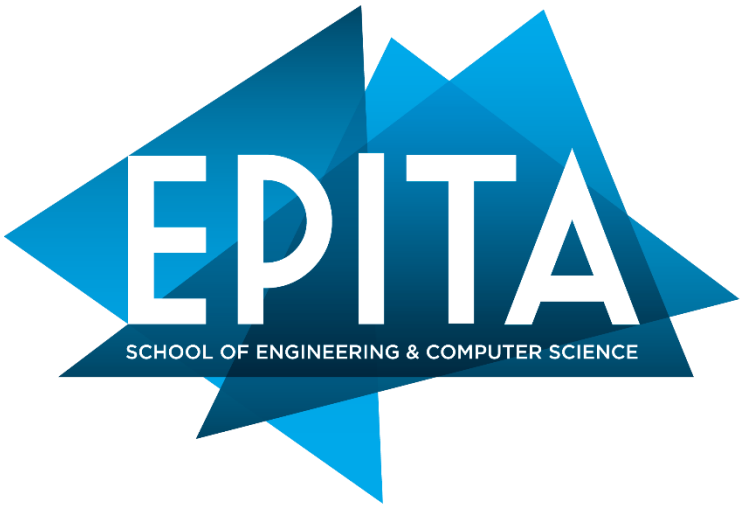
EPITA – PROJET OCR 2024



ShinyDuck's Wings

Rapport de première soutenance

- BONIS Dodie – KLEIDER Victor – SOLER Thomas – ZHOU Laurent -

04/11/2024

Sommaire

1. Introduction

1.1. Présentation du projet

L'objectif du projet est de réaliser un OCR (Optical Character Recognition) qui permet de résoudre une grille de mots cachés.

Il se décompose en plusieurs étapes, le chargement et le traitement d'une image, la détection de la grille de lettres et de la liste de mots, un réseau de neurones permettant de reconnaître les différentes lettres et enfin un solver pour résoudre la grille de mots cachés.

1.2. Présentation de l'équipe

Pour commencer, nous allons faire une courte présentation de l'équipe de ShinyDuck's Wings, composée de 4 membres.

Dodie Bonis s'est intéressée au monde de l'informatique depuis son plus jeune âge. Elle se passionne pour les jeux-vidéos et les puzzles en tout genre, ce pourquoi elle a choisi de s'occuper de la partie solveur de ce projet. De plus, grâce à ce projet, elle a pû en apprendre plus sur la gestion d'images et le fonctionnement des intelligences artificielles puisqu'elle s'est aussi occupée de la détection des lettres et de la grille.

Laurent Zhou a choisi de s'investir dans le domaine de l'intelligence artificielle car il trouve que ce domaine ouvre des perspectives fascinantes pour créer des solutions aux problèmes complexes du quotidien. En travaillant sur ce type de projet, il espère approfondir ses compétences en intelligence artificielle, notamment sur les réseaux de neurones.

Thomas Soler est le chef de projet, c'est à lui que revient la tâche d'orchestrer ce travail pour tirer le meilleur des compétences de chacun et pour produire un logiciel de qualité. Passionné par les nouvelles technologies, il n'est pas indifférent au domaine de l'intelligence artificielle qui l'a toujours intéressé mais aussi au traitement d'images. Ce projet lui permet de se plonger dans le monde des réseaux de neurones et de la topologie pour en découvrir plus sur ces notions fondatrices.

Victor Kleider, ayant rejoint l'équipe ShinyDuck's Wings récemment, est allé récupérer du travail pour alléger la charge de travail de ses camarades. Pour cela, il a été déduit qu'il s'occuperait de la rotation de l'image pour cette première soutenance.

1.3. Répartition de charges et état d'avancement

Repartition des taches - Soutenance 1				
	Dodie	Laurent	Thomas	Victor
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)	■		■	
Reseaux de neurones			■	
POC			■	
Final			■	
Solver	■			
Redaction du rapport	■	■	■	■

Fig1. Tableau de répartition des tâches

Avancements - Soutenance 1		
	Prevu	Reel
Chargement d'une image et suppression des couleurs	80%	70%
Rotation manuelle de l'image	100%	100%
Détection de la position	100%	95%
De la grille	100%	95%
De la liste de mots	100%	95%
Des lettres dans la grille	100%	95%
Des mots de la liste	100%	95%
Des lettres dans les mots de la liste	100%	90%
Découpage de l'image (sauvegarde de chaque lettre sous la forme d'une image)	100%	93%
Reseaux de neurones	50%	75%
POC	100%	100%
Final	0%	25%
Solver	100%	100%
Redaction du rapport	100%	100%

Fig2. Tableau d'avancements

2. Traitement des images

2.1. Rotation de l'image

Afin de simplifier la procédure d'analyse de l'IA, il nous faut réussir à tourner une image sur elle-même. Tout d'abord, nous avons élaboré une liste des points qui pourraient poser un problème lors de la rotation. Cette liste ainsi que la manière dont nous avons réglé ces problèmes se trouvent ci-dessous :

- Tourner l'image dans le bon sens.

Pour pallier ce problème, le programme de rotation accepte les nombres négatifs et l'image tourne dans le sens horaire pour un angle positif et anti-horaire pour un nombre négatif. De plus, avant de tourner l'image, nous avons utilisé la fonction IMG_LoadTexture afin de charger l'image, ainsi que SDL_QueryTexture pour récupérer les dimensions de cette image.

- Avoir une image centrée.

Cela n'a finalement pas posé de problème. En effet, il suffit de récupérer les dimensions de l'image et de les diviser par deux.

- Sauvegarder l'image.

Ici, la solution a été de créer simplement un duplicata de l'image concernée en l'enregistrant sous format ".png".

- Avoir une image qui ne dépasse pas une fois tournée.

La solution pour ce point s'est avérée être plus simple que prévue, puisqu'il suffisait de calculer la nouvelle hauteur et largeur en fonction de l'angle de rotation grâce à la librairie math.

- Tourner l'image et la mettre dans une fenêtre SDL.

Pour cela nous avons dû faire deux prototypes. Le premier calculait les nouvelles coordonnées de chaque pixel et les mettait un par un dans la nouvelle image. Malheureusement, ce prototype n'était pas viable car l'exécution était longue du fait des arrondis crées par la quantité de multiplication. Ainsi, on constatait des décalages et cela créait du bruit. Nous avons donc opté pour une seconde option qui consiste à charger l'image en tant que texture et l'insérer sur une nouvelle fenêtre SDL, puis de l'insérer à l'angle désiré. Cela a posé des problèmes de distorsion d'image et des décalages mais ces nouveaux problèmes étaient plus simples à gérer et à corriger.

