



**Future Vision Transport**

# **Projet Future Vision Transport**

-

## **Note technique**

### **SOMMAIRE**

#### **A - INTRODUCTION**

- 1 - Présentation du projet
- 2 - Objectif du projet
- 3 - Contraintes à respecter

#### **B - PRÉSENTATION DE L'ÉTAT DE L'ART DE LA SEGMENTATION D'IMAGES**

- 1 - Vision par ordinateur
- 2 - Classification d'images
- 3 - Segmentation sémantique

#### **C - PRÉPARATION DE LA MODÉLISATION**

- 1 - Présentation des données
- 2 - Préparation des données
- 3 - Identification de la cible
- 4 - Mise en place de l'environnement Azure

#### **D - MODÉLISATION**

- 1 - Récupération des éléments Azure
- 2 - Choix de la métrique d'évaluation
- 3 - Fonction de perte
- 4 - Manipulation d'un jeu de données volumineux
- 5 - Augmentation des images
- 6 - Implémentation des modèles et optimisation des hyper paramètres
- 7 - Synthèse comparative des différents modèles
- 8 - Présentation détaillée du meilleur modèle

#### **E - API REST**

- 1 - Implémentation de l'API
- 2 - Déploiement de l'API

#### **F - CONCLUSION ET PISTES D'AMÉLIORATIONS**

## A - INTRODUCTION

### 1 - Présentation du projet

Ce document technique a pour but de présenter le travail effectué et les résultats obtenus dans le cadre du projet **Future Vision Transport**. Future Vision Transport est une entreprise qui conçoit des systèmes embarqués de Vision par ordinateur pour les véhicules autonomes.

Le système se compose de 4 parties :

- Partie 1 : acquisition des images en temps réel
  - Partie 2 : traitement des images
  - Partie 3 : segmentation des images
  - Partie 4 : système de décision
- **Le but du projet est de travailler sur la partie 3 : la segmentation d'images.**

### 2 - Objectif du projet

L'objectif de ce projet est donc de concevoir **un modèle de segmentation d'images**. Ce modèle sera alimenté par le bloc de traitement des images et il alimentera lui-même le système de décision.

### 3 - Contraintes à respecter

Les contraintes à respecter pour ce projet sont les suivantes.

*Données en entrée :*

- Le jeu de données utilisé est **Cityscape** et il faut travailler avec les 8 catégories principales.
- Les images en entrée peuvent changer.
- Le volume des données est **important**.

*Résultat :*

- Le résultat de la segmentation doit être rendu dans une **API** simple à utiliser.
- L'API prend en entrée **l'identifiant d'une image** et renvoie **la segmentation de l'image réelle et la segmentation prédite par le modèle**.

*Infrastructure de travail :*

- Le projet devra utiliser l'infrastructure **Azure Machine Learning** pour le développement et le déploiement des modèles.
- Le framework de travail utilisé devra être **Keras**.
- L'API devra être mise en place en utilisant le framework **Flask** et déployée en utilisant les services Azure dédiés.

## B - PRÉSENTATION DE L'ÉTAT DE L'ART DE LA SEGMENTATION D'IMAGES

### 1 - Vision par ordinateur

La segmentation d'images fait partie du domaine de l'intelligence artificielles appelé **Vision par Ordinateur** (*Computer Vision en anglais*). Ce domaine comporte trois principales tâches :

- La **Classification** : consiste à classer des images selon des catégories préalablement établies (exemple : image de chien ou de chat). Chaque image ne peut appartenir qu'à une seule catégorie.
- La **Détection d'objets** : consiste à identifier pour tous les objets présents sur l'image leur position et leur catégorie. Une image peut comporter plusieurs catégories.
- La **Segmentation** : consiste à classer chaque pixel d'une image selon leur catégorie.

