Segmentation pour voiture autonome

Future Vision Transport

Contexte & objectif du projet

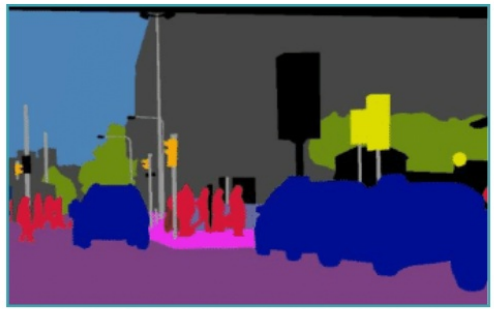
Future Vision Transport est une entreprise qui conçoit des systèmes embarqués de vision par ordinateur pour les **véhicules autonomes**.

Au sein de cette entreprise, nous développons un système composé de 4 parties principales :

1. acquisition des images en temps réel
2. traitement des images
3. **segmentation des images**
4. système de décision

Segmentation d'images

Image source



Segmentation sémantique

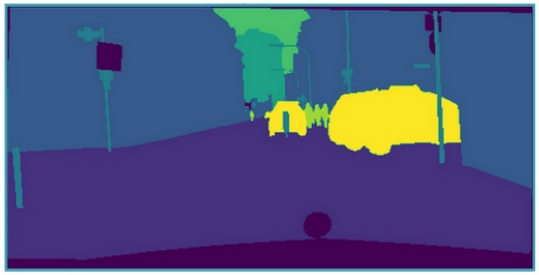
Segmentation d'instances

Segmentation panoptique

Segmentation Sémantique

X (image source)

Y (mask source)



Y' (mask prédit)

Jeu de données

Présentation du jeu de données

Le jeu de données Cityscapes contient une sélection de 25 000 photos de scènes urbaines collectées dans 50 villes, de jour, pendant 3 saisons et dans des conditions météo bonnes ou moyennes.

Chacune a un mask d'instances et de catégories.

- 20 000 ont des 'masks' approximatifs
- 5000 ont des 'masks' finement annotés

- train-set : 2975 photos & masks
- validation-set : 500 photos & masks
- test-set : non exploitable

(total : 3475 photos & masks)

Tous les médias sont en 2048 x 1024

Jeu de données

Préparatifs & pré-traitements

Conversion des catégories

Les masks fournis pour les catégories étant prévus pour 30 catégories il a fallu ajuster chaque fichier pour qu'il représente nos 8 catégories.

Réduction de la résolution des images

En raison de moyens limités, il a fallu pour accélérer la phase de recherche et entraîner le modèle final préparer un dataset en 256x128 et un en 512x256

Réorganisation des fichiers

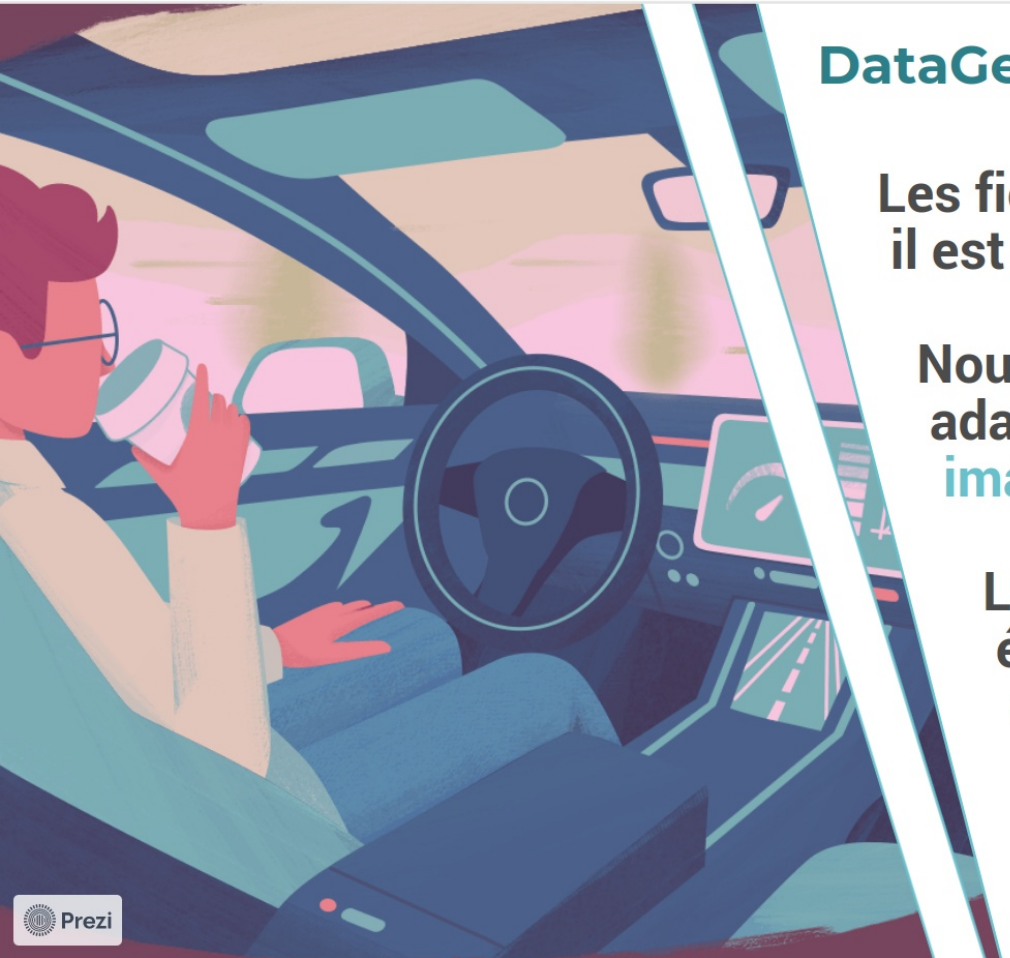
Pour faciliter le travail du DataGenerator les fichiers ont été renommés et rassemblés dans leur set (train ou validation) respectifs

Normalisation des valeurs

Pour aider les algorithmes les valeurs sont divisées par 255.0

Application des pré-processing des backbones utilisés

Recherche du modèle le plus adapté



DataGenerator

Les fichiers image étant gros par essence, il est difficile de tous les charger en mémoire.

