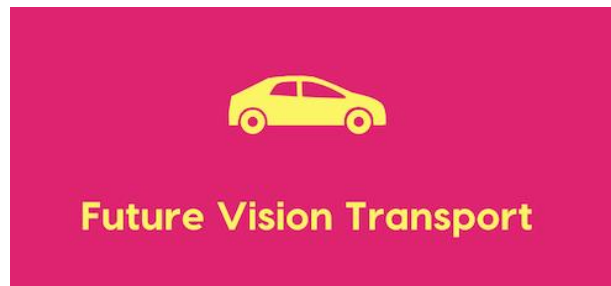


Segmentation d'images



Note Technique

Cette note technique accompagne et présente le travail effectué sur la partie **Segmentation d'images** du projet de *Future Vision Transport*.

I. Le projet

1. Véhicule autonome

Future Vision Transport est une entreprise spécialisée dans la conception de **systèmes embarqués de vision par ordinateur pour les véhicules autonomes**.

Le développement du système comporte plusieurs facettes :

- acquisition des images en temps réel
- traitement des images
- segmentation des images, **qui sera développée dans la présente note**
- système de décision

L'objectif est de développer **un premier modèle de segmentation d'image** :

- alimenté par le système de traitement des images
- et qui devra alimenter le système de prise de décision

2. Le travail à effectuer

Sur la partie segmentation d'image, nous devons :

Manipuler le jeu de données mis à notre disposition :

Le dataset d'entraînement est *Cityscape*, qui comporte notamment des **photos prises depuis un véhicule** ainsi que les **masques associés**. Le masque 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 :

name	id	trainId	category	catId
'unlabeled'	, 0 ,	255 ,	'void'	, 0
'ego vehicle'	, 1 ,	255 ,	'void'	, 0
'rectification border'	, 2 ,	255 ,	'void'	, 0
'out of roi'	, 3 ,	255 ,	'void'	, 0
'static'	, 4 ,	255 ,	'void'	, 0
'dynamic'	, 5 ,	255 ,	'void'	, 0
'ground'	, 6 ,	255 ,	'void'	, 0
'road'	, 7 ,	0 ,	'flat'	, 1
'sidewalk'	, 8 ,	1 ,	'flat'	, 1
'parking'	, 9 ,	255 ,	'flat'	, 1
'rail track'	, 10 ,	255 ,	'flat'	, 1
'building'	, 11 ,	2 ,	'construction'	, 2
'wall'	, 12 ,	3 ,	'construction'	, 2
'fence'	, 13 ,	4 ,	'construction'	, 2
'guard rail'	, 14 ,	255 ,	'construction'	, 2
'bridge'	, 15 ,	255 ,	'construction'	, 2
'tunnel'	, 16 ,	255 ,	'construction'	, 2
'pole'	, 17 ,	5 ,	'object'	, 3
'polegroup'	, 18 ,	255 ,	'object'	, 3
'traffic light'	, 19 ,	6 ,	'object'	, 3
'traffic sign'	, 20 ,	7 ,	'object'	, 3
'vegetation'	, 21 ,	8 ,	'nature'	, 4
'terrain'	, 22 ,	9 ,	'nature'	, 4
'sky'	, 23 ,	10 ,	'sky'	, 5
'person'	, 24 ,	11 ,	'human'	, 6
'rider'	, 25 ,	12 ,	'human'	, 6
'car'	, 26 ,	13 ,	'vehicle'	, 7
'truck'	, 27 ,	14 ,	'vehicle'	, 7
'bus'	, 28 ,	15 ,	'vehicle'	, 7
'caravan'	, 29 ,	255 ,	'vehicle'	, 7
'trailer'	, 30 ,	255 ,	'vehicle'	, 7
'train'	, 31 ,	16 ,	'vehicle'	, 7
'motorcycle'	, 32 ,	17 ,	'vehicle'	, 7
'bicycle'	, 33 ,	18 ,	'vehicle'	, 7
'license plate'	, -1 ,	-1 ,	'vehicle'	, 7

Figure 2 : classes et catégories

Le jeu de données est **volumineux**, il faut donc le traiter de façon adéquate.

Modéliser :

Plusieurs solutions sont testées jusqu'à obtenir notre modèle de segmentation.

Mettre à disposition le modèle au sein d'un API :

Cette API prend en **entrée une image** et **renvoie le masque prédit**.

Concevoir une application web :

Cette application est l'**interface utilisateur** pour utiliser l'API sur des images de test. Elle affiche l'image, le mask réel et le mask prédit.

