

Segmentation Semantic (*CityScapes dataset*)

Note technique – *TUCCIO Sébastien*



Table des matières

I.	Etat de l'art.....	4
II.	Recherche de la meilleure approche de récupération des Mask.....	5
III.	Synthèse des différentes modélisation et résultats.....	7
1.	Générateur de données et choix de la métrique	7
2.	Création d'une Baseline	7
3.	Résultat des différentes « loss » fonction	9
IV.	Optimisation/Choix structure finale.....	10
1.	Augmentation des images.....	10
2.	Résultat des différentes structures	11
V.	Déploiement du modèle final (Flask API)	12
1.	Sauvegarde/entrainement du modèle	12
1.1.	Connexion Azure	12
1.2.	Création du Datastore	13
1.3.	Création du script d'entrainement.....	14
2.	Création d'une WebAPP Azure/ déploiement du model (ACI).....	15
2.1.	ACI déploiement	15
2.2.	Flask déploiement	15
3.	Consommation de l'API	16
3.1.	ACI.....	16
3.2.	Flask.....	16
VI.	Amélioration possible (Pour aller plus loin)	17
1.	Optimisation des paramètres.....	17
2.	Mise à jour du Datastore	17
3.	Déploiement automatisé.....	17
VII.	Source.....	18



Tables des figures

Figure 1: segmentation sémantique (Label Encoder 8 classes)	6
Figure 2: segmentation sémantique (One Hot Encoder 8 classes)	6
Figure 3: Résultat du benchmark (récupération des masques)	6
Figure 4: Learning Curve Baseline	8
Figure 5 : Learning curve (overfitting)	9
Figure 6 : Augmentation des images	10
Figure 7 : Synthèse des différentes modélisations.....	11
Figure 8 : Azure active directory (application)	12
Figure 9 : Azure active directory (certificat & secrets).....	12
Figure 10 : Groupe de ressources (Contrôle d'accès).....	13
Figure 11 : Code ajout de données Datastore.....	13
Figure 12 : Code Entraînement d'un modèle	14
Figure 13 : Code déploiement d'un modèle	15
Figure 14 : Flask choix identifiant images	16
Figure 15 : Flask prédiction	16



I. Etat de l'art

Le projet a pour but la création d'un système embarqué pour la conception d'une voiture autonome qui permettra la **prise de décision de la voiture autonome** en fonction de son environnement (récupérer par une caméra à l'intérieur de la voiture).

La conception du système embarqué à été découpé en 4 étapes clés :

1. Acquisition des images en temps réel
2. Traitement des images
3. **Segmentation des images**
4. Système de décision

J'ai ici la charge de la partie 3 du système embarqué : la partie **segmentation sémantique des images**.

La segmentation d'images nécessaire pour le système de décision de l'étape 4 doit comporter 8 classes :

1. Vide
2. Route
3. Construction
4. Objets
5. Nature
6. Ciel
7. Humain
8. Véhicule

Et la segmentation d'image doit-être accessible depuis une **API** (que l'on hébergera sur Azure)



II. Recherche de la meilleure approche de récupération des Mask

Durant les phases 1 et 2 du système embarqué nous récupérerons des données sur la même base que le jeu de données que fournit [cityscapes-dataset](#).

Le jeu de données nous fournit 2 dossiers avec les données d'entrée (les images provenant de la DashCam) et les données de sortie (le Masque de la segmentation sémantique).

Pour les données d'entrées nous avons une image de dimension 2048 par 1024 en sur 3 canaux (RGB).

Pour les données de sortie (le masque) il existe 4 solution disponible :

- Un fichier JSON qui contient toutes les informations sur les masques d'une image nommé : « `_gtFine_polygons.json` ».
- Une image en couleur png nommé : « `_gtFine_color.png` » avec les différentes segmentations représentées par une couleur sur la base des 3 canaux de couleur RGB (ex : (255,120,255))
- Une image avec uniquement les IDs des différents labels de segmentation nommé : « `_gtFine_labelIds.png` », de -1 à 32 répété sur 3 canaux (ex : « (2,2,2) » pour l'ID « (2) »)
- Une image avec uniquement des IDs sélectionnés sur la même base que le fichier « `labelIds` » sans tous les labels présents nommé : « `_gtFine_instancelds.png` ».

Finalement on gardera comme solution uniquement le fichier JSON et le fichier « `labelIds` » (le fichier `labelIds` permet d'obtenir le même résultat que le fichier couleur mais en ne récupérant que 1 seul canal de couleur).

Le fichier « `instancelds.png` » ne contient pas toutes les informations nécessaires et le fichier « `color.png` » est similaire au fichier « `labelIds.png` ».



J'ai donc effectué un benchmark pour voir quelle méthode permet d'obtenir les masques le plus rapidement possible formater correctement (passer des 33 classes à 8 classes) avec Label Encoder (c'est-à-dire 1 canaux contenant les 8 classes comme le labelIds comme représenter sur la Figure 1) et avec One Hot Encoder (c'est-à-dire 8 canaux binaire comme représenter sur la Figure 2).

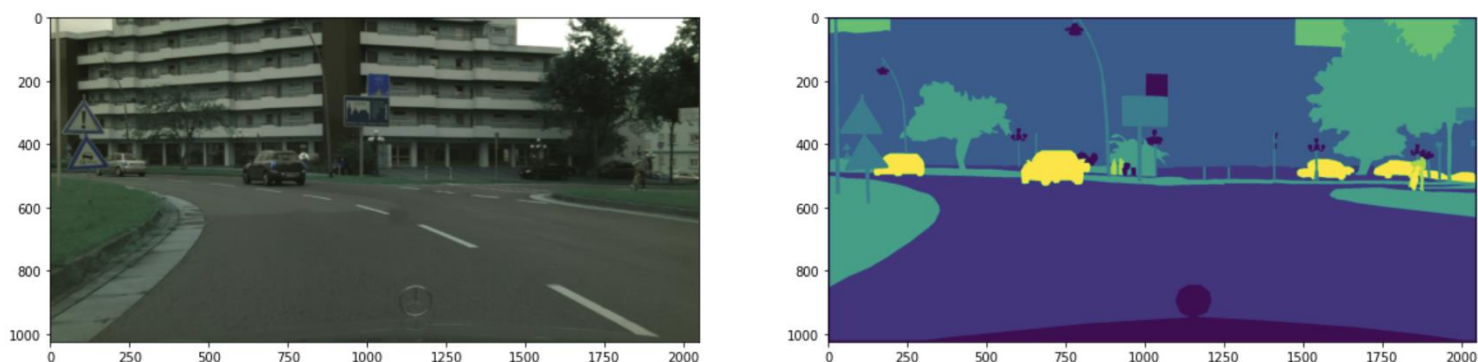


Figure 1: segmentation sémantique (Label Encoder 8 classes)

