la création de son modèle la fonction de coût calculant ce taux.

Au total, nous en avons implémenté trois : La première est la *Mean Squared Error Loss Function*. Celle-ci est très utilisée pour l'entraînement du réseau de neurones à une fonction de porte logique (dans le cas présent, une porte ou-exclusif), et se définit comme suivant :

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 \quad (10)$$

La seconde est la *Categorical Cross Entropy* :

$$Loss = - \sum_{i=1}^n (Y_i \log \hat{Y}_i) \quad (11)$$

---

11. Voir le "logging"

Et la dernière est la *binary cross entropy* :

$$Loss = -\frac{1}{n} \sum_{i=1}^n (Y_i \log \hat{Y}_i + (1 - Y_i) \log 1 - \hat{Y}_i) \quad (12)$$

#### 6.1.8 Sauvegarde et Chargement d'un modèle

Pour pouvoir sauvegarder les entraînements génériques que nous avons formé sur différents réseaux de neurones, nous avons développé deux fonctions cruciales pour l'exportation/importation de modèle : `load_model` et `save_model`.

Les prototypes de ses fonctions sont les suivants :

```
1 // file : src/nn/model/load_model.h
2
3 nn_Model* nn_loadModel(char* dirpath);
```

```
1 // file : src/nn/model/save_model.h
2
3 void _nn_Model_saveModel(nn_Model* model, char* dirpath);
```

La fonction `save` est accessible sous le format de méthode, c'est-à-dire que le struct `nn_Model` contient un pointeur pointant vers cette fonction. La fonction est donc appellable de cette manière :

```
1 model->saveModel(model, dirpath);
```

Ces deux fonctions sont *chirales* en quelque sorte, c'est-à-dire qu'elles opèrent les mêmes opérations, mais pas dans la même direction.

En effet, en premier lieu la fonction `save_model` va sauvegarder l'architecture du modèle dans un fichier nommé « `architecture.mdl` ». Ce fichier contiendra le nombre de couches du réseau, sa fonction de coût, son optimisateur, et la forme de chaque couche du réseau, id est, le nombre de nœud dans chaque couche.

Puis, la fonction `save_model` va sauvegarder les poids et biais de chaque nœud dans un fichier nommé « `weightsandbias.mdl` ».

De manière analogue, la fonction `load_model` va cette fois ci, récupérer l'architecture du modèle sauvegardé dans le fichier cité précédemment. Puis, à l'aide des informations récoltés, la fonction va allouer le modèle avec les bonnes caractéristiques et les bonnes dimensions de couche.

Enfin, la fonction va lire l'ensemble des poids et biais de chaque nœud stocké dans le fichier cité précédemment, pour les assigner au nouveau modèle alloué. Enfin la fonction renvoie un pointeur vers le modèle alloué en mémoire.

### 6.1.9 Utilisation du modèle

Désormais, pour pouvoir utiliser le modèle correctement et de manière industrielle, nous avons développé une fonction permettant de propager une entrée dans le réseau de neurones, puis renvoyer le résultat présenté par ce dernier.

Le prototype de cette fonction se présente comme ceci :

```
1 // file : src/nn/model/use_model.h
2
3 double* _nn_useModel(nn_Model* model, double* input);
```

Cette fonction nécessite en premier lieu de faire une propagation avant du tableau de double entrée en argument (on admet que le tableau de double est de même dimension que le layer d'entrée du réseau de neurones). Puis, on alloue un espace mémoire à un tableau de double ayant la même dimension que la couche de sortie du réseau de neurones, et l'on copie les valeurs stockées dans chaque nœud du réseau de neurones. Puis l'on renvoie un pointeur vers ce tableau fraîchement généré.

Comme pour la fonction `save_model`, la fonction `use` est accessible sous le format d'une méthode de struct. Ainsi l'utilisateur d'un modèle va appeler la fonction `use` de cette manière :

```
1 model->use(model, input);
```

## 6.2 Aspects techniques - Annexe

Nous avons mis beaucoup de temps et d'efforts dans le développement d'outils qui peuvent être considérés comme *annexes* au code de la librairie. Ce sont des outils qui peuvent par exemple nous permettre un entraînement, un test ou une évaluation plus rapide d'un réseau de neurones. Nous avons également beaucoup travaillé sur des outils pour faciliter la manipulation de bases de données.

### 6.2.1 Conversion de cellules en objet numérique

Maintenant que nous avons bien isolé toutes les cellules de notre grille du sudoku dans un format bitmap, il nous faut convertir ces bitmaps en un tableau de double.

Pour ce faire, nous avons développé une fonction `convert`. Le prototype de celle-ci est le suivant :

```
1 // file : src/gui/convert/convert.h
2
3 void convert(char* path, double** converted_cells, Cell* cells_position);
```

Les arguments se décomposent de la manière suivante :

- `Path` est le chemin vers lequel se situe le dossier contenant toutes les cellules découpées de l'image
- `Converted_cells` est un pointeur vers un tableau de tableau de double, on suppose que ce tableau a déjà été alloué avant avec la bonne taille (en préambule de l'appel de la fonction `convert`, on va compter le nombre de fichiers contenu dans le dossier pointé par `path`).

- Cells\_position est un tableau contenant des objets Cell. Les objets Cell sont tous simplement des structs représentant la position de la cellule dans la grille. Le struct se présente donc de la manière suivante :

```
1 typedef struct Cell
2 {
3     unsigned int x;
4     unsigned int y;
5 } Cell;
6
7
```

Dans la fonction convertir, on va dans un premier temps parcourir l'ensemble du dossier que l'on suppose uniquement rempli de fichier bitmap représentant chacun une cellule unique de la grille de sudoku (on est sûr que le travail fait en prétraitement est magnifiquement bien achevé).

A chaque cellule, on alloue un nouveau tableau de double de taille 28x28 (c'est-à-dire la taille standardisée de chaque bitmap de cellule).

Puis on parcourt l'ensemble du bitmap représentant la cellule, en itérant sur l'ensemble des pixels de cette dernière. Comme chaque cellule est binarisé, les composantes r g b de chaque pixel sont égales. Il nous suffit donc de prendre la composante r par exemple, puis de la diviser par 255 pour avoir un ratio appartenant à  $[0;1]$  et traduisant l'intensité de couleur blanche du pixel.

A la fin de ce traitement, on passe d'un bitmap à un tableau de booléen, qui pourrait se traduire visuellement par :



