

Compte rendu du travail fait sur le projet : Semaine 5

Corentin :	2
Jami :	3
Segmentation sans traitement :	4
Segmentation avec threshold simple basé sur la moyenne des nuances de gris de l'image :	4
Avec un traitement Threshold adaptatif :	5
Segmentation avec un traitement HSV simple en plus du threshold simple précédent :	7
Avec un traitement HSV simple en plus du threshold OTSU :	8
Arno :	10
Julien :	12
Eliaz :	15

Corentin :

Difference rgb / hsv :

image :



hsv :



image gray :



hsv gray :



threshold :



threshold :



th adaptatif :



th adaptatif :

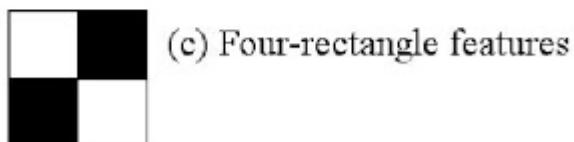
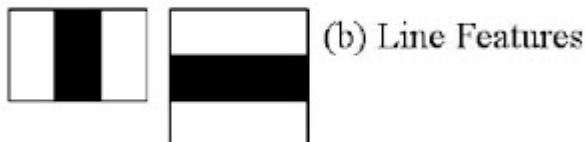
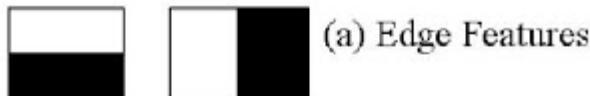


Haar Cascade OpenCV :

Un classificateur Haar cascade a besoin d'images "positives", ce sont des images contenant principalement l'objet que l'on veut trouver. (soit des images contenant principalement l'objet, soit des images contenant l'objet seul) et des images "négatives" qui peuvent être des images d'arrière-plan qui ne contiennent pas l'objet qu'on recherche.
Un classificateur Haar Cascade est formé en superposant l'image positive sur un ensemble d'images négatives.

Les caractéristiques sont obtenues en soustrayant la somme des pixels de l'image couverts par la zone blanche du filtre à la somme des pixels couverts par la zone noir. Ce processus génère énormément de caractéristiques étant donné que le filtre est appliqué sur toute l'image.

Une caractéristique de Haar est essentiellement des calculs effectués sur des régions rectangulaires adjacentes à un emplacement spécifique dans une fenêtre de détection. Le calcul consiste à additionner les intensités de pixels dans chaque région et à calculer les différences entre les sommes.



Le nombre d'opérations est réduit en utilisant l'image intégrale.

Au lieu de calculer à chaque pixel, il crée à la place des sous-rectangles et crée des références de tableau pour chacun de ces sous-rectangles. Ceux-ci sont ensuite utilisés pour calculer les caractéristiques de Haar.

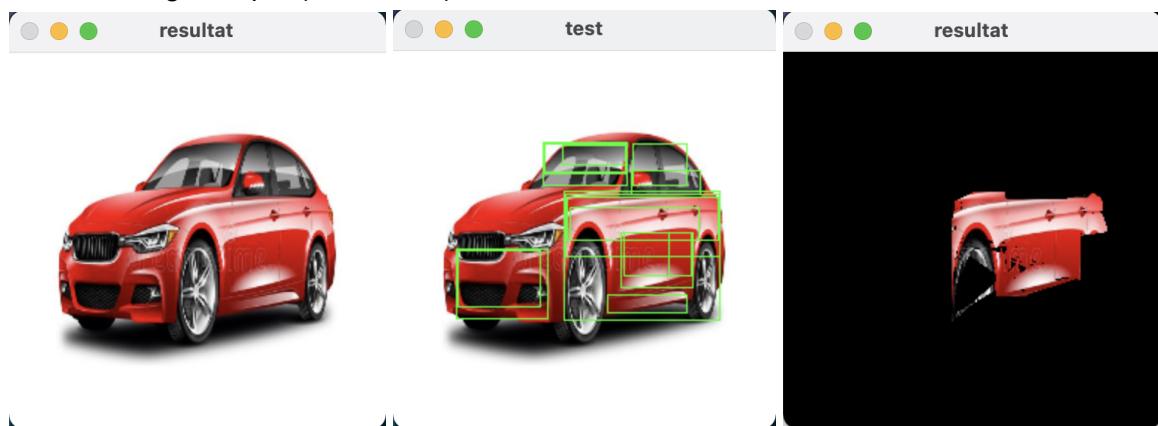
Jami :

Essais de la fonction cv2.detectRegions() d'openCV avec différents traitements de l'image.

