



PROJET OCR CLAPS

RAPPORT DE SOUTENANCE 1

Coureau Adrien
Del-Pozo Corentin
Fournier Pierre-Antoine
Lou Sophie

Table des matières

1	Introduction	3
2	Répartition des Tâches	4
3	Avancement	4
4	Notre Equipe	4
4.1	Lou Sophie	4
4.2	Fournier Pierre-Antoine	5
4.3	Del-Pozo Corentin	5
4.4	Coureau Adrien	5
5	Pourquoi le format BMP	6
5.1	Parcourir une image BMP	6
6	Réduction de Bruit	7
6.1	Le flou médian	7
6.2	Le Flou Gaussien	7
6.3	Contraste	8
7	Rotation de l'image	10
7.1	Rotation avec une seul matrice	10
7.1.1	Problème rencontré	10
7.2	Rotation par cisaillement	11
7.2.1	Problème rencontré	11
7.3	Rotation avec matrice de rotation	12
8	Binarisation	13
9	Détection de la Grille	14
9.1	Détection des formes	15
9.2	Filtrage des formes	17
9.3	Détection des groupes	18
9.4	Filtrage des groupes	19
9.5	Récupération des lettres	20
10	Réseau de neurones	21
10.1	XOR	21
11	Solver	21
12	Interface	22
12.1	Du concept...	22
12.2	... Au concret	23
13	Conclusion	25

1 Introduction

Tout au long du projet CLAPS Word Search OCR, la présence d'un rapport de projet est primordiale. Il nous permettra de nous concentrer sur le bilan de ce qui a été fait et la manière dont nous allons traiter le projet sur le temps imparti (du mois de septembre au mois de décembre). Nous présenterons ainsi l'état d'avancement de notre projet, ainsi qu'un compte rendu qui détaillera les tâches prévues dans le cahier des charges.

L'objectif de ce projet est de nous offrir la possibilité de mettre en oeuvre de manière concrète, les connaissances acquises en cours. Cela nous donnera ainsi l'occasion d'augmenter nos acquis personnels nécessaires au projet.

Le but final de ce projet est de réaliser un logiciel de type OCR qui résout une grille de mots cachés en 3 mois. Notre application prendra donc en entrée une image représentant une grille de mots cachés et affichera en sortie la grille résolue.

Ainsi, nous avons découpé ce projet en plusieurs parties :

- Reconnaissance des caractères (réseau de neurones)
- Résolution d'une grille de mots cachés (solver)
- Prétraitement de l'image
- Interface graphique

2 Répartition des Tâches

Tâches	Pierre-Antoine	Corentin	Adrien	Sophie
Interface Utilisateur			✓	
Niveau de gris	✓			
Réduction de bruits	✓			
Rotation				✓
Binarisation			✓	
Détection de Grille		✓		
Solver				✓
Réseaux de Neurones		✓		

3 Avancement

Tâches	04/10/2024	09/12/2024
Interface Utilisateur	60%	100%
Niveau de gris	100%	100%
Réduction de bruits	50%	100%
Rotation	50%	100%
Binarisation	100%	100%
Détection de Grille	70%	100%
Solver	100%	100%
Réseaux de Neurones	50%	100%

4 Notre Equipe

4.1 Lou Sophie

Animée par mon attrirance pour les mathématiques, la réalisation de la rotation manuelle d'image, m'a permis de nourrir cette curiosité, par l'application à des choses concrètes et visuelles (ici, des grilles de mots mêlées). Le fait d'avoir l'occasion de mettre en pratique des connaissances mathématiques, pouvant paraître abstraites, en l'associant à de l'informatique, m'a rappelé l'importance de ces deux domaines.

De plus, en mettant en place un algorithme de Solver, cela m'a permis de comprendre l'implémentation d'un algorithme de résolution de mots cachés. Ensemble, nous avons pu déterminer quel était le plus efficace à implémenter. Toujours en quête d'application concrète, le fait de savoir de manière immédiate si un mot a été trouvé ou non, était très encourageant.

4.2 Fournier Pierre-Antoine

Pour ce projet, j'ai choisi de faire les niveaux de gris et la réduction de bruit car le traitement d'image m'intéresse. De plus, j'envisage de faire la majeur des images à EPITA. Ce projet est donc un bon moyen de découvrir si le traitement d'images me plaît.

4.3 Del-Pozo Corentin

Pour ce projet, j'ai choisi de m'occuper du réseau de neurones, car c'est un sujet en informatique qui me passionne beaucoup. Avant de commencer le projet, je m'étais déjà renseigné sur son fonctionnement et sur toutes les possibilités qu'il offre. Le développement du réseau de neurones m'a permis d'approfondir mes connaissances en apprentissage supervisé et d'appliquer des concepts théoriques à des problèmes pratiques. Cela m'a également permis de mieux comprendre comment les algorithmes peuvent apprendre et s'adapter à des données variées, ce qui est essentiel dans le contexte de l'intelligence artificielle.

En ce qui concerne la détection de la grille, j'ai choisi de m'occuper de ce problème parce qu'au début, notre groupe n'avait aucune idée de quelle direction prendre. J'apprécie particulièrement les défis qui consistent à chercher des solutions dans un environnement où nous disposons de peu d'indices et de ressources. Cette contrainte nous pousse à développer nos propres solutions et à faire preuve de créativité, ce qui rend le processus d'apprentissage d'autant plus stimulant.

4.4 Coureau Adrien

Passionné d'informatique depuis petit, l'univers de la création m'attire beaucoup (dessin, dessin digital, production de musique). J'ai choisi de développer l'interface du projet en gtk3 avec l'aide de Glade parce que j'aime créer un produit fini et cohérent à partir de zéro.

La conception d'une interface facile à prendre en main est cruciale pour fluidifier l'expérience de l'utilisateur ainsi que renforcer la rapidité des tâches et des tests pour les développeurs. Elle est la liaison entre ce que l'on voit, les boutons, et toutes les actions que peut effectuer notre application d'OCR.

5 Pourquoi le format BMP

Nous avons choisi ce format car il rend plus simple le parcours d'une image, tout en conservant une bonne qualité avec ce format. Cependant, il est plus volumineux que le format PNG, il est donc important de supprimer toutes les images en BMP que nous avons créées avant un éventuel crash. Pour cela, un script suffit. Nous devons aussi convertir l'image fournie par l'utilisateur avant d'effectuer toute opération.

