Segmentation d'image pour véhicule intelligent

Fabien Cappelli
R&D
Future Vision Transport

Table des matières

Table des figures

Résumé

Cette étude explore diverses approches de segmentation sémantique adaptées à un contexte embarqué pour véhicule intelligent. Partant d'une baseline U-Net classique entraînée « from scratch » sur le jeu de données Cityscapes, nous montrons comment l'intégration d'un encodeur pré-entraîné (EfficientNet-B0) couplé à une architecture FPN et à des techniques spécifiques telles que l'augmentation de données avec Albumentations et l'utilisation d'une Dice loss permet d'atteindre des performances élevées (mIoU de 0,816) avec un nombre très modéré de paramètres ($\sim$7 millions). En confrontant nos résultats à l'état de l'art récent, notamment SERNet-Former (mécanismes attentionnels) et Depth Anything (estimation monoculaire de profondeur), nous mettons en évidence la pertinence d'intégrer dans nos futures recherches des mécanismes d'attention légers et des indices de profondeur estimés automatiquement. Enfin, l'évolution récente du matériel embarqué spécialisé facilite l'intégration réaliste de ces approches plus avancées dans les véhicules intelligents.

I. Introduction

La segmentation sémantique est une tâche de vision par ordinateur qui consiste à attribuer une étiquette de classe à chaque pixel d'une image. Contrairement à la classification d'image classique (qui donne une étiquette globale) ou à la détection d'objets (qui dessine des boîtes englobantes), la segmentation fournit une compréhension dense et précise de la scène au niveau pixel. Par exemple, un modèle de segmentation sémantique ne se contente pas de détecter qu'une voiture est présente, il indique exactement quels pixels appartiennent à des voitures, versus lesquels sont la route, le ciel, les piétons, etc. L'intérêt est de diviser l'image en régions significatives correspondant aux différentes classes d'objets, permettant une compréhension fine de l'environnement visuel.

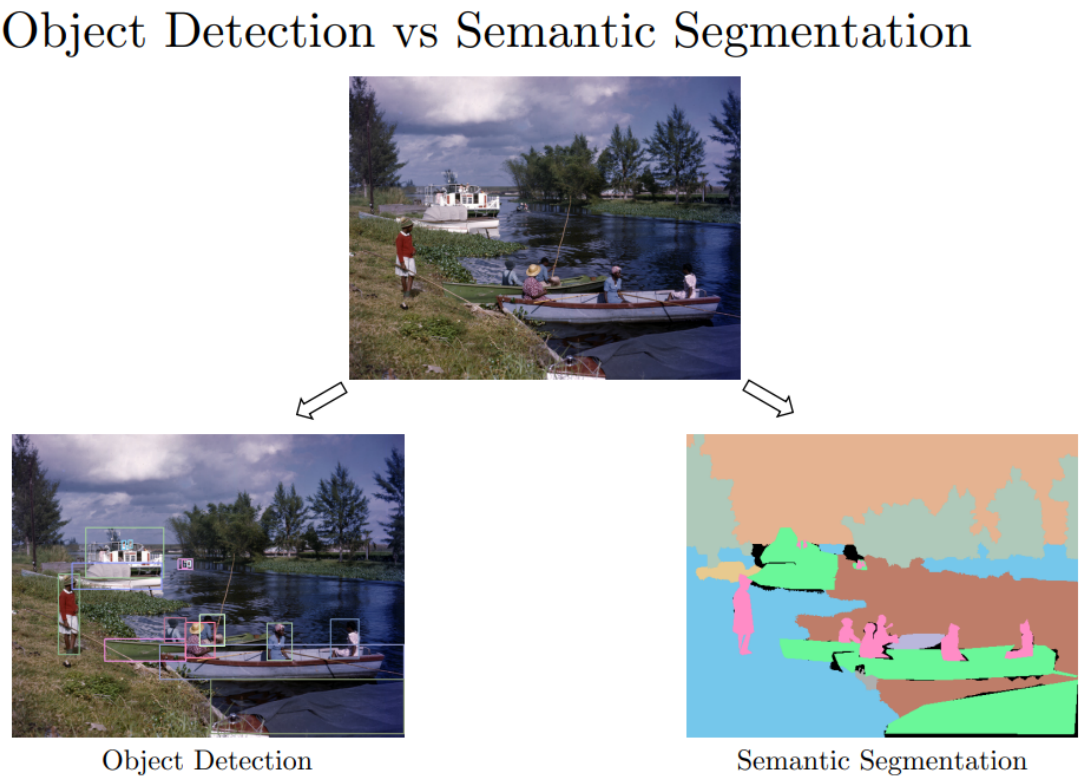

Figure 1 – Détection vs. Segmentation

Le jeu de données fourni comprend 3 475 images haute résolution (2048×1024 pixels) capturées dans les rues de 27 villes, annotées finement en segments correspondant à 32 classes (route, bâtiment, piéton, véhicule, etc.). Ces 32 classes sont groupées en 8 super-catégories (ou groupes) qui sont le plat (route, trottoir, ...), les humains, les véhicules, les constructions, les objets, la nature, le ciel et le vide.

Contrairement à des données textuelles ou tabulaires, les images haute résolution et leurs masques de segmentation occupent un volume mémoire très important, bien au-delà de la mémoire GPU (VRAM) disponible. Il est donc impossible de charger l'intégralité du jeu de données en VRAM d'un seul coup.

Il faut mettre en place un chargement progressif via un *data loader* : l'entraînement se fait par mini-lots (*batches*) successifs en mémoire GPU au lieu de traiter toutes les images à la fois. Ce pipeline de données permet également d'effectuer à la volée les prétraitements nécessaires (redimensionnement des images, augmentations de données, normalisation) sur chaque batch. Par ailleurs, l'utilisation d'un GPU est indispensable pour accélérer les calculs, mais sa VRAM limitée oblige à utiliser de petits batchs (parfois une seule image) et à réduire la résolution des images si nécessaire. Ces contraintes allongent la durée de l'entraînement, car le modèle ne peut traiter qu'un nombre restreint d'images en parallèle.

Ici le GPU utilisé sera un L4 sur Google Colab. Il coûte un peu plus de deux unités de calcul par heure, et une itération sur le jeu total de données prend environ quatre heures (fluctuant selon les modèles).

Nous allons limiter notre investigation et notre modélisation aux 8 groupes, et non aux 32 classes : cela signifie que chaque pixel du masque aura une valeur de 0 à 7 – en pratique, on passe par huit masques pour chaque groupe, dont les pixels ont soit la valeur 0 soit la valeur 1 (on appelle cela le *one-hot encoding*).

II. Baseline

Le modèle de base qui nous servira d'évaluation sera un modèle de type U-Net *from scratch* en Keras.