## Principales tâches de la vision par ordinateur :

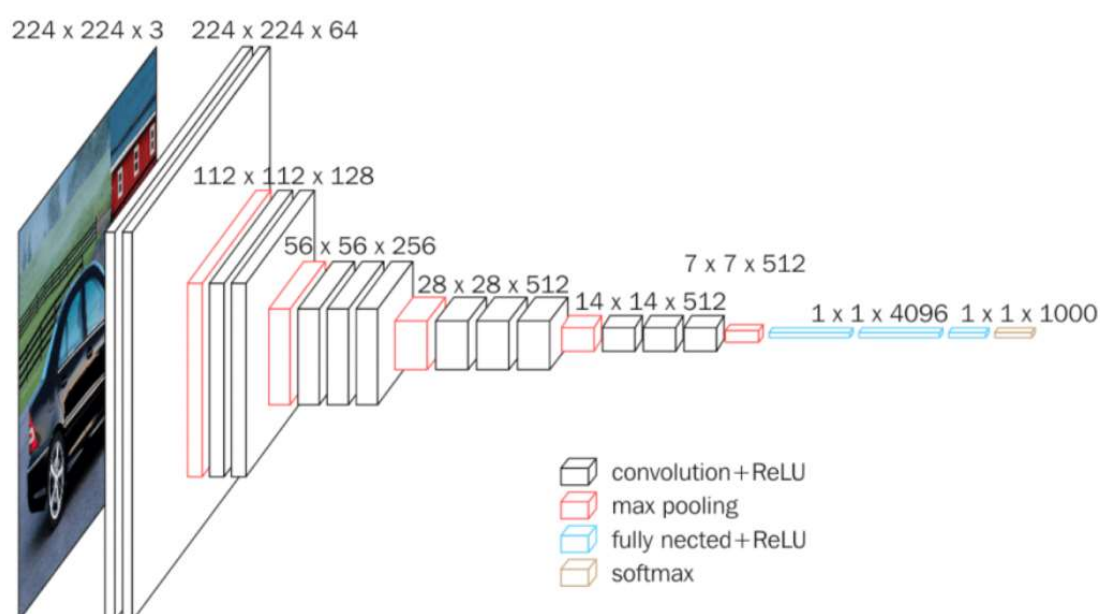


La segmentation d'images qui nous intéresse utilise des techniques de la classification. Nous allons donc décrire les algorithmes utilisés pour cette tâche avant de décrire plus en détails les algorithmes pour la segmentation.

### 2 - Classification d'images

La classification utilise des **réseaux de neurones à convolution classiques** : ces réseaux sont un enchaînement de couches de convolution et de pooling terminées par une ou plusieurs couches denses (*fully connected*). Les **couches de convolution** servent à extraire les caractéristiques des images (*features*) et produisent en sortie des cartes de caractéristiques (*features maps*). Ces cartes de caractéristiques contiennent les éléments (*pattern*) identifiés par le réseau dans l'image. Les **couches de Pooling** permettent de résumer l'information recueillie par les cartes de caractéristiques et servent à diminuer le nombre de paramètres du réseau. Le pooling le plus utilisé est le **Max Pooling**. Les dernières **couches denses** permettent de faire la classification et se terminent par une couche avec activation de type Sigmoid pour les classifications binaires et Softmax pour les classifications avec plusieurs catégories. Un des réseaux de ce type les plus connus est : **VGG Net**.

### Schéma de l'architecture du réseau à convolution VGG16 :



Plusieurs évolutions ont permis au cours du temps d'améliorer ces réseaux. Voici les principales :

- Réseaux **FCN** (*Fully Convolutional Network*) : on remplace les couches finales fully connected par une ou plusieurs couches de convolution.
- Réseaux **ResNet** (*développés par Microsoft*) : on ajoute des **blocs résiduels** (*skip connections*) qui créent une connexion entre la sortie d'une couche de convolution et leur entrée originale, ce qui permet de ne pas oublier l'information initiale.
- Réseaux **Inception** (*développés par Google*) : on ajoute des **blocs inception** qui sont des couches de convolution parallèles et concaténées (concept de réseau dans le réseau (*Network In Network*)). Ces couches améliorent les performances en augmentant la non linéarité et en permettant de faire une analyse à plusieurs échelles spatiales grâce aux branches en parallèle qui sont des couches de convolutions avec des filtres de tailles différentes.

Ces algorithmes de classification seront utilisés pour la segmentation en tant que '**Backbone**'. C'est à dire qu'ils seront utilisés pour extraire les caractéristiques des images.

### 3 - Segmentation sémantique

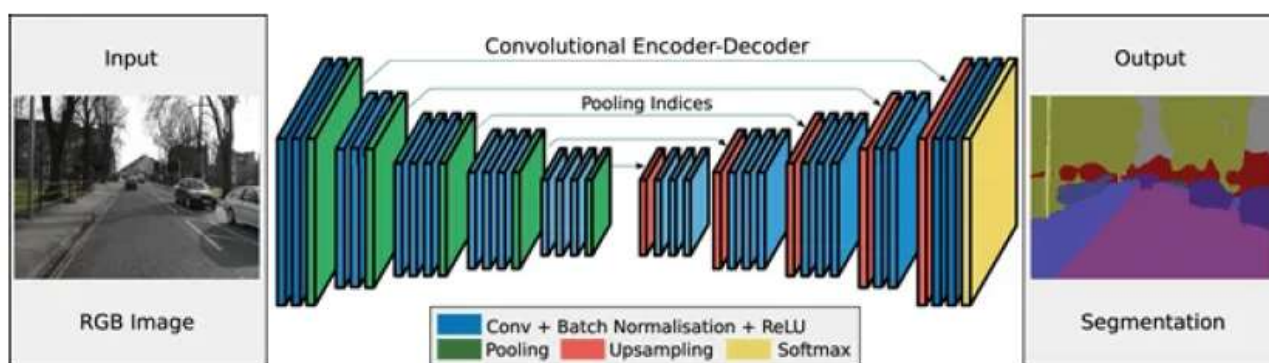
La segmentation sémantique est une tâche de classification qui consiste à attribuer une classe pour chaque pixel de l'image.

Pour réaliser cette tâche on utilise des algorithmes basés sur la technologie **Encoder-Decoder** (principe du sablier) : l'Encoder **diminue la dimension de l'image** (*downsampling*), puis le Decoder la **réaugmente** (*upsampling*). L'Encoder diminue la dimension de l'image en extrayant les caractéristiques de l'image grâce à un réseau convolutif utilisé pour la classification (Backbone). Le Decoder augmente la dimension afin de retrouver la dimension de l'image originale et créer une image de sortie (appelée **masque**) contenant la segmentation, c'est-à-dire la catégorie correspondant à chaque pixel.

