

# Projet S3

## OCR

### Rapport de projet

15 décembre 2020

DEPLAGNE Hugo  
LITOUX Pierre  
PINGARD Adrien  
VEYRE Thimot

VERNAY Rémi

# Sommaire :

- I. Introduction**
- II. Présentation du groupe**
  - a. Hugo Deplagne**
  - b. Pierre Litoux**
  - c. Adrien Pingard**
  - d. Thimot Veyre**
- III. Répartition des tâches**
- IV. Traitement des images**
  - a. Chargement des images**
  - b. Suppression des couleurs**
  - c. Détection des caractères**
  - d. Pré-traitement**
- V. IA**
  - a. Fonction XOR**
  - b. Réseau de neurones de l'OCR**
  - c. Sauvegarde et chargement du réseau de neurones**
- VI. Interface graphique**
  - a. Fenêtre**
  - b. Canvas**
  - c. Multithreading**
- VII. Compilation**
- VIII. Conclusion**

## **I. Introduction**

Ce projet a pour but de réaliser un ROC (Reconnaissance Optique de Caractères) 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 reconnaître 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.

## **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 connaître 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 entrée 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 tâches

	Hugo Deplagne	Pierre Litoux	Adrien Pingard	Thimot Veyre
Chargement des images			<b>X</b>	
Suppression des couleurs	<b>X</b>		x	
Découpages des caractères				<b>X</b>
Réseau de neurones XOR		<b>X</b>	x	x
Pré- traitement de l'image	<b>X</b>		x	x
Sauvegarde du réseau de neurones			<b>X</b>	
Base de données			<b>X</b>	x
Interface graphique	<b>X</b>	x	x	x
Réseau de neurone		x	<b>X</b>	
Compilation		x		<b>X</b>

**X** : Principal

x : Secondaire

## IV. Traitement des images

### 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 <b>BM</b> )
0x02	4 octets	La <b>taille du fichier</b> 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' <b>adresse de départ</b> 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 (' <i>bpp</i> ')
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 :

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

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 05 00 00 00 01 00 18 00 00 00	.....D...D.....
00000020:	00 00 00 00 00 00 C4 0E 00 00 C4 0E 00 00 00	.....a1^.....8.N
00000030:	00 00 00 00 00 00 E1 31 DE 8B 7F 87 81 B8 82 CE	[0`69.y%.on....M
00000040:	5B B0 E0 36 B9 00 79 A5 86 6F 6E 80 99 AE 87 CD	.4.HG.6PVEDP0v.\
00000050:	8E B4 98 48 C7 00 B6 50 D6 C5 44 D0 B0 76 86 DC	..\$%>.2d4...X6].
00000060:	91 7F A4 A5 BE 00 B2 64 B4 9B 84 93 58 B6 5D 85	(~q0..6.y,..J/]K
00000070:	A8 7E 71 CF 88 00 B6 89 79 AC 8B 93 CA 2F DD CB	MX.v'.
00000080:	4D D8 9C 76 A7 00	

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 :

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



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 inférieure 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 précédent

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 height; /* The height 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)

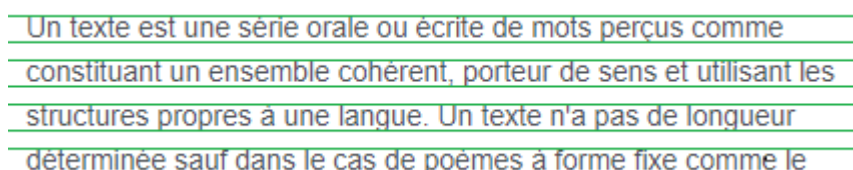


Figure 5 – Détection des lignes

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 paramètres, 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 par 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 reconnaître uniquement des textes avec des caractères assez espacés.

Pour finir notre IA, ne prend en paramètre que des images de vingt-huit par vingt-huit pixels. Nous devons donc dans un premier faire une image carrée, multiple vingt-huit avec le caractère toujours centré au milieu. Ensuite nous redimensionnons l'image pour qu'elle atteigne la taille voulue.

#### d. 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
- L'inversement des couleurs

#### La rotation :

Nous pouvons régler la rotation de l'image manuellement pour n'importe quel angle en créant une nouvelle image et en changeant les pixels d'emplacement sur la nouvelle image.

Pour savoir où positionner les pixels sur la nouvelle image nous appliquons un calcul de matrice comme suit :

#### Rotation

- La rotation d'un pixel  $(i, j)$  d'angle  $\theta$  (dans un repère au centre de l'image) s'exprime :

$$\begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

Ce principe fonctionne bien pour des angles multiples de 90° ou proche mais nous perdons des données sur les autres angles et notamment sur un angle de 45° qui provoque la superposition de certains bits et donc la perte de ceux-ci. Cependant l'image reste valide pour le réseau de neurones et la segmentation.

#### Le déparasitage :

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.

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

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

#### Le renforcement des contrastes :

Sur chaque image que nous traitons, nous avons décidé d'appliquer une augmentation de contraste. Cela permet de faciliter le passage au noir et blanc plus tard.

#### Le négatif :

Le réseau fonctionne mieux avec des lettres en blanc sur fond noir, nous avons donc implémenté une fonction permettant d'inverser les couleurs.

L'algorithme est très simple, il suffit, pour chaque couleur de soustraire la valeur rouge au maxime rouge et idem pour le vert et le bleu.

## V. IA

### a. 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'entraînement 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 seule sortie output. Nous avons aussi une couche de neurones sigmoïde cachés qui traitent les poids des deux inputs et renvoient leur résultat au neurone output.