A. Principe de l'architecture U-Net

Le U-Net est devenu un classique de la segmentation d'images, initialement proposé pour la bio-imagerie médicale. Son architecture en U symétrique comporte :

- Un encodeur (branche de gauche) qui est en fait un réseau de classification convolutionnel tronqué. Il applique plusieurs blocs de convolution (souvent 2 conv 3×3 + ReLU + BN) suivis chacun d'une pooling 2×2 pour réduire la résolution. À chaque sous-échantillonnage, le nombre de canaux de features est doublé, tandis que les dimensions spatiales sont réduites de moitié, capturant des caractéristiques de plus en plus globales. Ainsi, l'image est encodée en une représentation compacte (dernière couche au fond du U) contenant les informations sémantiques riches mais à faible résolution.

- Un décodeur (branche de droite) qui effectue l'opération inverse : on réalise un textitupsampling (par exemple via une convolution transpose 2×2) pour augmenter la résolution, tout en diminuant le nombre de canaux. À chaque étape du décodage, U-Net concatène les *features* upsamplées avec les *features* correspondantes du niveau d'encodeur de même résolution (d'où les flèches de *skip*). Cette concaténation injecte les détails fins provenant de l'encodeur (qui auraient été autrement perdus) dans le décodeur, permettant de reconstruire des contours précis. Après la concaténation, des convolutions 3×3 + ReLU affinent la combinaison. Ce processus se répète à chaque niveau. Enfin, une dernière convolution 1×1 produit la carte de segmentation finale, en assignant à chaque pixel un score pour chaque classe.

L'astuce du U-Net est donc de préserver l'information spatiale à chaque échelle via les *skip connections*, ce qui le distingue d'un simple autoencodeur où l'information passerait par un goulet d'étranglement très compressé. En évitant une trop forte compression, U-Net parvient à conserver la localisation des objets tout en exploitant le contexte global. Cette architecture a montré son efficacité sur de nombreux jeux de données, offrant un bon compromis entre précision et complexité. Néanmoins, un U-Net standard avec un encodeur profond peut comporter des dizaines de millions de paramètres. On peut le rendre plus léger en choisissant un encodeur moins complexe ou en réduisant le nombre de filtres par couche, mais cela peut dégrader la précision sur des images complexes comme celles de notre jeu de données.

B. Mise en place

Notre architecture suit la conception U-Net originale : un chemin d'encodeur contractant constitué de blocs de deux convolutions 3x3 suivies d'une *max pooling*, avec un nombre de filtres qui double à chaque étape (64, 128, 256, 512, puis 1024 au centre). Le décodeur effectue des upsamplings successifs, concatène les features correspondantes de l'encodeur (skip connections), puis applique deux convolutions 3x3 à chaque niveau de remontée, jusqu'à restituer une carte de segmentation multi-classe à la taille d'origine,

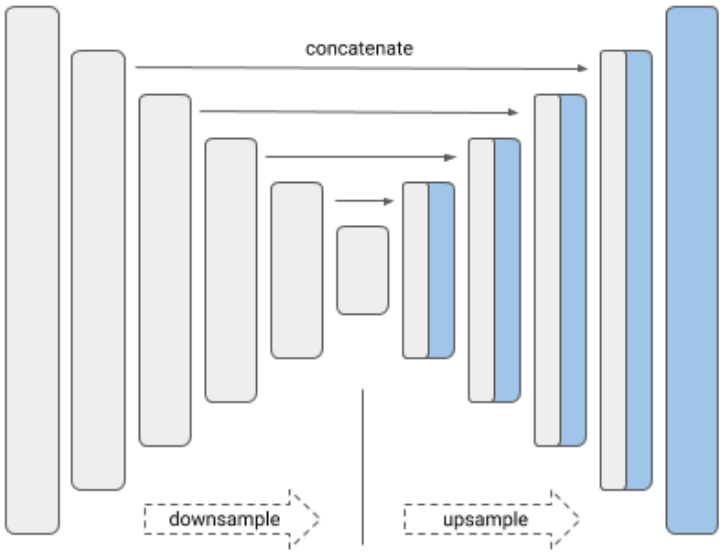

FIGURE 2 – Architecture U-Net

avec activation *softmax* en sortie. La structure est parfaitement symétrique et sans prétraitement particulier sur les poids, ce qui permet d'évaluer les performances d'un U-Net "pur" sur nos images, dont nous avons préalablement réduit la taille à 256×512 pixels (i. e. division par quatre de la résolution).

C. Performance

La performance de cette baseline est honorable. Pour la mesurer, on utilise l'IoU (Intersection over Union), ou coefficient de Jaccard, une métrique utilisée pour évaluer la qualité d'une segmentation : elle mesure le rapport entre la surface d'intersection et la surface de l'union entre le masque prédit et le masque de référence. Plus l'IoU est proche de 1, plus la prédiction recouvre fidèlement la vérité terrain pixel par pixel.