Cet upsampling peut se faire de deux façons :

- Approche **sans paramètres** : plus proches voisins (*nearest neighbours*), max unpooling...
- Approche **avec paramètres** : on utilise des filtres appris par le réseau

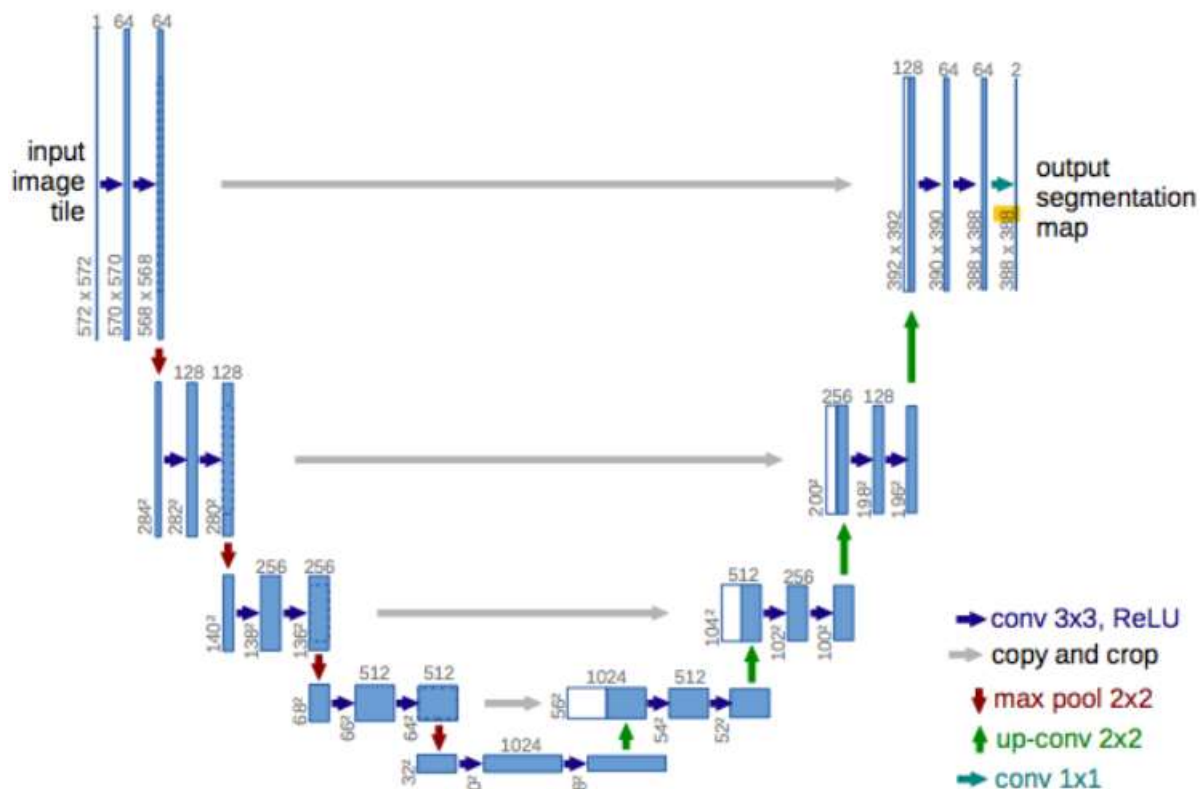
#### Schéma de l'architecture Encoder-Decoder :



Le problème avec ce type d'architecture simple est une perte d'information spatiale fine : l'approche sablier fait perdre l'information de haute résolution qui est nécessaire pour la segmentation. Pour corriger cela, on ajoute des connexions entre les couches de même résolution (*skip connections*). Pour réaliser ces connexions, on prend le tenseur de la phase de downsampling et on vient le concaténer lors de la phase d'upsampling. Cette technique est très efficace car elle ne rajoute pas de paramètres.

L'architecture emblématique de ce type de réseau est Unet qui doit son nom à sa forme typique en U.

### Schéma de l'architecture Unet :



Cette architecture permet d'extraire les caractéristiques et corrige la baisse de résolution engendrée grâce à des connections entre couches.

D'autres architectures peuvent être utilisées comme **Linknet**, **PSPNet** ou **FPN**. Ces architectures sont basées sur le même principe que Unet mais avec des modifications, notamment au niveau des connections entre couches.

## **C - PRÉPARATION DE LA MODÉLISATION**

### 1 - Présentation des données

Nous utilisons pour ce projet les données issues du dataset **Cityscape**. C'est un jeu de données comportant des images qui sont des photos prises d'une voiture dans différentes villes. Le dataset contient également les **annotations** (aussi appelées masques) de chaque image. Le **masque** est également une image dont les pixels correspondent à la **catégorie** dans l'image originale (piéton, nature, véhicule...).

### Exemples issus du jeu de données Cityscape :



## 2 - Préparation des données

### *a - Séparation du jeu de données*

Le dataset contient les jeux de données suivants : **Train** (2975 images et masques), **Validation** (500 images et masques) et **Test**. Les masques pour le jeu de Test ne sont pas disponibles, nous ne pourrions donc pas les utiliser pour évaluer le modèle final.