FIGURE 15 – Un 7 extrait d'une cellule d'une grille de sudoku

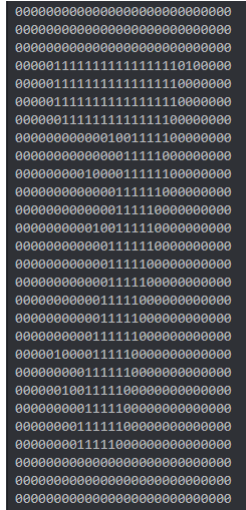


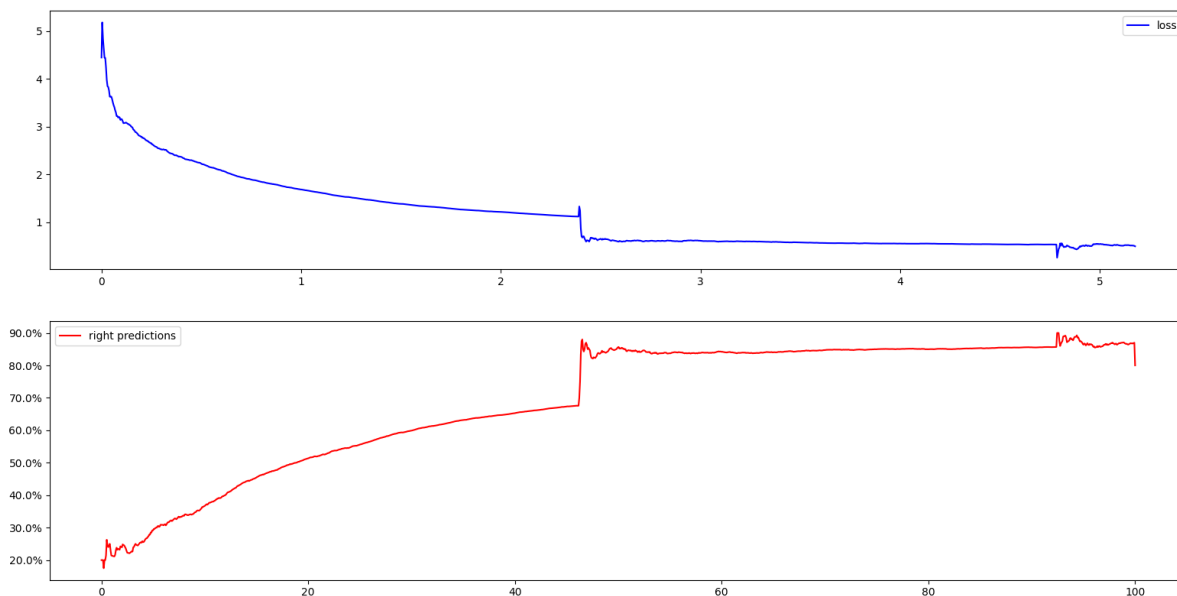
FIGURE 16 – Un 7 exprimé en un tableau de double unidimensionnel

Ceci est une représentation ascii, en temps normal les cases valant « 1 » ne valent « 1 » qu'à l'intérieur du chiffre, tandis qu'au bord du chiffre, ce sera un nombre compris entre 0 et 1 strictement. De plus, le tableau est de dimension unique, mais pour les besoins de démonstrations, nous avons formaté l'image afin d'obtenir un carré de 28 nombres par 28 lignes.

### 6.2.2 Logging

Afin d'avoir une meilleure compréhension de ce qui se passe lors d'une phase d'apprentissage d'un modèle, on a implémenté un système de *logging*, qui permet de sauvegarder les taux d'erreurs et pourcentages de réussite d'un modèle. On a également créé un script en python qui permet de visualiser ces données avec matplotlib<sup>12</sup>. Voici quelques exemples de graphes que nous pouvons obtenir :

12. Lien vers le projet matplotlib : <https://matplotlib.org/>



Visualisation du taux d'erreur (bleu) et de réussite (rouge) d'un modèle lors d'une phase d'entraînement

Il est à noter que l'on peut visualiser un tel graphique pendant la phase d'entraînement. L'utilisateur n'est pas obligé d'attendre qu'un long entraînement soit fini pour avoir accès à ce genre de visualisation.

Cependant, les données qui sont affichées dans le terminal sont également très complètes. La visualisation de ce genre de graphique devient utile pour détecter, par exemple, si le modèle n'est pas tombé dans un minimum local.

### 6.2.3 Manipulation de bases de données

Notre librairie étant ce qu'elle est, nous ne pouvions pas que charger des images comme entrée. Nous avons donc décidé de créer notre propre format de fichier pour stocker nos datasets. Un fichier de donnée pour notre librairie doit être formaté comme suit :

```
NB-DE-DONNÉES DIM-DES-DONNÉES NB-SUR-X NB-SUR-Y NB-SUR-Z
-- nombres flottants séparés par des espaces représentant la première donnée --
-- nombres flottants séparés par des espaces représentant la deuxième donnée --
...
-- nombres flottants séparés par des espaces représentant la n-ième donnée --
```

Prenons pour exemple le xor, le contenu du fichier **xor-data.in** était comme suit :

```
4 1 2 1 1
1.0 1.0
1.0 0.0
0.0 1.0
0.0 0.0
```

Analysons la première ligne : le **4** signifie qu'il y a 4 données qui suivent (ou 4 lignes). Le **1** signifie que nos données sont uni-dimensionnelles. La donnée de dimension sert surtout à vérifier que le reste colle bien. Le **2** qui suit ce 1 donne la dimension en **x** de nos données. Sur l'axe **x**, il y a en effet 2 données (qui sont les deux bits d'entrée du xor). Les deux **1** qui suivent ce **1** signifient qu'il n'y a qu'une seule donnée sur les axes **y** et **z**. On peut faire l'analogie avec les espaces vectoriels.

Et le fichier **xor-data.out** :

```
4 1 1 1 1
0.0
1.0
1.0
0.0
```

Dans un processus de recherche de la meilleure solution d'entraînement, nous avons rassemblé un certain nombre de bases de données. La principale difficulté était de convertir certaines bases vers notre format. Il y a également des bases de données que nous avons généré nous-mêmes, directement dans le format de notre librairie.

## Outils qui ont été créés pour les bases de données

### Aide à l'étiquetage de données réelles

Générer des ensembles de données est quelques fois bien long et ennuyant. Il faut regarder l'image, lui donner un nom qui n'a pas déjà été pris, et spécifier dans le nom le nombre que l'image représente. Appuyer sur une pluralité de touche pour ouvrir l'image, la renommer, refermer l'image, et passer à la suivante est un temps perdu. C'est pour cela que nous avons fait un petit outil qui nous permet d'automatiser ces tâches.

Cet outil est écrit en python, donc nous n'allons pas expliquer de manière technique, mais plutôt en expliquer le principe.

Il vous suffit de lancer le script en y référant au travers d'un argument CLI quel dossier d'images vous souhaitez étiqueter. Le script va alors pour chaque image suivre un protocole simple :

- L'image est ouverte, on demande à l'utilisateur quel nombre représente cette image.
- Une fois une réponse valide reçue, on incrémente de 1 la case représentant le nombre stocké dans un histogramme, puis on renomme l'image sous le format [réponse]\_[valeur stocké dans l'histogramme], de sorte à avoir à chaque fois un nom d'image unique.
- On passe à l'image suivante.
- On réitère l'étape 1 jusqu'à que ce soit la dernière image.

Avec cet outil, la formation d'ensembles de données est très efficace et super rapide, puisque l'on étiquette en moyenne 30 images par minutes, contre 5 sans l'outil (valeurs formulées par empirisme).

**Convertisseur d'images en python** Afin de convertir un dossier rempli d'images au préalable étiquetées avec l'outil d'étiquetage décrit juste au-dessus vers notre format de base de donnée, nous avons écrit un autre script en python, dont le pseudo-code ressemble à ceci :

```
convertisseur (dossier)
  pour chaque 'fichier' dans le 'dossier':
    img <- lecture du 'fichier'
    donnée <- convertir 'img'
    écrire 'donnée' dans la base de donnée

convertir(img)
  M <- matrix des pixels noir et blanc 'img'
  M <- changement du type de (M) : 'uint8' vers 'float32'
  arr <- applatissemenet de M: (28, 28, 1) vers (784, 1)
  arr <- map(arr, 0, 255, 0, 1) // de 0-255 vers 0-1
  s <- formatage de 'arr' vers le format custom de nos bases de données
  retourner s
```