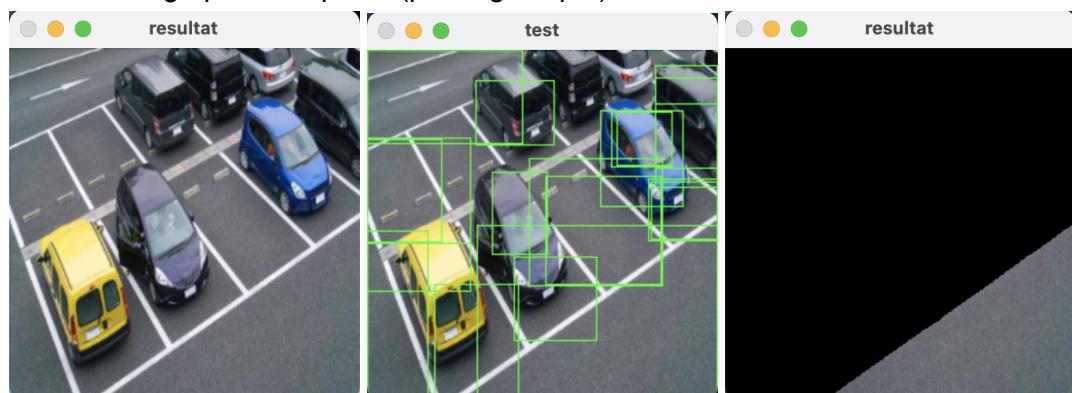
La fonction crée ajoute un masque noir à l'image de base. Ce masque est là pour appuyer les zones découpées. Il est donc possible d'entourer les objets comme Arno. Dans tous les résultats suivants, la fonction retourne une liste de coordonnées de points sur l'image donnant des zones, l'affichage n'affiche que les zones supérieur à la taille moyenne des zones. C'est pour ça que uniquement les grandes zones ne sont pas sélectionnées.

Segmentation sans traitement :

Sur une Image simple (fond blanc) :



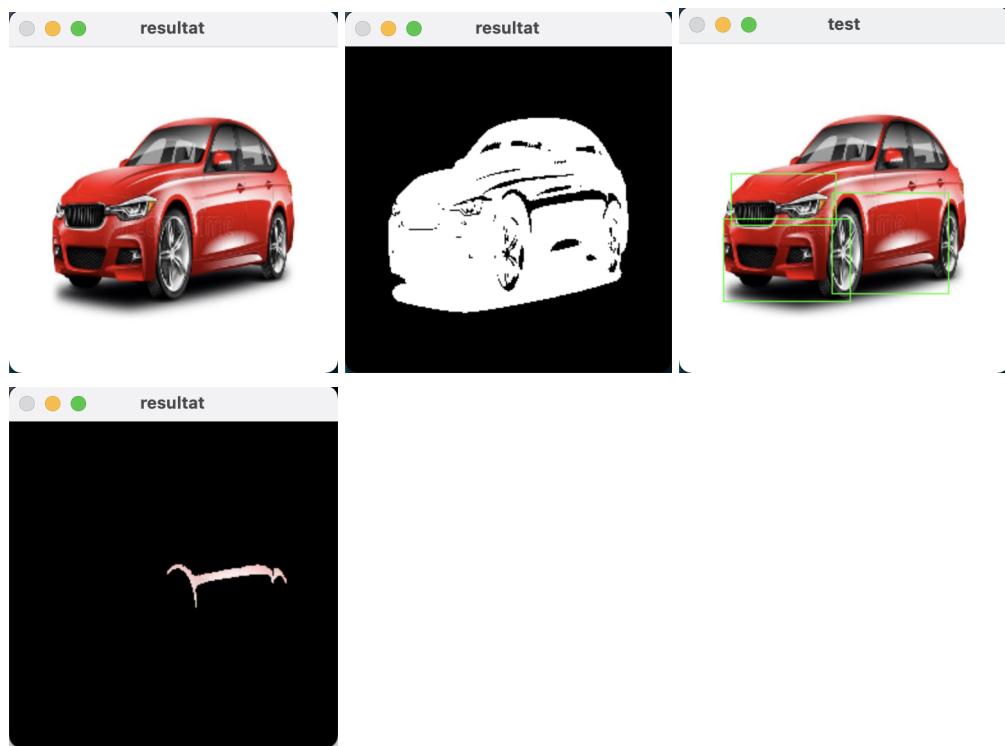
Sur une image plus complexe (parking simple) :