Pour remédier à ce problème nous avons procédé à la séparation du jeu de données de la façon suivante :

- Nous avons créé un jeu de données Test à partir du jeu de données Validation original.
  - Ce jeu de données **Test** nous permettra **d'évaluer le meilleur modèle final optimisé**.
- Nous avons séparé le jeu de données Train original en un jeu de données **Train** et un jeu de données **Validation** d'après un ratio TRAIN\_VAL\_SPLIT spécifié.
  - Le jeu de données **Train** nous permettra **d'entraîner les différents modèles**.
  - Le jeu de données **Validation** nous permettra **d'évaluer les différents modèles, de les comparer et de les optimiser**.
- Le jeu de données Test original n'est pas utilisé.
- Nous avons également créé des jeux de données 'Train échantillon' et 'Validation échantillon' afin d'entraîner les modèles intermédiaires sur un petit échantillon d'images afin de faire des tests.
  - ➔ Nous avons été particulièrement attentifs à ne pas toucher au jeu de données Test afin qu'il n'y ait **pas de fuite de données**.

### *b - Mise en place des 8 catégories principales*

Le jeu de données Cityscape contient 8 catégories et 32 sous-catégories. Comme cela nous est demandé nous allons travailler sur les **8 catégories** (et pas sur les 32 sous-catégories).

Les 8 catégories sont : **void, flat, construction, object, nature, sky, human** et **vehicle**. Pour cela, nous avons implémenté un dictionnaire et une fonction de mapping permettant de passer des 32 sous-catégories aux 8 catégories.

### *c - Renommage des images et masques*

Afin de respecter la contrainte d'utiliser des **identifiants** (ids) pour les images et masques, nous les avons renommés à l'aide d'identifiants (entiers numériques). Cela permet plus de clarté et une manipulation plus facile des images et masques.

### *d - Upload des données dans Azure*

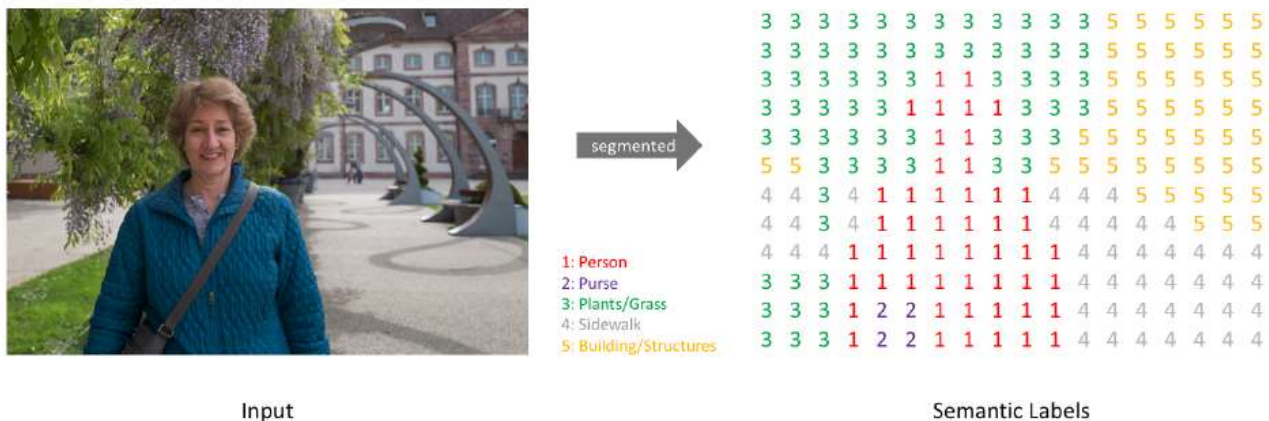
Les données préparées sont uploadées dans Azure afin d'être par la suite utilisée pour la modélisation. Pour cela, un **Datastore Azure** est créé et les données y sont déposées.

## 3 - Identification de la cible

L'objectif du projet est de faire de la **segmentation d'images**, c'est à dire de réaliser un découpage de chaque image originale en différentes catégories prédéfinies. Pour cela il faut prédire à quelle catégorie appartient chaque pixel de l'image originale. L'ensemble des prédictions de l'appartenance des pixels aux catégories va former une image que l'on appelle un **masque**.

La cible à prédire (également appelé le 'label') est donc le masque de chaque image qui correspond à la catégorie de chaque pixel de l'image. **Pour chaque image en entrée, il faut donc prédire un masque contenant pour chaque pixel la catégorie correspondante à ce pixel (nature, ciel, humain...).**

Exemple de segmentation d'une image – résultat sous forme d'une matrice contenant les catégories de chaque pixel:



#### 4 - Mise en place de l'environnement Azure

Le cycle de modélisation dans Azure Machine Learning est le suivant :

- Authentification auprès de Azure et récupération du Workspace de travail
- Préparation de l'environnement de travail
- Création de scripts comportant la modélisation
- Envoie des scripts à Azur pour l'entraînement des modèles
- Obtention des résultats

Pour l'authentification auprès de Azure nous avons utilisé le service d'authentification : '**Service Principal Authentication**'. C'est le service recommandé pour s'authentifier de façon **sécurisée**. D'autre part, les mots de passe sont stockés dans des variables d'environnement pour plus de sécurité.

