Segmentation sémantique basée sur CITYSCAPES DATASET

Letremble Emmanuel

https://github.com/Valkea/OC_AI_08

Résumé : dans ce document, nous allons passer en revue et expliquer les étapes qui nous ont menées au modèle de segmentation sémantique actuellement déployé pour notre projet de voiture autonome « Futur Vision Transport ».

1. Introduction

Dans le cadre de notre projet de « conception d'une voiture autonome », il m'a été demandé de prendre en charge la **segmentation des images** qui est l'une des multiples parties du système embarqué de vision par ordinateur. L'objectif étant de fournir pour chaque image, un « mask » exploitable par le **système de décision** succédant à l'étape de segmentation.

Une segmentation tente de diviser une image en plusieurs zones visuellement significatives ou intéressantes permettant la compréhension visuelle et l'analyse de l'image.

Mais il existe plusieurs types de segmentation applicables sur ce genre de projet *(voir* Fig-1*)* :

- La **segmentation sémantique** *(qui classe chaque pixel dans l'une des N catégories)*
- La **segmentation d'instances** *(qui détecte les instances et délimite leurs frontières)*
- La **segmentation panoptique** *(qui combine les deux précédent types de segmentation)*

Or, pour répondre aux besoins du système de décision, il nous faut classer chaque pixel dans l'une des 8 catégories fournies (*void, flat, sky, construction, object, nature, human, vehicule).*

J'ai donc choisi la **segmentation sémantique**.

Dans la section 2, nous explorerons plus avant la segmentation sémantique en faisant un tour d'horizon des dernières avancées *(état de l'art)* dans ce domaine. Puis dans la section 3, je présenterai le protocole expérimental mis en place pour produire et choisir nos modèles intermédiaires. En section 4, je partagerai les résultats des différentes étapes de ce processus de sélection avant d'en tirer quelques leçons et conclusions en section 5. Enfin, en section 6, je présenterai les axes d'amélioration que nous pouvons *(et devrions)* encore explorer pour affiner notre projet.

2. État de l'art (state of the art)

Cette section est basée sur les articles suivant : *« An Overview of State of the Art DNNs* [1]*»*, *« Semantic Segmentation on Cityscapes val* [2]*»* et *« Review the state-of-the-art technologies of semantic segmentation based on deep learning* [3]*»* qui m'ont servi de référence.

Dans un premier temps, je vais vous présenter l'état de l'art des méthodes de segmentation sémantique basées sur des architectures de type Visual Transformers ou ViT, puis ensuite l'état de l'art pour les architectures de type Convolutional Neural Networks ou CNN.

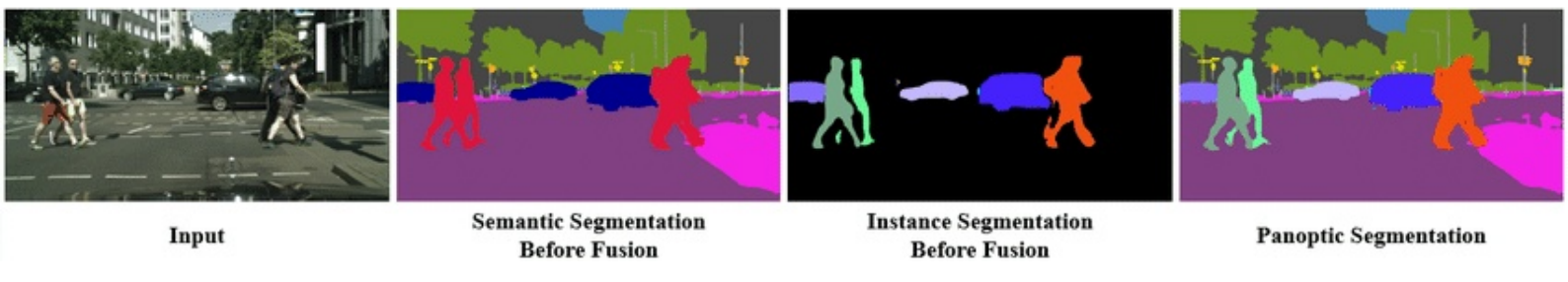


Fig-1*: Illustration des différents types de segmentation*

[ViT] SegFormer [4 et 5]

Cette architecture de segmentation sémantique simple, efficace et puissante, se compose de Transformers auxquels ont été ajoutés des décodeurs légers composés de perceptrons multicouche (MLP) *(voir* Fig-3*)*.

Cette approche offre deux atouts attrayants :

1) SegFormer propose un nouvel encodeur de Transformer avec une structure hiérarchique qui produit des features à différentes échelles. Avec cette structure, l'encodage positionnel n'est plus requis, ce qui évite les problèmes d'interpolation de cet encodage lorsqu'il y a un changement de résolution de l'image à inférer. *(problème que j'ai pu constater sur le modèle qui a été déployé pour notre projet)*.

2) SegFormer évite les décodeurs complexes. Le décodeur MLP proposé agrège les informations provenant de différentes couches et combine ainsi l'attention locale et l'attention globale pour obtenir des représentations puissantes. Cette conception simple et légère est la clé d'une segmentation efficace sur les Transformers.

Une série de modèles allant de SegFormer-B0 à SegFormer-B5, ont été testés et atteignent des performances remarquable.

Par exemple, SegFormer-B5 atteint **84.0 %** mIoU sur le set de validation de Cityscapes.

