Projet OCR
Soutenance 1

Abadie Hugo
Jomier Magali
Machavoine Ethan
Poelger Jonathan

Novembre 2020
Encadrant : Rémi Vernay

Table des matières

1 Introduction

Cette deuxième année de cycle préparatoire à l'EPITA a démarré en force avec l'arrivée du second projet à réaliser dans notre cursus d'ingénieur en informatique. Ce projet, un logiciel de reconnaissance optique de caractère (OCR), est développé par une équipe prête à tout donner pour réussir.

Cette équipe est composée de 4 étudiants motivés : Abadie Hugo, Jomier Magali, Machavoine Ethan et Poelger Jonathan.

Pour arriver à la réalisation d'un OCR fiable et fonctionnel, plusieurs parties devront être traitées. Tout d'abord un chargement et un traitement de l'image devra être réalisé, transformant une image quelconque en une image en noir et blanc. Suite à cela, il faudra reconnaître et découper les zones de textes présentes dans cette image. On découpera de même chaque caractère de chaque zone de texte. Ces caractères seront ensuite envoyés à un réseau de neurones qui s'occupera de les reconnaître. Enfin, il nous faudra reconstituer le texte avec les caractères reconnus afin de le sauvegarder et de pouvoir l'utiliser.

Ce rapport présente le travail effectué depuis le début de la réalisation de ce projet jusqu'à maintenant et se découpe en plusieurs parties.

Une première partie se concentrera sur le découpage des tâches à réaliser entre les différents membres du groupe.

Une seconde partie présentera l'avancement général du projet en abordant notamment le chargement et le traitement de l'image, ainsi qu'un "proof of concept" de réseau de neurones pouvant apprendre la fonction XOR.

Enfin la dernière partie présentera le jeu d'images créé pour l'entraînement du réseau de neurones, la méthode utilisée pour sauvegarder nos données et les prochains objectifs à réaliser pour l'obtention d'un OCR fonctionnel.

2 Découpage du projet

2.1 Avancement des tâches

Ce tableau montre la répartition des tâches.

Répartition	Hugo	Magali	Ethan	Jonathan
Chargement de l'image		X		
Suppression des couleurs		X		
Pré-traitement		X		
Découpage des caractères	X	X		
Proof of concept			X	X
Jeu d'images	X			
Manipulation de fichiers			X	X

Ce tableau montre l'avancée des différentes tâches du projet.

Tâches	Avancement (en %)
Chargement de l'image	100
Suppression des couleurs	100
Pré-traitement	70
Découpage des caractères	80
IA	30
Jeu d'images	50
Manipulation de fichiers	50
Reconstruction de texte	0
Interface	0

3 Avancement

3.1 Chargement des images et suppression des couleurs

3.1.1 Chargement des images

- La bibliothèque SDL2

Afin de pouvoir travailler sur des images, nous avons choisi d'utiliser la librairie SDL2. Cette librairie contient les mêmes fonctionnalités que SDL mais offre la possibilité de travailler sur de nombreux formats d'images comme *JPEG* (.jpg) ou *PNG* (.png) et non uniquement sur des fichiers *Bitmap* (.bmp).

Avant toute utilisation de cette librairie, il est nécessaire de l'initialiser grâce à la commande *SDL_Init*. De même, à la fin des manipulations la bibliothèque doit être fermée avec *SDL_Quit*.

Nous avons ainsi accès à la fonction *IMG_Load* qui permet de charger une image et *SDL_SetVideoMode* qui permet d'afficher une image à l'écran. Ainsi nous pouvons travailler sur des images et les afficher manuellement sans utiliser une interface graphique comme *GTK*.

Cette librairie offre aussi la possibilité d'accéder et de modifier plusieurs paramètres de l'image (sa largeur et sa hauteur) ainsi que sur les pixels qui la compose (leur couleur en format *RGB* ou *RGBA*). Ces fonctions seront très utiles pour le reste du projet.

3.1.2 Suppression des couleurs

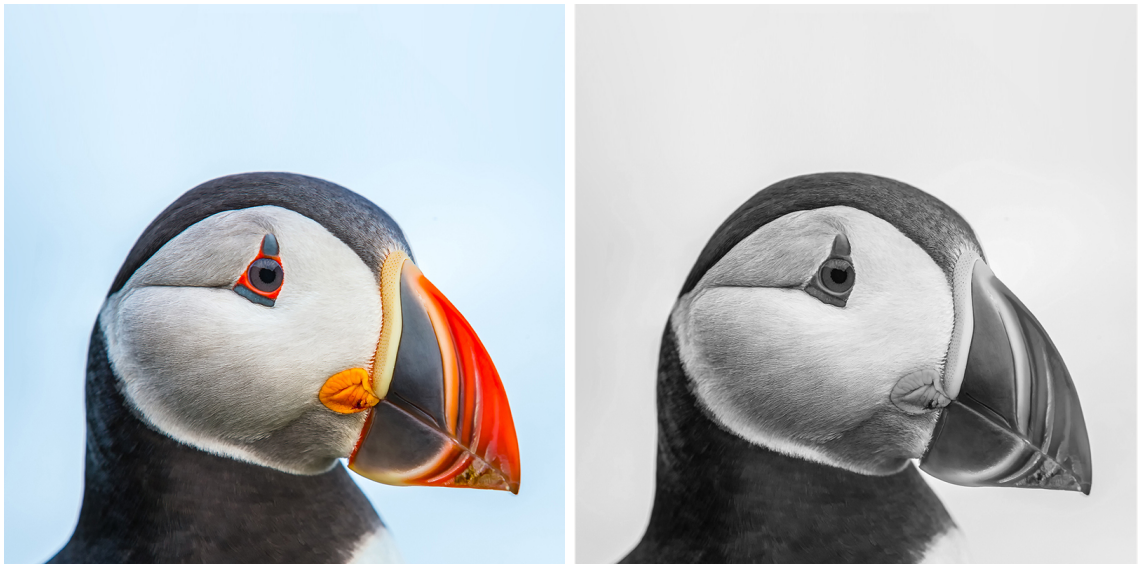
- *Greyscale*

Avant de commencer à traiter l'image, les couleurs doivent être supprimées pour permettre une meilleur analyse des pixels de celle-ci et permettre la binarisation.