Nous avons donc mis en place un data-generator adapté à notre jeu de données qui distribue les images par batch de taille paramétrable.

La classe a été conçue pour appliquer les éventuels pré-processing ou augmentations de données qui lui ont été transmis.

Recherche du modèle le plus adapté

Les métriques > classification

Pour un problème de classification,
les mesures de référence sont:

- **Accuracy** : taux de bonnes prédictions au total
- **Precision** : taux de vrai-positifs parmi les positifs
- **Recall** : capacité à détecter les vrai-positifs
- **Specificity** : capacité à éviter les faux-positifs

- **F1-Measure** : moyenne harmonique entre Precision et Recall -> 2*(P*R)/(P+R)

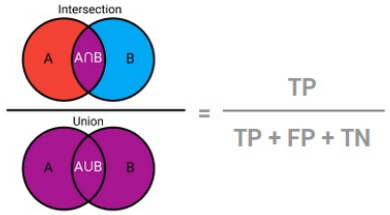
Recherche du modèle le plus adapté

Les métriques > segmentation sémantique

Les habituelles métriques de classification sont assez sensibles aux déséquilibre des classes présentes dans le jeu de données...

IoU(Jaccard index)

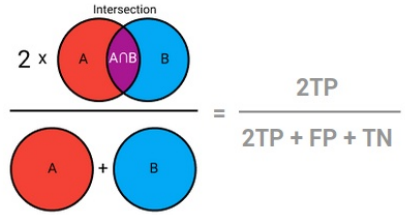
[0-1] évaluation la plus défaforable



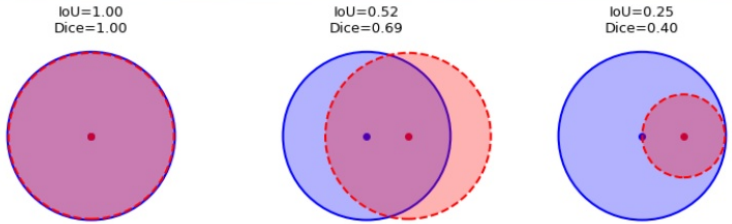
$$\frac{A \cap B}{A \cup B} = \frac{TP}{TP + FP + TN}$$

Sørensen–Dice (F1)