5.1 Parcourir une image BMP

Pour faire un parcours d'une image, il nous suffit d'appeler la fonction

```
1    SDL_LoadBMP("img.bmp")
```

qui renvoie une **surface**. La **surface** est une structure de la librairie SDL qui contient la hauteur, la largeur et le pointeur vers le tout premier pixel de l'image.

L'image est contenue dans un seul tableau : les 3 premières valeurs sont les pixels rouge, vert, et bleu du premier pixel, puis les 3 suivantes ceux du second et ainsi de suite.

Pour parcourir une image nous utilisons le code suivant :

```
1  for (int i = 0; i < height; i++)
2  {
3      for (int j = 0; j < width; j++)
4      {
5          Uint8* pixel = pix + j * pitch + i *
6                      bpp;
7          // do stuff on the image
8      }
}
```

Nous avons une double boucle plutôt classique. Ensuite, pour récupérer un pointeur vers la couleur rouge du pixel actuel, il nous suffit de faire le calcul suivant :

pix est le pointeur du tout premier pixel de l'image, accessible avec **surface->pixels**. On lui ajoute **i** qui correspond à la rangée horizontale actuelle multipliée par le **pitch** (accessible via **surface->pitch**). Le **pitch** est le nombre d'octets entre les lignes. Il n'est pas toujours égal au nombre de pixels multiplié par le nombre d'octets par pixel (**bpp**), car des octets peuvent être ajoutés en bout de ligne pour des raisons de performance. Ainsi, **pix + i * pitch** nous permet de récupérer la bonne ligne de pixels de l'image. Nous ajoutons **j**, correspondant au **j**-ème pixel de la **i**-ème rangée, et nous multiplions **j** par le nombre d'octets par pixel (**surface->format->BytesPerPixel**). Dans notre cas, **BytesPerPixel** vaut 3.

6 Réduction de Bruit

Dans les images données dans le cahier des charges, le niveau 2 contient des images avec du bruit. Nous avons implémenté deux algorithmes pour réduire le bruit : le flou médian et le flou Gaussien. Pour simplifier les algorithmes, nous appliquons un "grayscale" car les valeurs des trois couleurs (RGB) sont égales.

6.1 Le flou médian

Le flou médian consiste à placer dans une liste les 8 valeurs des pixels autour, puis à prendre la valeur du milieu pour l'attribuer au pixel actuel. Nous utilisons un tri par insertion pour ranger les valeurs. À noter qu'il est possible d'élargir le noyau de 3×3 à un noyau de 5×5 . Cet algorithme permet d'éliminer les pixels ayant une valeur très différente des pixels proches, tout en conservant les contours, ce qui est pratique pour préserver la forme des lettres.

6.2 Le Flou Gaussien

Le flou Gaussien utilise également un noyau, mais il est prédéfini. On peut choisir sa taille (3×3 , 5×5 , ...), et dans notre cas nous utilisons un noyau de taille 3×3 . Pour calculer les valeurs du noyau, nous appliquons la fonction Gaussienne, qui produit une courbe similaire à l'image fournie (elle peut être plus pentue ou plus allongée).

Les valeurs fournies à la fonction Gaussienne, qui prend trois arguments, sont X et Y , représentant respectivement les coordonnées (x , y) des pixels autour du milieu du noyau de coordonnées $(0,0)$. Le troisième argument est `sigma`, qui permet d'ajuster la luminosité de l'image. Pour déterminer le `sigma` optimal, nous avons testé plusieurs valeurs croissantes pour trouver la meilleure. La fonction est la suivante :

$$\mathcal{G}(x, y) = \frac{1}{2\pi\sigma^2} \exp^{-\frac{x^2+y^2}{2\sigma^2}} \text{ et } \sigma \in \mathbb{R}$$

```

1 #define M_PI           3.14159265358979323846
2
3 double gFunc(int x, int y)
4 {
5     const double sig = 0.58;
6     double part1 = 1 / (2 * M_PI * sig * sig);
7     double power = -(x * x + y * y) / (2 * sig
8         * sig);
9     double part2 = exp(power);
10    return part1 * part2;
}
```

6.3 Contraste

Nous avons aussi utilisé un algorithme de contraste d'image pour accentuer le noir et le blanc sur les images avant d'appliquer les algorithmes de réduction de bruit. Le contraste fonctionne de la manière suivante :

On construit une liste des 8 pixels autour du pixel actuel et on compare ensuite avec la condition suivante :

```
1 // sidepixel is an array containing the 8  
  pixels around pixel  
2 if (side_pixel[k] < pixel[0] && pixel[0] < 200)  
  { /*...*/ }
```

Si cette condition est validée, on retire 50 à la valeur actuelle du pixel. Voici quelques exemples avant et après l'application de notre algorithme :