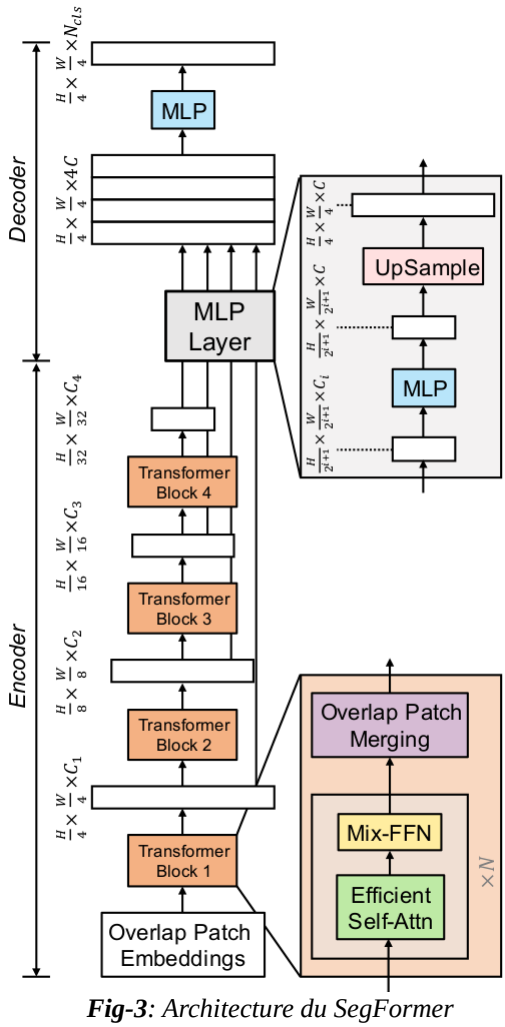


***Fig-3**: Architecture du SegFormer*

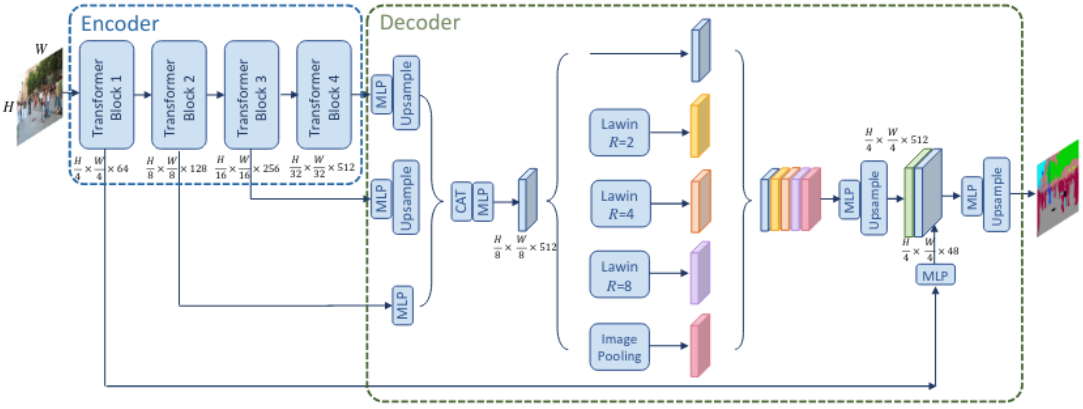
[ViT] Lawin Transformer [6]

Lawin Transformer est composé d'un encodeur *hierachical vision transformer* (HVT) et d'un décodeur *LawinASPP (voir* Fig-2*)*.

Comme nous l'avons vu précédemment, les multi-scale features sont la clé de la réussite des Transformers dans le domaine de la segmentation sémantique. Ce ViT s'appuie donc sur le même principe sous-jacent que le SegFormer, mais y ajoutant quelques blocs supplémentaires mettant en place un mécanisme d'attention plus puissant et permettant en particulier d'obtenir une plus grande attention locale au prix d'une surcharge de calcul minime.

L'une des versions proposées atteint **84.4 %** mIoU sur le set de test de Cityscapes.

***Fig-2**: Architecture du Lawin Transformer*



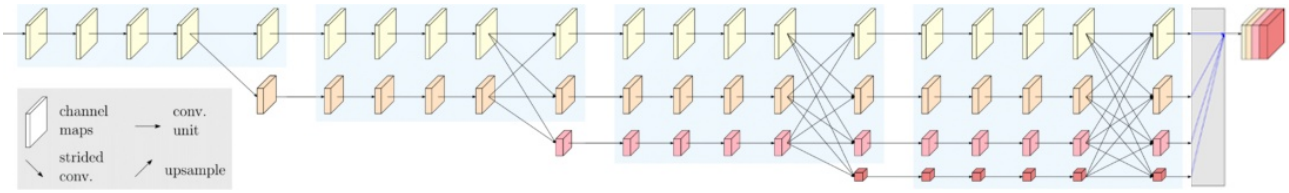


***Fig-4**: Architecture du HRNet configurée pour une segmentation sémantique*

[CNN] HRNetV2 OCR [7]

Le **HRNet** *(high-resolution network)*, est un modèle qui maintient des représentations de haute résolution tout au long du processus *(contrairement aux architectures que j'ai testées et qui down-sample puis up-sample)*.

L'idée (*voir* Fig-4*)* est d'entraîner en parallèle des séries de convolutions de divers niveaux de résolutions, que l'on combine entre chaque gros bloc de l'architecture pour obtenir des représentations plus qualitatives de chaque résolution. Puis à la fin, on combine toutes ces représentations en fonction de l'objectif *(HRNetV1: pose estimation, V2: semantic seg. V3: instance seg. or object detection)*.

Les représentations à haute résolution apprises à partir de HRNet sont sémantiquement fortes, et spatialement précises, pour 2 raisons :

1) HRNet connecte les flux de convolution de toutes les résolutions en parallèle plutôt qu'en série. Se faisant, cette approche est capable de garder la haute résolution tout du long, au lieu de la récupérer à partir de la basse résolution.

2) On répète régulièrement les fusions multi-résolution pour renforcer les représentations hautes résolution à l'aide des représentations basses résolution, et vice-versa. De cette façon, toutes les représentations de haute à basse résolution sont sémantiquement plus fortes.

L'**OCR** *(object-contextual representations)* est un algorithme que l'on applique après une segmentation sémantique pour en améliorer les performances.

Il consiste en une agrégation pondérée de toutes les régions de chaque objet, avec des poids calculés en fonction des relations entre les pixels et les régions de ces objets.

Cet ensemble HRNetV2 + OCR atteint **86.3** % mIoU sur le validation-set de Cityscapes.

[CNN] HRNetV2 OCR + PSA [8]

Le bloc **PSA** *(Polarized Self-Attention)* intègre deux mécanismes essentiels pour une régression pixel à pixel de très haute qualité :

1) Le **filtrage polarisé** *(voir* Fig-6*)* qui maintient une résolution interne élevée dans le calcul de l'attention des canaux *(3 en RGB...)* et de l'attention spatiale tout en réduisant la taille des tensors d'entrée.

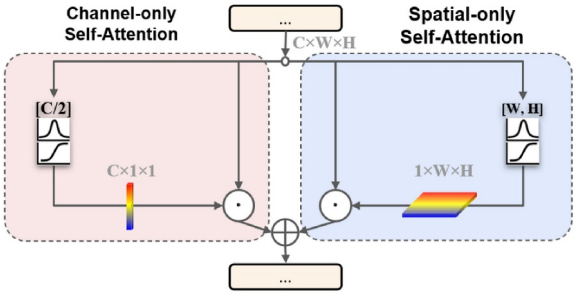


***Fig-6**: architecture du bloc de filtrage polarisé*

2) La prise en compte de la **non-linéarité** en s'adaptant directement à la distribution de sortie d'une régression pixel à pixel typique, telle que la distribution binomiale.