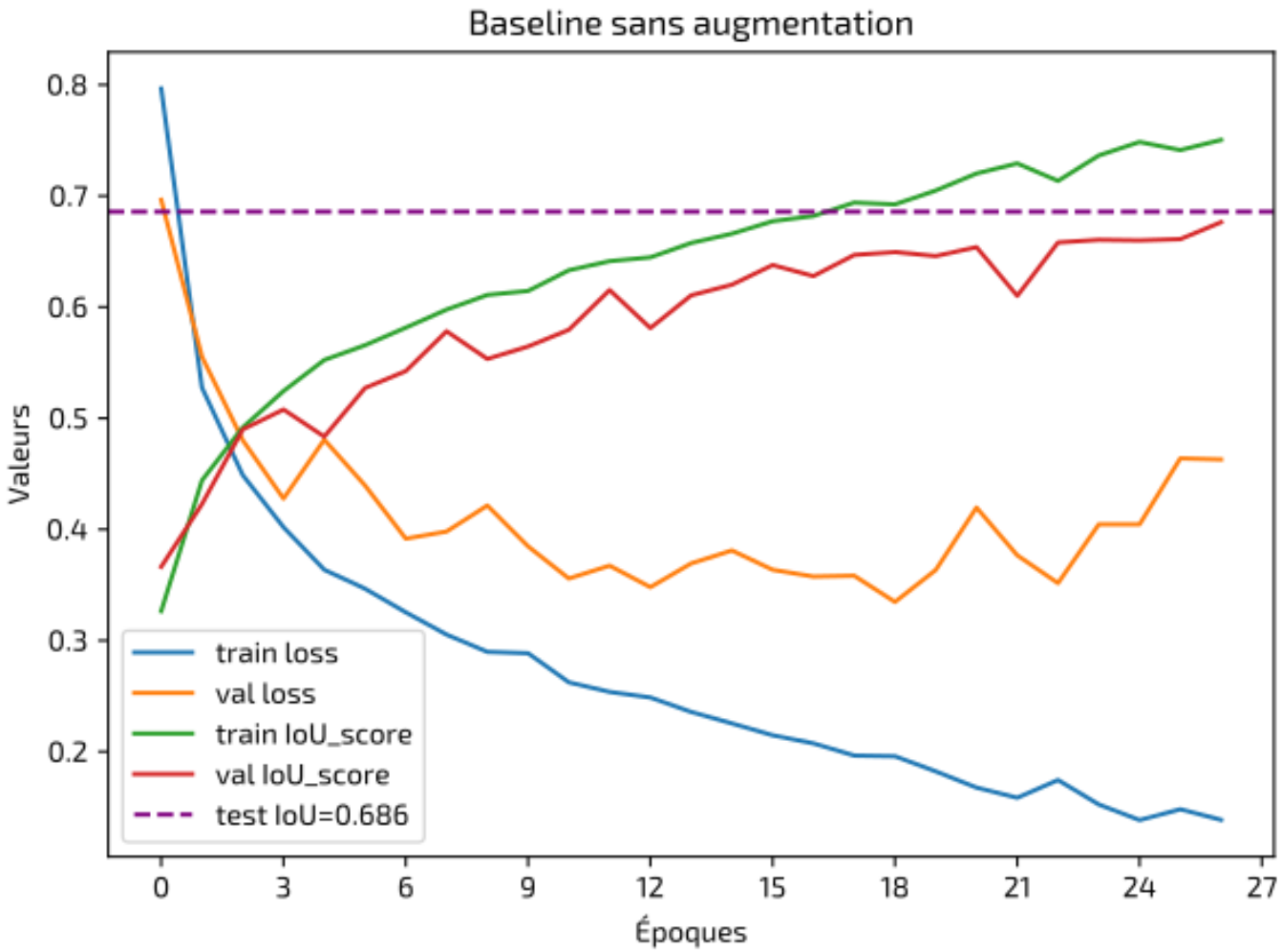

FIGURE 3 – Baseline U-Net

Néanmoins les calculs sont lourds, et malgré la puissance du GPU L4, nous ne pouvons procéder qu'avec des batches de 4 images (et ce alors même que nous avons déjà divisé par quatre leur résolution).

D. Augmentation de données

L'augmentation de données via la bibliothèque `Albumentations` consiste à appliquer aléatoirement des transformations (rotations, symétries, changements de luminosité, etc., cf. figure 4) sur les images et leurs masques pendant l'entraînement. Cela permet d'accroître artificiellement la diversité du dataset, d'améliorer la robustesse du modèle face aux variations de la scène, et de limiter le risque de surapprentissage.

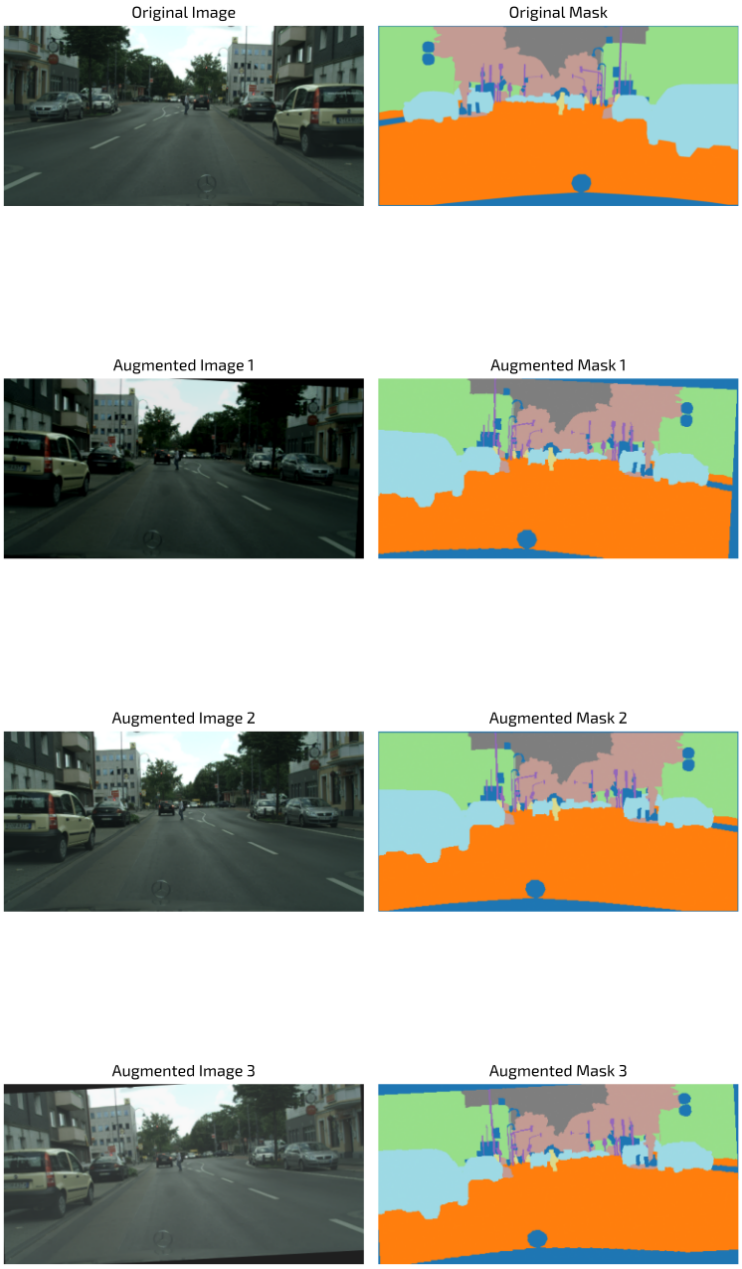

FIGURE 4 – Exemples d'augmentation

Après plusieurs tests, nous avons établi que la variation de contraste permettait une petite hausse de la performance mais surtout une réduction du surapprentissage.

