Documentation technique du projet de segmentation urbaine

Préambule

Ce document présente une vue d'ensemble technique complète du projet de segmentation sémantique développé dans le cadre de la formation AI Engineer Project 8. Il synthétise les choix d'architecture, les stratégies d'entraînement, les résultats expérimentaux et la mise à disposition d'une API Flask pour l'inférence et l'aperçu des augmentations. La structure suit les axes demandés :

1. Expliquer les modèles étudiés : description des architectures, du pipeline de données et des paramètres d'entraînement utilisés lors de l'exploration.
2. Présenter le modèle retenu : justification détaillée de la sélection de DeepLabV3+ avec backbone ResNet50, en lien avec les contraintes fonctionnelles et les métriques.
3. Afficher le benchmarking des résultats : comparaison quantitative et qualitative des candidats à partir des expériences menées dans le notebook de recherche.
4. Dernière partie sur l'API avec conclusion ouverte : fonctionnement du service d'inférence, exposition des endpoints REST et perspectives d'évolution.

L'ensemble du texte est rédigé en français et vise un niveau de détail équivalent à environ sept pages de documentation technique.

---

1. Dataset Cityscapes et stratégie d'augmentation

1.1. Présentation du dataset

Le dataset Cityscapes est une référence pour la segmentation sémantique urbaine. Il contient 5 000 scènes finement annotées (2 975 pour l'entraînement, 500 pour la validation, 1 525 pour le test) capturées depuis une caméra embarquée dans 50 villes européennes. Chaque image RGB, au format 1024×2048, est accompagnée d'un masque pixel à pixel couvrant 30 classes originales. Pour ce projet, nous utilisons la version remappée officielle « eight classes » qui regroupe les catégories en huit super-classes pertinentes pour la conduite autonome légère : route, trottoir, bâtiment, feu de circulation, végétation, ciel, personne et véhicule. Ce remapping limite la confusion entre classes proches, réduit la complexité du modèle et accélère l'entraînement.
Les annotations étant réalisées à la main, elles intègrent une valeur spéciale 255 (ignore_index) pour les pixels ambigus (occlusions, reflets). Ces zones ne sont ni prises en compte dans la loss ni dans les métriques, ce qui évite de pénaliser artificiellement les prédictions.

