



FUTURE VISION TRANSPORT

Rapport technique

Segmentation d'Images pour pour le système embarqué d'une voiture autonome



Projet développé par [David Scanu](#) dans le cadre du parcours [AI Engineer](#) d'OpenClassrooms
Projet 8 - Traitez les images pour le système embarqué d'une voiture autonome

Introduction

Ce projet s'inscrit dans le développement d'un **système embarqué de vision par ordinateur** pour véhicules autonomes chez **Future Vision Transport**. L'entreprise conçoit des systèmes permettant aux véhicules autonomes de percevoir leur environnement grâce à l'analyse d'images en temps réel.

En tant qu'**ingénieur IA dans l'équipe R&D**, notre mission est de **développer le module de segmentation d'images** (composant 3) qui s'intègre entre le module de traitement d'images (2) et le système de décision (4). Ce module doit être capable d'**identifier et de segmenter précisément 8 catégories principales d'objets** dans des images de caméras embarquées.

Cette note technique présente la conception et l'implémentation d'un modèle de segmentation d'images optimisé pour la compréhension des scènes urbaines.

Objectifs

- **Développer un modèle de segmentation d'images performant** avec [Keras/TensorFlow](#)
- Concevoir et déployer une **API REST** avec [FastAPI](#)
- Créer une **application web de démonstration** avec [Next.js](#)
- **Documenter le processus et les résultats** de façon claire et professionnelle

Dépôt GitHub

L'intégralité du code du projet (notebook, backend et frontend) est disponible sur le dépôt GitHub :

- [Projet 8 : Traitement d'images pour le système embarqué d'une voiture autonome](#)

Documentation

Cette documentation technique est également disponible dans un article disponible à l'adresse suivante :

- [🚗👁 Segmentation d'Images pour pour le système embarqué d'une voiture autonome](#)

À propos

Projet développé par [David Scanu](#) dans le cadre du parcours [AI Engineer](#) d'OpenClassrooms : **Projet 8 - Traitez les images pour le système embarqué d'une voiture autonome.**

Récapitulatif des livrables

- [GitHub - DavidScanu/oc-ai-engineer-p08-images-systeme-voiture-autonome](#)
- [Notebook de modélisation sur Colab](#)
- [Backend API REST \[FastAPI\]](#)
- [Interface utilisateur \[Next.js\]](#)
- [Rapport technique](#)
- [Article complet disponible sur dev.to](#)
- [Support de présentation pour l'équipe de Laura](#)

1. Développement du Modèle de Segmentation

Le [notebook complet de conception du modèle de segmentation d'image](#) est disponible sur Colab.

1.1 Contexte et Enjeux

La [segmentation sémantique d'image](#) dans le contexte automobile représente un défi technique majeur. Contrairement à la classification d'images qui attribue une étiquette à une image entière, la segmentation sémantique doit **identifier et catégoriser chaque pixel de l'image**. Pour un véhicule autonome, cette capacité est cruciale : il faut distinguer la route des trottoirs, identifier les véhicules, détecter les piétons, reconnaître la végétation, etc.

La segmentation sémantique consiste à **étiqueter chaque pixel d'une image avec une classe correspondante** à ce qui est représenté. On parle aussi de "prédiction dense", car chaque pixel doit être prédit. Elle génère une image, sur laquelle chaque pixel est classifié.

1.2 Étude du jeu de données et préparation pour l'entraînement

1.2.1 Le jeu de données Cityscapes



Nous avons choisi le jeu de données [Cityscapes](#), une référence dans le domaine de la segmentation urbaine. Ce dataset présente plusieurs avantages décisifs pour notre cas d'usage :

- **5 000 images avec annotations fines** réparties en 2 975 images d'entraînement, 500 de validation et 1 525 de test
- **Diversité géographique** : 50 villes européennes différentes
- **Conditions variées** : Plusieurs saisons, conditions météorologiques bonnes à moyennes
- **Richesse des annotations** : 30 classes détaillées avec des annotations polygonales précises
- **Réalisme** : Images capturées depuis des véhicules en circulation réelle

