Note méthodologique : preuve de concept

Dataset

L'option sélectionnée est de nous appuyer sur le jeu de données *Cityscapes*. Ce dataset d'images et d'annotations permet d'entraîner des modèles sur des tâches de **segmentation sémantique** ou d'instance, de génération d'images, etc.
Il qui comporte notamment des **photos prises depuis un véhicule** ainsi que les ***masks* associés.** Le *mask* d'une photo est une image dont les valeurs des pixels correspondent aux différentes catégories d'objets/zones que l'on cherche à distinguer :

Figure 1 : exemple de mask

Dans l'exemple ci-dessus, où une image et son masque ont été superposés, l'on remarque que chaque couleur correspond bien à une catégorie donnée.
Pour le projet, nous n'utilisons pas la totalité des 32 catégories présentes dans le *dataset*. Nous les regroupons en 8 macro-catégories.

Le jeu de données est séparé en deux parties : les images, contenues dans `P8_Cityscapes_leftImg8bit_trainvaltest`, et les *masks* (notre *target*), contenus dans `P8_Cityscapes_gtFine_trainvaltest`.
Le jeu de données initial est composé de 4 types d'informations de masque :

- *Masks* couleur : chaque classe se voit attribuer une couleur différente.
- *Masks* d'identifiant d'instance : masques en niveaux de gris étiquetés avec des identifiants d'instance (chaque élément différent se voit attribuer une classe et un identifiant d'objet unique)
- **Identifiants d'étiquette : *mask* en niveaux de gris étiqueté avec l'identifiant de classe. Comme nous souhaitons faire de la segmentation sémantique, ce sont ces fichiers que nous utilisons**
- Polygones de masque : incluent les coordonnées détaillées de chaque segment.

Un découpage préalable *Train*/*Validation*/**Test** a été effectué :

- Entraînement : 2975 images / *masks*
- Validation : 500 images / *masks*
- Test : 1525 images / *masks*

Le souci est que les *masks* de Test ne sont pas exploitables, avec des identifiants d'étiquettes absents.
Nous utilisons donc seulement les données `train` et `val` pour notre travail.

Pour traiter ce dataset **volumineux** (11 Go), nous avions créé dans le cadre du projet précédent une sous-classe de `keras.utils.Sequence` : `generator`. Celle-ci permet de manipuler le *dataset* à l'échelle d'un lot (un `batch`), et ainsi de ménager la mémoire vive. Outre le chargement des images, elle offre également la possibilité de prétraiter directement les données selon les besoins du projet.

- Récupère des chemins d'accès des images et des *masks*
- (facultatif) Applique un **échantillonnage** `n_images`
- (facultatif) **Divise** les données avec la fonction `train_test_split` de `scikit-learn`
- **Lit** les images et les *masks* par lot, à la taille souhaitée `batch_size` et aux dimensions souhaitées `image_size`. Ici, nous réduisons la taille : (512, 1024, 3) → (256, 512, 3) pour réduire le temps d'apprentissage
- (facultatif) Applique de l'**augmentation** des données
- (facultatif) **Prétraite** les images pour être compatibles avec un `backbone` donné
- (facultatif) Effectue le mappage des **classes en macro-catégories** `cats`
- Met les labels du *mask* dans leurs propres canaux (256, 512) → (256, 512, 8)
- Met à disposition le batch créé
- (facultatif) A la fin de chaque `epoch`, **mélange** les chemins d'accès des images/*masks*

Les concepts du SegFormer

Le modèle choisi est le *SegFormer*, présenté en 2021 dans l'étude *SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers.* Cet algorithme utilise une architecture *Encoder* / *Decoder*, classique pour effectuer de la segmentation sémantique :

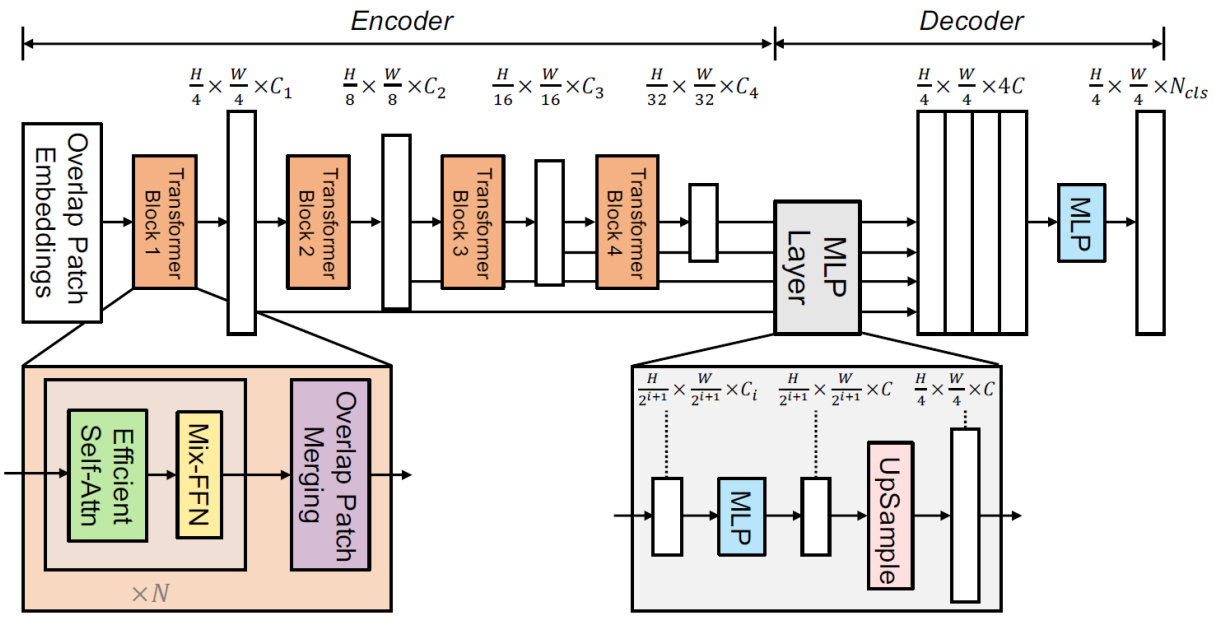


Figure 2 : architecture du SegFormer

Mais contrairement à un modèle basé sur des couches de convolutions dans les deux sections (on parlait alors de *Fully Convolutional Networks - FCN*), on a ici :