Une fois authentifié, nous pouvons récupérer le **Workspace Azure Machine Learning** ce qui nous permet d'interagir avec les services Azure.

Nous avons ensuite mis en place l'environnement nécessaire à la modélisation dans Azure. Pour cela nous avons d'abord créé un **Dataset** à partir du Datastore préalablement mis en place, puis nous avons créé **l'environnement de travail Python** et enfin nous avons mis en place **l'instance de calcul**.

Pour simplifier le travail, tous ces éléments sont mis en place une seule fois et sont enregistrés dans Azure. Cela permet par la suite de les récupérer pour la modélisation et rend notre travail 'industrialisable'.

## D - MODÉLISATION

### 1 - Récupération des éléments Azure

En étape préambule à la modélisation on récupère tout d'abord les éléments Azure préalablement préparés : le Workspace, le Dataset contenant les données, l'environnement de travail Python et l'instance de calcul.

Tous ces éléments permettent d'envoyer des scripts à Azure. Ces scripts contiennent les modélisations et sont envoyés à Azure pour l'entraînement des modèles.

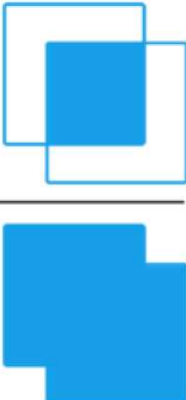


## 2 - Choix de la métrique d'évaluation

La métrique d'évaluation 'classique' utilisée dans les problèmes de classification est l'Accuracy. Cependant cette métrique n'est pas adaptée à des jeux de données déséquilibrés comme c'est le cas pour la segmentation d'images.

Nous avons retenu la métrique '**Mean IoU Score**' pour évaluer les modèles. Ce score correspond à la moyenne du coefficient '**Intersection over Union**' des différentes catégories. Le coefficient Intersection over Union est défini comme la zone de chevauchement (*overlap*) entre la segmentation prédite et la segmentation réelle divisé par la zone d'union entre la segmentation prédite et la segmentation réelle.

Représentation graphique et formule mathématique du coefficient 'Intersection over Union' :


$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$
$$\text{IoU} = \frac{TP}{TP + FP + FN} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|}$$

La métrique **Mean IoU Score** est particulièrement bien adaptée à la problématique métier de segmentation d'images. Le score est compris entre 0 et 1 et plus on s'approche de 1 plus la similarité entre la prédiction et la réalité est élevée et meilleure est la segmentation.

En plus de cette mesure, nous avons également pris en compte le **temps d'exécution** et le **nombre de paramètres** des différents modèles.

## 3 - Fonction de perte

La fonction de perte 'classique' pour les modèles de classification à plusieurs catégories est la '**Categorical Cross Entropy**'. Lors de la phase d'optimisation des hyper paramètres, nous chercherons à optimiser cette fonction. Pour cela nous essaierons une fonction plus adaptée à la segmentation d'images : la fonction de perte '**Dice Loss**'.

La fonction de perte Dice Loss est calculée à partir du **coefficient Dice** qui est égal à deux fois la zone de chevauchement entre la prédiction et la réalité divisé par le nombre total de pixels de l'image et du masque.

## 4 - Manipulation d'un jeu de données volumineux

Les images sont des données de **grande dimension** qui sont plus **volumineuses** et plus **difficiles à gérer** que des données tabulaires 'classiques'. Le volume important des images fait que le jeu de données ne peut pas être chargé dans son entièreté en mémoire.

Pour résoudre ce problème, nous avons mis en place un **générateur de données**.

Le générateur de données est implémenté sous forme de classe Python qui '**dérive**' de la classe Keras 'Sequence'. Le générateur de données fournira les images au modèle au fur et à mesure de l'apprentissage sous forme de **batches**. La taille du batch correspond au nombre d'images envoyés en même temps au modèle. Le générateur de données permet également le traitement des images sur plusieurs cœurs de calcul.



## 5 - Augmentation des données

Le nombre d'images annotées est relativement **faible** car l'annotation des images requiert un **travail important**. C'est un problème classique que l'on retrouve en segmentation d'images.

Pour résoudre ce problème, nous avons utilisé une technique appelée **augmentation des données**.

Pour réaliser cela, nous avons utilisé la librairie *Albumentation*. Cette librairie permet d'augmenter les images en appliquant des transformations aux images originales (rotation, ajout de bruit gaussien...). Cela permet donc d'obtenir de nouvelles images à partir des images originales et donc d'améliorer l'apprentissage. L'augmentation des données se fait à la volée dans le générateur de données.