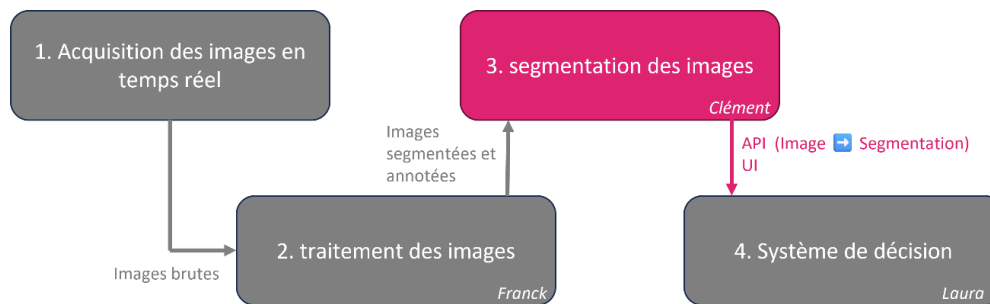


Figure 3 : organigramme

II. Segmentation d'image, principe

1. Qu'est-ce que la segmentation d'image ?

La segmentation d'image est une technique d'intelligence artificielle faisant partie des tâches principales et historiques de la *Computer Vision* :

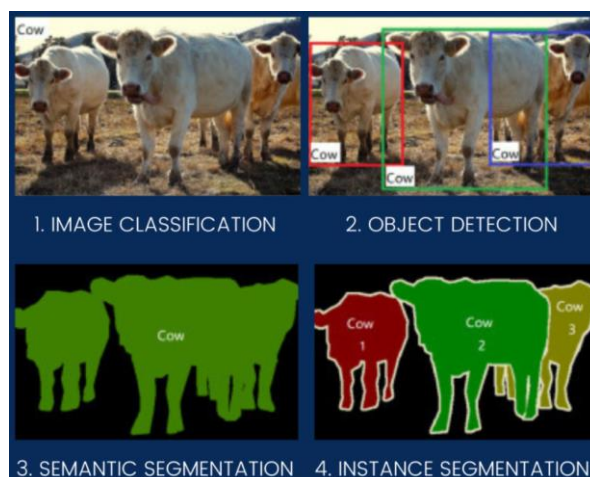


Figure 4 : tâches historiques de la Computer Vision

La **classification d'image** consiste à identifier une classe au sein de l'image.

Il est aussi possible de faire de la **détection d'objet**, qui consiste à indiquer la position de la classe détectée (via une *bounding box*).

La **segmentation d'image** consiste à **classifier chaque pixel et lui attribuer un label**. Cela a pour effet de séparer l'image en zones bien délimitées. Il en existe deux types :

- Segmentation **sémantique**, qui attribue le même label à tous les pixels des objets d'une même classe

- Segmentation **d'instance**, qui va plus loin, et applique de la détection d'objet préalable. Elle attribue alors aux pixels de deux instances d'une même classe deux labels différents.

Pour le projet nous réalisons de la segmentation sémantique.

2. Fonctionnement de principe

Historiquement la segmentation sémantique utilise des algorithmes basés sur l'architecture Encoder-Decoder.

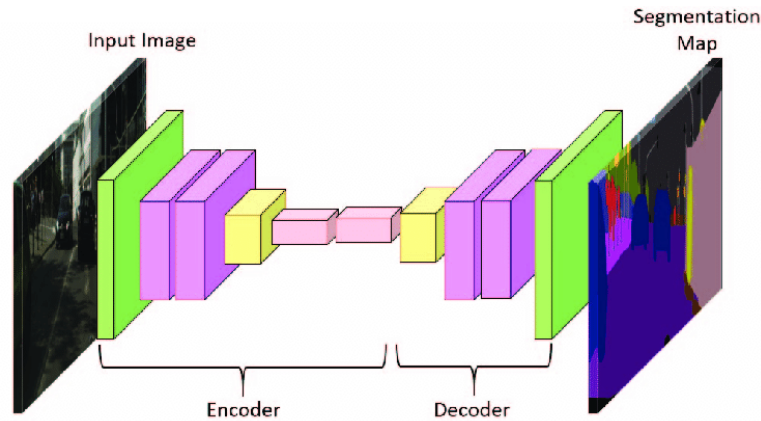


Figure 5 : architecture encoder decoder

L'**encoder** se charge d'**extraire les caractéristiques des images** grâce à un enchaînement de **couches de convolution** produisant des **features maps**.

Des couches de **Pooling réduisent la taille** des images entre chaque bloc convolutif afin de résumer l'information, **conserver les features les plus importantes** et réduire le surapprentissage. On dit que l'encoder s'occupe de la phase de **downsampling**.

Ce fonctionnement est le même que pour les modèles de classification d'image. On utilise d'ailleurs des modèles de classification connus (sans leurs couches fully-connected de décision) dans la plupart des cas. On parle alors de **backbone**.

Le **decoder** se charge de la phase d'**upsampling**. En effet l'objectif de la segmentation est d'obtenir une classification de chaque pixel de l'image initial. Il nous faut donc revenir à une taille de sortie identique à l'entrée. Cela peut s'opérer de différentes manières (max unpooling, transposed convolution, nearest neighbors, interpolation bilinéaire, etc.).