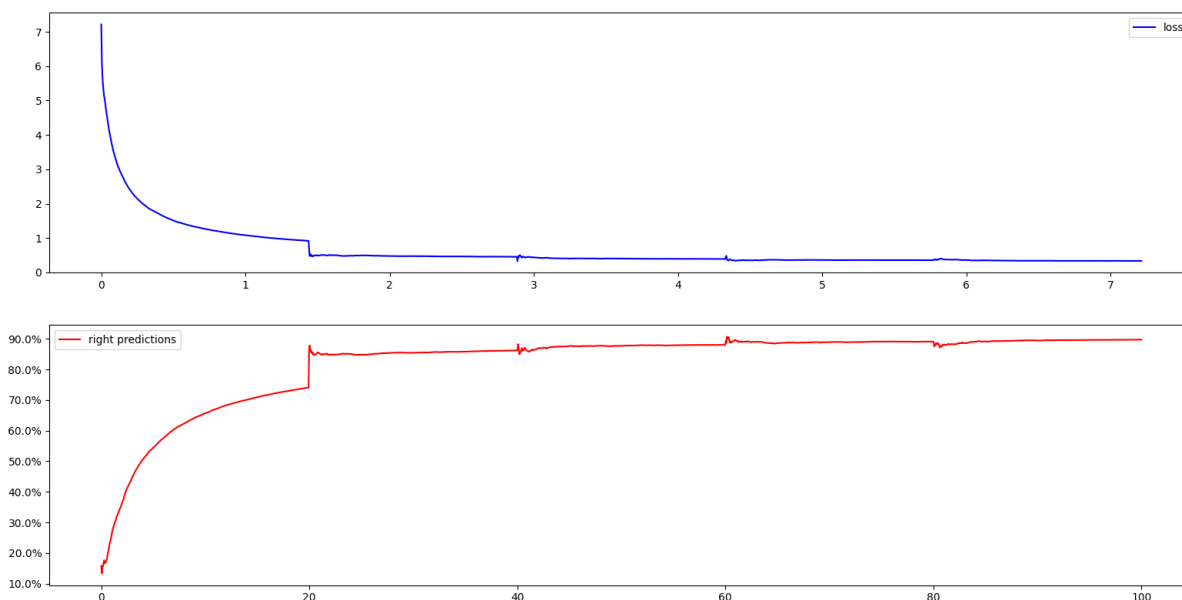
Voici les bases de données que nous avons utilisées :

## MNIST

La base de donnée mnist est certainement la plus connue dans le milieu de l'apprentissage profond. Elle contient 60'000 images en noir et blanc en 28x28 et 10'000 images supplémentaires (pour le test) de nombres manuscrits. C'est la première base que nous avons utilisé. Nous avons converti cette base en notre format avec *python*, *numpy* et *pillow*, tout en filtrant les zéros. Lorsque la base de donnée est formatée, le fichier d'entraînement fait **271Mb**.

Description	Valeur
Nombre d'époques	5
Taux d'apprentissage	0.05
Taille(s) couche(s) intermédiaire(s)	200
Fonctions d'activations intermédiaires	sigmoid
Fonction d'activation de la dernière couche	softmax
Fonction de taux d'erreur	CCE
Taux d'erreur final	0.338532
Taux de réussite sur la portion 'test' de la base de donnée	89.71 %
Taux de réussite sur des images réelles	44.68 %

TABLE 1 – Meilleure performance sur MNIST



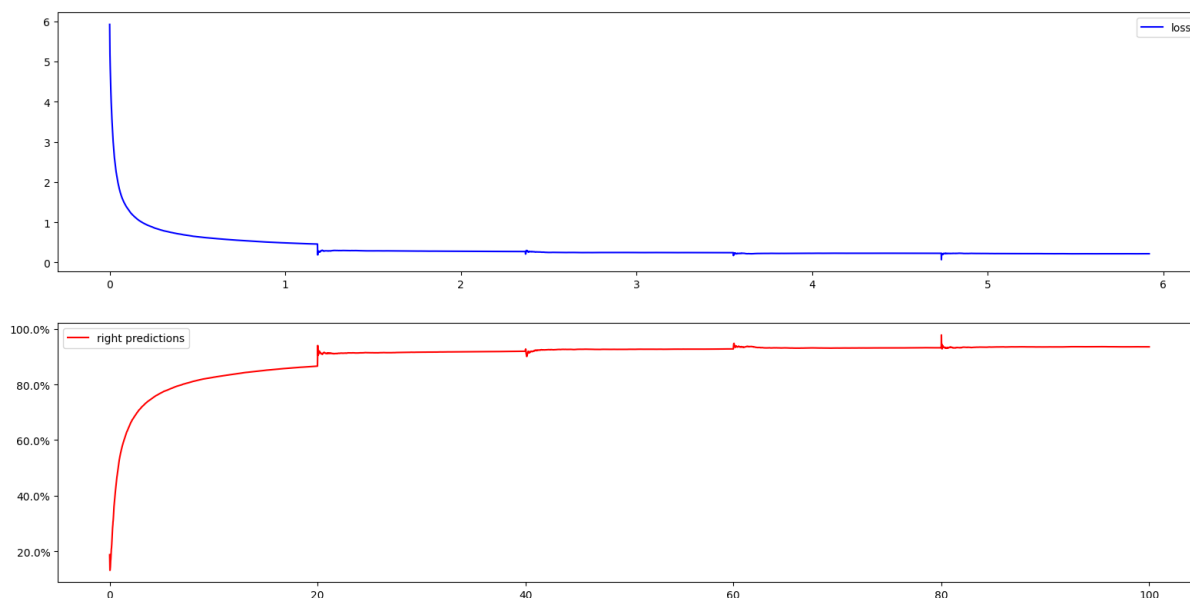
Logs de l'apprentissage sur le dataset MNIST

## EMNIST

EMNIST (ou Extended-MNIST) est le successeur de MNIST. Elle contient non pas 60'000, mais 240'000 chiffres pour l'entraînement. Et 40'000 chiffres pour le test. Cette base a posé des problèmes plus difficiles à régler. Nos ordinateurs n'étaient pas assez performants. Et notre code certainement pas assez optimisé. Nous avons pu, grâce à cette base optimiser de grandes parties de notre code. Mais nous ne l'avons pas utilisé, car tout était très lent, que ce soit le chargement des données ou l'entraînement. Si nous avions un peu plus de temps et des machines un peu plus puissantes, nous aurions certainement tiré plus de cette base que ce que nous avons pu faire jusque-là. Une fois formatée, le fichier de *train* fait **993Mb**.

Description	Valeur
Nombre d'époques	5
Taux d'apprentissage	0.05
Taille(s) couche(s) intermédiaire(s)	200
Fonctions d'activations intermédiaires	sigmoid
Fonction d'activation de la dernière couche	softmax
Fonction de taux d'erreur	CCE
Taux d'erreur final	1.980217
Taux de réussite sur la portion 'test' de la base de donnée	93.30 %
Taux de réussite sur des images réelles	50.00 %

