**Segmentation d’images routières via API**

**Avant-propos :**

Cette note technique a pour but d’aiguiller l’équipe dans son utilisation du modèle de segmentation d’images routières pour construire la chaîne de décision d’un véhicule autonome. Elle décrit à la fois les raisons des choix techniques effectués et les interfaces des différents éléments permettant la mise en œuvre du modèle. Elle supporte la mise en œuvre du premier modèle fourni, mais doit surtout servir de base à un travail d’amélioration continue et de recherche pour optimiser la fonction désirée au sein de la chaîne de décision.

**Table des matières :**

I – Contexte

I-1 Bref état de l’art en segmentation sémantique d’images

I-2 Données d’entraînement

I-3 Structure du code

II – Modèles testés

II-1 VGG16 et classifieur

II-2 FCN

II-3 Unet

II-4 Dilatednet

II-5 HRnet

II-6 Comparatif

III – Optimisation du modèle :

III-1 Comparaison de métriques sur unet :

III-2 Augmentation de données sur hrnet :

IV – API

Conclusion

Bibliographie

**I – Contexte**

**I-1 Bref état de l’art en segmentation sémantique d’images :**

La segmentation sémantique d’images est en constante évolution et le succès du deep learning a fait exploser la quantité de recherches effectuées dans ce domaine. Le monde académique et les acteurs majeurs de l’IA développent sans cesse de nouvelles techniques et architectures. Les informations données ici constituent une synthèse non exhaustive de cette évolution sur les dernières années, qui s’appuie principalement sur les réseaux LeNet et AlexNet.

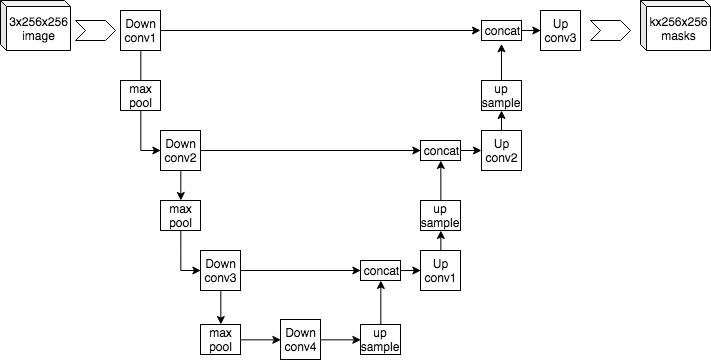
**I-1-a – FCN :**

L’un des premiers développements en la matière est l’encodeur décodeur FCN [1=431] de Long et al.[5], qui généralise l’usage de la déconvolution et amène la notion de Skip Connection.

Ce type de réseau utilise pour la première fois des couches de déconvolution qui permettent d’apprendre à reconstruire l’image encodée par rétropropagation au lieu d’interpoler. Il est aussi le premier réseau de segmentation sémantique qui remplace les couches fully-connected par des blocs convolutionnels et permet donc de s’affranchir des problèmes de taille d’images.

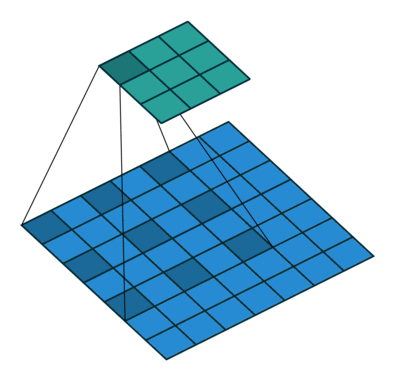
**I-1-b – U-net :**

Le U-net, proposé par l’université de Fribourg [1], étend l’utilisation des skip connections et propose des améliorations fonctionnelles qui élèvent la performance du modèle – notamment l’augmentation du nombre de filtres pour la déconvolution. Ce modèle proche du FCN a servi de base architecturale pour de nombreuses applications et des implémentations permettant de sélectionner des modèles existants voire pré-entraînés ont été développées.