Pour cela nous avons implémenté la fonction *greyscale*. Elle transforme tous les pixels de l'image en nuances de gris. Pour chaque pixel, nous calculons son niveau de gris grâce à sa valeur en format *RGB* dans lequel r correspond au rouge, g au vert et b au bleu. Ainsi le niveau de gris correspond à :

$$greyscale = 0.3 * r + 0.59 * g + 0.11 * b$$

Image de pingouin d'origine et image après suppression des couleurs



Les modifications suivantes sont opérées sur les images en nuances de gris donc avec des valeurs r, g et b identiques.

3.2 Pré-traitement

Lorsque l'utilisateur va donner une image au logiciel, elle peut ne pas être "parfaite" pour être utilisée dans le réseau de neurones. En effet, les lignes peuvent ne pas être horizontales, il peut y avoir une couleur de fond autre que blanc ou encore avoir du "bruit", c'est-à-dire des pixels dont la couleur tranche avec celle des pixels voisins et qui parasitent l'image. Différents traitements sont donc nécessaires pour corriger ces imperfections avant la suite des opérations.

Nous avons jusqu'ici implémenté une réduction des bruits et une binarisation de l'image.

3.2.1 Réduction des bruits

Comme dit précédemment, le but de la réduction des bruits est d'atténuer les imperfections de l'image donnée. Pour cela plusieurs méthodes existent.

Nous avons choisi d'implémenter le filtre médian. Ce filtre permet de diminuer les bruits tout en conservant les contours des objets présents sur une image ce qui est très important pour nous car les lettres ne doivent pas être modifiées. Nous conservons ainsi les caractères.

Le principe de l'algorithme est de remplacer chaque pixel par la valeur médiane des pixels de son voisinage dans un certain rayon. Plus le rayon est grand, plus le filtre floute l'image. Nous avons choisi un rayon de 1 pour réduire les bruits sans flouter les lettres correspondant à une matrice 3x3.

Avant application du filtre médian sur le pixel de valeur 50

5	6	7
6	**50**	8
7	8	9

Il faut tout d'abord trier ces valeurs par ordre croissant, afin d'avoir la valeur médiane à la case 4 (le milieu) du tableau.

Tri par ordre croissant des valeurs avoisinantes du pixel de valeur 50

5	6	6	7	**7**	8	8	9	50

On peut ainsi en déduire que la valeur médiane est de 7. Il ne reste plus qu'à remplacer la valeur d'origine par la valeur obtenue grâce au filtre médian.

Après application du filtre médian sur le pixel de valeur 50

5	6	7
6	**7**	8
7	8	9

Image de lions d'origine et image après réduction des bruits

3.2.2 Binarisation

La binarisation est une étape incontournable avant la détection des lignes et caractères. En effet, sans elle, l'image est remplie de différentes nuances de gris et les algorithmes seraient bien plus complexes sans cette binarisation. Nous avons choisi d'implémenter la méthode d'Otsu mais de nombreuses autres existes.

- Méthode classique

La méthode classique utilise un seuil fixe. Elle consiste à dire que tout pixel ayant une valeur de nuance de gris supérieure à 127 devient un pixel blanc et les autres des pixels noirs. Cette méthode insinue que le texte est de couleur suffisamment foncée et le fond de l'image suffisamment clair.

$$\text{Bin(greyscale)}= \begin{cases} 0 & \textit{si greyscale} > 127 \\ 255 & \textit{sinon} \end{cases}$$

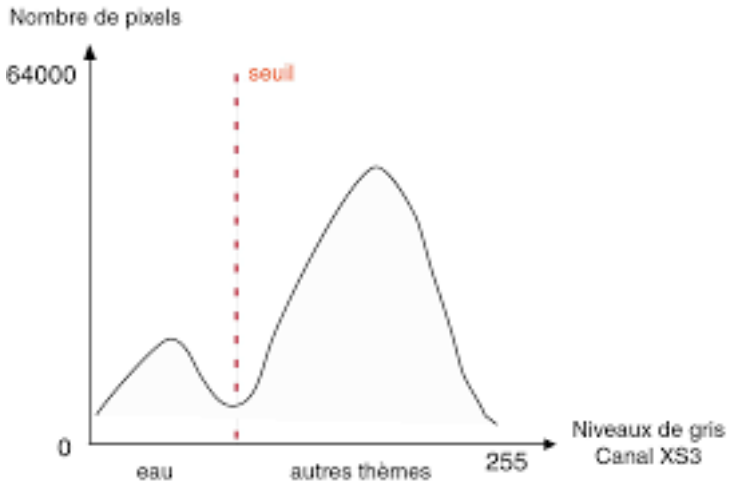
Image de test d'origine et image après binarisation

- Méthode d'Otsu

La méthode d'Otsu, au contraire, utilise un seuil automatique. Nous calculons le meilleur seuil de binarisation pour permettre de gérer les cas où le fond de l'image est sombre ou le texte clair. Cet algorithme se base sur le fait qu'une image dans la plupart des cas ne contient que deux plans, le premier plan et l'arrière plan. L'algorithme calcule le seuil optimal qui sépare les deux plans de l'image.

Les différents calculs de ce seuil se basent sur la probabilité qu'un pixel soit d'une certaine nuance de gris ou non. Pour cela, un histogramme des nuances de gris est nécessaire. Cela va nous permettre de recréer virtuellement un graphique en deux dimensions montrant à l'aide de "pics" le nombre de pixels qui ont un niveau de gris en commun. Les niveaux de gris sont classés par ordre croissant. A un niveau de gris particulier, on va pouvoir noter une très grande différence du nombre de pixels ayant une valeur de gris i et un niveau de gris i+1. Cette valeur est le seuil de binarisation optimal. Pour le déterminer d'autres calculs sont obligatoires.

Histogramme d'une image constituée d'eau et d'un décor



Il est ensuite nécessaire de normaliser cet histogramme pour obtenir la probabilité qu'un pixel soit d'une nuance donnée. Pour cela, nous divisons chaque valeur de l'histogramme par le nombre de pixels total de l'image. Nous obtenons ainsi un tableau de probabilités.