[0-1] évaluation moyenne



$$\frac{2 \times |A \cap B|}{|A| + |B|} = \frac{2TP}{2TP + FP + TN}$$



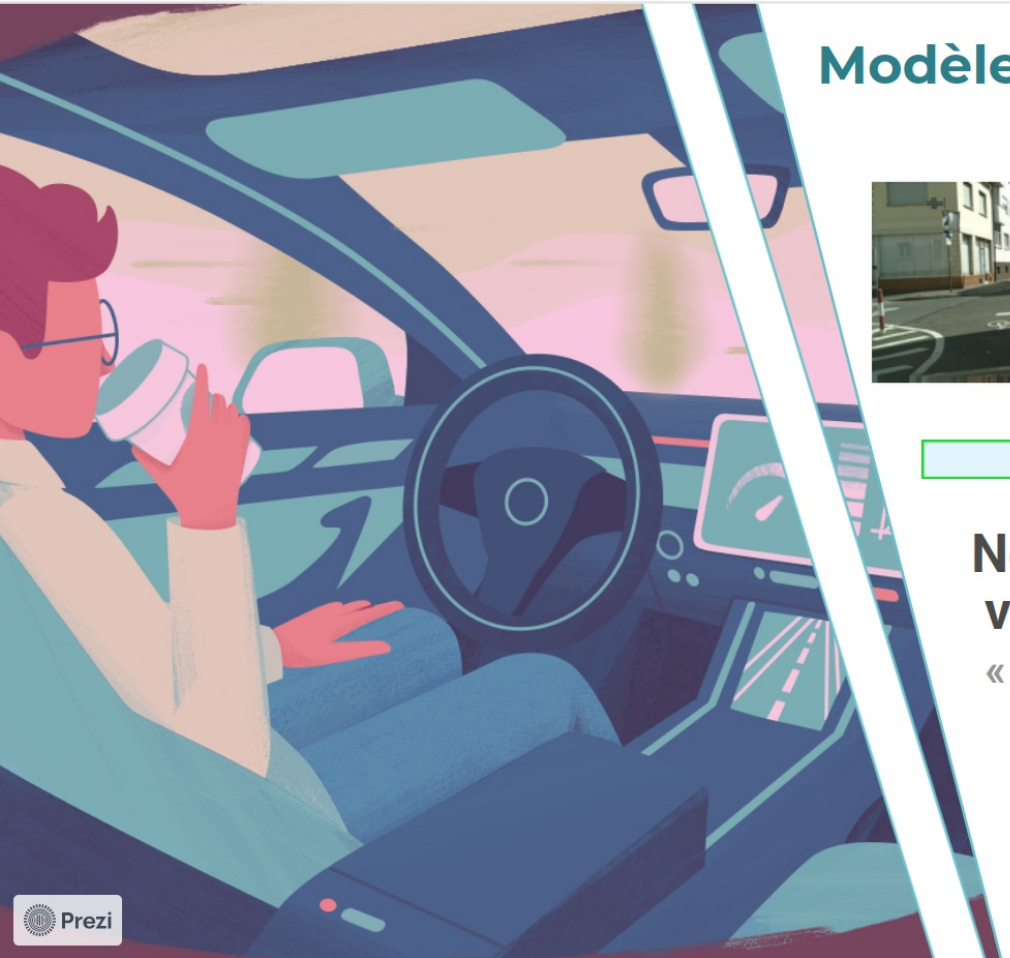
Recherche du modèle le plus adapté

Hyper-paramètres & callbacks

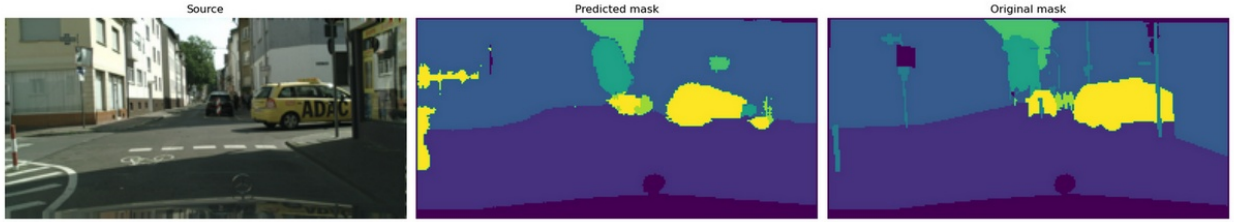
Pour optimiser le temps de calcul consacré à chaque modèle, nous avons mis en place plusieurs callbacks Keras.

- **EarlyStopping**
  (100 epochs max, patience=9, basé sur val_loss)
  pour arrêter l'entraînement quand le modèle ne progresse plus
- **ReduceLROnPlateau**
  (Adam LR=0.0005, patience=9, basé sur val_loss)
  pour réduire le learning-rate en fin d'apprentissage
- **ModelCheckpoint**
  (basé sur val_iou_score)
  pour enregistrer la version du modèle la plus performante
- **Tensorboard**
  pour suivre l'évolution du modèle en cours
  et le comparer aux autres ensuite...

Recherche du modèle le plus adapté



Modèle naïf



	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net_baseline	0.552171	0.667224	517.575655	1.851263	0.895139

Nous avons utilisé un U-Net sans backbone venant d'un exemple de la lib. Keras initulé

« Image segmentation with a U-Net-like architecture »

C'est un modèle fonctionnel, permettant une itération rapide et compatible avec les contraintes techniques du projet.

Recherche du modèle le plus adapté


```python
transform1 = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.Rotate(15, p=0.5),
    A.RandomBrightnessContrast(p=0.2),
])
```

```python
transform2 = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.OneOf([
        A.RandomBrightnessContrast(
            brightness_limit=[-0.2, 0.2],
            contrast_limit=0.2,
            p=0.5,
        ),
        A.RandomGamma(p=0.5),
    ],p=0.8,),
])
```

```python
transformAuto = A.load("autoalbument/policy_28.json")
```

Augmentation de données

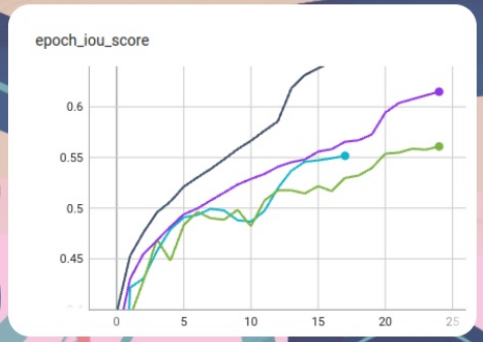
L'étape suivante consistait à renforcer notre modèle en diversifiant les exemples utilisés pour l'entraîner, avec l'espoir de mieux généraliser et donc d'en améliorer les performances.

Nous avons donc utilisé Albumentation et AutoAlbument pour créer trois ensembles de transformations pouvant potentiellement s'appliquer sur chaque image.

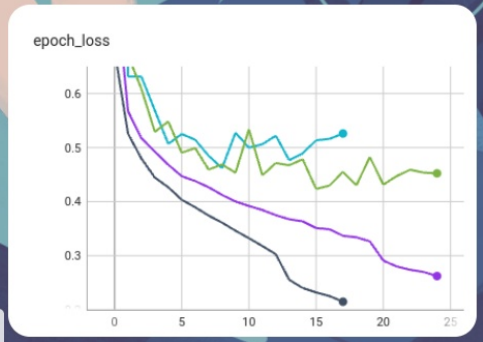
- par intuition
- par recommandation
- par recherche automatique