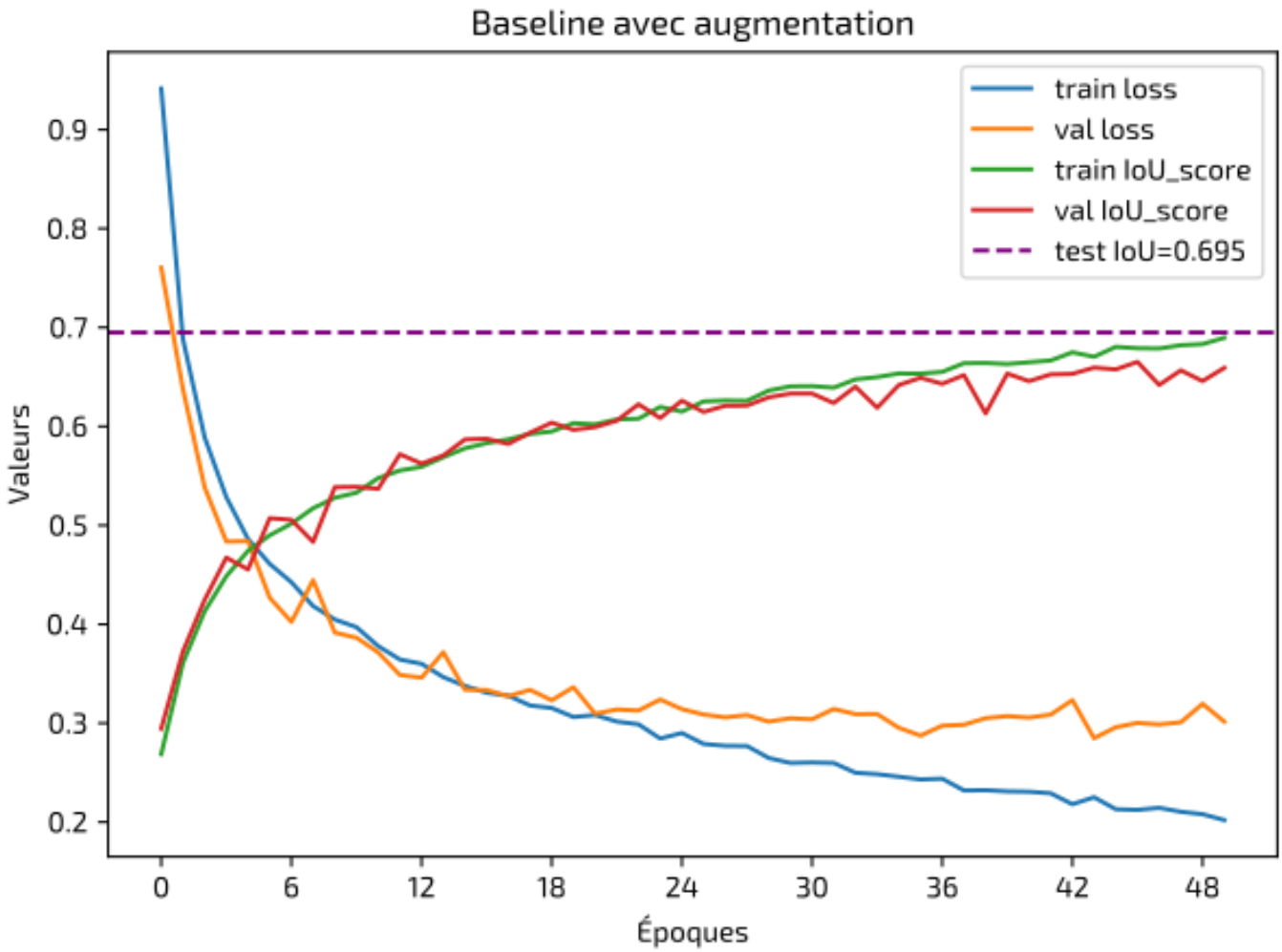

FIGURE 5 – Baseline U-Net avec augmentation de données

III. EXPLOITATION DE LA LIBRAIRIE `segmentation_models`

La bibliothèque `segmentation_models` (de Qubvel) est une librairie open source pour Python / Keras / TensorFlow qui facilite l'entraînement et l'utilisation de modèles de segmentation d'images de pointe. Elle propose des implémentations prêtes à l'emploi de nombreuses architectures populaires (U-Net, LinkNet, FPN, PSPNet), avec la possibilité de choisir parmi plus de vingt encodeurs (*backbones*) pré-entraînés sur ImageNet. Grâce à son API simple, il est possible de construire, entraîner et évaluer des modèles de segmentation performants en quelques lignes, tout en bénéficiant d'outils intégrés pour les métriques (IoU, F-score), les pertes adaptées, et la gestion des poids pré-entraînés. Cette librairie permet ainsi de prototyper rapidement et de comparer différentes approches de segmentation dans un cadre unifié et flexible.

L'encodeur n'extrait donc plus les informations à partir des seules données d'entraînement, comme dans le cas de notre *baseline*, mais applique une extraction de motifs déjà appris lors d'un entraînement beaucoup plus élaboré. C'est la partie *downsample* sur notre Figure 2.

Le décodeur, quant à lui, représente la partie *upsample*, mais en fin de compte réfère à l'architecture du modèle utilisé.

A. Décodeurs

Nous avons déjà présenté l'architecture U-Net, voici un rapide panorama des deux autres architectures sur les trois proposées par cette librairie que nous avons utilisées.

1) *LinkNet:* LinkNet est une architecture conçue explicitement pour l'efficacité et le temps réel en segmentation. Sa philosophie est de réutiliser au maximum les *features* calculées par l'encodeur dans le décodeur, sans alourdir le modèle. Concrètement, LinkNet reprend un encodeur convolutionnel classique et insère des liens directs (*links*) entre chaque bloc d'encodeur et le bloc de décodeur correspondant. Ces liens transmettent les cartes de *features* d'encodeur par addition aux *features* upsamplées du décodeur, au lieu de les concaténer. En faisant une addition, on n'augmente pas le nombre de canaux dans le décodeur (contrairement à la concaténation de U-Net qui double les canaux, nécessitant ensuite des couches de convolution pour fusionner) – cela évite d'introduire des paramètres supplémentaires. L'information spatiale est bypassée directement du côté encodeur vers le décodeur, ce qui aide à récupérer les détails perdus par le *pooling*, tout en maintenant le réseau compact.

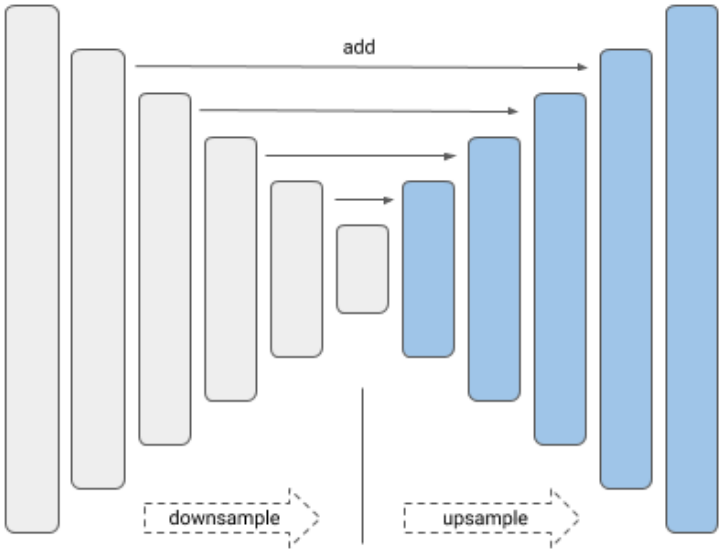

FIGURE 6 – Architecture LinkNet

LinkNet adopte un compromis judicieux : sacrifier un peu de complexité (et potentiellement une légère baisse de précision par rapport à des modèles plus lourds) en échange d'une forte augmentation de la vitesse et d'une empreinte mémoire réduite. Il démontre qu'en exploitant efficacement les *features* de l'encodeur via des liens directs, on peut obtenir une segmentation de bonne qualité sans réseau massif. LinkNet est donc un candidat tout désigné lorsque la contrainte de calcul est forte ou pour déployer un service de segmentation sur une plateforme à ressources limitées (typiquement les systèmes embarqués, et donc les voitures).