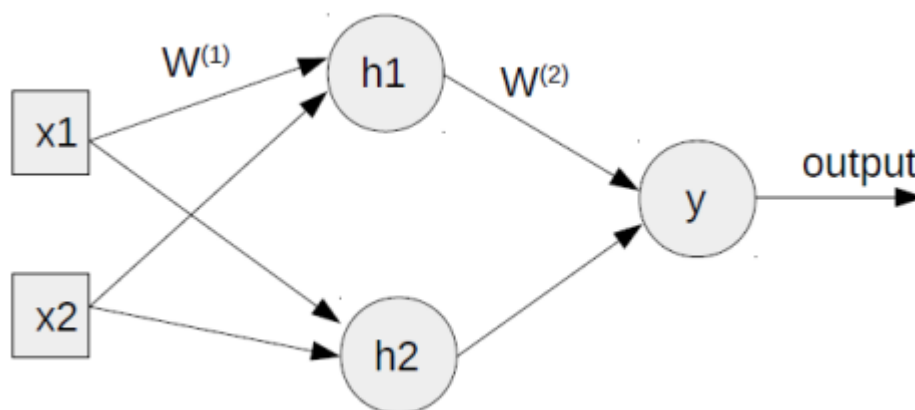


Figure 8 – Réseaux de neurone du XOR

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'entraînement.

Etape 3 : Récupération des valeurs des biais et des poids.

L'algorithme d'entraînement 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 :

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

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 à 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.



## b. Réseau de neurones de l'OCR

Grace au réseau de neurones que nous avons réalisé pour la fonction XOR, nous nous sommes posé la question de comment transformer celui-ci pour qu'il fonctionne sur des images et qu'il puisse déduire des caractères à partir de celle-ci.

Tout d'abord, contrairement au XOR, nous n'avons pas d'entrées et de sorties pour entraîner notre réseau de neurones. C'est pour cela que nous avons besoin d'une base de données pour pouvoir commencer notre intelligence artificielle.

### 1) Premier jeu d'images

Nous avons donc cherché une base de données sur internet qui associe des images de caractères à leurs caractères, et nous sommes tombés sur une qui contient exactement 124800 lettres manuscrites entre A et Z. Même si celle-ci ne permet pas d'identifier autre chose que les lettres de l'alphabet, elle est amplement suffisante pour commencer notre I.A. Cette base de données est la « EMNIST database », elle contient deux fichiers différents ; un fichier contenant les images et un fichier contenant les caractères correspondants.

Les images ont une dimension de 28 x 28 pixels, et chaque pixel est codé sur un octet. Nous avons donc créé des fonctions permettant de charger et sauvegarder des fichiers dans le format des fichiers « EMNIST ».

Voici la structure des fichiers « EMNIST » :

#### TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

#### TRAINING SET IMAGE FILE (train-images-idx3-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

## 2) Première version de notre IA

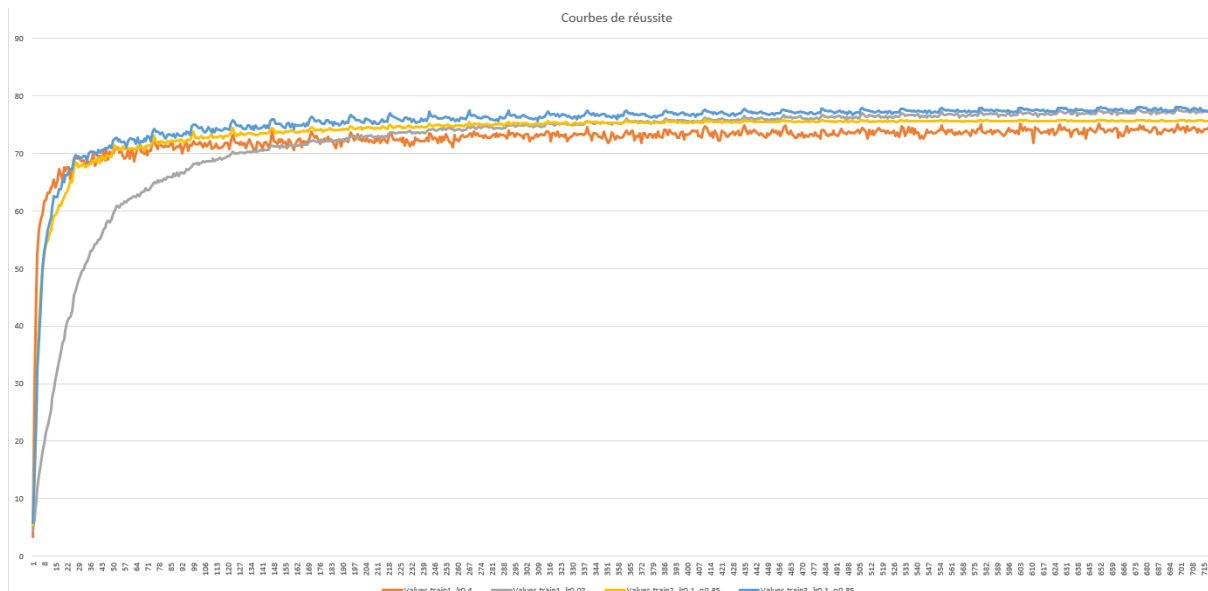
Au vu du peu de temps que nous avons pour comprendre et implémenter une IA fonctionnelle en langage C, nous nous sommes d'abord lancés dans la réalisation de celle-ci dans un langage de plus haut niveau, le JavaScript. Ce qui nous permettait d'avoir un bon débogueur pour comprendre et réussir notre IA.