Cette ultime combinaison HRNetV2 OCR + PSA *(qui comme on peut le voir sur la* Fig-5 *est l'actuel état de l'art)* atteint **86.93** % mIoU sur le set de validation de Cityscapes.

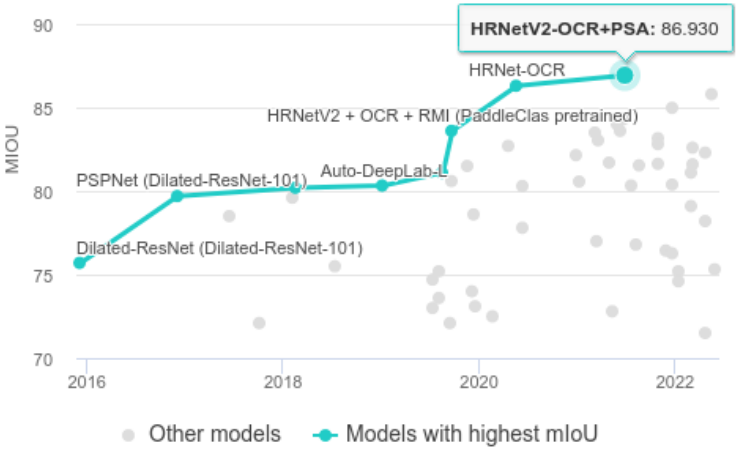


***Fig-5**: Évolution de l'état de l'art pour la segmentation sémantique*

Enfin, les **Decinets**[9] peuvent parfois surpasser les modèles présentés, mais ne sont pas libres.

3. Protocole expérimental

Après ce bref survol de l'état de l'art, et avant de vous présenter les résultats obtenus, il me faut d'abord vous présenter les choix qui ont menés au modèle déployé.

3.1 Jeu de données

Comme nous ne disposons pas encore d'un jeu de données interne, j'ai dû utiliser celui proposé en libre accès par **Cityscapes**[10].

C'est un jeu de données spécialisé dans les scènes urbaines dont les photographies ont été soigneusement collectées dans 50 villes, en journée, durant 3 saisons et dans de moyennes ou bonnes conditions météo.

Les 25 000 photos proposées sont pourvues de « masks » pour les découpages en instances et en catégories *(il y en a 30)*. 5000 de ces masks sont finement annotés, tandis que les 20 000 restantes sont beaucoup plus approximatif.

Enfin, les photographies et les masks qui leur sont associés sont tous proposés en résolution 2048 x 1024.

Pour notre projet, il a été décidé de travailler avec les photographies disposant d'un mask de catégories finement annoté. Mais le test-set n'étant pas disponible car gardé pour les tests officiels, j'ai pu disposer d'un train-set de 2975 fichiers et d'un validation-set de 500 fichiers. Soit 3475 fichiers au total.

3.2 Préparatifs & Pré-traitements

Tous les préparatifs présentés ci-dessous ont été réalisés durant la même opération.

1) Les « masks » de catégories étant prévus pour 30 catégories, il a fallu les convertir pour qu'ils représentent les 8 catégories principales. Chacune des 30 catégories a donc été affectée à l'une des 8 nouvelles catégories et les fichiers convertis en conséquence.

2) Comme il est difficile avec des moyens limités et de grandes images (2048 x 1024) de tester de nombreux modèles, j'ai préparé deux versions réduites en taille qui m'ont permis d'accélérer le processus ; un set 256x128 pour itérer rapidement durant la phase de recherche et un set 512x256 pour la phase finale.

3) Enfin, pour simplifier le travail avec le DataGenerator, les fichiers photos et les masks associés ont été renommés et rassemblés dans un même répertoire *(tout en conservant les dossiers train et val séparés)*. Les noms des photos ont été changés en {ID}.png et ceux des masks en {ID}_labels.png.

Le reste des pré-traitements ont été appliqués au moment de l'entraînement ou de l'inférence, car dépendant du modèle testé ou nécessitant d'être appliqué sur les nouvelles images.

1) Les photographies ont été normalisées en divisant les valeurs RGB par 255.0

2) Lorsque nécessaire, les pré-processing liés aux backbones utilisés ont été appliqués.

3.3 DataGenerator

Les fichiers image étant gros par essence, il est difficile de tous les charger en mémoire pour entraîner nos modèles. J'ai donc mis en place un data-generator adapté à notre jeu de données pour éviter ce problème.

La classe DataGenerator crée, distribue donc les images par batch de taille paramétrable et en profite pour appliquer les éventuels pré-processing ou augmentations de données qui lui ont été transmis.

3.4 Logging Tensorboard

Pour garder un œil sur l'évolution d'un modèle en cours d'entraînement, puis ensuite comparer le comparer avec les différents modèles testés, j'ai choisi d'utiliser *TensorBoard*.

Le callback Keras pour Tensorboard le rend très facile à intégrer, et l'interface existante est dynamique, agréable et pratique.

Enfin, comme les fichiers générés sont légers, j'ai sauvegardé les logs de chaque modèle sur le git-hub du projet… De cette façon, il est facile de les récupérer pour la suite du projet.

3.5 Learning rate & ReduceLROnPlateau

Sur ce projet, j'ai systématiquement utilisé une fonction d'optimisation Adam avec un learning rate initial de 0,0005.

J'ai également appliqué sur chaque modèle un *ReduceLROnPlateau* basé sur *val_loss* avec un facteur 0.2 et une patience de 4, qui a permis d'affiner les modèles en fin d'entraînement.

3.6 Epochs & EarlyStopping

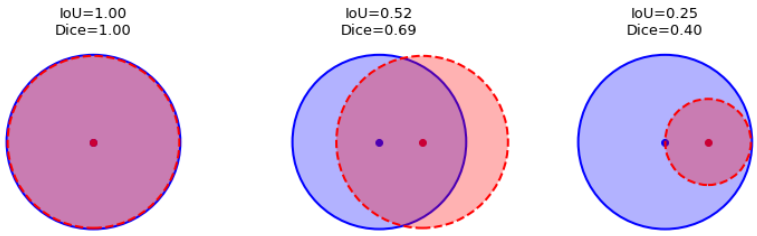
Tous les modèles ont été paramétrés pour exécuter 100 epochs par défaut.