TABLE 2 – Meilleure performance sur EMNIST



Logs de l'apprentissage sur le dataset EMNIST

## Numeric

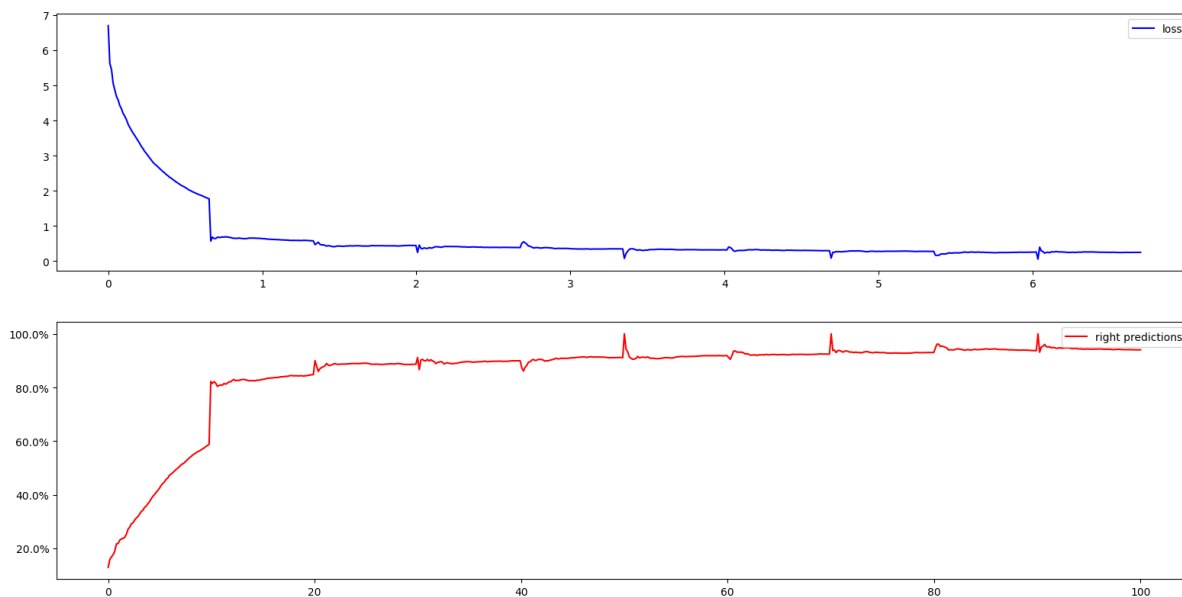
Numeric est un ensemble de nombres écrits par ordinateur avec différentes polices d'écriture. Cette base de données contient plus de 9'000 images en noir et blanc en 28x28. C'est une base qui nous a été partagée avec un membre d'un autre groupe avec lequel nous avons fait un échange « commercial ». Le fait est que la base de données initiale contenait des images en 128x128 en noir sur blanc, ce qui n'était pas adapté à notre projet. Nous avons donc fait un marché avec le membre cité précédemment en lui renvoyant l'ensemble convertis en 28x28 si celui-ci nous partageait la base de données initiale. Puis nous avons encore une fois fait une conversion en passant l'image du format « écriture noire sur fond blanc » à « écriture blanche sur fond noir ».

Après formatage, le fichier d'entraînement fait **xMb**.

numeric dataset

Description	Valeur
Nombre d'époques	10
Taux d'apprentissage	0.1
Taille(s) couche(s) intermédiaire(s)	200
Fonctions d'activations intermédiaires	sigmoid
Fonction d'activation de la dernière couche	softmax
Fonction de taux d'erreur	CCE
Taux d'erreur final	0.3
Taux de réussite sur la portion 'test' de la base de donnée	91.88 %
Taux de réussite sur des images réelles	86.18 %

TABLE 3 – Meilleure performance sur Numeric



Logs de l'apprentissage sur le dataset Numeric

## Noisy

La base de donnée *noisy* a entièrement été générée par nos soins. Cette base a été créée selon un algorithme assez simple :

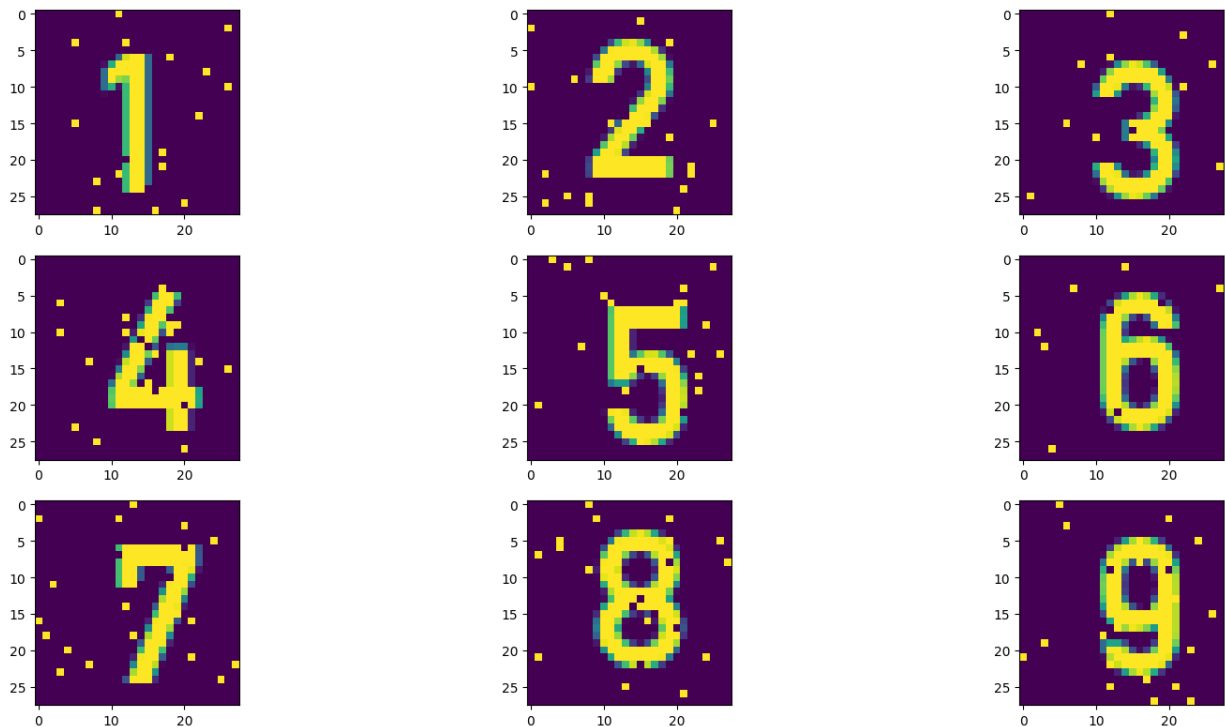
génération des images (N, seuil) :

```
pour i allant de 1 à N
  r <- nombre aléatoire entre 1 et 9
  img <- génération de ce nombre dans une police de caractère donnée (image 28x28)
  dimg <- décalage de ce nombre sur les axes x, y dans 'img'
  dding <- bruitage(dimg, seuil) // voir un peu plus bas
  sauvegarde de 'ddimg' dans notre format
```