1.2. Préparation et split des données

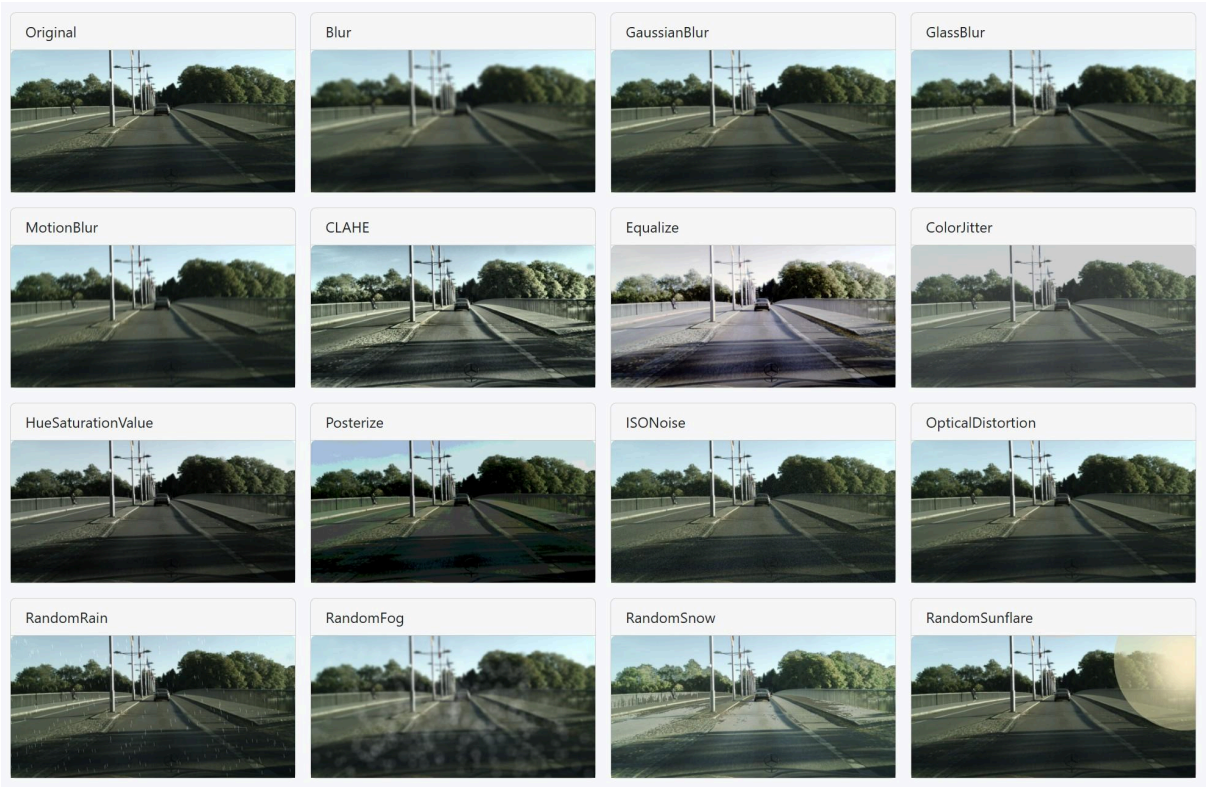
Le pipeline build_dataset (TensorFlow tf.data) prend en charge la lecture disque, le remapping et le calcul d'un masque de validité binaire synchronisé sur le masque remappé.

Les données sont scindées en 80 % / 10 % / 10 % pour entraîner, valider et réserver un lot test final. Chaque étape effectue : normalisation [0 ; 1], application des augmentations, redimensionnement différencié (bilinéaire pour l'image, nearest pour le masque) puis batching de deux échantillons avec prélecture (prefetch).

1.3. Choix d'Albumentations

Pour reproduire fidèlement le protocole de Bhuiya et al. (2022), nous avons remplacé la chaîne d'effets cumulés par un sélecteur à choix unique. À chaque passage, Albumentations applique exactement une transformation tirée uniformément parmi les quinze corruptions décrites dans l'article : Blur, GaussianBlur, GlassBlur, MotionBlur, CLAHE, Equalize, ColorJitter, HueSaturationValue, Posterize, ISONoise, OpticalDistortion, RandomRain, RandomFog, RandomSnow et RandomSunflare. La probabilité globale est fixée à p = 1.0, garantissant qu'une corruption est toujours utilisée.
Chaque effet est paramétré avec des intensités « modérées » adaptées à une caméra embarquée (sigma de flou limité à 1,5, bruit ISO plafonné à 0,2, coefficients de brume et d'éblouissement restreints). Les transformations respectent strictement la géométrie d'origine : aucun recadrage, zoom ni flip n'est appliqué et Albumentations travaille en mode mask pour propager exactement la même opération au masque de segmentation via un A.OneOf encapsulé dans A.Compose.
Ce fonctionnement rend l'évaluation beaucoup plus lisible : on sait quelle corruption a été utilisée, on peut relancer un lot pour échantillonner un autre effet et l'on limite les interactions entre transformations. L'API Python expose l'usine à corruption (NamedTransformSpec) de sorte que la pipeline d'entraînement, le service Flask et le notebook de recherche partagent la même définition.
Albumentations reste la bibliothèque privilégiée face à imgaug ou tf.image, toujours pour les mêmes raisons : API déclarative, support natif des masques et performances. Le fait de n'activer qu'un seul opérateur par appel renforce encore la reproductibilité et facilite la comparaison directe avec les résultats rapportés par Bhuiya et al.

