IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TECHNICAL REPORT - DATA PARSING

Eustache LE BIHAN
MSc in Engineering student at IMT Atlantique
August 2022

1 Introduction

1.1 Le *dataset*

La base de données est constituée de 192556 images accompagnées de leurs annotations. De ces 192556 images, seules 42556 contiennent au moins un bateau
(cf. tools/data_info.ipynb).

1.2 Les annotations

Ces annotations sont stockées sous le format texte *Comma-separated values* (CSV) et segmentent l'image et ses pixels en deux catégories : pixels qui désignent un bateau et pixels qui n'en désignent pas. Cette information est stockée sous le format *run-length-encoding* (RLE), c'est-à-dire que chaque bateau est décrit par mot (RLE) désignant les pixels de l'image qui lui correspondent.

1.3 Les images

Les images sont au format *JPEG* et d'une taille uniforme de 768*768 pixels. Elles sont extraites de parties d'images acquises par le satellite Pléiades puis subdivisées par pas de 256 pixels.

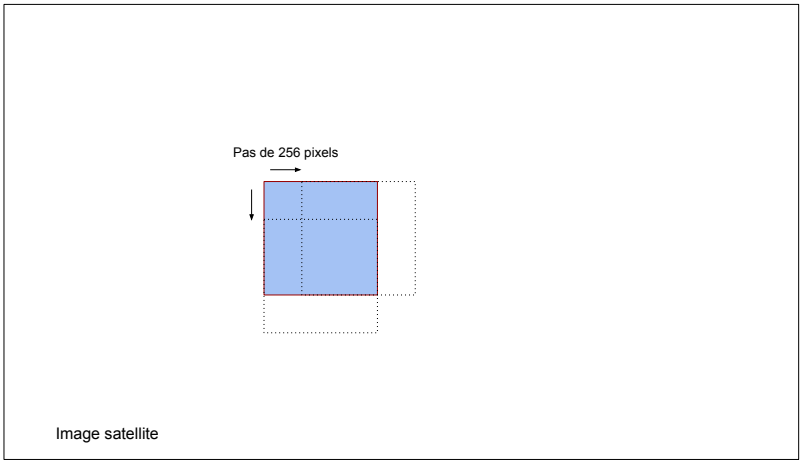

FIGURE 1 – Les images de la base sont extraites d'images satellites

1.4 Les duplicatas

Une image peut donc avoir jusqu'à vingt-quatre images « voisines », c'est-à-dire des images dont une partie recouvre la première. Un même bateau peut donc être présent sur plusieurs images.

Ces duplicatas sont une barrière à la constitution des jeux de données d'entraînement et de validation, pourtant nécessaires à l'entraînement de modèles de *deep learning*. En effet,

une simple séparation aléatoire du jeu initial en deux jeux entraînement/validation amènerait un même bateau à apparaître dans les deux bases (puisque présent sur deux images voisines). Les résultats apportés par un tel jeu de validation seraient inexploitables. Cela reviendrait à évaluer notre modèle sur des images sur lesquelles il a été entraîné et par conséquent ne permettrait pas d'évaluer ses performances de généralisation.

Il s'agira donc dans un premier temps de s'assurer que la répartition des images dans ces deux jeux soit faite de façon à éviter une telle situation. De plus, il ne suffit pas d'isoler toutes les images qui contiennent un bateau. En effet, un image peut également en contenir d'autres, eux-mêmes dupliqués sur d'autres images. Il faut donc former des *clusters* d'images contenant des bateaux qui n'apparaissent sur aucune autre image que celles présentes dans le *cluster*. Ce seront ensuite ces *clusters* qui seront répartis dans nos deux bases.

FIGURE 2 – Une image et huit de ses voisines

2 Formation des *clusters*

2.1 Identification d'un bateau

Postulat :

À chaque bateau du jeu de données correspond une unique forme. Si deux bateaux sur deux images ont la même forme, c'est qu'il s'agit du même bateau. Les images se recouvrent donc en partie, elles sont « voisines ».