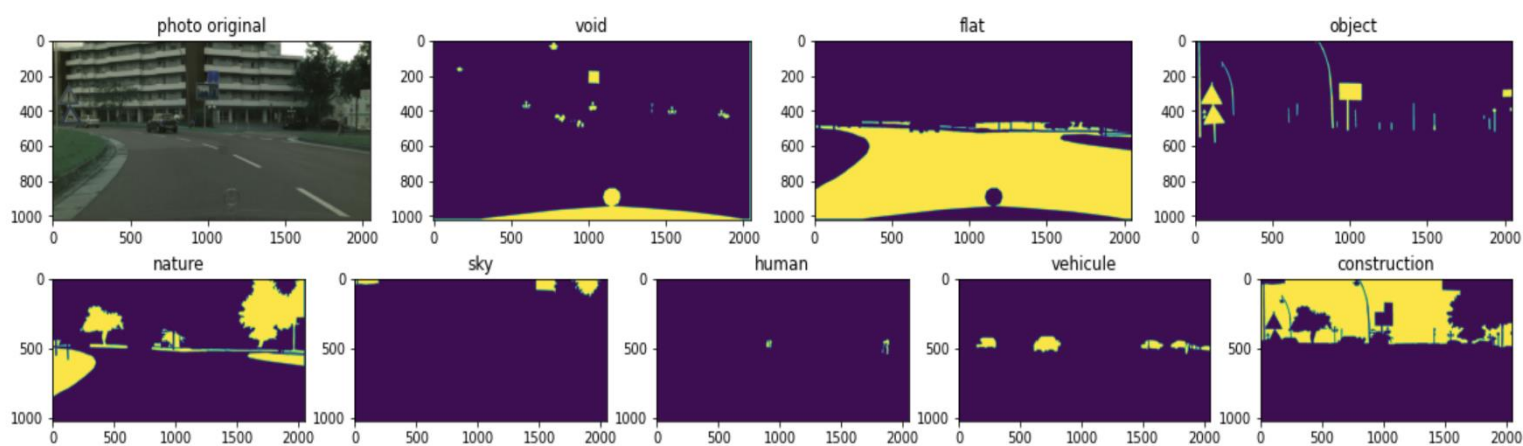


Figure 2: segmentation sémantique (One Hot Encoder 8 classes)

Les résultats (Figure 3, ci-dessous) sont indiscutables, la récupération des masques par le fichier labels_ids et bien plus performantes avec 0.6 secondes de différence pour la méthode Label Encoder et 4.3 secondes pour la méthode One Hot Encoder !

	label_ids	json_file	gain_seconde_labelid_vs_json
label_encoder	0.1946	0.8548	0.6602
one_hot_encoder	0.1876	4.5758	4.3882

Figure 3: Résultat du benchmark (récupération des masques)



III. Synthèse des différentes modélisation et résultats

Maintenant que les fonctions pour récupérer correctement les données d'entrée et les masques de sorties sont prêtes, viens la phase de modélisation.

1. Générateur de données et choix de la métrique

La première chose à faire est donc de créer un générateur de données afin d'entraîner nos modèles.

Le générateur de données permet d'utiliser l'intégralité du jeu de données sans passer par le stockage rapide de la RAM (stockage de l'intégralité des images dans la RAM) en chargeant les images une à une en les manipulant "à la volé". Les générateurs son essentiel pour ce type d'utilisation, avec des milliers d'images, il serait difficile de stocker l'intégralité des images en mémoire RAM, d'autant plus avec une augmentation des images qui multiplie grandement le nombre d'images.

J'ai donc réalisé 2 générateurs de données, un générateur de données qui sert uniquement à récupérer « à la volé » les images et masques avec pour entrer les chemins d'accès uniquement, et un autre générateur de données qui lui récupérer les données ET augmente les données X fois en appliquant différentes transformation (plus de détail dans la partie IV. Optimisation => 1. Augmentation des images).

Pour la métrique on utilisera le score Jaccard ou MeanIoU (Moyenne Intersection over Union) qui répond bien à notre problématique de segmentation sémantique multi-classes.

Cette métrique permet d'obtenir un score sur la base du nombre de pixel similaire des classes recherché, on obtient donc un score plus juste que l'Accuracy qui compte les pixels correctement classifier uniquement.

2. Création d'une Baseline

Dans un second temps on créer un modèle simple qui nous servira de Baseline afin d'avoir un modèle de comparaison pour voir l'amélioration de nos autres modélisations.

Pour la Baseline je me suis basé sur 2 approches :

- Modélisation ML classique linéaire (régression logistique)
- Modélisation UNet sans augmentation d'images

J'ai choisi ces 2 approches car je compare un modèle Machine Learning qui n'est pas adapté à notre problématique (pour voir jusqu'où peut aller cette approche) et un modèle plus complexe mais « basique » avec une modélisation UNet simple sans augmentation et une loss function classique (« categorical_crossentropy ») afin d'avoir un bon ordre d'idée sur les améliorations possible.



La modélisation linéaire obtient un résultat assez correct avec un score de **0,2941**. Cette modélisation se base sur les pixels et leur transformation par des filtres (gabor, sobel, roberts, prewitt, gaussian filtre...), et si on regarde la matrice de confusion, on obtient de très bon résultat sur la prédiction des pixels de la route (« flat ») et de bon résultat sur le ciel, les construction et la nature (« sky, construction, nature »).

Pour la modélisation UNet sans augmentation on obtient un score MeanIoU de seulement **0.2584** après 50 epoch avec une Learning curve très instable (Figure 4, ci-dessous).