En fin de processus, l'objectif est d'obtenir autant de maps que de catégories (via une 1x1 conv par exemple, qui agit comme une « feature map pooling layer »), sur lesquelles on applique une fonction de décision (ex **Softmax**). En retenant pour chaque pixel la catégorie la plus probable, on obtient alors enfin une image de sortie, le **mask**.

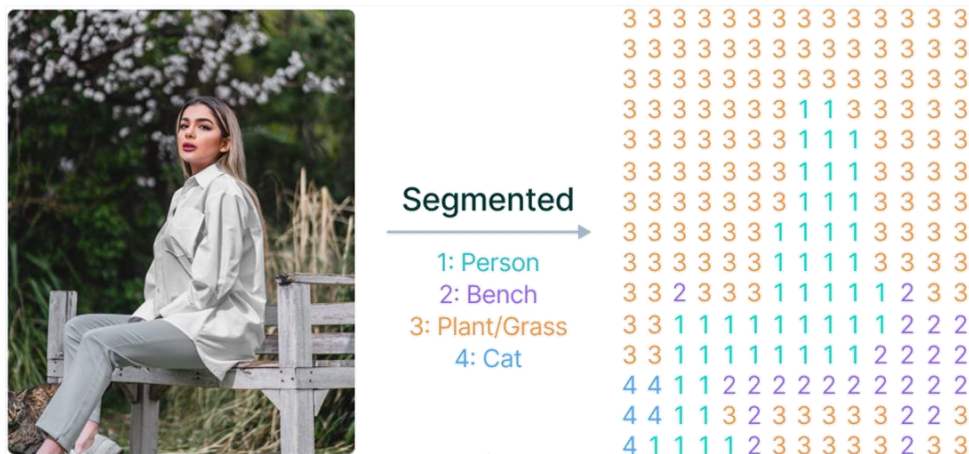


Figure 6 : Illustration de la segmentation

3. Quelles architectures disponibles ?

Plusieurs architectures ont été développées afin d'optimiser la méthode décrite ci-dessus :

U-net : cette architecture a été la première à **corriger un problème** récurrent avec l'architecture basique : la **perte d'information spatiale**. En effet, lors de la phase de *downsampling*, la localisation des features sur les maps est de moins en moins précise (cela en est d'ailleurs l'objectif principal : repérer des éléments importants, peu importe sa localisation ou son contexte). Le souci est que contrairement à de la classification d'image, la localisation a une importance capitale pour la segmentation. Pour régler le problème, U-net intègre des connexions (par concaténation) entre les couches de l'encoder et du decoder de même définition. Ainsi, **l'information spatiale est mieux prise en compte** au cours de l'apprentissage.

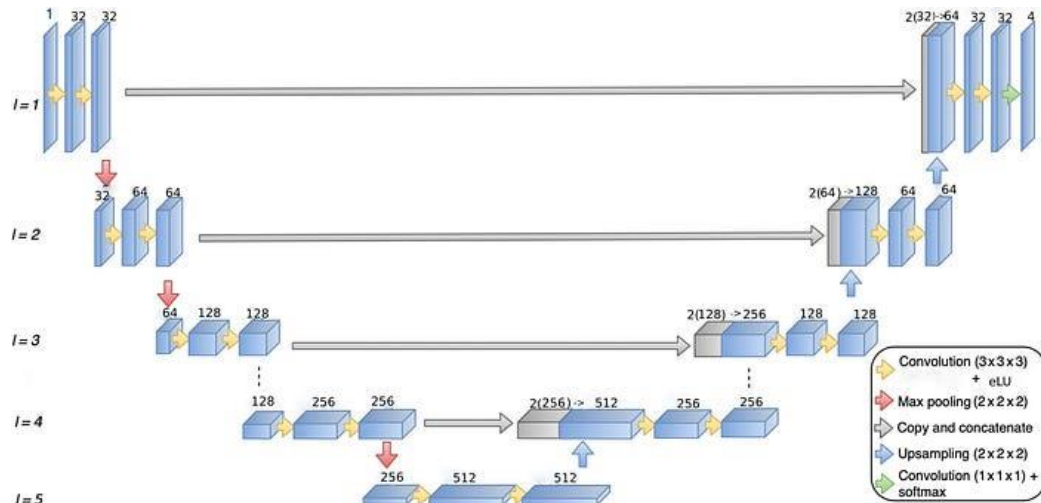


Figure 7 : U-net

Pour le projet nous utilisons ce modèle U-net.

Il existe d'autres architectures comme **LinkNet**, **HyperSeg**, ou **FPN**.