Une fois tous ces points réglés, nous avons terminé la rotation.

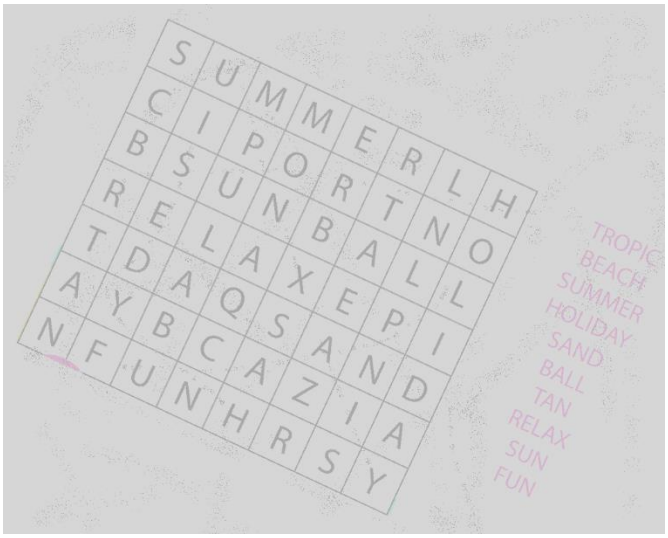


Fig3. Image avant rotation

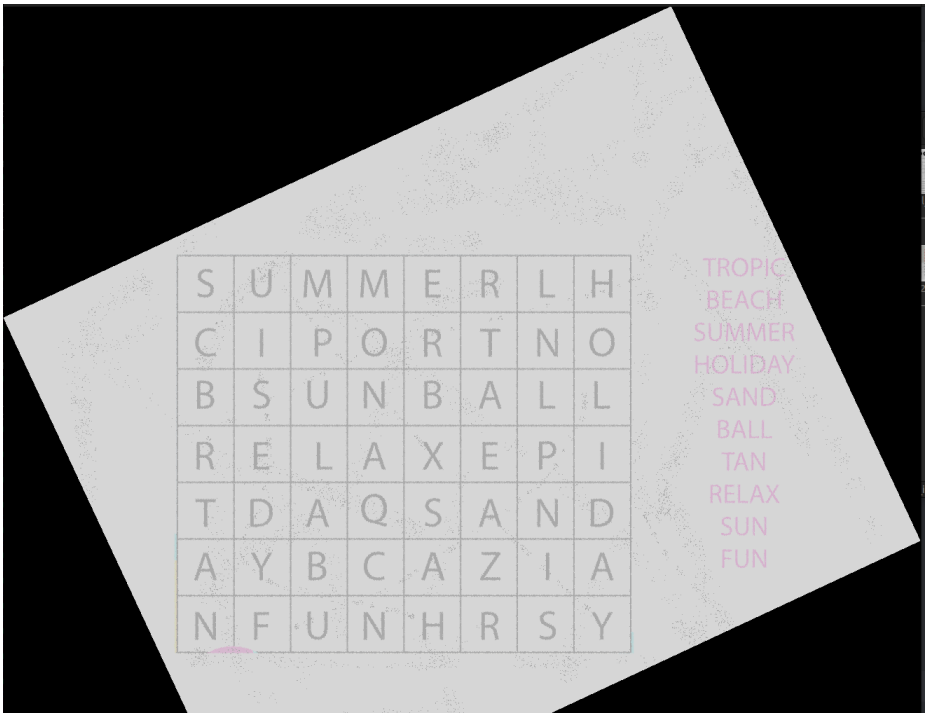


Fig4. Image après rotation de -25 degrés

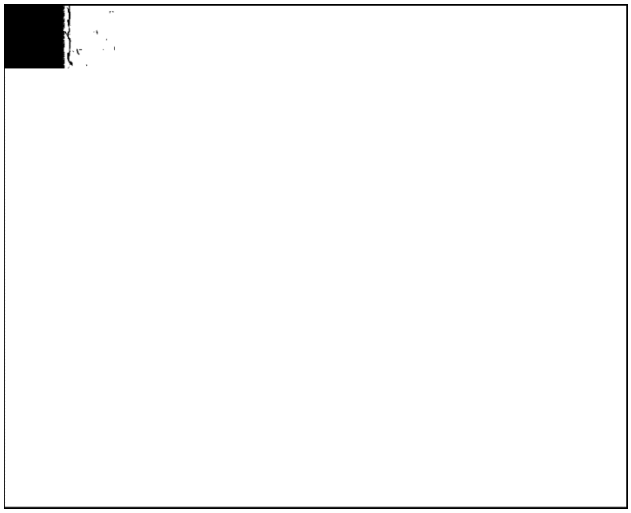


Fig5. Distorsion de l'image dû à un problème de centrage

2.2. Réduction du bruit

L'étape d'après dans l'image processing est la réduction de bruit. Pour réduire le bruit d'une image, il est possible d'employer un filtre gaussien ou un filtre médian.

Premièrement, le filtre gaussien permet généralement de réduire les bruits gaussiens, qui sont des variations aléatoires de la luminosité ou de la couleur d'une image. Il est également efficace lorsque le bruit est à basse fréquence ou réparti de manière homogène sur l'image. Ce filtre consiste à faire une moyenne entre le pixel qu'on est en train de traiter et les pixels autours, puis donner au pixel en cours de traitement cette valeur moyenne. Puis, en parcourant tous les pixels d'une image, cette image va être floutée ce qui va enlever toutes les tâches. Cependant, flouter l'image risque de mener à une perte de précision et l'effacement de bords qui sont importants pour la reconnaissance des lettres.

Deuxièmement, le filtre médian permet de préserver les contours tout en supprimant les bruits à haute fréquence tels que les bruits de type « poivre et sel » donc ce filtre fonctionne bien avec les pics de bruit isolés. Ce filtre consiste à remplacer chaque pixel avec le médian calculé en fonction du pixel et des pixels autours. Le désavantage de ce filtre est qu'il n'est pas très efficace contre les bruits larges, légers et homogènes.

Afin de faire le filtre gaussien ou médian, il faut quatre boucles encastrées. Les deux premières boucles permettent de parcourir tous les pixels de l'image, et les deux autres permettent de parcourir les pixels autours pour faire une moyenne ou médiane. Pour faire la moyenne, il suffit de cumuler les valeurs dans une variable puis de diviser par le nombre de pixels utilisés autour. Pour faire la médiane, il faut stocker toutes les valeurs dans un tableau, puis ranger les valeurs du tableau dans l'ordre croissant ou décroissant afin de récupérer la médiane qui est la valeur au milieu.

