Augustin Boissier
Théophile Rogy

MODAL INF473V
AIRBUS SHIP DETECTION

RAPPORT FINAL

TABLE DES MATIÈRES

1
PROBLEM AND MAIN IDEAS

1.1 Airbus ship detection

Shipping traffic has become vital in our economy, and is one of the main gears of international commerce today. By growing in importance, ships have become the target of criminal organizations, perhaps best exemplified by growing piracy, as well as drug cartels who use ships for drug trafficking. Ships are also the culprits in various offenses such as illegal fishing, oil spills ... It is therefore of a growing importance for environmental, economic and political organizations to create a system that can monitor ships on an international level.

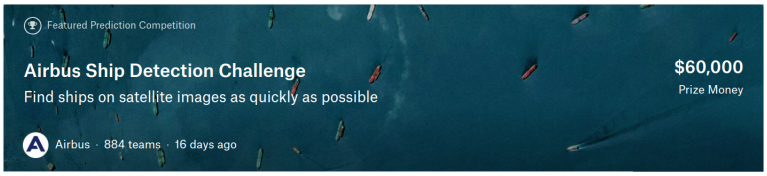

Fig. 1 – *Competition banner*

Airbus is one of the main players today in ship monitoring, and offers solution to support companies and organizations alike in monitoring their fleets. Airbus proposed kagglers to improve their current detection models by submitting an extensive database of satellite imagery of sea images. Our objective was therefore, given an image database to produce an **image segmentation model** to classify whether an image has ships or not, and if it does, to create bounding boxes of the ships detected.

1.2 Metrics used

To evaluate our mdoels, we will be using a criterion given by kaggle, the average F2-score which, in this case, is derived from the IoU (Intersection over Union), seen in TD7. By naming A and B two subsets of our image :

$$IoU = \frac{A \cap B}{A \cup B}$$

If the IoU is over a given threshold t, the model predicts a positive, and likewise, if it is below, the model returns a negative. If we name TP (True Positives), TN (True Negatives), FP (False Positives), FN (False Negatives), the F_2 score is computed by :

$$F_2(t) = \frac{5TP(t)}{5TP(t) + 4FN(t) + FP(t)}$$

The final criterion we will be using is the *average* F_2 *score*, whose formula is given by :

$$\overline{F_2} = \frac{1}{\#\{thresholds\}} \sum_{t \in [0.5,\dots,0.95]} F_2(t).$$

1.3 Main Methods - redondant par rapport à partie II

1.3.1 Augmentation d'image

Le principal défi du problème a été que le jeu de données que nous avons reçu contenait beaucoup d'images sans bateaux. Une méthode importante que nous avons alors dû mettre en oeuvre est **l'augmentation d'images**, sur laquelle nous reviendrons plus en détail ensuite.

1.3.2 Fonction de loss

??

2
CORE WORK AND IMPLEMENTATION

2.1 Problems to overcome

2.1.1 Jeu de données et Images

Le premier problème qui s'est présenté à nous a été *l'encodage RLE* ou *Run Length-Encoding* avec lequel nous étions peu familiers, alors que nous avions vu en cours uniquement les *bounding boxes* délimitées par leurs coordonnées. Il s'agit d'un format d'encodage utilisé en compression de données, et encodant le nombre de pixels d'une couleur donnée se suivant, dont l'application pour encoder des masques est évidente.

```
ImageId                                                 000194a2d.jpg
EncodedPixels    360486 1 361252 4 362019 5 362785 8 363552 10 ...
Name: 3, dtype: object
```

Fig. 2 – *Encodage RLE*

Le second problème que nous devions surmonter est le fait que la plupart (environ 75 %) des images ne contenaient pas de bateaux, il s'agissait d'un *imbalanced dataset* . Ainsi, ces images ne pouvaient pas être utilisées pour entraîner notre segmenteur, et pire, lorsque nous les laissions dans le jeu d'entraînement les performances globales chutaient. Nous nous avons donc décidé d'exclure les images ne contenant pas de bateaux. Cela nous laissait avec 42 256 d'images de bateaux. Cependant même pour celles-ci, une large partie ne contenait que 1 ou 2 bateaux (*cf. Fig 2.*), et donc n'aboutissait *in fine* qu'à la création de 1 ou 2 masques.