1.2.2 Stratégie de regroupement des classes

L'une des décisions techniques les plus importantes a été le **regroupement des 30+ classes originales de Cityscapes en 8 catégories pertinentes pour la navigation autonome**. Cette simplification répond à plusieurs objectifs :

1. **Pertinence fonctionnelle** : Regrouper les éléments ayant la même importance pour la décision de navigation
2. **Performance computationnelle** : Réduire la complexité du modèle
3. **Robustesse** : Éviter la sur-spécialisation sur des classes très spécifiques

Notre mapping final :

```
Python
class_groups = {
    'flat': ['road', 'sidewalk', 'parking', 'rail track'],
    'human': ['person', 'rider'],
    'vehicle': ['car', 'truck', 'bus', 'motorcycle', 'bicycle', ...],
    'construction': ['building', 'wall', 'fence', 'bridge', ...],
    'object': ['pole', 'traffic sign', 'traffic light', ...],
    'nature': ['vegetation', 'terrain'],
    'sky': ['sky'],
    'void': ['unlabeled', 'out of roi', ...]
}
```

Cette approche nous permet de maintenir une granularité suffisante pour la prise de décision tout en gardant un modèle tractable.

1.2.3 Pipeline de Préparation des Données

Notre pipeline de préparation des données intègre plusieurs étapes :

Redimensionnement intelligent : Les images Cityscapes originales (2048×1024) sont redimensionnées à 224×224 . Cette taille est optimale pour l'utilisation de [MobileNetV2 pré-entraîné](#).

Stratégie de validation : Pour éviter le "[data leakage](#)", nous avons adopté une approche rigoureuse :

- Division du **dataset 'train'** original en **80% entraînement / 20% validation**
- Utilisation du **dataset 'val'** original comme **dataset de test final** (garantit que notre évaluation finale est réalisée sur des données jamais vues)
- Les **masques du dataset 'test' étaient indisponibles (complètement noirs)** (nous n'avons pas utilisé ce dossier)

Augmentation de données : Nous avons implémenté des techniques d'augmentation adaptées à la segmentation :

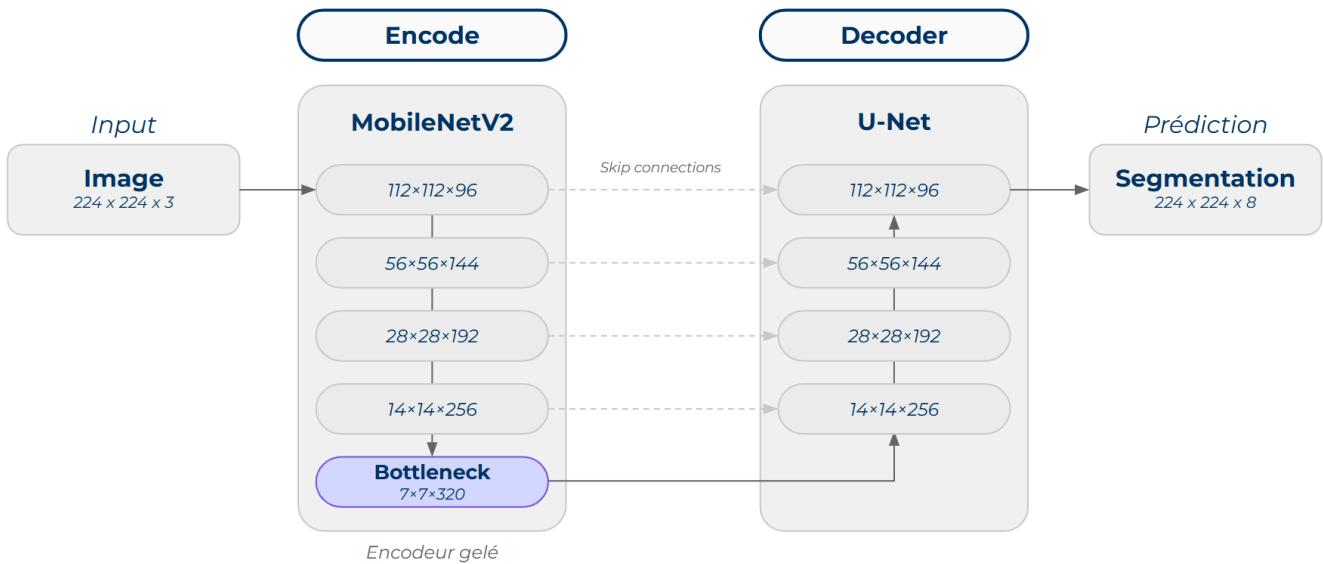
1. **Flip horizontal aléatoire** (50% de probabilité)
 - Appliqué de manière synchronisée sur l'image et le masque
2. **Variation de luminosité aléatoire**
 - Appliquée uniquement sur l'image (pas sur le masque)
 - Amplitude : ± 0.1