2) *FPN:* Le Feature Pyramid Network n'est pas à l'origine un modèle de segmentation autonome, mais un module introduit pour la détection d'objets et qui s'est révélé très utile pour la segmentation également. L'idée de FPN est de résoudre un problème clé : comment traiter efficacement les différentes échelles d'objets présentes dans l'image. Des objets petits (un piéton éloigné) et grands (un bâtiment) doivent tous deux être correctement segmentés, or un seul niveau de feature convolutionnelle ne peut pas tout capturer. FPN crée donc une pyramide de features multi-niveaux en exploitant les cartes de feature issues de plusieurs niveaux de l'encodeur.

Concrètement, on prend un encodeur (par ex. ResNet) et on extrait ses features à différentes profondeurs (disons après chaque pool, on a des features aux résolutions 1/4, 1/8, 1/16, 1/32 de l'image). Le FPN va construire un chemin *top-down* : on upsample la feature la plus profonde (1/32) pour la porter à 1/16, et on l'additionne avec la feature de l'encodeur déjà à 1/16 (après lui avoir appliqué une légère conv 1×1 pour l'harmoniser). Puis on prend ce résultat, on l'upsample à 1/8 et on additionne avec la feature encodeur

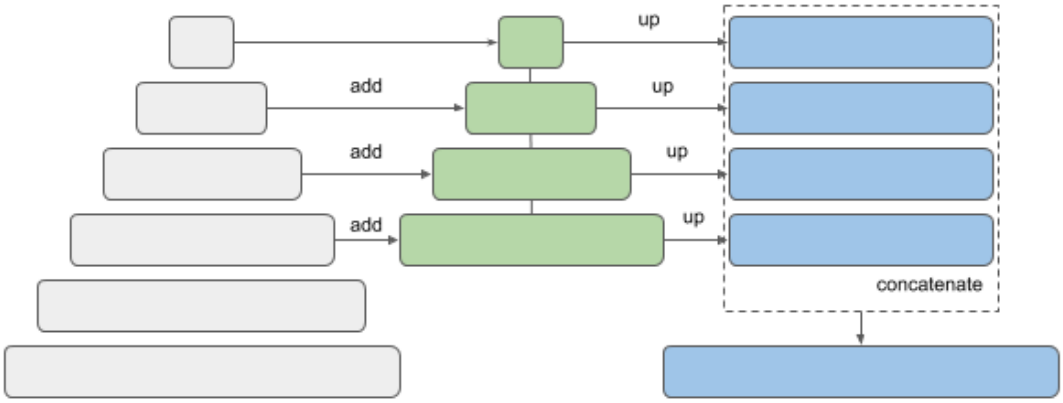

FIGURE 7 – Architecture FPN

1/8, et ainsi de suite. Ce processus de *lateral connection* + *upsampling* se répète à chaque niveau jusqu'à la résolution la plus fine. On obtient ainsi à chaque échelle une carte de feature qui combine à la fois des informations sémantiques élevées (venues du bas de l'encodeur) et des détails fins hérités des couches hautes de l'encodeur. Chaque niveau de la pyramide est donc riche sémantiquement, y compris le plus haute résolution, ce qui améliore la détection des petits objets. Enfin, ces features multi-échelles peuvent être fusionnées (par concaténation ou autre) ou traitées chacune par une tête de prédiction. En segmentation sémantique, typiquement on peut upsampler toutes les features au niveau de l'image et les concaténer pour prédire le masque final, ou bien prédire plusieurs masques grossiers à fins et les combiner. L'architecture FPN assure en tout cas que toutes les échelles d'objets sont considérées.

3) Résumé: U-Net mise sur des skip connections par concaténation pour conserver les détails, LinkNet privilégie la légèreté avec des skip par addition et un décodeur simplifié, et FPN introduit une fusion multi-échelles des features pour mieux gérer la variété de tailles d'objets. Ces trois approches partagent l'idée d'un encodeur pré-entraîné alimentant un décodeur spécialisé.

Nous allons voir maintenant comment le choix de l'encodeur sous-jacent peut également influencer les performances.

B. Encodeurs

Dans toutes les architectures précédentes, la partie encodeur est responsable d'extraire des descripteurs pertinents de l'image d'entrée. Il s'agit généralement d'un CNN de classification déjà bien établi (ResNet, VGG, EfficientNet, etc.) dont on réutilise la structure et éventuellement les poids. L'utilisation d'un encodeur pré-entraîné sur ImageNet est une pratique courante en segmentation : ces poids initiaux, appris pour distinguer des milliers d'objets, fournissent une excellente base de filtres visuels, ce qui accélère la convergence et améliore souvent la précision finale du modèle (*a contrario*, dans notre baseline, notre modèle apprenait lui-même ses features, et créait lui-même ses poids). La bibliothèque `segmentation_models` de Qubvel facilite le processus en offrant un large éventail d'encodeurs au choix – 25 backbones disponibles, tous avec des poids pré-entraînés sur ImageNet 2012. On peut ainsi créer un U-Net ou FPN en quelques lignes en spécifiant par exemple `sm.Unet('resnet34', encoder_weights='imagenet')`.

Pour ce projet, nous avons testé les encodeurs suivants :

1) *ResNet:* un backbone très populaire. Les versions plus profondes (ResNet101/152) capturent des caractéristiques complexes mais sont plus lourdes à entraîner et à inférer que ResNet34 par exemple. Un compromis fréquent pour la segmentation est ResNet-34 ou 50, qui offrent un bon équilibre précision/vitesse. Le premier est choisi ici.

2) *EfficientNet:* famille de backbones récents optimisés pour le rapport précision/paramètres. EfficientNet-B5/B7 atteignent de très hautes performances ImageNet, ce qui peut se traduire en meilleure segmentation, mais ils sont aussi très lourds. Les versions B0-B3 offrent un bon compromis. Encore une fois, la version la plus légère (B0) est celle retenue.

IV. Paramétrage

Le *learning rate* (taux d'apprentissage) est un paramètre fondamental dans l'entraînement des modèles de segmentation : il détermine l'ampleur des ajustements réalisés sur les poids du réseau à chaque itération. Un *learning rate* trop élevé risque de faire diverger l'apprentissage, tandis qu'un taux trop faible ralentit la convergence et peut piéger le modèle dans un minimum local.