Fig. 3 – *Histogrammes des répartitions des données*

Enfin un dernier obstacle. Une partie des images (≈ 1000) était corrompue : elle contenait de larges bandes bleues ou vertes par exemple. Nous avons donc dû trouver ces images et les enlever du jeu d'entrainement, pour éviter que le modèle soit entrainé sur des données aberrantes.

2.1.2 Problèmes liés à kaggle

Nous avons voulu travailler sur kaggle, car il présentait une API qui permettait d'importer avec facilité le jeu de données, dont le poids (29.25 GB) rendait le passage sur une autre plateforme difficile. Ce choix a amené avec lui ses propres problèmes :

– Kaggle a une limitation d'utilisation de GPU de 30h, ce qui a limité le nombre possible d'expériences

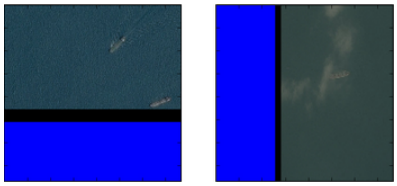

Fig. 4 – *Exemple d'images corrompues*

avec des hyperparamètres

- Kaggle déconnecte l'environnement d'exécution en cas d'inactivité prolongée, en particulier pour entrainer le modèle, nous avons eu plusieurs fois des problèmes de déconnexion
- Kaggle rend difficile la collaboration à deux sur le même noteboook et nous avons souvent eu des problèmes de versions non mises à jour
- Enfin Kaggle nous a parfois joué des tours, et a planté pour des erreurs de RAM que nous aurions surconsommé. Cela a parfois posé problème lorsqu'à une phase d'entraînement avancé notre programme a planté.

```
RuntimeError: CUDA out of memory. Tried to allocate 256.00 MiB (GPU 0; 15.90 GiB total capacity; 14.9
1 GiB already allocated; 129.75 MiB free; 15.01 GiB reserved in total by PyTorch)
```

Fig. 5 – *Exemple de problème posé par kaggle*

2.2 Main Building blocks of our implementation

2.2.1 Data cleaning

Les images sans bateaux n'étaient pas très intéressantes dans notre situation. En effet, elles ne contiennent pas d'informations utiles pour entraîner notre réseau de neurones. Etant donné le poids de ces images dans notre base de données, nous avons pris la décision de supprimer les images ne contenant pas de bateaux.

2.2.2 Image Augmentation

Pour surmonter les problèmes liés au jeu de données, nous avons dû faire de l'image augmentation. Cette technique est utile dans notre cas et sert à éviter *l'overfitting* sur les données. Elle consiste à appliquer des modifications légères aux images pour augmenter la taille du jeu de données. Nous étions partis au départ sur un framework utilisant la librairie *Image data preprocessing de Keras* [2]. Néanmoins, cela a posé des problèmes de compatibilité avec la suite de notre code, et nous avons finalement opté pour programmer manuellement l'augmentation d'images, en utilisant cette fois-ci les fonctions de la librairie transforms de Pytorch.

