Projet S3

OCR

Rapport de projet

20 novembre 2020

VERNAY Rémi

DEPLAGNE Hugo
LITOUX Pierre
PINGARD Adrien
VEYRE Thimot

Sommaire :

I. Introduction

Ce projet a pour but de réaliser un ROC (Reconnaissance Optique de Charactères) dont la fonctionnalité principale est d'extraire depuis une image (bitmap) le texte contenu dans celle-ci.

Pour cela nous utilisons une IA qui se charge de reconnaitre les caractères d'une image, avant d'utiliser le réseau de neurones de notre IA il faut traiter l'image puis séparer les lignes, les mots et les caractères.

Le programme doit d'être utilisable facilement avec une application qui prendra une image en entrée dans un format standard et produira un texte reconnu.

Ce ROC est réalisé dans le cadre du projet de troisième semestre du cycle préparatoire de l'EPITA. Notre groupe est formé des mêmes membres que pour le projet de fin de première année. En effet, nous nous étions bien entendu et avons réussi à rendre un projet dont nous étions tous fiers.

Ce rapport a pour but de montrer la répartition des tâches entre les membres du groupe, d'expliquer ce que nous avons fait et comment, et si nous avons rencontré des difficultés.

Nous terminerons par parler de l'avancement du projet et ce qu'il reste à faire pour finir le projet.

II. Présentation du groupe

a. Hugo Deplagne

Ancien élève de PCSI en CPGE, je me suis fortement intéressé à l'informatique ce qui explique mon changement de voie pour l'EPITA avec l'envie d'intégrer en cycle ingénieur la majeure en intelligence informatique. Ce projet est donc un parfait exercice pour la majeure que je choisirai plus tard.

b. Pierre Litoux

Je me suis intéressé à l'informatique grâce aux algorithmes fait en maths et grâce à la spécialité ISN. J'ai rejoint l'EPITA initialement pour la majeure intelligence artificielle, ce projet est donc un bon avant-gout de mon futur choix.

c. Adrien Pingard

Etudiant en deuxième année du cycle préparatoire à l'EPITA. Je suis passionné par la programmation depuis six ans. Je ne connais que très peu de chose en intelligence artificielle et ce projet me permettra d'en connaitre plus.

d. Thimot Veyre

Je suis, un ancien élève de S-SI, actuellement en deuxième année du cycle préparatoire à l'EPITA. Je suis passionné par l'informatique depuis mon entré au lycée et ce projet me permet de découvrir un tout nouveau domaine, l'IA. C'est un domaine assez flou pour moi, je n'en connais que les grands principes de fonctionnement. Je suis, de plus, motivé car nous devrions obtenir un logiciel réellement utilisable.

III. Répartition des taches

	Hugo Deplagne	Pierre Litoux	Adrien Pingard	Thimot Veyre
Chargement des images			**X**	
Suppression des couleurs	**X**		X	
Découpages des caractères				**X**
Réseau de neurones XOR		**X**	X	X
Pré-traitement de l'image	**X**			
Sauvegarde du réseau de neurones		**X**		
Gestion des fichiers avec les résultats	X			
Jeu d'images				**X**

X : Principal

X : Secondaire

IV. Eléments obligatoires

a. Chargement des images

Pour l'importation des images dans notre logiciel, nous avons choisi de gérer, pour l'instant, un seul format d'image, le bitmap. Le format bitmap est capable de stocker des images en monochromes ou en couleurs.

Format bitmap

Pour le chargement de l'image à partir du fichier BMP (bitmap), nous avons dû extraire les données binaires, octet par octet, du fichier pour y récupérer les informations du fichier : l'entête principal du fichier, l'entête informations et les données des pixels.