4. Quels backbones ?

Pour le projet, nous testons deux backbones :

VGG16 - Un réseau convolutif historique développé par l'université d'Oxford :

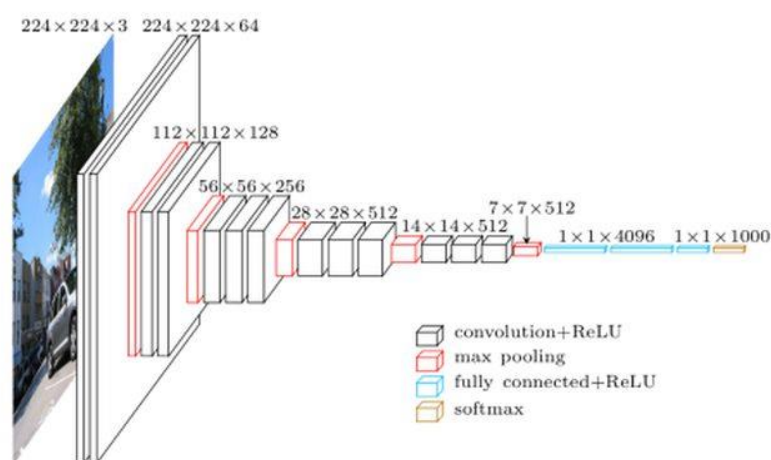


Figure 8 : VGG16

Resnet 34 – Modèle intégrant des connexions résiduelles :

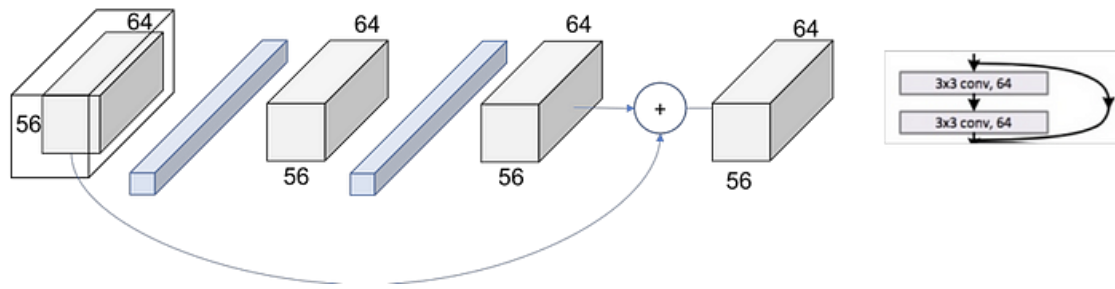


Figure 9 : Exemple connexion résiduelle

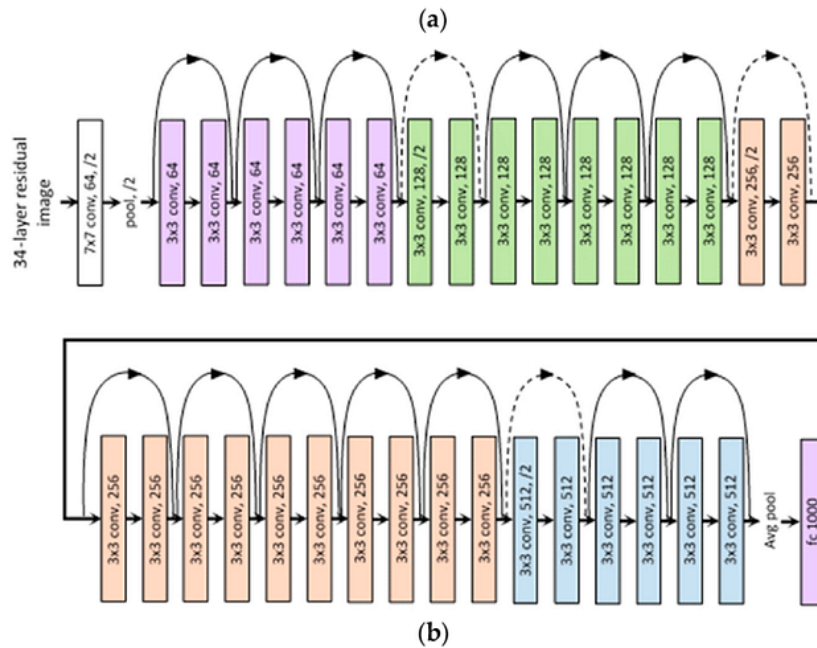


Figure 10 : Resnet34

5. Les outils utilisés

Pour développer le projet, nous utilisons :



TensorFlow



- Keras et Tensorflow pour :
 - la mise à disposition des images via un générateur de données
 - l'export du modèle pour l'API
- Segmentation_models, librairie spécialisée pour la segmentation d'image, pour :
 - La construction du modèle de segmentation
 - L'intégration du backbone avec des poids pré-entraînés
- Albumentations, librairie d'augmentation de données, pour :
 - Explorer différentes techniques
 - Intégrer l'augmentation d'image dans le générateur de données
- Flask pour la construction de la page Web UI
- FastAPI pour la construction de l'API
- Docker pour créer les conteneurs exécutables pour l'API et l'UI
- Azure WebApp Services pour le déploiement
- GitHub pour l'intégration continue du code
- GitHub Action pour le déploiement continu

III. Préparation

1. Les données

Le jeu de données Cityscapes mis à notre disposition 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 :

- Masques couleur : chaque classe se voit attribuer une couleur différente.
- Masques 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.

