

OpenClassRooms
- Projet 8 -
Note technique

Raoof RACHIDI

SOMMAIRE

I. Préambule	3
II. Jeu de données	3
1. Génération des masques	3
2. Répartition des classes	4
III. Générateur de données	5
1. Principe	5
2. Augmentation de données	5
IV. Architecture des réseaux de neurones	6
1. VGG	6
2. FCN8	6
3. U-net	7
4. Métriques d'évaluation	9
V. Déploiement avec Azure	11
VI. Pistes d'amélioration	11

I. Préambule

Future Vision Transport est une entreprise qui conçoit des systèmes embarqués de vision par ordinateur pour les véhicules autonomes.

Je suis l'un des ingénieurs IA au sein de l'équipe R&D de cette entreprise. Mon équipe est composée d'ingénieurs aux profils variés. Chacun des membres de l'équipe est spécialisé sur une des parties du système embarqué de vision par ordinateur.

Mon rôle est de concevoir un premier modèle de segmentation d'images qui devra s'intégrer facilement dans la chaîne complète du système embarqué.

II. Jeu de données

Le jeu de données est divisé en 3 parties :

- le jeu d'entraînement, constitué de 2975 images et masques annotés ;
- le jeu de validation, constitué de 500 images et masques annotés ;
- le jeu de test, constitué de 1525 images dont les annotations ne sont pas divulguées à des fins d'évaluation.

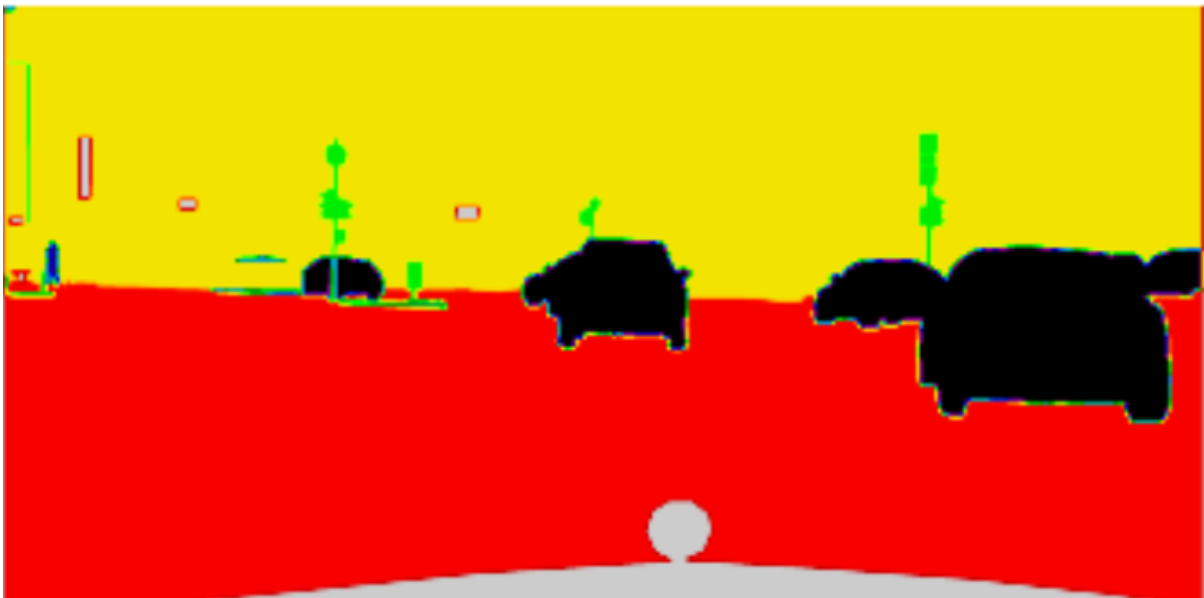
De plus, les images et masques sont contenus dans 2 dossiers :

- **leftImg8bit**, qui contient les dossiers train, val et test des images d'origines ;
- **gtFine**, qui contient les dossiers train, val et test des images labellisées avec 30 catégories.

Nous avons corrigé cette structure particulière, afin d'assurer l'intégrité des différents jeux de données, et de faciliter l'appel des images et des masques par nos modèles.