En effet, le format bitmap est composé de trois parties principales :

- L'entête principal du fichier, qui contient les informations principales du fichier bitmap. Il est constitué sur quatorze octets de la manière suivante :

Adresse Hexadécimale	Taille	Donnée
0x00	2 octets	Le nombre magique correspondant au fichier BMP (dans notre cas celui sera égal à **BM**)
0x02	4 octets	La **taille du fichier** en octets (entêtes comprises)
0x06	2 octets	Réservé pour l'identifiant de l'application qui a créé le fichier
0x08	2 octets	Réservé pour l'identifiant de l'application qui a créé le fichier
0x0A	4 octets	L'**adresse de départ** du contenu des pixels

Figure 1 – Entête principal bitmap

- L'entête d'informations ('*DIB header*'), il existe sept versions différentes de cet entête, mais pour notre logiciel nous n'en utiliserons qu'un seul, le *'BITMAPINFOHEADER'*, celui-ci est constitué de quarante octets de la manière suivante :

Adresse Hex	Taille	Donnée
0x0E	4 octets	La taille de cet entête en octets (40 octets)
0x12	4 octets	La largeur de l'image en pixels
0x16	4 octets	La hauteur de l'image en pixels
0x1A	2 octets	Le nombre de plans de couleur (doit être 1)
0x1C	2 octets	Le nombre de bits par pixel (*'bpp'*)
0x1E	4 octets	La méthode de compression utilisée (0 s'il n'y a pas de compression)
0x22	4 octets	La taille de l'image (0 s'il n'y a pas de compression)
0x26	4 octets	La résolution horizontale de l'image
0x2A	4 octets	La résolution verticale de l'image
0x2E	4 octets	Le nombre de couleur dans la palette de couleur
0x32	4 octets	Le nombre de couleurs importantes utilisées

Figure 2 – Entête d'informations bitmap

- Les données des pixels sont disposées ligne par ligne, en commençant par la ligne inférieur de l'image codé de gauche à droite. Chaque ligne de l'image doit occuper un nombre d'octets multiple de quatre, ce qui nous donne la formule suivante :

$$taille\ d'une\ ligne\ (octets) = \left\lceil \frac{bpp\ \times\ ImageLargeur}{32} \right\rceil \times 4$$

(bpp : bits par pixel)

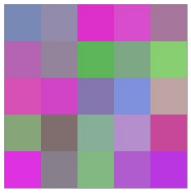
Et donc la taille totale des pixels occupe :

$$taille\ de\ l'image\ (octets) = taille\ d'une\ ligne\ \times ImageHauteur$$
$$= \left\lceil \frac{bpp\ \times\ ImageLargeur}{32} \right\rceil \times 4 \times ImageHauteur$$

Pour simplifier l'importation des images, nous avons restreint les possibilités des types de bitmap. Notre logiciel, pour l'instant, accepte seulement les bitmaps de ce format :

Nombre magique	'BM' soit 0x424D
L'entête d'information	'*BITMAPINFOHEADER*'
Largeur de l'image	Supérieur à 0
Hauteur de l'image	Supérieur à 0
Bits par pixel	24bits
Méthode de compression	0 soit 'pas de compression'

Prenons l'exemple d'une image :



- Taille = 5 x 5 pixels (largeur x hauteur)
- Bits par pixel = 24 bits (8 bits x 3)
- Compression = 0

```
  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 42 4D 86 00 00 00 00 00 00 00 36 00 00 00 28 00   BM........6...(.
00000010: 00 00 05 00 00 00 05 00 00 00 01 00 18 00 00 00   ................
00000020: 00 00 00 00 00 00 C4 0E 00 00 C4 0E 00 00 00 00   ......D...D.....
00000030: 00 00 00 00 00 00 E1 31 DE 8B 7F 87 81 B8 82 CE   ......a1^....8.N
00000040: 5B B0 E0 36 B9 00 79 A5 86 6F 6E 80 99 AE 87 CD   [0`69.y%.on....M
00000050: 8E B4 98 48 C7 00 B6 50 D6 C5 44 D0 B0 76 86 DC   .4.HG.6PVEDP0v.\
00000060: 91 7F A4 A5 BE 00 B2 64 B4 9B 84 93 58 B6 5D 85   ..$%>.2d4...X6].
00000070: A8 7E 71 CF 88 00 B6 89 79 AC 8B 93 CA 2F DD CB   (~qO..6.y,..J/]K
00000080: 4D D8 9C 76 A7 00                                 MX.v'.
```

Représentation hexadécimale des données de l'image ci-dessus

Nous pouvons voir, sur la représentation hexadécimale, les trois différentes parties, avec l'entête principale, l'entête d'informations et les données des pixels. D'après la formule de la taille d'une ligne énoncé précédemment, avec cette image nous obtenons :

$$taille\ d'une\ ligne = \left\lceil \frac{bpp \times ImageLargeur}{32} \right\rceil \times 4 = \left\lceil \frac{24 \times 5}{32} \right\rceil \times 4 = 16\ octets$$

Nous savons également, d'après la représentation hexadécimale, l'adresse de départ des données des pixels avec la valeur de 4 octets en adresse 0x0A (police rouge). Ainsi l'adresse de départ est égale à **0x36**.

