Optical Character Recognition

I) Introduction

Objectif du projet :

Notre objectif lors de ce projet était d'écrire un programme permettant à nos machines de reconnaître différents symboles manuscrits sans utiliser de librairie pré faites.

La base de données :

Pour ce projet, nous disposions d'une base de données comportant 120 images de symboles manuscrits fournis par nos professeurs. La base contient 10 images pour chaque symbole (+, -, chiffre), chacun écrit plus ou moins bien et avec des écritures différentes.

Enjeux scientifiques du projet :

- La difficulté majeure de ce projet résidera évidemment dans l'interdiction d'utiliser des librairies déjà écrites par autrui, nous obligeant à mettre au point nos propres algorithmes de manipulation ainsi que d'analyse d'images.
- Une autre difficulté sera de trouver un processus de traitement d'images qui nous permettra d'obtenir les meilleurs résultats possibles. En effet, il existe une multitude de façons de transformer une image (changement d'échelle, rotation, translation, érosion et dilatation, etc) afin de la rendre propice à son analyse. Cependant, il existe aussi de nombreux critères que nous pouvons obtenir de cette analyse et qui nous permettront de catégoriser chaque image. Il nous faudra trouver la meilleure combinaison de tous ces facteurs si nous voulons obtenir un algorithme efficace.

II) Notre solution

Vous trouverez ci-dessous l'explication en détails de la chaîne de traitement et d'analyse que nous avons mise au point, afin de répondre aux demandes de ce projet.

Étape 1 : Standardisation

Il est évident que toutes les images de notre base de données ne se ressemblent pas, autant pour leur taille, leurs détails ou leur calligraphie. Afin de pouvoir les comparer, nous devons nous assurer que chacune de ces images respectent bien certaines règles. Pour cela, nous faisons subir à nos images quatre transformations dans un ordre bien précis.

1.1) Mise à l'échelle

Nous avons donc choisi d'imposer à nos images une taille commune de 50px X 50px. Cela rendra certaines transformations et comparaisons radicalement plus simples par la suite.

1.2) Suppression du flou

Afin de travailler sur des images binaires (pixels noirs ou blancs), nous nous débarrassons des niveaux de gris. Pour cela, nous prenons les pixels de l'image et, si leur niveau de gris est supérieur à 127, on lui assigne 255 à la place. Inversement, si ce dernier est inférieur à 127, on lui assigne 0.

1.3) Érosion puis dilatation

Après ces transformations, nous nous retrouvons avec des images en noir ou blanc pouvant comporter du bruit selon l'écriture du symbole. Afin de s'en débarrasser, nous appliquons un algorithme d'érosion à notre image, puis de dilatation.

Nous avons choisi cet ordre dans le but de pouvoir supprimer certains pixels ou groupes de pixels solitaires, que l'on pouvait retrouver sur certains de nos résultats.

Etape 2 : Extraction de caractéristiques

Pour ce projet, nous avons choisi d'extraire trois caractéristiques afin de catégoriser nos images. Voici dans le détail leur algorithme d'extraction :

2.1) Profils horizontal et vertical

Le profil d'une image est simplement son nombre de pixels noirs sur chaque colonne (profil vertical) ou bien sur chaque ligne (profil horizontal).

Pour son algorithme d'extraction, il est très simple. On récupère les pixels de notre image sous forme de tableau de 1-dimension, que l'on va directement reformer en un tableau en 2-dimension, de même format que notre image.

Selon la direction demandée, nous parcourons chaque ligne ou colonne en comptant le nombre de pixels noirs rencontrés, et stockant ce nombre dans un tuple 'résultat' en arrivant au bout de notre ligne ou colonne.

Une fois l'image entièrement parcourue, on renvoie notre 'résultat'.

2.2) Zoning

Le zoning consiste à séparer une image en un certain nombre de zones dans une grille, et, pour chaque zone, déterminer le pourcentage de pixels noirs.