Les navires contenus par les images sont décrits par un RLE (*run-length-encoding*). Le format RLE porte deux informations :

1. la forme du bateau
2. sa position dans l'image

La première est universelle au sein du jeu de donnée (cf. postulat), la deuxième diffère selon les images. En tirant du RLE de chaque bateau la forme désignée, on identifie donc ce bateau de façon unique. Pour cela, il suffit de calculer le RLE correspondant à la même forme de bateau mais placée de le coin supérieur gauche de l'image. Puis, il faut construire un identifiant entier à partir du mot RLE ainsi normalisé à l'aide d'une fonction de hachage. Cet entier sera nommé *BoatHash*. (cf. data_parsing/CSV/boats_hash.csv). Ce principe est implémenté dans la fonction *hash_boats_rle* du module data_parsing/hash/img_hash.py.

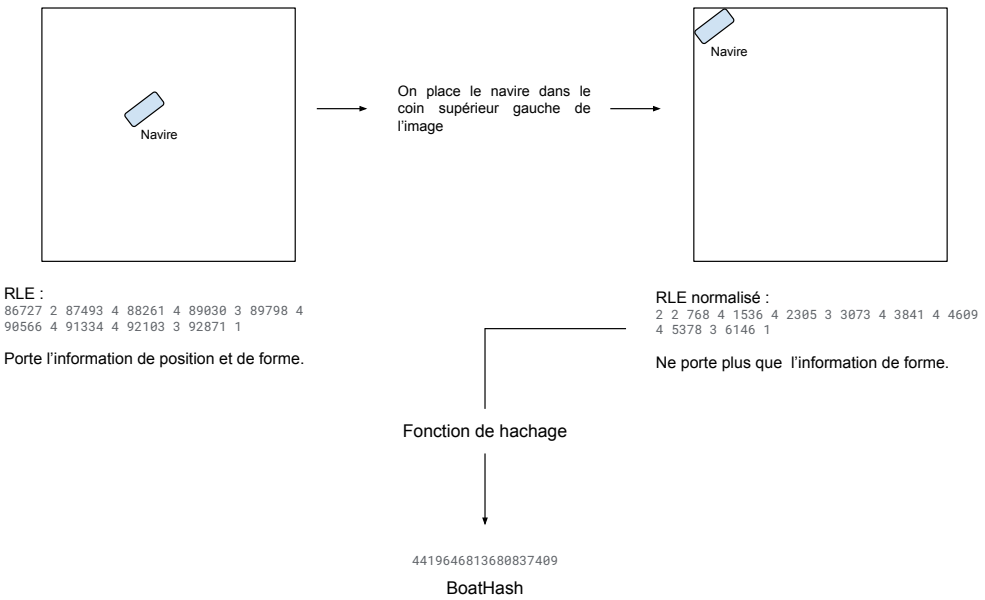

FIGURE 3 – Génération d'un identifiant

2.2 Objectif et stratégie

Il s'agit de former des *clusters*, c'est-à-dire des groupes d'images qui ne doivent pas être séparées lors de la division de la base en base d'entraînement et de validation. Pour cela, on peut traiter le problème comme un problème réseau. En effet, on peut voir deux images contenant le même bateau comme deux sommets reliés d'un graphe. Ainsi, trouver les *clusters* d'images revient à trouver les *clusters* de sommets reliés entre eux et isolés du reste. Cette méthode est implémenté dans la fonction find_clusters du module data_parsing/hash/img_hash.py.