2. Générateur de données

Pour traiter ce dataset volumineux (11 Go), nous créons 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 x 1024 x 3) → (256 x 512 x 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 x 512) → (256 x 512 x 8)
- Met à disposition le batch créé
- (facultatif) A la fin de chaque `epoch`, **mélange** les chemins d'accès des images/masks

3. Métrique d'évaluation

Utilisation du `mean IoU score` (ou *indice de Jaccard*), moyenne des *intersection over union* de chaque classe prédite. L'**IoU** 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 11 : IoU

Nous enregistrons également le coefficient de *Dice*, ou **FScore**, qui est une autre mesure de similarité, qui se calcule comme le rapport entre :

- 2 X la zone correctement prédite (les TP)
- et
- les zones réelle (TP+FN) et prédite (TP+FP)

Il s'agit également de la moyenne harmonique entre :

- la part de zone réelle correctement prédite (appelée Recall)
- et
- la part de zone prédite qui est bien réelle (appelée Precision)

Enfin, capture du **temps d'entraînement** des modèles.

4. Fonction de perte

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

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

Figure 12 : Categorical Cross Entropy (cas binaire)

Mais celle-ci n'est pas forcément la plus utilisée pour la segmentation.

On lui préfère parfois la *Focal Loss*, qui en est une variation, et qui permet de pénaliser l'impact des exemples faciles et de mettre en avant les exemples les plus compliqués. Cela peut par exemple permettre au modèle de se concentrer plus sur les délimitations entre classes.

$$FL(p, \hat{p}) = -(\alpha(1 - \hat{p})^\gamma p \log(\hat{p}) + (1 - \alpha)\hat{p}^\gamma(1 - p) \log(1 - \hat{p}))$$

Figure 13 : Focal Loss (cas binaire)

On peut utiliser également la Dice Loss qui est calculé à partir du coefficient de Dice (**= 1 - FScore**). Cette fonction de perte a la particularité de se calculer à l'échelle de globale.

$$DL(p, \hat{p}) = 1 - \frac{2 \sum p_{h,w} \hat{p}_{h,w}}{\sum p_{h,w} + \sum \hat{p}_{h,w}}$$

Figure 14 : Diss Loss (cas binaire)

Pour le projet, nous utilisons **une combinaison de la Focal Loss et de la Dice Loss**, ce qui nous permet de considérer l'échelle locale et l'échelle globale.

Nota : la Dice Loss possède un argument **class_weights** permettant d'avantager ou de pénaliser telle ou telle classe (lors du calcul de la moyenne des FScores de chaque classe). Dans un premier temps, nous n'utilisons pas cette possibilité.

5. 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).