Ces probabilités sont ensuite divisées en 2 classes : la probabilité que la nuance de gris fasse partie du fond de l'image et la probabilité qu'elle fasse partie des objets de l'image (ici les caractères). La première classe s'obtient en faisant la somme de toutes les probabilités de cette classe :

$$\omega_{C1}(i) = \sum_{k=1}^{i} Proba(i)$$

avec :
i : la nuance de gris de valeur i sur laquelle on travaille
$\omega_{C1}(i)$: la probabilité que i appartienne à la classe $C1$
$Proba(i)$: la probabilité qu'un pixel soit de nuance i d'après le tableau de probabilités

De plus on remarque facilement que la deuxième classe découle de la première car la somme des probabilités vaut 1 :

$$\omega_{C2}(i) = 1 - \omega_{C1}(i)$$

avec :
i : la nuance de gris de valeur i sur laquelle on travaille
$\omega_{C1}(i)$: la probabilité que i appartienne à la classe $C1$
$\omega_{C2}(i)$: la probabilité que i appartienne à la classe $C2$

On peut ensuite calculer la moyenne de chaque classe qui correspond à la somme des probabilités multipliées par la valeur de la nuance de gris de cette probabilité :

$$Moy_{C1}(i) = \sum_{k=1}^{i}(k * Proba(k))$$

avec :
i : la nuance de gris de valeur i sur laquelle on travaille
$Moy_{C1}(i)$: la moyenne de la classe $C1$ de 0 à i
$Proba(i)$: la probabilité qu'un pixel soit de nuance i d'après le tableau de probabilités

La méthode d'Otsu va maintenant calculer la variance interclasse entre les classes $C1$ et $C2$ pour tous les i possibles (de 0 à 255). La variance interclasse caractérise la différence qui existe entre les pixels des deux classes. Plus elle est grande, moins les deux classes se ressemblent. La réciproque est également vraie.

Par conséquent, le seuil optimal est le i qui obtient la plus haute variance interclasse.

La variance interclasse entre les classes $C1$ et $C2$ correspond à :

$$Var_{C1interC2}(i) = \frac{(Moy * \omega_{C1}(i) - Moy_{C1}(i))^2}{\omega_{C1}(i) * \omega_{C2}(i))} = \frac{(Moy * \omega_{C1}(i) - Moy_{C1}(i))^2}{\omega_{C1}(i) * (1 - \omega_{C1}(i))}$$

avec :
i : la nuance de gris de valeur i sur laquelle on travaille
$\omega_{C1}(i)$: la probabilité que i appartienne à la classe $C1$
$\omega_{C2}(i)$: la probabilité que i appartienne à la classe $C2$
$Moy_{C1}(i)$: la moyenne de la classe $C1$ de 0 à i
Moy : la moyenne de l'image soit la moyenne de la classe $C1$ pour 255

Ainsi en calculant la variance interclasse entre les classes $C1$ et $C2$ pour tous les i de à 255, on peut trouver pour quelle valeur de i cette variance est maximale. Il ne reste plus qu'à binariser l'image de la même façon qu'avec la méthode classique mais avec un seuil de binarisation valant i et non plus 127.

$$\text{Bin(greyscale)}= \begin{cases} 0 & si\ greyscale > \text{i} \\ 255 & sinon \end{cases}$$

Image de test d'origine et image après binarisation avec la méthode d'Otsu

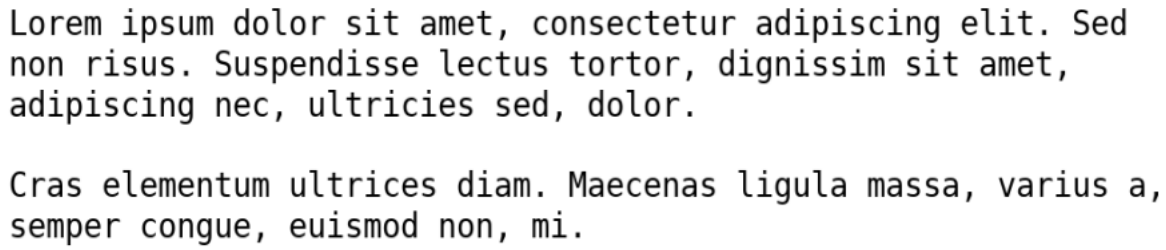
3.3 Détection et découpage en blocs, lignes et caractères

Les images qui vont être traitées lors de cette partie l'ont déjà été lors du prétraitement. Autrement dit les images traitées seront composées uniquement de pixels blancs ou noirs. Avant le traitement toutes les valeurs de couleurs seront donc identiques pour chaque pixel. La segmentation est une partie très importante lors de la réalisation de l'OCR. En effet, si le réseau de neurones fonctionne, mais que les images d'entrée ne sont pas bonnes, il sera alors difficile ou impossible d'arriver à un résultat correct.

Nous considérerons pour le découpage que les caractères sont des rectangles de pixels. Ainsi nous allons découper en lignes de caractères puis en colonnes entre caractères. Nous avons utilisé deux couleurs pour ce découpage, le rouge pour les interlignes et en vert pour les espaces entre caractères. Nous pourrons ainsi reconnaître les changements de paragraphe et les espaces facilement.

Pour la suite de cette partie, nous travaillerons sur l'image suivante qui évoluera au fur et à mesure des parties.

Image de texte d'origine



3.3.1 Découpage en lignes

Le découpage en ligne est la première étape pour extraire des caractères. Avant de commencer, il est nécessaire de définir ce qu'est une ligne. Une ligne est "une série de choses disposées selon une direction donnée" (d'après le Dictionnaire Hachette). Ainsi pour nous, une ligne est un ensemble de caractères alignés séparés par une bande blanche grâce à la binarisation.