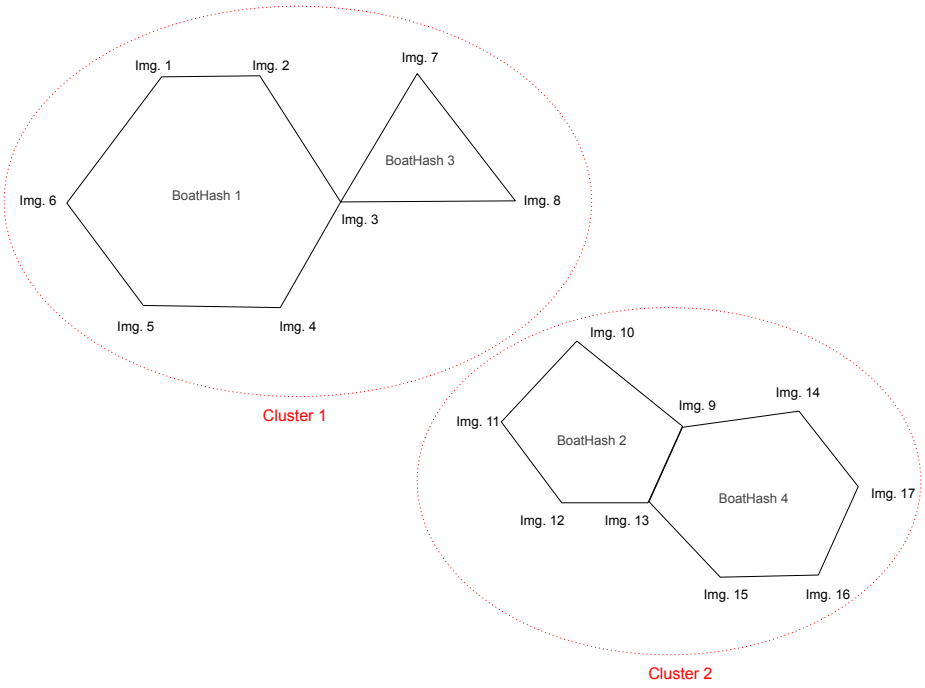

FIGURE 4 – Analogie réseau du problème

De cette manière, la base de 192556 images est réduite en 5040 *clusters* contenant les 42556 images avec bateau. Ces *clusters* sont dans la liste data_parsing/CSV/clusters_h.pkl enregistrée au format binaire (*pickle*).

Jusqu'ici, la méthode utilisée à été optimale. Plus précisément, si l'on considère qu'un bateau est entièrement caractérisé par sa forme (cf. postulat), l'objectif est ici atteint. Aucun bateau, c'est-à-dire aucun *BoatHash* n'est présent dans deux *clusters* à la fois. Cependant, nous allons maintenant voir que cette approche n'est pas suffisante.

2.3 Défauts de la base de données

Le bon fonctionnement de la méthode précédente repose sur l'hypothèse qu'un même bateau contenu dans deux images se superposant ait nécessairement la même forme (cf. postulat). Pourtant, on trouve dans le *dataset* des images portant le même bateau mais dont

la forme (décrite par un RLE) diffère. Sur la fig. 5 ont été dessinées les boîtes les plus petites englobant un bateau tel que décrit par son RLE. Les formes diffèrent donc ces deux images n'ont pas été placées dans le même *cluster* et par conséquent les 5040 *clusters* trouvés ont des images qui se superposent.

FIGURE 5 – Différentes formes pour le même bateau

2.4 Résolution du problème engendré

Pour palier à ce problème, une approche est de tenter de reconstruire les images satellites dont ont été prises nos images afin de connaître les *clusters* qui doivent être assemblés.

2.4.1 Une première approche naïve

Nous savons que nos images ont été formées par pas de 256 pixels et donc qu'une image peut avoir jusqu'à 8 voisines. Par conséquent, nous savons sur quelles zones nos images peuvent potentiellement se recouvrir. À partir d'une image, il suffirait de chercher dans l'ensemble du *dataset* ses 8 voisines en comparant pixels par pixels. Cette approche naïve est d'une complexité irréalisable.

Pour réduire le temps le calcul, on pourrait imaginer utiliser une fonction de hachage qui permettrait de projeter un cadre de 256*256 pixels (soit 256*256*3 valeurs) sur un entier de façon déterministe. Ainsi, on pourrait commencer par calculer les 9 haches de chaque images (puisqu'une image de 768*768 se divise en 9 cadres lorsque utilise un pas de 256 pixels) puis les comparer image par image, ce qui diviserait donc le nombre de comparaison par 256*256*3.

Cependant, lorsque la base a été formé et donc que nos images de 768*768 ont été extraites des images satellites, une compression *JPEG* a été utilisée. Par conséquent, des artefacts de compression, même imperceptibles à l'œil nu résultent en deux haches différentes pour deux cadres censés correspondre.

2.4.2 Une approche non optimale - algorithme glouton

Il est nécessaire d'introduire une approche qui permette d'estimer la similarité entre deux cadres. En effet, deux cadres censés se superposer doivent être quasi identiques mis

à part quelques pixels résultant d'une compression différente. Cependant, encore une fois, une comparaison pixels par pixels, permettant d'estimer un taux de similarité est d'une complexité irréalisable.