**I-1-c – DilatedNet :**

Les DilatedNets sont basés sur la convolution dilatée ou « à trous » :



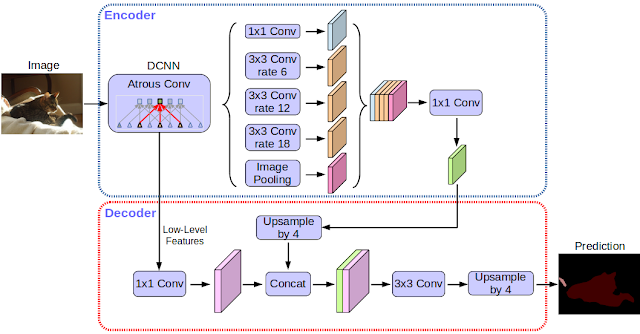
Les réseaux utilisant ce principe de convolution permettent d’extraire des informations contextuelles de façon moins coûteuse qu’avec les ConvNets classiques. F.Yu[2] en a initialement développé l’idée, qui a été largement intégrée aux architectures de Deep Learning pour la segmentation sémantique.

L’extraction d’informations contextuelles via convolution à trous a permis de surpasser les modèles précédemment existants.

**I-1-d – ASPP/Deeplab :**

Chen et al. [3] proposent une architecture pyramidale à base de convolution à trous, permettant la mise en œuvre de tout modèle CNN pré-entraîné sur une tâche de classification d’image :

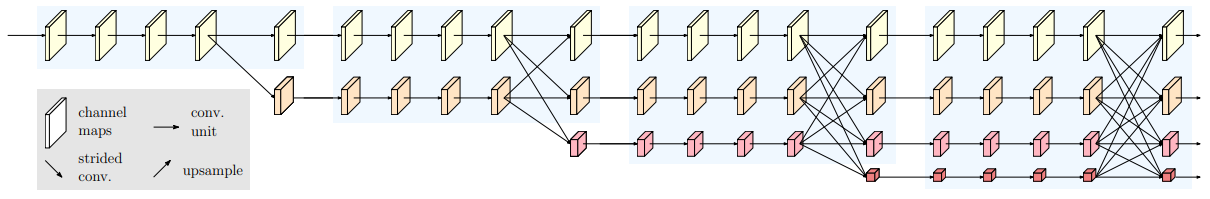
Aucune implémentation de la version papier n’a été trouvée sous Keras et ce modèle n’a donc pas été testé, mais il existe aujourd’hui en 3 versions et fait partie des candidats potentiels pour notre tâche et mériterait d’être implémenté et évalué.



**I-1-e – HRNet :**

Initialement conçue pour les applications médicales, l’architecture HRnet permet comme son nom l’indique un traitement rapide d’images en haute résolution. Elle consiste à faire opérer en parallèle des réseaux convolutionnels qui travaillent sur différentes résolutions, et à regrouper les outputs de chaque « branche » du réseau en entrée de la couche suivante.

L’exemple suivant est donné dans le papier [4] , et l’implémentation testée au sein du projet est assez similaire :



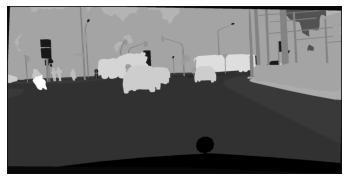
Pour notre application :

Il est à noter que la vision pour véhicules autonome n’utilise pas que des modèles de Deep Learning et que certaines techniques autres que la segmentation sémantique pourraient être envisagées, notamment afin de permettre une prédiction des déplacements à venir des objets segmentés – cela pourrait permettre de faciliter le travail du système de décision sans sacrifier les performances.

**I-2 Données d’entraînement :**

Le jeu de données Cityscapes est constitué de photos routières de haute qualité et de haute résolution (2048 \* 1024) capturées dans 50 villes différentes d’Allemagne.

Il contient des photos routières et des masques d’annotation qui divisent ces photos routières en 34 catégories, groupables en 8 catégories majeures.



L’ensemble contient 5000 images réparties en 2975 images d’entraînement, 500 images de validation et 1525 images de test.

Compte tenu de leur haute résolution ces images contiennent des informations très détaillées sur les classes inhérentes à notre problème de segmentation et contiennent déjà probablement une diversité d’échelles assez large pour chaque classe d’objet routier, mais elles posent plusieurs problèmes de par leur encombrement :

* Les temps d’entrainement et de prédictions pourraient être très longs
* L’entraînement sur simple GPU est irréalisable car la mémoire serait immédiatement saturée

Il a été décidé que les évaluations porteraient sur des modèles travaillant sur des images de 224\*224 ou 512\*512 selon possibilités. Selon la taille d’image obtenue par le système d’acquisition définitif, une adaptation du resize en entrée sera peut-être nécessaire pour maintenir de bonnes prédictions.