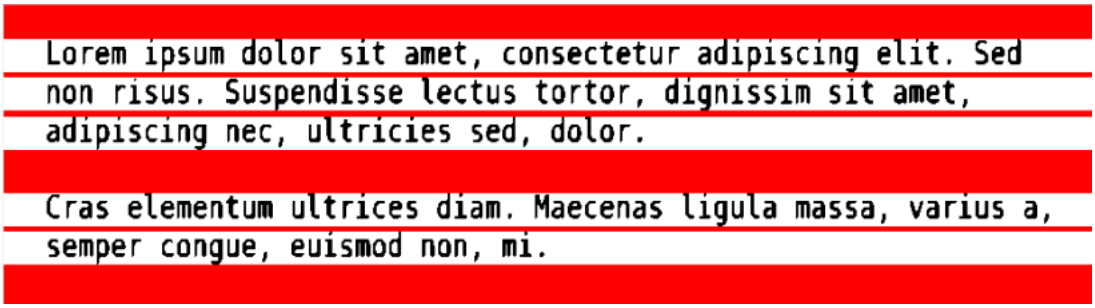
Pour le découpage en ligne, nous avons donc choisi de chercher les lignes composées uniquement de pixels blancs qui correspondent aux lignes entre les caractères.

On parcourt la ligne de pixels actuelle : si elle est entièrement composée de pixels blancs alors on remplace l'entièreté de ces pixels par des pixels rouges, soit de valeur *(r, g, b) = (255, 0, 0)*. Si un pixel noir est trouvé, on passe à la ligne suivante car

la ligne actuelle correspond à une ligne de caractères.

On applique ce procédé de recherche sur toutes les lignes de l'image ce qui permet de supprimer les marges hautes et basses d'un texte en même temps que les espaces entre les lignes.

Image de texte après découpage des lignes



3.3.2 Découpage en caractères

Le découpage en caractères consiste à isoler chaque caractère d'une ligne des autres. Un caractère est défini par un ensemble variable de pixels noirs très rapprochés entourés de pixels blancs. Ainsi chaque caractère est séparé par une colonne blanche grâce à la binarisation.

Pour le découpage en caractères, nous avons donc choisi de chercher les colonnes composées uniquement de pixels blancs qui correspondent aux espaces entre les caractères. Cette recherche doit être appliquée sur chaque ligne découpée précédemment.

Avant de chercher les espaces, il est nécessaire de déterminer les lignes de pixels sur lesquelles il faut tester. Pour cela, nous regardons le premier pixel d'une ligne :

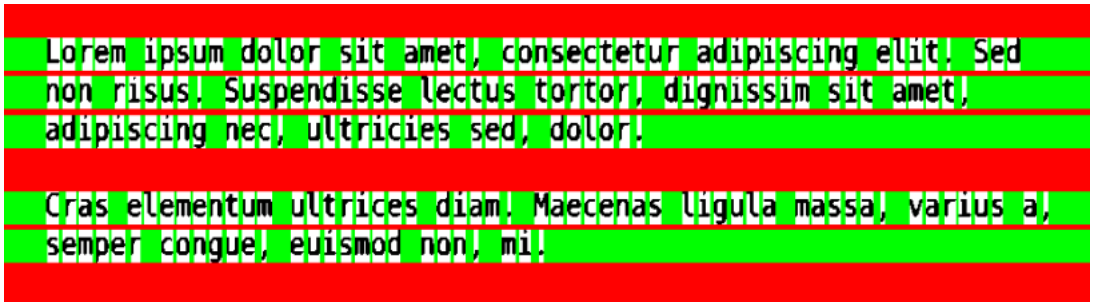
Si, il est rouge, la ligne est un espace et ne contient pas de caractère, on passe alors à la ligne suivante.

Sinon, cela veut dire que cette ligne de pixel appartient à une ligne de caractères. Dans ce cas, on note *begin* l'index de cette ligne et on commence à chercher la fin de la ligne de caractères. On cherche alors la première ligne suivante pour laquelle le premier pixel est rouge. Une fois trouvé, on note *end* l'index précédent.

Maintenant que nous connaissons les lignes de pixels sur lesquelles il faut travailler, on peut appliquer un algorithme semblable à celui des lignes entre *begin* et *end*. Pour reconnaître les espaces entre les caractères, on regarde ensuite pour toute la longueur de la ligne si la colonne entre *begin* et *end* est entièrement composée

de pixels blancs. Cela signifie qu'aucun caractère n'est contenu dessus, on remplace l'entièreté de ces pixels par des pixels verts, soit de valeur *(r, g, b) = (0, 255, 0)*. Si un pixel noir est trouvé, on passe à la colonne suivante car la colonne actuelle correspond à un caractère.

Image de texte après découpage des caractères



3.3.3 Mise en page

Les fonctions précédentes découpent le texte en lignes et en caractères sans distinction pour les paragraphes et les espaces entre les mots. Il est donc nécessaire d'implémenter une fonction permettant de les détecter. Celle-ci n'a pas encore été écrite, cependant voici son fonctionnement.

Les paragraphes sont une "subdivision d'un texte en prose, typographiquement définie" (d'après le Dictionnaire Hachette). Ainsi un paragraphe a une typographie globalement constante qui peut se traduire par un espace plus important entre deux lignes de texte. Pour déterminer l'écart moyen entre deux lignes, nous pouvons parcourir le premier pixel de toutes les lignes (en excluant la marge haute et la marge basse qui pourraient fausser les calculs) de l'image. En divisant le nombre de pixels rouges obtenu par le nombre d'interlignes, nous obtenons une valeur moyenne de l'espacement. Si l'espacement est supérieur à cette valeur alors il s'agit d'un nouveau paragraphe sinon d'un simple interligne.