Avant tout nous nous sommes interrogés sur la structure du réseau de neurones (le nombre d'entrées, le nombre de couches cachées, et le nombre de sorties). Le nombre d'entrée a été défini par la base de données. En effet, les images de la base de données ont une taille de  $28 * 28$  pixels soit 784 pixels donc le nombre d'entrées de notre réseau de neurones est de 784. Le nombre de sorties dépend du nombre de caractère que nous voulons pouvoir détecter, ici, nous nous limitons à l'alphabet donc vingt-six. Pour le nombre de couches cachées de notre réseau, cela ne dépend pas de la base de données. Dans un premier temps nous avons choisis de n'en mettre qu'une par soucis de facilité.

Après avoir repris et modifié les fonctions du XOR, nous avons lancé nos premiers tests. Nous avons lancé plusieurs entraînements avec différents paramètres :

- Le nombre de neurones dans la couche cachées
- Le taux d'apprentissage (« Learning rate »)
- La variation du taux d'apprentissage au cours des itérations

Malgré de nombreux tests, nos réseaux de neurones entraînés stagnent toujours entre 70% et 80% de réussite. Comme le montre le graphique ci-dessous :



Le problème que nous avons rencontré ensuite, c'est que les pourcentages de réussite obtenues ne sont pas représentatifs des réels tests sur des photos. Nous sommes arrivées à une moyenne d'environ 15% de réussites sur les tests réels. Nous avons donc choisi de revoir notre façon d'entraîner notre réseau de neurones.

### 3) Deuxième jeu d'images

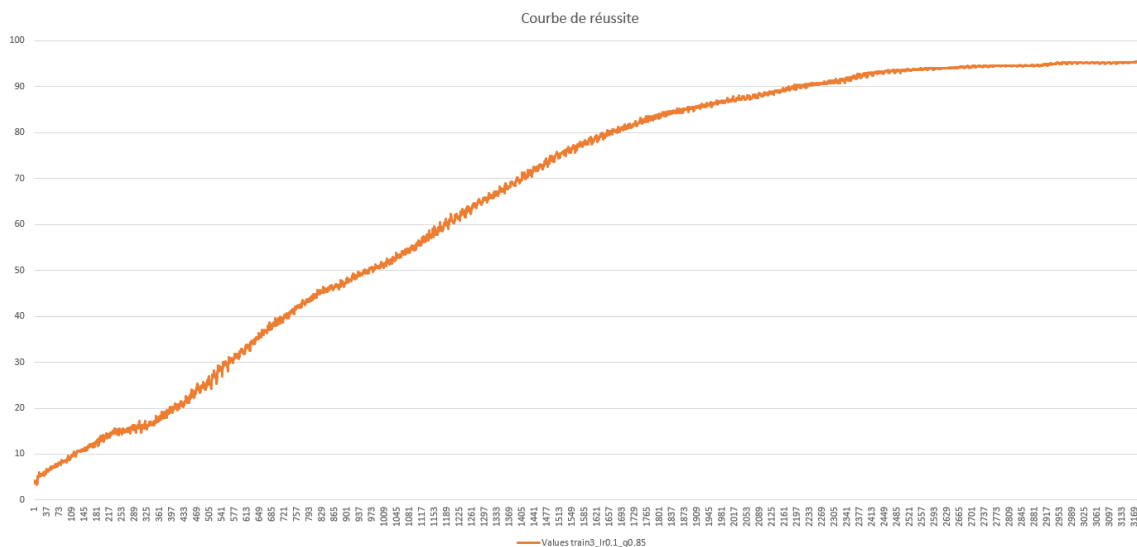
Nous nous sommes dit que le problème venait sûrement de la base de données « EMNIST », c'est pour cela que nous avons décidé de créer notre propre base de données. Tout d'abord, nous avons abandonné les lettres manuscrites. Et nous avons rempli notre base de données par 5038 lettres avec 97 polices d'écriture différentes.

Voici un extrait de celle-ci :

qrstuvwxyzABCDEFGHIJKLMNOP  
QRSTUVWXYZabcdefghijklmnopqrstuvwxyz  
qrstuvwxyzABCDEFGHIJKLMNO  
PQRSTUVWXYZabcdefghijklmnopqrstuvwxyz  
pqrstuvwxyzABCDEFGHIJKLMNOP  
QRSTUVWXYZabcdefghijklmnopqrstuvwxyzqr  
stuvwxyzABCDEFGHIJKLMNOPQRSTU  
VWXYZabcdefghijklmnopqrstuvwxyzA  
BCDEFGHIJKLMNOPQRSTUVWXYZ  
XYZabcdefghijklmnopqrstuvwxyz  
yzABCDEFGHIJKLMNOPQRSTUVWXYZ  
YZabcdefghijklmnopqrstuvwxyzA  
BCDEFGHIJKLMNOPQRSTUVWXYZ  
abcdefghijklmnopqrstuvwxyzABC  
DEFGHIJKLMNOPQRSTUVWXYZa  
bcdefghijklmnopqrstuvwxyz**ABCDE**

#### 4) Deuxième version de notre IA

En plus d'avoir remplacé la base de données « EMNIST » par la nôtre, nous avons traduit notre algorithme Javascript dans notre projet en C. Après plusieurs entraînements avec la nouvelle base de données, nous avons constaté de nettes améliorations. En effet, l'entraînement est beaucoup plus rapide car le nombre d'images à entraîner est bien inférieure au nombre de la base données précédente. Et le taux de réussite est bien plus élevé que la version précédente, comme le montre ce graphique :



On arrive très rapidement à un taux de réussite autour de 95%, mais encore une fois le taux de réussite réel obtenu reste encore bas. C'est pour cela que nous avons créé une troisième version du réseau de neurone en rajoutant une deuxième couche cachée. Les résultats obtenus après l'implémentation de la deuxième couche cachée, le taux de réussite réel a un peu augmenté mais cela reste minime.

Ainsi notre IA marche plus ou moins bien selon les différentes polices d'écriture.