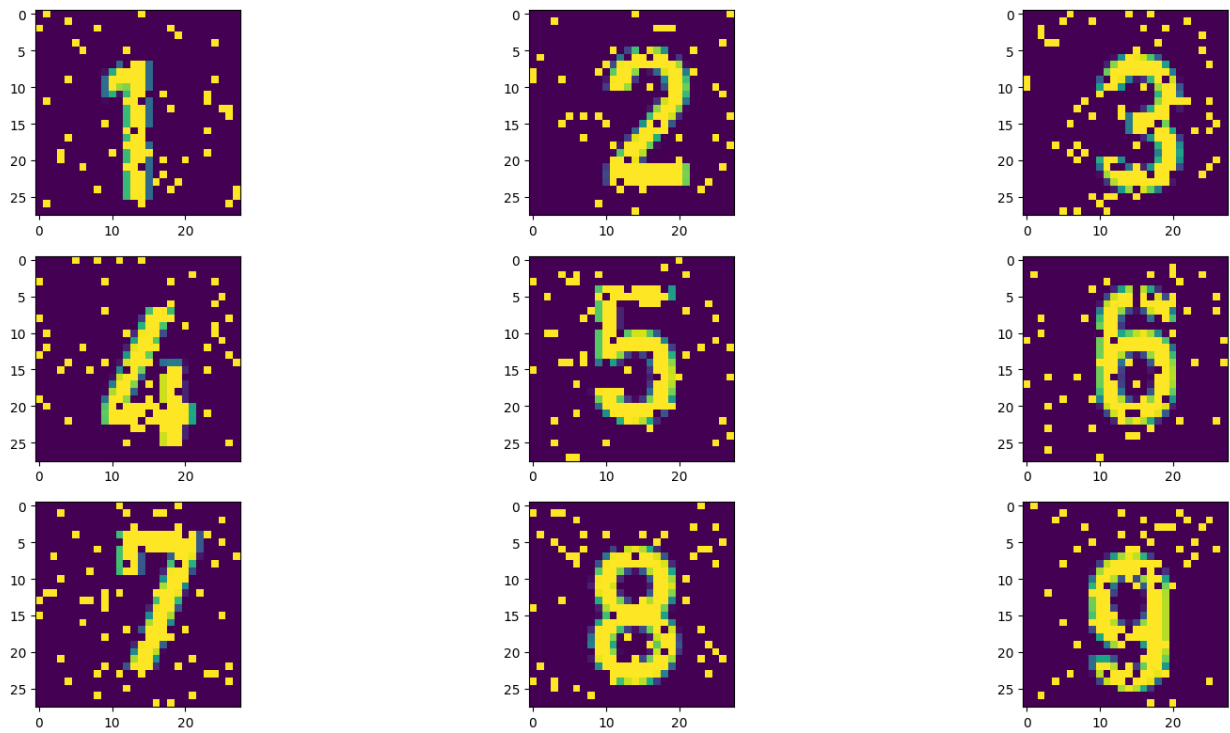
bruitage (dimg, seuil) :

```
m <- matricice de distributions normales de 0 à 1
ddimg <- copie de dimg
ddimg[lorsque m > seuil] = 255
ddimg[lorsque m < -seuil] = 0
valeur de retour: dding
```

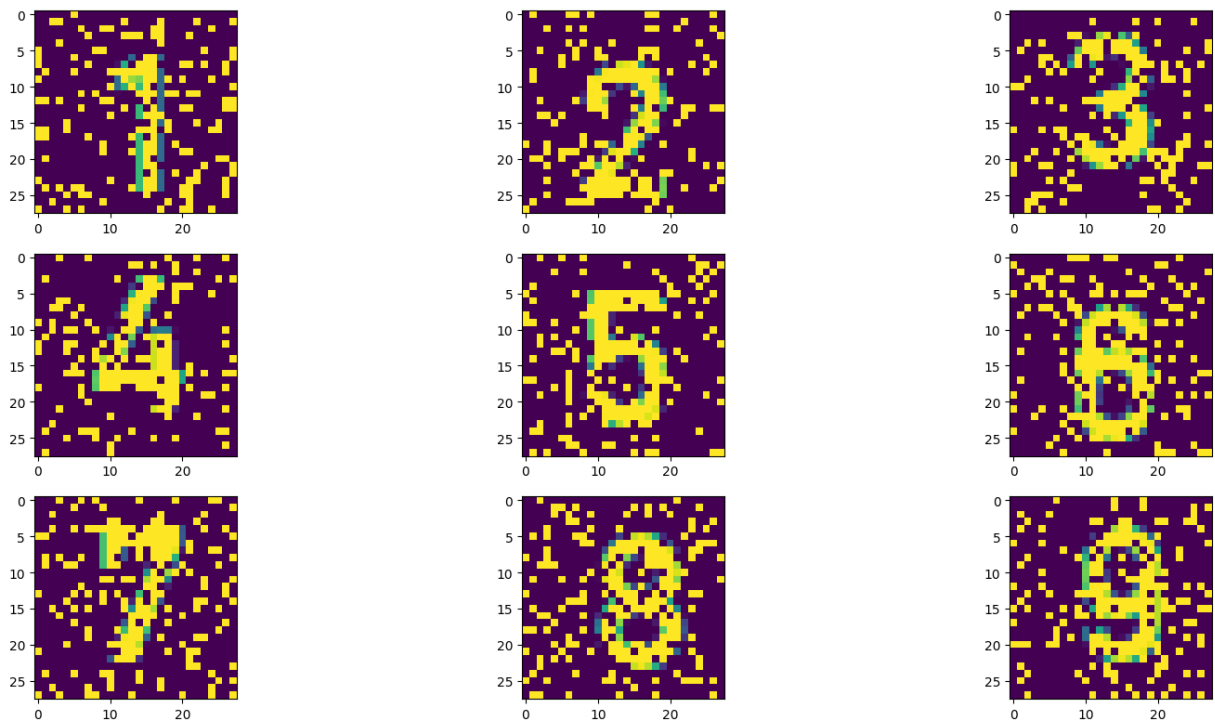
Voici l'effet des différentes valeurs de seuil :



Valeur de seuil de 2.0



Valeur de seuil de 1.5



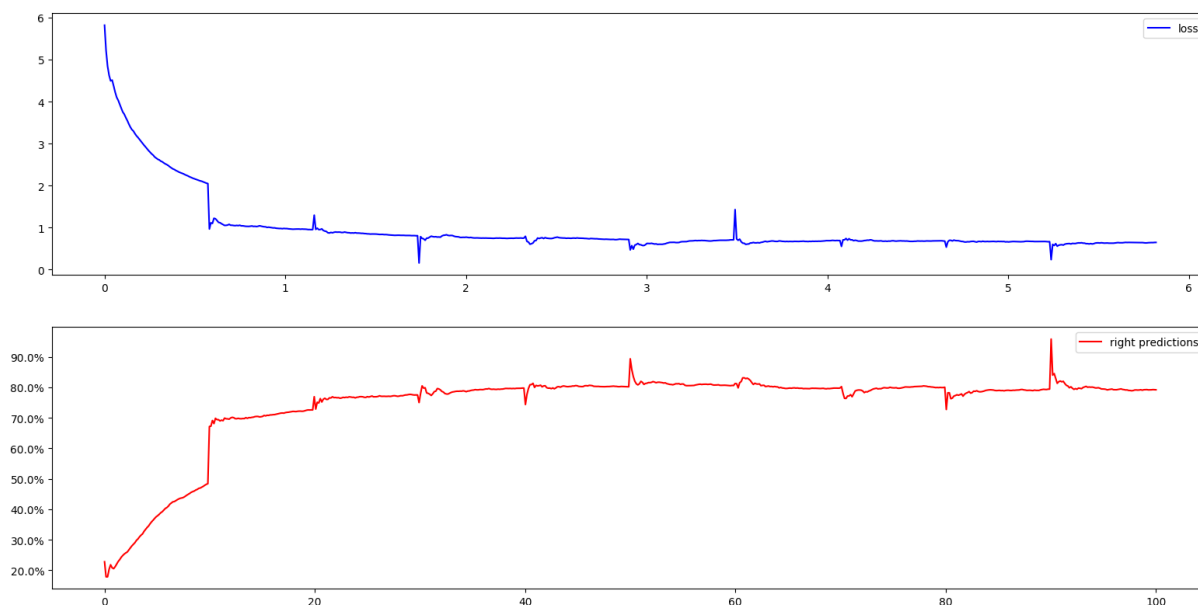
Valeur de seuil de 1.0

Plus la valeur de seuil est basse, plus la probabilité qu'un pixel soit modifié est élevée.

Si nous avions voulu, nous aurions pu générer 1 million d'entrées pour cette base, mais il y a des risques de redondance. Il est à noter que plus la valeur de seuil est élevée, plus la redondance est probable. Nous avons finalement utilisé la valeur de seuil de 1.0 et un N de 10'000.