Pour améliorer ce comportement on peu essayer de changer la fonction de coût du réseau, pour avoir de meilleur résultat. Dans tous les cas il semblerais préférable d'effectuer bien plus d'épochs pour atteindre une stabilisation de la learning curve et avoir des résultat plus prometteur.

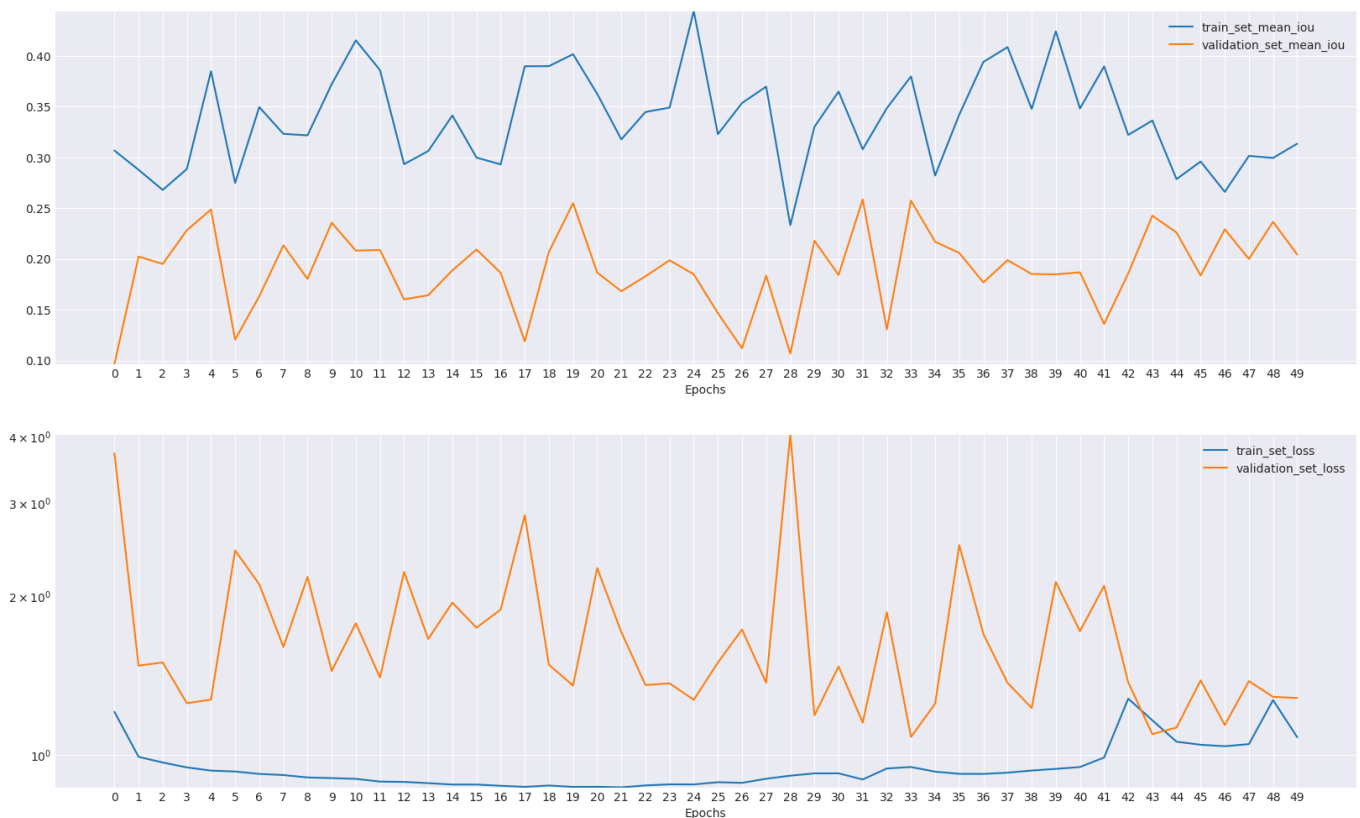


Figure 4: Learning Curve Baseline



3. Résultat des différentes « loss » fonction

Le but ici est de rechercher la fonction de coût la plus adaptée et qui permet de meilleur résultat, pour ça j'ai testé 3 approches :

- "categorical_crossentropy" : Cette fonction viens directement de Tensorflow elle permet de calculer l'entropie croisée de notre modèle multi-classes.
- "dice_loss" : Cette fonction n'est autre que « $1 - \frac{2 * \text{intersection}}{\text{union}}$ » (qui correspond à : $2 * \text{intersection} / \text{union}$)
- "combinaison_loss" : Cette fonction est une combinaison de 2 fonction coût ($\text{categorical_crossentropy} + (3 * \text{dice_loss})$)
- "combinaison_loss_v2" : Cette fonction est une combinaison de 2 fonction coût ($\text{categorical_crossentropy} + \text{dice_loss}$)

Voici les résultats des différentes modélisations avec les fonctions coût :

- "categorical_crossentropy" : **0.2584** score (**0.3079** training score)
- "dice_loss" : **0.2710** score (**0.3480** training score)
- "combinaison_loss" : **0.3096** score (**0.3554** training score)
- "combinaison_loss_v2" : **0.3084** score (**0.3508** training score)

En l'état, la Meilleur fonction coût semblent être le « combinaison_loss », il est difficile d'interpréter les résultats avec clarté, il serais nécessaire de voir ce que donne cette apprentissage sur un plus grand nombre d'épochs pour avoir des résultats plus réaliste. Il semble cependant que la learning curve indique un léger Overfitting du modèle avec la fonction coût : « combinaison_loss ».