Et chaque fois, un callback *EarlyStopping* basé sur la minimisation du *val_loss* a été utilisé afin de déterminer le meilleur moment pour arrêter l'entraînement *(avec une patience 2 x la patience du ReduceLROnPlateau + 1 = 9).*

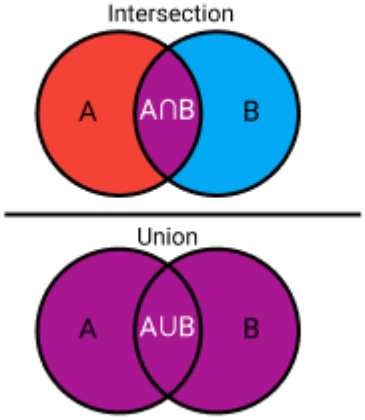
3.7 Métriques

Dans le cadre d'un projet de segmentation *(c'est à dire une classification des pixels)*, les traditionnelles métriques de classification ne sont pas adaptées.

Il convenait donc d'utiliser l'***IoU*** et le ***Dice*** pour correctement évaluer la progression.

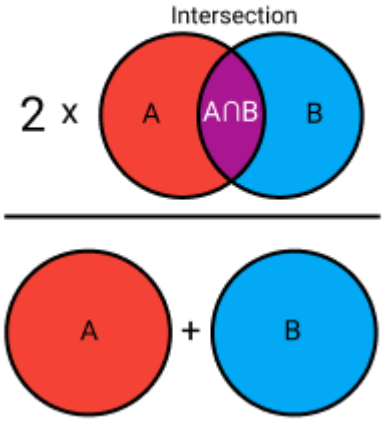


1) L'**IoU** [11] *(Intersection over Union)* qui est aussi appelée *Jaccard coefficient* permet d'évaluer de 0 à 1 la qualité de segmentation d'une classe donnée avec la formule suivante :



Mais si l'on évalue chaque classe et que l'on en fait la moyenne, on obtient le ***mIoU*** *(mean...)*.

2) Le ***coefficient Sørensen–Dice*** [12] est une métrique allant de 0 à 1, permettant d'évaluer la qualité de segmentation d'une classe, mais avec laquelle les erreurs sont moins fortement sanctionnées qu'avec l'IoU. Voici sa formule :



Pour comprendre la différence entre les deux scores, on peut se dire que le **Dice** représente l'évaluation moyenne d'un modèle, tandis que l'**IoU** correspond davantage à l'évaluation de la performance la plus défavorable d'un modèle.

J'ai choisi d'utiliser le mIoU comme métrique principale *(tout en gardant un œil sur le Dice)* pour deux raisons principales :

1) sur ce genre de projet, il me semble plus adapté d'utiliser un score qui sanctionne davantage les erreurs pour éviter les scores trop optimistes.

2) c'est la métrique de référence utilisée pour comparer les modèles de l'état de l'art.

3.8 Modèle de référence

Pour nous assurer que notre projet évolue dans la bonne direction, il nous a d'abord fallu créer un modèle de référence avec lequel comparer le reste de notre travail.

Pour ce faire, j'ai décidé d'utiliser le modèle simple proposé dans l'exemple de la librairie Keras « *Image segmentation with a U-Net-like architecture* [13] ». Ce choix me garantissant d'avoir un modèle fonctionnel *(c'est Keras)*, rapide et modérément performant *(car simple)*.

3.9 Augmentation des données

Une fois notre modèle baseline obtenu, l'étape suivante à consisté à diversifier les exemples utilisés pour entraîner le modèle, avec l'espoir d'en améliorer les performances.

J'ai donc utilisé la lib. ***Albumentation*** [14] pour préparer trois ensembles de transformations qui pouvaient peut-être rendre notre modèle de base plus sensible aux features importantes.

1) Un premier modèle à donc été entraîné avec des modifications basées sur mon intuition.

```
transform1 = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.Rotate(15, p=0.5),
    A.RandomBrightnessContrast(p=0.2),
])
```

2) Puis un second modèle a été entraîné avec des modifications recommandées par un collègue *(mon mentor)*.

```
transform2 = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.OneOf([
        A.RandomBrightnessContrast(
            brightness_limit=[-0.2, 0.2],
            contrast_limit=0.2,
            p=0.5,
        ),
        A.RandomGamma(p=0.5),
    ],p=0.8,),
])
```

3) Enfin, j'ai saisi cette opportunité pour essayer la lib. ***AutoAlbument*** [15] qui permet théoriquement de trouver un ensemble optimal de transformations permettant de maximiser les résultats pour un type d'opération précis sur un jeu de données précis.

Il a fallu réécrire le *DataGenerator en pytorch*, configurer le champ de recherche et laisser *AutoAlbument* travailler pour fournir un fichier *policy.json* contenant les dites transformations.

```
transformAuto = A.load("autoalbument/policy_28.json")
```

On verra dans la section 4, que les résultats ne sont pas forcément ceux attendus...

3.10 Recherche d'une fonction de perte

L'étape suivante consistait à optimiser l'hyper-paramètre qu'est la fonction de perte utilisée pour entraîner les modèles.

Jusqu'à cette étape, la fonction de perte utilisée était la ***Categorical Cross Entropy***. C'est une fonction de perte parfaitement adaptée aux tâches de classification multi-classes, mais pas nécessairement celle qui est la plus optimale pour une segmentation sémantique.

J'ai donc décidé d'essayer un certain nombre de fonctions de pertes recommandées dans la littérature et les lib. spécialisées.

1) ***DiceLoss*** : l'IoU n'étant pas dérivable il ne peut pas être utilisé comme fonction de perte, mais ce n'est pas du Dice que je me suis empressé d'essayer.

J'ai tout d'abord utilisé un *DiceLoss simple*, puis un *DiceLoss pondéré* selon les classes *(avec cette approche, on peut par exemple donner plus de poids aux piétons qu'au reste)*.

2) ***FocalLoss*** [16] : c'est une version améliorée de la *Cross-Entropy* qui tente de pallier le problème de déséquilibre des classes. L'idée étant d'attribuer plus de poids aux exemples difficiles ou facilement mal classés *(l'arrière-plan avec une texture bruyante ou un objet partiel)* et de réduire le poids des exemples faciles *(les objets d'arrière-plan simples)*.

Certaines catégories comme les piétons étant nécessairement moins présentes que certaines autres, il m'a semblé judicieux de l'essayer.