La taille de la grille est donnée en paramètre de la fonction. Par défaut, on crée une grille de 4*4 donc 16 zones.

On convertit l'image en binaire puis on définit la taille et les coordonnées de départ de la première zone.

On calcule ensuite le pourcentage de pixels noir dans cette zone, on l'arrondit, puis on stocke cette valeur dans un tuple.

On répète ces étapes jusqu'à avoir calculé toutes les zones de la grille et on finit par renvoyer ce tuple.

Etape 3 : Analyse, entrainement et test

Maintenant que nous avons nos caractéristiques pour nos images, il ne nous reste plus qu'à les analyser. Pour ce faire, nous avons choisi d'utiliser un algorithme KNN.

Voici une brève explication du fonctionnement de cet algorithme et de notre implémentation.

Premièrement, nous avons séparé notre jeu de données en 2 parties : 80% des images sont dans la partie *TRAIN* et 20% dans la partie *TEST*. Cela signifie que nous utilisons 96 des 120 images pour faire apprendre notre algorithme, et les 24 images restantes pour tester cet apprentissage.

L'algorithme KNN consistera ensuite à tester nos images de la partie *TEST* comme expliqué ensuite, afin de ne retenir que les k voisins les plus proches (k étant choisi par l'utilisateur). Ces k voisins nous donneront une idée de ce que notre image a le plus de chance de représenter.

Deuxièmement, nous avons implémenté notre propre méthode permettant de calculer la distance euclidienne entre des vecteurs. Elle nous sera très utile car les caractéristiques de nos images ne sont pas des valeurs simples, mais des tuples (~= vecteur). En calculant la distance euclidienne entre les caractéristiques de deux images, nous obtenons une valeur représentative du degré de similitudes entre ces deux images.

Troisièmement, pour chaque image de notre partie *TEST,* on compare ses caractéristiques (profils horizontal et vertical, zoning) à celles de toutes les images de notre partie *TRAIN*. De cette manière, nous obtenons, pour chaque image test, sa distance euclidienne à chaque image d'apprentissage.

Dernièrement, nous récupérons les k distances les plus basses pour chaque image test. En observant à quelles images d'apprentissage ces distances correspondent, on peut déduire quelles images possèdent les caractéristiques les plus proches, et donc quel symbole a le plus de chance d'être représenté sur l'image.

Résultats

On peut voir que la plupart des erreurs de notre algorithme sont au niveau des symboles plus ‘complexes’. Ces erreurs apparaissent principalement pour des symboles plutôt semblables (7 et 4, 4 et +)

Taux de réussite : 83.33% avec 20 positifs et 4 négatifs

	+	-	0	1	2	3	4	5	6	7	8	9
+	2	0	0	0	0	0	0	0	0	0	0	0
-	0	2	0	0	0	0	0	0	0	0	0	0
0	0	0	2	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	1	0	0	0
3	0	0	0	0	0	2	0	0	0	0	0	0
4	0	1	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	0	2	0	0	0	0
6	0	0	0	0	0	0	0	0	2	0	0	0
7	0	0	0	0	0	0	1	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	0	2	0
9	0	0	0	0	0	0	0	1	0	0	0	1

III) Conclusion

En moyenne, les résultats obtenus par notre algorithme environnent les 80-85% de reconnaissance pour une base d’apprentissage de 96 images et une de test de 24, ainsi qu’en comparant 3 caractéristiques.

Afin d’améliorer notre algorithme, nous pourrions premièrement élargir nos bases, afin d’obtenir des résultats plus représentatifs et de confronter notre solution à des données encore plus diverses. Une fois cela fait, nous pourrions également affiner notre partition des bases, 80/20 n’étant pas forcément la meilleure sur une plus large échelle.

Une deuxième piste d’amélioration serait d’extraire plus de caractéristiques à nos images, nous permettant de comparer encore plus d’aspects de nos images.