En appliquant le filtre gaussien et le filtre médian sur la même image, on est sûrs de se débarrasser de la plupart des bruits. Ainsi, en appliquant ces deux filtres sur la Figure 6, on obtient la Figure 7 (images ci-dessous).

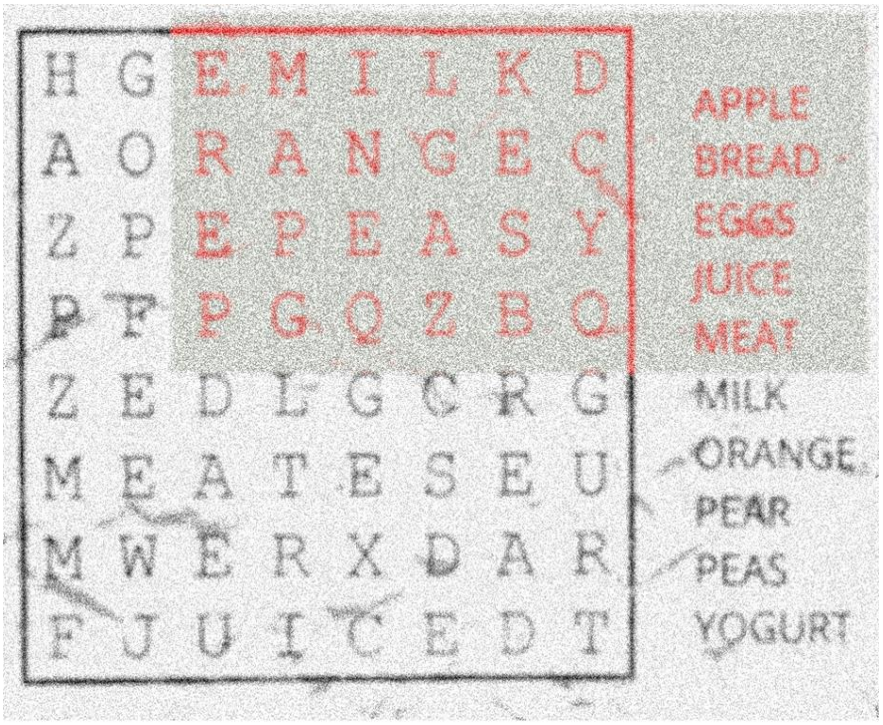


Fig6. Image avant filtrage

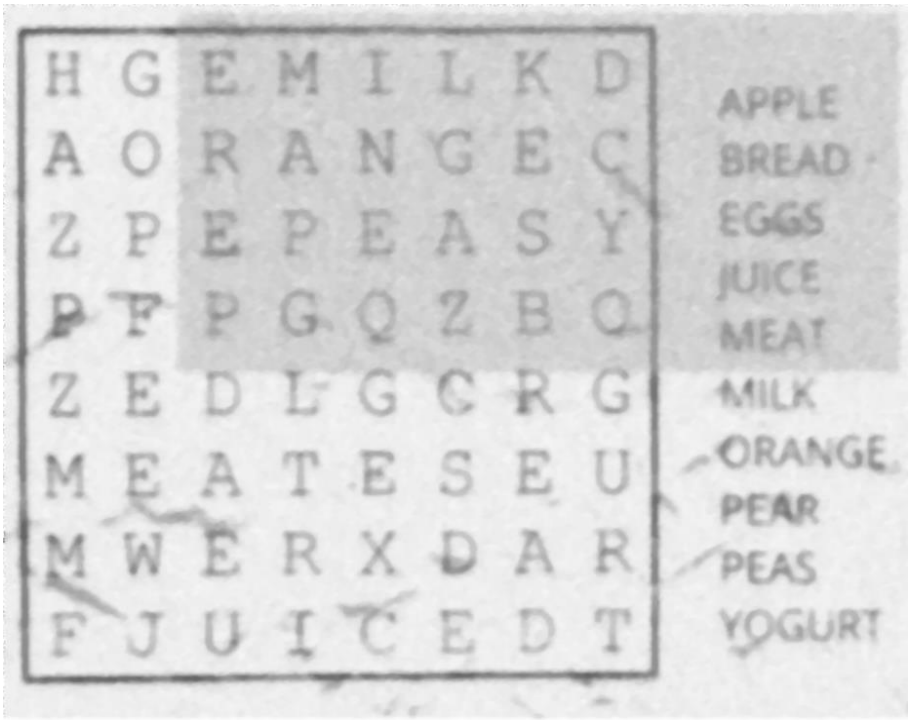


Fig7. Image après filtrage

2.3. Noir et blanc

On cherche ici à passer l'image complètement en noir et blanc. Pour cela, nous avons utilisé un seuil. Il est possible de trouver ce seuil en faisant une moyenne entre la valeur maximale et minimale des pixels d'une image. Ce seuil va ensuite être utilisé pour faire passer les pixels en dessous de ce seuil en noir et le reste en blanc. Cependant, cet algorithme peut ne pas marcher dans certains cas. Si la couleur de l'arrière-plan change ou si les lettres sont très claires, le résultat risque de ne pas être ce qu'on désire, ce qui est le cas pour le résultat ci-dessous.