Description	Valeur
Nombre d'époques	10
Taux d'apprentissage	0.1
Taille(s) couche(s) intermédiaire(s)	200
Fonctions d'activations intermédiaires	sigmoid
Fonction d'activation de la dernière couche	softmax
Fonction de taux d'erreur	CCE
Taux d'erreur final	0.977832
Taux de réussite sur la portion 'test' de la base de donnée	74.80 %
Taux de réussite sur des images réelles	29.79 %

TABLE 4 – Meilleure performance sur Noisy



Logs de l'apprentissage sur le dataset Noisy

Cette base de donnée, indépendamment des autres, n'a pas apporté grand-chose. C'est lors de la fusion de bases qu'elle nous a été très utile (voir un peu plus bas).

## Artisanal

Artisanal est un ensemble de nombres extraits des grilles de sudoku fourni par le sujet. L'ensemble contient 100 images des six grilles de sudoku. C'est une base de données qui est très utile pour apprendre au réseau de neurones à omettre les bruits qui peuvent subvenir sur le nombre à deviner. La base de données a été formée à l'aide de l'outil d'étiquetage python ainsi que l'outil de conversion python décrit précédemment et développé par nos soins. Comme toutes les bases décrites précédemment, c'est une base de données d'images en 28x28 en écriture blanche sur fond noir. Cet ensemble n'est pas utilisé de manière solitaire, mais est plus une base de données qui est mixée à d'autres pour obtenir des résultats bien plus cohérents lorsque le réseau est utilisé en production.

Après formatage, le fichier d'entraînement fait **xMb**  
artisanal dataset

Description	Valeur
Nombre d'époques	50
Taux d'apprentissage	0.1
Taille(s) couche(s) intermédiaire(s)	256 128
Fonctions d'activations intermédiaires	sigmoid sigmoid
Fonction d'activation de la dernière couche	sigmoid
Fonction de taux d'erreur	CCE
Taux d'erreur final	0.52
Taux de réussite sur la portion 'test' de la base de donnée	90

TABLE 5 – Meilleure performance sur Artisanal

## Fusions de bases de données

Nos modèles les plus performants ont été entraînés sur plusieurs datasets (qui ont été fusionnées).

### Noisy+Artisanal

En fusionnant les datasets 'Noisy' et 'Artisanal', nous avons pu obtenir des résultats très concluants sur à la fois la portion 'test' de la base de donnée résultante de la fusion, mais également sur de vraies images provenant de l'extraction de sudoku.

### MNIST+Numeric

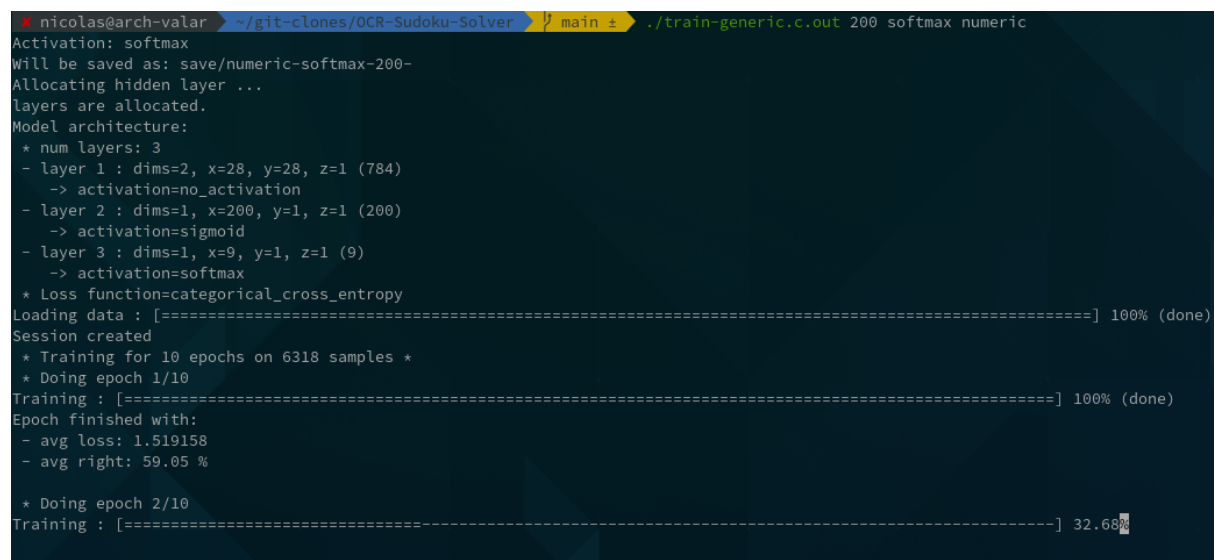
Nous avons également entraîné des sudokus sur une fusion entre 19'000 couples de la base MNIST et les 10'000 couples de 'Numeric'. Malheureusement, les performances de ces modèles sur des exemples de cellules provenant de l'extraction d'image sont beaucoup moins concluants.

### 6.2.4 Verbosity

Nous donnons beaucoup d'importance à ce que notre output dans le terminal soit modulable et le plus clair possible. En effet, si ce qui est affichée sur la sortie standard ou la sortie erreur est du texte bien formaté, il est plus facilement compréhensible. C'est pourquoi nous avons mis en place un système qui permet d'activer ou de désactiver la *verbosité* de certaines parties du code. Cela se fait en appelant la fonction `nn_setVerbose`.

Lorsque nous entraînons des réseaux de neurones, il se peut que l'on doive charger des bases de données dont la taille dépasse **1G** (par exemple avec EMNIST). On ne peut donc objectivement pas faire un `printf` toutes les 1000 données chargées ou autre. Le même problème se pose lors de l'entraînement. Comment avoir une sortie qui ressemble à quelque chose tout en affichant des choses utiles ?

Nous avons donc implémenté quelque-chose de révolutionnaire et de jamais vu : **la barre de chargement**.



```
* nicolas@arch-valar > ~/git-clones/OCR-Sudoku-Solver > main ± ./train-generic.c.out 200 softmax numeric
Activation: softmax
Will be saved as: save/numeric-softmax-200-
Allocating hidden layer ...
layers are allocated.
Model architecture:
* num layers: 3
- layer 1 : dims=2, x=28, y=28, z=1 (784)
  -> activation=no_activation
- layer 2 : dims=1, x=200, y=1, z=1 (200)
  -> activation=sigmoid
- layer 3 : dims=1, x=9, y=1, z=1 (9)
  -> activation=softmax
* Loss function=categorical_cross_entropy
Loading data : [=====] 100% (done)
Session created
* Training for 10 epochs on 6318 samples *
* Doing epoch 1/10
Training : [=====] 100% (done)
Epoch finished with:
- avg loss: 1.519158
- avg right: 59.05 %
* Doing epoch 2/10
Training : [=====] 32.68%
```

Capture d'écran illustrant les barres de chargement lors d'un entraînement

### 6.2.5 Entraînement générique et tests spécialisés

Afin pouvoir entraîner et tester des modèles plus facilement, nous avons créé deux scripts C : le premier, **train-generic.c** prend 3 arguments : la taille de la couche cachée (partons du principe qu'on ne veuille qu'une seule couche intermédiaire)<sup>13</sup>, la fonction d'activation de la dernière couche (les autres sont sigmoid) et le nom du dataset sur lequel on veut entraîner notre réseau.

Ce script va donc entraîner notre réseau et puis lui faire passer une phase de test sur la portion 'test' du dataset.

Enfin, nous avons un script pour tester un modèle sur des cellules qui viennent directement de l'extraction d'un sudoku : **test-real-output.c**. Ce script donne un pourcentage

13. Nous aurions pu donner le nombre de couches en argument, mais cela nous semblait inutile, car nous entraînons très souvent avec une seule couche intermédiaire



de réussite de prédiction global, un pourcentage de réussite par chiffre et d'autres données utiles (comme le taux d'erreur CCE, par exemple).