⇒ Sans traitements, la segmentation n'est vraiment pas efficace, elle ne détecte rien d'intéressant pour nous.

Segmentation avec threshold simple basé sur le moyen de nuances de gris de l'image :

Sur une Image simple (fond blanc) :



Sur une image plus complexe (parking simple) :

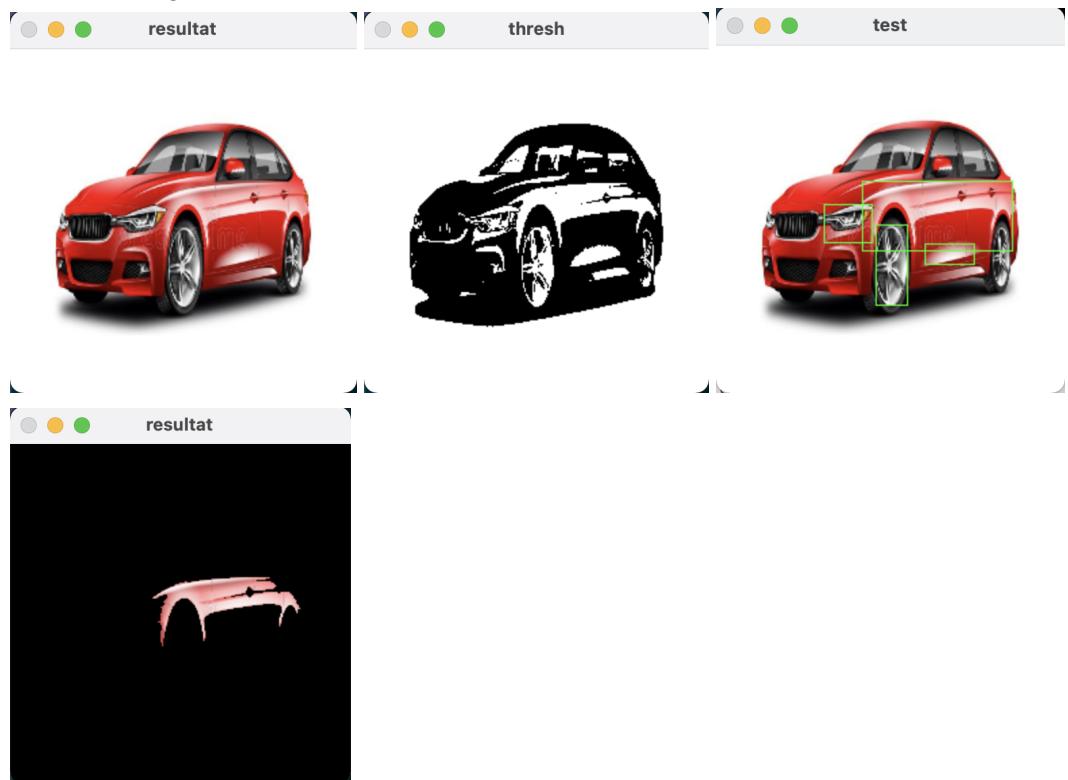


⇒ Sans traitements, la segmentation n'est vraiment pas efficace, elle crée plein de zones sur l'image mais le résultat est mieux que sans aucun traitement.