- Pour la partie *Encoder*, l'utilisation du mécanisme d'attention, s'inspirant des *Vision-Transformers* (*ViT*)
- Pour la partie *Decoder*, l'utilisation d'un Perceptron Multicouche (*MultiLayer Perceptron - MLP*) classique

Les *ViT* :

L'architecture *FCN* présentait quelques problèmes, comme l'aspect très local des features détectées, sans vraiment réussir à capturer des relations entre elles. Pour remédier à cela, l'idée a été de s'inspirer du succès des modèles *NLP* utilisant des *Transformers*, et de leur mécanisme d'attention. Tout comme en *NLP*, l'idée est de considérer notre image comme une séquence, pour pouvoir y détecter des relations très fines grâce à l'attention. L'image étant par essence en 2D, le calcul de l'attention doit se réaliser dans les deux dimensions, ce qui rend délicat le calcul. L'astuce a alors consisté à diviser l'image en séquences de patchs, et non plus de pixels, réduisant ainsi la complexité.

Regardons maintenant les différentes parties du *SegFormer* :

Hierarchical Transformer Encoder

Il permet de générer des *feature maps* de différentes résolutions, pour mettre à disposition du *decoder* de segmentation des *features* de différentes tailles.
Comme indiqué sur la figure 2, on les obtient en répétant plusieurs fois les mêmes étapes.

Overlapped Patch Merging

Dans l'implémentation classique des *ViT*, le *patch merging* (qui peut consister à réaliser une opération de convolution avec, dans le cadre d'une taille de patch égale à 4, `K = 4` , `S = 4` , `P = 0` et `n_filters = Ci` la dimension de *l'embedding*) permet de résumer chaque patch, pour qu'il puisse être traité ensuite comme un *token* dans une phrase. Il n'y a alors **pas de chevauchement** entre chaque patch. Les créateurs de *SegFormer* considèrent que **cette solution fait perdre la continuité locale autour des patchs**.
C'est pourquoi les concepteurs utilisent pour leur part une solution permettant de **faire se chevaucher les patchs** :

- Avec des *patchs* de taille 4 : K = 7 , S = 4 et P = 3 pour l'étape où l'on cherche à diviser la dimension par 4
- Avec des *patchs* de taille 2 : K = 3 , S = 2 et P = 1 pour les autres étapes, où l'on cherche à diviser la dimension par 2

Après la convolution, la sortie est redimensionnée : $(Hi, Wi, Ci) \rightarrow (Hi \times Wi, Ci)$, avant de passer dans le *Transformer Block* :

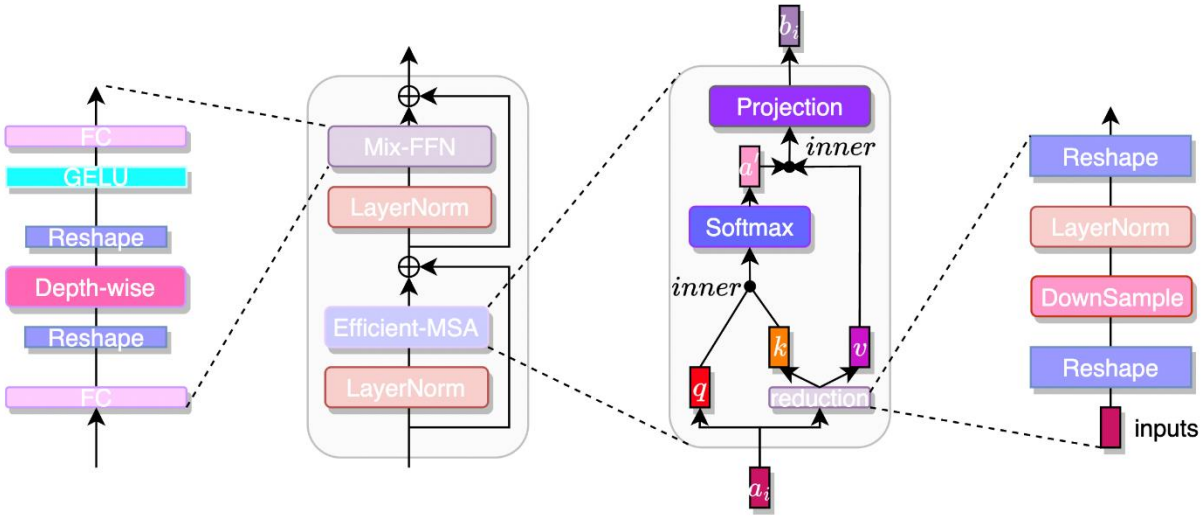


Figure 3 : schéma du block transformer

Efficient Self-Attention

La **différence** entre le **module d'attention** utilisé par le ***ViT*** et celui utilisé par le ***SegFormer*** réside dans l'introduction d'un **ratio de réduction R** permettant de passer d'une complexité $O(N^2)$, où N correspond au $H_i \times W_i$ de l'étape considérée, à une complexité $O(N^2/R)$. Cette réduction peut s'effectuer via :