L'approche choisie (méthode « mosaïque ») est de compresser chacun des 9 cadres de nos images en de nouvelles images de 6*6 en utilisant une interpolation *pixel area*. Ainsi, deux cadres quasi-similaires auront été compressés en deux nouveaux cadres de 6*6 presque identiques avec les quelques pixels différents lissés par l'interpolation. Un score de similarité peut maintenant être calculé en comparant nos 6*6*3 valeurs ainsi trouvées, ce qui réduit énormément la comparaison de 256*256*3 initialement nécessaire. Ce principe est implémenté dans le module data_parsing/mosaics/build_crops.py.

Ensuite, l'utilisation de *multiprocessing* afin d'être certain d'utiliser les 96 coeurs de calcul disponibles et d'établir des tunnels de données afin d'être certain que les images d'un *cluster* formé ne soient plus analysées lors de la recherche de voisins d'une images permet également d'énormément réduire le temps de calcul (implémenté dans le script data_parsing/moscaics/cluster.py)

Finalement, un calcul irréalisable a été réduit à un calcul d'une nuit apportant une solution non optimale mais satisfaisante. Cependant, cette approche reste non optimale et il est probable que nombre d'images n'aientf pas été entièrement recomposées. Aussi, l'approche *multiprocessing* fait que certains *clusters* ont été trouvés en parallèle par plusieurs cœurs de calcul, ce qui induit des duplicatas. Le script data_parsing/moscaics/clean_cluster.py supprime ces duplicatas et une version "propre" du résultat, c'est-à-dire de la liste des *clusters* d'images appartenant à une même image satellite, est enregistrée au format binaire (*pickle*) sous data_parsing/CSV/clusters_clean.pkl.

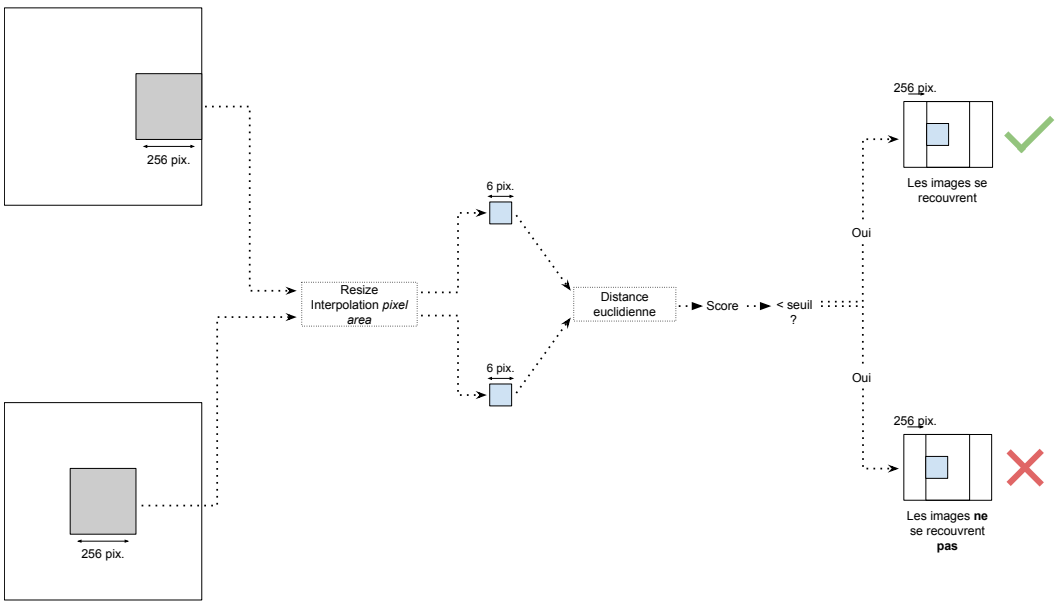

FIGURE 6 – Méthode "mosaïque"