Avec un traitement Threshold adaptatif :

voir partie de Corentin pour explications.

Sur une Image simple (fond blanc) :



Sur une image plus complexe (parking simple) :



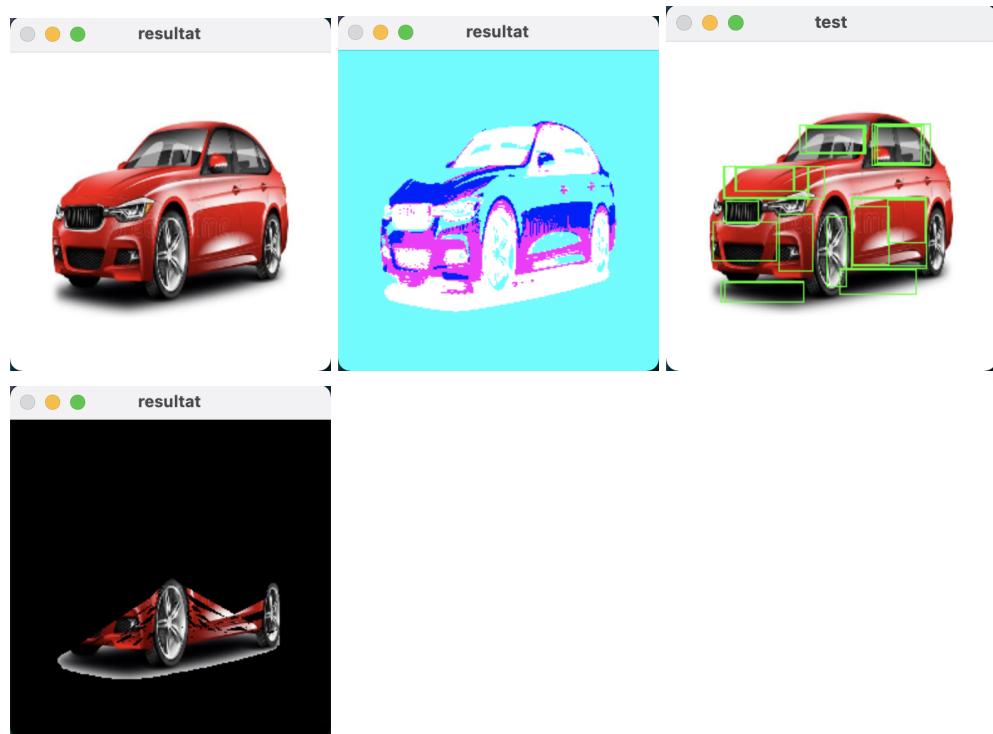
Cette fois les résultats sont moyennement intéressants, un des centres d'attention est bien détecté. Cependant, tous les objets ne sont pas relevés, les zones sont trop imprécises.

Segmentation avec un traitement HSV simple en plus du threshold simple précédent :