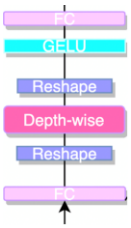
- une convolution de $C_i \times R$ canaux avec $K = R = \sqrt{R}$, permettant de réduire les dimensions de l'image par $\sqrt{R}$, et donc in fine la dimension d'une séquence par $\sqrt{R} \times \sqrt{R} = R$. On passe alors d'une entrée $H_i \times W_i$, C_i à une sortie $\frac{H_i \times W_i}{R}$, $C_i \times R$
- une couche linéaire pour revenir à C_i canaux

Mix-FFN

Le *ViT* original utilise un ***positional encoding (PE)* pour apprendre à partir de la position** des patchs aussi bien que de leur contenu. Cependant, dans le cas de la segmentation sémantique, la résolution change d'une étape à l'autre, ce qui implique de trouver un moyen d'adapter également ce *PE*, par interpolation par exemple.
Cela conduit à une diminution de la performance.
Les créateurs du *SegFormer* ont donc décidé de simplifier l'architecture en supprimant ce *PE*.

Pour tout de même apprendre à partir de l'information spatiale, les auteurs ont développé le *Mix-FFN* (pour *Feed-Forward Network*), qui utilise :

- des couches *fully-connected*
- une *depthwise convolution* 3 ×3 avec un *zero-padding* pour capturer l'information spatiale



$$\mathbf{x}_{out} = \text{MLP}(\text{GELU}(\text{Conv}_{3\times3}(\text{MLP}(\mathbf{x}_{in})))) + \mathbf{x}_{in}$$

Figure 4 : Mix-FFN

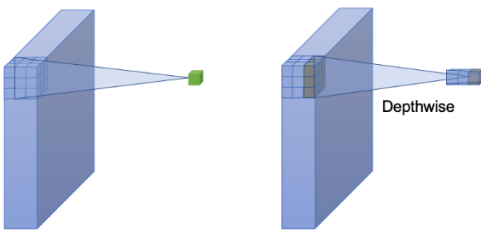


Figure 5 : convolution VS depthwise convolution

Lightweight All-MLP Decoder

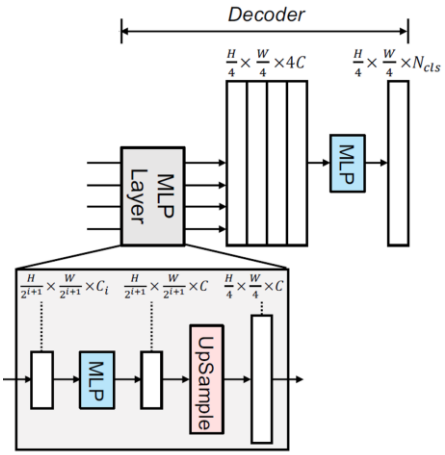


Figure 6 : MLP decoder

SegFormer présente un *decoder* très simple, composé de couches de neurones. Cela diminue grandement le nombre de paramètres à apprendre, par rapport à des architectures classiques devant entraîner de nombreuses couches convolutives successives avant d'arriver à la résolution de la segmentation.
La raison pour laquelle les auteurs se permettent cela est la grande qualité des *feature maps* obtenues. Celles-ci ont l'avantage de comporter des informations à différentes échelles. Le tout de manière très locale, mais aussi très globale, grâce au mécanisme d'attention.
Le fonctionnement :

- passage des 4 *feature maps* dans un premier bloc *MLP* pour unifier le nombre de canaux
  $(Hi \times Wi, Ci) \rightarrow (Hi \times Wi, C)$
- augmenter la taille des 3 dernières *feature maps* pour arriver à la taille de la première
  $(Hi \times Wi, C) \rightarrow \left(\frac{H}{4} \times \frac{W}{4}, C\right)$
- concaténation des 4 *feature maps* (*channel-wise*) :
  $\left(\frac{H}{4} \times \frac{W}{4}, C\right)_1, \left(\frac{H}{4} \times \frac{W}{4}, C\right)_2, \left(\frac{H}{4} \times \frac{W}{4}, C\right)_3, \left(\frac{H}{4} \times \frac{W}{4}, C\right)_4 \rightarrow \left(\frac{H}{4} \times \frac{W}{4}, 4C\right)$
- passage via un bloc *MLP* pour fusionner les *feature maps* concaténée :
  $\left(\frac{H}{4} \times \frac{W}{4}, 4C\right) \rightarrow \left(\frac{H}{4} \times \frac{W}{4}, C\right)$
- enfin un dernier *MLP* pour arriver au nombre de classes à détecter :
  $\left(\frac{H}{4} \times \frac{W}{4}, C\right) \rightarrow \left(\frac{H}{4}, \frac{W}{4}, N_{classes}\right)$

Modélisation

Pour faire notre preuve de concept, nous comparons le modèle *SegFormer* à un modèle *baseline* basé sur l'architecture *Unet* avec une *backbone Resnet*, qui était ressorti gagnant dans le cadre du projet précédent.