- Avec clipping pour maintenir les valeurs dans [0,1]

Python

```
def augment_data(image, mask):
    # Flip horizontal synchronisé image/masque
    if tf.random.uniform() > 0.5:
        image = tf.image.flip_left_right(image)
        mask = tf.image.flip_left_right(mask)
    # Variation de luminosité (image seulement)
    image = tf.image.random_brightness(image, 0.1)
    return image, mask
```

1.3 Architecture du Modèle : MobileNetV2-UNet



Architecture U-Net avec Backbone MobileNetV2 pour segmentation sémantique d'images

1.3.1 Structure générale en forme de U

Nous avons développé une architecture en forme de U qui combine la puissance de [MobileNetV2](#) (un réseau léger et efficace) avec la structure en forme de U de l'[architecture U-Net](#) pour réaliser une segmentation d'images. Notre modèle hybride utilise **MobileNetV2** pour l'extraction de caractéristiques et **U-Net** pour reconstruire des segmentations précises.

- **Objectif** : Attribuer une classe à chaque pixel de l'image (segmentation).
- **Forme en U** : L'architecture suit une forme en "U" avec une partie descendante (encodeur) et une partie ascendante (décodeur).

Architecture U-Net Partie encodeur (descendante)

- **Backbone MobileNetV2** : Utilisation de [MobileNetV2](#) comme encodeur léger et optimisé pour les appareils mobiles.
- **Gel des poids** : Avec `base_model.trainable = False`, les poids pré-entraînés sur ImageNet sont conservés, ce qui accélère l'entraînement et limite le surapprentissage.

- **Extraction multi-échelle** : Les caractéristiques sont extraites à différents niveaux de résolution pour les connexions de saut :
 - `skip1` : 112×112
 - `skip2` : 56×56
 - `skip3` : 28×28
 - `skip4` : 14×14
 - `bottleneck` : 7×7 (le niveau le plus profond du U-Net)

L'encodeur extrait des caractéristiques à cinq niveaux de résolution différents. Cette extraction multi-échelle est cruciale : **les premières couches capturent des détails fins** (textures, contours) tandis que **les couches profondes encodent des informations sémantiques de haut niveau**.

Stratégie de Transfer Learning : Nous avons opté pour une approche de "**frozen encoder**" (encoder gelé) dans notre configuration.

Cette décision présente plusieurs avantages :

- **Entraînement plus rapide** (moins de paramètres à optimiser)
- **Stabilité accrue (les caractéristiques ImageNet sont déjà robustes)**
- **Réduction du risque de sur-apprentissage**
- Possibilité d'entraînement avec des ressources limitées

Partie décodeur (ascendante)

- **Reconstruction progressive** : À chaque étape, la résolution est doublée à l'aide de convolutions transposées :
 - $7 \times 7 \rightarrow 14 \times 14$
 - $14 \times 14 \rightarrow 28 \times 28$
 - $28 \times 28 \rightarrow 56 \times 56$
 - $56 \times 56 \rightarrow 112 \times 112$
 - $112 \times 112 \rightarrow 224 \times 224$
- **Connections de saut (Skip connections)** : Chaque niveau du décodeur est enrichi par les caractéristiques du niveau correspondant de l'encodeur pour préserver les détails spatiaux.
- **Traitement post-upsampling** : Chaque étape comprend :
 - [Convolution transposée](#)
 - [BatchNormalization](#)
 - [Activation ReLU](#)
 - Concaténation avec les features de l'encodeur
 - Convolution 3×3 pour affiner