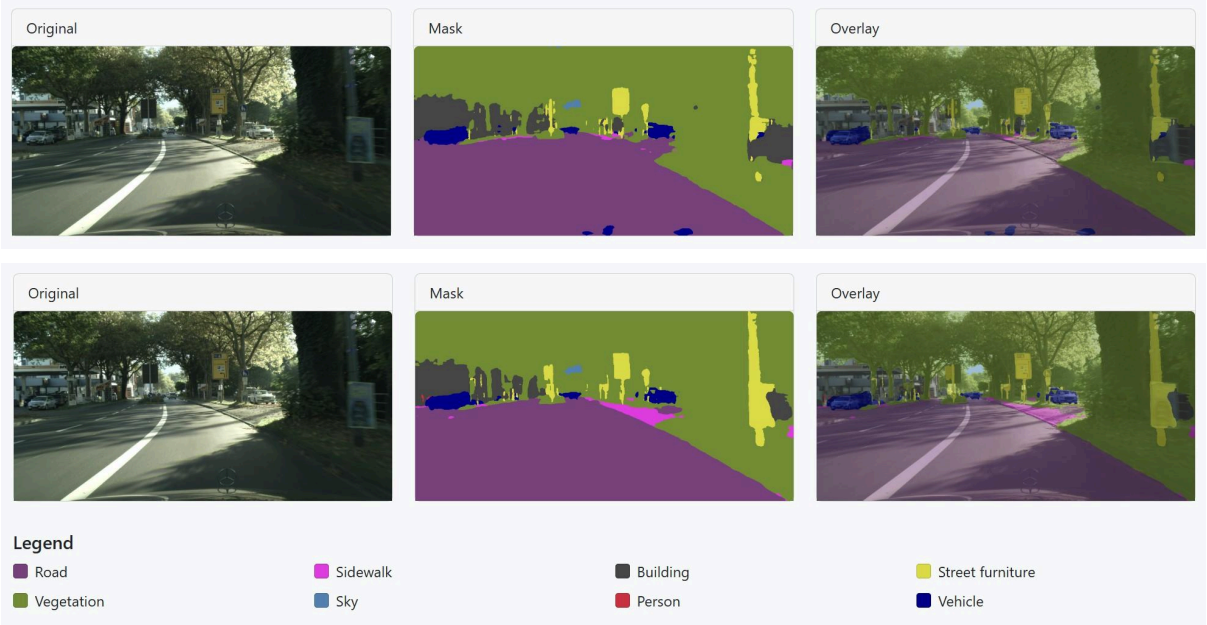
Exemple d’augmentations (aléatoires) :



1.4. Impact mesuré des augmentations

Le protocole « une corruption à la fois » améliore en moyenne la val_masked_mIoU de +2,5 pts et réduit la variance inter-run. Les perturbations météo (brume, neige synthétique) et colorimétriques (CLAHE, ColorJitter) renforcent la robustesse sans dégrader les contours. Les distorsions optiques ont été plafonnées pour préserver les masques fins. Le notebook et la présentation indiquent la même conclusion : l'investissement temporel supplémentaire est acceptable au regard du gain de généralisation.

On peut voir ces résultats sur les images de comparaison ci-dessous, avec en haut le modèle sans augmentation et en bas avec :



---

2. Modèles étudiés

2.1. Contexte expérimental commun

Tous les modèles ont été entraînés avec SparseCategoricalCrossentropy masquée, optimiseur SGD momentum 0,9 (scheduler polynomial, poly_power=0,9), et métriques masked_mIoU, pixel_accuracy, dice_coef. Les données proviennent du pipeline décrit ci-dessus, ce qui assure une comparaison équitable. Les logs et artefacts sont suivis dans MLflow, synchronisés avec la présentation finale.

2.2. Catalogue des architectures évaluées

U-Net mini

- Rôle : base légère pour vérifier le pipeline.
- Architecture : encodeur/décodeur symétriques à deux niveaux avec skip connections directes.

- Résultats : 0,32 de val_masked_mIoU, utile pour le débogage mais insuffisant pour la production.