Généralités et Hypothèses

Métrique

Nous utilisons le *mean IoU score* (ou indice de Jaccard), moyenne des *intersection over union* de chaque classe prédite. L'*IoU* d'une classe est calculé en divisant l'aire de l'intersection entre la zone prédite par le modèle et la zone réelle de l'objet, par l'aire de leur union. Il s'agit d'une mesure de la **similarité** entre ces zones, qui se calcule donc comme le rapport entre :

- la zone correctement prédite (les *TP*)

et

- les zones correctement prédite (*TP*), incorrectement prédite (*FP*) et réelle non détectée (*FN*)

$$IoU = \frac{TP}{(TP + FP + FN)}$$

Figure 7 : IoU

Nous capturons également le **temps d'entraînement** des deux modèles.

Fonction de perte

La fonction de perte « classique » pour les modèles de classification à plusieurs catégories est la *Categorical Cross Entropy*.

$$\mathrm{CE}\,(p,\hat{p}) = -\left(p\log\left(\hat{p}\right) + (1-p)\log\left(1-\hat{p}\right)\right)$$

Figure 8 : Categorical Cross Entropy (cas binaire)

Nous avions dans le cadre du projet précédent utilisé d'autres fonctions de pertes, et notamment une combinaison entre la *Focal Loss* et la *Dice Loss*, en jouant aussi sur le paramètre `class_weights`. L'implémentation du *SegFormer* mise à disposition par *Hugging Face* ne nous permettant pas de modifier simplement la fonction de perte, nous nous sommes contentés de la *Categorical Cross Entropy.*

Quelles variations de `SegFormer` et `Unet-Resnet` ?

Ces deux modèles possèdent plusieurs variantes, qui diffèrent en termes de complexité / **nombres de paramètres** :

- *SegFormer* : 5 variantes, de *B0* à *B5*
- *Unet-Resnet* : 5 variantes, de *Resnet18* à *Resnet152*

Afin de comparer les deux algorithmes de segmentation **sur des bases proches**, nous faisons le choix de **sélectionner des variantes équivalentes en termes de nombres de paramètres**.
Après plusieurs tests, nous avons sélectionné le *SegFormer-B1* (la version B2 étant trop gourmande en capacités de calcul) :

```
================================================================
Total params: 13679816 (52.18 MB)
Trainable params: 13679304 (52.18 MB)
Non-trainable params: 512 (2.00 KB)
```

Figure 9 : SegFormer B1 - nombre de paramètres

Le modèle utilisé dans le cadre du projet précédent était le *Unet-Resnet34*, qui possèdent bien plus de paramètres :

```
================================================================
Total params: 24457169 (93.30 MB)
Trainable params: 24439819 (93.23 MB)
Non-trainable params: 17350 (67.77 KB)
```

Figure 10 : Unet-Resnet34 - nombre de paramètres

Nous nous sommes alors tournés vers le *Unet-Resnet18* :

```
================================================================
Total params: 14341585 (54.71 MB)
Trainable params: 14331659 (54.67 MB)
Non-trainable params: 9926 (38.77 KB)
```

Figure 11 - Unet-Resnet18 - nombre de paramètres

Nous avons donc bien deux modèles avec des tailles comparables.

Réduction du temps d'apprentissage

Afin de prendre en compte nos capacités matériel et de limiter le temps d'entraînement, nous faisons des choix :

- Limitation de la taille des images : 256×512
- Nombre d'*epochs* fixe : 10

Augmentation de données

L'augmentation de données **crée artificiellement plus de données pour l'entraînement**, augmentant la **robustesse** du modèle.
C'est une technique pour lutter **contre le sur-apprentissage**. L'augmentation de données expose les modèles à une plus grande variété de données, ce qui les empêche de trop s'adapter au set d'entraînement et améliore leur performance sur de nouvelles données.
Dans le domaine de la Computer Vision, celle-ci se traduit par différentes techniques, qui peuvent être par exemple **géométriques** (comme une rotation) ou **au niveau du pixel** (comme des modifications de couleur).
Dans le cadre du projet précédent, nous avons **démontré l'opportunité** d'utiliser ces techniques d'augmentation de données. Nous utiliserons la même liste de transformations :

```python
# define augmentations for training
list_of_transforms = [
    A.OneOf(
        [
            A.FancyPCA(p=1, alpha=1),
            A.HueSaturationValue(p=1, hue_shift_limit=20, sat_shift_limit=30, val_shift_limit=20),
            A.ColorJitter(p=1, brightness=0.2, contrast=0.2, saturation=0.2),
        ],
        p=0.5
    ),
    A.RandomShadow(
        p=0.5,
        shadow_roi=(
            0, 0.4,
            1, 1
            ),
        num_shadows_lower=1,
        num_shadows_upper=4,
        shadow_dimension=5
        ),
    A.CoarseDropout(
        p=0.5,
        min_holes=2,
        max_holes=8,
        min_height=0.05,
        max_height=0.1,
        min_width=0.025,
        max_width=0.05
        )
]
```

Figure 12 - liste des transformations (idem P8)

Entraînement *Baseline*

Nous entraînons donc un modèle Unet-Resnet18 sur nos données préparées et augmentées.