A partir de l'adresse de départ (**0x36**) et taille d'une ligne (**16 octets**), nous pouvons extraire les données de la première ligne inferieur de l'image :

```
00000036: E1 31 DE 8B 7F 87 81 B8 82 CE 5B B0 E0 36 B9 00
```

Nous constatons qu'il y a un octet qui ne correspond à aucun pixel, en fin de ligne en vert. Cet octet peut être égal à n'importe quelle valeur, il ne sert qu'à compléter la ligne pour que le nombre d'octets soit un multiple de 4.
Nous pouvons donc voir les valeurs de chaque pixel de cette ligne, 3 octets par 3 octets. Si nous prenons le premier pixel de cette ligne, nous obtiendrons un pixel de valeur BGR (bleu/vert/rouge) de **0xE131DE**.

Implémentation dans notre projet en C

Pour intégrer le chargement des bitmaps, nous avons créé deux fichiers :

- 'bitmap.c' : fichier contenant toutes les fonctions relatives au chargement et la manipulation d'un bitmap.
- 'bitmap.h' : le 'Header' du fichier .c précèdent

Pour gérer les fichiers bitmap, nous avons donc créé une structure '*BMPIMAGE*', une '*RGB*' et une '*BMPHEADER*'. La structure '*RGB*' contient 3 variables entières positives de 8bits correspondant aux trois couleurs d'un pixel. La structure '*BMPHEADER*' permet de stocker toutes les valeurs nécessaires des entêtes du bitmap (Entête principale et l'entête d'information).

```
struct BMP_HEADER_STCT
{
          uint16_t type; /* The type of the format. need to be 0x424d */
          uint32_t bfSize; /* The size of the file (header include) in bytes */
          uint16_t unused1;
          uint16_t unused2;
          uint32_t imageDataOffset; /* Offset to the start of image data_rgb */
          uint32_t headerSize; /* The size of the second part of the header in
bytes*/
          uint32_t width; /* The width of the image in pixels*/
          uint32_t heigth; /* The heigth of the image in pixels */
          uint16_t num_planes;
          uint16_t bits_per_pixel; /* The size of an unique pixel. need to be 24
bits*/
          uint32_t compression;
          uint32_t unused3;
          uint32_t unused4;
          uint32_t unused5;
          uint32_t unused6;
          uint32_t unused7;
} __attribute__((packed));
typedef struct  BMP_HEADER_STCT BMPHEADER;
```

Figure 3 – Structure '*BMPHEADER*'

La structure 'BMPIMAGE' est la structure qui permet de stocker l'intégralité d'un bitmap avec les valeurs de ses entêtes et les valeurs de tous ses pixels. C'est avec cette structure que toutes les fonctions de pré-traitement vont travailler.

```
typedef struct
{
         BMPHEADER header; /* The information of the bitmap image*/
         RGB **data; /* Matrix that contains the RGBs value of each
pixels*/
} BMPIMAGE;
```

Figure 4 – Structure '*BMPIMAGE*'

Nous avons implémenté une fonction '*LoadBitmap*' qui a le prototype suivant :

```
BMPIMAGE *LoadBitmap(char *filename);
```

C'est donc cette fonction qui permet le chargement des images dans notre logiciel. Nous avons rajouté la fonction inverse, qui a partir d'une structure '*BMPIMAGE*' et d'un chemin d'accès, enregistre une image sous le format bitmap. Cela nous sert à tester nos fonctions de pré-traitement, en visualisant les changements sur les images. La fonction est la suivante :

```
void SaveBitmap(BMPIMAGE *image, char *filename);
```

Nous avons codé d'autres fonctions usuelles permettant de simplifier la manipulation des images :