J'ai tout d'abord utilisé une *FocalLoss avec un gamma à 0.5* qui pénalisait modérément les exemples difficiles, puis une *FocalLoss avec un gamma à 1.0* qui pénalisait un peu plus sévèrement ces mêmes exemples difficiles.

3) ***Mix Categorical Cross Entropy + Dice*** : pour finir, j'ai essayé de combiner les deux fonctions de pertes qui m'avaient donné les meilleurs résultats.

D'abord avec un poids équivalent. Puis ensuite avec deux fois plus de poids sur le *Dice* que sur la *Categorical Cross Entropy*...

3.11 Recherche d'une architecture

Après avoir trouvé des transformations et une fonction de perte améliorant les résultats, il a ensuite fallu évaluer de nouvelles architectures potentiellement performante sur les tâches de segmentation sémantique.

Il y a beaucoup d'architectures spécialement conçues pour cette problématique. Et le fait que certaines d'entre elles permettent aussi de changer le backbone *(le CNN utilisé comme encoder pour l'extraction de features)* rendait impossible de toutes les tester dans un délai raisonnable et avec un accès limité à des GPU.

J'ai donc choisi de rester sur des classiques du domaine comme le ***U-Net***, le ***LinkNet*** ou le ***FPN*** avec les backbones suivants : ***VGG19***, ***ResNet152***, ***ResNeXt101*** et ***EfficientNetB7***.

1) ***U-Net*** [17]: c'est un CNN développé pour la segmentation d'images biomédicales, qui a la particularité d'être entièrement convolutionnel *(et qui donc n'utilise pas de Dense layers)*.

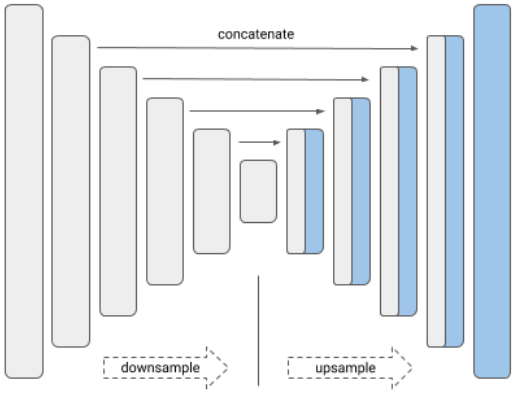


***Fig-7**: Architecture simplifiée d'un U-Net*

Son architecture en U permet de connecter les différents étages de l'encodeur aux mêmes étages du décodeur pour créer un mécanisme d'attention locale et globale plus à même de restituer précisément la position de chaque pixel de l'image d'origine après classification.

2) ***LinkNet*** : c'est une variante du *U-Net* qui se différencie tout de même sur deux points :

- l'encoder est idéalement basé sur un backbone de type *res* : *ResNet, ResNeXt*...

- la transmission d'informations des étages d'encodages aux étages de décodages ne se fait plus par concaténation, mais par addition.

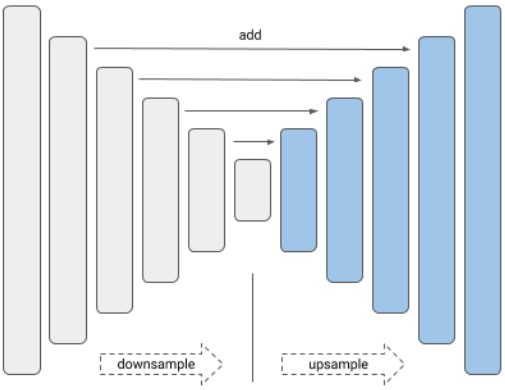


***Fig-8**: Architecture simplifiée du LinkNet*

3) ***FPN*** *(Feature Pyramid Networks)* : c'est en fait une implémentation alternative du concept sous-jacent des U-Nets.

L'une des principales différences étant que les *U-nets* produisent en sortie une image de même taille que celle fournie en entrée, alors que les *FPN* produisent une image de sortie pour chaque étage de la pyramide qui sont ensuite concaténées *(voir* Fig-9*)*.

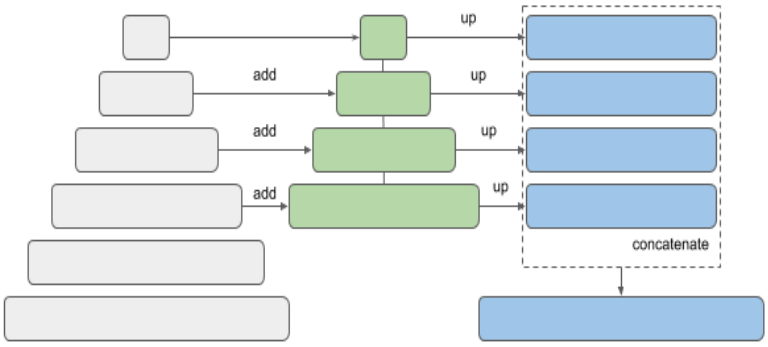


***Fig-9**: Architecture simplifiée d'un FPN*

Mais si les concepts sous-jacents sont les mêmes, on verra que dans les faits, les résultats peuvent être assez différents.

3.12 Augmentation de la résolution

Enfin, pour vérifier l'influence de la taille des images sur la qualité de segmentation de mon modèle, j'ai décidé de ré-entraîner le meilleur modèle obtenu avec un jeu de données préparé en 512x256 au lieu de 256x124.

Et c'est le modèle obtenu à cette étape qui a été déployé dans le cloud pour le moment.

4. Résultats des expériences

Présentons maintenant les résultats obtenus au cours des différentes étapes de ce projet.

4.1 Modèle de référence

Dans un premier temps, j'ai établis une baseline dont on peut voir les scores en Fig-10 et un exemple d'inférence que l'on peut comparer avec l'image originale et le mask cityscapes en Fig-11.