**I-3 : Structure du code :**

Le code est structuré pour permettre l’entraînement, le suivi des métriques et le déploiement via Azure et est donc constitué de :

* 1 script de contrôle permettant l’exécution des scripts d’entraînement sur un cluster de calcul
* 1 script contenant la classe « générateur augmenté » développée dans le cadre du présent projet
* Plusieurs scripts d’entraînement contenant ledit générateur ainsi que diverses implémentations de modèles testés
* 1 script d’application Flask permettant de déployer le modèle sur API et webapp pour inférence
* 1 script client permettant l’exécution de requêtes à l’API

Le tout fonctionne avec AzureML selon la logique de liaison suivante :

Navigateur Web

Plan App Service

Webapp

Script client

Script d’application

Cluster

Workspace

Script de contrôle

AzureML

Dataset

Script d’entraînement

Model

**II – Modèles testés :**

**II-1 : VGG16 + Random Forest**

Ce modèle destiné à servir de baseline pour la comparaison a été mis en œuvre en utilisant 2 blocs convolutionnels du VGG16 avec poids pré-entraînés Imagenet de Keras et le Random Forest de SKLearn. Seul un entraînement partiel a pu être réalisé sachant que les feature maps produites par ce réseau occupent 75GB pour le seul jeu d’entraînement, et que la librairie scikit-learn ne permet pas la génération de données d’entraînement « à la volée » depuis un fichier. L’entraînement du random forest a donc été réalisé sur la base de 400 images. L’ensemble donne des images prédites intelligibles bien qu’approximatives, mais les indicateurs montrent des résultats très bas (mean IoU : 3e-4, Dice coeff : 0.33).

**II-2 : FCN**

Dans cette implémentation de S.Humbarwadi, des couches de filtres convolutionnels de dimension (3,3) sont systématiquement associées à une couche de batch-normalization et à une activation relu.

Chaque bloc contient 2 couches en cascade, et quelques couches de régularisation sont utilisées. Lors de la déconvolution, les prédictions de chaque bloc de convolution sont concaténées aux inputs du bloc de déconvolution de même taille, à la manière d’un U-net. C’est donc déjà une version avancée d’un FCN.

Les blocs sont associés de la manière suivante :

32 filters

64 filters

128 filters

256 filters

32 filters

64 filters

128 filters

256 filters

512 filters

Dropout

Ce réseau a un temps d’entraînement de 8 à 10h sur 3 machines virtuelles équipées de GPU.

Les résultats chiffrés sont bons mais les masques obtenus étant assez peu intelligibles, ce modèle n’a pas été retenu pour l’application.

**II-3 : DilatedNet + interpolation bilinéaire :**

Ce réseau également proposé par Humbarwadi suit les mêmes principes architecturaux que le précédent mais est basé sur des couches de convolution/déconvolution à trous, le retour aux dimensions originale de l’image prédite est assuré par interpolation bilinéaire.

L’encodeur contient 8 couches de convolution dont 2 dilatées.

Temps d’entraînement : 8h

Les résultats chiffrés ainsi que les masques prédits sont les moins bons du comparatif.

**II-4 : Unet :**

II-4-a : Unet à base de Mobilenetv2 pré-entraîné

Ce Unet utilise comme encodeur un Mobilenetv2 (FCN). Cette implémentation comprend des skip connections qui récupèrent les outputs des différents étages de l’encodeur pré-entraîné. Celles-ci entrent dans des blocs de déconvolution pix2pix.upsample (couche deconv2D+BN pré-implémentée dans la collection d’exemples tensorflow) le redimensionnement final est assuré par un étage de déconvolution.

Mobilenetv2 est implémenté sous keras pour des images de 224,224,3 et ne sera pas testé sur des images de 512\*512.

Les temps d’entraînements étaient d’environ 4h mais les résultats étaient assez mauvais (Dice coeff < 0.6) et les masques inintelligibles, les couches mobilenet ont donc été rendues trainable.

Le temps d’entraînement sur 3 vm avec GPU a été de 24h.

Les résultats chiffrés et les masques prédits sont plutôt satisfaisants, mais il n’y a pas eu un grand avantage à utiliser un modèle pré-entraîné.

II-4-b : Unet à base de VGG16 pré-entraîné :

Ce Unet utilise comme encodeur un VGG16 avec poids imagenet fourni par la librairie Segmentation\_models dont les paramètres sont gelés pour l’entraînement.