Pour le projet, nous visualisons quelques techniques, puis nous en sélectionnons quelques-unes :

```
# 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 15 : Augmentations pour l'entraînement

Les techniques purement géométriques sont écartées, partant du principe que pour notre cas d'usage, la voiture autonome, le positionnement de la caméra sur le véhicule, son orientation ou encore le zoom n'avaient pas à changer. Le modèle n'a par exemple pas d'intérêt à apprendre des cas où le ciel est en bas, et nous partons du principe que la caméra du système embarqué sera toujours placée au même endroit sur le véhicule (dans notre cas à gauche).

Il en est de même avec les transformations jouant sur la **netteté** ou sur le **rognage**.

Les opérations **sélectionnées** :

- Altération des **couleurs** et de leur **saturation** (exemple : un piéton peut avoir un t-shirt rouge ou un t-shirt violet, le modèle doit pouvoir le détecter)
- Intégration de formes géométriques jouant le rôle d'**ombres**
- Intégration de carrés noirs de différentes formes (cela peut permettre de modéliser un **obstacle visuel**)

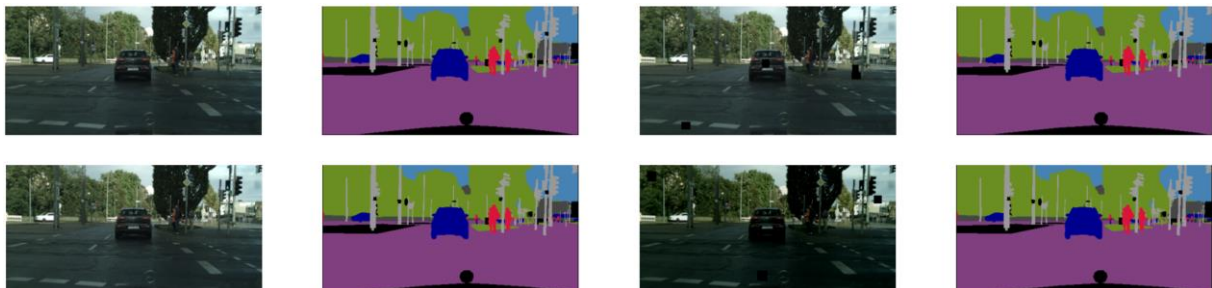




Figure 16 : Augmentation de données – visualisation

IV. Modélisation

Nous testons et évaluons différents modèles dans cette première phase du projet. Afin de **réduire le temps d'apprentissage**, nous faisons des choix :

- Limitation du jeu d'entraînement à 512 images
- Sets de validation et test = 128 images, avec une répartition 70/30
- Limitation de la taille des images à (256 x 512)

Pour l'entraînement lui-même, nous ne fixons pas le nombre d'`epochs`. Nous utilisons un callback `EarlyStopping` avec une `patience` = 3 et une période de `warm-up` `start_from_epoch` = 10.

1. Baseline

Nous entraînons tout d'abord un **modèle simple**, basé sur un réseau convolutif peu profond et sans *skip-connections*, qui nous sert de *baseline*.

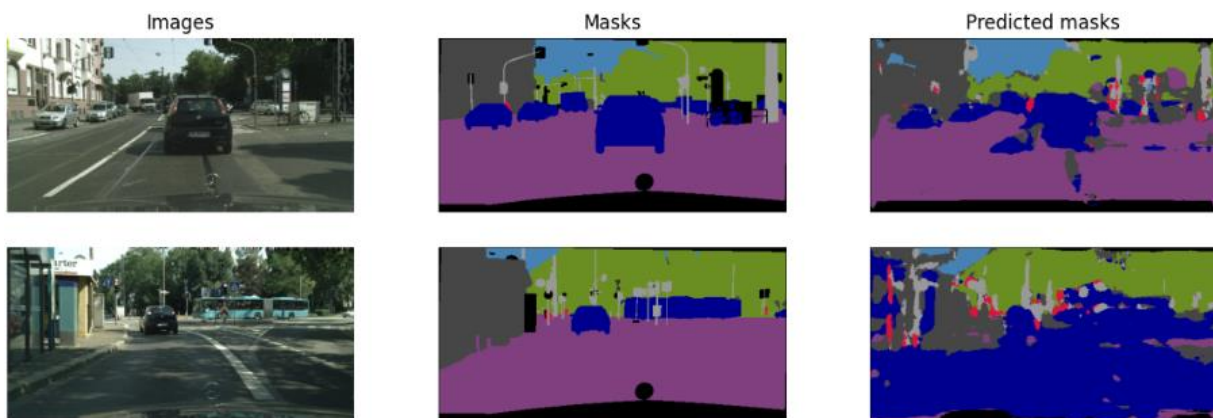


Figure 17 : modèle baseline - exemples prédictions

2. L'architecture U-Net, choix du *backbone*

Ensuite, nous mettons en place l'**architecture U-net**, que nous testons sur **deux backbones** : Resnet34 et VGG16.

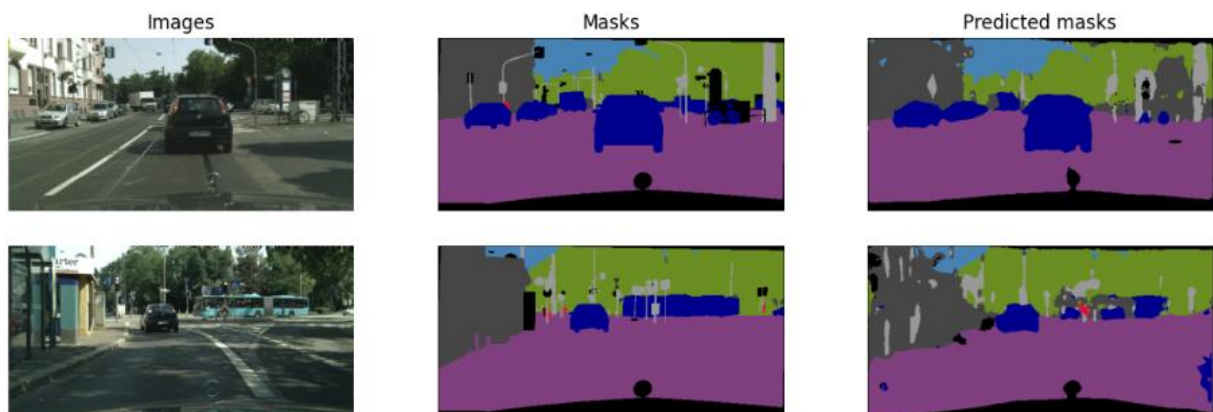


Figure 18 : modèle Resnet34 - exemples prédictions

3. Impact de l'augmentation de données

Les modèles précédents sont entraînés sur le dataset brut. Nous intégrons ensuite de l'augmentation de données dans le `generator` **des données d'entraînement** `train_gen`, grâce à `list_of_transforms` définie précédemment.

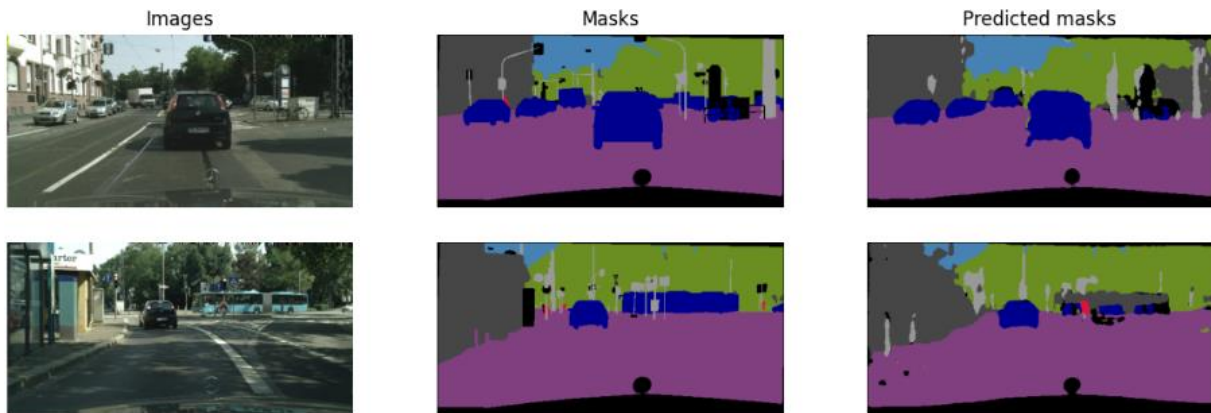


Figure 19 : modèle Resnet34 & augmentation données - exemples prédictions

4. Optimisation de la fonction de perte avec `class weight`