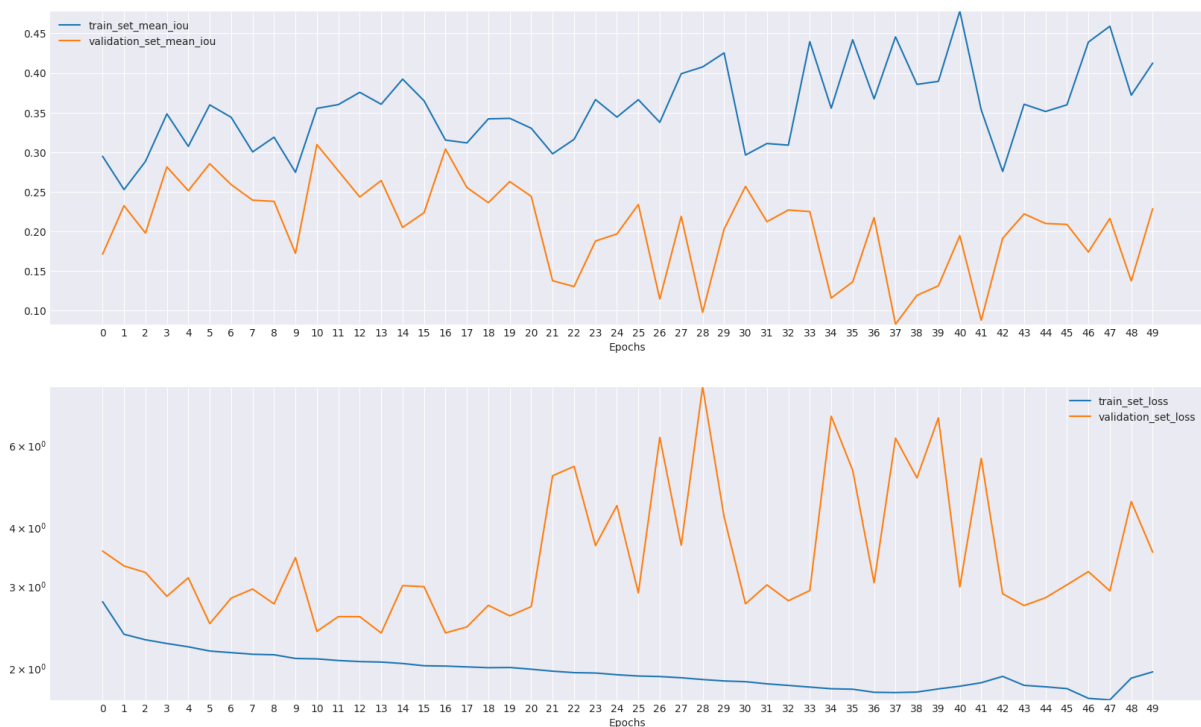


Figure 5 : Learning curve (overfitting)



IV. Optimisation/Choix structure finale

1. Augmentation des images

Pour essayer d'améliorer nos modélisations j'ai utilisé une augmentation des images, le jeu de données fournis nous met à disposition « seulement » 3000 images.

Pour ce faire j'ai utilisé la librairie « **imgaug** » qui permet d'appliquer des transformations sur une images et d'appliquer cette même transformation sur le masque de notre segmentation sémantique.

Cette méthode permet de fournir un plus grand nombre d'images en « transformant » l'image d'origine pour en obtenir un certain nombre de dérivé.

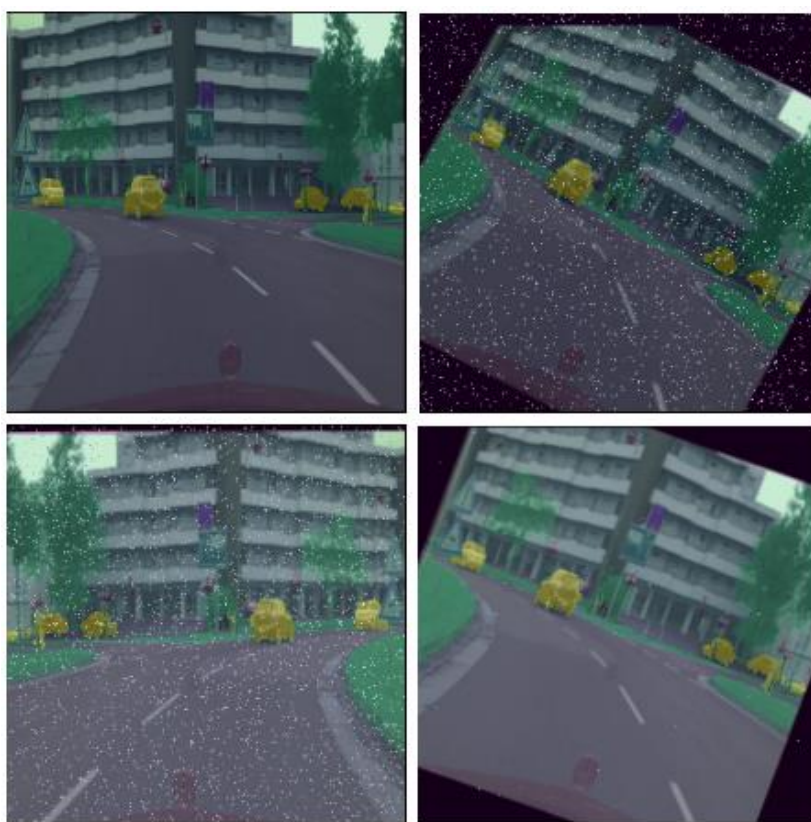


Figure 6 : Augmentation des images

L'image d'origine et son masque (Figure x, image en haut à gauche) on dans cet exemple était augmenter 4 fois avec des transformation aléatoire, comme des rotations, un retournement, du bruit salé (point blanc) et des déplacement horizontal et vertical.