1.3.2 Construction et Entraînement du Modèle

- **Modèle U-Net personnalisé** basé sur **MobileNetV2**, avec :
 - **Paramètres ajustables :**
 - Nombre de classes
 - Taille d'entrée
 - Taux d'apprentissage
 - Backbone gelé ou non
 - **Sortie** : carte de segmentation pixel-par-pixel via une convolution 1×1 suivie d'une activation softmax.
 - **Compilation :**
 - Optimiseur Adam
 - Perte SparseCategoricalCrossentropy
 - Métrique SparseCategoricalAccuracy (adapté car nos masques contiennent directement les indices des classes (format sparse) plutôt que des vecteurs one-hot)

Python

```
def create_segmentation_model(num_classes=8, input_size=(224, 224, 3),
                             learning_rate=0.0001, base_trainable=False):
    """
    Create a segmentation model with configurable parameters
    """

    # Input tensor
    inputs = keras.layers.Input(shape=input_size)

    # Create MobileNetV2 base model
    base_model = keras.applications.MobileNetV2(
        input_shape=input_size,
        include_top=False,
        weights='imagenet'
    )

    # Configuration de l'entraînement du modèle de base
    base_model.trainable = base_trainable

    # Create encoder using MobileNetV2
    # Get outputs at different levels for skip connections
    skip1 = base_model.get_layer('block_1_expand_relu').output      # 112x112
    skip2 = base_model.get_layer('block_3_expand_relu').output      # 56x56
    skip3 = base_model.get_layer('block_6_expand_relu').output      # 28x28
    skip4 = base_model.get_layer('block_13_expand_relu').output     # 14x14

    # Get bottleneck
    bottleneck = base_model.get_layer('block_16_project').output    # 7x7

    # Use Model API to create encoder model
    encoder = keras.Model(inputs=base_model.input, outputs=[skip1, skip2, skip3, skip4, bottleneck])

    # Encoder part - use the encoder model
    x = inputs
    skips = encoder(x)

    # Bottleneck
    x = skips[4] # Use the bottleneck from encoder

    # Decoder part - upsampling and skip connections
    # First upsampling: 7x7 -> 14x14
    x = keras.layers.Conv2DTranspose(256, 3, strides=2, padding='same')(x)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Activation('relu')(x)
    x = keras.layers.Concatenate()([x, skips[3]])
```

```

x = keras.layers.Conv2D(256, 3, padding='same', activation='relu')(x)

# Second upsampling: 14x14 -> 28x28
x = keras.layers.Conv2DTranspose(128, 3, strides=2, padding='same')(x)
x = keras.layers.BatchNormalization()(x)
x = keras.layers.Activation('relu')(x)
x = keras.layers.Concatenate()([x, skips[2]])
x = keras.layers.Conv2D(128, 3, padding='same', activation='relu')(x)

# Third upsampling: 28x28 -> 56x56
x = keras.layers.Conv2DTranspose(64, 3, strides=2, padding='same')(x)
x = keras.layers.BatchNormalization()(x)
x = keras.layers.Activation('relu')(x)
x = keras.layers.Concatenate()([x, skips[1]])
x = keras.layers.Conv2D(64, 3, padding='same', activation='relu')(x)

# Fourth upsampling: 56x56 -> 112x112
x = keras.layers.Conv2DTranspose(32, 3, strides=2, padding='same')(x)
x = keras.layers.BatchNormalization()(x)
x = keras.layers.Activation('relu')(x)
x = keras.layers.Concatenate()([x, skips[0]])
x = keras.layers.Conv2D(32, 3, padding='same', activation='relu')(x)

# Final upsampling: 112x112 -> 224x224
x = keras.layers.Conv2DTranspose(16, 3, strides=2, padding='same')(x)
x = keras.layers.BatchNormalization()(x)
x = keras.layers.Activation('relu')(x)
x = keras.layers.Conv2D(16, 3, padding='same', activation='relu')(x)

# Output layer
outputs = keras.layers.Conv2D(num_classes, 1, padding='same', activation='softmax')(x)

# Create model
model = keras.Model(inputs=inputs, outputs=outputs, name="MobileNetV2-UNet")

# Compile model avec learning rate configurable
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(),
        # keras.metrics.MeanIoU(num_classes=num_classes)
    ]
)

return model