#### c. Sauvegarde et chargement du réseau de neurones

Pour sauvegarder le réseau de neurones nous devons stocker les valeurs de tous les poids et de tous les biais dans un fichier binaire (\*.nn).

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'entraînement, nous récupérons ceux stockés dans un fichier.

## VI. Interface Graphique

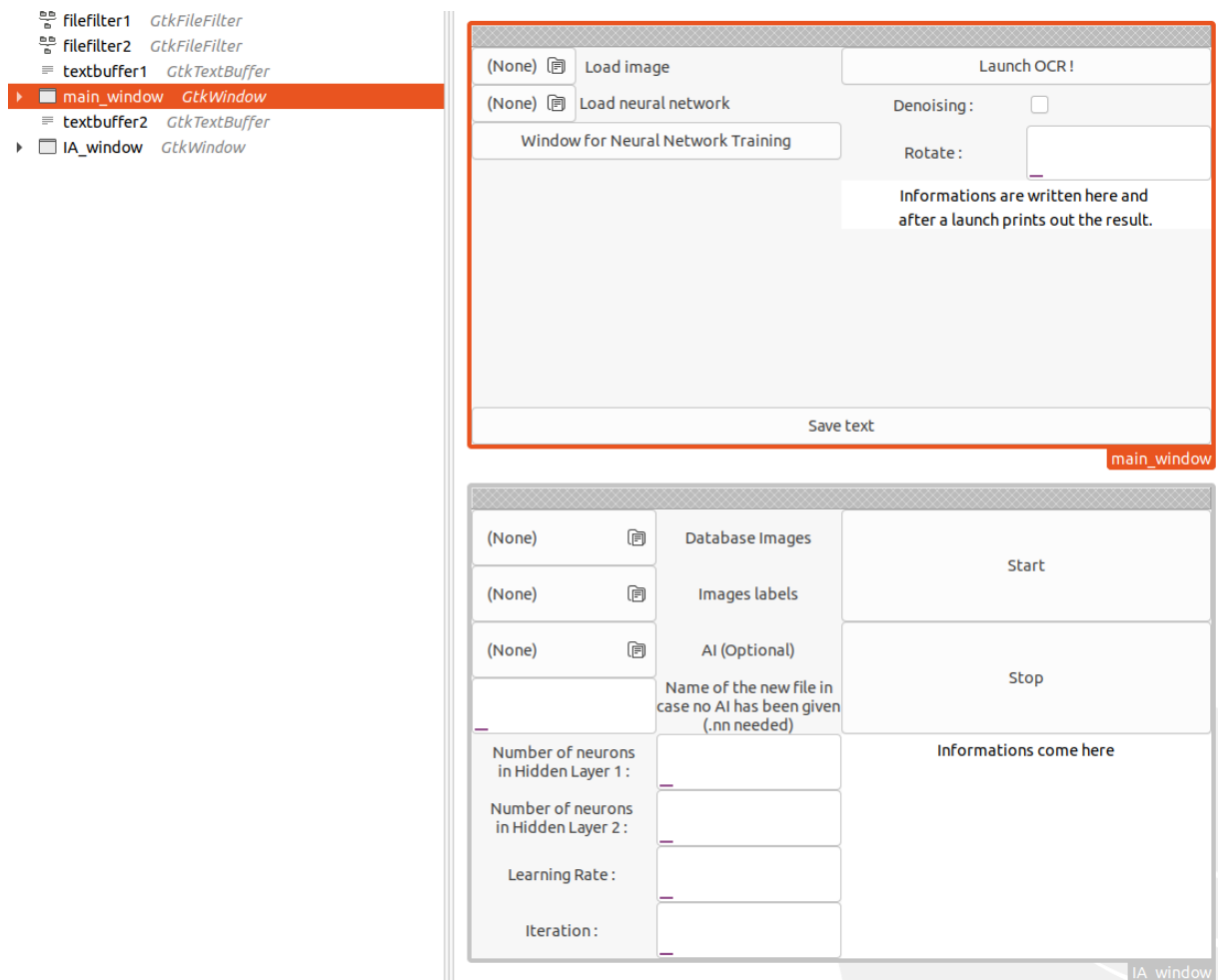
### a. Fenêtre

En ce qui concerne l'interface graphique nous avons décidé d'utiliser l'outil de conception d'interface graphique glade (version 3.20) se servant de la bibliothèque GTK et l'environnement de bureau GNOME.

Il prend en charge toute la partie de gestion/génération de l'interface pour permettre au développeur de se concentrer sur le code « utile ».