La taille du batch choisie dans le *data loader* influence directement la stabilité de la descente de gradient : des batchs plus grands offrent des estimations de gradient plus stables mais exigent plus de mémoire GPU, alors que de petits batchs permettent de s'adapter à des contraintes matérielles mais introduisent davantage de bruit dans l'optimisation. Dans la pratique, le choix du *learning rate* optimal dépend souvent de la taille du batch : il est courant d'augmenter le *learning rate* si l'on augmente la taille du batch, et inversement, afin de garder une dynamique d'apprentissage efficace et stable. La combinaison de ces deux paramètres conditionne donc à la fois la qualité de la convergence et la vitesse d'entraînement du modèle.

D'une part j'ai donc mené des tests sur un échantillon des données pour voir quelles paires *learning rate* / taille de batch semblait donner des résultats stables. D'autre part, j'ai utilisé des *callbacks* comme `ReduceLROnPlateau` et `EarlyStopping`, essentiels pour optimiser le processus d'apprentissage. `ReduceLROnPlateau` ajuste automatiquement le *learning rate* : si la performance (par exemple la validation loss ou le score IoU) cesse de s'améliorer pendant un certain nombre d'époques, ce *callback* réduit le *learning rate*, permettant ainsi au modèle de continuer à progresser plus finement. `EarlyStopping`, de son côté, surveille l'évolution d'une métrique et interrompt l'entraînement dès qu'il constate une stagnation prolongée, évitant ainsi le surapprentissage et le gaspillage de ressources. Ensemble, ces *callbacks* assurent une convergence plus efficace, préviennent le surapprentissage et accélèrent l'obtention d'un modèle performant.

Par défaut, dans la librairie `segmentation_models`, la fonction de perte (*loss*) est la `categorical_crossentropy`, une fonction de perte classique pour les tâches de classification multi-classe. Elle mesure la différence entre la distribution des probabilités prédites par le modèle et la valeur réelle des pixels. Concrètement, la `categorical crossentropy` pénalise fortement les prédictions erronées : plus la probabilité prédite pour la bonne classe est faible, plus la perte est élevée. Minimiser cette fonction de perte pousse donc le modèle à attribuer la probabilité la plus élevée possible à la classe correcte pour chaque pixel, ce qui favorise une segmentation précise et fiable.

Enfin j'ai ajouté à mon modèle le plus performant la *Dice loss*, une fonction de perte spécialement conçue pour les tâches de segmentation d'images, en particulier lorsque les classes sont déséquilibrées. Elle mesure la similarité entre le masque prédit et le masque réel en calculant un indice de recouvrement : plus la *Dice loss* est faible, plus la prédiction épouse fidèlement la vérité terrain. Son principal avantage est de donner plus de poids aux petits objets ou classes rares, ce qui aide le modèle à mieux segmenter les zones peu représentées dans les images (ce qui peut être crucial, quand on pense au peu de surface qu'occupera un humain sur une photo, comparé à un building ou au ciel, par exemple).

V. Résultats

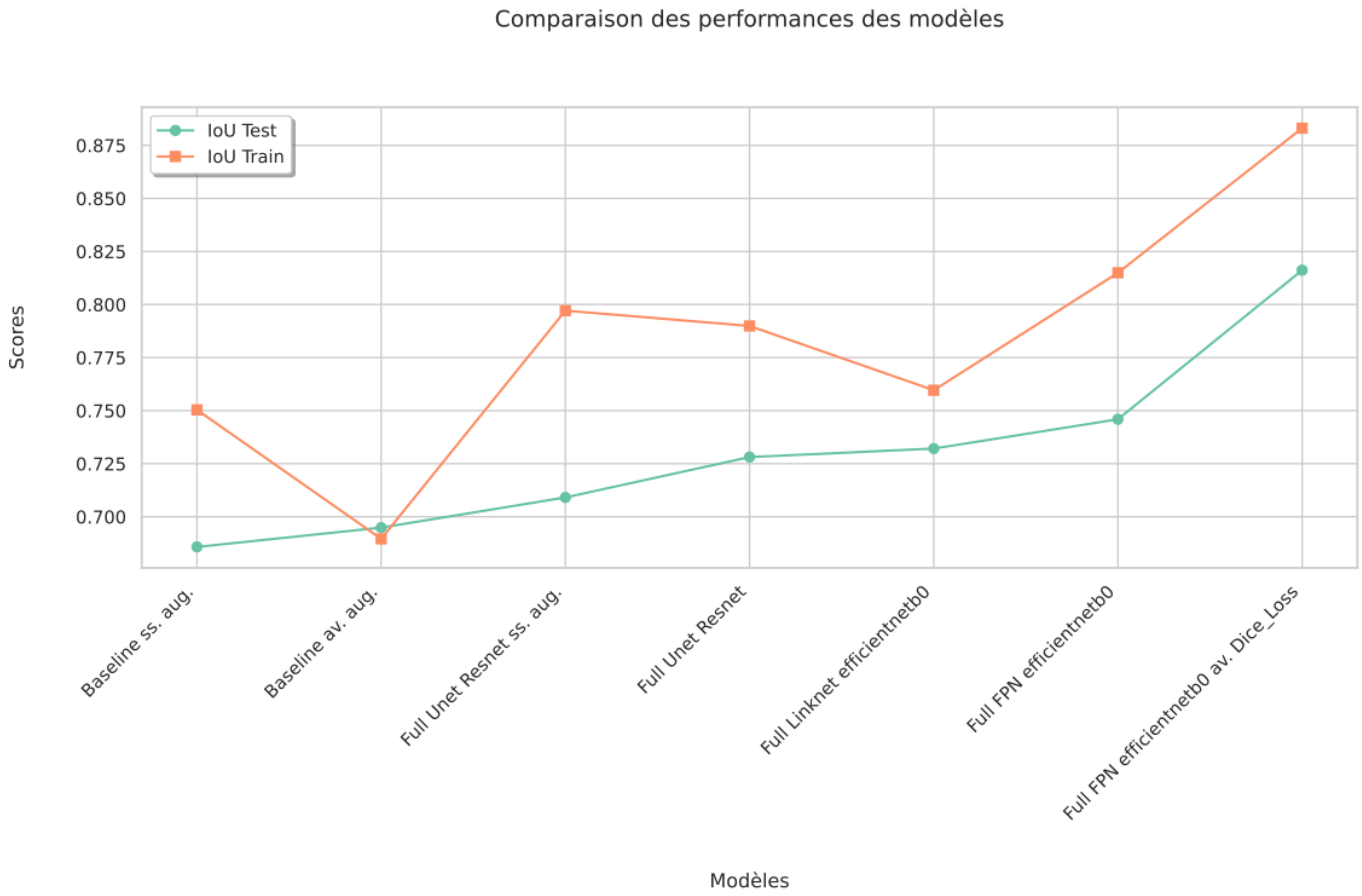

Figure 8 – Résultats (IoU) des différents modèles expérimentés

On voit dans la figure 8 le score IoU pour l'entraînement et le test. Cela permet à la fois d'apprécier la performance sur le jeu de test (ici 10% des données destinées à l'entraînement qui ont été écartées avant tout travail de modélisation) et le risque de surapprentissage (un mot que nous avons déjà utilisé, et qui désigne l'incapacité du modèle à généraliser en dehors des données qui lui ont été fournies pour l'entraînement).