Nous avons testé deux types d'augmentation d'images :

- Une première version 'légère' qui utilise **trois techniques d'augmentation** :
  - ✓ 1 : Horizontal Flip (basculement horizontal de l'image)
  - ✓ 2 : Random Gamma (modification aléatoire du gamma)
  - ✓ 3 : Blur (ajout de bruit)

➤ Chacune des trois techniques d'augmentation a une **probabilité de 0.25 d'être appliquée**
- Une seconde version plus 'conséquente' qui utilise **six techniques d'augmentation** :
  - ✓ 1 : Elastic Transform (transformation élastique)
  - ✓ 2 : Horizontal Flip (basculement horizontal)
  - ✓ 3 : Grid Distortion (distorsion de l'image)
  - ✓ 4 : Random Gamma (modification aléatoire du gamma)
  - ✓ 5 : Emboss (ciselage de l'image)
  - ✓ 6 : Blur (ajout de bruit)

➤ Chacune des six techniques d'augmentation a une **probabilité de 0.5 d'être appliquée**

La seconde version transforme plus les images que la première : le nombre de modifications et la probabilité d'application sont plus élevés.

Exemple d'augmentation de données – version 1 (image originale à gauche, image transformée à droite) :



Exemple d'augmentation de données – version 2 (image originale à gauche, image transformée à droite) :



## 6 - Implémentation des modèles et optimisation des hyper paramètres

Les modélisations que nous avons implémentées peuvent être regroupées en 2 catégories :

- **Modélisations 'from scratch'** : les modèles sont implémentés 'à la main' en utilisant Keras.
  - Cette première approche permet de bien comprendre et appréhender les modèles et l'API Keras.
- **Modélisations utilisant la librairie 'segmentation-models'** : cette librairie Python, qui s'appuie sur Keras, contient des modèles, des métriques et des fonctions de coût dédiés à la segmentation.
  - Cette seconde approche permet d'utiliser des modèles plus avancés, ainsi que des fonctions de coût différentes déjà implémentées.

L'utilisation de cette librairie nous permettra notamment **d'optimiser les hyper paramètres** :

- La **fonction de coût** : essai de la fonction **Dice Loss** en plus de la fonction **Categorical Cross Entropy**
- Le **Backbone** : essais des backbones **VGGNet**, **ResNet** et **EfficientNet** avec et sans poids pré entraînés sur les données **Imagenet** pour l'initialisation
- L'**architecture** : essais des architectures **Unet**, **FPN (Features Pyramid Network)** et **Linknet**

Les différentes modélisations implémentées forment une suite logique dont le but est d'améliorer le modèle au fur et à mesure afin d'obtenir le meilleur modèle qui sera déployé.

Pour l'entraînement des modèles nous avons réduit la taille des images afin d'avoir des temps de calcul raisonnables et également pour tenir compte de la capacité mémoire du matériel qui nous est alloué.

## 7 - Synthèse comparative des différents modèles

Tableau comparatif des résultats obtenus sur les différents modèles :

| Modèle de référence   |   |                     |         |                         |                          |              |
|---|---|---------------------|---------|-------------------------|--------------------------|--------------|
| Meilleur modèle   |   |                     |         |                         |                          |              |
| Meilleur modèle entraîné sur des images de plus grande taille |   |                     |         |                         |                          |              |
| N° Experiment   |   | Implementation      | Model   | Backbone                | Loss Fonction            | Augmentation |
| 1   | 1_cityscape_fs_simple                                       | from scratch        | Simple  | -                       | categorical_crossentropy | no           |
| 2   | 2_cityscape_fs_simple_aug                                   | from scratch        | Simple  | -                       | categorical_crossentropy | yes - V1     |
| 3   | 3_cityscape_fs_simple_augv2                                 | from scratch        | Simple  | -                       | categorical_crossentropy | yes - V2     |
| 4   | 4_cityscape_fs_unet_aug                                     | from scratch        | Unet    | -                       | categorical_crossentropy | yes - V1     |
| 5   | 5_cityscape_sm_unet_aug_vgg                                 | segmentation-models | Unet    | VGGnet                  | categorical_crossentropy | yes - V1     |
| 6   | 6_cityscape_sm_unet_aug_vgg_imagenet                        | segmentation-models | Unet    | VGGnet - Imagenet       | categorical_crossentropy | yes - V1     |
| 7   | 7_cityscape_sm_unet_aug_vgg_imagenet_dice_loss              | segmentation-models | Unet    | VGGnet - Imagenet       | dice_loss                | yes - V1     |
| 8   | 8_cityscape_sm_unet_aug_resnet_imagenet_dice_loss           | segmentation-models | Unet    | Resnet - Imagenet       | dice_loss                | yes - V1     |
| 9   | 9_cityscape_sm_unet_aug_efficientnet_imagenet_dice_loss     | segmentation-models | Unet    | Efficientnet - Imagenet | dice_loss                | yes - V1     |
| 10  | 10_cityscape_sm_fpn_aug_efficientnet_imagenet_dice_loss     | segmentation-models | FPN     | Efficientnet - Imagenet | dice_loss                | yes - V1     |
| 11  | 11_cityscape_sm_linknet_aug_efficientnet_imagenet_dice_loss | segmentation-models | Linknet | Efficientnet - Imagenet | dice_loss                | yes - V1     |
| 12  | 12_cityscape_best_model                                     | segmentation-models | FPN     | Efficientnet - Imagenet | dice_loss                | yes - V1     |