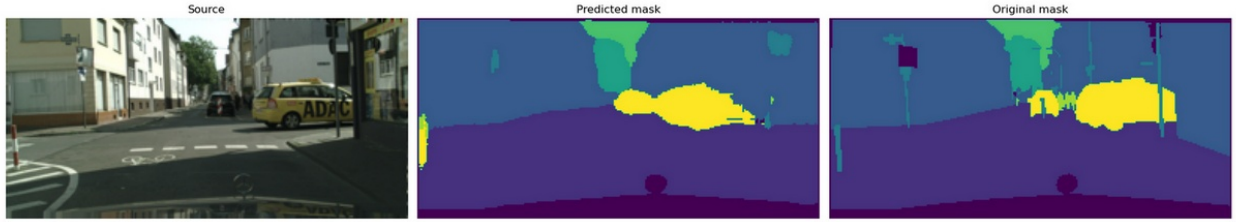
Recherche du modèle le plus adapté

Augmentation de données > résultats



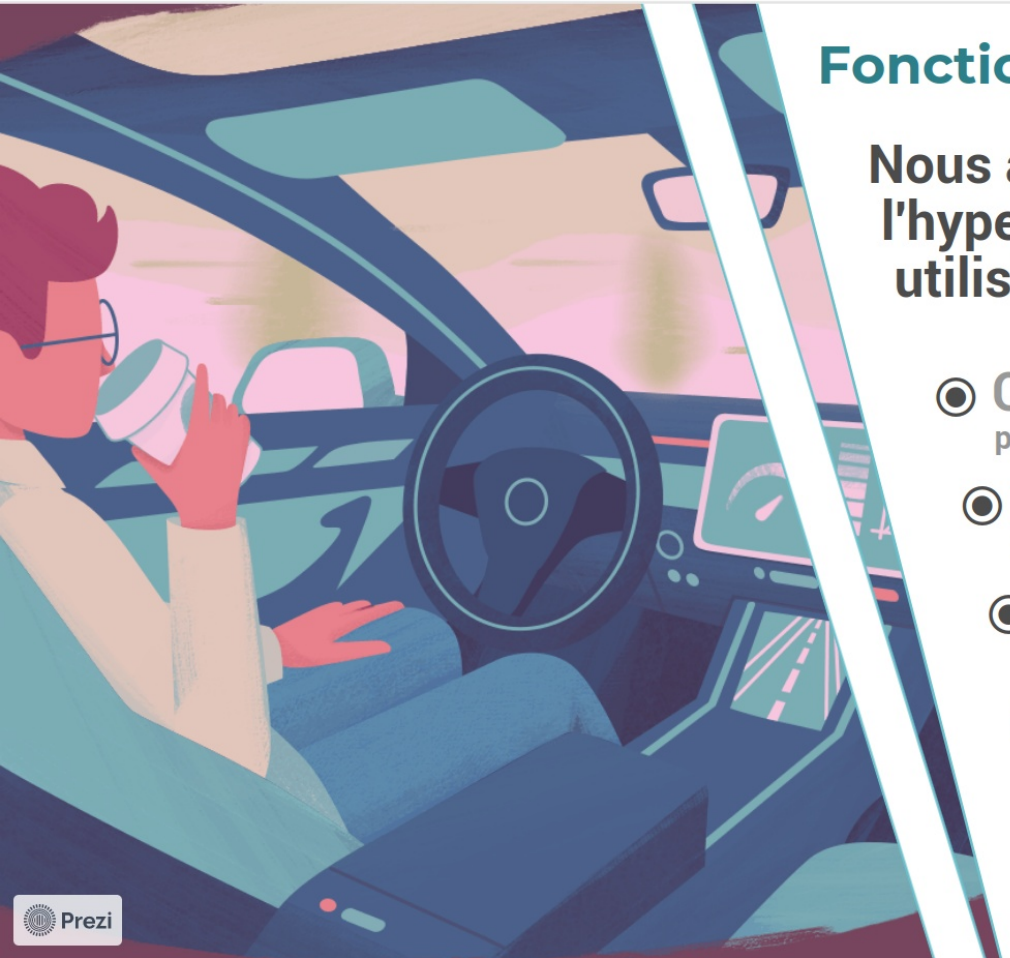
Run	IoU	Dice	Loss
U-Net_baseline/validation	0.5516	0.6664	0.5261
U-Net_baseline/train	0.6523	0.753	0.2148
U-Net_baseline_with_data_augmentation_2/validation	0.5608	0.6763	0.4524
U-Net_baseline_with_data_augmentation_2/train	0.6149	0.7219	0.2623





	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net_baseline	0.552171	0.667224	517.575655	1.851263	0.895139
U-Net_baseline_with_data_augmentation_1	0.564558	0.680571	1029.694759	2.369722	0.423762
U-Net_baseline_with_data_augmentation_2	0.559365	0.674603	714.297488	2.345822	0.924520
U-Net_baseline_with_data_augmentation_3	0.117844	0.197346	284.685457	2.193978	0.414083

Recherche du modèle le plus adapté



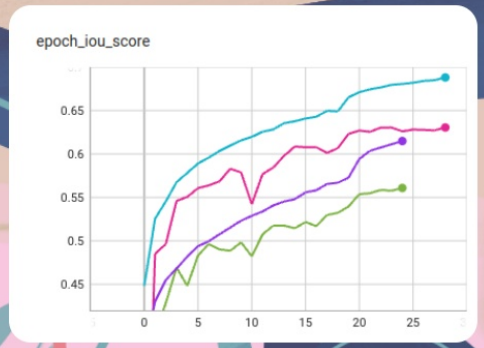
Fonction de perte pour segmentation

Nous avons ensuite cherché à optimiser l'hyper-paramètre qu'est la fonction de perte utilisée pour entraîner les modèles.

- **Categorical Cross Entropy**
  parfaitement adaptée à la classification multi-classes
- **DiceLoss**
  fonction de perte habituellement associée à l'IoU (car proche et dérivable)
- **DiceLoss pondéré**
  variante permettant d'ajuster le poids des classes
- **FocalLoss** (gamma 0.5 et 1)
  version améliorée de la CE qui tente de pallier le déséquilibre des classes
- **Mix CCE + DiceLoss** (1/1 et 0.5/1)
  fonction de perte personnalisée basée sur les 2 meilleurs fonctions de pertes identifiées au préalable