Ces modifications sont représentées ci-dessous, et sont une combinaison des modifications suivantes :

- symétrie axiale horizontale (*horizontal flip*)
- symétrie axiale verticale (*vertical flip*)
- rotation
- rognage aléatoire

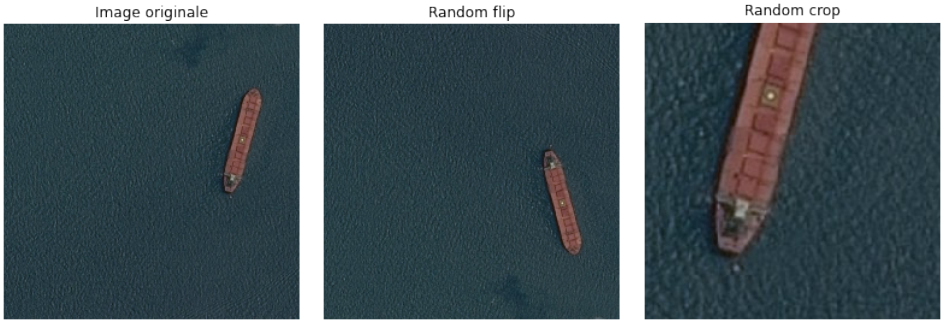

Fig. 6 – *Quelques transformations d'images*

L'image augmentation a grandement contribué à augmenter notre jeu de données, en générant de ñouvelles ïmages labellisées, ainsi qu'à éviter l'overfitting en élargissant le nombre de situations auxquelles notre modèle avait à faire.

2.2.3 Construire une architecture pour la segmentation

Après des recherches sur les modèles de segmentation d'images, nous avons délimité les principaux modèles utilisés en segmentation d'image : U-Net (2015), Mask R-CNN (2017), FastFCN (2019).

Nous avons appris que le U-Net [1] avait été construit historiquement par l'Université de Fridburg pour la détection d'objets biomédicaux. Or ce problème est en apparence assez similaire au nôtre : trouver un objet assez petit (comparativement à l'image) et spécifique au sein d'une image de fond relativement uniforme. C'est ce qui nous a conduit à choisir le U-Net dont l'architecture est détaillée ci-dessous.

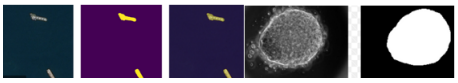

Fig. 7 – *Comparaison de la nôtre tâche vs la tâche intiale du U-Net*

Le U-Net se compose ainsi de deux parties, de formes symétriques, lui conférant sa structure spécifique en U. La première partie, composé de structures dites Down, sert à contracter les informations spatiales, tout en augmentant le nombre de channels. La seconde partie de la structure est une Up ou expansive, qui combine ces informations grâce à des réseaux convolutifs.

Images/ModÃÍle/Unet.PNG

Fig. 8 – *Architecture du U-Net [1]*

Nous avons initialement adopté le U-Net classique. Nous avons dû effectuer des modifications à la structure du réseau pour améliorer la performance du réseau. Le U-Net classique est constitué de 5 blocs *down* ainsi que de 4 blocs *up*. Nous avons rajouté 3 blocs *down* et *up* ce qui a eu pour effet d'améliorer la performance. Au sein de chaque bloc nous avons ajouté 2 couches de convolution/batch normalisation. Enfin nous avons ajouté 2 couches de convolution et batch normalisation entre le dernier bloc down et le premier bloc up. La différence entre l'architecture classique est représentée en annexe.

2.2.4 Choisir une fonction de loss

Pour choisir la fonction de loss, nous avons étudié les fonctions de loss utilisées classiquement en segmentation d'image [3]. Ayant réduit le choix de fonctions de loss, nous avons choisi les trois fonctions de loss les plus proches de l'IoU, à savoir : la distance de Jaccard, Dice Loss et BCE Dice loss, dont nous rappelons les définitions.

$$Dice = \frac{2|X \cap Y|}{|X| + |Y|}$$

$$Jaccard(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

$$BCEDice(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} H\left(p_n,q_n\right) = -\frac{1}{N}\sum_{n=1}^{N}\left[y_n \log \hat{y}_n + (1-y_n)\log\left(1-\hat{y}_n\right)\right]$$

Initialement, nous avions prévu de combiner différentes fonctions de loss, comme précisé en annexe. Finalement, les gains de cette combinaison de fonctions étant minimes, nous avons retenu la BCE Dice Loss, car c'est elle qui nous permettait d'obtenir les meilleurs résultats.