Nous utilisons ensuite la possibilité d'ajuster les `class_weights` de la composante *Dice Loss* de notre fonction de perte totale.

Pour cela nous faisons le choix de nous baser sur l'IoU par classe. On attribue alors un poids inversement proportionnel à la performance du modèle sur ces macro-catégories :

	Category	IoU	class_weights
0	void	0.768	0.110
1	flat	0.976	0.086
2	construction	0.921	0.092
3	object	0.359	0.236
4	nature	0.854	0.099
5	sky	0.925	0.091
6	human	0.506	0.167
7	vehicle	0.715	0.118

Figure 20 : détermination hyperparamètre `class_weights`

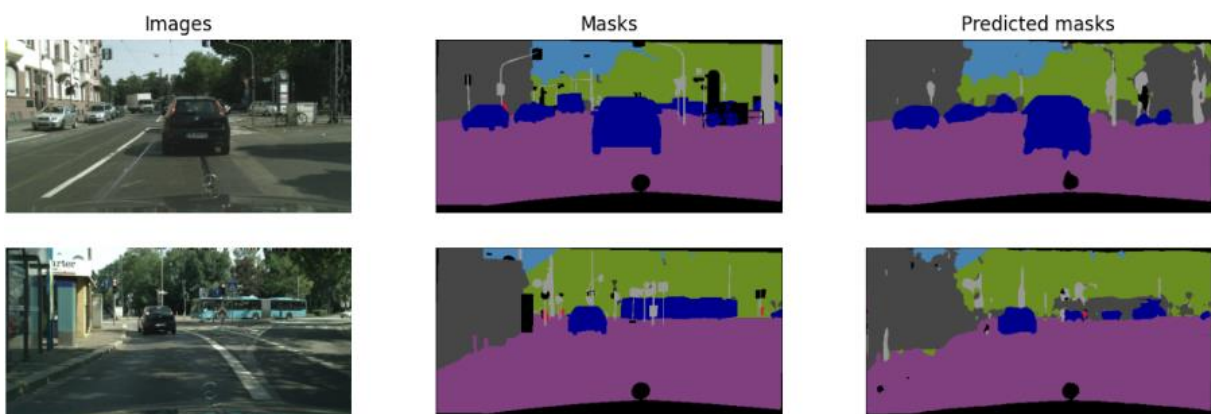


Figure 21 : avec `class_weights` - exemples prédictions

5. Synthèse des résultats

Voici les résultats obtenus :

model	backbone	loss_function	val_iou	val_f_score	training_time	augmentation
baseline		dice+focal	0.388	0.490	259.22 min	No
unet	resnet34	dice+focal	0.657	0.702	79.74 min	No
unet	vgg16	dice+focal	0.302	0.347	179.79 min	No
unet	resnet34	dice+focal	0.658	0.722	157.30 min	Yes
unet	resnet34	dice_CW+focal	0.664	0.732	147.38 min	Yes

Figure 22 : résultats modélisation

Analyse des résultats :

- Le **choix du backbone** se porte sur le **Resnet34** avec une *mean IoU* de 0.657 contre 0.302 pour le **VGG16**. La performance de ce dernier est étonnante et est même plus basse que la **baseline**. Cela pourrait s'expliquer par nos choix de simplification (vraiment trop peu de données d'entraînement pour ce type *backbone* ?)
- L'**augmentation de données** a un impact mesuré mais **bénéfique** sur le résultat avec une *mean IoU* de 0.658.
- La recherche d'optimisation de notre fonction de perte (**Focal Loss + Dice Loss**) grâce à l'hyperparamètre **class_weights** de sa composante **Dice Loss** a été positive elle aussi, avec une *mean IoU* de 0.664.

Le modèle gagnant possède donc les caractéristiques suivantes :

- **Architecture** : U-net
- **Backbone** : resnet34
- **Fonction de perte** : Focal Loss + Dice Loss_{class_weights}
- **Mean IoU sur le set de validation** : 0.664 (contre 0.388 pour le modèle de référence)
- **Dice score sur le set de validation** : 0.732 (contre 0.490 pour le modèle de référence)
- **Temps d'entraînement** : 147 min (contre 259 min pour le modèle de référence)

Pour la mise en production, nous entraînons ensuite ce modèle **sur la totalité du set d'entraînement** (passage de 512 à **2975 images**), avec des performances logiquement plus intéressantes :

- **Mean IoU sur le set de test** : 0.749
- **Dice score sur le set de test** : 0.828

V. Création et mise à disposition de l'API

1. API

L'objectif est de donner accès au pouvoir prédictif du modèle pour faire des inférences en ligne. L'API prend en entrée un fichier *.png* et retourne le mask prédit.