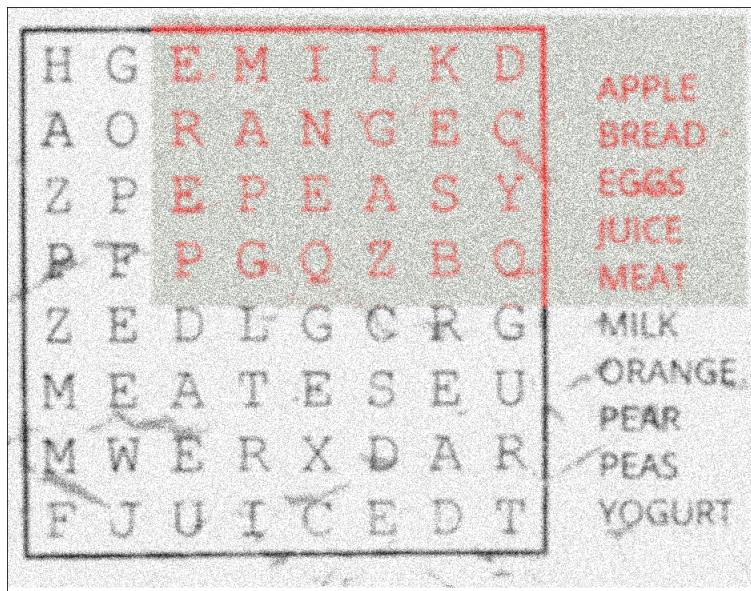


FIGURE 1 – Avant

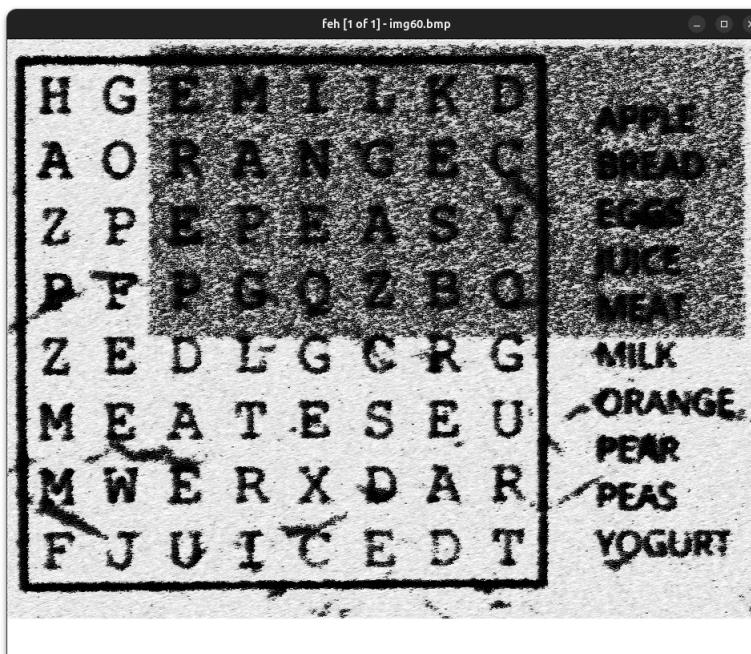


FIGURE 2 – Après

7 Rotation de l'image

Plusieurs méthodes peuvent être appliquées afin d'effectuer une rotation d'image. Voici 2 différentes techniques, aboutissant à celle conservée, étant la plus performante.

7.1 Rotation avec une seul matrice

Cette méthode de rotation consiste à utiliser cette formule :

$$\begin{aligned}x' &= \cos(\theta) * x - \sin(\theta) * y \\y' &= \sin(\theta) * x + \cos(\theta) * y\end{aligned}$$

Provenant de la représentation matricielle suivante :

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

(avec θ l'angle en degré)

7.1.1 Problème rencontré

Malgré la simplicité de cette formule, l'application de celle-ci provoque de manière persistante des problèmes d'aliasing, dû à l'apparition non voulue de points noirs, pouvant nuire à la détection des lettres. La rotation avec une seule matrice affecte donc la qualité de l'image, dû à une mauvaise utilisation. En effet, nous sommes partis des coordonnées des pixels de l'image d'origine et nous avons calculé leur position après rotation dans la nouvelle image. Cette technique s'appelle l'échantillonnage en avant, ayant pour effet de laisser des trous dans l'image.

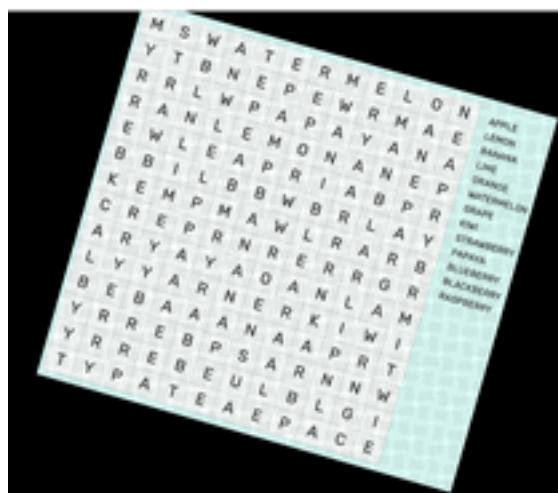


FIGURE 3 – Problème 1

7.2 Rotation par cisaillement

Ne sachant pas que le problème rencontré était dû à une mauvaise utilisation, nous avons ainsi voulu appliquer le principe de rotation par cisaillement (aussi appelé rotation par Shearing), qui utilise l'algorithme de Shearing, prenant chaque pixel de l'image, puis applique 3 multiplications successives, par des matrices avec θ notre angle en radian.

Elle utilise donc cette représentation matricielle :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & -\tan(\frac{\theta}{2}) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \sin(\theta) & 1 \end{pmatrix} \begin{pmatrix} 1 & -\tan(\frac{\theta}{2}) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

(avec θ l'angle en degré)

Afin que la rotation puisse s'effectuer dans le sens horaire et non anti-horaire, nous avons ainsi inverser les signes obtenant les formules suivantes :

— 1^{er} cisaillement :

$$x' = x + \tan(\theta) * yy' = y$$

— 2^{eme} cisaillement :

$$x'' = x'y'' = x'' + \sin(\theta) * (-1) * x' * y'$$

— 3^{eme} cisaillement :

$$x''' = x'' + \tan(\theta) * yy''' = y''$$

(avec θ l'angle en degré)

Voici un exemple de représentation avec un angle de 20° :

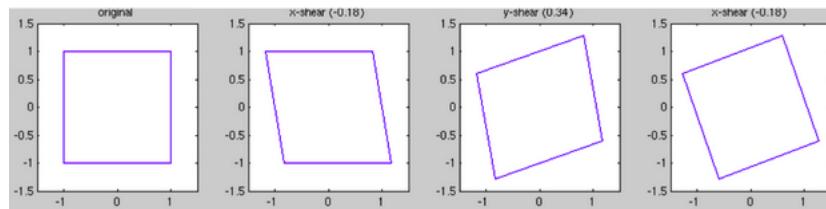


FIGURE 4 – Rotation de 20°

7.2.1 Problème rencontré

Malgré la netteté de l'image, celle-ci affichait des failles, par une perte d'une partie de l'image lors d'angles, comme celui de 35°. En effet, il était difficile de la centrer correctement.