```

Détails des paramètres

- **Total des paramètres** : 5,418,600
 - **Entraînables** : 3,575,624 (décodeur et tête de segmentation)
 - **Non-entraînables** : 1,842,976 (backbone gelé)
- Si **base_trainable=True**, tous les paramètres deviennent entraînables, offrant plus de flexibilité au prix d'un entraînement plus long.

1.3.3 Justification du choix architectural

Le choix de l'**architecture MobileNetV2-UNet** résulte d'une analyse approfondie des contraintes de notre projet embarqué, combinant les avantages complémentaires de deux architectures éprouvées :

Pourquoi U-Net ?

- **Architecture de référence** pour la segmentation sémantique avec structure encoder-decoder
- **Connexions résiduelles (skip connections)** préservant les détails spatiaux fins lors de la reconstruction
- **Adaptée à la reconstruction** de résolution complète depuis des caractéristiques compressées

Pourquoi MobileNetV2 ?

- **Conception spécifique** pour les applications mobiles et embarquées avec ressources limitées
- **Transfer learning efficace** grâce aux poids pré-entraînés sur ImageNet
- **Vitesse d'exécution** compatible avec les contraintes temps réel du véhicule autonome

Cette **architecture hybride MobileNetV2-UNet** présente des bénéfices uniques pour notre cas d'usage :

- **Efficacité computationnelle** : Optimisée pour les environnements à ressources contraintes
- **Robustesse par transfer learning** : Exploitation des représentations ImageNet pour une convergence rapide
- **Adaptabilité multi-tâches** : Architecture générique applicable à diverses tâches de segmentation urbaine

1.4 Entraînement et optimisation

1.4.1 Stratégie d'entraînement

Notre approche d'entraînement intègre plusieurs techniques d'optimisation :

Callbacks Avancés

```
Python
callbacks = [
    keras.callbacks.ReduceLROnPlateau(
        monitor='val_loss', factor=0.2, patience=2, min_lr=1e-6
    ),
    keras.callbacks.EarlyStopping(
        monitor='val_sparse_categorical_accuracy',
        patience=5, restore_best_weights=True
    ),
    keras.callbacks.ModelCheckpoint(
        filepath='best_model.keras',
        save_best_only=True, mode='max'
    )
]
```

- **ReduceLROnPlateau** : Réduction adaptative du taux d'apprentissage
- **EarlyStopping** : Arrêt anticipé pour éviter le sur-apprentissage

- **ModelCheckpoint** : Sauvegarde du meilleur modèle

Gestion des expériences de Machine Learning avec MLflow

Nous avons intégré [MLflow](#) pour le suivi des expériences, permettant :

- **Versioning des modèles** et configurations
- **Traçabilité complète des hyperparamètres**
- **Comparaison objective des performances**
- **Reproductibilité des résultats**

1.4.2 Configuration pour l'entraînement ([Hyperparamètres](#))

La configuration de nos hyperparamètres est la suivante :