	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net_baseline	0.552171	0.667224	517.575655	1.851263	0.895139

***Fig-10**: Résultats du modèle servant de baseline*

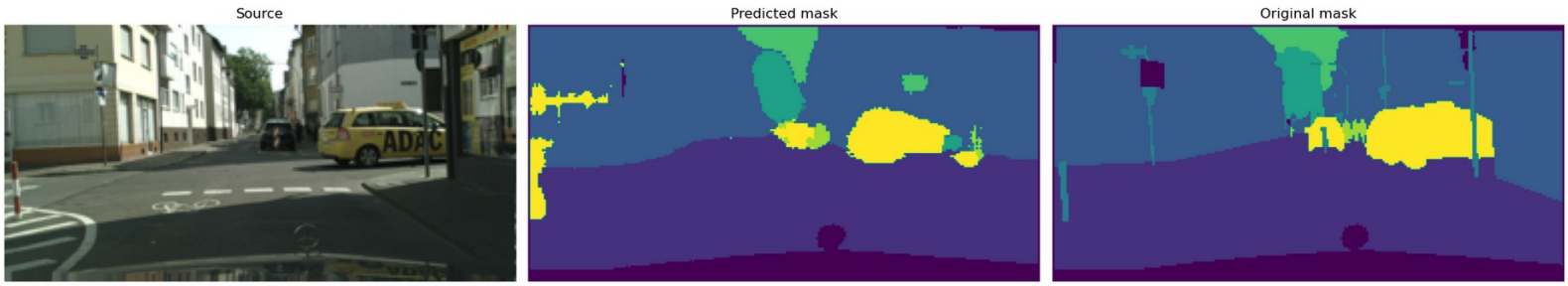


***Fig-11**: Comparaison du mask prédit par le modèle baseline avec le mask de référence*

Le modèle produit un résultat honorable *(mIoU de 0.55)* mais on constate facilement que les segmentations sont approximatives et que plusieurs erreurs sont visibles *(en particulier à gauche)*.

4.2 Augmentation des données

J'ai ensuite tenté d'améliorer le modèle avec diverses transformations de données préparées manuellement ou automatiquement avec *Albumentation* et *AutoAlbument*.
La Fig-12 ci-dessous permet de comparer avec le modèle de référence, les trois nouveaux modèles.

	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net_baseline	0.552171	0.667224	517.575655	1.851263	0.895139
U-Net_baseline_with_data_augmentation_1	0.564558	0.680571	1029.694759	2.369722	0.423762
U-Net_baseline_with_data_augmentation_2	0.559365	0.674603	714.297488	2.345822	0.924520
U-Net_baseline_with_data_augmentation_3	0.117844	0.197346	284.685457	2.193978	0.414083

***Fig-12**: Résultats comparatif des différentes augmentations de données*

Les modèles ***_data_augmentation_1 et _data_augmentation_2*** qui utilisent les transformations préparées manuellement, surpassent la baseline d'environ 1 % *(c'est peu, mais c'est déjà ça)*.

Cependant, le modèle ***_data_augmentation_3*** qui a utilisé les transformations calculées par *AutoAlbument* fait une très mauvaise performance ! *(peut être un mauvais réglage ?)*

Bien que le premier modèle soit le meilleur, j'ai pris le parti d'utiliser le second pour la suite du projet car présentant des scores proches tout en offrant un temps d'entraînement ~30 % plus bas.

4.3 Recherche d'une fonction de pertes

Puis, j'ai entrepris de chercher une fonction de perte plus adaptée à notre problématique.
La Fig-13 ci dessous présente les résultats obtenus avec les 6 fonctions de pertes testées.

	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net_baseline_with_data_augmentation_2_diceLoss	0.631681	0.743959	843.582155	2.138717	0.903396
U-Net_baseline_with_data_augmentation_2_diceLossWeighted	0.612228	0.728071	697.217847	2.189904	0.415433
U-Net_baseline_with_data_augmentation_2_focalLoss_0.25_0.5	0.486711	0.612405	326.424069	2.646039	0.425659
U-Net_baseline_with_data_augmentation_2_focalLoss_0.25_1.0	0.454141	0.585703	318.810731	2.206589	0.418523
U-Net_baseline_with_data_augmentation_2_cceDiceLoss_50_50	0.622789	0.738499	899.124581	2.216438	0.415926
U-Net_baseline_with_data_augmentation_2_cceDiceLoss_0.5_1.0	0.628407	0.742861	995.207425	2.260605	0.444320

***Fig-13**: Résultats comparatifs des différentes fonctions de pertes*

On constate que les modèles ***_focal_Loss_*** n'ont pas donnés de bons résultats, puisqu'ils sont en dessous des scores du modèle de référence *(il faudrait essayer avec un gamma de 3 ou 4).*

Cependant, le reste des modèles ont largement dépassé les scores du meilleur modèle obtenu *(+6%)*

*Et parmi tous ces candidats, c'est finalement le modèle utilisant une **_diceLoss** simple qui s'impose.*

4.4 Recherche d'une architecture

La fonction de perte idéale étant identifiée, j'ai cherché une architecture plus adaptée à notre problématique. La Fig-14 ci dessous présente les résultats obtenus avec les 9 architectures testées.

	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net-resetnet152_with_data_augmentation_2_diceLoss	0.722085	0.822108	3538.612212	6.312516	3.414692
U-Net-vgg19_with_data_augmentation_2_diceLoss	0.734644	0.832938	1932.282903	3.101655	0.048437
U-Net-resnext101_with_data_augmentation_2_diceLoss	0.720611	0.820914	3738.743413	5.380407	0.130063
FPN-resnet152_with_data_augmentation_2_diceLoss	0.738715	0.834818	4316.767552	5.874191	0.069000
FPN-efficientnetb7_with_data_augmentation_2_diceLoss	0.768454	0.858004	5930.868126	7.299468	0.082367
FPN-vgg19_with_data_augmentation_2_diceLoss	0.744457	0.839688	2601.794258	5.406630	0.721739
U-Net-custom_with_data_augmentation_2_diceLoss	0.541770	0.657728	820.391873	1.998511	0.646094
LinkNet-resnet152_with_data_augmentation_2_diceLoss	0.707575	0.809275	2433.885920	6.768224	3.061178
LinkNet-efficientnetb7_with_data_augmentation_2_diceLoss	0.751172	0.844442	4191.200948	10.014075	5.479347

***Fig-14**: Résultats comparatifs des différentes architectures*

Toutes ces architectures *(à l'exception du U-Net-custom qui était un test de U-Net Mini)* ont donné d'excellents résultats en comparaison de ce que nous avions précédemment *(de +7 % à +13 %)* !

Parmi les backbones utilisés, c'est l'***efficientNetB7*** qui s'est le plus démarqué le plus en donnant les meilleurs résultats sur *FPN* et sur *LinkNet (je n'ai visiblement pas pensé à le tester sur un U-Net…).*

Et au final, c'est donc une architecture ***FPN*** basée sur un backbone ***efficientNetB7*** qui a été retenue.

4.5 Modèle final

Pour finir, j'ai entraîné une seconde fois notre meilleur modèle avec un jeu de données proposant des images de plus grande résolution *(512x256 contre 256x128)* afin de vérifier l'apport potentiel de l'augmentation de la résolution.