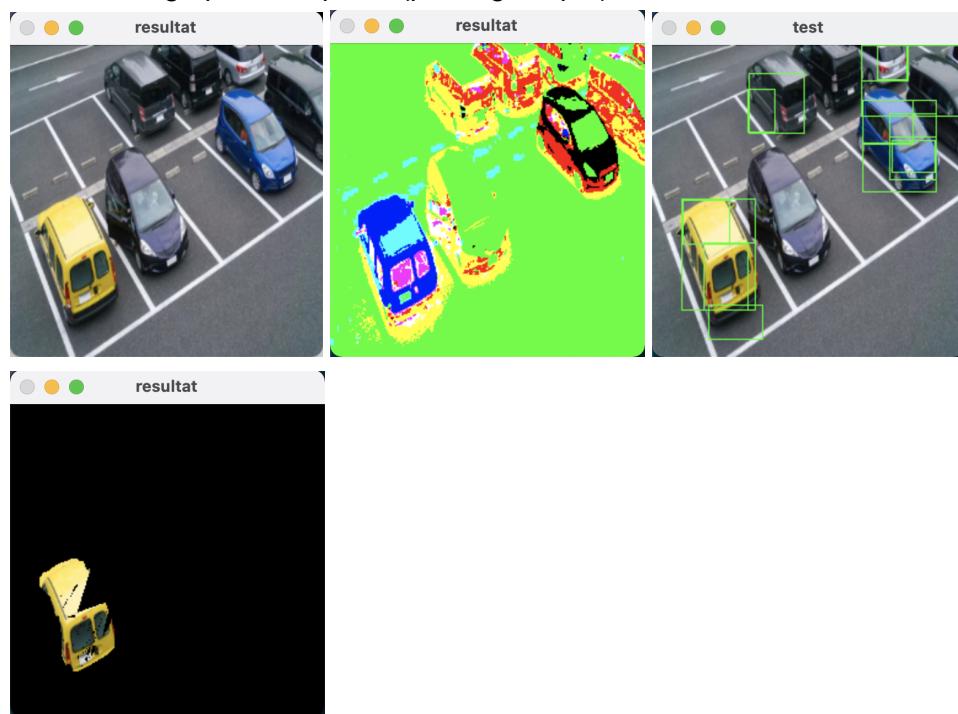
voire parti de Corentin pour explications.

Mettre uniquement un traitement csv avant la segmentation n'est pas pertinent.

Sur une Image simple (fond blanc) :



Sur une image plus complexe (parking simple) :

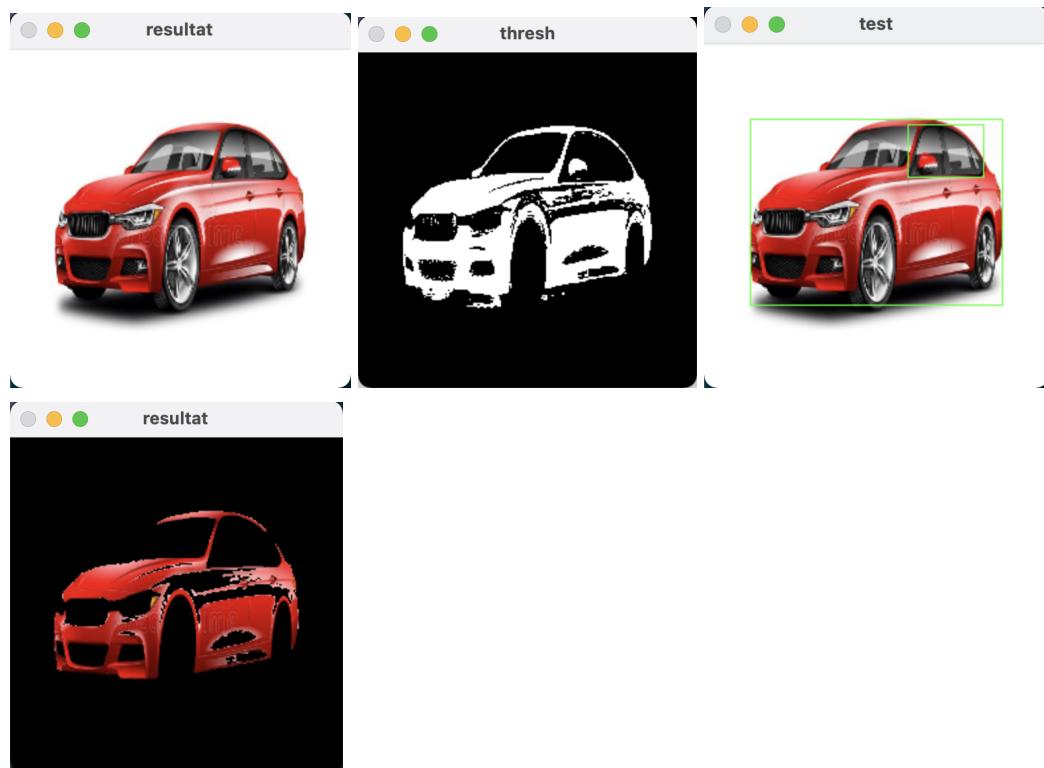


Cette méthode n'est pas la plus efficace, aucun des objets cible n'a été mis en avant.