On voit notamment sur les deux premiers types de modèles (Baseline et U-Net/ResNet), qui ont été implémentés avec et sans augmentation de données, que l'augmentation de données permet à la fois une hausse de la performance sur le jeu de test (même si on a une baisse de la performance sur le jeu d'entraînement) et une réduction du sur-apprentissage. C'est pourquoi cette augmentation de données à été conservée sur les modèles suivants.

On a aussi évalué la vitesse d'inférence à chaud du modèle (cf. Figure 9).

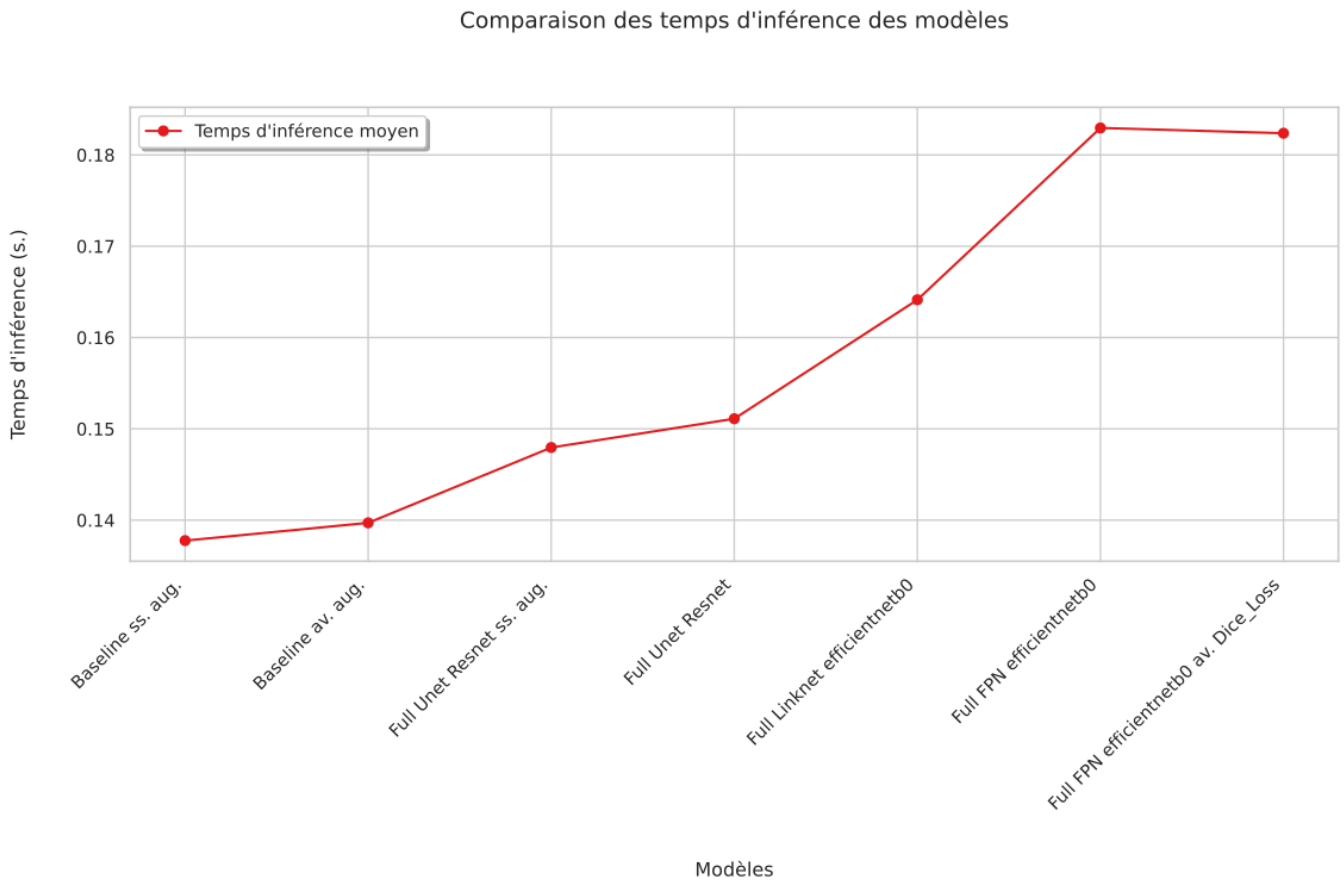

FIGURE 9 – Résultats (Temps d'inférence) des différents modèles

Le duo LinkNet – EfficientNet nous semblait prometteur, dans le sens où à la fois l'encodeur et le décodeur étaient dédiés à la légereté et à l'intégration dans des systèmes embarqués. Si le modèle lui-même est relativement léger par rapport au duo U-Net – ResNet (71 Mo *vs.* 281 Mo), il est un peu plus lent sur l'inférence (de l'ordre du centième de seconde). Pour deux centièmes de plus, le gain en performance avec une architecture FPN, toujours un encodeur EfficientNet et la loss DICE ajoutée est significatif. C'est donc ce modèle qui sera proposé au terme de cette première étude.

Le graphique des métriques de l'entraînement de ce modèle est particulièrement éclairant. Les premières époques voient une grande variabilité dans la fonction de perte et dans le score IoU sur les données de validation : la taille du batch (huit images) entraîne des soucis de généralisation qui disparaissent au fur et à mesure. Le modèle s'affine progressivement sur les dernières époques grâce au callback `ReduceLROnPlateau`. Le surapprentissage est minime (à ce stade on le considère comme inexistant), ce qui garantit une bonne portabilité du modèle : malgré ses "petits" 7 millions de paramètres, il va pouvoir garder une performance extrêmement satisfaisante (un IoU de 0.816 pour ce type de modèle reste un score très bon sur ces données) sur des données nouvelles, comme vous pouvez d'ailleurs l'expérimenter sur http://projet8oc.fabiencappelli.com/.