On peut voir les scores de ce modèle final en Fig-15 et un exemple d'inférence que l'on peut comparer avec l'image originale et le mask cityscapes en Fig-16.

	IoU	Dice	training_time	inference_time_500	inference_time_1
FPN-efficientnetb7_with_data_augmentation_2_diceLoss_512x256	0.811932	0.887532	22570.196906	25.308100	0.258906

***Fig-15**: Résultats du modèle déployé dans le cloud*

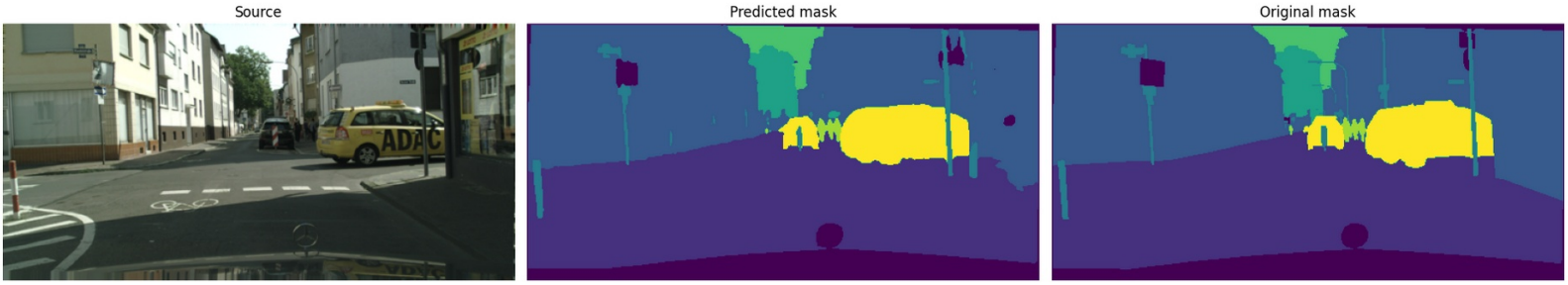


***Fig-16**: Comparaison du mask prédit par le modèle déployé avec le mask de référence*

On constate une augmentation du mIoU d'environ 4.3 % qui se traduit par des segments plus nets que ce que nous avions pu voir. Il y a encore des erreurs visibles, mais elles semblent minimes.

5. Conclusions

Avec cet ultime modèle à ***81.19 % de mIoU***, il reste une belle marge de progression avant d'atteindre le niveau du *HRNetV2 OCR + PSA et ses 86.93 % mIoU*, mais on constate d'ores et déjà en Fig-17 et Fig-18 une très nette évolution par rapport au modèle qui nous a servi de baseline.

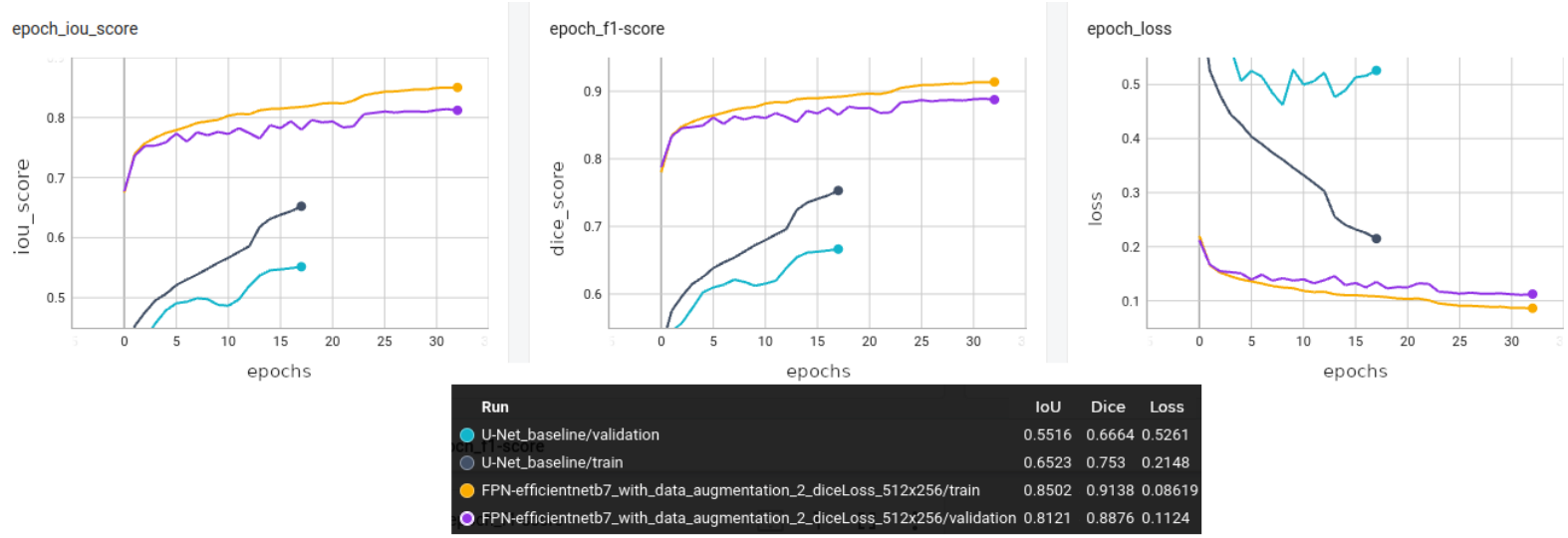


***Fig-17**: Évolution des métriques entre le modèle baseline et le modèle déployé*

Bien entendu, le modèle est imparfait et le protocole expérimental peut certainement être amélioré *(j'attends vos retours chers collègues)*, mais je pense que nous avons un premier modèle tout à fait honorable qui ne demande qu'a être amélioré en appliquant les idées de la section 6.