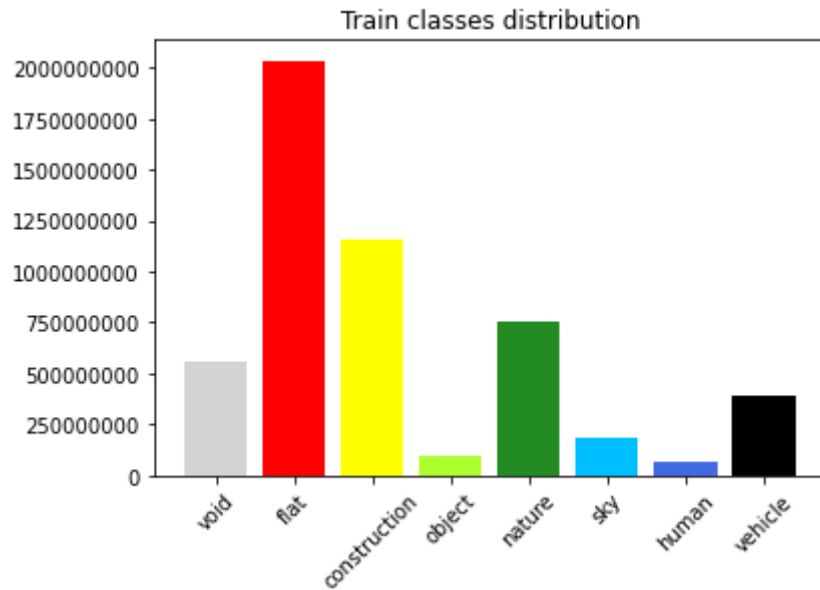
1. Génération des masques

Le masque étant initialement composé de nuances de gris sur 30 catégories, comme il a été demandé pour le projet, nous avons implémenté une fonction permettant de transposer ces 30 catégories en 8 catégories principales. Pour ça, nous avons utilisé la méthodologie d'un des créateurs du jeu de données : [Marius Cordts](#).



Une image et son masque après traitement

2. Répartition des classes



Le jeu de données contient des classes déséquilibrées, ce qui est une situation relativement fréquente sur les jeux de données. En effet, le masque **flat** apparaît par exemple plus de 2 fois plus fréquemment que le masque **nature**.

Cette information est importante lorsqu'il s'agit de mesurer la performance de nos modèles. En effet, il faudra le prendre en compte car le modèle prédira probablement plus souvent des classes sur-représentées que des classes sous-représentées.

III. Générateur de données

1. Principe

Afin de pouvoir manipuler un grand jeu de données ne pouvant être chargé en mémoire vive entièrement, une classe instanciant un objet de type générateur a été implémentée. Cet objet générateur permet de séparer le jeu de données en batchs. Les augmentations de données ainsi que le redimensionnement des images ont été intégrés à ce générateur pour le pré-traitement automatique.

2. Augmentation de données

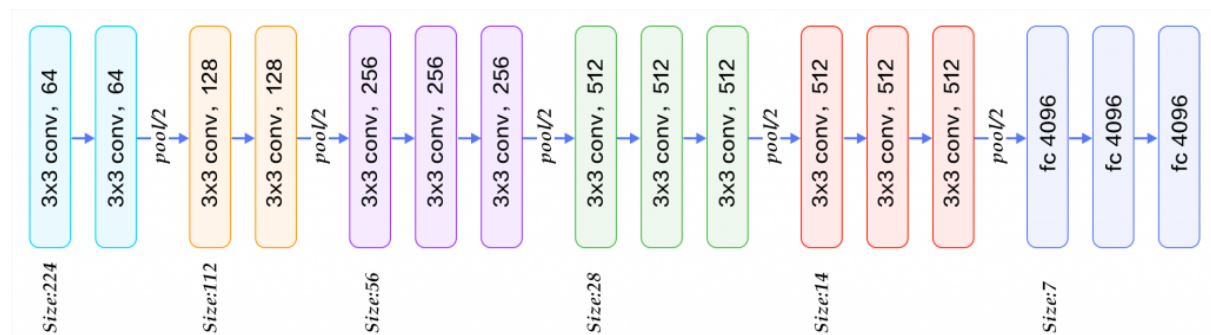
Plusieurs techniques d'augmentation de données ont été implémentées dans le but d'étendre le jeu d'entraînement. Dans notre cas, 4 types

d'augmentation ont été implémentés et testés : augmentation par ajout du bruit gaussien, augmentation par agrandissement aléatoire, augmentation par luminosité aléatoire et augmentation par inversement horizontal. Le but de l'augmentation est d'améliorer les performances de généralisation du modèle en l'exposant à une plus grande variété d'exemples d'entraînement.