Ces deux scripts nous ont grandement facilité la tâche d'apprentissage, car nous n'avions pas à tout recompiler à chaque fois que nous voulions changer de dataset, d'architecture ou de fonction d'activation.

### 6.2.6 Calibrage des modèles

## MNIST

MNIST est la première base de donnée que nous avons utilisé. Nous avons commencé par chercher sur internet quelles étaient les meilleures architectures de réseaux de neurones. Nous avons trouvé le graphique suivant sur Wikipédia :

Type	Classifier	Distortion	Preprocessing	Error rate (%)
Linear classifier	Pairwise linear classifier	None	Deskewing	7.6 <sup>[10]</sup>
K-Nearest Neighbors	K-NN with non-linear deformation (P2DHMDM)	None	Shiftable edges	0.52 <sup>[23]</sup>
Boosted Stumps	Product of stumps on Haar features	None	Haar features	0.87 <sup>[24]</sup>
Non-linear classifier	40 PCA + quadratic classifier	None	None	3.3 <sup>[10]</sup>
Random Forest	Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC) <sup>[25]</sup>	None	Simple statistical pixel importance	2.8 <sup>[26]</sup>
Support-vector machine (SVM)	Virtual SVM, deg-9 poly, 2-pixel jittered	None	Deskewing	0.56 <sup>[27]</sup>
Deep neural network (DNN)	2-layer 784-800-10	None	None	1.6 <sup>[28]</sup>
Deep neural network	2-layer 784-800-10	Elastic distortions	None	0.7 <sup>[28]</sup>
Deep neural network	6-layer 784-2500-2000-1500-1000-500-10	Elastic distortions	None	0.35 <sup>[29]</sup>
Convolutional neural network (CNN)	6-layer 784-40-80-500-1000-2000-10	None	Expansion of the training data	0.31 <sup>[30]</sup>
Convolutional neural network	6-layer 784-50-100-500-1000-10-10	None	Expansion of the training data	0.27 <sup>[31]</sup>
Convolutional neural network (CNN)	13-layer 64-128(5x)-256(3x)-512-2048-256-256-10	None	None	0.25 <sup>[17]</sup>
Convolutional neural network	Committee of 35 CNNs, 1-20-P-40-P-150-10	Elastic distortions	Width normalizations	0.23 <sup>[12]</sup>
Convolutional neural network	Committee of 5 CNNs, 6-layer 784-50-100-500-1000-10-10	None	Expansion of the training data	0.21 <sup>[19][20]</sup>
Random Multimodel Deep Learning (RMDL)	10 NN-10 RNN - 10 CNN	None	None	0.18 <sup>[22]</sup>
Convolutional neural network	Committee of 20 CNNs with Squeeze-and-Excitation Networks <sup>[32]</sup>	None	Data augmentation	0.17 <sup>[33]</sup>

FIGURE 17 – Tableau d'architectures de modèles traditionnels

Dans un premier temps, nous avons testé une architecture formée par trois couches, avec une seule couche cachée composée par 800 neurones. Cependant, les résultats n'étaient pas concluant. Puis nous avons testé avec une architecture de 7 couches, tels qu'il existe 5 couches cachées : 2500 2000 1500 1000 500. Le soucis étant qu'il existe plus d'un million de poids à mettre à jour à chaque propagation arrière, l'entraînement fût extrêmement long, ce qui est problématique. Finalement, nous avons retenu une architecture composé de trois couches, avec une couche cachée comportant 200 neurones. Toutes les couches ont comme fonction d'activation la fonction sigmoid.

## Numeric

La base de donnée *numeric* étant un peu plus petite que MNIST (on passe tout de même de 60'000 à 10'000), nous avons pu converger vers un résultat acceptable beaucoup plus rapidement. Nous avons commencé avec des architectures avec peu de neurones sur l'unique couche cachée, par exemple 64 neurones. Nous utilisons alors que *sigmoid* comme fonction d'activation. Avec le temps, nous avons augmenté les neurones sur la couche cachée. Nous sommes passés de 64 à 128, puis à 200 et enfin à 800. Nous avons également commencé à utiliser la fonction *softmax* comme fonction d'activation pour la dernière couche. Les architectures contenant 800 neurones sur la couche interne n'apprenaient pas assez bien. C'est pourquoi nous avons trouvé que la meilleure architecture était :

- 200 neurones sur la couche cachée
- fonction d'activation de la couche cachée : sigmoid
- fonction d'activation de la dernière couche : softmax

## Noisy

La base de donnée *noisy* étant une base générée par nos soins, nous avons d'abord entraîné sur une base avec beaucoup de bruit ; un seuil à 1.0 (cf. Base de donnée 'Noisy'). En plus d'être une base de donnée qui est difficile à appréhender pour un réseau de neurones avec uniquement des couches dites 'denses' (donc sans couches 'convolutionnelles' ou de 'max-pooling'), la base n'était pas représentative des images des cellules qui allaient être extraites des sudokus. Nous avons donc re-généré la base de donnée avec un nouveau seuil de 2.0.

Les essais d'entraînement sur cette base n'étaient pas très convaincants. Mais lorsque nous avons fusionné cette base de donnée avec la base 'Artisanal', les résultats étaient stupéfiants, si on les comparait à ceux avant la fusion. Il est important de noter que nous avons fusionné ces bases en ajoutant 40 copies de la base 'Artisanal' et une seule fois 'Noisy'.

Quant aux architectures testées, nous avons utilisé exactement le même processus que pour la base 'Numeric' : nous sommes partis de modèles plus petits, jusqu'à trouver une limite de grandeur. En l'occurrence, nous avons également testé avec des tailles de couche intermédiaires de 64, puis, 128, puis 200 et enfin 800. Le modèle le plus performant avait comme architecture :

- 128 neurones sur la couche cachée
- fonction d'activation de la couche cachée : sigmoid
- fonction d'activation de la dernière couche : sigmoid

## Artisanal

La base de donnée *Artisanal* étant une base de donnée relativement petite, nous avons en premier lieu effectué quelques tests d'architecture. Nous avons ainsi repris l'architecture de *mnist* avec trois couches dont une cachée comportant 800 neurones. Puis nous l'avons testé en applications réelles : uniquement la première grille de sudoku fourni par le sujet était reconnu.

Nous avons ainsi formulé le problème d'une autre manière et jugeait que la base de donnée était trop petite pour être correctement reconnue et entraîné dessus. Nous avons ainsi pris la base de donnée *Artisanal* et l'avons mélangé avec la base de donnée *Numeric* pour former l'alliage de donnée *Mix*.

Puis, nous avons entraîné un réseau de neurones formé par quatre couches dont deux couches cachées avec respectivement 128 et 64 neurones. Les résultats furent excellents et lors de la phase de test, le réseau reconnut 90% des nombres proposés au réseau, ce qui est une grande réussite.

## EMNIST

La base de donnée EMNIST étant si énorme, nous n'avons pas beaucoup essayé d'entraîner des modèles dessus. Nous avons entraîné deux modèles :

D'abord un modèle plus petit avec 200 neurones sur la couche intermédiaire et *sigmoid-softmax* comme fonctions d'activations. Nous avons obtenu 93.30 % de bonnes prédictions sur les données de test, mais sur les images de sudoku, ce taux est tombé à 50.00 %.

Nous avons par la suite formé un réseau de neurones très grand, composé de cinq couches dont trois couches cachées avec respectivement 1024 512 et 128 neurones pour chacune des couches cachées. Nous avons ainsi entraîné ce modèle sur l'ensemble de EMNIST (216 000 samples !) sur 5 époques et les résultats sont corrects, mais malheureusement pas au point d'être satisfaisants. En effet, à la fin de la phase de test, on reconnaît alors que le modèle possède un taux moyen de réussite de reconnaissance de nombre de 77%, ce qui n'est pas suffisant pour nous, pour utiliser ce modèle comme modèle de référence pour le projet.