Ce modèle a été utilisé pour comparer les fonctions de loss et les effets des différentes techniques d’augmentation, mais ne fait pas originellement partie des modèles comparés.

**II-5 : HRNet**

Ce modèle décrit en partie 1 a majoritairement été implémenté sous pytorch, cependant une implémentation Keras a été codée et partagée sur Github par Nick Nie (aka NieCongChong).

C’est cette implémentation qui a été adaptée à nos données, à notre générateur et à nos moyens d’entraînement.

Le temps d’entraînement est de 24h et les résultats sont probants tant sur le plan visuel que sur le plan du coefficient de Dice obtenu : ce modèle se distingue par sa capacité à segmenter de petits objets comme les panneaux et les poteaux.

**II-6 : Comparatif**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Modèle** | **VGG16 +**  **Random Forest** | **DilatedNet** | **FCN** | **Unet Mobilenet** | **HRnet** | **Unet VGG16** |
| **Temps d’entraînement (h)** | 2 | 8 | 8 | 24 | 16 | 15 |
| **Coefficient de Dice** | 0.33 | 0.72 | 0.79 | 0.80 | 0.82 | 0.91 |
| **IoU** |  | 0.33 | 0.45 | 0.46 | 0.693 | 0.665 |
| **Taille d’image traitée (pixels)** | 224 | 224 | 224 | 224 | 512 | 512 |
| **Temps d’inférence (s)** | 0.2 | Ø | 0.65 | 0.65 | 0.65 | Ø |

Les 2 réseaux les plus performants sont aussi les plus longs à entraîner (sur un cluster de 3 vm avec GPU). Le temps d’inférence est peut-être plus important, mais vu l’absence de réelle différence en la matière, on optera simplement pour les modèles aux scores les plus élevés – il semble que le prétraitement appliqué (redimensionnement cv2 + normalisation) coûte plus de temps que l’opération de prédiction.

Nb : le Unet VGG16 ne fait pas partie du comparatif initial et a été ajouté a posteriori – celui-ci ayant optimisé un autre fonction de perte, sa performance n’est donc pas directement comparable à celles des autres modèles, ses données sont rendues disponibles pour information, et dans un souci de cohérence avec les données présentées ci-dessous.

**III – Optimisation du modèle :**

A propos des métriques :

3 métriques ont été envisagées et testées :

* Accuracy
* Coefficient de Dice
* Indice de Jaccard (=mean IoU)

Le déséquilibre de classes étant fort, l’accuracy n’est clairement pas une métrique adaptée. Les 2 coefficients utilisés sont normalisés et leur complément à 1 peut donc directement être utilisé comme fonction de perte. La métrique principale est donc choisie dans la partie suivante pour aller de pair avec la fonction de loss.

**III-1 Comparaison de fonctions loss sur U-net pré-entraîné :**

Les fonctions de loss ont été comparées en entraînant un modèle U-net pré-entraîné.

Ont été testées :

* CategoricalCrossentropy
* Dice\_loss (=1-Dice Coeff)
* Jaccard\_Loss (=1-IoU) + CCE

Le coefficient de Dice a donc été conservé en tant que métrique principale.

Résultats des essais de fonctions de perte :

|  |  |  |  |
| --- | --- | --- | --- |
| Loss | CCE | Dice | Jaccard+CCE |
| Accuracy | 0.68 | 0.74 | 0.96 |
| Dice coeff | 0.740 | 0.856 | 0.957 |
| Val Dice coeff | 0.721 | 0.818 | 0.910 |
| Jaccard Index | 0.521 | 0.460 | 0.782 |
| Val Jaccard Index | 0.488 | 0.426 | 0.666 |

La loss Jaccard + CCE est donc retenue comme fonction d’optimisation du modèle.

**III-2 Comparaison de procédés d’augmentation de données sur HRnet :**

Les augmentations ont été testées avec le Unet VGG16 sur Tensorflow 2.5.0 avec GPU et CPU.

Le GPU est limité de par sa mémoire limitée à une batch size de 1, alors que la batch size sur CPU est fixée à 8. La différence de batch size cause une différence dans les scores obtenus, on observe aussi des écarts plus prononcés entre les différents modèles entraînés sur CPU avec une batch size de 8.

Les temps d’entraînement sont d’environ 45 minutes par epoch sur GPU et 1h15 par epoch sur CPU.