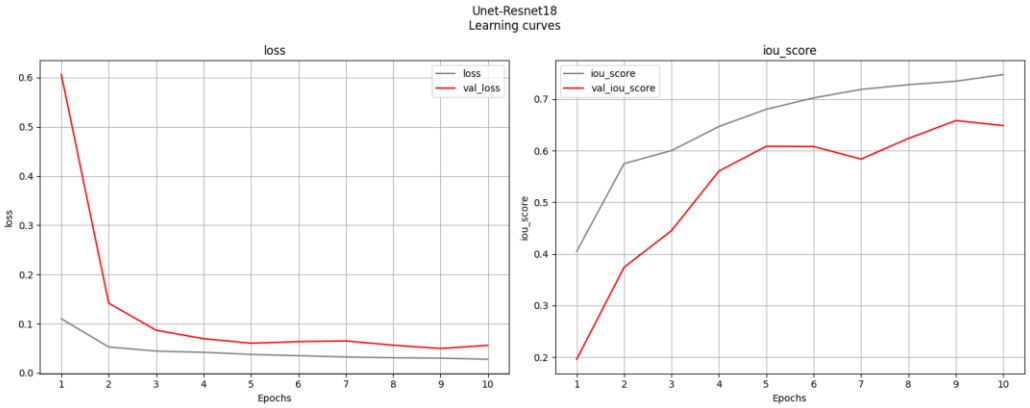


Figure 13 : apprentissage Unet-Resnet18

Entraînement *SegFormer*

Un modèle `TFModel` *HuggingFace* est compatible avec le *framework keras*, mais conserve des particularités, notamment, au niveau du format :

- des données à lui fournir pour l'entraînement
- du résultat de la méthode `predict()`

Il a notamment besoin d'image carrées en entrée. Nous passons donc d'une taille 256×512 à 384×384, qui permet de conserver sensiblement la même résolution tout en conservant un multiple de 32.
Nous devons transformer aussi le générateur de données :

- utilisation à la place d'un `tf.data.Dataset`
- création de celui-ci grâce à l'utilisation d'une méthode `load_data()` qui réalise les mêmes opérations que précédemment, alliée avec la méthode `map()`
- utilisation de `tf.numpy_function()` pour entre autres pouvoir continuer à utiliser la librairie `albumentations`
- préparation des images pour le `SegFormer` : normalisation avec des paramètres `mean` et `std`
- passage à une configuration *"channel first"*, pour la compatibilité avec *HuggingFace*
- *batchs* sous forme de dictionnaires `{"pixel_values" : image, "labels" : mask}`, pour la compatibilité avec *HuggingFace*
- utilisation des fonctionnalités de `tf.data.Dataset` pour l'échantillonnage, la mise en `batchs` et l'éventuel `shuffle`

Enfin, les auteurs du SegFormer indiquent dans l'article : « *The learning rate was set to an initial value of 0.00006 and then used a "poly" LR schedule with factor 1.0 by default* ». On a donc utilisé un `keras.optimizers.schedules.PolynomialDecay` pour satisfaire cette condition d'utilisation.

Nous pouvons alors entraîner le *SegFormer* :

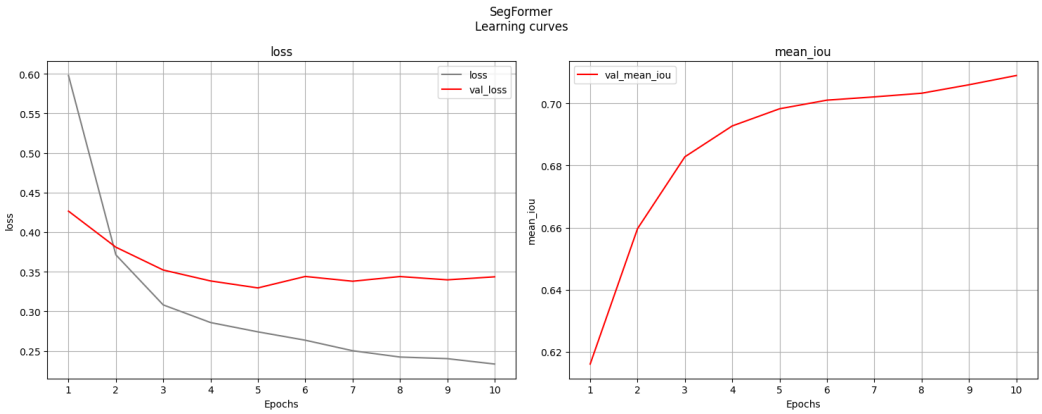


Figure 14 - apprentissage SegFormer

Nota : le `tf.keras.callbacks.History()` issu de l'entraînement d'un `TFModel` **HuggingFace** ne nous donne pas accès aux valeurs de la métrique sur les données d'entraînement.

Synthèse des résultats

Voici les résultats obtenus :

model	backbone	val_iou	training_time
Unet	Resnet18	0.65	3 h
SegFormer	MIT-B1	0.71	15 h

Figure 15 : résultats modélisation

L'algorithme ***SegFormer*** **surpasse l'*Unet*** dans les conditions de l'expérimentation, au prix d'un **temps d'entraînement bien plus important**.

Nous pouvons comparer les deux modèles sur quelques images :