IV. Architecture des réseaux de neurones

1. VGG

VGG16 est un modèle de réseau de neurones à convolution conçu par K. Simonyan et A. Zisserman. On retrouve les détails d'implémentation dans le document « [Very Deep Convolutional Networks for Large-Scale Image Recognition](#) ». Ce modèle atteint une précision de test de **92,7%** dans [ImageNet](#), qui regroupe plus de 14 millions d'images appartenant à 1000 classes. Pourquoi vgg-16 et bien tout simplement parce que ce réseau de neurones comprend 16 couches profondes :

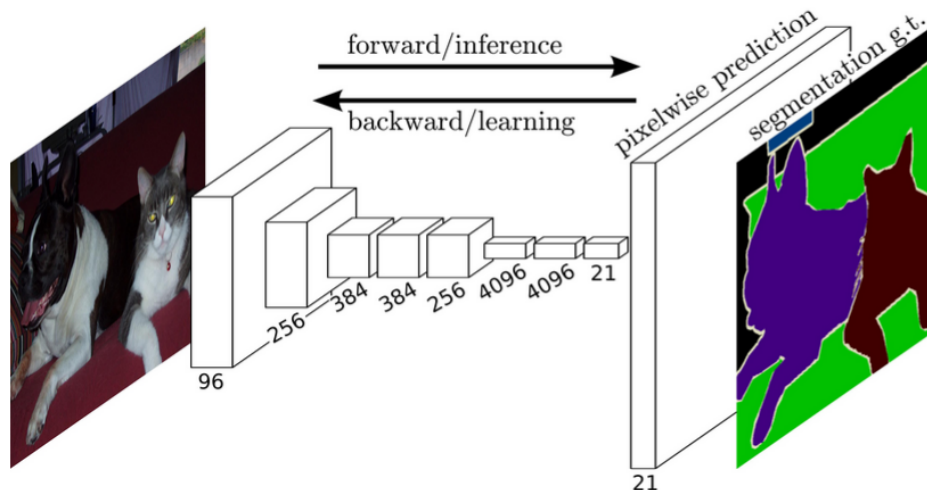


2. FCN8

FCN8 (Fully Convolutional Network) est une architecture utilisée principalement pour la segmentation sémantique. Il utilise uniquement des couches connectées localement, telles que la convolution, le pooling et le suréchantillonnage. Éviter l'utilisation de couches denses signifie moins de paramètres ce qui rend les réseaux plus rapides à former. Cela signifie également qu'un FCN peut fonctionner pour des tailles d'image variables étant donné que toutes les connexions sont locales.

Le réseau se compose d'un chemin de sous-échantillonnage, utilisé pour extraire et interpréter le contexte, et d'un chemin de suréchantillonnage, qui permet la localisation.

Les FCN utilisent également des connexions de saut pour récupérer les informations spatiales à grain fin perdues dans le chemin de sous-échantillonnage.