2. Résultat des différentes structures

Modélisation ML classique linéaire : **0.2941** score (preparation_data => 3600s + fit => 6000s)

Modélisation UNet sans augmentation d'images **0.3096** score (50 epochs => 190s)

Modélisation UNet différente Loss fonction :

- "categorical_crossentropy" : **0.2584** score (50 epochs =>190s)
- "dice_loss" : **0.2710** score (50 epochs =>190s)
- "combinaison_loss" : **0.3096** score (50 epochs =>190s)
- "combinaison_loss_v2" : **0.3084** score (50 epochs => 190s)

Modélisation UNet avec augmentation d'images **0.3059** score (50 epochs => 405s)

Modélisation UNet transfert learning (VGG16 et ResNet50 partie encoder) :

- VGG16 sans augmentation des données : **0.3703** score (50 epochs =>195s)
- ResNet 50 sans augmentation des données : **0.3491** score (50 epochs => 200s)

Modélisation FPN (backbone efficientnet) :

- FPN sans augmentation des données : **0.3691** score (50 epochs => 195s)

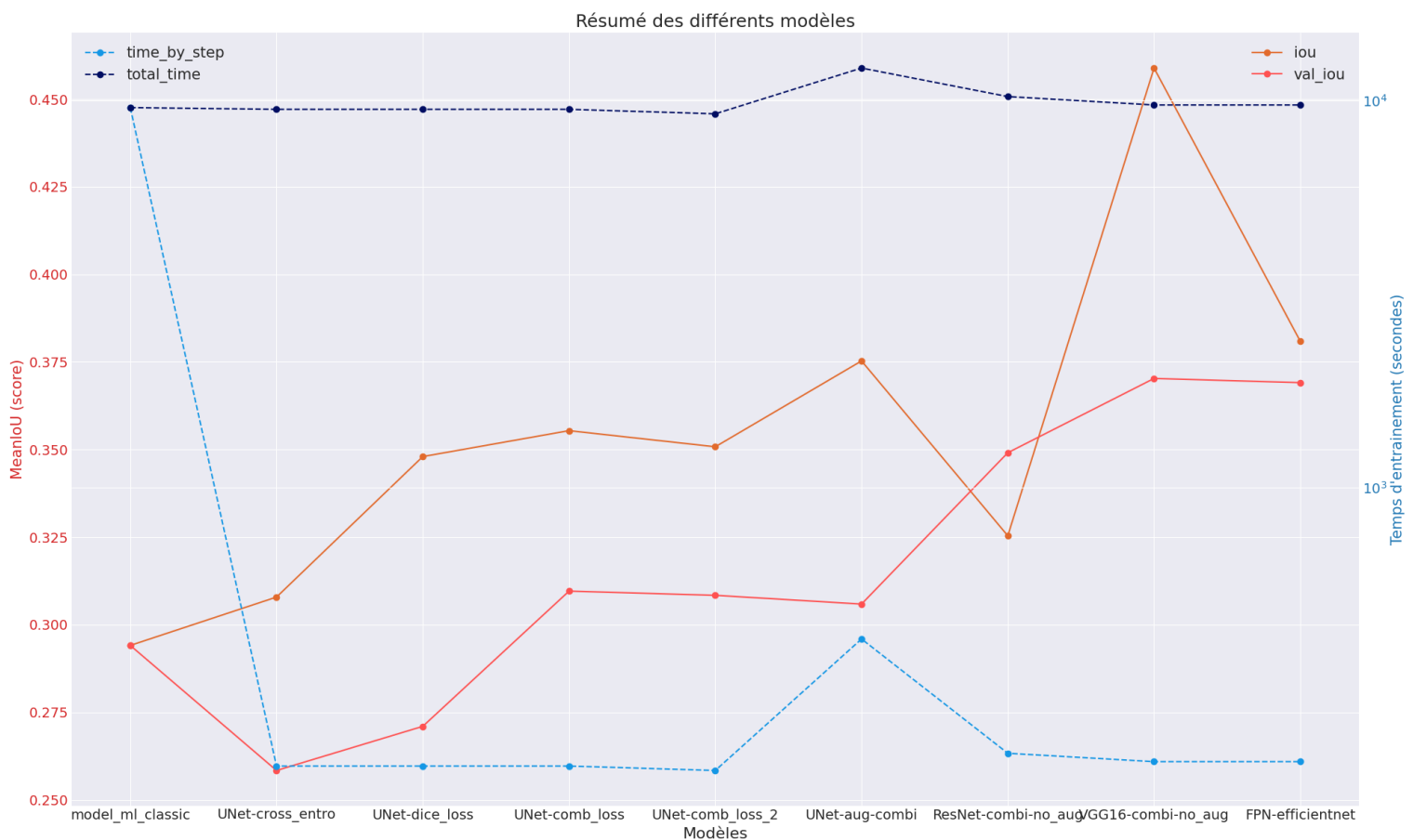


Figure 7 : Synthèse des différentes modélisations

Si on prend les résultats de ce tableau, les 2 modélisations les plus prometteuses semblent être la modélisation UNet avec transfert learning sur la partie encodeur (structure VGG16), les données non augmentées et la fonction de coût « combinaison_loss » avec un score de validation qui atteint **0.3703**.



V. Déploiement du modèle final (Flask API)

1. Sauvegarde/entrainement du modèle

1.1. Connexion Azure

Il est nécessaire de créer une authentification « Service Principal » pour la connexion sécurisée au groupe de ressource, nécessaire pour récupérer le model stocker sur Azure Machine Learning.

Pour ça il suffit de se connecter sur le portail Azure et sélectionner la ressources « [Azure Active Directory](#) » et de sélectionner « inscription d'application » dans le panel à gauche

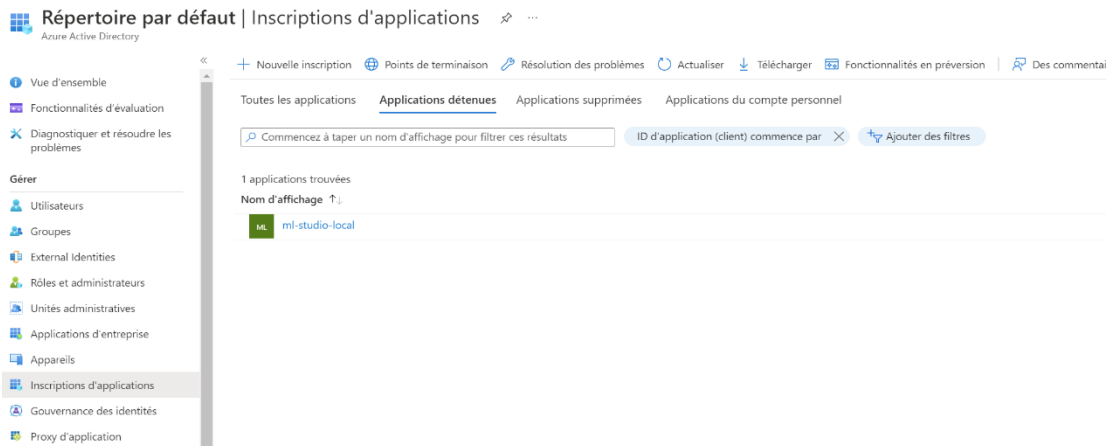


Figure 8 : Azure active directory (application)