| RESULTS |            |           |       |          |          |              |
|---------|------------|-----------|-------|----------|----------|--------------|
| N°      | Nb params  | Exec Time | Loss  | Val_Loss | Mean IoU | Val_Mean IoU |
| 1       | 2 883 528  | 2h48mns   | 0.449 | 0.423    | 0.491    | 0.480        |
| 2       | 2 883 528  | 2h42mns   | 0.435 | 0.410    | 0.499    | 0.503        |
| 3       | 2 883 528  | 3h16mns   | 0.767 | 0.551    | 0.325    | 0.397        |
| 4       | 31 059 592 | 3h11mns   | 0.323 | 0.332    | 0.590    | 0.593        |
| 5       | 23 753 288 | 3h02mns   | 0.396 | 0.363    | 0.535    | 0.524        |
| 6       | 23 753 288 | 3h02mns   | 0.331 | 0.33     | 0.583    | 0.574        |
| 7       | 23 753 288 | 2h58mns   | 0.185 | 0.211    | 0.709    | 0.683        |
| 8       | 24 457 169 | 2h44mns   | 0.170 | 0.200    | 0.730    | 0.699        |
| 9       | 17 868 848 | 3h05mns   | 0.152 | 0.174    | 0.754    | 0.729        |
| 10      | 13 919 792 | 2h15mns   | 0.137 | 0.169    | 0.774    | 0.735        |
| 11      | 13 762 000 | 2h01mns   | 0.154 | 0.178    | 0.752    | 0.725        |
| 12      | 13 919 792 | 4h32mns   | 0.114 | 0.134    | 0.808    | 0.781        |

### Analyse des résultats :

Le premier modèle est un modèle simple qui est une suite de couches de convolution enchaînées avec une suite de couches de convolution transposées.

Ce modèle nous servira de **référence (baseline)**. Ce modèle donne un score IoU sur le jeu de validation de 0.48.

**Augmentation des données** : la première version de l'augmentation des données améliore les résultats alors que la deuxième version dégrade les résultats. On peut en conclure qu'une transformation modérée des images est bénéfique pour l'apprentissage mais qu'une transformation trop conséquente est défavorable pour l'apprentissage. Cela peut s'expliquer par le fait que la première version effectue une transformation de l'image mais tout en gardant sa structure, alors que la seconde version 'destructure' l'image comme nous l'avons vu dans l'exemple précédent.

**Fonction de coût** : la fonction de coût **Dice Loss** améliore les résultats par rapport à la fonction Categorical Cross Entropy. Cela confirme que la fonction de coût Dice Loss est bien adaptée au problème de Segmentation.

**Backbone** : le backbone qui donne les meilleurs résultats est un réseau **EfficientNet** avec les poids pré entraînés sur le jeu de données **Imagenet**. L'utilisation de poids pré entraînés améliore les résultats, ainsi que le réseau EfficientNet qui augmente le score IoU tout en diminuant le nombre de paramètres et le temps d'exécution.

**Architecture** : l'architecture qui donne les meilleurs résultats est **FPN**. Cette architecture améliore nettement les résultats : le score IoU augmente et le temps d'exécution diminue.

Le **meilleur modèle** est donc le suivant :

- Augmentation des données version 1
- Fonction de coût : Dice Loss
- Backbone : EfficientNet avec poids pré entraînés sur Imagenet
- Architecture : FPN

Grâce à ce modèle nous obtenons des **résultats de qualité qui améliorent nettement le modèle de référence** :

- ✓ Le score IoU de validation augmente nettement de 0.480 à **0.735**
- ✓ Le temps d'exécution baisse de 2h48mns et **2h15mns**
- ✓ Le nombre de paramètres reste **très raisonnable** (13 919 792)

Nous avons ensuite entraîné ce meilleur modèle sur **des images de plus grande taille**. Cela a encore **nettement amélioré le résultat** : nous obtenons un score IoU de validation égal à **0.781**, ce qui est **un très bon score**.

Evaluation du meilleur modèle sur le jeu de test :

Pour finir, nous avons évalué le meilleur modèle sur le jeu de données de test :

- ✓ On obtient **un score IoU sur le jeu de test égal à 0.778**
- ✓ Ce score est très proche de celui obtenu lors de la modélisation sur le jeu de validation
- ✓ Cela montre **la qualité et la bonne capacité de généralisation de notre modèle**

## 8 - Présentation détaillée du meilleur modèle

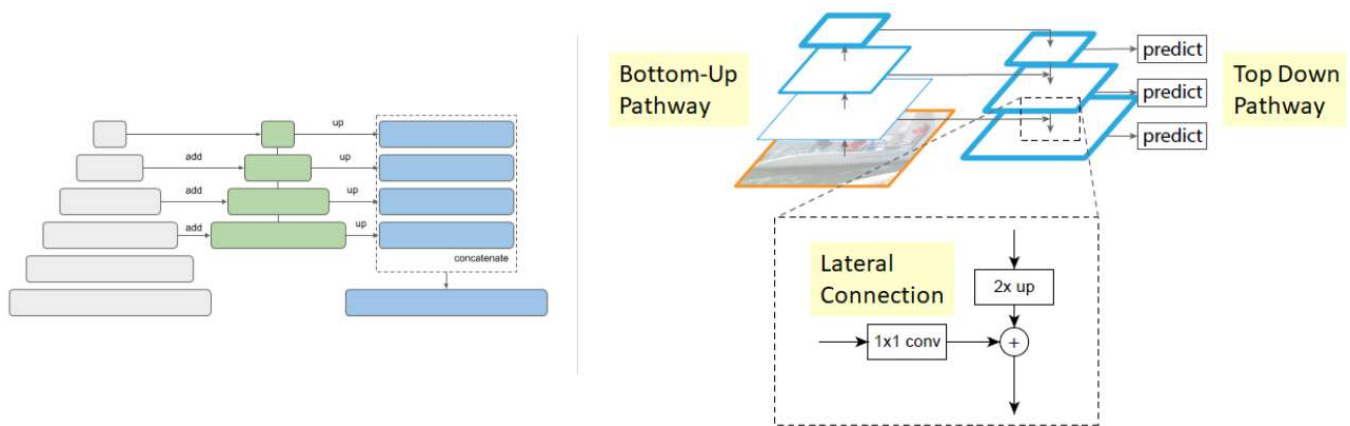
Backbone EfficientNet :