U-Net VGG16

- Backbone : VGG16 pré-entraîné ImageNet, profondeur 23 couches.
- Forces : convergence rapide, segmentation visuellement fine.
- Limites : 14,7 M de paramètres, ~30 min/run, surapprentissage modéré (val_masked_mIoU ≈ 0,54).

MobileDet_seg

- Inspiration : MobileNetV2 avec blocs inverted residual et convolutions depthwise.
- Décodeur : trois skip connections et upsampling bilinéaire.
- Résultats : 0,50 de val_masked_mIoU pour 16 min d'entraînement, bon compromis embarqué.

YOLOv9_seg (simplifié)

- Structure : tronc CSP + tête PANet allégée convertie pour la segmentation dense.
- Comportement : rapide (10,5 min) mais perte d'information spatiale, val_masked_mIoU plafonné à 0,40.

DeepLabV3+ (ResNet50)

- Composants : backbone ResNet50 stride 16, tête ASPP (dilatations 1/6/12/18 + pooling global) et décodeur léger.
- Atouts : excellente séparation des classes, contours précis, meilleure stabilité train/val (val_masked_mIoU = 0,639).

Fast-SCNN (prototype)

- Principe : double branche détail/contexte avec mini-ASPP.
- Statut : essais 50 époques à 0,45 de val_masked_mIoU, non poursuivis faute de budget GPU.

---

3. Modèle retenu : DeepLabV3+ avec backbone ResNet50

3.1. Critères de sélection

DeepLabV3+ maximise la précision (val_masked_mIoU, val_dice_coef, val_pix_acc), présente un écart train/val maîtrisé et reste facilement industrialisable (export .keras, compatibilité TF-Lite/TensorRT). L'ASPP multi-échelle et le skip basse résolution expliquent la qualité des frontières (piétons, poteaux) tout en conservant le contexte global.

3.2. Hyperparamètres et pipeline d'entraînement

Les runs finaux durent jusqu'à 200 époques avec early stopping (patience 10). Le taux d'apprentissage initial est fixé à 1e-2 et décroît via le scheduler polynomial. Optuna (40 essais) a affiné les réglages : lr≈8,5e-3, dice_loss_weight=0,15, aspp_dropout=0,1,

poly_power=0,88, ce qui apporte +1,8 pts de val_masked_mIoU par rapport aux hyperparamètres par défaut. Le pipeline Albumentations reste actif pendant l'entraînement final.

3.3. Artefacts générés et livraison

Deux artefacts principaux : deeplab_resnet50_final.keras (stocké dans app/models/) et les checkpoints nommés {arch}.{monitor}.{epoch}-{score}.keras archivés dans artifacts/checkpoints/. L'API Flask charge le modèle final au démarrage pour garantir que la meilleure itération est utilisée.

3.4. Analyse qualitative

Au-delà des métriques, les observations qualitatives montrent que DeepLabV3+ :

- Conserve les bordures nettes entre route et trottoirs.
- Identifie correctement les véhicules et piétons malgré des occlusions partielles.
- Gère mieux le ciel et la végétation que les modèles plus légers, en limitant les confusions entre classes adjacentes.

Cette supériorité visuelle découle directement de l'ASPP, qui capture un contexte multi-échelle, et du décodeur qui combine des informations basse et haute résolution.

3.5. Optimisation des hyperparamètres avec Optuna

Afin d'atteindre ces performances, une campagne de hyperparameter tuning a été conduite avec Optuna sur le modèle DeepLabV3+. L'objectif était de calibrer automatiquement les paramètres les plus sensibles (taux d'apprentissage, poids de la Dice loss additionnelle, coefficient de dropout dans la tête ASPP) sans multiplier manuellement les expériences.