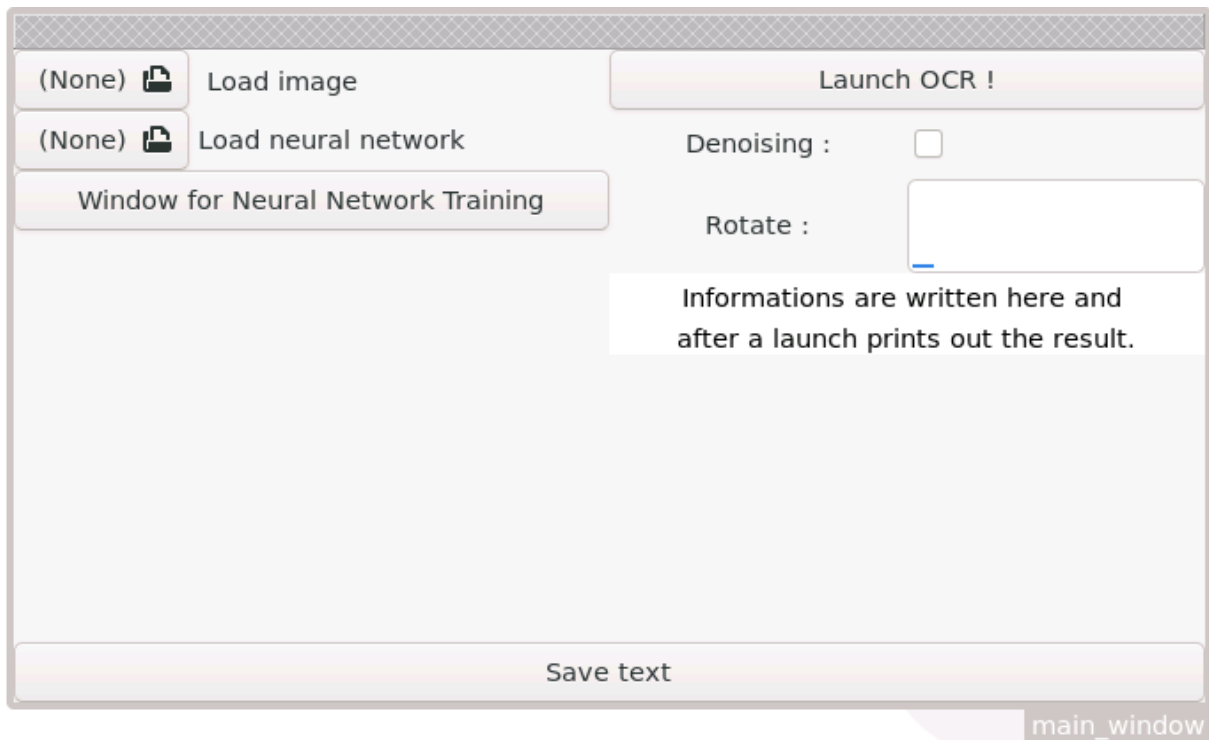
Ainsi glade nous semblait idéale pour créer une interface graphique rapidement selon nos besoins et nos programmes dans le langage C.

Voici notre fichier glade :



Nous avons décidé de créer deux interfaces différentes dans un seul fichier glade. La première qui s'ouvre dès le lancement de l'exécutable permet de choisir une image, choisir une IA, appliquer ou non une rotation, appliquer ou non un déparasitage et enfin lancer l'OCR.

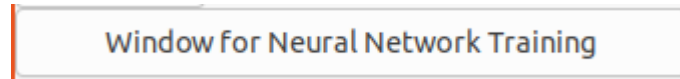
La deuxième permet de créer et d'entraîner une IA selon les différents paramètres rentrés par l'utilisateur. Un bouton Start permet de débiter l'entraînement, un bouton Stop de l'arrêter et lors de l'entraînement le taux de progression est affiché en pourcentage.



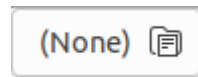
**Voici ci-dessus le GtkWidget qui représente notre fenêtre principale.**

Notre programme « Ulgtk.c » nous permet d'ouvrir la fenêtre principale grâce à une instance GtkBuilder et un fichier main.glade où se trouve la description de notre interface graphique.

L'interface de glade est principalement composée de boutons (GtkButton)



Mais aussi de boutons qui choisissent les fichiers (GtkFileChooserButton)



Et des boîtes pour rentrer des informations (GtkEntry). Des labels sont parfois positionnés avec ces boutons pour indiquer leur utilisation.

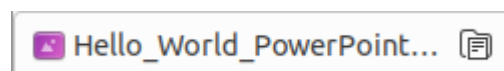


Pour que les boutons (GtkButton) puissent faire des actions dans notre programme nous utilisons des signaux. Chaque bouton dispose de son signal qui est associé à une fonction (nommé comme on le souhaite) et qui est appelé selon une condition précise, quant à l'interaction avec le bouton, le signal peut s'activer lorsque l'on clique sur un bouton ou que l'on appuie sur entrer en ayant sélectionné le bouton auparavant.

Dans l'exemple ci-dessous on peut voir le signal du bouton « Launch OCR ! » de l'interface graphique qui appelle la fonction « on\_launchOCR\_button\_clicked() » lorsqu'on clique sur le bouton.

General	Packing	Common	Signals
Signal	Detail	Handler	
▼ GtkButton			
activate		<Type here>	
<b>clicked</b>		on_launchOCR_button_clicked	


Tandis que les boutons choisissant les fichiers (GtkFileChooserButton) nous permettent de sélectionner les images, les bases de données et les réseaux de neurones nécessaires pour lancer l'OCR ou entraîner une IA.




Ces boutons sont liés à des « GtkFileFilter » qui permettent de filtrer les fichiers selon leur type ou selon leurs noms pour éviter les erreurs dans le programme.

Ci-dessous les « FileFilter » nommé filefilter1 et filefilter2 qui n'autorise que les images de type bitmap d'être sélectionnées ou bien les réseaux de neurones finissant par l'extension « .nn ».

---

	<b>filefilter1</b>	GtkFileFilter
	<b>filefilter2</b>	GtkFileFilter

Et ci-dessous leur configuration :

General	Packing	Common	Signals	
ID:	filefilter1			
Mime Types:	image/bmp <Type Here>			
Patterns:	<Type Here>			

ID:	filefilter2
Mime Types:	<Type Here>
Patterns:	*.nn <Type Here>



Enfin, les boutons à entrées permettent de demander à l'utilisateur des données pour les fonctions associées à leur utilité (voir label du bouton). Ces boutons ne prennent en compte la donnée rentrée que lorsque l'utilisateur appuie sur une touche Entrée du clavier, le signal associé sera alors activé.