Puis créer une « nouvelle inscription », il suffit ensuite de choisir un nom d'application (ici pour mon test j'ai choisi « ml-studio-local ») et de sélectionner le type de pris en charge souhaiter (garder le premier dans notre cas)

Une fois créer nous avons notre compte d'application qui est disponible, il suffit ensuite de créer un mot de passe pour la connexion sécurisée à notre Workspace par l'application

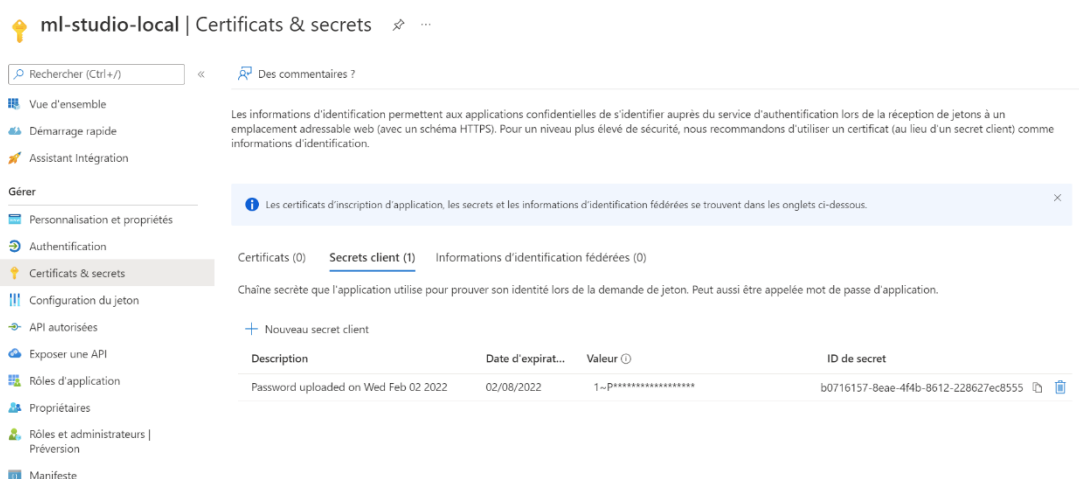


Figure 9 : Azure active directory (certificat & secrets)

Une fois créer nous avons toutes les données nécessaires pour une connexion sécurisé, il suffit ensuite de rajouter cette application dans notre groupe de ressources de la manière suivante



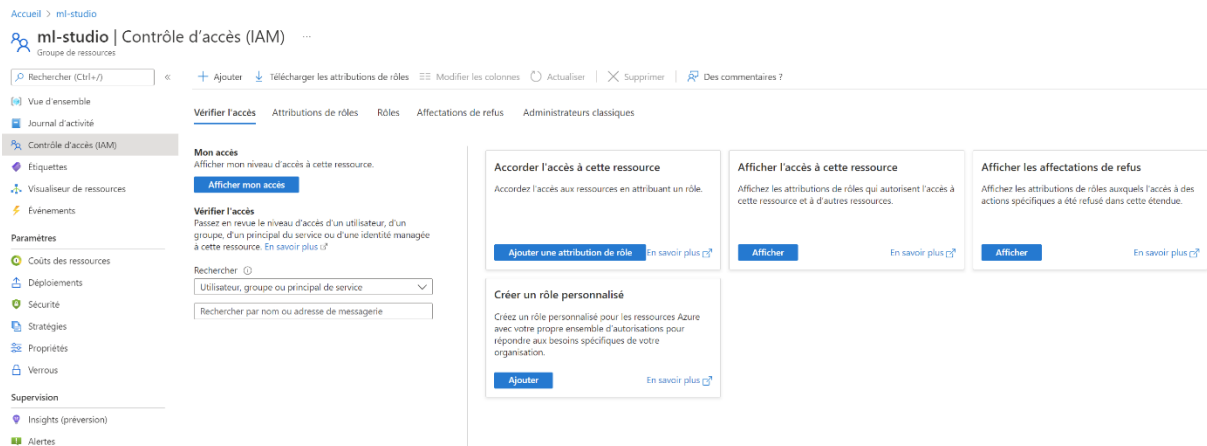


Figure 10 : Groupe de ressources (Contrôle d'accès)

Il suffit d'aller dans le « Contrôle d'accès (IAM) » et d' « Ajouter une attribution de rôle » en contributeur.

1.2. Création du Datastore

Après avoir créer le système de connexion par « Service Principal Authentification », il est nécessaire de stocker les différentes images nécessaires à l'entraînement de notre modèle.

Pour ça il existe plusieurs solutions, la création d'un Datastore ou utiliser un Datastore existant, Pour ma part j'ai pris le Datastore par défaut « workspaceblobstore » qui est créer automatiquement lors de la création d'un espace de travail Azure Machine Learning.

Après avoir créé/identifié le Datastore, il suffit d'exécuter le code suivant :

```
from azureml.data.datapath import DataPath
from azureml.core import Dataset, Datastore

datastore = Datastore.get(ws, 'workspaceblobstore')

_ = Dataset.File.upload_directory(src_dir='./data/output/',
                                target=DataPath(datastore, 'dataset/ouputs'),
                                show_progress=True, overwrite=True)
```

Figure 11 : Code ajout de données Datastore

Il suffit ensuite de spécifier les fichier/dossiers sources (« src_dir ») et le dossier de destination (« target »).