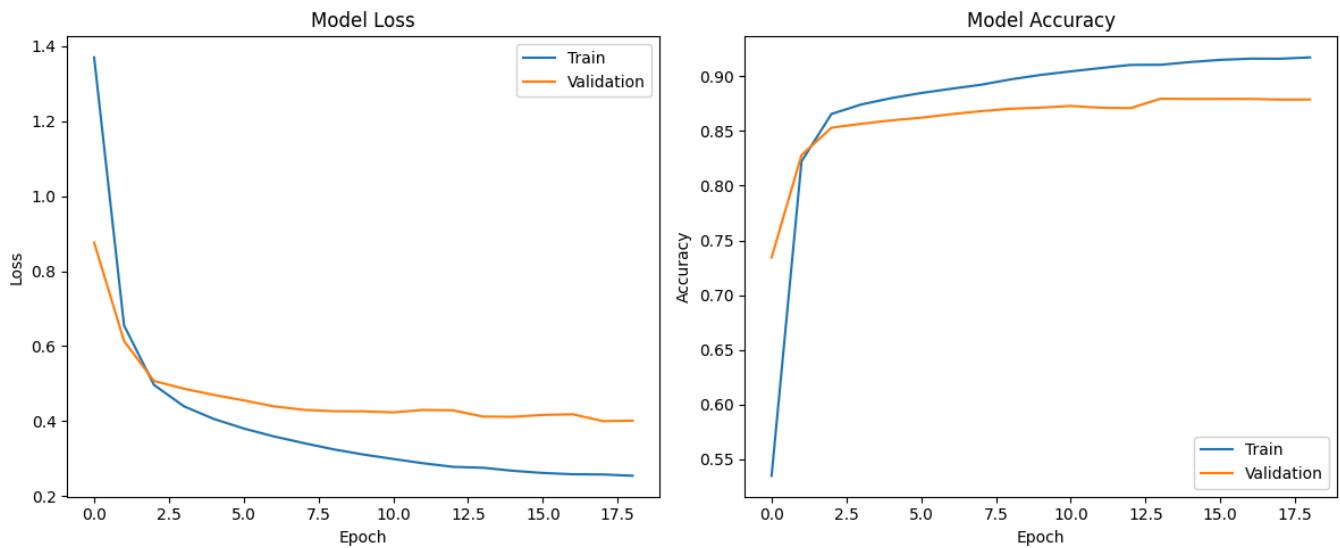
Python

```
EXPERIMENT_CONFIG = {
    "exp_name": "exp_001_baseline",
    "description": "Baseline MobileNetV2-UNet avec frozen encoder",
    "params": {
        "learning_rate": 0.0001,
        "batch_size": 8,
        "epochs": 20,
        "base_model_trainable": False,
        "validation_split": 0.2,
        "augmentation": True
    }
}
```

Cette configuration privilégie la stabilité et la reproductibilité, servant de fondation pour les expérimentations futures.

1.4.3 Analyse de l'entraînement du modèle

L'entraînement de notre modèle **MobileNetV2-UNet** s'est déroulé sur **18 époques** avec une **convergence satisfaisante**. Les graphiques d'évolution révèlent un comportement d'apprentissage sain et typique d'un modèle bien configuré.



Évolution de la Perte : La courbe de loss montre une décroissance rapide et régulière depuis des valeurs initiales élevées (≈ 1.4) vers une stabilisation autour de 0.25 pour l'entraînement et 0.4 pour la validation. Cette convergence rapide durant les 5 premières époques témoigne de l'**efficacité du transfer learning avec MobileNetV2 pré-entraîné sur ImageNet**. L'écart modéré entre les courbes train et validation (≈ 0.15) indique un **niveau de sur-apprentissage acceptable**, sans divergence critique.

Progression de l'Accuracy : L'évolution de la précision confirme l'apprentissage efficace avec une montée rapide de 55% à plus de 85% en quelques époques, puis une stabilisation progressive vers 91% pour l'entraînement et 87% pour la validation. Cette convergence démontre que **le modèle a atteint sa capacité d'apprentissage optimale avec la configuration actuelle (encoder gelé)**. L'écart final de 4% entre train et validation reste dans une fourchette acceptable pour un modèle de production, suggérant une **bonne généralisation** sur des données non vues.

Cette dynamique d'entraînement valide notre approche conservative avec **encoder gelé**, offrant un modèle stable et performant en un temps d'entraînement réduit, parfaitement adapté aux contraintes de développement rapide d'un système embarqué.

1.5 Évaluation et Métriques

1.5.1 Métriques de Performance

L'évaluation de modèles de segmentation nécessite des métriques spécialisées. Nous utilisons principalement :

Intersection over Union (IoU)

Mesure le **chevauchement entre la prédition et la vérité terrain.**

None

$\text{IoU} = \text{Intersection} / \text{Union}$

Cette métrique est particulièrement pertinente car elle pénalise à la fois les faux positifs et les faux négatifs.