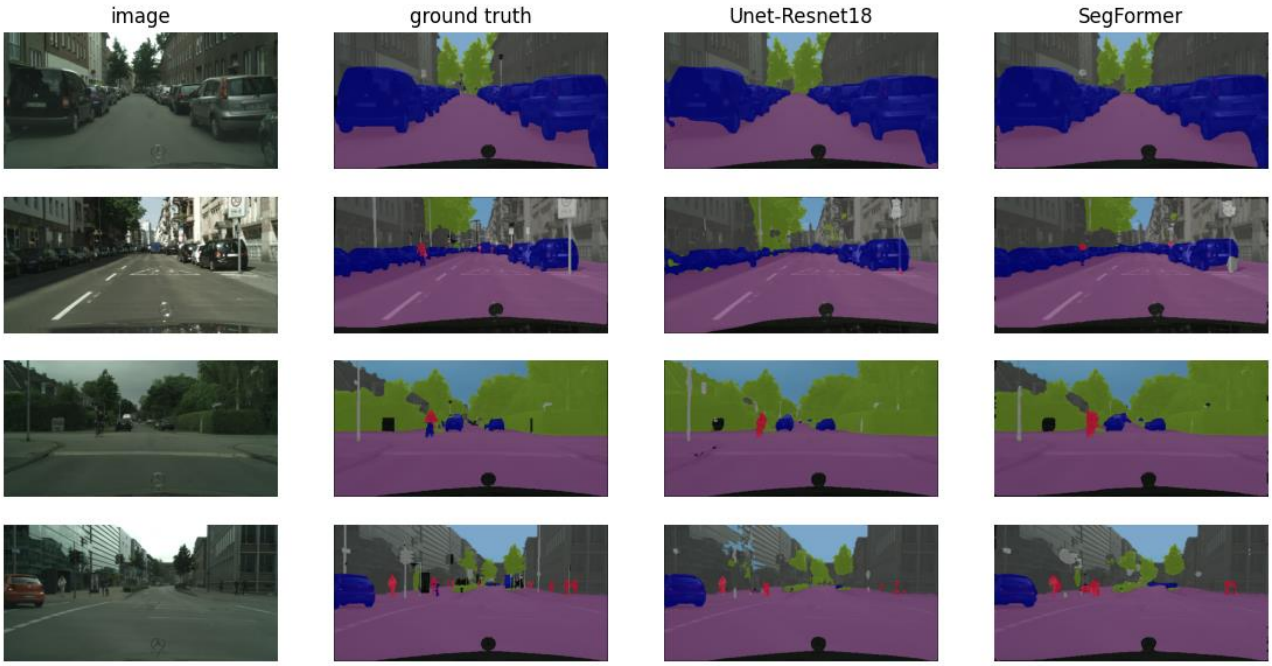


Figure 16 : Comparaison des prédictions sur quelques exemples

Le *SegFormer* a moins de difficultés à détecter les humains, même si la forme est plus grossière. Il se comporte aussi mieux sur les bâtiments et les espaces verts. L'Unet-Resnet rencontre des difficultés. Il a par exemple confondu le ciel avec son reflet sur un immeuble, manqué un panneau ou encore intégré des maisons dans les espaces verts.

Nous pouvons comparer les performances par classe sur l'ensemble du set de validation :

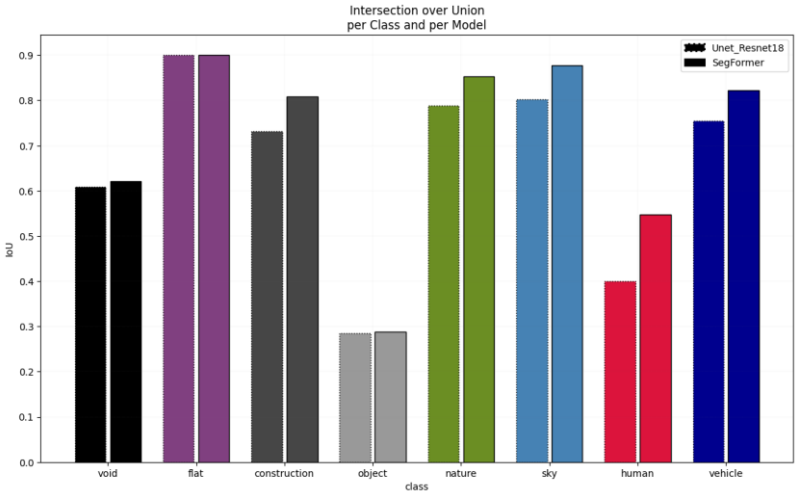


Figure 17 : IoU par macro-catégorie, pour les deux modèles

La prédominance du *SegFormer* se vérifie **sur l'ensemble des classes**. Il se distingue particulièrement sur la **détection d'humains**.
Nous pouvons conclure que le nouveau modèle **améliore bien la performance** par rapport l'algorithme précédemment utilisé. Cela se vérifie aussi bien d'une manière globale que pour **chaque macro-catégorie prise individuellement**. Cette performance se confirme sur le set de test (*mean IoU* de 0.72). Le *SegFormer* est **en revanche bien plus long à entraîner**.

Explicabilité

Il n'est pas aisé d'appliquer des techniques d'interprétabilité pour les modèles de Computer Vision. En effet les images sont des données non structurées, il n'est donc pas possible de ressortir aisément des features (comme pour des données tabulaires). Il n'est pas non plus possible de ressortir des mots (comme pour des données non structurées de type texte). Des techniques ont néanmoins été développées, et l'on peut en retrouver certaines au sein de la librairie `Xplique`.

La librairie `Xplique`

Cette librairie a été développée par les équipes de *DEpendable & Explainable Learning* (*DEEL*), et met à disposition des outils d'interprétabilité pour des modèles de *Computer Vision*.

Elle permet d'explorer différentes facettes de l'explicabilité en *CV* :

- la ***Feature attribution***, celle qui va nous intéresser :
    - **méthode locale**
    - d'une manière générale, consiste à modifier l'image en entrée et à mesurer l'impact que cela a sur la sortie du modèle
    - il existe plusieurs méthodes :
        - faire varier sensiblement chaque pixel. On mesure alors l'impact précis de ce pixel
        - cacher une partie de l'image (par exemple avec un *patch* noir) et faire glisser ce *patch* petit à petit. On peut alors déduire des variations du modèle les zones critiques
        - utiliser directement les *features maps*, les poids, les gradients du modèle. Par exemple, pour une image donnée, pondérer les *features maps* avec le gradient associé à la classe prédite. Cela permet de mettre en valeur les zones ayant le plus contribué à cette classe.