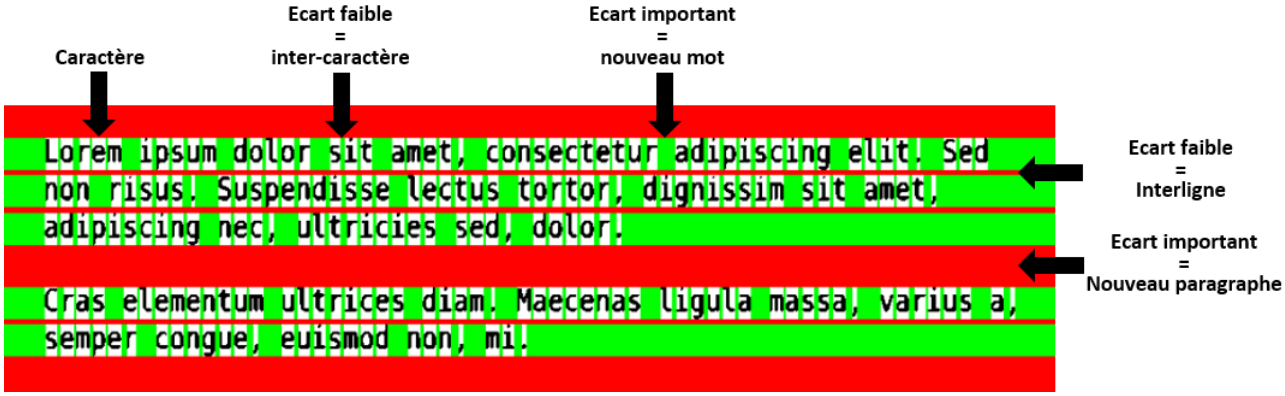
La différence entre un espace inter-caractères et un espace entre deux mots est la largeur de cet espace. Un espace entre deux mots est généralement minimum trois fois plus gros qu'un espace entre caractères. Nous souhaitons utiliser un algorithme semblable à celui de la détection des paragraphes. En calculant la moyenne des espaces d'un texte, nous pouvons déterminer à quel type d'espace nous avons à faire pour chaque cas.

Une autre façon de déterminer à partir de quelle taille un écartement peut être considéré comme un nouveau paragraphe ou un nouveau mot pourrait être d'utiliser un histogramme. A partir de celui-ci nous pourrions déterminer le seuil à partir duquel ces écartements signifient un nouveau paragraphe ou un nouveau mot.

Après les découpes du texte, nous parcourons l'image dans le sens de lecture (gauche à droite et haut en bas). Dans le parcours hauteur, dès qu'un pixel autre que rouge est trouvé cela signifie qu'une ligne de caractères a été détectée. Le parcours largeur est lancé pour trouver un pixel non vert qui symbolise la présence d'un caractère.

Le caractère trouvé est redimensionné et stocké dans une matrice de 0 et de 1. Les pixels noirs correspondent à des 1 et les blancs à des 0. La fonction appelle ensuite le réseau de neurones pour déterminer de quel caractère il s'agit. Enfin le caractère est stocké dans une chaîne de caractères. L'algorithme continue ensuite sur le reste de la ligne puis sur le reste de l'image.

Détection des différents éléments sur une image



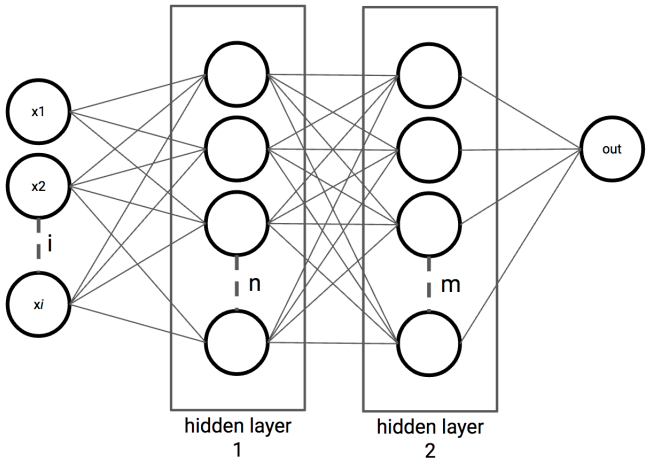
3.4 Proof of concept : la fonction XOR

3.4.1 Définition et fonctionnement

- Qu'est ce qu'un réseau de neurones ?

En informatique, un neurone est simplement une case mémoire qui contient un chiffre, très souvent entre 0 et 1, qui est sa valeur d'activation. Lorsqu'on agence des neurones en couches et que nous les relions entre elles, nous obtenons un réseau de neurones. On appellera la première couche (celle où l'information à traiter est entrée) *input layer*, la dernière (celle qui contiendra la réponse du réseau) *output layer*, et toutes les autres entre ces deux-là seront nommées *hidden layer*. Pour les relier entre elles, il faut mettre en place une série de poids qui déterminera l'influence de chaque neurone sur chaque neurone de la couche suivante. En plus de cela, il faut mettre en place pour chaque neurone un biais, qui est un indicateur de sa tendance à être activé (proche de 1) ou éteint (proche de 0).

Exemple de représentation d'un réseau de neurones avec ses poids



- Comment fonctionne un réseau de neurones ?

Pour le calcul de la valeur d'activation d'un neurone d'une couche, il faut effectuer la somme pondérée des activations de la couche précédente multipliée par les poids reliant chaque neurone de la couche précédente à celui que l'on souhaite calculer. Reste ensuite à ajouter le biais du neurone au résultat. Il est possible et pratiquement obligatoire dans les cas complexes d'utiliser une fonction d'activation qui forcera le résultat des calculs à rester dans l'intervalle d'activation des neurones. Les trois possibilités pour ce travail, qui sont les plus utilisés dans les réseaux de neurones sont :

— La fonction *Sigmoïd* : $f(x) = \frac{1}{1 - e^x}$

— La fonction *ReLu* : $f(x) = max(0, x)$

— La fonction *Softmax* : $\delta(z)_j = \frac{e_j^z}{\sum_{k=1}^{K} e_k^z}$ pour tout j de {1,...,K}.

Au final, on peut condenser le passage d'une couche i-1 à la suivante ainsi :
$a_i = f(W * a_{i-1} + B)$
a_i : vecteur contenant la couche i.
a_{i-1} : vecteur contenant la couche i-1
W : matrice des poids de dimension taille de a_i par taille de a_{i-1}
B : vecteur contenant les biais de la couche a_i
f : fonction d'activation appliquée à chaque élément de a_i

Il faut ensuite répéter l'opération de la première à la dernière couche.

- Comment entraîner un réseau de neurones ?