- La fonction '*FreeBitmap'* qui permet de libérer de la mémoire une image
- La fonction 'GetPixel' qui permet de récupérer la valeur 'RGB' d'un pixel d'une image
- La fonction 'GetRGB' permet d'extraire les valeurs de rouge, de bleu et de vert d'une valeur 'RGB'
- La fonction 'PrintBitmap' qui permet d'afficher le contenu de la matrice de pixels
- La fonction 'SubBitmap' qui permet une redéfinition plus petite d'une image

b. Suppression des couleurs

Nous supprimons les couleurs en faisant un niveau de gris puis en binarisant l'image en noir et blanc. Comme énoncé précédemment les pixels ont trois composantes de couleur rouge, vert et bleu. Ces composantes sont des entiers compris entre 0 et 255.

Pour faire le niveau de gris nous prenons pour chaque pixel de l'image a transformé les valeurs des composantes rouge (r), vert (v) et bleu (b) et nous les remplaçons par une seule valeur : (r+v+b)/3.

Une fois le niveau de gris effectué nous passons à la binarisation qui regarde si le niveau de gris est supérieur à 127. Si oui le pixel est changé en noir (c-à-d r=0 v=0 b=0) sinon en blanc (r=255 v=255 b=255).

N.B: Il est recommandé avant de faire la suppression des couleurs de déparasiter l'image. La fonction pour déparasiter est décrit dans la section prétraitement.

c. Détection des caractères

La détection de caractères utilise des images avec du texte noir sur fond blanc uniquement. En effet le système se base sur la détection de ligne et de colonne blanche pour délimiter les caractères.

La fonction commence dans un premier temps par détecter toutes lignes du texte (voir figure 5). Pour cela, nous enregistrons dans un tableau la ligne de début, celle-ci et la première ligne non totalement blanche que nous trouvons. Nous cherchons ensuite la prochaine ligne totalement blanche, et nous enregistrons la ligne précédente (la dernière ligne non totalement blanche)

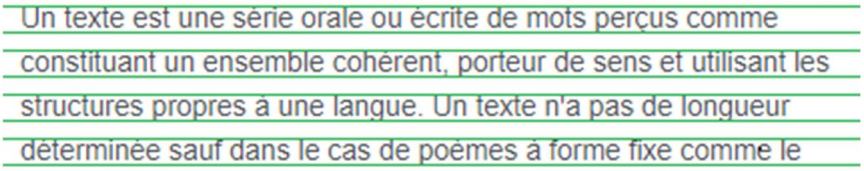


Figure 5 – Détection des lignes

Nous créons ensuite un tableau de nouvelles images, chacune correspond une des lignes précédemment trouvées. Nous recommençons ensuite le même processus mais de manière verticale cette fois ci (voir Figure 6).

Figure 6 – Détection des caractères

Une fois cela fait, nous créons à nouveau tableau d'image ne contenant que les caractères détectés (il contient aussi tous les caractères n'étant pas des lettres).

Une fonction générale gère ces différents passages et renvoie un pointeur vers le tableau d'image. Elle prend en entrée deux paramètre, une image à scanner et un entier indiquant si elle doit ou non enregistrer les caractères dans un dossier.

Nous aurions pu optimiser la détection en ne construisant pas les images des lignes mais en repassant directement sur l'image au complet mais cette méthode nous permettra de plus facilement intégrer la gestion de la forme du texte.

L'inconvénient majeur de cette méthode de découpe est que si les caractères ne sont pas séparés pas au moins une colonne blanche la détection ne se fait pas et nous nous retrouvons avec plusieurs lettres sur une seule image. Nous n'avons trouvé aucun moyen plus efficace de gérer la détection. Le programme ne pourra donc reconnaitre uniquement des textes avec des caractères assez espacés.

d. Fonction XOR

Pour comprendre le fonctionnement d'une IA nous nous sommes attelés à la réalisation et la compréhension d'une IA gérant le fonctionnement d'un XOR.
Pour cela nous sommes partis de la table de vérité d'un XOR et nous avons utilisés ces données comme référence pour l'entrainement de l'IA.

Table de vérité de XOR (OU EXCLUSIF)		
a	**b**	**a XOR b**
0	0	0
0	1	1
1	0	1
1	1	0

Figure 7 – Table de vérité du XOR

Le réseau de neurones implémenté contient donc deux inputs et une seul sortie output. Nous avons aussi une couche de neurones sigmoid cachés qui traitent les poids des deux inputs et renvoient leur résultat au neurone output.