- la ***Feature visualization*** :
    - **méthode globale**
    - nécessite d'avoir accès aux valeurs de chaque neurone. Nous ne pourrons donc pas l'utiliser sur le *SegFormer*.
    - permet d'expliquer comment un neurone spécifique, un canal spécifique ou encore une couche entière a construit sa compréhension des images d'entraînement
    - pour réaliser cela :
        - on sélectionne l'élément à expliquer
        - on calcule le gradient juste entre cet élément et l'entrée du modèle obtenue avec une image aléatoire
        - on utilise ce gradient, très partiel donc, pour optimiser l'image d'entrée afin qu'elle maximise la valeur de l'élément
        - on obtient ainsi une image caractérisant l'élément en question

- le ***Concept-based*** :
    - **méthode globale**
    - applicable pour de la classification ET nécessite d'avoir accès aux couches du modèles. Nous ne l'utiliserons donc pas sur le *Segformer*
    - l'idée est de recherche non pas des zones d'une image, mais des concepts familiers pour un humain
    - une méthode, appelée *CRAFT* consiste à :
        - choisir une classe à explorer, par exemple "chien"
        - choisir une sélection d'images pour lesquelles le modèle a prédit la classe "chien"
        - choisir une couche du modèle :
            - avant cette couche, sont calculées les valeurs d'activation de nos images
            - après cette couche, les logits
        - sur toutes les images, on choisit aléatoirement des crops que l'on passe au modèle, et on stocke les activations de celles-ci
        - on factorise ces activations pour en extraire les concepts
        - on détermine l'importance globale de chaque concept
        - il est même possible de déterminer les concepts les plus importants pour une image donnée, et de les localiser dans l'image

Pour le *Segformer* nous seront obligés de nous contenter de la *Feature Attribution*. `Xplique` met à disposition un module dédié à la segmentation sémantique. L'idée directrice est d'expliquer une partie de l'image d'une certaine classe (**à pointer à la main**), et d'utiliser une méthode de *Feature Attribution* sur cette zone, en revenant en quelque sorte à un problème de classification.

Compatibilité `transformers` et `Xplique`

`Xplique` est conçu pour fonctionner avec le *framework TensorFlow*. Un `TFModel` *Huggingface* est compatible, mais ce n'est tout de même pas un `tensorflow.keras.Model`. La librairie met à disposition un `wrapper` pour rendre compatible certaines fonctionnalités également avec *PyTorch*, mais rien pour *HuggingFace*.

Après étude du code du `wrapper` *PyTorch* et échange avec les auteurs de la librairie, nous avons créé la classe `WrapperSegFormer`, qui nous a alors permis d'utiliser certaines fonctionnalités de la librairie `Xplique`.

Exemples d'image et zones à expliquer

Nous avons choisi deux exemples, et affiché une grille afin de sélectionner les zones où l'on souhaite interpréter la prédiction du modèle :

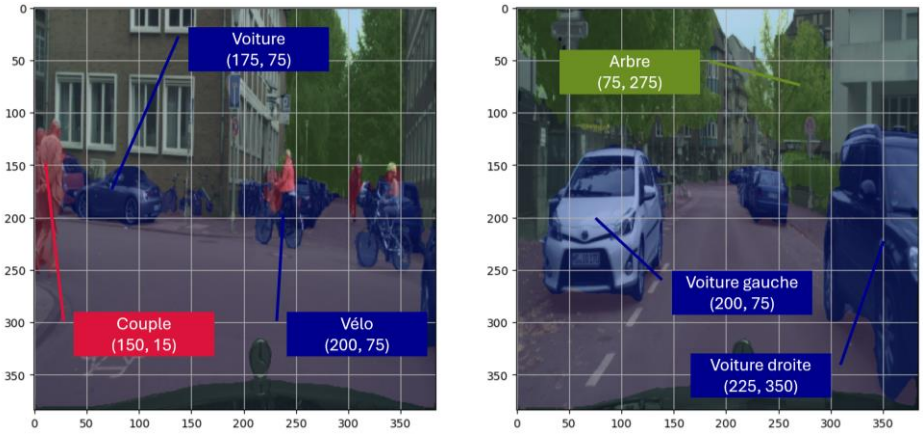


Figure 18 : deux exemples de prédiction, et sélection de zones à expliquer

Explications

Nous créons tout d'abord nos `targets` permettant de spécifier ce que l'on doit expliquer. Nous utilisons 2 fonctions : `get_connected_zone` (pour toute la zone avec la classe en question) et `get_in_out_border` (pour la limite entre la zone sélectionnée et les autres zones).
Nous créons ensuite un `explainer` permettant d'appliquer une technique de *feature attribution* à partir de notre modèle rendu compatible grâce au `WrapperSegFormer`. Nous choisissons `SobolAttributionMethod`, qui n'est pas juste une technique reprise au sein de `Xplique`, car elle a été développée directement par ses créateurs.