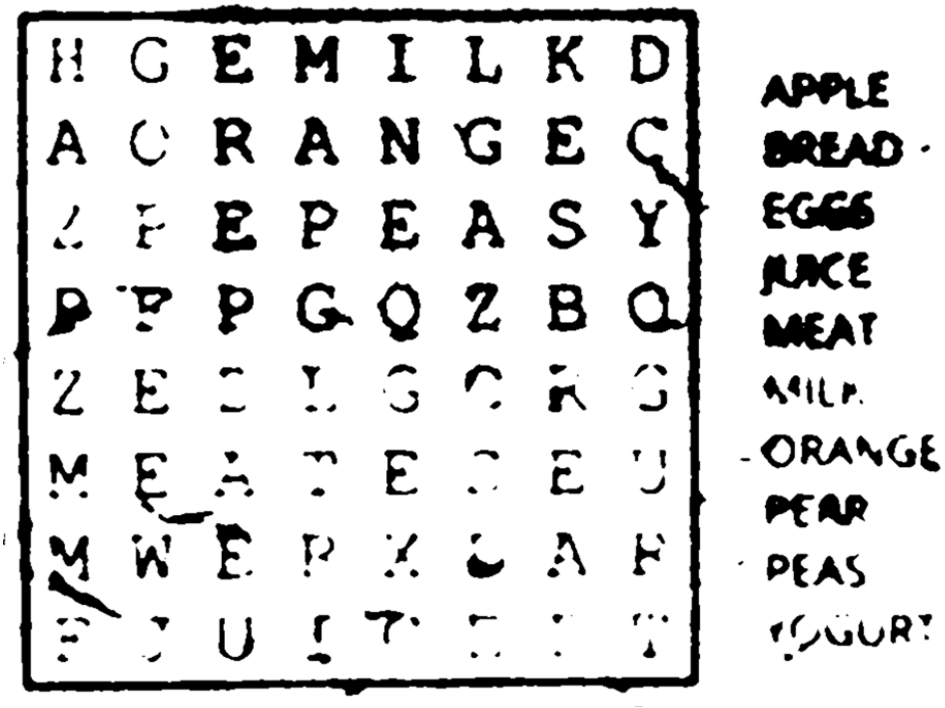


Fig8. Image après passage en noir et blanc grâce à un seuil

Ainsi, une meilleure solution serait de détecter les bords avec le masque de Sobel. Ce masque consiste à détecter séparément les lignes horizontales et les lignes verticales. Afin de détecter les lignes horizontales, il faut parcourir tous les pixels et pour chaque pixel comparer les trois pixels du haut et les trois pixels du bas et stocker chaque nouvelle valeur calculée par cette comparaison dans une matrice de la même taille que l'image analysée. De même, pour détecter les lignes verticales, il faut pour chaque pixel comparer les trois pixels à gauche et les trois pixels à droite et stocker chaque nouvelle valeur calculée dans une deuxième matrice similaire à la première. Une fois les deux matrices obtenues, on doit créer en créer une troisième de la même taille que l'image analysée, qui stockera les magnitudes.

Pour calculer les magnitudes, on prend les deux valeurs depuis la première matrice et la deuxième matrice qui sont à la même position, on les met au carré, on les additionne et on met le tout à la racine. Une fois la magnitude calculée, on la place dans la nouvelle

matrice à la même position que les deux premiers. À l'aide de cette nouvelle matrice, on peut calculer un seuil en faisant une moyenne de toutes les magnitudes. En effet, si une magnitude est au-dessus de ce seuil, alors il y a un rebord dans l'image à la même position que la magnitude par rapport à la matrice des magnitudes. En passant les bords en noir et le reste en blanc, on obtient l'image ci-dessous.

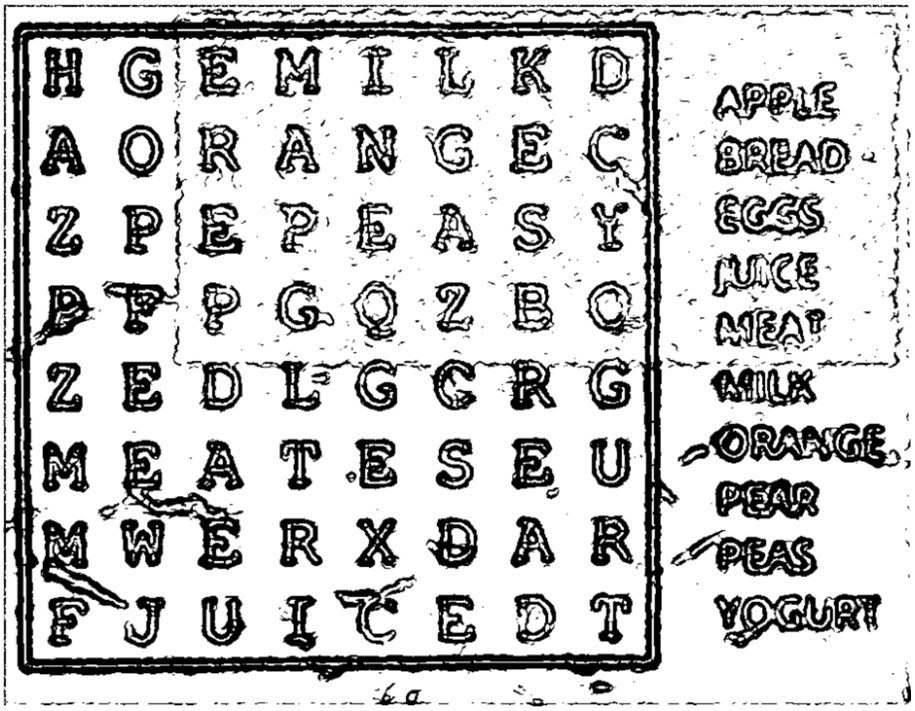


Fig9. Image après masque de Sobel

Tout comme la première méthode, le masque de Sobel présente des désavantages. Un désavantage assez flagrant est que cet algorithme est très sensible au bruit. Même après avoir filtré l'image, on peut remarquer qu'il y a toujours des tâches.

3. Détections et découpage

3.1. Détection des lettres

Pour pouvoir venir à bout de grilles de mots mêlés, il faut bien évidemment que notre programme trouve la position des lettres. Pour cela nous avons utilisé une technique qui part d'une logique assez simple. Ainsi, nous allons récolter toutes les composantes connexes de l'image.
Tout d'abord, qu'est-ce qu'une composante connexe ? C'est un terme topologique dont la définition peut être vulgarisée en : "un seul morceau non interrompu de pixels noirs distincts". Cela est donc très utile puisque les lettres sont des composantes connexes.

La première idée que nous avons eue pour détecter et stocker ces composantes connexes était de créer une struct "compcon" (composante connexe) contenant la liste de la position en x des pixels appartenant à cette composante connexe et la liste de leur position en y. Puis, on pouvait stocker toutes les composantes connexes dans une liste de ces structs. Cependant, pour accéder aux composantes en x et en y à l'intérieur d'une struct elle-même à l'intérieur d'une liste, cette solution n'était pas pratique.

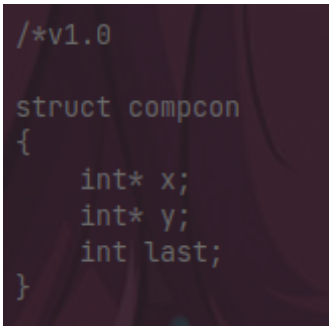


Fig10. "Struct compcon"

Finalement, l'idée que nous avons adoptée s'est révélée être bien plus simple à implémenter. Cette nouvelle solution consiste à créer une nouvelle matrice d'entiers de la même taille que l'image en pixels. Ensuite, on initialise chaque case de la matrice en 0 si le pixel correspondant est blanc. Pour les pixels noirs (donc ceux qui ne sont pas encore attribués dans la matrice), on attribue un entier différent à chaque composante connexe grâce à un algorithme récursif. Nous avons donc trouvé toutes les composantes connexes de l'image et on peut les différencier par leur "id", c'est-à-dire l'entier qui leur a été associé dans la matrice.