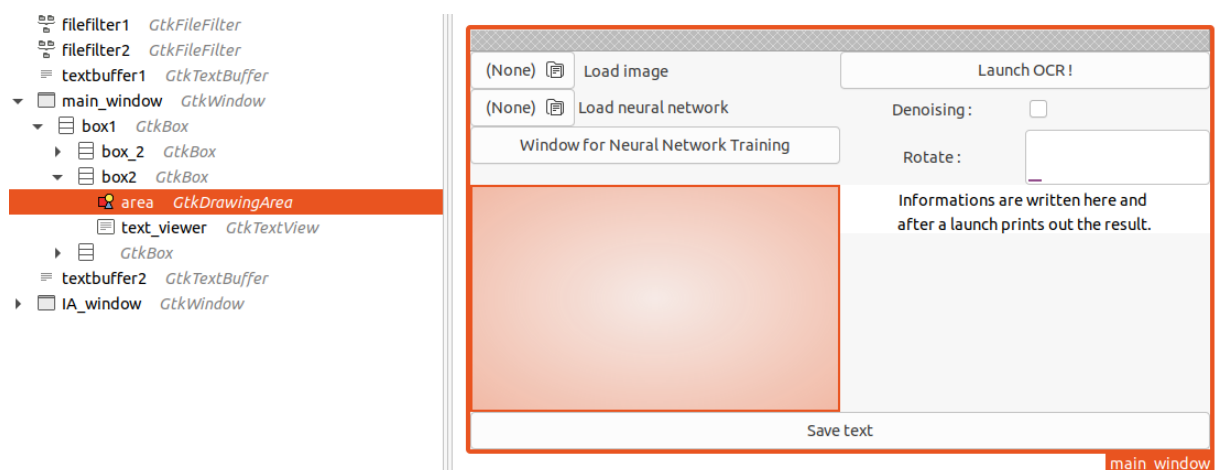
Voir photo ci-dessous :



Sur la première fenêtre une image peut être affichée grâce à un « GtkDrawingArea » qui permet l'affichage de l'image lorsque celle-ci est chargée dans le bouton associé.

Si une rotation est rentrée ou le déparasitage est coché l'affichage de l'image s'adaptera en fonction de l'image de base c'est-à-dire que si nous effectuons plusieurs fois une rotation à chaque changement l'image de base sera repris comme source.

Voici la position de l'image en orange :

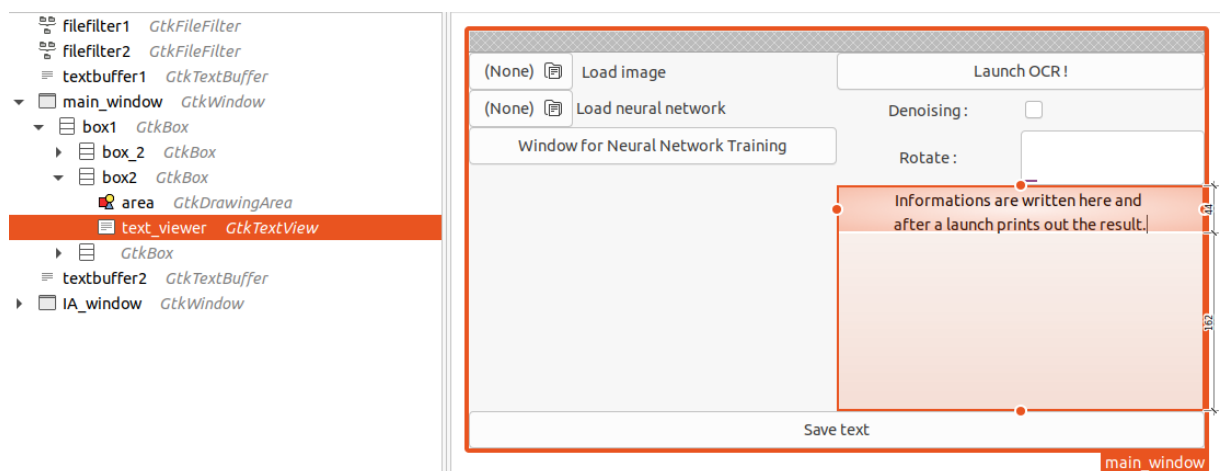


L'implémentation de l'image est quelque peu différente des autres car nous créons le signal de la fonction dans le script de la fonction principale « UI() » pour pouvoir y insérer d'autres arguments nécessaire à l'affiche de l'image et au redimensionnement de celle-ci. Voir les images ci-dessous :

```
area = GTK_DRAWING_AREA(gtk_builder_get_object(builder, "area"));
```

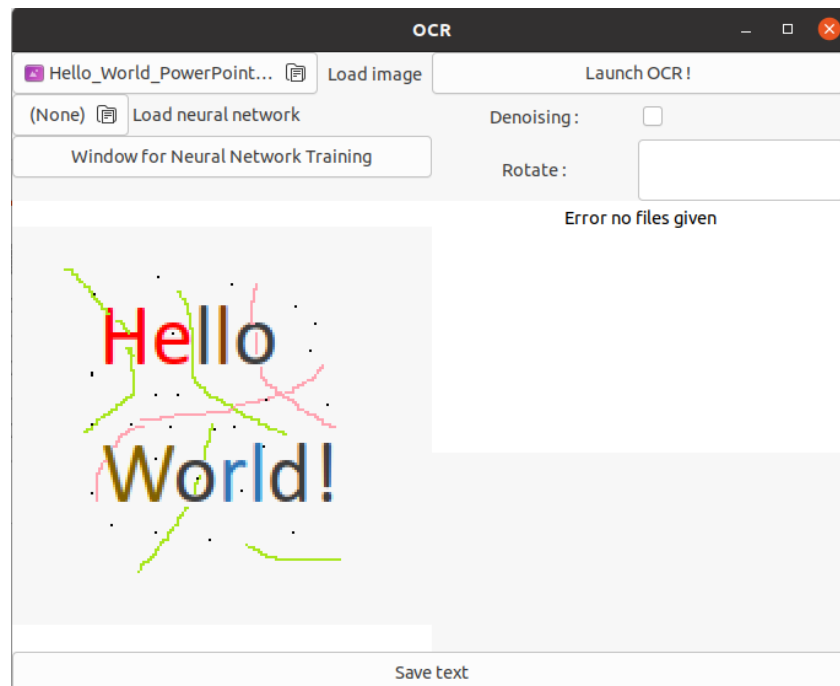
```
gboolean on_draw_image(GtkWidget *widget, cairo_t *cr, gpointer user_data)
```