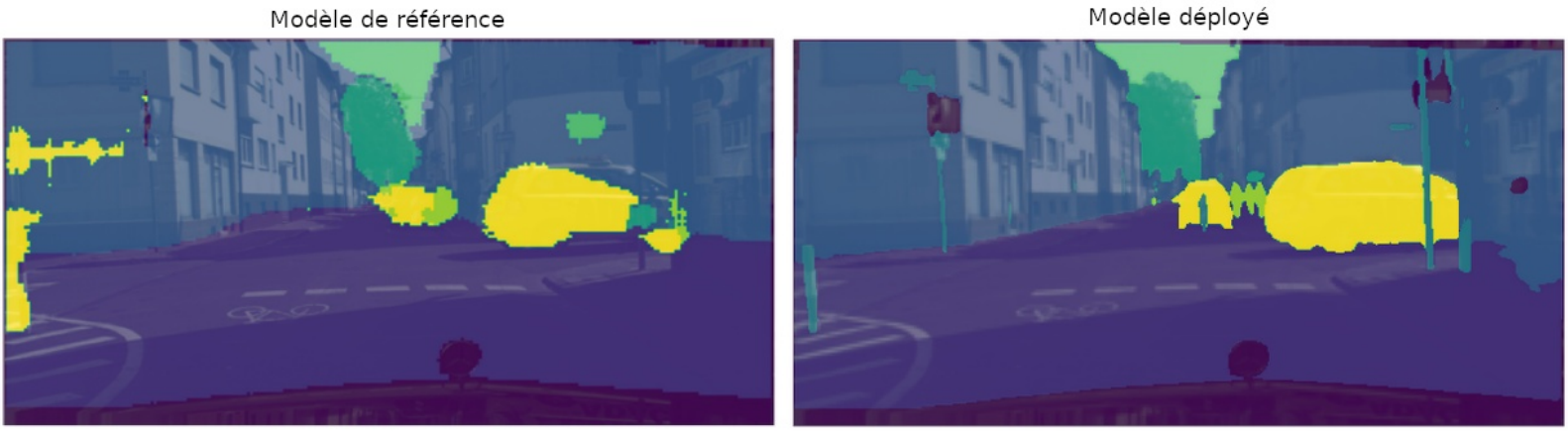


***Fig-18**: Comparaison des masks inférés pour le modèle baseline et le modèle final*

6. Axes d'amélioration

Le modèle fonctionne très raisonnablement, mais il existe cependant encore de nombreuses pistes à explorer pour le rendre plus pertinent.

Résolution des images

Il n'a pas été possible d'entraîner un modèle avec une plus haute résolution des fichiers.

Mais dans la mesure où nous avons démontré une nette progression *(+4.3%)* entre le modèle final entraîné sur du 248x128 et le ce même modèle entraîné en 512x256, il semble raisonnable de supposer que l'on peut encore améliorer le modèle avec un jeu de données en 1024x512 ou en 2028x1024.

Outils d'optimisation

Nous pourrions également utiliser des outils d'optimisation sur notre modèle, comme par exemple ***OpenVINO*** [18] ou ***TensorRT*** [19] qui font de la quantization pour réduire le temps d'inférence, ***ONNX*** [20] qui propose d'exécuter le modèle sur un interpréteur optimisé *(le modèle déployé utilise partiellement cette option)* ou encore ***DeciNets*** [21] qui optimise les modèles en fonction de l'architecture sur laquelle il est exécuté.

Ce sont des options qu'il convient d'étudier sérieusement si l'on veut appliquer en temps réel une segmentation sémantique sur un flux vidéo.

Autres architectures

Il y a beaucoup d'autres architectures qui pourraient être testées, et en particulier des architectures avec des systèmes d'attention multi-échelles comme ***SegFormer*** [22] ou encore ***HRNetV2-OCR+PSA*** [23] qui sont les modèles SotA du moment.

Autres backbones

J'ai systématiquement choisi les versions des backbones les plus lourdes *(donc normalement celles qui donnent les meilleurs résultats, mais aussi les plus lentes…)* et il pourrait être intéressant de voir l'influence de backbones plus légers.

Mesure du temps

Enfin, puisque le temps d'inférence est crucial dans un système comme celui-ci, il serait intéressant de retravailler la méthode utilisée pour les mesures de temps.

En effet, j'ai constaté de grande variation dans les temps d'inférence sur un même modèle et il n'est pas certain que les mesures utilisées jusque-là soient suffisamment fiables pour une comparaison plus poussée de nos modèles.

Références

[1] https://deci.ai/blog/sota-dnns-overview/
[2] https://paperswithcode.com/sota/semantic-segmentation-on-cityscapes-val
[3] https://www.researchgate.net/publication/357761289_Review_the_state-of-the-art_technologies_of_semantic_segmentation_based_on_deep_learning
[4] https://arxiv.org/abs/2105.15203
[5] https://huggingface.co/docs/transformers/main/en/model_doc/segformer
[6] https://arxiv.org/pdf/2201.01615.pdf
[7] https://arxiv.org/pdf/2005.10821v1.pdf
[8] https://arxiv.org/pdf/2005.10821v1.pdf
[9] https://deci.ai/blog/sota-dnns-overview/
[10] https://www.cityscapes-dataset.com/dataset-overview
[11] https://en.wikipedia.org/wiki/Jaccard_index
[12] https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient
[13] https://keras.io/examples/vision/oxford_pets_image_segmentation
[14] https://albumentations.ai
[15] https://albumentations.ai/docs/autoalbument
[16] https://paperswithcode.com/method/focal-loss
[17] https://fr.wikipedia.org/wiki/U-Net & https://paperswithcode.com/method/u-net
[18] https://docs.openvino.ai/latest/get_started.html
[19] https://github.com/NVIDIA/TensorRT
[20] https://onnx.ai
[21] https://deci.ai
[22] https://arxiv.org/abs/2105.15203
[23] https://arxiv.org/pdf/1904.04514.pdf

Sources des schemas

[Fig-1] https://www.researchgate.net/publication/329616112_Fusion_Scheme_for_Semantic_and_Instance-level_Segmentation#pf6
[Fig-2] https://arxiv.org/abs/2201.01615v1
[Fig-3] https://huggingface.co/docs/transformers/main/en/model_doc/segformer
[Fig-4] https://www.microsoft.com/en-us/research/blog/high-resolution-network-a-universal-neural-architecture-for-visual-recognition
[Fig-5] https://paperswithcode.com/sota/semantic-segmentation-on-cityscapes-val
[Fig-6] https://arxiv.org/pdf/2005.10821v1.pdf
[Fig-7] https://github.com/qubvel/segmentation_models
[Fig-8] https://github.com/qubvel/segmentation_models
[Fig-9] https://github.com/qubvel/segmentation_models
[Fig-10] par l'auteur
[Fig-11] par l'auteur
[Fig-12] par l'auteur
[Fig-13] par l'auteur
[Fig-14] par l'auteur
[Fig-15] par l'auteur
[Fig-16] par l'auteur
[Fig-17] par l'auteur
[Fig-18] par l'auteur