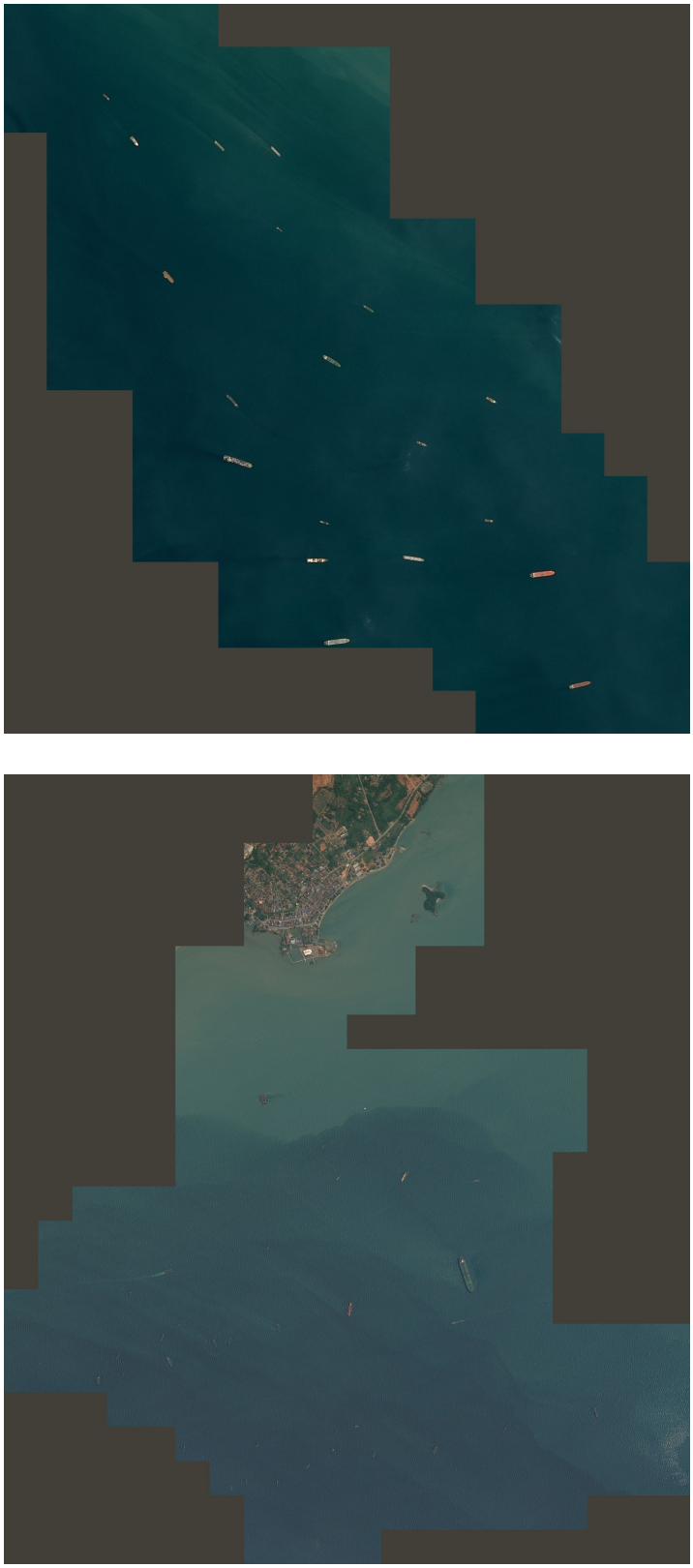

FIGURE 7 – Exemples d'images satellites recomposées

2.4.3 Fusionner les deux approches

Les résultats obtenus, c'est-à-dire les *clusters* d'images appartenant à la même image satellite peuvent être associés aux résultats apportés par la première approche. En effet, même si à cause d'artefacts de compression deux bateaux pourtant identiques n'ont pas exactement la même forme, si deux *clusters* d'images appartiennent à la même image satellite, on peut considérer qu'ils doivent être rassemblés en un même *cluster*. Ainsi, le nombre de *clusters* est réduit à 3115. Cette méthode est implémentée dans le script data_parsing/hash/reassemble_clusters.py et la liste des 3115 *clusters* est enregistrée au format binaire (*pickle*) sous data_parsing/CSV/cluster_reassembled.pkl. De plus, l'ensemble des *clusters* est décrit par un fichier csv sous data_parsing/CSV/clusters.csv associant les *clusters* aux images qu'ils contiennent ainsi que les bateaux contenus.