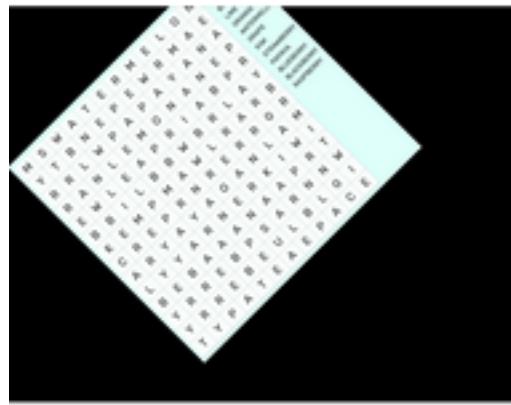


FIGURE 5 – Problème 2

7.3 Rotation avec matrice de rotation

Enfin, cette méthode a été privilégiée par son efficacité, utilisant la formule de “matrice de rotation” vu en 3.1 :

$$x' = \cos(\theta) * x - \sin(\theta) * y$$

$$y' = \sin(\theta) * x + \cos(\theta) * y$$

(avec θ l’angle en degré)

Cependant, au lieu de partir des coordonnées des pixels de l’image d’origine et nous avons calculé leur position après rotation dans la nouvelle image. Nous sommes partis des coordonnées de pixels dans la nouvelle image puis nous avons recherché les pixels correspondants dans l’image d’origine (c’est un ”échantillonnage inverse”), ce qui garantit que chaque pixel de la nouvelle image est couvert et minimise les pertes de pixels.



FIGURE 6 – Résultat final

8 Binarisation

Pour effectuer la binarisation de l'image, nous avons appliqué l'algorithme de Sauvola. Cet algorithme est basé sur la méthode de seuil local, qui adapte le seuil de binarisation en fonction des variations d'intensité de l'image. Il est particulièrement efficace pour les images ayant un contraste faible ou des variations d'éclairage.

La formule de Sauvola pour le calcul du seuil $T(x, y)$ au pixel (x, y) est donnée par :

$$T(x, y) = \mu(x, y) + k \cdot (\sigma(x, y) - m)$$

où :

- $\mu(x, y)$ est la moyenne des intensités des pixels dans un voisinage local autour du pixel (x, y) .
- $\sigma(x, y)$ est l'écart type des intensités des pixels dans le même voisinage.
- k est un facteur de sensibilité qui peut être ajusté pour contrôler la binarisation.
- m est un paramètre constant (souvent pris égal à 128 pour des images en niveaux de gris).

Pour chaque pixel de l'image, nous comparons son intensité $I(x, y)$ avec le seuil calculé $T(x, y)$. Si $I(x, y) > T(x, y)$, le pixel est classé comme blanc (valeur 1), sinon il est classé comme noir (valeur 0). Cette approche permet de segmenter efficacement les objets d'intérêt dans l'image, en tenant compte des variations locales.

En utilisant cet algorithme, nous avons réussi à obtenir une binarisation robuste qui améliore la qualité des étapes suivantes dans le traitement de l'image.

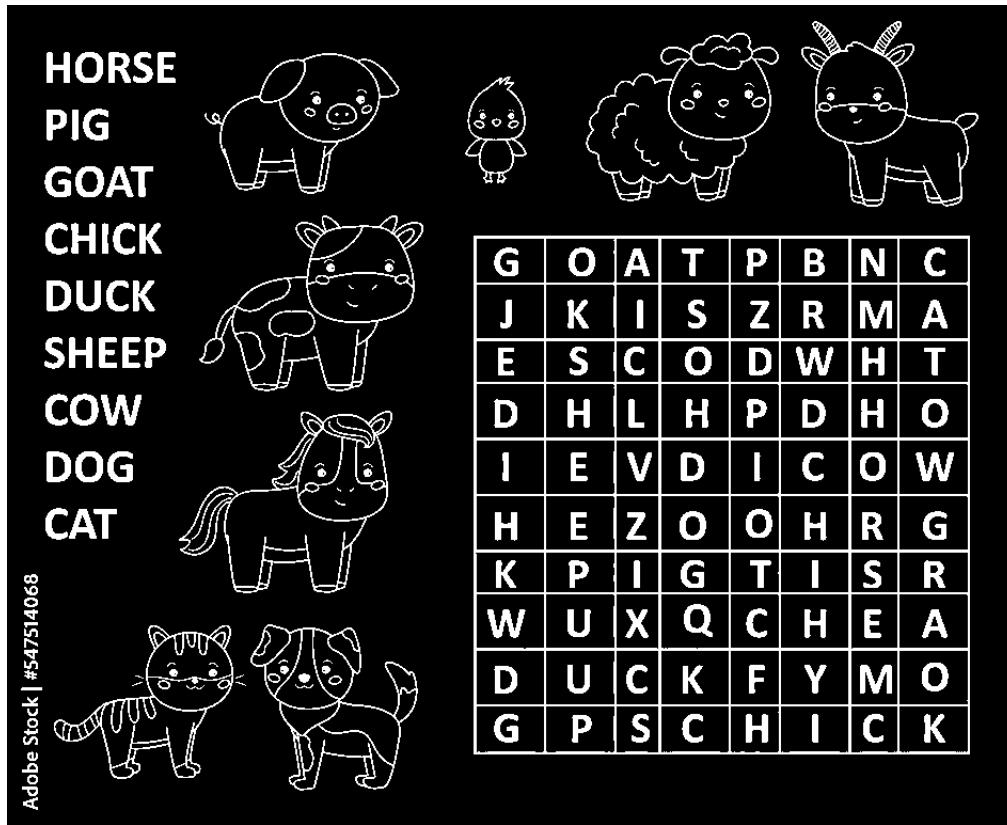


FIGURE 7 – Image après la binarisation

9 Détection de la Grille

Une fois l'image binarisée, nous allons commencer à détecter la grille et la liste de mots. Pour développer une stratégie efficace, nous avons étudié les méthodes de détection de grille et consulté de nombreuses ressources en ligne, notamment sur l'utilisation de Sobel et d'autres algorithmes de détection de grille.

Cependant, dans notre cas, il se peut qu'il n'y ait pas de grille sur l'image, et la liste de mots, quant à elle, n'est jamais encadrée par une grille. Nous avons donc choisi de procéder à l'inverse : au lieu de détecter la grille pour ensuite identifier les lettres, nous allons d'abord détecter les lettres, puis déterminer lesquelles appartiennent à la grille et lesquelles à la liste de mots.

Pour cela, nous appliquerons plusieurs étapes afin d'atteindre ce résultat.

9.1 Détection des formes

Nous commençons par détecter tous les amas de pixels blancs connectés entre eux, en utilisant un algorithme similaire à celui du "pot de peinture" des logiciels de dessin, qui parcourt l'image de manière récursive.