## 6.3 Conclusion - Réseaux de neurones

Nous pensons avoir atteint nos objectifs. En effet, le script suivant fonctionne à merveille :

```
1 #include "nn/nn.h"
2
3 int main()
4 {
5     /* initialisation de :
6      * aléatoire
7      * prévention fuites mémoire
8      * affichages 'verbose'
9      */
10    nn_initRandom();
11    initMemoryTracking();
12    setVerbose(true);
13
14    // architecture du modèle
15    nn_ShapeDescription model_architecture[] = {
16        nn_create2DShapeDescription(28, 28), // 28 x 28 x 1 (image mnist classique)
17        nn_create2DShapeDescription(30, 30), // 30 x 30 x 1
18        nn_create1DShapeDescription(800), // 800 x 1 x 1
19        nn_create1DShapeDescription(200), // 200 x 1 x 1
20        nn_create1DShapeDescription(9), // 9 x 1 x 1
21    };
22    // activation functions
23    activation activations[] = {
24        TANH, TANH, SIGMOID, SOFTMAX // fonctions d'activations du modèle
25    };
26    // loss & optimizer
27    losses loss = CATEGORICALCROSSENTROPY; // fonction de taux d'erreur
28    optimizer optimizer = ADAM;
29
30    // allocation du model dans la mémoire
31    nn_Model* model = nn_createModel(5, model_architecture, activations, loss, optimizer);
32    // allocation des données
33    nn_DataSet dataset = nn_loadDataSet("datas/mnist/", true);
34    // session
35    nn_Session* session = nn_createSession(
36        &dataset,
37        15, // nombres d'époques d'entraînement
38        0.0, // seuil taux d'erreur
39        false, // arrêt de l'entraînement lorsqu'on atteint le seuil
40        true, // session 'verbose'
41        0.05, // taux d'apprentissage
42        "avg-loss.log", // chemin vers le fichier log de taux d'erreur
43        "avg-right.log" // chemin vers le fichier log de pourcentage de prédictions réussies
44    );
45    // phase d'entraînement
46    session->train_one_hot(session, model);
47    // phase de test
48    session->test_one_hot(session, model);
49
50    // sauvegarde du modèle
51    model->saveModel(model, "save/mnist/custom-model-");
52
53    /* on pourrait faire :
54     * nn_freeModel(model);
55     * nn_freeSession(session);
56     */
57    // mais on va faire
58    mem_freeGPL(false); // on libère l'espace mémoire qui a été alloué par la librairie
59
60    return 0;
61 }
```

Et valgrind nous dit :

```
==92690==
==92690== HEAP SUMMARY:
==92690==      in use at exit: 0 bytes in 0 blocks
==92690==    total heap usage: 18,435 allocs, 18,435 frees, 36,562,482 bytes allocated
==92690==
==92690== All heap blocks were freed -- no leaks are possible
==92690==
==92690== For lists of detected and suppressed errors, rerun with: -s
==92690== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Nous pensons que notre librairie permet à son utilisateur de manipuler des réseaux de neurones avec facilité et toute l'élégance que le C permet.

## 6.4 Le solveur de sudoku

### 6.4.1 Aspects techniques

Le solveur de sudoku est découpé en trois grands membres :

- Le premier est le lexer. Le rôle du lexer est de remplir une matrice statique de 9 par 9 à l'aide d'un fichier texte qui est renseigné par l'utilisateur lors de l'appel de l'exécutable en question. Du fait du format spécifique de la grille de sudoku représenté dans le fichier renseigné par l'utilisateur, le lexer doit prendre en compte l'espace tous les 3 caractères et le saut de lignes toutes les 3 lignes. Dès lors que le lexer lit un caractère non autorisé, un espace ou un saut de ligne mal placé, le lexer plante et renvoie un code erreur spécifiant à l'utilisateur que le formatage du fichier d'entrée est mauvais.
- Vient ensuite le solver. Le solver va résoudre la grille de sudoku représentée dans la matrice statique remplie par le lexer, à l'aide d'un algorithme récursif et possédant une caractéristique de *backtracking*. L'algorithme va tout simplement tester l'ensemble des possibilités de placement de nombre en fonction de l'état de départ de la grille. S'il n'est pas possible pour lui de remplir l'ensemble des cases en suivant les règles de sudoku, le solver, va alors revenir une étape en arrière et tester une nouvelle approche de placement de nombre.
- Enfin vient l'é writer. Une fois la matrice résolue par le solveur, le writer va écrire la matrice dans un fichier avec pour extension *.output* et en suivant les mêmes règles de formatages que le fichier entré par l'utilisateur.

## 7 Aspects techniques - Autres

En plus des tâches décrites plus loin dans ce rapport, nous avons implémenté certaines fonctionnalités qui n'appartiennent à aucune des catégories du tableau de la répartition des tâches.

### 7.1 Prévention de fuite mémoire

Afin d'éviter les fuites de mémoire, nous avons créé des wrappers<sup>14</sup> autour des fonctions 'malloc', 'calloc' et 'free', nommées respectivement 'mem\_malloc', 'mem\_calloc' et 'mem\_free'. Ces fonctions nous permettent de garder en mémoire les allocations de mémoires dans le heap, de vérifier qu'à la fin du programme, nous avons bien libéré l'espace mémoire alloué, et de moins se faire crier dessus par valgrind<sup>15</sup>.

Dans tous les cas, à la fin d'un programme, on peut appeler **mem\_freeGPL(false)**, ce qui libérera tous les pointeurs stockés dans la GPL. Avec cela en place, nous n'avons aucune fuite de mémoire dans la partie réseaux de neurones !

---

14. wrappers : [https://wikipedia.org/wiki/Wrapper\\_function](https://wikipedia.org/wiki/Wrapper_function)

15. valgrind : <https://wikipedia.org/wiki/Valgrind>

## 7.2 Bourne Again Test Framework

Afin de pouvoir coder de manière stable et sereine, sans avoir peur de tout casser à chaque commit, nous avons également créé un framework de tests unitaires en langage bash (d'où le nom *Bourne Again Test Framework*, car bash veut dire *Bourne Again Shell*).

Ce framework est utilisé pour tester des parties de notre programme. Dès que nous implémentons une fonctionnalité, nous testons toutes les autres fonctionnalités, afin de nous assurer que nous n'avons rien cassé (pour tester, il faut exécuter la commande 'make test').

## 8 Conclusion

C'est ici que se conclut ce projet. Sa réalisation nous a apporté une multitude de connaissances, tant théoriques au travers des différents domaines de mathématiques appliqués à ce dernier, que pratiques avec la manipulation du langage C.

Un des points forts de la réussite du projet a été la bonne entente et l'entraide au sein du groupe. L'avance que nous avons pour la première soutenance nous a permise d'entreprendre le développement de fonctionnalités plus complexes que celles requises par le cahier des charges, tout en optimisant certaines parties de notre projet.

Pour finir, nous pensons avoir atteint les objectifs fixés par le cahier des charges, voire plus ! Nous avons d'ailleurs entamé quelques boni, comme le site web <sup>16</sup>, le redressement de l'image (suppression de perspective) ou la reconnaissance des chiffres manuscrits. La réalisation et le développement de l'*OCR Sudoku Solver* nous a réellement plu, et nous avons hâte de découvrir les nouveaux projets qu'Epita peut nous proposer.

---

16. Lien : <https://reyland.dev/sudoku>