1.3. Création du script d'entraînement

Pour la création du script d'entraînement il suffit de créer un fichier « train.py » qui sera exécuté lors du déploiement de « l'expérience », pour ça il suffit de créer un nom « d'expérience » et de créer une config et un environnement pour exécuter le déploiement du script d'entraînement.

```
from azureml.core import Workspace, Experiment, Environment, ScriptRunConfig

experiment = Experiment(workspace=ws, name='experiment_train_tensorflow')

config = ScriptRunConfig(source_directory='./src',
                        script='train.py',
                        compute_target='train-model-fvt')

env_tf = ws.environments['AzureML-tensorflow-2.4-ubuntu18.04-py37-cuda11-gpu']

config.run_config.environment = env_tf

run = experiment.submit(config)

run.wait_for_completion(show_output=True)
```

Figure 12 : Code Entraînement d'un modèle

Il est également possible de créer son propre environnement et d'ajouter d'autre fichier dans le « source_directory » complémentaire au script « train.py », et il suffit ensuite de créer une « compute_target » créer préalablement dans le workspace.



2. Création d'une WebAPP Azure/ déploiement du model (ACI)

Pour l'ACI et l'application Flask, la modélisation est récupérée directement sur l'espace de travail Azure, cela permet de mieux gérer le versionning des différentes modélisations et d'avoir un espace commun pour l'enregistrement des modélisations.

2.1. ACI déploiement

Le déploiement par ACI (Azure Container Instance) permet de compléter le cycle Azure MLOps offert par l'espace de travail Azure Machine Learning.

Une fois notre modélisation entraînée et enregistrée dans l'espace AzureML, il est très facile de déployer notre modélisation dans une ACI ou un cluster AKS si c'est une mise en production.

Pour ce faire il suffit d'exécuter les lignes de code suivantes et de créer un script d'inférence qui permet les interactions avec la modélisation sauvegardée.

```
from azureml.core.environment import Environment
from azureml.core.model import InferenceConfig, Model
from azureml.core.webservice import AciWebservice, Webservice

env_tf = ws.environments['AzureML-tensorflow-2.4-ubuntu18.04-py37-cpu-inference']

inference_config = InferenceConfig(source_directory=source_directory,
                                   entry_script="x/y/score.py",
                                   environment=env_tf)

deployment_config = AciWebservice.deploy_configuration(cpu_cores = 2, memory_gb = 4, auth_enabled=True)

service = Model.deploy(
    workspace = ws,
    name = "semantic-segmentation",
    models = [final_model],
    inference_config = inference_config,
    deployment_config = deployment_config)

service.wait_for_deployment(show_output = True)
```

Figure 13 : Code déploiement d'un modèle

Avec ça notre modélisation est automatiquement déployée et accessible (l'inférence se fait avec le script fourni, ici le « score.py »).

2.2. Flask déploiement

Pour le déploiement Flask il existe plusieurs solutions explicitées dans la [documentation Azure](#) pour faciliter le déploiement de notre application Flask.

Après avoir créé son application Flask il suffit de faire le déploiement via Azure Web APP (par GitHub, ou directement via Azure CLI par exemple)



3. Consommation de l'API

3.1. ACI

Pour le déploiement ACI/AKS il s'agit uniquement d'une API sans interface, pour récupérer les informations il suffit de rentrer l'ID de l'image que l'on souhaite segmenter (Les données sont récupérées via un Datastore et sont téléchargées au déploiement du modèle pour notre exemple)

Le script d'inférence nous renvoie les données de l'image segmenter au format Numpy sous forme de List Python il suffit donc de transformer la List en tableau Numpy et de lire l'image segmenter.

3.2. Flask

Avec l'application Flask le rendu est plus visuel, on retrouve une interface pour sélectionner l'ID de l'image que l'on souhaite segmenter, et le rendu directement accessible via l'interface.

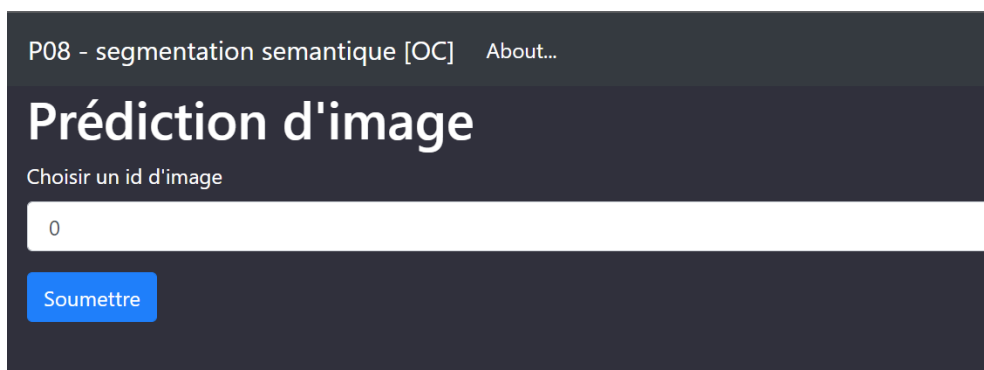


Figure 14 : Flask choix identifiant images

L'application contient 20 images du jeu de données initial dans le répertoire « static/data/img ».

Voici un exemple de rendu de l'application après la segmentation effectuée sur l'image, on retrouve également le code couleur utilisé dans la segmentation.