Deux matrices sont créées : l'une pour indiquer si un pixel a été visité et à quelle forme il appartient, et une autre qui stocke l'image sous forme de 1 (pixels noirs) et 0 (pixels blancs), afin de faciliter la distinction des pixels blancs et noirs dans la fonction récursive.

L'algorithme parcourt toute l'image et, dès qu'un pixel blanc non visité est trouvé, il initialise une nouvelle forme.

Structure de la forme :

```
1  typedef struct Shape
2  {
3      int id; // Identifier of the shape
4      int Cx, Cy; // Coordinates of the center of
           // the shape
5      int h, w; // Height and width of the shape
6
7      // Coordinates of the edges of the shape
8      int Maxj, Maxi;
9      int Minj, Mini;
10
11     int Matj, Mati; // Coordinates of the
           // letter in the grid
12
13     int Len; // Number of pixels in the shape
14 } Shape;
```

Un identifiant unique est attribué à chaque forme, puis l'algorithme parcourt récursivement les quatre pixels directement adjacents. Pour chaque pixel visité, sa valeur dans la matrice de vérification est remplacée par l'identifiant de la forme. Cette opération est répétée pour les pixels adjacents suivants.

Une fois la forme entièrement détectée, des informations comme le nombre de pixels constituant la forme ainsi que sa largeur et sa hauteur sont stockées.

Ensuite, un premier tri est effectué pour éliminer les formes trop grandes, par exemple celles qui occupent plus de la moitié de la taille de l'image.

Si une forme est valide, elle est ajoutée à la liste des formes. Pour cet algorithme, nous avons implémenté une structure de liste chaînée pour faciliter la manipulation des formes trouvées.

Structure de la liste chaînée :

```
1 typedef struct Node
2 {
3     Shape* data;
4     struct Node* next;
5 } Node;
```

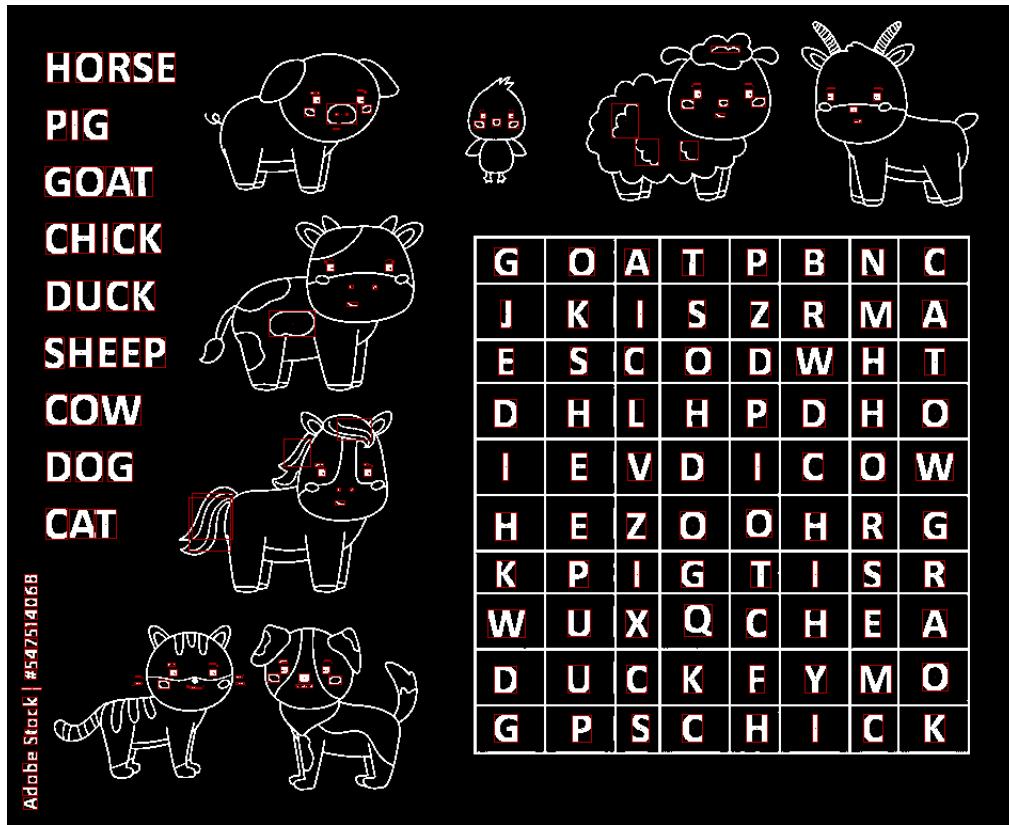


FIGURE 8 – Image après la détection des formes

9.2 Filtrage des formes

Une fois toutes les formes détectées, on filtre les formes non valides pour garder uniquement les lettres. Pour cela, nous calculons la somme de la largeur, de la hauteur et de la taille de toutes les formes, puis supprimons celles qui s'écartent trop de la moyenne. Ce filtre permet d'éliminer les formes trop petites.

En calculant la moyenne des hauteurs, on obtient la taille moyenne de la police, un paramètre efficace pour le filtrage.