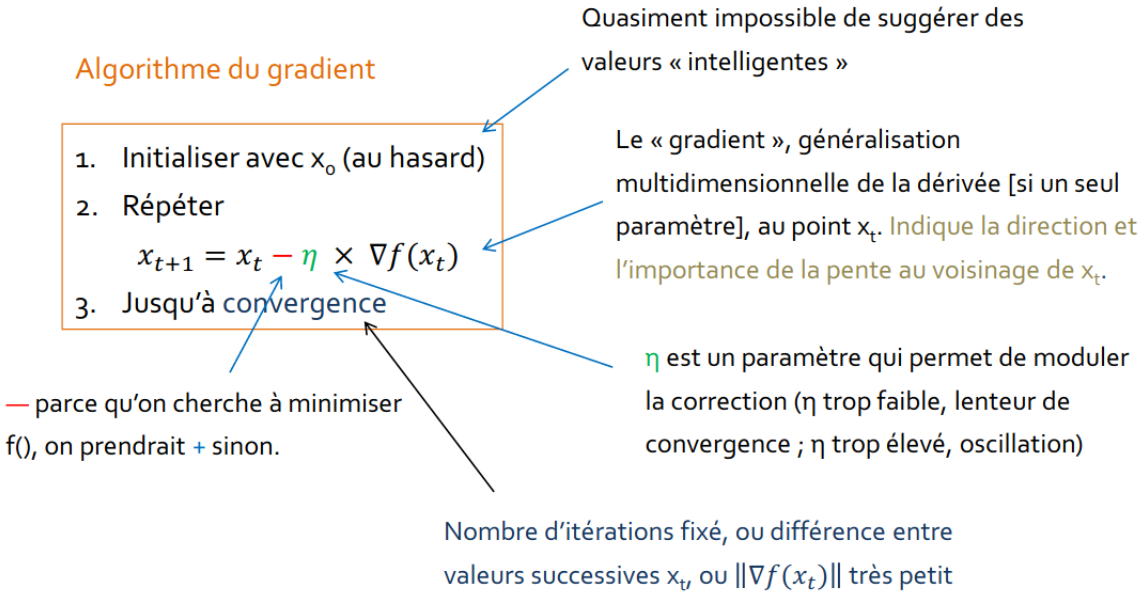
L'entraînement du réseau consistera à calibrer tous les poids et biais jusqu'à ce que la réponse soit au plus proche de ce que l'on souhaite.

Pour commencer, il faut savoir à quel point le réseau est loin de ce que l'on attend de lui. Pour cela on utilise une fonction, appelée fonction de coût. Cette fonction prend en paramètre un vecteur contenant les sorties du réseau et un autre vecteur contenant les réponses attendues pour chaque neurone de la dernière couche pour une entrée donnée. La fonction fera la somme des différences au carré entre ce que l'on a et ce que l'on veut pour chaque neurone de la couche de sortie, donc la somme des distances au carré entre notre modèle et celui vers lequel on voudrait tendre.

Le but de l'entraînement sera de minimiser cette fonction de coût, donc d'arriver à un réseau qui pour chaque entrée donne des réponses très proches de ce que l'on souhaite.

En mathématiques, pour minimiser une fonction, une méthode efficace est la descente de gradient :

Algorithme du gradient (Source : univ-lyon2)



Pour entraîner un réseau de neurones il faudra réaliser l'algorithme du gradient pour chaque poids et chaque biais, jusqu'à minimisation de la fonction de coût.

3.4.2 Création du réseau

Pour le XOR, nous avons opté pour une structure en 3 couches :

— La couche d'entrée est composée de deux neurones (les deux bits dont on doit faire le XOR)

— La couche cachée a 4 neurones, avec l'espérance de base qu'elle détecte les pattern 00, 01, 10 et 11.

— La couche de sortie possède aussi deux neurones, représentant à quel point le réseau "pense" que la réponse est respectivement 1 et 0.

De plus, nous avons décidé d'utiliser une représentation matricielle pour ce réseau de neurones. Chaque couche est donc représentée par une matrice contenant les valeurs d'activation des différents neurones de cette couche. Pour les poids et les biais, nous avons décidé de les représenter sous forme de matrices. Comme notre réseau est composé de 3 couches, notre implémentation est donc composée de 3 tableaux pour les valeurs d'activation des neurones des 3 couches, 2 tableaux de poids et 2 tableaux de biais. Pour chaque couple de tableaux de poids et de biais, le premier tableau relie la première couche à la deuxième et le deuxième tableau relie la deuxième couche à la troisième couche.

Ainsi pour une couche n, le poids ${w^n}_{ij}$ relie le i^{ieme} neurone de la n^{ieme} couche au j^{ieme} neurone de la $n+1^{ieme}$ couche, et de même pour le biais $b^n_{i\,j}$.

Il a ensuite fallu implémenter cette représentation. Pour cela chaque matrice de la représentation est implémentée sous la forme d'un tableau. Les tableaux de poids et de biais sont initialisés avec des nombres flottants aléatoires entre -1 et 1. Il nous suffit ensuite d'activer le réseau de neurones, ce qui sera détaillé dans la prochaine partie.

3.4.3 Exécution du réseau

Afin de pouvoir exécuter notre réseau de neurones, nous avons d'abord écrit les fonctions d'addition et de multiplication de matrices. Ensuite, pour l'exécution en elle même, comme décrit plus haut, pour activer un neurone d'une couche n, nous sommons le produit des valeurs d'activation des neurones de la couche $n-1$ avec les poids les reliant au neurone à activer et enfin nous ajoutons un biais à cette somme.

Dans notre implémentation, pour avoir le tableau des valeurs d'activation des neurones d'une couche n, nous multiplions la matrice de poids reliant la couche $n-1$ à la couche n et le tableau des valeurs d'activation des neurones de la couche

$n-1$ et ensuite nous ajoutons la matrice de biais reliant la couche $n-1$ à la couche n au résultat obtenu.