L'utilisateur a aussi besoin que le programme renvoie des informations pour savoir ce qui est en train de se passer et donc pour cela nous affichons les informations dans un « GtkTextViewer » associé à un « GtkTextBuffer ». Voir représentation ci-dessous :

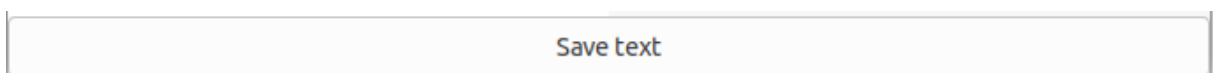


Le « GtkTextBuffer » nous permet de pouvoir écrire dans le « GtkTextViewer » et donc nous affichons des informations lorsqu'un procédé est fini ou lorsqu'il manque des informations pour lancer un programme.

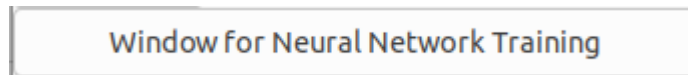
Voici ce que l'interface affiche lorsque nous avons lancé l'OCR sans réseau de neurones :



Lorsque l'OCR a été appliqué le texte traduit se trouvera écrit sur l'interface. A ce moment-là l'utilisateur peut décider de sauvegarder le texte en appuyant sur le bouton nommé « Save Text » qui le laissera choisir le dossier où sauvegarder le texte et le nom du fichier texte :



Maintenant parlons de notre deuxième fenêtre incluse dans notre interface graphique. Cette fenêtre est appelée en appuyant sur le bouton « Window for Neural Network Training » voir l'image ci-dessous :



Ce bouton active un signal qui crée la deuxième nommé « IA\_window » de la même façon que la fenêtre principale.

Comme nous l'avons dit précédemment cette fenêtre permet la création et l'entraînement d'une IA.

Voici l'apparence de la deuxième fenêtre :

(None)	Database Images	Start
(None)	Images labels	
(None)	AI (Optional)	
	Name of the new file in case no AI has been given (.nn needed)	Stop
Number of neurons in Hidden Layer 1 :		Informations come here
Number of neurons in Hidden Layer 2 :		
Learning Rate :		
Iteration :		

IA\_window

Nous pouvons créer une nouvelle IA pour pouvoir l'entraîner et donc il faudra rentrer un nouveau nom pour cette IA, ou nous pouvons appeler une IA déjà existante pour l'entraîner un peu plus.

Les chemins des fichiers choisis et les données rentrés dans les cases sont stockés dans des « gchar\* » et les chemins sont transformés en « (char\*) ». Tandis que les données sont transformées en « int » ou « double » grâce à la fonction « sscanf() » qui transforme des chaînes de caractères en nombre si possible.



Toutes les erreurs ou contresens ont été pris en compte par des valeurs dans le programme qui permettent ou non l'appelle d'une fonction. Par exemple il n'est pas possible de lancer l'entraînement deux fois d'affilés en appuyant sur le bouton « Start » sans avoir appuyer entre temps sur le bouton « Stop » pour que ça puisse débloquer le bouton « Start ».

Pour l'entraînement de l'IA des paramètres sont déjà fournies par défaut :

```
GRAPH_TRAIN_INFO train_info={NULL,NULL,NULL,0,0.1,15,15,&stop,&progression,&running};
```

Cependant il est conseillé de rentrer ses propres valeurs pour mieux comprendre et interpréter le comportement de l'IA.

Une base de données d'image et de labels seront absolument nécessaire pour l'entraînement de l'IA :

(None)		Database Images
(None)		Images labels

Nous pouvons fermer la deuxième fenêtre en cliquant sur la croix en haut à gauche de la fenêtre, cela stoppera tout processus qui ont pu avoir été rentré dans la deuxième fenêtre.

Finalement la fermeture de la fenêtre principale arrêtera le programme.

## **b. Le multithreading**

Dans un premier temps, lors de nos tests nous lançons l'entraînement du réseau dans le processus principal. Cela pose un gros problème car le logiciel est complètement bloqué tant que l'entraînement n'est pas terminé. Il était donc obligatoire de lancer, ce processus dans un autre thread afin de ne pas bloquer l'interface. Nous avons utilisé la bibliothèque pthread qui permet de créer facilement un nouveau thread. Celui se lance lorsque nous cliquons sur le bouton start. Pour nous permettre d'avoir un contrôle sur le thread, nous passons dans les arguments deux pointeurs vers des entiers. Ces entiers nommés « running » et « stop ». Le premier, « running » est modifié par le thread pour indiquer s'il est toujours actif. Nous pouvons modifier le deuxième pour indiquer qu'il faut arrêter l'entraînement plus tôt que prévu au thread. Si stop est à un, l'entraînement va s'arrêter et enregistrer le réseau de neurone pour ne pas perdre l'avancement.