Nous passons tout d'abord le modèle au format , puis nous contruisons notre API grâce à .

Celle-ci :


- reçoit l'image
- la decode, la redimensionne et la prépare
- construit un outil d'inférence à partir du fichier **.tflite**
- et l'utilise sur l'image
- applique un mappage des couleurs
- retourne la prediction

Pour le déploiement, nous :

- Rédigeons un **Dockerfile**  docker
- Construisons une image  docker
- Créons un *resource group* Azure 
- Créons un *appservice plan* 
- Lançons la webapp 

- Paramétrons le déploiement continu avec  GitHub Actions

2. UI

L'objectif est de déployer une interface  simple pour utiliser l'API sur quelques images tests.

Dans un premier temps nous enregistrons 10 images et leurs masks dans les dossiers `/static/test_images` et `/static/test_masks` de l'application.

L'application :

- construit un bouton par image déclenchant l'appel à l'API
- enregistre le mask prédit au format `.png` dans le dossier `/static/predicted_mask`
- affiche l'image, le mask réel et le mask prédit

Pour le déploiement, la même méthode que pour l'API est utilisée, au sein des mêmes *resource group* et *appservice plan*.



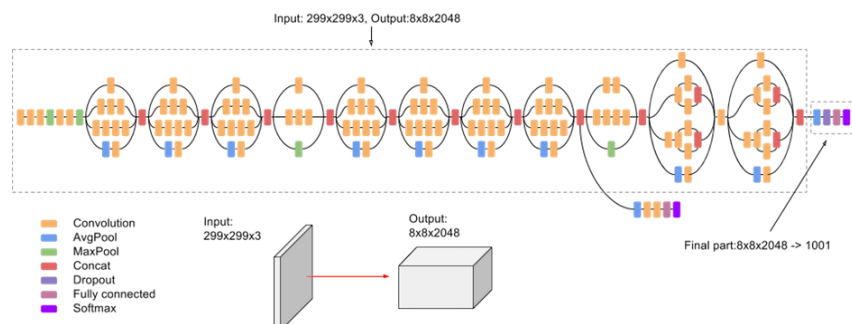
Figure 23 : UI de test de l'API, déployées

VI. Conclusion

Nous obtenons bien un premier modèle de segmentation d'images accessible via une API simple d'utilisation.

Pour optimiser nos résultats, plusieurs pistes sont possibles :

- Tester d'autres *backbones* : Il existe une grande variété de modèles qui nous pourrions utiliser dans l'*encoder* :
 - o Variation de VGG : *VGG19*
 - o Variations de Resnet : *Resnet18*, *Resnet50*, ...
 - o *Inception*



L'innovation de ce réseau est l'introduction d'opérations de convolution/pooling en **parallèle**. Cela permet d'utiliser des tailles très variées de kernel sans avoir à rendre le réseau très profond (et réduire ainsi l'overfitting). *Inception* a été amélioré avec le temps et décliné en plusieurs versions

- *EfficientNet*
Historiquement la performance des réseaux convolutifs a été améliorée en modifiant les échelles de profondeur (le nombre de couches), d'épaisseur (le nombre de channels / features maps) ou de résolution (la taille de l'image). *EfficientNet* fait varier ces paramètres non pas de façon empirique, mais optimisée et **uniforme**, permettant d'utiliser un modèle adapté aux ressources disponibles
- Tester d'autres architectures (LinkNet, HyperSeg, ou FPN)
- Optimiser d'autres hyperparamètres (optimizer, learning_rate, early_stopping_patience, ...)
- Ne plus réduire la taille des images
- Tester une tout autre approche : les *vision transformers*.