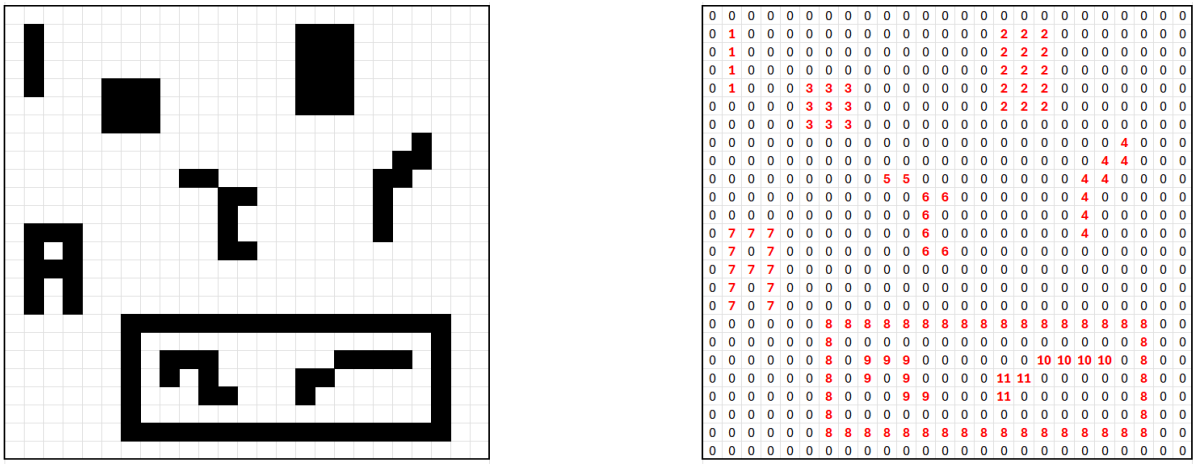


Fig11. Exemple visuel de détection des composantes connexes

Maintenant que nous avons cette matrice, il faut séparer ce qui est une lettre de ce qui n'en est pas. Pour cela, on va supprimer des composantes connexes de notre matrice selon 2 critères : leur position et leur taille. Si le pixel est en bordure de l'image, on considère que c'est du texte superficiel. Ensuite, on parcourt la matrice afin de calculer la moyenne de la taille des composantes connexes. Puis, on élimine les composantes connexes qui ont une taille de moins d'un quart de la moyenne et celles qui ont une taille de plus de 6 fois la moyenne (ces valeurs ont été trouvées par l'expérimentation). Les composantes connexes restantes sont alors principalement des lettres. Il reste encore quelques déchets mais ceux-ci ne nous poseront pas trop de problèmes.
Une fois les lettres trouvées, on va les délimiter dans un cadre afin de pouvoir manipuler leur position assez facilement. Pour cela, on utilise une struct "letter" qui enregistre ses positions importantes.

```
struct letter
{
    int xmid;
    int ymid;
    int xmin;
    int ymin;
    int xmax;
    int ymax;
};
```

Fig12. "Struct letter"

3.2. Détection de la grille

Une fois la liste de toutes les lettres et leur position trouvées, on peut maintenant trouver la position et les dimensions de la grille. Pour cela, on itère doublement à travers la liste des struct letter. C'est-à-dire qu'on lance une boucle for qui itère à travers toute la liste puis une autre boucle for qui itère de l'index actuel de la première boucle à la fin de la

liste. Le programme va alors agir comme si les deux lettres correspondant aux deux index actuels sont les coins de la grille.

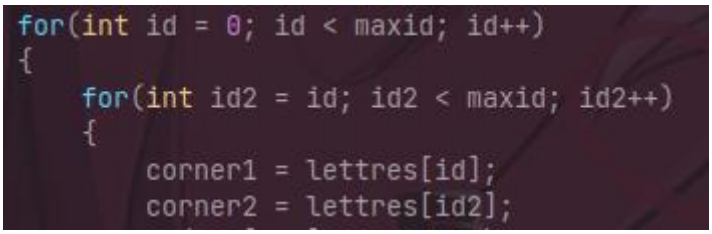


Fig13. Bout de code de détection de la grille

En partant de ce postulat, l'algorithme va alors effectuer une série de tests pour confirmer son hypothèse ou non. Dans un premier temps, il va parcourir l'image horizontalement en commençant au x du premier coin de la grille jusqu'au x du deuxième coin. Il compte le nombre de composantes connexes trouvées dans ce parcours et admet que c'est donc le nombre de lettres présentes dans la grille horizontalement. Pour vérifier que les deux lettres actuelles forment bien une grille, il va alors itérer en y dans l'image et vérifier que le nombre de composantes connexes trouvées en itérant à nouveau horizontalement est bien le même que le compte de la première ligne.
Après avoir validé que les deux lettres forment bien une grille, il faut maintenant qu'il décide si elles forment bien la plus grande grille possible. Pour cela, lorsque l'algorithme trouve une grille, il fait le produit de sa taille horizontale par sa taille verticale et compare ce produit à celui de la dernière grille trouvée. Si celui-ci est plus grand, alors il remplace l'ancienne grille trouvée par la nouvelle. En sortant de ce programme, on a donc bien la position et la taille de la grille la plus grande.

3.3. Détection de la liste de mots

Une fois la grille trouvée et ses données sauvegardées, nous pouvons supprimer toutes les composantes connexes qui en font partie pour nous concentrer sur la liste de mots.
En effet, sans la grille, les seules composantes restantes sont les quelques déchets restants et les lettres des mots de la liste.