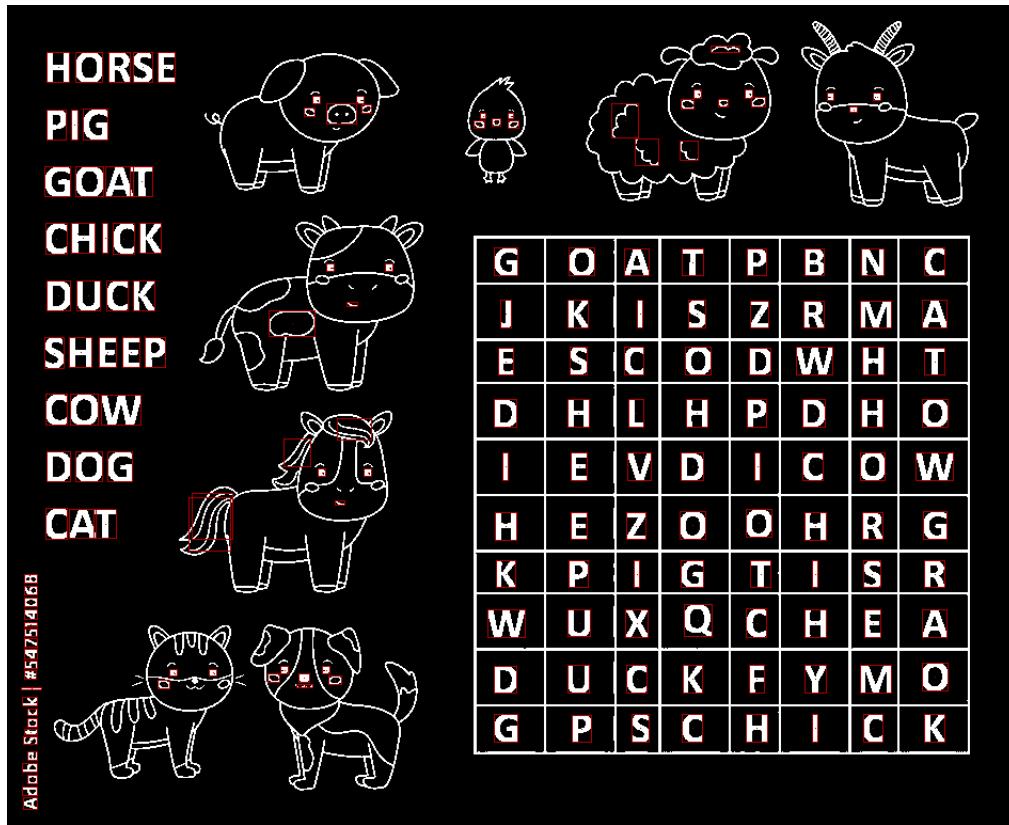


FIGURE 9 – Image après le filtrage des formes

9.3 Détection des groupes

À ce stade, la plupart des formes sont des lettres, mais certaines formes invalides peuvent subsister. Nous passons à la détection des groupes et filtrerons les groupes invalides.

Pour détecter les groupes, nous parcourons chaque forme et vérifions plusieurs critères :

- La distance entre deux formes est-elle suffisamment petite par rapport à la taille de la forme ?
- La hauteur de la forme est-elle proche de la moyenne des hauteurs des formes dans le groupe ?

Pour chaque forme assez proche, on vérifie qu'elle n'appartient pas déjà à un autre groupe en consultant une liste de formes rencontrées. Si elle est inédite, nous l'ajoutons au groupe et à la liste des formes visitées.

Les groupes sont des listes chaînées de formes.



compare/latex-

adrien ?expand=1

FIGURE 10 – Image après la détection des groupes

9.4 Filtrage des groupes

Une fois tous les groupes créés, nous appliquons un filtre pour ne conserver que les groupes de mots ou de lettres. Pour cela, nous calculons la taille moyenne des groupes, définie par la somme des tailles de chaque forme contenue, puis éliminons ceux en dessous de cette moyenne.

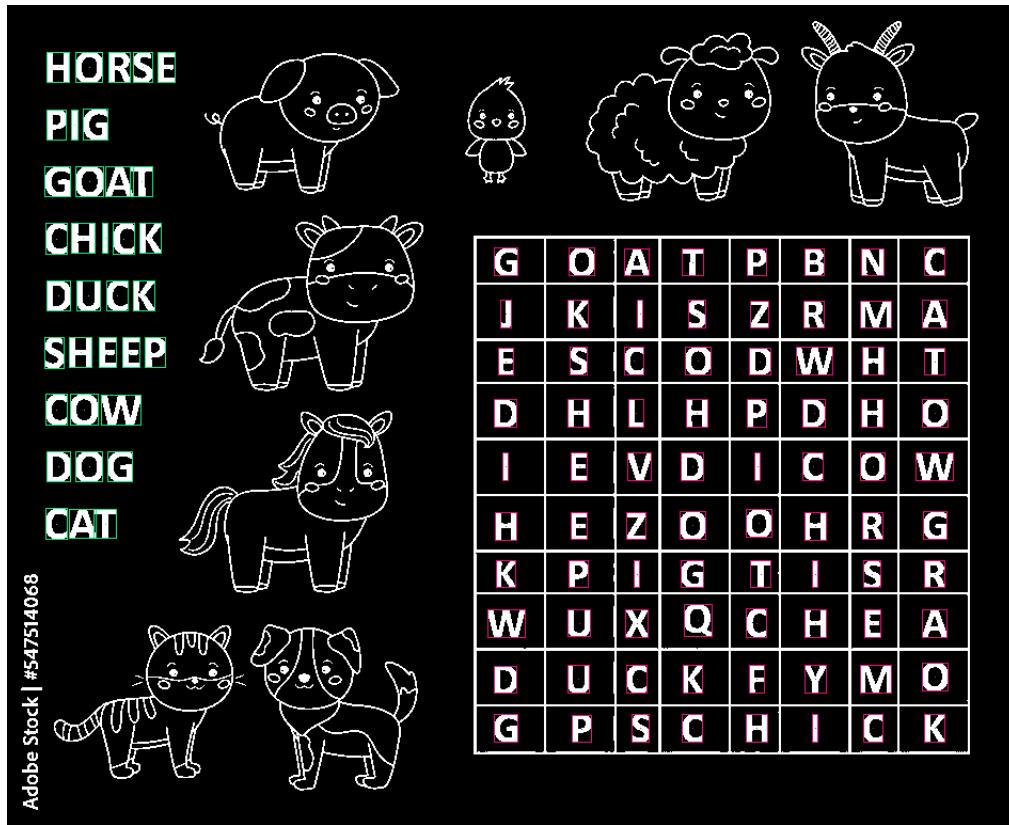


FIGURE 11 – Image après le filtrage des groupes

Après le filtre, nous supposons que le plus grand groupe est la grille, les autres groupes représentant les mots.