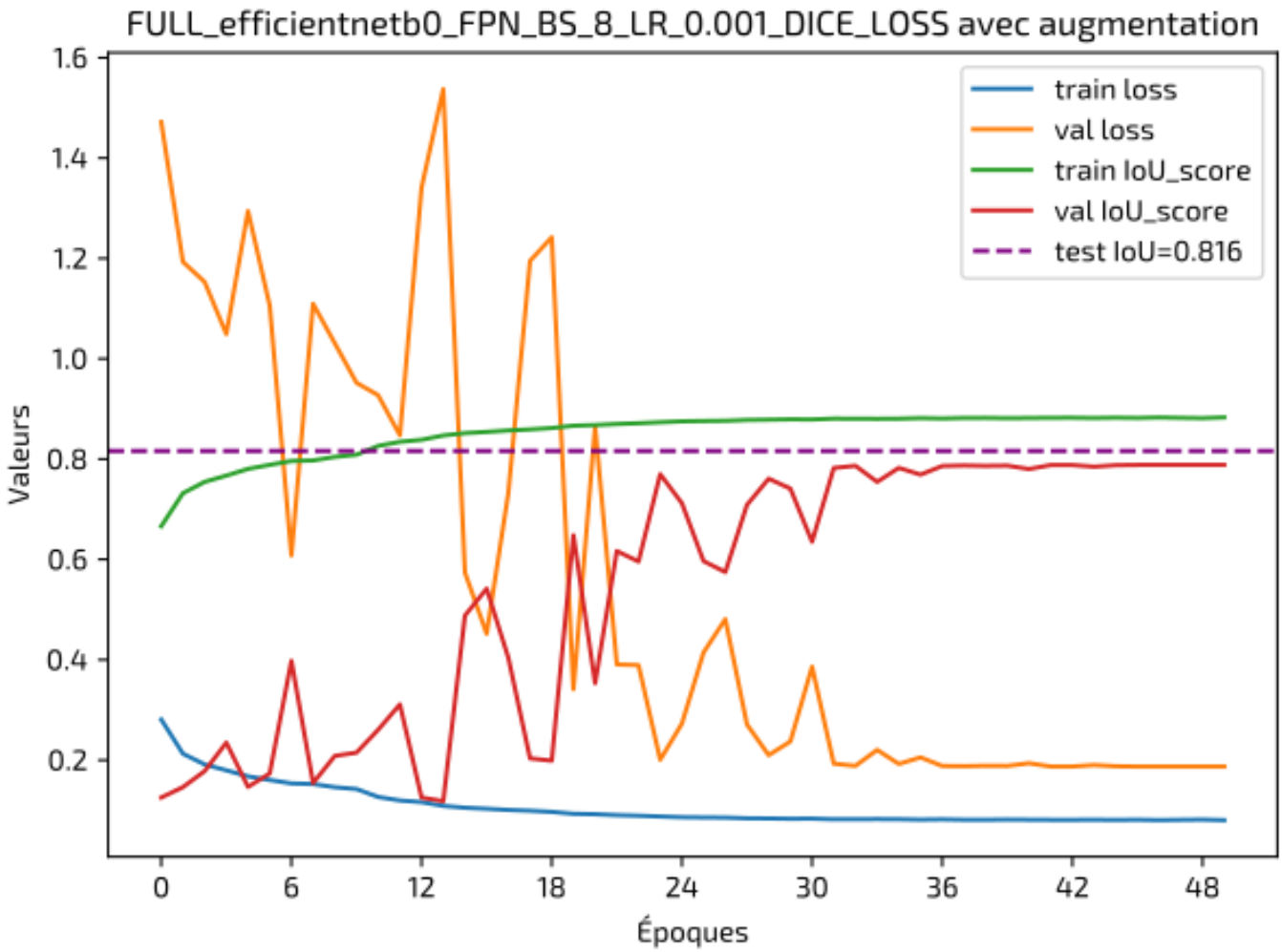

FIGURE 10 – Entraînement du modèle choisi

VI. CONCLUSION

Comme dit précédemment, le modèle retenu est assez peu complexe en regard de l'état de l'art. La performance que nous avons obtenue pour ce nombre de paramètres est cohérente avec l'état de la recherche (voir par exemple [2], p. 2, mais nous devons garder en mémoire que nous avons segmenté sur 8 super-catégories, et non les 19 habituellement utilisées sur les benchmark pour Cityscapes, ce qui explique peut-être notre performance en fin de compte plutôt bonne malgré une modélisation relativement simple).

Le site paperswithcode (https://paperswithcode.com/sota/semantic-segmentation-on-cityscapes-val) récapitule l'avancée de la recherche sur le dataset Cityscapes, et nous pouvons voir de nouvelles approches qui permettent un gain de performance significatif. Voici les deux papiers vraiment importants de 2024 :

- [3] enrichit l'architecture classique encodeur-décodeur de mécanismes attentionnels spécifiques. Le modèle reste relativement léger et rapide à entraîner, et livre des performances excellentes (IoU moyen de 0.874)
- Dans [4], les auteurs augmentent les capacités de segmentation en palliant au fait que la perception soit monoculaire (notion de profondeur). Cela permet au modèle de se faire des représentations spatiales et sémantiques plus robustes, avec une généralisation excellente même dans des contextes inédits (capacité *zero-shot*)

D'un autre côté, les progrès récents en termes de matériel permettent de fournir aux systèmes embarqués des modèles plus exigeants qu'à l'époque de l'éclosion des modèles de segmentation explorés dans cette étude.

VII. Notes

La figure 1 est extraite de la présentation [1]

Les figures 2, 7 et 6 viennent du Readme du Github de la librairie `segmentation_models` : https://github.com/qubvel/segmentation_models/

Références

[1] Alexander Kirillov, Kaiming He, Ross Girshick and Piotr Dollár, "A Unified Architecture for Instance and Semantic Segmentation" http://presentations.cocodataset.org/COCO17-Stuff-FAIR.pdf. [Accessed : 10 Juil. 2025].

[2] Tianjian Meng, Golnaz Ghiasi, Reza Mahjourian, Quoc V. Le and Mingxing Tan, "Revisiting Multi-Scale Feature Fusion for Semantic Segmentation", *arXiv*, 2022. [Full Text]. https://arxiv.org/abs/2203.12683. [Accessed on : 17 Juil. 2025].

[3] Serdar Erisen, "SERNet-Former : Semantic Segmentation by Efficient Residual Network with Attention-Boosting Gates and Attention-Fusion Networks", *arXiv*, 2024. [Full Text]. https://arxiv.org/abs/2401.15741. [Accessed on : 17 Juil. 2025].

[4] Lihe Yang, Bingyi Kang, Zilong Huang, Xiaogang Xu, Jiashi Feng and Hengshuang Zhao, "Depth Anything : Unleashing the Power of Large-Scale Unlabeled Data", *arXiv*, 2024. [Full Text]. https://arxiv.org/abs/2401.10891. [Accessed on : 17 Juil. 2025].