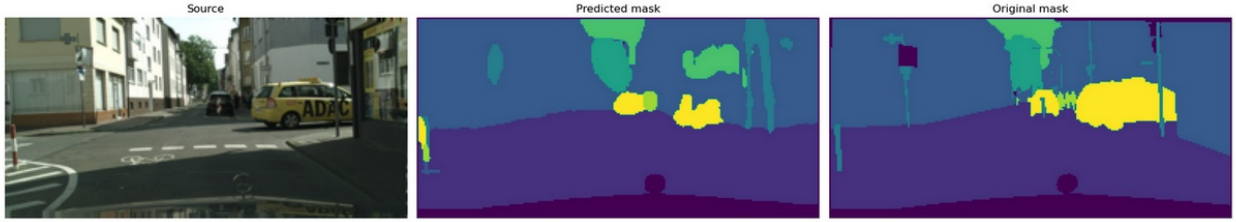
Recherche du modèle le plus adapté

Future Vision Transport



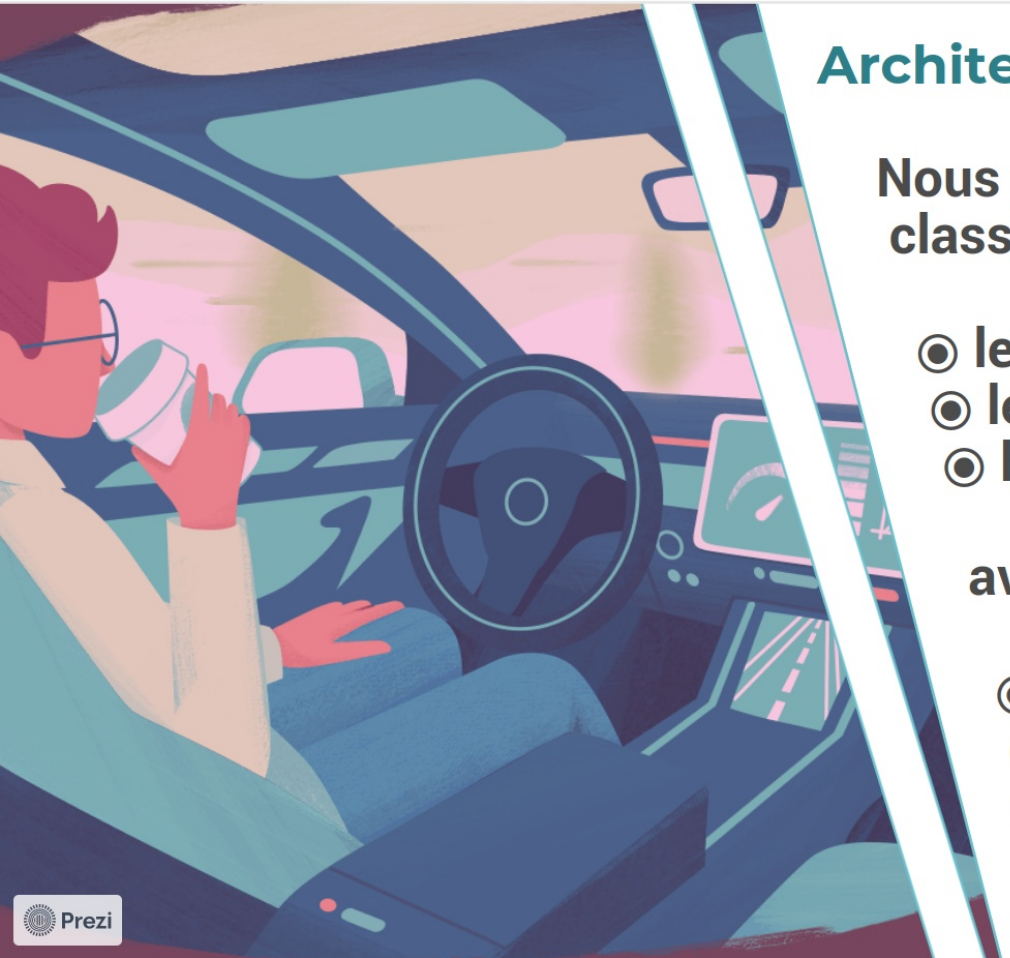
Run	IoU	Dice	Loss
U-Net_baseline_with_data_augmentation_2_diceLoss/train	0.6881	0.7944	0.2056
U-Net_baseline_with_data_augmentation_2_diceLoss/validation	0.6305	0.7429	0.2571
U-Net_baseline_with_data_augmentation_2/validation	0.5608	0.6763	0.4524
U-Net_baseline_with_data_augmentation_2/train	0.6149	0.7219	0.2623

Fonction de perte > résultats



	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net_baseline_with_data_augmentation_2	0.559365	0.674603	714.297488	2.345822	0.924520
U-Net_baseline_with_data_augmentation_2_diceLoss	0.631681	0.743959	843.582155	2.138717	0.903396
U-Net_baseline_with_data_augmentation_2_diceLossWeighted	0.612228	0.728071	697.217847	2.189904	0.415433
U-Net_baseline_with_data_augmentation_2_focalLoss_0.25_0.5	0.486711	0.612405	326.424069	2.646039	0.425659
U-Net_baseline_with_data_augmentation_2_focalLoss_0.25_1.0	0.454141	0.585703	318.810731	2.206589	0.418523
U-Net_baseline_with_data_augmentation_2_cceDiceLoss_50_50	0.622789	0.738499	899.124581	2.216438	0.415926
U-Net_baseline_with_data_augmentation_2_cceDiceLoss_0.5_1.0	0.628407	0.742861	995.207425	2.260605	0.444320