9.5 Récupération des lettres

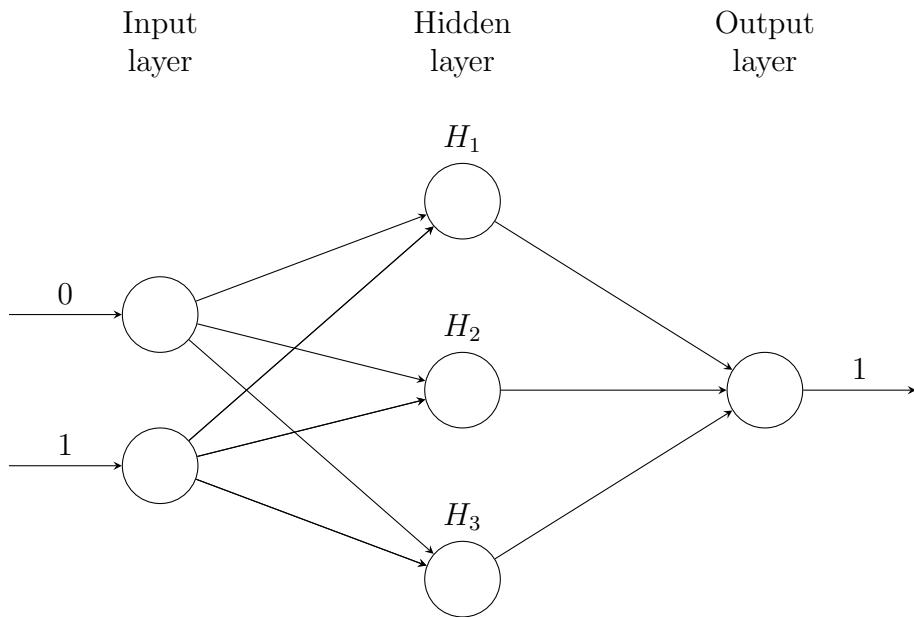
Avec seulement les groupes de mots ou de lettres conservés, nous parcourons chaque groupe pour en extraire les lettres et les sauvegarder sous forme d'images dans un dossier par groupe, en vue d'un traitement ultérieur avec le réseau de neurones.

Pour ce faire, nous utilisons la matrice de pixels déjà visités. Nous parcourons cette matrice entre les coordonnées de début et de fin de la forme dans la structure Shape. Pour chaque élément contenant l'identifiant de la forme, nous reportons ce pixel dans une nouvelle image aux dimensions de la lettre, sauvegardée dans un dossier dédié.

10 Réseau de neurones

Notre réseau de neurones suit une structure classique avec une couche d'entrée, une couche cachée et une couche de sortie. Le nombre de nœuds par couche varie selon l'application, comme le XOR ou le traitement d'image.

10.1 XOR



11 Solver

Pour résoudre le mot mêlé, on récupère la grille de lettres dans un fichier texte généré par le code. Cette grille est ensuite stockée dans une matrice de caractères.

Étape 1 Trouver la première lettre : on parcourt la grille jusqu'à la première lettre du mot.

Étape 2 Trouver la deuxième lettre : une fois la première lettre trouvée, on recherche la deuxième dans les 8 cases adjacentes. Si elle est trouvée, on détermine la direction pour le reste des lettres.

Étape 3 Trouver le reste du mot : en vérifiant que le mot peut tenir dans la direction sans dépasser les limites de la grille, on parcourt chaque lettre restante du mot.

Grâce à cet algorithme, on est certain de trouver le mot s'il existe dans la grille et d'en renvoyer les coordonnées. Ces informations seront très utiles plus tard pour indiquer visuellement l'emplacement de chaque mot dans la grille.

12 Interface

La création d'une interface cohérente et simple à utiliser est cruciale pour fluidifier l'expérience utilisateur. Elle a constitué une part importante du travail à effectuer sur le projet. L'interface est la liaison entre ce que voit l'utilisateur de notre logiciel et le traitement appliqué pour trouver la solution finale du problème initial.

12.1 Du concept...

La première étape a été de conceptualiser l'interface visuelle à l'aide du logiciel **Glade**. Notre objectif était de concevoir un ensemble simple avec uniquement des boutons à gauche pour modifier les paramètres de traitement, ainsi qu'une image à droite pour visualiser la solution et toutes les étapes qui ont mené à cette solution.

L'interface est donc composée d'une colonne de paramètres modifiables à gauche et d'une image dynamiquement modifiée à droite.

Dans la colonne de gauche, on retrouve différentes sections :

- Un titre et un logo créés par notre équipe, montrant à la fois deux mains qui s'entrechoquent pour le nom **CLAPS** ainsi qu'un petit robot analyseur pour symboliser la partie intelligence artificielle du projet.
- Un bouton pour importer depuis les fichiers de l'ordinateur une image à faire traiter par notre **OCR**. Celle-ci peut être modifiée à tout moment.
- Un curseur pour ajuster la rotation manuelle de l'image originale entre -90 et +90 degrés. Un bouton est également présent pour réinitialiser la valeur du curseur à 0.
- Un bouton pour lancer le traitement de l'image. Celui-ci fait disparaître la section de rotation ainsi que sa propre section.
- Une série de boutons alternatifs pour sélectionner la visualisation de l'étape actuelle du traitement. Cela modifie dynamiquement l'image affichée à droite.
- Un bouton pour exporter l'image dans le format souhaité, à un emplacement sélectionné par l'utilisateur.

Dans la partie droite, on retrouve une image qui est modifiée en temps réel en fonction des actions de l'utilisateur. Par exemple, au lancement, il n'y a pas d'image, pendant la rotation, elle affiche l'image tournée, et lors de l'appui sur le bouton de traitement, elle affiche directement la solution du mot caché.