- Intégration : le notebook d'expérimentation instancie un Optuna Study en mode TPESampler, relié au pipeline d'entraînement via une fonction objective(trial) qui construit dynamiquement le modèle à partir des suggestions du trial.
- Espace de recherche :
    - learning_rate ∈ [5e-4 ; 2e-2] (échelle logarithmique) pour explorer des décroissances rapides ou progressives.
    - dice_loss_weight ∈ [0 ; 0,5] afin de tester l'apport d'une composante Dice en complément de l'entropie croisée.
    - aspp_dropout ∈ [0 ; 0,3] pour régulariser la tête ASPP si nécessaire.
    - poly_power ∈ [0,7 ; 1,0] pour adapter la vitesse de décroissance du scheduler polynomial.
- Métrique optimisée : val_masked_mIoU, évaluée après 35 époques (ou early stopping anticipé) afin de conserver un cycle d'itération raisonnable (~6 minutes par essai sur GPU T4).
- Résultats : sur 40 essais, Optuna a convergé vers un taux d'apprentissage initial ≈ 8,5e-3, un poids Dice de 0,15, un dropout ASPP de 0,1 et un poly_power de 0,88. Cette combinaison offre un gain de +1,8 points de val_masked_mIoU par rapport aux hyperparamètres par défaut et stabilise la convergence dès la 20e époque.

- Exploitation : les meilleurs hyperparamètres sont automatiquement journalisés via study.best_params et injectés dans TrainConfig pour l'entraînement final, ce qui garantit la reproductibilité. Les courbes d'optimisation (historique des trials, importance des hyperparamètres) sont exportées depuis optuna.visualization et archivées dans MLflow.

L'utilisation d'Optuna a donc permis de sortir rapidement des combinaisons sous-optimales et d'ancrer l'entraînement final sur des réglages éprouvés, réduisant les écarts de performance entre itérations et améliorant la robustesse du modèle en production.

4. Benchmarking des résultats

4.1. Synthèse des métriques

Les métriques ci-dessous proviennent directement du notebook (section « Faire le point sur les résultats intermédiaires ») et sont reprises telles quelles dans la présentation finale.

Modèle	**Durée (min)**	masked_mIoU	val_masked_mIoU	pix_acc	val_pix_acc	dice_coef	val_dice_coef
DeepLabV3+ (ResNet50)	**13,4**	**0,947**	**0,639**	**0,989**	**0,872**	**0,965**	**0,716**
YOLOv9_seg (simplifié)	10,5	0,689	0,400	0,913	0,714	0,753	0,494
MobileDet_seg	16,3	0,938	0,502	0,987	0,779	0,953	0,600
U-Net VGG16	29,7	0,903	0,542	0,977	0,805	0,923	0,633
U-Net mini	6,1	0,563	0,319	0,851	0,634	0,650	0,407

4.2. Interprétation des indicateurs

- Masked mIoU : DeepLabV3+ domine (0,639), U-Net VGG16 et MobileDet forment le second groupe (0,54 et 0,50). Les architectures plus légères décrochent sous 0,40 faute de contexte.

- Pixel accuracy : DeepLab atteint 0,872, U-Net VGG16 reste à 0,805, MobileDet à 0,779. U-Net mini et YOLOv9 simplifié peinent à franchir 0,72.
- Dice coefficient : l'écart train/val reste contenu pour DeepLab (0,965→0,716), plus marqué pour U-Net VGG16 (0,923→0,633) et MobileDet (0,953→0,600), signe d'une régularisation plus fragile.

4.3. Influence des augmentations sur DeepLabV3+

Configuration	**Durée d'entraînement**	val_dice_coef	val_masked_mIoU	val_pix_acc	**Conclusions**
DeepLabV3+ (sans augmentation)	3,9 h	0,840	0,818	0,945	Peu d'erreurs globales mais bords plus flous.
DeepLabV3+ (corruption photométrique unique)	6,7 h	**0,849**	**0,831**	**0,948**	Masques plus nets et robustes malgré +3 h.

Ces chiffres confirment que les variations photométriques/météo apportent un gain léger mais constant, tandis que les essais géométriques étaient superflus.