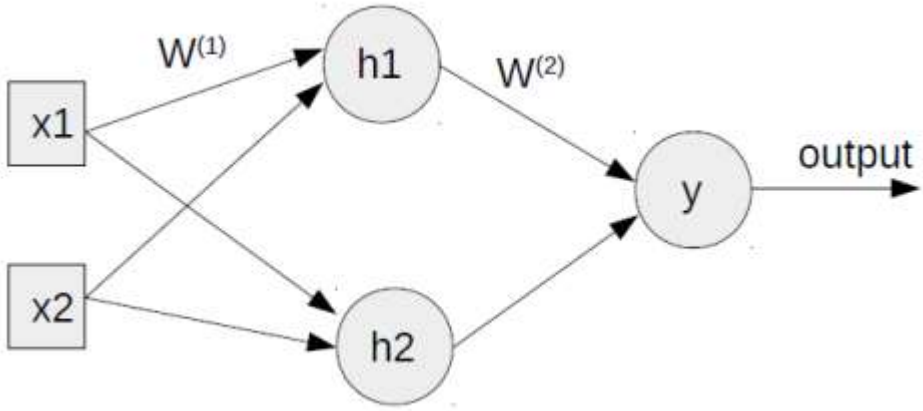


Figure 8 – Réseaux de neurone du XOR

Nous avons donc implémenté la fonction XOR de la manière suivante :

Etape 1 : initialisation des biais et des poids au hasard suivant une loi uniforme.
Etape 2 : Itération 10 000 fois de l'algorithme d'entrainement.
Etape 3 : Récupération des valeurs des biais et des poids.

L'algorithme d'entrainement a pour but d'actualiser les poids et les biais de manière à minimiser le taux d'erreurs.

L'algorithme se décompose en trois étapes :

- « Forward propagation »
- « Gradient descent »
- Actualiser les poids et les biais

La « forward propagation » consiste à évaluer le résultat sortit par l'IA avec les biais et poids actuel en utilisant la fonction sigmoid de manière que :

$$y = \frac{1}{1 + e^{-(\sum wixi + \ )}}$$

Avec $\sum wixi$ somme des poids arrivant vers le neurone et b biais du neurone.

On va ensuite comparer ce résultat obtenu au résultat voulu pour trouver l'erreur. Pour cela nous faisons un algorithme de « gradient descent » qui revient à trouver le minimum d'une fonction erreur en fonction du poids. Cette erreur est calculée grâce a la différence entre le résultat voulu. Puis elle est multipliée par la dérivée de sigmoid des résultats obtenus lors de la « forward propagation ».

On peut ensuite actualiser les poids et les biais en ajoutant à leurs valeurs les erreurs trouvées. Celle-ci sont multipliée par learning rate (=0.1) pour ne pas changer trop brusquement les valeurs des poids.

Répéter cet algorithme un grand nombre de fois a donc pour but de minimiser les erreurs.

Pour passer de ce mini réseau a un réseau capable de traiter une image nous comptons augmenter le nombre d'input. Pour des lettres codées par des 0 et des 1 sur une image de taille x * y, nous allons donc avoir x * y neurones d'input et tout autant de neurones cachés sur la première couche. Le nombre de neurones de sortie seras équivalent au nombre de caractère que l'algorithme pourra reconnaitre. Pour ce qui es des neurones cachés le nombre de couches et de neurones reste à déterminer.

Débogage et amélioration du réseau :

La création du réseau a créé de nombreux problème notamment sur le stockage des biais et des poids. Nous avons opté pour une implémentation dynamique et nous avons stockés les valeurs dans des matrices de « double » mais si cette implémentation permet de bien visualiser les matrices elle demande aussi de free à la fin du programme et se relève peut-être moins efficace qu'une implémentation par liste fixes formant des pseudo matrices.

Nous avons aussi eu des problèmes de compréhension car si nous arrivons à implémenter les fonctions mathématiques dans le code nous ne les comprenons pas forcément. Surtout la fonction de « gradient descent » dont le fonctionnement et le calcul de l'erreur restent abstraits.