Recherche du modèle le plus adapté



Architectures pour segmentation sémantique

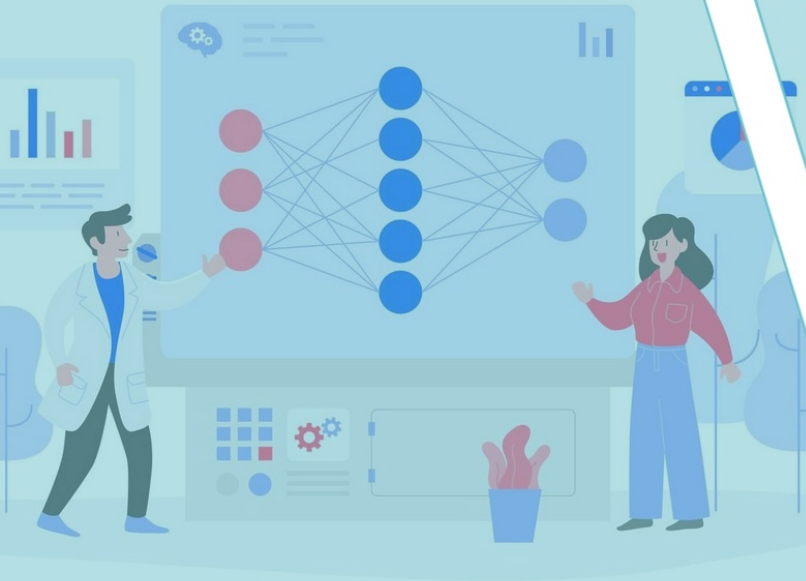
Nous avons choisi d'évaluer des architectures classiques du domaine comme :

- le U-Net,
- le LinkNet,
- le FPN (Feature Pyramid Networks)

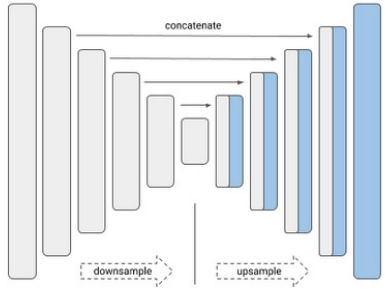
avec les backbones suivants :

- VGG19,
- ResNet152,
- ResNeXt101,
- EfficientNetB7.

Recherche du modèle le plus adapté

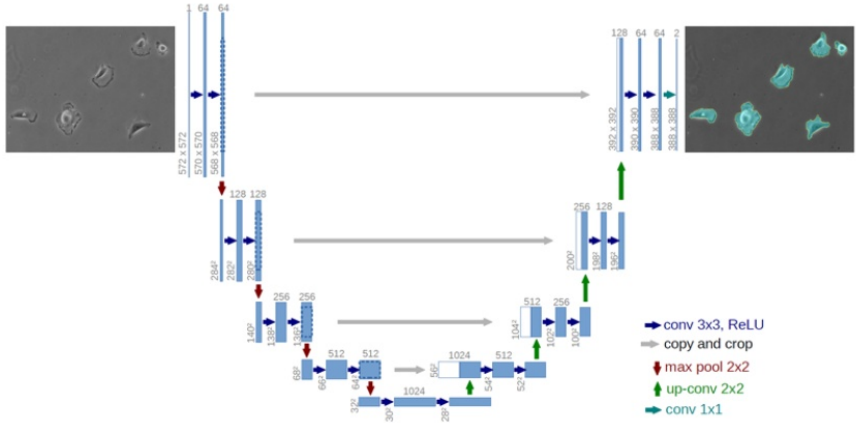


Architecture - U-Net

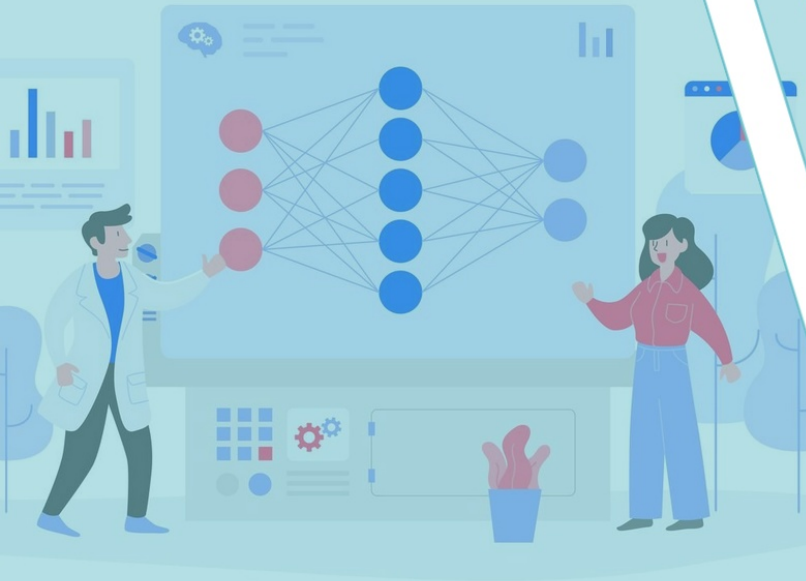


C'est un CNN qui a la particularité d'être entièrement convolutionnel.

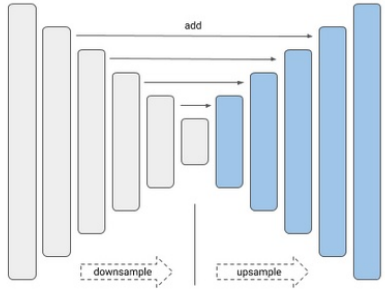
Son architecture en U connectant l'encoder et le décodeur crée un mécanisme d'attention locale et globale à même de restituer précisément la position de chaque pixel de l'image d'origine avec la classification.