Les résultats semblent reproductibles puisque les scores d’un modèle entraîné sur CPU sans augmentation avec une batch size de 1 sont très proches de ceux obtenus sur GPU dans les mêmes conditions.

Les augmentations sont traitées à la volée par le générateur, si le traitement d’une image ou non est décidé aléatoirement, la probabilité d’augmentation est fixée ainsi que les probabilités individuelles de chaque procédé utilisé.

L’écart type de l’angle de rotation et du niveau de zoom sont également fixés dans le code du générateur.

Les techniques suivantes ont été testées :

* Rotation
* Retournement horizontal
* Changement d’échelle

Essais de rotation sur GPU :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Augmentation | Aucune | 1:5 – 15 deg | 1:5 – 30 deg | 1:5 – 45 deg |
| Temps d’entraînement (h) | 16 | 17 | 18 | 17 |
| IoU | 0.755 | 0.760 | 0.762 | 0.708 |
| Val IoU | 0.665 | 0.670 | 0.670 | 0.658 |
| Dice | 0.944 | 0.945 | 0.948 | 0.934 |
| Val Dice | 0.905 | 0.906 | 0.906 | 0.903 |
| Loss | 0.291 | 0.284 | 0.279 | 0.350 |
| Val loss | 0.435 | 0.430 | 0.429 | 0.486 |

Essais d’augmentations diverses sur CPU :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Augmentation | Aucune | Rotation 1 :5  30 deg | Blur 1 :4 | Zoom 1:5 s=0.5 |
| Temps d’entraînement (h) | 31 | 50 | 20 | 19 |
| IoU | 0.7819 | 0.801 | 0.812 | 0.741 |
| Val IoU | 0.667 | 0.723 | 0.719 | 0.710 |
| Dice | 0.957 | 0.937 | 0.940 | 0.920 |
| Val Dice | 0.906 | 0.904 | 0.903 | 0.895 |
| Loss | 0.242 | 0.255 | 0.238 | 0.310 |
| Val loss | 0.448 | 0.375 | 0.381 | 0.392 |

Essais sur GPU de retournement horizontal, variations de luminosité, égalisation d’histogramme :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Augmentation | Aucune | Horizontal Flip 1 :5 | Brightness  1:4 s = 0.5 | Egalisation systématique | Zoom  1:5 s=0.5 |
| Temps d’entraînement (h) | 15 | 18 | 15 | 12 | 16 |
| IoU | 0.755 | 0.763 | 0.761 | 0.766 | 0.728 |
| Val IoU | 0.665 | 0.670 | 0.672 | 0.671 | 0.670 |
| Dice | 0.944 | 0.949 | 0.945 | 0.950 | 0.934 |
| Val Dice | 0.905 | 0.906 | 0.908 | 0.907 | 0.905 |
| Loss | 0.291 | 0.280 | 0.285 | 0.276 | 0.285 |
| Val loss | 0.435 | 0.434 | 0.430 | 0.433 | 0.430 |

Certaines techniques semblent retarder l’apprentissage, et le coefficient de Dice en apprentissage baisse naturellement mais aucune des méthodes testées n’a permis d’améliorer les scores en validation du modèle testé au terme de son apprentissage.

Les meilleurs scores étant atteints sans aucune augmentation de données, on conserve finalement le modèle entraîné sans augmentation de données.

Essais finaux sur GPU avec combinaisons d’augmentations de données :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cas | Unet – no\_augmentation | HRnet\_no\_augmentation | Unet full\_augm | HRnet full\_augm |
| Temps d’entraînement (h) | 15 | 18 | 16 | 24 |
| IoU | 0.755 | 0.776 | 0.775 | 0.783 |
| Val IoU | 0.665 | 0.693 | 0.675 | 0.702 |
| Dice | 0.944 | 0.945 | 0.947 | 0.944 |
| Val Dice | 0.905 | 0.906 | 0.9101 | 0.913 |
| Loss | 0.291 | 0.270 | 0.251 | 0.262 |
| Val loss | 0.435 | 0.384 | 0.387 | 0.365 |

**IV – API :**

L’API Flask a été déployée à l’aide d’Azure App Service. Une authentification Service Principal a été mise en place pour permettre à l’app d’accéder aux ressources projet (données, modèle…) mais le service reste public.

La page de bienvenue permettant d’en vérifier l’activité est consultable à l’adresse suivante :

[https://cityscapes-webapi.azurewebsites.net](https://cityscapes-webapi.azurewebsites.net/)

Un script client python est fourni en annexe du présent document afin de permettre une utilisation simplifiée de l’API.