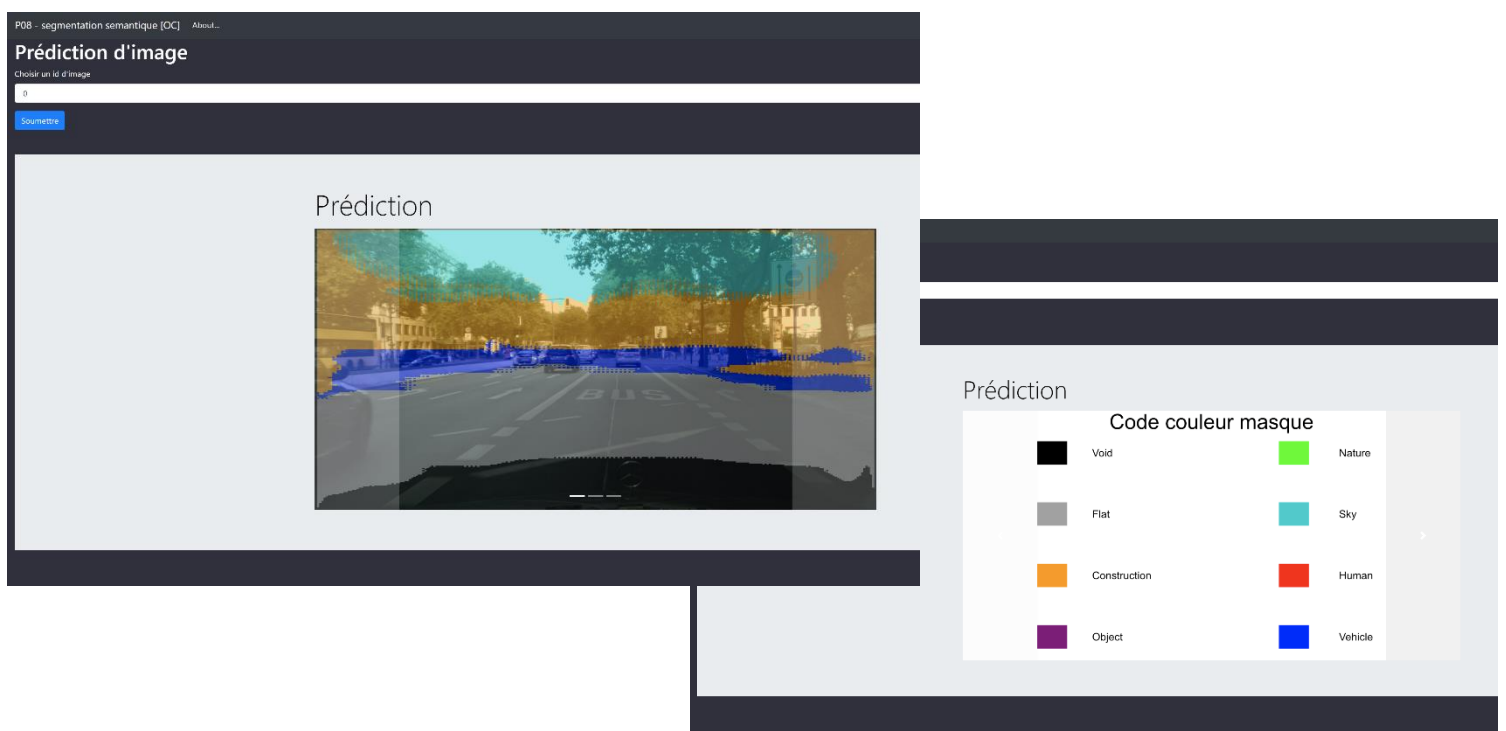


Figure 15 : Flask prédiction



VI. Amélioration possible (Pour aller plus loin)

1. Optimisation des paramètres

Actuellement les modélisations sont loin d'être optimales (score MeanIoU de seulement 0.37xx).

Pour augmenter ce score il existe plusieurs solutions envisageables :

- Optimisation des transformations d'images pour l'augmentation d'images.
- Changer « l'optimizer » de notre modélisation (adam, rmsprop, adamax, SGD) ou ajuster le learning rate.
- Trouver d'autres modélisations plus intéressantes.
- Augmenter le nombre d'images initiales.
- Réduire/Augmenter le nombre de batch
- Choisir des tailles d'image plus grandes (256x256 => 512x512 par exemple)

Et une fois que l'optimisation est finalisée :

- Augmenter grandement le nombre d'époques (actuellement 50, passer à 10.000 ou plus)

2. Mise à jour du Datastore

Le Datastore contient actuellement 2950 images pour l'entraînement 500 pour l'évaluation puis 1525 pour les tests (sans output).

La mise en place du système de récupération des images permettra d'augmenter facilement le nombre d'images, il suffira ensuite de faire le découpage (segmentation/masque) des images récupérées pour augmenter le nombre d'images disponibles à l'entraînement de notre modélisation.

Pour ça Azure permet la mise à jour de Dataset et met en place un système de versionning des différents Dataset qui peut être intéressant à mettre en place par la suite.

3. Déploiement automatisé

Actuellement le flow MLOps n'est pas encore abouti, l'acquisition des données, l'entraînement, le déploiement ne sont pas automatisés et fonctionnent uniquement avec des scripts que l'on exécute manuellement.

Il serait donc intéressant de compléter le cycle MLOps de ce projet avec la mise à jour du Datastore, le entraînement/déploiement automatique (via GitHub si WebAPP ou mlflow si AKS/ACI).



VII. Source

- Création d'une application WebApp : <https://docs.microsoft.com/fr-fr/azure/app-service/quickstart-python?tabs=cmd&pivots=python-framework-flask>
- Importation des variables d'environnement dans l'environnement Azure : <https://azure.github.io/azureml-cheatsheets/docs/cheatsheets/python/v1/environment/>
- Authentification Azure : <https://github.com/Azure/MachineLearningNotebooks/blob/master/how-to-use-azureml/manage-azureml-service/authentication-in-azureml/authentication-in-azureml.ipynb>
- Entraînement et déploiement du modèle sur Azure : <https://docs.microsoft.com/fr-fr/azure/machine-learning/tutorial-train-deploy-notebook>
- Réalisation des générateur de données : <https://deeplylearning.fr/cours-pratiques-deep-learning/realiser-son-propre-generateur-de-donnees/>
- Segmentation sémantique des images : <https://www.coursera.org/lecture/advanced-computer-vision-with-tensorflow/image-segmentation-overview-m8zpr>
- Augmentation des images : https://imgaug.readthedocs.io/en/latest/source/examples_segmentation_maps.html
- Cityscapes GitHub : <https://github.com/srihari-humbarwadi/cityscapes-segmentation-with-Unet>
- Modélisation Linéaire (ML) segmentation sémantique : <https://www.youtube.com/watch?v=uWTzkUD3V9g>
- Métrique segmentation sémantique : <https://ilmonteux.github.io/2019/05/10/segmentation-metrics.html>
- Librairie segmentation-model : <https://segmentation-models.readthedocs.io/en/latest/tutorial.html>
- Azure Dataset : <https://docs.microsoft.com/en-us/azure/machine-learning/how-to-create-register-datasets>
- Déploiement WebApp : <https://docs.microsoft.com/fr-fr/learn/modules/host-a-web-app-with-azure-app-service/6-deploying-code-to-app-service>