2.5 Formation des bases d'entraînement et de validation

Former la base de validation revient à prélever un échantillon base de départ, usuellement 20%. La base d'entraînement sera ensuite formée avec les images restantes. Les résultats obtenus sur la base de validation permettent de suivre l'évolution de l'entraînement du modèle. Ainsi, ils seront d'autant plus pertinents que cette base de validation sera représentative de la base de départ. Un simple échantillonnage aléatoire ne permet pas d'en être certain. La méthode à utiliser est l'échantillonnage stratifié qui consiste, après avoir classé une base selon certains critères et en formant ainsi des strates, à prélever le pourcentage souhaité de chacune de ces strates. Dans notre cas, puisqu'on ne prélève pas directement des images mais des *clusters* d'images, il s'agit de définir nos critères, autrement dit nos différentes strates, sur ces *clusters*. Ainsi, on va se baser sur le nombre de bateaux présents dans un *cluster* ainsi que leurs largeur et hauteur moyennes. Puis, on classe les hauteurs moyennes de nos *clusters* en 10 catégories, de même pour les largeurs. Le nombre de bateaux présents par *clusters* sont quant à eux classés en 5 catégories. En groupant nos *clusters* par catégorie de largeur moyenne, puis par catégorie de hauteur moyenne et enfin par nombre moyen de bateaux, ont obtient nos strates. Il suffit donc de prélever le pourcentage voulu de chacune de ces strates. Cette méthode est implémentée dans la fonction *split* du script train_test_split/split.py. En traçant la distribution des tailles de bateau (largeur et hauteur) pour les deux bases ainsi obtenues, on peut vérifier que notre base de validation est bien représentative de notre base d'entraînement. Voici ci-dessous un exemple du résultat obtenu pour une base composée de 70% d'images contenant au moins un bateau et séparée à 80% dans la base d'entraînement et 20% dans la base de test.

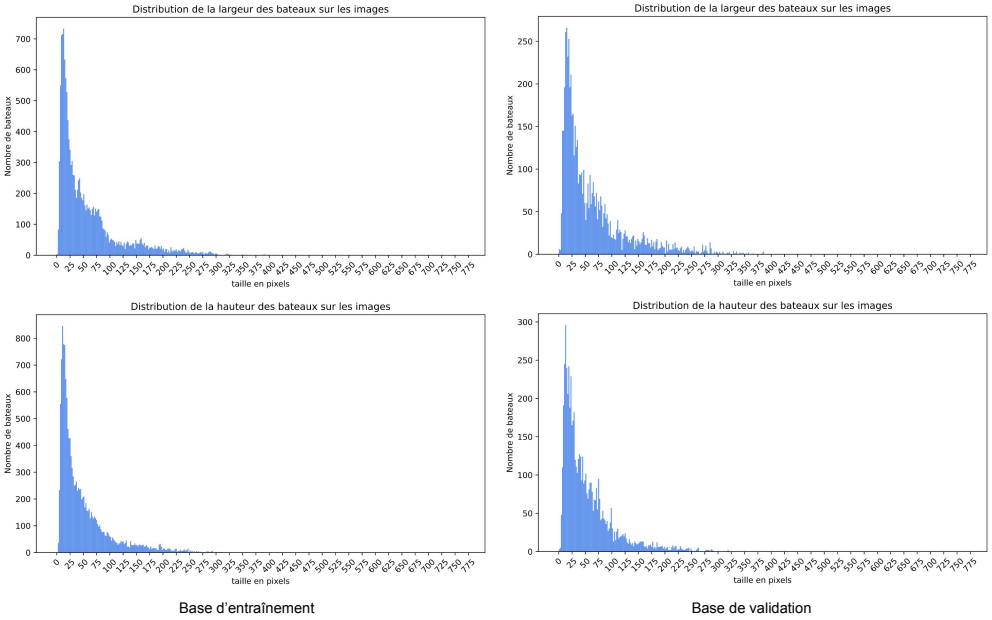

FIGURE 8 – Distributions des tailles de bateaux