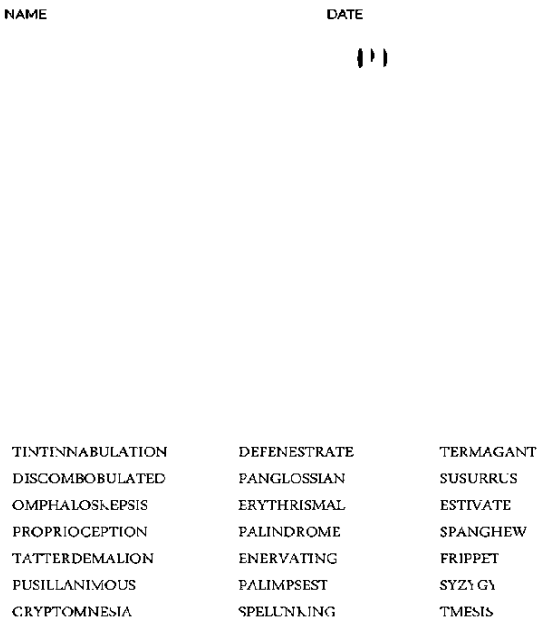


Fig14. Exemple de l'image obtenu après suppression de la grille et de déchets

Dans un premier temps, nous avons à nouveau filtré l'image en ne gardant que les composantes ayant une distance avec la composante la plus proche inférieure à un seuil trouvé expérimentalement. La nouvelle image obtenue est donc encore plus propre que la précédente mais il reste encore de légers déchets à enlever.
Pour cela, nous avons choisi d'utiliser un pan de la topologie nommé morphologie mathématique et plus précisément ses opérandes élémentaires qui sont la dilatation et l'érosion.

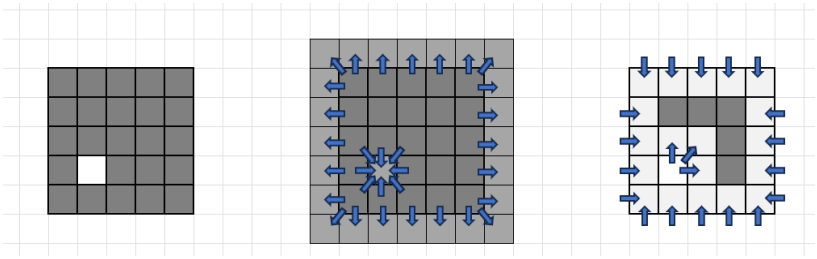


Fig15. Schéma de la dilatation et de l'érosion de la matrice de pixels la plus à droite

Après avoir testé plusieurs combinaisons de ces opérandes, il était clair que la meilleure solution pour relier les lettres des mots était de réaliser 4 dilatations successives en x. Ce faisant, l'image obtenue est composée de déchets dilatés et de blocs qui sont les mots dont les lettres se sont reliées. A partir de là, il suffit de vérifier si chaque supposé mot est à une distance de moins de 25 pixels en y (distance trouvée expérimentalement), et si c'est le cas on le considère vraiment comme un mot et on le place dans un tableau de struct "word" défini, comme pour une lettre, par ses coordonnées xmid, ymid, xmin, ymin, xmax et ymax. Mais cette struct possède deux composantes en plus : le nombre de lettres par mots et le tableau des "id" de chacune des lettres composants le mot. A partir de ce tableau on peut donc retrouver chacune des lettres d'un mot.

3.4. Sauvegarde en fichier image

Pour faire en sorte que l'IA puisse trouver quelle lettre correspond à quelle composante connexe, il faut lui envoyer des petites images découpées de l'image principale représentant les différentes lettres. Pour cela, notre struct letter déclarée précédemment est très pratique et fait déjà presque tout le travail pour nous. En effet, elle pose déjà un cadre autour de chaque lettre. Il suffit alors de créer une nouvelle surface de la taille qu'on veut, puis d'utiliser SDL_BlitSurface pour copier une partie de l'image sur une nouvelle surface, puis de changer sa taille (notre IA ayant besoin d'images de 28 pixels par 28 pixels) et de l'enregistrer avec SDL_SaveJPG.

Fig16. Lettre 'N' découpée en 28x28

On enregistre chacune des lettres avec son id (son numéro de composante connexe) dans le nom de l'image créée afin de pouvoir replacer les lettres déterminées par l'IA à la bonne place.

Fig17. Images de lettres découpées

4. Réseaux de neurones

4.1. Réseau non XOR

La première étape de la création de notre réseau de neurones était d'en comprendre le fonctionnement. Pour cela, nous avons commencé par créer un simple réseau permettant de faire l'opération non XOR.

Pour se rapprocher du réseau de neurones qui reconnaîtra des lettres à partir d'un tableau de pixels, nous allons utiliser deux "hidden layers" (couches cachées) pour augmenter la précision de notre réseau. En voilà un schéma :

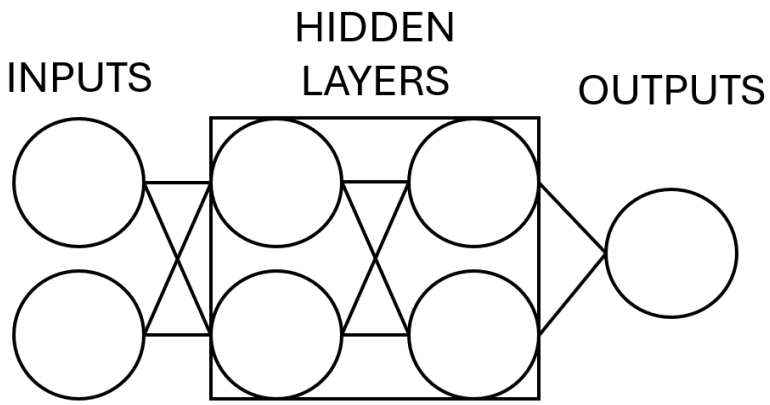


Fig18. Schéma d'un réseau de neurones avec des hidden layers

Les INPUTS sont les éléments d'entrées, qui sont pour un XAND au nombre de deux et peuvent être 0/0, 0/1, 1/0 et 1/1. Chaque input est relié à chaque neurone de la couche cachée, cette liaison permettant d'effectuer une opération. La valeur obtenue dans chaque neurone est donc $i_1w_1 + i_2w_2 + b$ avec i les inputs, w les poids de chaque inputs et b le biais. Cette opération se répète pour la deuxième hidden layer où les inputs seront ici les valeurs calculées par la première hidden layer. Enfin, même principe pour l'output (unique car l'opération XAND renvoie soit 0 soit 1).

Mais pour qu'un tel réseau fonctionne, il faut l'entraîner. Pour cela, nous allons traiter le réseau en remontant depuis l'output vers les inputs et répéter cet aller-retour un grand nombre de fois.

Dans un premier temps, nous calculons l'erreur, qui est la différence entre le résultat attendu et celui vraiment obtenu, ce qui nous permet d'obtenir une correction pour les poids et les biais de l'output. On remonte ensuite dans la deuxième hidden layer où on calcule la correction des poids et des biais à partir de la correction de l'output et enfin dans la première hidden layer où on calcule de la même manière la correction. Enfin, on applique ces corrections pour que le résultat se rapproche de plus en plus de la réalité.