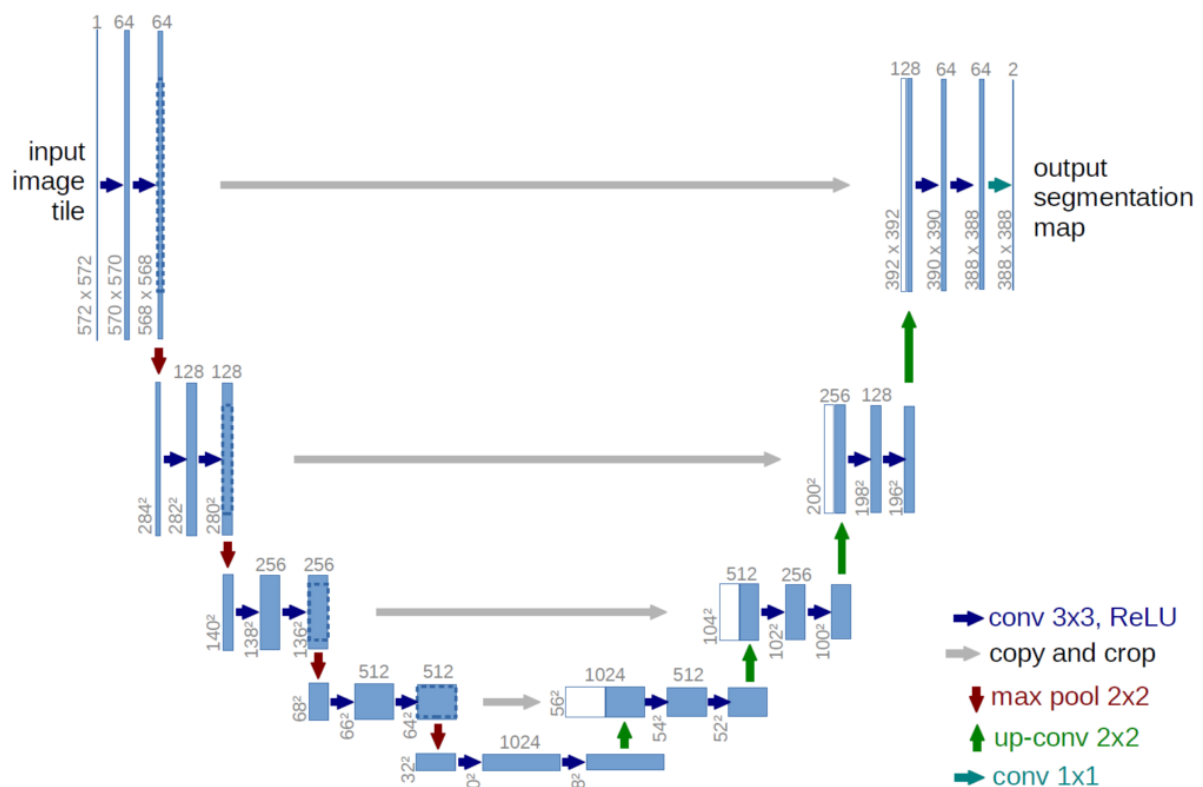
3. U-net

U-NET est un modèle de réseau de neurones dédié aux tâches de Vision par Ordinateur (Computer Vision) et plus particulièrement aux problèmes de Segmentation Sémantique. C'est ce modèle que nous avons retenu et que nous avons déployé.

L'architecture de **U-NET** est composée de deux chemins. Le premier est le chemin de contraction, aussi appelé encodeur. Il est utilisé pour capturer le contexte d'une image.

Il s'agit en fait d'un assemblage de couches de convolution et de couches de " max pooling " permettant de créer une carte de caractéristiques d'une image et de réduire sa taille pour diminuer le nombre de paramètres du réseau.

Le second chemin est celui de l'expansion symétrique, aussi appelé décodeur. Il permet aussi une localisation précise grâce à la convolution transposée.



Dans le domaine du Deep Learning, il est nécessaire d'utiliser de larges ensembles de données pour entraîner les modèles. Il peut être difficile d'assembler de tels volumes de données pour résoudre un problème de classification d'images, en termes de temps, de budget et de ressources hardware.

L'étiquetage des données requiert aussi l'expertise de plusieurs développeurs et ingénieurs. C'est particulièrement le cas pour des domaines hautement spécialisés comme les diagnostics médicaux.

U-NET permet de remédier à ces problèmes, puisqu'il s'avère efficace même avec un ensemble de données limité. Il offre aussi une précision supérieure aux modèles conventionnels.

Une architecture autoencoder classique réduit la taille des informations entrées, puis les couches suivantes. Le décodage commence ensuite, la représentation de caractéristiques linéaires est apprise et la taille

augmente progressivement. À la fin de cette architecture, la taille de sortie est égale à la taille d'entrée.

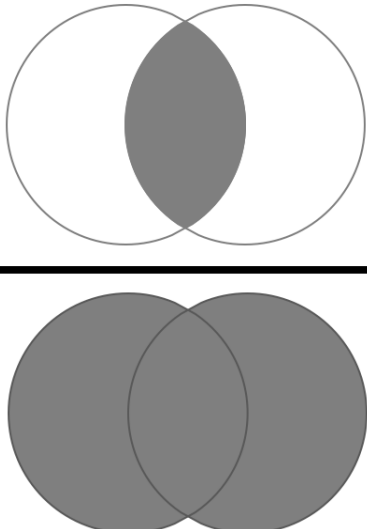
Une telle architecture est idéale pour préserver la taille initiale. Le problème est qu'elle compresse l'input de façon linéaire, ce qui empêche la transmission de la totalité des caractéristiques.