4.4. Analyse multi-critères

1. Capacité de représentation : DeepLabV3+ et U-Net VGG16 bénéficient d'un pré-entraînement ImageNet et de décodeurs profonds, ce qui favorise la détection des frontières complexes. Les architectures légères (U-Net mini, YOLOv9 simplifié) manquent de profondeur ou de skip connections riches et perdent des détails.
2. Gestion du contexte : l'ASPP de DeepLab capture plusieurs échelles simultanément, ce qui aide à distinguer des classes visuellement proches (bâtiment vs ciel). MobileDet, avec ses convolutions depthwise, capture moins de contexte global, expliquant une légère chute sur les classes aux frontières diffuses.
3. Compatibilité avec les augmentations : U-Net VGG16 et DeepLab exploitent pleinement la diversité photométrique générée par Albumentations (flous, météo, bruit), tandis que YOLOv9 simplifié réagit moins bien aux distorsions optiques et aux variations de luminosité car sa tête PANet reste sensible aux textures fines.
4. Optimisation : l'entraînement SGD avec scheduler polynomial s'adapte mieux aux architectures profondes. Les modèles plus légers auraient pu bénéficier d'un AdamW avec weight decay ; cette piste est listée dans les travaux futurs.

Critère	Modèle recommandé	Rationale
Précision globale (mIoU / Dice)generate renvoie l'image originale et samples corruptions aléatoires issues du A.OneOf (identique à l'entraînement). • gallery applique séquentiellement les 15 transformations pour inspection détaillée. • Les sorties sont emballées dans AugmentedImage puis converties en data URLs côté route.	**DeepLabV3+ ResNet50**	Meilleure performance et stabilité.
Compromis vitesse / qualité	**MobileDet_seg**	Entraînement modéré, inférence légère.
Haute fidélité visuelle (VRAM ample)	**U-Net VGG16**	Rendu très détaillé mais coûteux.
Prototypage / tests pipeline	**U-Net mini**	Rapide à entraîner, utile pour valider les scripts.

4.5. Observations complémentaires

- Les modèles lourds (DeepLab, U-Net VGG16) bénéficient pleinement des corruptions photométriques et météo isolées, réduisant l'overfit sans perturber la géométrie des objets fins.
- Les architectures basées sur MobileNet montrent une bonne efficacité énergétique mais nécessitent un fine-tuning plus poussé pour rivaliser avec DeepLab.
- YOLOv9_seg, pensé pour la détection, souffre ici de sa tête segmentation simplifiée ; un rééquilibrage du décodeur multi-échelle serait nécessaire pour combler l'écart.

Ces constats confortent la décision de retenir DeepLabV3+ pour la mise en production via l'API Flask.

5. API Flask, architecture d'inférence et conclusion ouverte

5.1. Vue d'ensemble applicative

run.py instancie Flask et enregistre deux services : SegmentationService (chargement du modèle, pré/post-traitements, prédiction) et AugmentationService (tirage aléatoire ou galerie complète des 15 corruptions). La même pipeline Albumentations que celle d'entraînement est utilisée pour garantir la cohérence des démonstrations.

5.2. SegmentationService

1. Chargement du modèle .keras au démarrage.
2. Redimensionnement bilinéaire vers 512×1024, normalisation [0 ; 1], ajout de la dimension batch.
3. Prédiction model.predict, argmax sur l'axe classe.
4. Redimensionnement inverse du masque (nearest neighbor), colorisation via PALETTE, fusion alpha 0,5 avec l'image d'origine.
5. Sérialisation en data URLs via SegmentationResult pour l'API JSON.

5.3. AugmentationService

- generate renvoie l'image originale et samples corruptions aléatoires issues du A.OneOf (identique à l'entraînement).
- gallery applique séquentiellement les 15 transformations pour inspection détaillée.
- Les sorties sont emballées dans AugmentedImage puis converties en data URLs côté route.