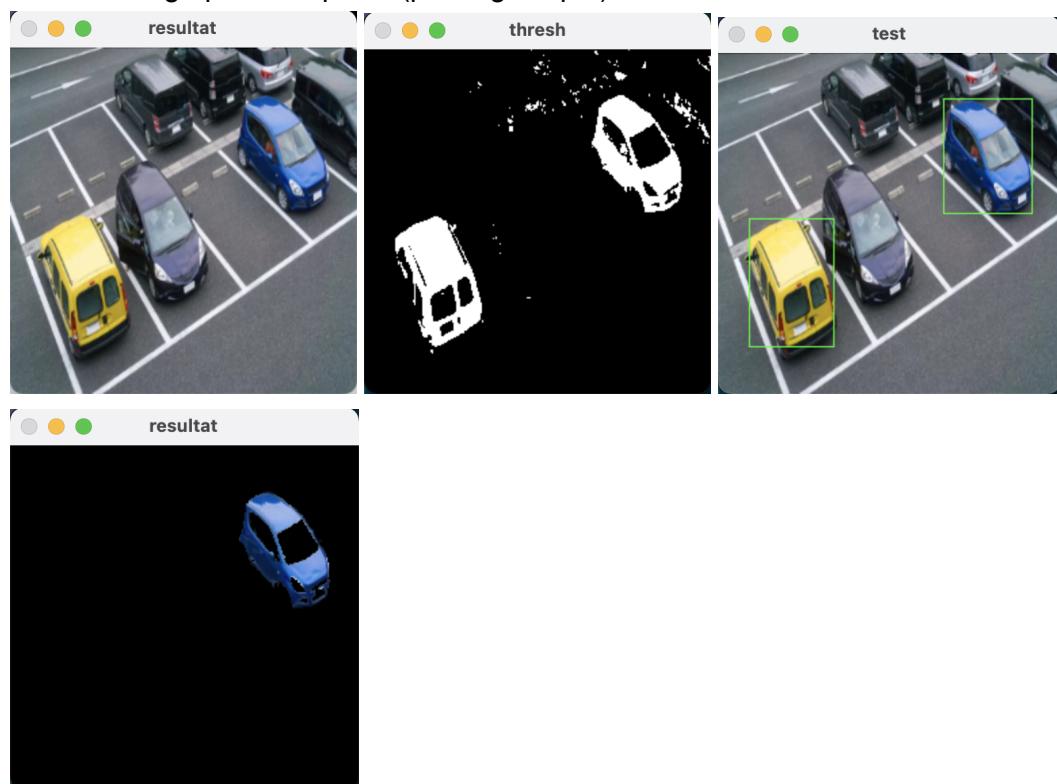
Avec un traitement HSV simple en plus du threshold OTSU :

voire parti de Corentin pour explications.