Les concepteurs de ce réseau ont constaté qu'habituellement on a tendance à augmenter les réseaux de neurones, en termes de largeur, profondeur et résolution, au fur et à mesure que la puissance de calcul augmente. Mais cette augmentation se fait de façon assez empirique voire aléatoire. L'idée des concepteurs est de **rationaliser le scaling** (mise à l'échelle). Pour cela, dans un réseau EfficientNet, le scaling se fait de façon intelligente grâce à une formule mathématique qui permet d'ajuster la largeur, la profondeur et la résolution de façon homogène et proportionnelle. Cela permet d'obtenir des réseaux optimisés en termes de dimensions et par conséquent de nombre de paramètres.

### Architecture FPN (Features Pyramid Network) :

Les réseaux FPN se composent de deux phases : une première phase appelée 'bottom-up' et une seconde phase appelée 'top-down'. Lors de la phase bottom-up, on construit une pyramide de cartes de caractéristiques. Lors de la phase top-down, on utilise les cartes de caractéristiques en pyramide de la précédente phase pour enrichir celles de la phase top-down lors de l'upsampling. Pour cela, les cartes de caractéristiques des 2 phases du même niveau, c'est à dire ayant la même résolution, sont regroupées en faisant une addition termes à termes. On a donc des connections latérales à chaque étage de la pyramide des cartes de caractéristiques de différentes résolutions, ce qui permet d'enrichir l'information lors de la partie upsampling.

### Schémas de l'architecture FPN :



Le meilleur modèle est enregistré au **format H5** dans Azure afin d'être déployé via une **API REST**.

## **E - API REST**

### 1 - Implémentation de l'API

Nous avons déployé le meilleur modèle via une API en utilisant le **framework Flask**.

Nous avons pour cela créé 3 routes :

- une *première route* qui retourne **la page d'accueil** : cette page contient un message de bienvenue ainsi que les liens vers les deux autres routes.
- une *deuxième route* qui retourne **la liste des identifiants des images disponibles pour la segmentation** : cela permet à l'utilisateur de savoir rapidement quelles sont les images disponibles et prêtes pour une segmentation.
- une *troisième route* qui retourne **le résultat de la segmentation** : cette route prend en entrée l'identifiant d'une image fournie par l'utilisateur dans sa requête et retourne l'image avec les segments identifiés par le modèle et l'image avec les segments identifiés annotés dans le jeu de données.

### 2 - Déploiement de l'API

Ensuite nous avons déployé cette API. Pour cela nous avons utilisé **les services d'Azure 'Web Applications Services' couplés à GitHub**. Cela permet de faire du déploiement continu : si on fait une modification sur le dépôt GitHub de l'API, alors le modèle est **automatiquement redéployé** par Azure pour prendre en compte les modifications effectuées.

Résultat obtenu - exemples de résultats renvoyés par l'API pour 2 segmentations d'images :

Segmentation de l'image ayant l'ID 0 :

### Cityscape segmentation - ID Image : 0

Image + Mask to predict



Image + Mask predicted



Segmentation de l'image ayant l'ID 10 :

### Cityscape segmentation - ID Image : 10

Image + Mask to predict



Image + Mask predicted



## F - CONCLUSION ET PISTES D'AMELIORATIONS

L'objectif fixé a été atteint : nous obtenons un modèle de qualité qui donne de bons résultats et avec une bonne capacité de généralisation, tout en ayant respecté les contraintes imposées.

Voici plusieurs  **pistes d'améliorations possibles**  : on pourrait essayer :

- d'optimiser d'autres hyper paramètres comme l'optimiser (SGD, RMSprop, Adagrad...)
- d'autres versions d'augmentation des images
- d'entraîner les modèles sur des images de plus grande taille (nécessiterait plus de ressources de calcul)
- d'autres fonctions de perte, par exemple une combinaison linéaire entre la Categorical Cross Entropy et la Dice Loss
- une architecture de type Mask R-CNN : architecture basée sur les modèles de détection tel que Faster R-CNN
- une approche de type Transformer. Ce type d'approche est actuellement l'état de l'art pour le Traitement du Langage Naturel (NLP Natural Language Processing). On pourrait pour cela utiliser la librairie Hugging Face qui est une référence dans ce domaine.