3
QUANTITATIVE EVALUATION OF OUR FINAL SOLUTION

3.1 Decomposition of building blocks

3.2 Convergence of network and overfitting

4 RESULTS

4.1 Results obtained

4.2 Parameter finetuning

5
CONCLUSION ET EXTENSIONS POSSIBLES

5.1 Conclusion générale

Ce problème proposé par Airbus s'est averé difficile à résoudre, car contrairement aux exemples que nous avions vus en cours où l'objet à identifier était l'élément central de la photographie, ici les bateaux étaient en général de simples tâches dans la mer. Nous n'avons donc pas pu simplement utiliser les algorithmes pré-entraînés sur lesquels nous avions travaillé en TD et il a fallu trouver une architecture spécifique pour ce problème précis. Le U-Net nous est assez vite apparu comme étant une piste prometeuse et nous avons ensuite conçu notre propre version optimisée pour ce problème.

Au final, notre solution offre un score publique de

5.2 Limites et extensions possibles

Si notre modèle nous a permis d'optenir une performance acceptable, celle-ci reste loin derrière les meilleurs modèles. En particulier le fait de se restreindre à la bibliothèque *Pytorch* nous a privé de certains outils et bibliothèques qui auraient pu améliorer notre performance, notamment en terme de suivi de performance des hyperparamètres (Comet ML par exemple). Ainsi nous avons constaté qu'utiliser un ResNet pré-entrainé sur ImageNet n'avait qu'un impact limité sur la performance, mais avoir accès à un algorithme plus performant, pré-entrainé sur un jeu de données plus large que ImageNet aurait sûrement pu améliorer notre performance.

Une autre piste que nous n'avons pas explorée est de faire varier le learning rate. En effet notre réseau convergeait au bout de quelques époques seulement et pour des raisons de temps (pour pouvoir explorer le plus de variations possible de notre modèle), nous n'avons pas pu le mettre en place.

RÉFÉRENCES

[1] Olaf Ronneberger, Philipp Fischer, Thomas Brox *U-Net: Convolutional Networks for Biomedical Image Segmentation*

[2] *https://keras.io/api/preprocessing/image/*

[3] Shruti Jadon *A survey of loss functions for semantic segmentation*

[4] Dimitry Larko *https://www.youtube.com/watch?v=0Opb8gB1p4w*

6 ANNNEXE

6.1 Approches explorées puis abandonnées

6.1.1 Suppression des duplicats

Comme l'avait illustré Dmitry Larko lors d'une conféfence à H2O.ai [4], le jeu de données contient certains duplicats, et des manipulations via les RLE peuvent permettre de les détecter (notamment en considérant la surface ou le positionnement des bateaux). Le risque de garder les duplicats est que le modèle overfit en apprenant plusieurs fois sur des instances identiques. Nous avions entrepris de supprimer les duplicats de bateaux, mais finalement l'impact sur la performance n'a pas été si net. Finalement, nous avons choisi de conserver les bateaux en doubles, pour ne pas enlourdir notre pipeline de traitement de données, nous avons choisi de ne privilégier cette approche.

Fig. 9 – *Exemple d'un bateau présent en double*

6.1.2 Combinaison de différentes fonctions de loss

Comme nous l'avions fait lors de notre PSC, nous avions initialement voulu combiner plusieurs fonctions de loss. C'est par exemple ce qu'avait fait Dmitry Larko [4] en combinant la distance de Jacquard et la Binary Cross Entropy. Nous avons effectué des tests, qui se sont révélés peu concluants avec différentes combinaison de fonction loss, comme par exemple :

$$\mathcal{L} = 2 * \text{IoU} + \text{Boundary Loss}$$

Finalement, nous avons choisi d'abandonner cette approche. Notons que [4] avait adopté la loss suivante, dont les résultats étaient meilleurs, mais que nous n'avons pas voulu implémenter :

$$\mathcal{L} = 5 * IoU + BCE$$