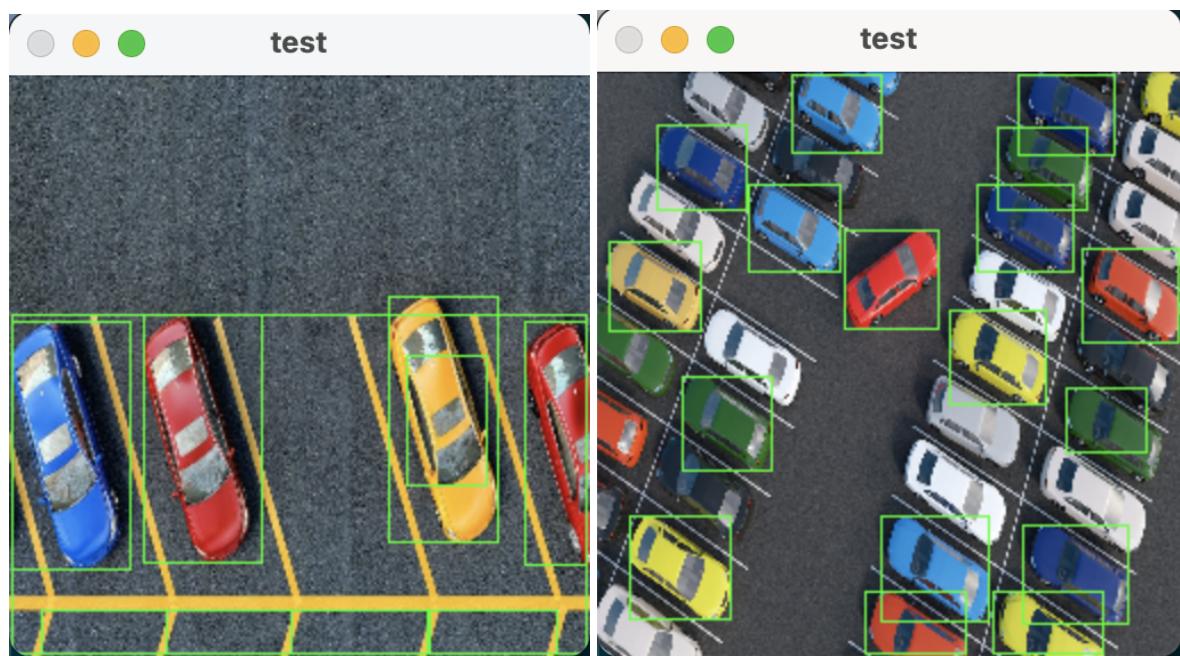
Sur une Image simple (fond blanc) :



Sur une image plus complexe (parking simple) :







Cette fois les résultats sont plutôt concluants, un des centres d'attention est bien détecté. Seulement, les objets de couleurs foncées ne sont pas mis en relief par le threshold.

Arno :

Désormais j'arrive à obtenir un ensemble de points faisant la silhouette de l'objet, tout en n'ayant aucun objet interne à la voiture



Mon but sera d'utiliser le travail de Corentin afin d'avoir la meilleure technique pour avoir les contours les plus précis de l'objet. Par la suite, je pourrais utiliser d'autres images, plus

complexes et / ou contenant plusieurs objets. Je vais donc devoir utiliser le résultat ci-dessus et le combiner avec le résultat ci-dessous :



Afin d'obtenir une ligne complète qui prend l'entièreté de la voiture, sans aucun objet à l'intérieur dessiné.

J'ai essayé ma fonction avec la fonction de Corentin et on obtient les résultats suivants :



On voit que les deux voitures en jaune et en bleue sont bien entourées mais les autres ne le sont pas. J'en conclus que la fonction est efficace uniquement sur les voitures de couleurs autres que noires / gris foncé

Julien :

Cette semaine j'ai commencé la méthode svm.

J'ai pris le dataset load_digits de sklearn.

Le modèle svc prend deux hyperparamètres Gamma et C.

C permet de contrôler l'erreur, c'est-à-dire si C.

Au début, j'ai mis des valeurs aléatoires mais le nombre d'erreurs était trop grand. J'ai donc fait des tests pour trouver les meilleures valeurs empiriques. Ce qui donne le tableau suivant :

Gamma	Nombre Erreurs
10	415
1	409
0,1	411
0,01	99
0,001	3
0,0001	6
0,00001	8
0,000001	15
0,0000001	50

C	Nombre Erreurs
0,00001	413
0,0001	410
0,001	408
0,01	380
0,1	16
1	8
10	3

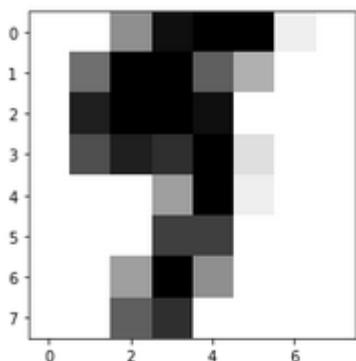
100	5
-----	---

À Partir de C=10 le nombre d'erreur n'évolue pas significativement

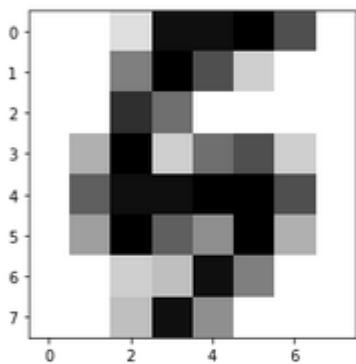
Donc pour gamma la meilleure valeur a priori serait 0,001 et pour C 10.