Projet Word Search OCR

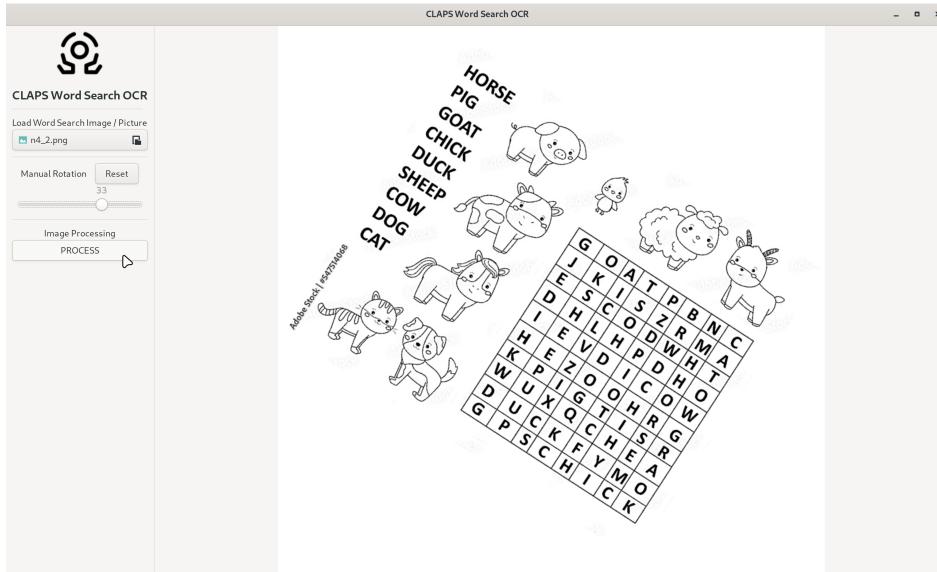


FIGURE 12 – Interface avant le processus de traitement

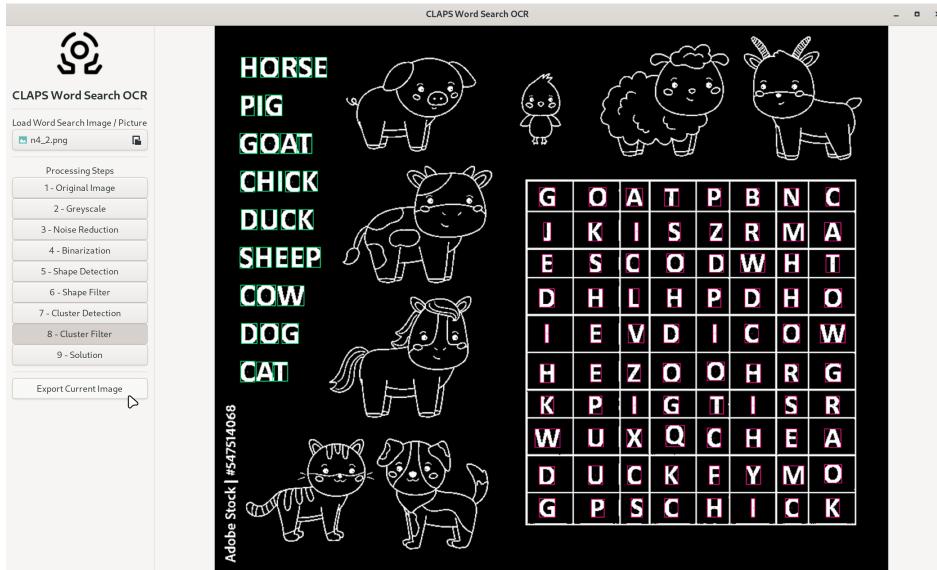


FIGURE 13 – Interface après le processus de traitement

12.2 ... Au concret

La deuxième étape a été de créer la liaison entre l'interface visuelle réalisée avec Glade et les fonctions codées dans le reste du projet, par l'intermédiaire de GTK3. GTK est par nature difficile à appréhender, car il est composé de dizaines de fonctions indépendantes du langage C.

Une fois la liaison créée, il a fallu faire fonctionner l'ensemble des éléments de l'interface.

L'image doit être facilement modifiable et doit conserver une taille adaptée à la hauteur de l'écran. Cela a été rendu possible grâce à **ImageMagick** et à son argument **-resize**, ainsi qu'à une implémentation dynamique de l'image.

Lors de l'importation d'une image, celle-ci doit être dupliquée et convertie en **.bmp** en deux exemplaires : l'un pour conserver la photo originale en cas d'exportation, et l'autre pour appliquer de potentielles rotations. Lors de l'appui sur le bouton de traitement, l'image doit être créée en 16 exemplaires différents : 8 pour l'exportation (l'image de l'étape X du traitement) en taille originale, et 8 autres pour l'affichage de l'image en taille redimensionnée, permettant ainsi de basculer rapidement entre les différents états une fois le traitement effectué, afin d'offrir l'expérience la plus fluide possible. Le traitement doit également créer un dossier d'images stockées sous le format BMP par cluster final, pour alimenter le réseau de neurones.

Le curseur de rotation, lors du relâchement de la souris après un appui, met à jour l'image stockée en lui appliquant la rotation du degré désiré. Lors de l'appui sur le bouton de réinitialisation, la valeur du curseur est remise à 0, et l'image est remplacée par l'image originale.

Le bouton de traitement masque la section de rotation ainsi que sa propre section, pour laisser place aux boutons d'étapes et à la section d'exportation. Cependant, il conserve le bouton d'importation, qui reste accessible à tout moment pour changer l'image d'entrée, ce qui a pour effet de réinitialiser la visualisation des boutons.

Lorsqu'un bouton de sélection d'étape est activé, il met à jour l'image à droite et désélectionne le bouton actuellement activé, agissant ainsi comme un bouton alternatif parmi les autres boutons de sélection d'étapes.

Lors de l'appui sur le bouton d'exportation, une fenêtre de dialogue s'ouvre et permet de choisir un emplacement sur l'ordinateur pour sauvegarder l'image actuellement affichée dans son format original. Le nom par défaut est **solution.png**, mais il peut être modifié, ainsi que le format, grâce à l'outil **ImageMagick**, qui convertit automatiquement l'image dans le format souhaité en fonction de l'état actuel du traitement (**CurrentState**).

À la fermeture du programme ou lors de l'importation d'une nouvelle image, le logiciel supprime toutes les images temporaires présentes dans **/output/** ainsi que toutes les images de lettres présentes dans les sous-dossiers des clusters. Il termine enfin en fermant proprement l'interface GTK.

13 Conclusion

Afin de clôturer ce premier rapport de soutenance, nous pouvons être fiers d'avoir pu réaliser l'entièreté de nos objectifs :

- Chargement d'une image et suppression des couleurs
- Rotation manuelle de l'image
- Détection de la position :
 - De la grille
 - De la liste de mots
 - Des lettres dans la grille
 - Des mots de la liste
 - Des lettres dans les mots de la liste
- Découpage de l'image (sauvegarde de chaque lettre sous la forme d'une image)
- Implémentation de l'algorithme de résolution d'une grille de mots cachés (solver)
- Une preuve de concept de votre réseau de neurones (XOR)

Ayant toutes ces fonctionnalités terminées pour la première soutenance, cela nous encourage à poursuivre dans cette voie, démontrant une bonne organisation au sein du groupe.