Une fois les biais et les poids calculés, nous les sauvegardons dans un fichier pour pouvoir les réutiliser et éviter de les recalculer à chaque utilisation. Ceci n'impacte pas beaucoup l'efficacité d'un réseau tel qu'un non XOR car il a peu de neurones mais pour des réseaux plus développés comme le réseau qui permet d'analyser des images de lettres, il y aura un impact.

4.2. Réseau de lettres

Pour réaliser notre réseau de neurones permettant de trouver une lettre, nous devons un peu revoir la structure de notre réseau de neurones.

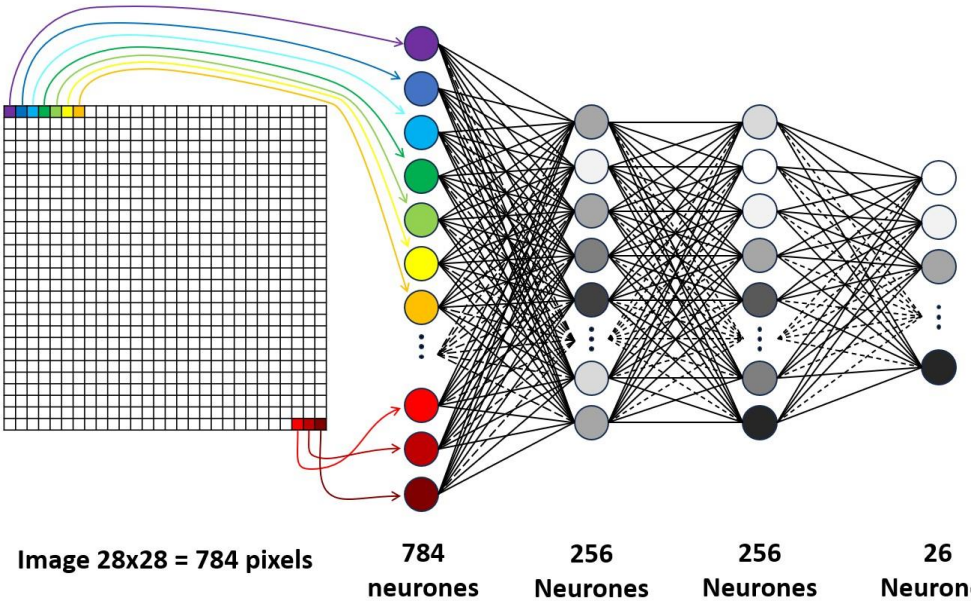


Fig19. Schéma de la structure du réseau de neurone

Ici nous avons besoin de beaucoup d'inputs qui représenteront tous les pixels d'une image de 28 par 28 pixels soit 784 pixels et donc 784 inputs. Pour chaque hidden layer il y aura 256 neurones et enfin 26 outputs pour représenter les 26 lettres de l'alphabet.

Le concept est le même que pour le non XOR, on entraîne d'abord le réseau avec des inputs dont on connaît l'output.

Nous avons donc choisi de créer des images d'entrainement représentant des lettres qui sont récupérées par un sous-programme et converties en tableau de 784 cases qui sont soit à 0 soit à 1 en fonction de la couleur du pixel (0 pour blanc et 1 pour noir).

Ces tableaux sont ensuite donnés en entrée de notre réseau pour l'entraîner 9984 fois avec ces 26 lettres d'entraînement. Le nombre d'images d'entraînement est pour l'instant fixe mais sera amplifié à l'avenir pour pouvoir interpréter un plus grand nombre de lettres.

5. Solver et affichages

5.1. Solver

Lors de l'exécution du solver, nous avons déjà trouvé la grille et ses lettres (stockées dans une matrice représentative de la grille), ainsi que la liste de mots à trouver. La fonction solver en elle-même est en réalité un brute-force optimisé. En effet, la fonction prend un mot à trouver et parcourt toute la matrice. Lorsqu'elle tombe sur la première lettre du mot à trouver, la fonction fait un deuxième parcours (appelé "Parcours_2") dans la matrice dans les 8 directions possibles (haut-gauche, haut, haut-droite, droite, bas-droite, bas, bas-gauche, gauche) tant qu'il n'a pas atteint le bord de la matrice et que les lettres trouvées sont bien les lettres suivantes du mot à rechercher. Si le mot est trouvé durant ce parcours, on retourne sa position, sinon, on continue le parcours principal.

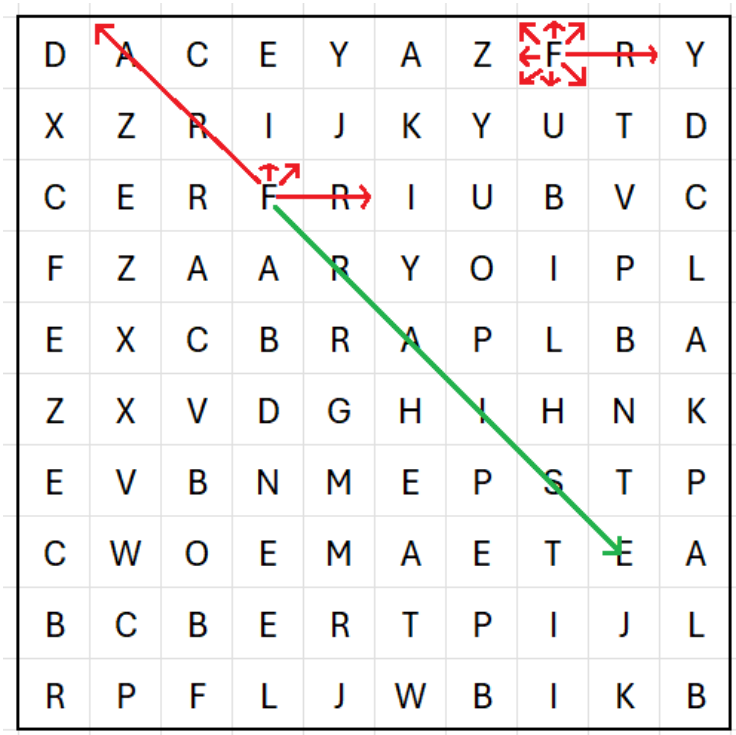


Fig20. Visualisation du "Parcours_2" sur le mot "FRAISE"

5.2. Affichage de la solution

Une fois qu'on a récupéré l'image, la position de la grille, et les coordonnées dans la grille de la première et de la dernière lettre du mot, il suffit de tracer un trait entre la coordonnée dans l'image de la première lettre du mot jusqu'à celle de la dernière lettre. Premièrement, pour trouver la position approximative en x (respectivement y) d'une lettre dans l'image notée "$XPos_{(lettre->image)}$" (respectivement $Ypos_{(lettre->image)}$), on utilise la formule ci-dessous.

$$XPos_{(lettre->image)} = XPos_{(grille->image)} + (XTaille_{(grille)} / NbLettresEnX_{(grille)})* XPos_{(lettre->grille)}$$

Fig21. Formule pour trouver les positions

Une fois les positions des deux lettres trouvées, on doit s'occuper de tracer un trait sur une surface. Pour cela, on utilise un algorithme de line drawing assez simple, mais chaque pixel de la ligne est directement remplacé sur la surface grâce à SDL_MapRGB. De plus, on remplace également tous les pixels autour afin d'avoir une ligne qui est épaisse de 3 pixels. On exécute alors le programme de line drawing sur tous les mots trouvés grâce au solver afin d'avoir une nouvelle surface avec tous les mots trouvés et barrés.