Nous pouvons résumer cette exécution à l'aide de l'équation suivante :

$$\begin{bmatrix} a_0^n \\ \vdots \\ a_j^n \end{bmatrix} = \begin{bmatrix} {w^{n-1}}_{00} & \cdots & {w^{n-1}}_{i0} \\ \vdots & \ddots & \vdots \\ {w^{n-1}}_{0j} & \cdots & {w^{n-1}}_{ij} \end{bmatrix} \cdot \begin{bmatrix} {a^{n-1}}_0 \\ \vdots \\ {a^{n-1}}_i \end{bmatrix} + \begin{bmatrix} {b^{n-1}}_0 \\ \vdots \\ {b^{n-1}}_j \end{bmatrix}$$

Pour plus de précision, on peut écrire :

$\forall k \in [\![0\ ;\ j]\!] : a_k^n = \sum_{t=0}^{i} {w^{n-1}}_{tk}.{a^{n-1}}_t + {b^{n-1}}_k$

avec :
a_k^n : valeur d'activation du k^{ieme} neurone de la n^{ieme} couche.
${w^{n-1}}_{tk}$: poids reliant le t^{ieme} neurone de la $n-1^{ieme}$ couche au k^{ieme} neurone de la n^{ieme} couche.
b_k^n : biais pour l'activation du k^{ieme} neurone de la n^{ieme}

Intéressons nous maintenant au cas de notre réseau cherchant à apprendre la fonction XOR. Comme dit plus haut, les différentes matrices sont représentées par des tableaux de flottants, et les valeurs d'activation des neurones de la première couche correspondent à la valeur des deux bits dont on veut faire le XOR. Les différents cas sont :

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ et } \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Généralement, dans un réseau de neurones, une fonction d'activation, telle qu'une fonction sigmoid ou softmax, est utilisée pour activer les neurones de la couche suivante. En prenant une fonction d'activation σ, on aurait alors :

$\forall k \in [\![0\ ;\ j]\!] : a_k^n = \sigma(\sum_{t=0}^{i} {w^{n-1}}_{tk}.{a^{n-1}}_t + {b^{n-1}}_k)$

Cependant, pour ce *proof of concept*, nous avons décidé de ne pas utiliser de fonction d'activation car la fonction XOR n'est pas une fonction complexe, ce qui permet d'obtenir de très bons résultats sans avoir besoin d'ajouter une fonction d'activation.

3.4.4 Entraînement du réseau

Tout d'abord, pour calculer le coût d'une exécution du réseau, il a fallu implémenter une fonction qui prend en argument deux vecteurs (une sortie et la réponse attendue), et qui calcule la somme des écarts au carré pour chaque indice du tableau.

$$\text{C(réseau)} = \frac{1}{n}\sum_{i=0}^{n-1}(output[i] - wanted[i])^2$$

Ensuite vient le calcul du gradient. Pour ce faire, on itère sur chaque élément des différents tableaux de poids et biais, puis la dérivée du coût par rapport à cet élément est calculé de la façon suivante : (on nomme Z l'activation d'un neurone à la couche i, qui est égal à la somme pondérée des activations de la couche précédente)

Si l'élément est entre la 2^{ieme} et la 3^{ieme} couche :

$$\frac{\partial C}{\partial Element} = \frac{\partial C}{\partial Z}\frac{\partial Z}{\partial Element}$$

Si l'élément est entre la première et la 2^{ieme} couche :

$$\frac{\partial C}{\partial Element} = \sum_{i=0}^{n-1}\frac{\partial C}{\partial Z_i}\frac{\partial Z_i}{\partial a_i}\frac{\partial a_i}{\partial Element}$$

avec :

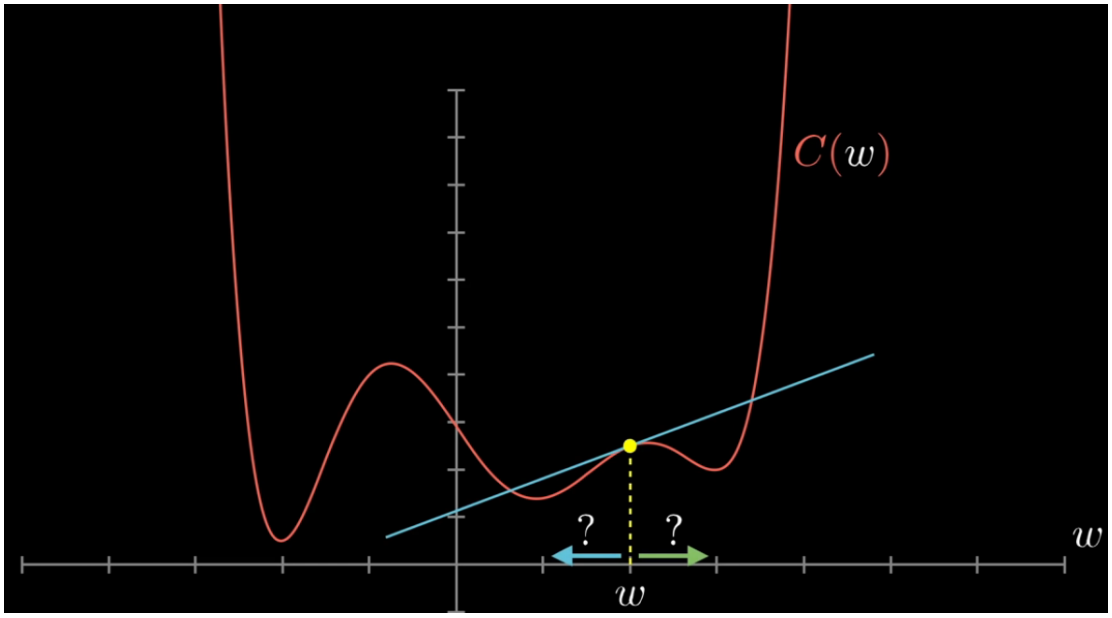
$$\frac{\partial C}{\partial Z} = (output[i] - wanted[i])$$

$$\frac{\partial Z}{\partial Element} =$$

- Si l'élément est un poids : l'activation multipliée par ce poids
- Si l'élément est une activation (pour $\frac{\partial Z_i}{\partial a_i}$) le poids multiplié par cette activation
- Si l'élément est un biais : 1

(à noter qu'ici il n'y a pas de fonctions d'activation, ce qui simplifie ces expressions)

Exemple du calcul du gradient du coût en fonction d'un paramètre



Pour l'entraînement en lui même, dans le cas de la fonction XOR, nous avons une liste exhaustive des entrées, ce qui aide grandement à la création de jeux d'entraînement.

En effet, pour entraîner notre réseau, nous le faisons s'exécuter sur les 4 entrées possibles (00,01,10,11) et calculons le gradient du coût à chaque fois. Puis nous retranchons aux paramètres du réseau les 4 dérivées correspondantes ajoutées entre elles et multipliées par un coefficient d'apprentissage. Cette opération est répétée tant que le coût est supérieur à une valeur (ici 0.01).