5.4. Endpoints REST

- /predict : fichier image requis, renvoie original, mask, overlay.
- /augment : retourne des corruptions aléatoires (paramètre samples).
- /augment/gallery : livre l'ensemble des effets, utile pour la présentation.

Les payloads sont limités à 16 MiB (MAX_CONTENT_LENGTH). En absence de fichier, l'API renvoie un 400 explicite.

5.5. Déploiement, dépendances et hébergement Heroku

- Dépendances : l'environnement Python 3.10+ est requis. Les bibliothèques critiques (TensorFlow 2.12, Flask, Albumentations, Pillow, numpy) sont listées dans requirements.txt. Pour alléger le conteneur, il est conseillé d'utiliser l'image de base python:3.10-slim (cf. Dockerfile).
- Artefacts nécessaires : le modèle deeplab_resnet50_final.keras doit être placé dans app/models/ avant le lancement du serveur, faute de quoi le chargement échoue.
- Déploiement local : docker build -t cityscapes-seg . puis docker run -p 5000:5000 cityscapes-seg pour vérifier l'API avant publication.
- Hébergement Heroku (mode conteneur) :
    - heroku login pour authentifier le CLI.
    - heroku create <nom-app> pour créer l'application (ex : heroku create cityscapes-seg-demo).
    - heroku stack:set container -a <nom-app> afin d'activer le déploiement basé sur Docker.
    - heroku container:login puis heroku container:push web -a <nom-app> pour construire et pousser l'image définie par le Dockerfile.
    - heroku container:release web -a <nom-app> pour déployer l'image et démarrer le dyno.
    - heroku logs --tail -a <nom-app> pour superviser le démarrage et vérifier que le modèle est bien chargé.

- Variables d'environnement recommandées :
    - FLASK_ENV=production pour désactiver le mode debug.
    - MODEL_PATH=app/models/deeplab_resnet50_final.keras si l'on souhaite personnaliser le chemin sans modifier le code.
- Scalabilité : pour absorber plus de trafic, utiliser heroku ps:scale web=2 -a <nom-app> et activer l'auto-scaling via le dashboard Heroku. TensorFlow en CPU sur dyno standard permet environ 1–2 inférences/s ; pour plus de throughput, envisager une version quantifiée ou un add-on GPU externe.

5.6. Conclusion ouverte et pistes d'évolution

Le système actuel fournit une base robuste pour la segmentation urbaine temps quasi-réel. Plusieurs axes peuvent prolonger ce travail :

1. Optimisation temps réel : conversion du modèle DeepLabV3+ en format TensorRT ou TFLite quantifié pour accélérer l'inférence sur GPU embarqué ou CPU ARM.
2. Enrichissement du benchmarking : intégrer Fast-SCNN et d'autres architectures légères (BiSeNet, DDRNet) pour explorer des compromis supplémentaires entre latence et précision.
3. Segmentation multi-classes : élargir le remapping à davantage de classes Cityscapes ou à d'autres datasets (Mapillary, BDD100K) pour une couverture urbaine plus fine.
4. Monitoring en production : ajouter des endpoints de santé, des métriques Prometheus et une journalisation centralisée pour suivre les performances en exploitation.
5. Expérience utilisateur : proposer un ajustement en direct de l'opacité du masque, permettre l'export du masque en GeoJSON ou intégrer un comparateur d'images.
6. API élargie : offrir un endpoint batch pour traiter plusieurs images en une requête, ou un mode streaming (WebSocket) pour des flux vidéo.

En synthèse, DeepLabV3+ fournit des performances de pointe dans ce cadre réduit à huit classes, et l'API Flask actuelle constitue un socle solide pour des développements futurs, tant sur le plan de la recherche que de l'industrialisation. La disponibilité d'un conteneur Docker prêt pour Heroku facilite l'expérimentation rapide et l'observation en conditions réelles, tandis que l'utilisation d'Albumentations et du dataset Cityscapes remappé garantit une base scientifique robuste pour itérer sur de nouveaux scénarios urbains.