Cette méthode pose un questionnement sur l'accès des variables par les différents threads, nous ne savons pas si les accès mémoires sont gérés nativement pour ne pas pouvoir avoir deux requêtes simultanées ou si au contraire le programme risque de planter. Nous penchons vers la première option car lors de tous nos tests nous n'avons aucun problème d'accès mémoire, néanmoins nous n'en sommes pas sûrs. Nous n'avons pas eu le temps d'assez nous renseigner pour confirmer cette hypothèse.

## VII. Compilation

Pour tester notre projet, nous avons besoin de le compiler, pour cela nous utilisons GCC. Nous ajoutons plusieurs options `-Wall`, `-Wextra` et `-std=c99` qui sont obligatoire. Nous ajoutons également l'option `-lm` pour permettre des calculs mathématiques plus poussé dans certaines fonctions. Enfin nous ajoutons des options nécessaires pour la compilation de la bibliothèque GTK qui gère la partie graphique de l'application, celles-ci sont `-pipe`, `-pthread`, ``pkg-config --cflags --libs gtk+-3.0'` et `-export-dynamic`.

Le fichier makefile se divise en quatre fonctions `all`, `main`, `copy` et `clean`.

- La fonction `clean()` permet de supprimer tous les fichiers de compilation inutiles. Elle supprime tous les fichiers «`.o`», «`.out`» et «`.exe`» présent dans les dossier et sous-dossier du répertoire courant. Néanmoins, elle ne supprime pas l'exécutable dans le dossier courant.
- La fonction `all()` appelle simplement la fonction `main`, en effet nous voulions avoir la possibilité de réaliser plusieurs choses pour nos tests dans la fonction avant ou après la compilation.
- La fonction `copy()` permet de copier le fichier makefile dans tous les dossier et sous-dossier du dossier courant. Cette fonction nous permet d'intégrer une modification à tous les autres makefile très rapidement.
- Enfin, la fonction `main()` s'occupe de la compilation. Elle suit des règles un peu particulières. Nous voulions pouvoir tester notre programme en plusieurs indépendamment (la pré-traitement, l'IA, le découpage des caractères) sans avoir à changer de dossier, ni recompiler tout le projet. Nous voulions donc pouvoir créer des fichiers `main.c` contenant une fonction `main()` n'étant pas prit en compte si nous recompilions tout le projet. C'est pour cela qu'un fichier makefile est présent dans chaque dossier. En effet, en tout point du projet nous pouvons compiler tous les fichiers «`.c`» présent dans le dossier courant et tous ses sous-dossiers. Néanmoins, nous ne prenons pas en compte les fichiers `main.c` présent ailleurs que dans le dossier courant pour éviter une multiple définition de la fonction `main()`.

```

pre-processing
├── contrast.c
├── contrast.h
├── main.c
├── makefile
├── negative.c
├── negative.h
├── Removing_Colors
│   ├── makefile
│   ├── rmcolors.c
│   └── rmcolors.h
├── Rotate
│   ├── makefile
│   ├── rotate.c
│   └── rotate.h
└── test.bmp

```

Ici nous pouvons tester par exemple les fonctions de la partie pré-traitement. La compilation prendra automatiquement en compte les tous les fichiers « .c ».

```

AUTHORS
characters_detection
├── characters_detection.c
├── characters_detection.h
├── main.c
├── makefile
├── ia_recognition
│   ├── database.h
│   ├── ia.h
│   ├── main.c
│   ├── makefile
│   ├── matrix_math.h
│   └── train3.nn
├── Images
│   ├── Characters
│   ├── Lines
│   ├── origin.bmp
│   └── test.bmp
├── main.c
├── main.h
├── makefile
├── other
│   ├── Bitmap
│   │   ├── bitmap.c
│   │   ├── bitmap.h
│   │   └── makefile
│   └── makefile
├── pre-processing
│   ├── contrast.c
│   ├── contrast.h
│   ├── main.c
│   ├── makefile
│   ├── negative.c
│   ├── negative.h
│   ├── Removing_Colors
│   │   ├── makefile
│   │   ├── rmcolors.c
│   │   └── rmcolors.h
│   ├── Rotate
│   │   ├── makefile
│   │   ├── rotate.c
│   │   └── rotate.h
│   └── test.bmp
├── UI
│   ├── glade
│   │   ├── main.glade
│   │   └── main.glade~
│   ├── makefile
│   └── src
│       ├── UIgtk.c
│       └── UIgtk.h

```

Dans ce dossier-là, nous allons compiler tout le projet mais nous ne prenons ne compte que le main.c présent dans le dossier courant



## VIII. Conclusion

Nous avons réussi à finir le projet, certains détails pourraient être améliorés mais nous sommes globalement assez fiers de notre travail. L'interface graphique est pleinement utilisable et, même si elles ne sont pas parfaites, nous arrivons à réaliser des IA reconnaissant à peu près la moitié des caractères. Nos principales difficultés ont été d'implémenter l'IA XOR et l'IA de reconnaissance de caractères. Nous avons eu également des difficultés pour entraîner les réseaux de neurones, en effet, nous ne savions et ne savons toujours pas vraiment comment les paramètres influent sur les résultats (mettre plusieurs couches de neurones ou augmenter le nombre de neurones sur une couche par exemple).

Le programme pourrait possiblement être amélioré au travers de plusieurs points. Le principal est l'entraînement de l'IA, en effet le processus est actuellement sur un seul thread mais pourrait surement utiliser les avantages du multithreading pour grandement améliorer ses performances. L'interface graphique pourrait également être améliorée pour la rendre plus fluide et plus ergonomique. Enfin, le dernier point d'amélioration serait l'optimisation de différentes fonctions.

Pour finir, notre groupe a aimé travailler sur ce projet, il nous a permis de découvrir un monde qui nous était encore totalement inconnu, l'IA, dans sa création et son implémentation.