Grâce à ces paramètres, il y a en moyenne 4 erreurs. Exemple de résultat :

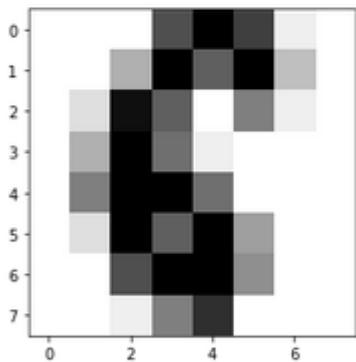
Predicted: 9 Actual: 5



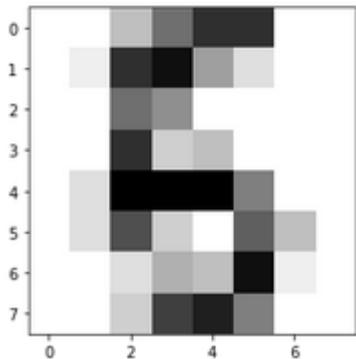
Predicted: 4 Actual: 5



Predicted: 8 Actual: 6

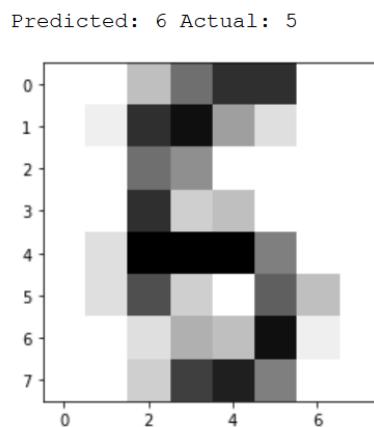


Predicted: 6 Actual: 5



Erreur: 4

J'ai remarqué qu'il y avait une image qui revenait souvent dans les erreurs :



Eliaz :

Cette semaine je suis passé sur des images en format 28x28. Ce dataset est toujours issu du MNIST, cette fois il est tiré de : <https://www.kaggle.com/oddrationale/mnist-in-csv>.

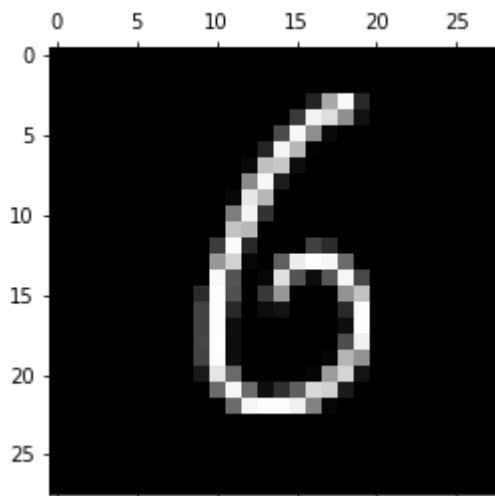


Figure ci-dessus : image n°18 du dataset de test, il s'agit ici d'un 6.

Tests autour de différentes “architectures” de couches cachées :

	... de 16 noeuds	... de 300 noeuds
1 couche cachée...	8.4% d'erreurs	2.5% d'erreurs
2 couches cachées...	7.2% d'erreurs	2.4% d'erreurs

(aller jusqu'à 3 couches et faire des graphs)

Les résultats de cette expérience, ainsi que [ce post](#) sur stackexchange, me font dire que le nombre idéal de noeuds/couches pour la couche cachée d'un réseau de neurone est en général une couche de avec un nombre de noeuds compris entre le nombre de noeuds sur la couche d'entrée et le nombre de noeuds sur la couche de sortie.

Après, dans l'optique de passer sur l'application aux datasets