Celui-ci contient 3 cellules de code, chacune permettant l’accès à un mode d’utilisation différent :

* Cellule 1 : reçoit comme arguments l’adresse du serveur d’API, le chemin du dataset local et au besoin l’index de l’image à traiter (s’il n’est pas renseigné, un index est généré aléatoirement). L’image et le masque sont ouverts localement, puis l’image est envoyée à l’API pour inférence. La réponse contient un message concernant le temps de traitement et la taille d’image, ainsi qu’une image fusionnée du masque et de l’image d’origine.
* Cellule 2 : reçoit comme arguments l’adresse du serveur d’API et le chemin d’une image locale. L’image va subir un padding à la valeur 0 aux dimensions des photos du dataset avant d’être redimensionnée pour l’inférence. La réponse contient un message concernant le temps de traitement et la taille d’image, ainsi qu’une image fusionnée du masque et de l’image d’origine. L’image reçue est rognée de façon à revenir aux dimensions d’origine. A utiliser si les déformations causées par le redimensionnement d’images hors dataset altèrent la qualité des prédictions.
* Cellule 3 : reçoit comme arguments l’adresse du serveur d’API, l’index de l’image à traiter parmi les images du jeu de validation ainsi que le taux de blend désiré (0 étant le masque prédit et 1 l’image évaluée).

Le script de l’API Flask contient 4 routes :

Une route de contrôle à l’adresse ‘/’ renvoie un message de bienvenue et permet de vérifier que l’API est en fonctionnement

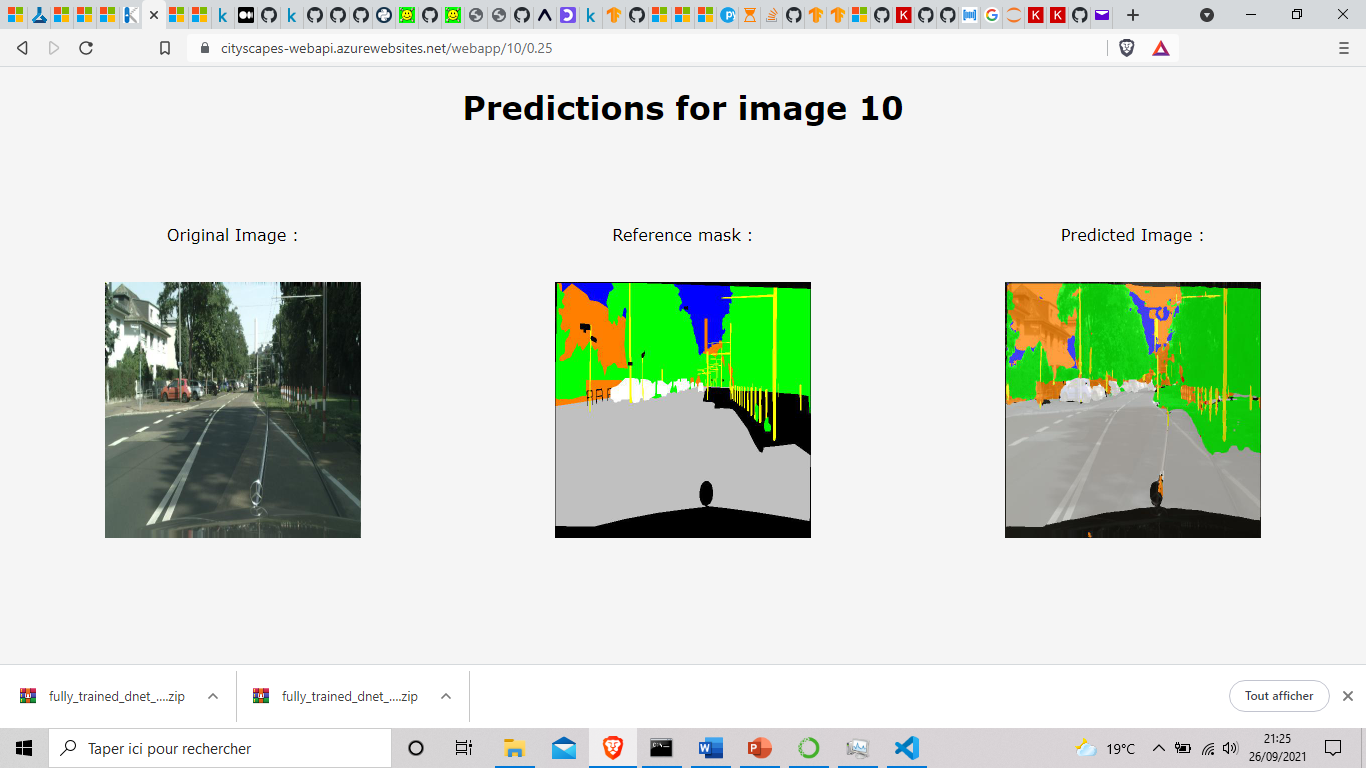
Une première version d’API à l’adresse ‘/theapi’ reçoit un index d’image à choisir dans la galerie. L’image et le masque idoine sont ouverts, la prédiction est faite et l’ensemble des données est envoyé au client. La réponse contient un message concernant le temps de traitement et la taille d’image, ainsi que l’image, le masque de référence et une image fusionnée de l’image et du masque prédit par le modèle.