C'est là que **U-NET** tire son épingle du jeu grâce à son architecture en U. La déconvolution est effectuée du côté du décodeur, ce qui permet d'éviter le problème de goulot rencontré avec une architecture auto-encodeur et donc d'éviter la perte de caractéristiques.

4. Métriques d'évaluation

Pour évaluer nos modèles, nous avons utilisé 2 métriques d'évaluation :

- l'**Intersection-Over-Union** (IoU), également connu sous le nom d'indice Jaccard, est la zone de chevauchement entre la segmentation prédite et la segmentation réelle divisée par la zone d'union entre la segmentation prédite et la segmentation réelle. Cette métrique va de 0 à 1 (0 à 100 %), 0 signifiant aucun chevauchement et 1 signifiant une segmentation parfaitement superposée.

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


- le **coefficient de Dice** est similaire à l'indice de Jaccard (Intersection over Union). Ils sont positivement corrélés, ce qui signifie que si l'on dit que le modèle A est meilleur que le modèle B pour segmenter une image, alors l'autre dira la même chose. Comme l'IoU, ils vont tous deux de 0 à 1, 1 signifiant la plus grande similitude entre la prédiction et la vérité.

$$Dice = \frac{2 \times TP}{(TP + FP) + (TP + FN)}$$

Après avoir entraîné les différentes architectures, augmentation et hyperparamètres, nous obtenons les résultats suivants :

	U-net Dice loss	U-net Total loss	U-net Balanced loss	VGG-16 Dice loss	VGG-16 Total loss	VGG-16 Balanced loss
IoU	0.977563	0.887825	0.980365	0.860144	0.934942	0.990926
Dice coefficient	0.920600	0.963801	0.930598	0.473710	0.973045	0.971707
Training time	6224.462833	6494.797175	6495.223846	13741.885128	14259.938646	14977.959314
	U-net Augmented Dice loss	U-net Augmented Total loss	U-net Augmented Balanced loss	VGG-16 Augmented Dice loss	VGG-16 Augmented Total loss	VGG-16 Augmented Balanced loss
	0.946985	0.763196	0.944746	0.745015	0.813090	0.950786
	0.910430	0.962638	0.906425	0.452401	0.961118	0.917195
	26592.311259	25642.705065	28087.876644	74760.303561	73406.170201	74565.478434

Comme on pouvait s'y attendre, l'augmentation permet d'améliorer le score des modèles. Néanmoins, les conséquences en termes de temps d'entraînement ne nous a pas semblé justifier ce choix. Les résultats nous ont logiquement conduit à continuer le travail en sélectionnant

l'architecture **U-NET** avec la fonction de perte **balanced cross entropy** sans augmentation de données.

V. Déploiement avec Azure

Azure prend en charge le déploiement automatisé directement à partir de plusieurs sources. Pour notre part, nous avons choisi comme option GitHub. Nous y avons déployé une application Flask comme demandé dans les spécifications. Nous pourrions utiliser l'intégration continue puisque dès que nous pousserons des modifications sur notre répertoire GitHub, l'application web se mettra à jour automatiquement.

Le fournisseur de build GitHub Actions est une option pour CI/CD à partir de GitHub. Il dépose un fichier de workflow GitHub Actions dans notre référentiel GitHub pour gérer les tâches de génération et de déploiement vers App Service. L'application Flask va se charger de sérialiser nos images en JSON, puis il va les envoyer au modèle afin d'obtenir la prédiction. Enfin, le modèle va nous renvoyer le masque prédit et l'application va désérialiser le masque.

VI. Pistes d'amélioration

Nous n'avons essayé que trois architectures, d'autres peuvent être implémentées telles que Segnet, PSPnet et bien d'autres encore. Chaque architecture présente ses points forts et ses points faibles et sera plus ou moins bien adaptée à notre problématique.

Lors du benchmark des différents modèles, nous avons comparé les résultats avec différents hyperparamètres. Cependant une investigation plus profonde permettrait certainement d'améliorer encore nos résultats.