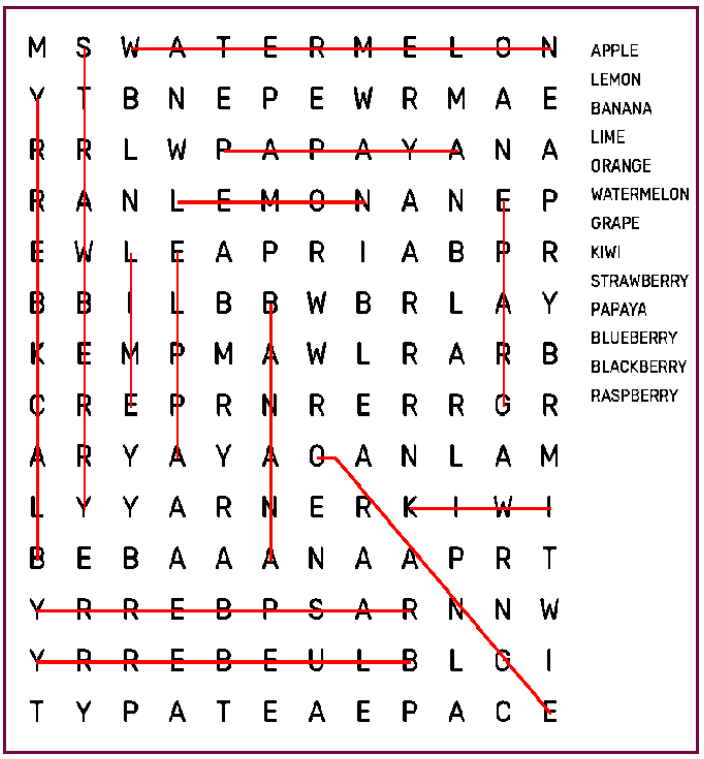


Fig22. Résultat de l'algorithme de line drawing

5.3. Premiers pas sur Glade

Afin de s'avancer sur la partie interface du projet, nous avons commencé à développer une interface à l'aide de Glade.

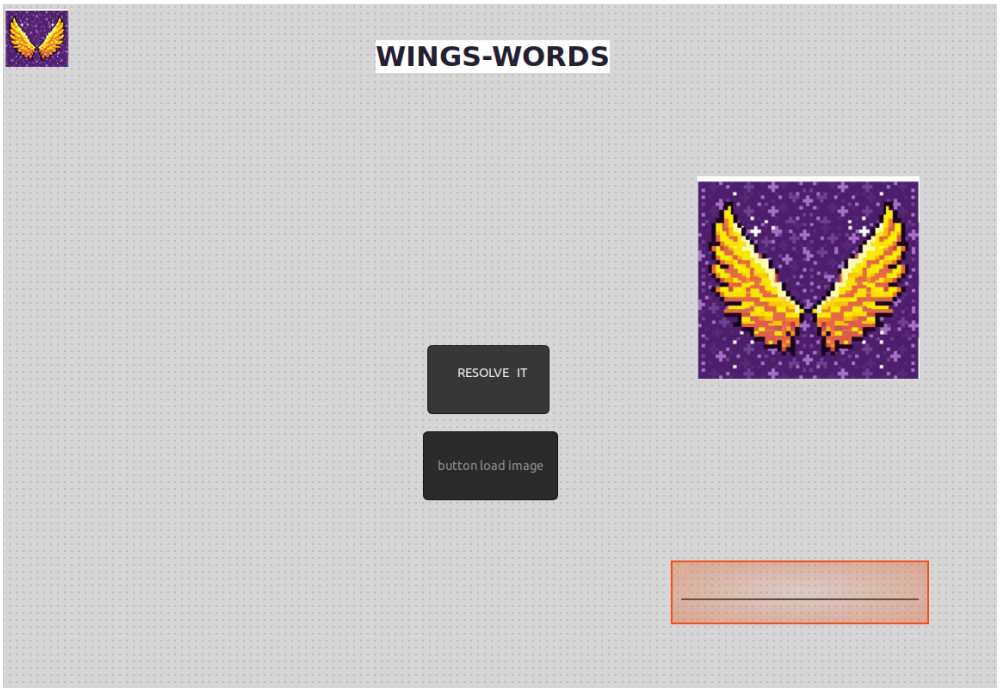


Fig23. Interface créée à partir de Glade

Cette interface n'est évidemment qu'un prototype. Celle-ci contient le logo de notre équipe, le nom de l'application (choisi comme étant Wings Words), deux boutons ainsi qu'un slider. Le bouton "Load image" ouvre une fenêtre de recherche de documents qui permet ensuite de charger une image. Le bouton "Solve" qui lancera la résolution de l'image une fois le projet fini. Enfin, le slider permet de faire tourner l'image sur elle-même si on le déplace vers la droite.

6. Conclusion

Pour conclure, toute la partie manipulation de l'image et détection des éléments est terminée. Afin de finir complètement ce projet, il nous faut maintenant achever la partie réseau de neurones pour identifier les lettres. De plus, il nous faut aussi lier tous nos fichiers pour pouvoir exécuter le programme en une fois. Mais assez parlé de technique. Nous ne sommes qu'à la moitié de ce projet et pourtant, tous nos membres ont vu leur aisance et leurs capacités en C augmenter durant cette courte période de temps.
En effet, l'équipe de ShinyDuck's Wings ainsi que ses membres ont évolué au cours de ces deux derniers mois et sont maintenant plus prêts que jamais pour achever ce projet. Malgré les problèmes rencontrés (d'organisation comme de programmation), l'équipe a su surmonter les obstacles afin de produire un résultat qui nous satisfait tous.