Une seconde version d’API à l’adresse ‘/simpleapi’ reçoit une image, réalise la prédiction et renvoie l’image fusionnée de l’image et du masque prédit.

L’utilisation du code fourni permettra de visualiser les jeux d’images cityscapes sous la forme suivante :

Une webapp à l’adresse ‘/webapp/{numéro de l’image dans le jeu de validation}/{taux de blend image/Masque}’ Permet d’afficher l’image sélectionnée, le masque de référence idoine ainsi que l’image mixée du masque prédit et de l’image d’origine.

Tandis que la webapp permet l’affichage de l’ensemble sur une page web :



**Conclusion :**

* Quelques architectures ont été testées, avec une fonction de perte qui a permis d’obtenir un modèle aux performances décentes mais la quantité de modèles existant laisse à penser que d’autres types d’architectures pourraient bien fonctionner pour notre application.
* Outre l’architecture elle-même, il y a de nombreuses autres possibilités à tester en pré-traitement et en augmentation de données.
* Par exemple, il est possible que l’ajout d’une distorsion des proportions diminue la performance brute mais permette de rendre le système plus robuste aux changements de dimensions des images prises par le système d’acquisition.
* Il serait également intéressant de travailler sur l’équilibrage des classes lors de l’apprentissage – l’intégration de la loss focal-crossentropy à notre fonction d’optimisation pourrait être envisagée
* Les scripts de contrôle, d’entraînement et le générateur peuvent être réutilisés pour entraîner d’autres modèles moyennant l’obtention de crédits azure supplémentaires
* L’API est déployée et fonctionnelle mais peut être mise à jour par un simple push en CLI, il sera donc simple d’y apporter toute modification désirée, notamment s’il s’agit d’un changement de modèle.
* D’autres méthodes de détection on de segmentation d’instances pourraient être utilisées par le système de décision du véhicule : les développements en la matière sont nombreux et il y a peut-être matière à utiliser d’autres types de modèles pour se conformer au cahier des charges du système de décision.
* Un script client supplémentaire est proposé pour adapter les tailles d’images : L’image est paddée à la taille des images d’entraînement avant d’être remise à l’échelle afin de conserver les proportions des photos et donc des objets pour faciliter l’inférence.

**Bibliographie / Sources :**

1. [O.Ronneberger](https://arxiv.org/search/cs?searchtype=author&query=Ronneberger%2C+O), [P.Fischer](https://arxiv.org/search/cs?searchtype=author&query=Fischer%2C+P), [T.Brox](https://arxiv.org/search/cs?searchtype=author&query=Brox%2C+T) : « U-Net: Convolutional Networks for Biomedical Image Segmentation »
2. F.Yu, V.Koltun : « Multiscale context aggregation by dilated convolutions »
3. L-C.Chen, G.Papandreou, I.Kokkinos, K.Murphy, Alan L. Yuille : « DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs »
4. Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao : « Deep High-Resolution Representation Learning for Visual Recognition »
5. J.Long, E.Shelhamer, T.Darrell : « Fully Convolutional Networks for Semantic Segmentation »
6. F.Sultanaa , A.Sufiana, P.Dutta : « Evolution of Image Segmentation using Deep Convolutional Neural Network: A Survey »
7. Srihari Humbarwadi’s GitHub : partage de nombreux modèles de deep learning en implémentation Keras
8. NieCongChong’s GitHub : implémentation Keras du HRnet