Dans la pratique, il y a un problème que nous n'avons pas réussi à régler : si nous entraînons notre réseau sur les 4 entrées, les réponses de la dernière couche convergerons vers 0.5 pour toute entrée, alors que si l'on en met que 3, le réseau produira de très bons résultats sur ces entrées.

Nous avons conjecturé que ce problème est dû à la taille du jeu d'entraînement qui est vraiment petit combiné au fait que la moitié des entrées fait tendre vers une réponse et l'autre moitié vers son contraire.

Ce réseau est néanmoins la preuve que l'exécution et la rétro-propagation sont fonctionnelles et peuvent permettre au réseau d'apprendre.

3.4.5 Sauvegarde et chargement des poids et des biais du réseau

Pour sauvegarder les poids et les biais du réseau de neurones, nous utilisons la manipulation de fichiers. Nous ouvrons un fichier puis, pour chaque poids et biais du réseau de neurones, nous l'écrivons dans ce fichier. Tous les poids et biais écrivant dans ce fichier sont séparés les uns des autres par une virgule, afin de faciliter le chargement de ces données. Enfin, nous fermons ce fichier.

Pour charger les poids et les biais du réseau, nous ouvrons le fichier où sont sauvegardés ces derniers. Puis, nous récupérons chaque donnée dans un tableau et nous chargeons chacune de ces données dans le bon tableau de poids ou de biais. Enfin nous fermons le fichier.

3.5 Jeu d'images pour l'apprentissage

Afin d'entraîner l'intelligence artificielle nous avons constitué une petite bibliothèque d'images. Elle est pour l'instant constituée uniquement de 3 images basiques contenant des caractères voulus :

- Une image d'un tableau contenant les lettres majuscules, minuscules, les chiffres et quelques caractères spéciaux

- Une image d'un "mot" contenant toutes les lettres en majuscule et minuscule

6 une image contenant tous les chiffres et caractères spéciaux classiques sur un clavier QWERTY américain

Cette bibliothèque pourra être agrandie dans le futur afin d'entraîner l'intelligence artificielle sur des images plus complexes.

Table de caractères

Lettres majuscules et minuscules

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

Caractères spéciaux

`1234567890-=~!@#$%^&*()_+[]{}'\"|,./<>?

3.6 Manipulation de fichiers pour la sauvegarde des résultats

Pour sauvegarder le résultat, nous avons décidé de sauvegarder la chaîne de caractère, reconstituée à partir de la reconnaissance de caractère effectuée par le réseau de neurone, dans un fichier. Pour cette soutenance, seule la fonction de sauvegarde d'une chaîne de caractères dans un fichier a été réalisée. Cette fonction ouvre un fichier, écrit la chaîne de caractères à l'intérieur et enfin ferme ce fichier.

3.7 Prochains objectifs

Lors de la prochaine soutenance nous souhaitons avoir une reconnaissance de caractères fonctionnelle pour toutes sortes d'images comprenant des couleurs ou des bruits importants. Pour cela, plusieurs fonctionnalités doivent encore être implémentées ou perfectionnées :

- Rotation de l'image : L'image donnée en paramètre n'est pas obligatoirement droite ce qui est nécessaire pour obtenir une segmentation correcte. Pour cela, deux choix s'offrent à nous : implémenter une rotation automatique (une fonction recadre automatiquement l'image en détectant les lignes de caractères) ou manuelle (l'utilisateur recadre l'image à l'aide de boutons implémentés dans l'interface graphique)

- Mise en page : Il s'agit d'implémenter la fonction évoquée dans la partie 3.3.3 Mise en page.

- Réseau de neurones : Le "XOR" a été une bonne expérience pour comprendre le fonctionnement des réseaux de neurones. La prochaine étape va donc être de créer le vrai réseau pour l'OCR, c'est-à-dire un plus grand réseau de neurones qui devra être capable, après de nombreux entraînements de reconnaître un caractère donné en entrée.

- Interface graphique : Cette interface permettra à l'utilisateur d'utiliser notre OCR plus facilement et intuitivement qu'avec une interface de commandes. Elle contiendra plusieurs fonctionnalités comme un champ pour entrer le chemin de l'image à analyser ou un écran pour visualiser cette image.

- Jeu d'images : Afin d'entraîner efficacement notre réseau de neurones, il est important d'avoir un jeu d'images complet et exact comportant tous les cas possibles que le réseau de neurones peut rencontrer.

- Bonus : Gestion du multi-colonnes : La gestion du multi-colonnes permettrait de prendre en charge plus d'images. Elle pourrait être appelée sur demande de l'utilisateur à travers l'interface graphique. Dans ce cas, avant la segmentation du texte une fonction se lancerait pour découper l'image en plusieurs parties à enregistrer dans d'autres fichiers. Ensuite la segmentation et la reconnaissance de caractères s'appliqueraient tour à tour sur chacune de ces nouvelles images.

4 Conclusion

Pour conclure ce premier rapport concernant notre projet, nous pouvons dire que la réalisation de ce logiciel de reconnaissance optique de caractères est en bonne voie. Une répartition claire des tâches entre les différents membres du groupe nous a permis de commencer rapidement la réalisation de nos objectifs.

Pour ce premier rapport, nous pensons être en adéquation avec les attentes de l'EPITA, ainsi que nos propres attentes, tant sur le traitement de l'image, le réseau de neurones, les jeux d'images pour l'apprentissage et la sauvegarde du résultat.

D'une part, cela nous a permis de découvrir des concepts et des sujets nouveaux tel que l'intelligence artificielle, les réseaux de neurones et la manipulation d'image avancée. D'autre part, le début de ce projet nous a montré que, malgré le sujet imposé, l'engouement et le travail fourni n'a pas été moindre en comparaison avec le projet précédent dont le choix du sujet avait été laissé aux étudiants.

Les membres de notre équipe toujours motivés et rigoureux devront à présent redoubler d'effort pour ne pas perdre nos prochains objectifs de vue afin de pouvoir aller au bout de ce projet.