Recherche du modèle le plus adapté

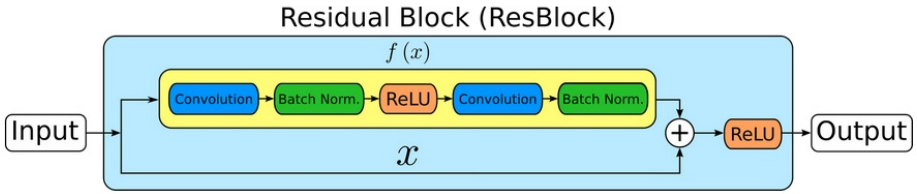


Architecture - LinkNet



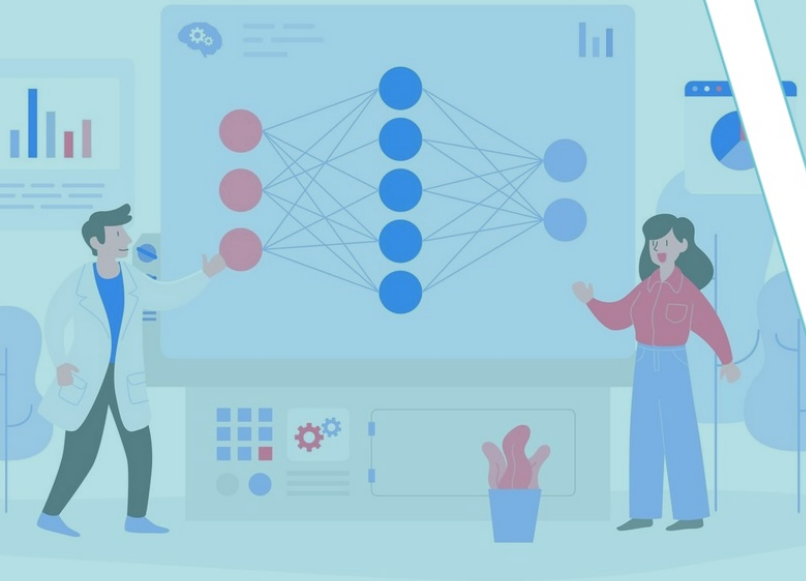
C'est une variante du U-Net qui se différencie sur deux points :

- il remplace les blocks de convolution ordinaire de l'U-Net par des blocks résiduel (res-block)



- la transmission d'informations des étages de l'encodeur à ceux du décodeur ne se fait plus par concaténation, mais par addition.

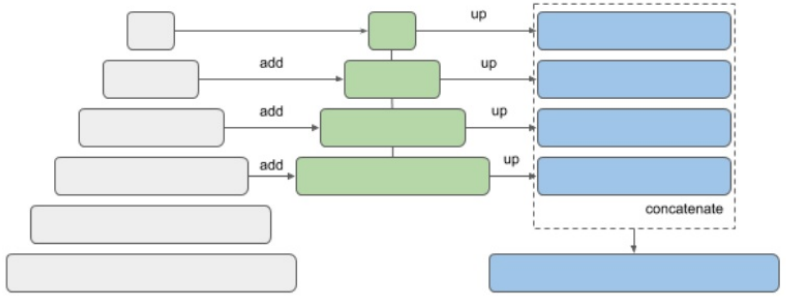
Recherche du modèle le plus adapté



Architecture - FPN (Feature Pyramid Networks)

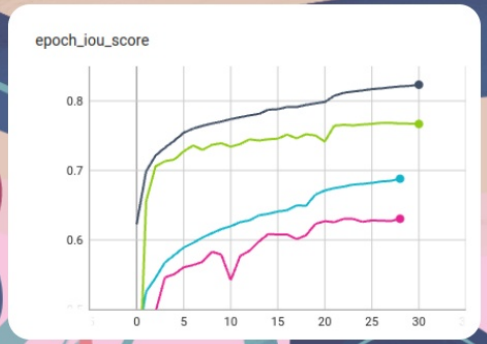
C'est une implémentation alternative des concepts sous-jacent des U-Nets

L'une des principales différences étant que les U-nets produisent en sortie une image de même taille que celle fournie en entrée, alors que les FPN produisent une image de sortie pour chaque étage de la pyramide qui sont ensuite concaténées.

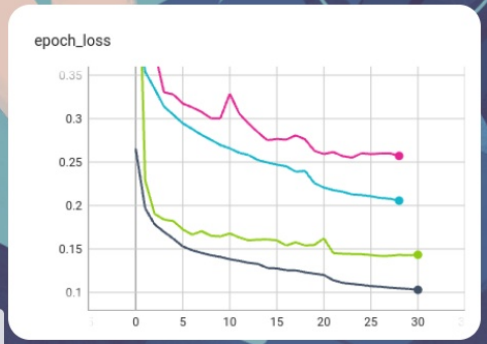


Recherche du modèle le plus adapté

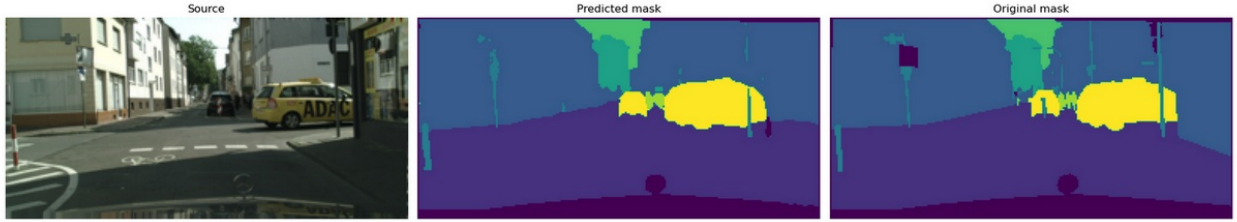
Future Vision Transport



Run	IoU	Dice	Loss
FPN-efficientnetb7_with_data_augmentation_2_diceLoss/validation	0.7668	0.8567	0.1433
FPN-efficientnetb7_with_data_augmentation_2_diceLoss/train	0.8234	0.8971	0.1029
U-Net_baseline_with_data_augmentation_2_diceLoss/validation	0.6305	0.7429	0.2056
U-Net_baseline_with_data_augmentation_2_diceLoss/train	0.6881	0.7944	0.2571