De plus nous avons implémenté le XOR de manière à pouvoir changer rapidement le nombre de neurones mais nous ne sommes pas sur si cette structure sera efficace dans le cadre de la reconnaissance de caractère.

V. Eléments entamés

a. Pré-traitement

Le prétraitement améliore la qualité des images traitées. Cette partie doit permettre d'accepter en entrée des documents directement issus d'un scanner ou d'une photo. Les prétraitements sont :

- Le redressement manuel de l'image
- Le redressement automatique de l'image
- L'élimination des bruits parasites
- Renforcement des contrastes

Ce que nous avons réalisé :

Nous pouvons régler la rotation de l'image manuellement pour des angles de 90, -90 et 180 degrés en changeant les pixels d'emplacement sur l'image. Pour la suite il restera à implémenter une fonction qui fait automatiquement cette rotation.

Pour déparasiter l'image nous effectuons une vérification sur chaque pixel en fonction des pixels environnants. Il s'agit de remplacer une valeur statistiquement aberrante par la moyenne des voisins soit la moyenne des valeurs ((r+g+b)/3) des 8 pixels environnants. Pour cela, nous calculons la moyenne et l'écart type des valeurs des voisins. Nous calculons ensuite l'écart entre le pixel central et cette moyenne.

Si cet écart est plus grand que l'écart type, alors nous considérons que la valeur du pixel central est aberrante et nous le remplaçons par la moyenne des voisins. Les calculs sont présentés ci-dessous :

Avec P la valeur du pixel.

$$Moyenne = \frac{\left(\sum_{i,j=-1,-1}^{1} Px+i, y+j\right) - Pi,j}{(8^2-1)}$$

$$Ecart\ type = \sqrt{\frac{\left(\sum_{i,j=-1,-1}^{1} (Px+i, y+j - Moyenne)^2\right) - (Pi,j - Moyenne)^2}{(8^2-1)}}$$

b. Sauvegarde et chargement du réseau de neurones

Pour sauvegarder le réseau de neurones nous devrons stocker les valeurs de tous les poids et de tous les biais dans un fichier txt.

Ensuite pour charger le réseau de neurones il suffit d'initialiser le réseau de neurones mais au lieu de mettre les poids et les biais au hasard comme à l'entrainement, nous récupérons ceux stockés dans un fichier.

c. Jeu d'images

Nous avons trouvé un jeu d'images assez complet sur internet, il contient plusieurs images pour chaque lettre (Figure 9). Il est assez complet pour commencer à entrainer l'IA. Néanmoins, il manque encore un jeu d'images pour la ponctuation et les chiffres.

Le jeu d'image contient également des fichiers textes (Figure 10), contenant l'image sous forme de zéro et d'un, permettant de vérifier si le pré-traitement fonctionne correctement. Nous pouvons faire cela, en comparant notre matrice avec celle obtenu dans le fichier texte.

Figure 9 – Lettre a

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000111111110000010000
00000000111111111111111110000
00000001111100000001111110000
00000011111000000000011110000
00000111110000000000001110000
00001111100000000000000110000
00001111000000000000000010000
00011110000000000000000010000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00011110000000000000000000000
00001111000000000000000000000
00001111100000000000000000000
00000111110000000000000010000
00000011111100000000000110000
00000001111110000000111000000
00000000011111111111110000000
00000000000011111110000000000
00000000000000000000000000000
```

Figure 10 – Lettre C en 0 et 1

d. Manipulation de fichiers contenant les résultats

Cette partie n'est pas encore entamée car nous n'en avons pas encore besoin. Elle permettra à terme de créer le fichier texte contenant le texte présent sur l'image.

VI. Conclusion

Nous avons rencontré une difficulté majeure qui est l'implémentation de l'IA XOR en langage C. Le projet a donc pris un peu de retard sur ce point. Néanmoins, nous ne nous inquiétons pas et pensons rapidement rattraper ce retard.

A l'avenir, nous devons créer une application qui permet à n'importe quelle personne d'utiliser notre programme simplement. Pour cela, il reste à terminer fonction de rotation automatique de l'image, éventuellement rajouter une fonction qui renforce les contrastes, modifier notre réseau de neurone pour qu'il puisse reconnaitre des caractères, et enfin réaliser une interface graphique complète.