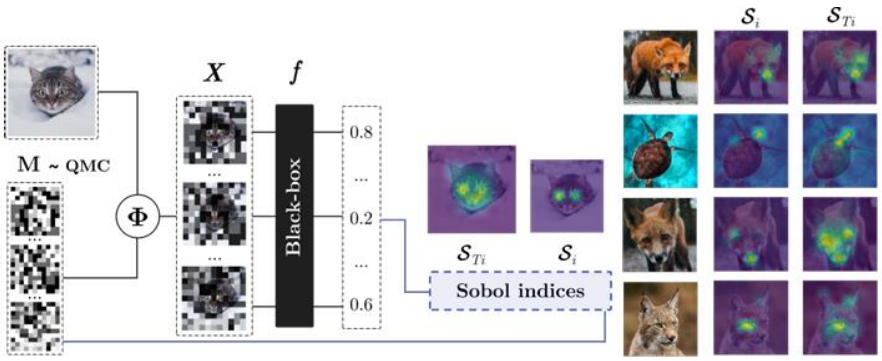


Figure 19 : Schéma de la méthode d'attribution basée sur les indices de Sobol

Nous pouvons enfin générer chaque explication à partir d'une `target` grâce à la méthode `.explain()`

Il est alors possible de visualiser les explications, c'est-à-dire les pixels de l'image ayant le plus contribué à la détection de telle zone, ou la limitation d'une zone par rapport aux autres.
Pour l'image 1 :

Figure 20 : Explications zones et limites de l'image 1

Ce que l'on peut en retenir :

- couple :
  - les bras et le dos semblent avoir conditionné la détection de cette zone
  - le dos et les jambes ont plus participé à la limitation de cette zone
- voiture :
  - zone : bas de caisse et roues
  - séparations : l'arrière de la voiture
- vélo :
  - zone : fondu avec le groupe de voitures en arrière-plan, mais le haut des roues semble avoir conduit à l'inclusion du vélo dans le groupe
  - limite : la roue arrière a eu beaucoup d'influence

Pour l'image 2 :

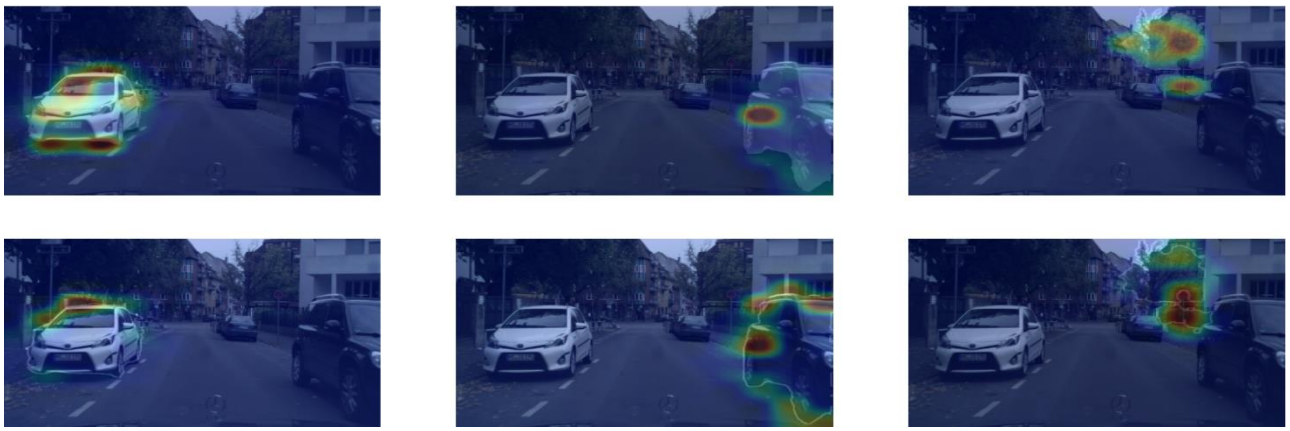


Figure 21 : Explications zones et limites de l'image 2

Ce que l'on peut en retenir :

- voiture de gauche :
  - zone : pare-brise, feu, roues avant
  - limite : le haut de la voiture a eu un gros impact sur le positionnement de cette séparation
- demi-voiture de droite :
  - zone : dessus roue arrière
  - séparations : idem, ainsi que la haut de la voiture
- arbre de droite :
  - zone : zones les plus denses
  - limite : le panneau a beaucoup d'influence sur cette séparation

Les limites et les améliorations possibles

Les résultats sont globalement positifs pour le modèle *SegFormer*, mais les conditions du test n'ont sans doute pas permis de l'exploiter convenablement. En effet le matériel utilisé ne nous a pas permis d'explorer pleinement les capacités du modèle. Nous avons dû faire des choix forts : limitation importante de la taille des images, du nombre d'*epochs*, de la taille du modèle lui-même (version *B1* seulement).
Pour **améliorer les performances**, il serait possible de :

- utiliser la taille originale des images
- explorer d'autres techniques d'augmentation de données, comme des algorithmes de génération d'images
- explorer d'autres versions du *SegFormer*. L'algorithme est disponible jusqu'au B5.
- ne pas limiter le nombre d'*epoch*, et utiliser un *callback* de type `EarlyStopping` à la place par exemple (cela permettra notamment de profiter pleinement du `PolynomialDecay`)

Pour permettre la mise en pratique de ces solutions, il serait possible d'utiliser un espace de travail *AzureML* pour disposer de capacités bien plus importantes pour l'entraînement et l'inférence.

Pour **améliorer l'explicabilité du modèle**, nous pourrions tester d'**autres méthodes** et les comparer. Pour cela la librairie `Xplique` met à disposition un module `metrics` d'évaluer la qualité d'une explication. En effet il ne faut pas se fier seulement à l'œil, car ce n'est pas parce qu'une explication fait sens pour l'être humain que cela signifie qu'elle reflète ce qu'il se passe réellement à l'intérieur du modèle (**biais de confirmation**).

Il serait également intéressant d'explorer des méthodes globales (***Feature visualization*** ou ***Concept-based***), même si comme nous l'avons vu cela implique d'avoir un accès beaucoup libre aux couches du modèle (utiliser une autre implémentation plus transparente ?).