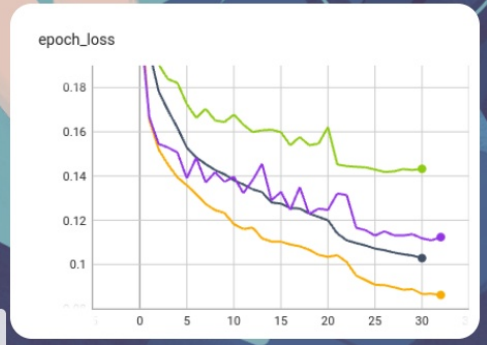
Architectures > résultats



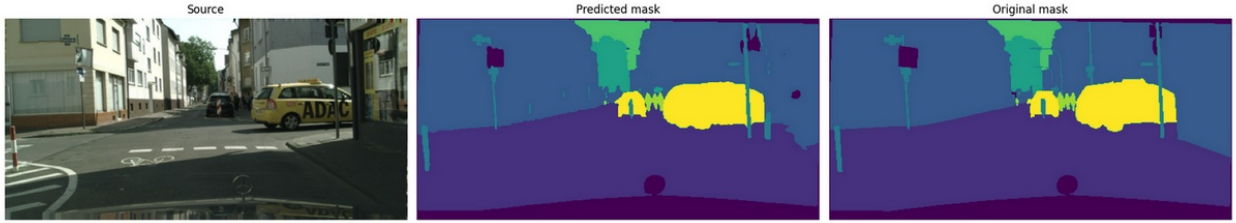
	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net_baseline_with_data_augmentation_2_diceLoss	0.631681	0.743959	843.582155	2.138717	0.903396
U-Net-resetnet152_with_data_augmentation_2_diceLoss	0.722085	0.822108	3538.612212	6.312516	3.414692
U-Net-vgg19_with_data_augmentation_2_diceLoss	0.734644	0.832938	1932.282903	3.101655	0.048437
U-Net-resnext101_with_data_augmentation_2_diceLoss	0.720611	0.820914	3738.743413	5.380407	0.130063
FPN-resnet152_with_data_augmentation_2_diceLoss	0.738715	0.834818	4316.767552	5.874191	0.069000
FPN-efficientnetb7_with_data_augmentation_2_diceLoss	0.768454	0.858004	5930.868126	7.299468	0.082367
FPN-vgg19_with_data_augmentation_2_diceLoss	0.744457	0.839688	2601.794258	5.406630	0.721739
U-Net-custom_with_data_augmentation_2_diceLoss	0.541770	0.657728	820.391873	1.998511	0.646094
LinkNet-resnet152_with_data_augmentation_2_diceLoss	0.707575	0.809275	2433.885920	6.768224	3.061178
LinkNet-efficientnetb7_with_data_augmentation_2_diceLoss	0.751172	0.844442	4191.200948	10.014075	5.479

Recherche du modèle le plus adapté

Run	IoU	Dice	Loss
FPN-efficientnetb7_with_data_augmentation_2_diceLoss/train	0.8234	0.8971	0.1029
FPN-efficientnetb7_with_data_augmentation_2_diceLoss/validation	0.7668	0.8567	0.1433
FPN-efficientnetb7_with_data_augmentation_2_diceLoss_512x256/train	0.8502	0.9138	0.08619
FPN-efficientnetb7_with_data_augmentation_2_diceLoss_512x256/validation	0.8121	0.8876	0.1124



Augmentation de la résolution



	IoU	Dice	training_time	inference_time_500	inference_time_1
FPN-efficientnetb7_with_data_augmentation_2_diceLoss	0.768454	0.858004	5930.868126	7.299468	0.082367
FPN-efficientnetb7_with_data_augmentation_2_diceLoss_512x256	0.811932	0.887532	22570.196906	25.308100	0.258906

Nous avons donc le modèle final suivant:

- Architecture : FPN
- Backbone : efficientnetB7
- Avec des augmentations de données
- Fonction de perte: DiceLoss
- Résolution: 512 x 256

Recherche du modèle le plus adapté

Évolution des modèles

Modèle de référence

Modèle déployé

	IoU	Dice	training_time	inference_time_500	inference_time_1
U-Net_baseline	0.552171	0.667224	517.575655	1.851263	0.895139
U-Net_baseline_with_data_augmentation_2	0.559365	0.674603	714.297488	2.345822	0.924520
U-Net_baseline_with_data_augmentation_2_diceLoss	0.631681	0.743959	843.582155	2.138717	0.903396
FPN-efficientnetb7_with_data_augmentation_2_diceLoss	0.768454	0.858004	5930.868126	7.299468	0.082367
FPN-efficientnetb7_with_data_augmentation_2_diceLoss_512x256	0.811932	0.887532	22570.196906	25.308100	0.258906

Segmentation sémantique

Déploiement du modèle > démo

Flask
web development,
one drop at a time

docker

heroku

Paperspace

Axes d'amélioration

- **Essayer d'augmenter la résolution du train-set**
  (le passage de 256x128 à 512x256 à boosté le modèle de 4.3%)
- **Essayer d'autres architectures**
  (en particulier les archi. SotA comme le **SegFormer** ou le **HRNetV2 OCR+PSA**)
- **Essayer d'autres backbones**
  (les versions plus légères des backbones choisis pourraient être plus rapides)
- **Utiliser des outils d'optimisation**
  (OpenVino / TensorRT pour la quantization, ONNX ou même les DeciNets)
- **Vérifier et améliorer les mesures de temps**
  (la version actuelle est imparfaite, mais c'est un élément important)
- **Améliorer le jeu de données**
  (4 saisons, toutes conditions météo, nuit...)

Merci de m'avoir écouté, évalué et conseillé.

Future Vision Transport