Mean IoU (mIoU)

Moyenne des IoU de toutes les classes, offrant une vue d'ensemble des performances.

Précision Pixel-wise

Pourcentage de pixels correctement classifiés, métrique intuitive mais pouvant être biaisée par les classes majoritaires.

1.5.2 Résultats de notre Configuration

Notre modèle atteint des performances encourageantes :

- **Mean IoU** : 63.25%
- **Précision globale** : 87.95%
- **Perte finale** : 0.4123

Analyse par Classe :

Classe	IoU	Commentaire
flat	0.9084	 Excellent (routes bien détectées)
human	0.3196	 Difficile (objets petits/variables)
vehicle	0.7469	 Bon (formes caractéristiques)
construction	0.7454	 Bon (structures larges)
object	0.0648	 Très difficile (objets fins)
nature	0.7983	 Bon (textures distinctives)
sky	0.8339	 Excellent (région homogène)
void	0.6425	 Acceptable

Matrice de confusion

La [matrice de confusion](#) constitue un outil d'évaluation pour les modèles de classification, particulièrement crucial en segmentation sémantique où chaque pixel doit être correctement assigné à sa classe.

Pour un système de vision automobile, cette analyse est importante car elle révèle **quelles confusions pourraient compromettre la sécurité** (ex: confondre un piéton avec un objet statique).



1.5.3 Analyse des Performances

Points Forts :

- **Excellent détection des surfaces planes** (routes, ciel)
- **Bonne segmentation des grandes structures** (bâtiments, végétation)

Points d'Amélioration :

- **Détection des objets fins** (poteaux, panneaux) perfectible
- **Segmentation des humains variable** (due à leur taille et variabilité)
- **Besoin d'optimisation pour les petits objets**

1.6 Pipeline de Prédiction

Pour valider les performances de notre modèle en conditions réelles, nous avons développé un **pipeline de prédiction complet** permettant de traiter aussi bien des **images du jeu de données de test** que des **images externes**. Ce pipeline intègre toutes les étapes nécessaires depuis le **chargement du modèle** jusqu'à l'**analyse statistique des résultats**.

Python

```
def predict_single_image(model, config, image_path=None, image_array=None, img_size=(224, 224)):  
    """Réalise une prédiction complète avec préprocessing et post-analyse"""\n  
  
    # Chargement et préprocessing adaptatif  
    if image_path is not None:  
        image_pil = Image.open(image_path).convert('RGB')  
        image_array = np.array(image_pil)  
  
    # Pipeline de normalisation identique à l'entraînement  
    image_resized = tf.image.resize(image_array, img_size, method='bilinear')  
    image_normalized = tf.cast(image_resized, tf.float32) / 255.0  
    image_batch = tf.expand_dims(image_normalized, axis=0)  
  
    # Prédiction et conversion en masque de classes  
    predictions = model.predict(image_batch, verbose=0)  
    pred_mask = tf.argmax(predictions, axis=-1)[0].numpy()  
  
    # Analyse statistique automatique par classe  
    unique_classes, counts = np.unique(pred_mask, return_counts=True)  
    total_pixels = pred_mask.size  
  
    class_stats = []  
    for class_id, count in zip(unique_classes, counts):  
        class_name = config['group_names'][class_id]  
        percentage = (count / total_pixels) * 100  
        class_stats.append({  
            'class_id': int(class_id),  
            'class_name': class_name,  
            'pixel_count': int(count),  
            'percentage': percentage  
        })  
  
    return {  
        'prediction_mask': pred_mask,  
        'class_statistics': class_stats,  
        'predictions_raw': predictions[0]  
    }
```

1.6.1. Résultats de prédiction

Les résultats de prédiction se présentent sous la forme suivante :

Masque de Segmentation (pixels) : Le résultat principal est un **masque 224x224** où chaque pixel contient l'**ID de la classe prédictive** (0-7). Cette représentation permet une analyse précise de la scène.

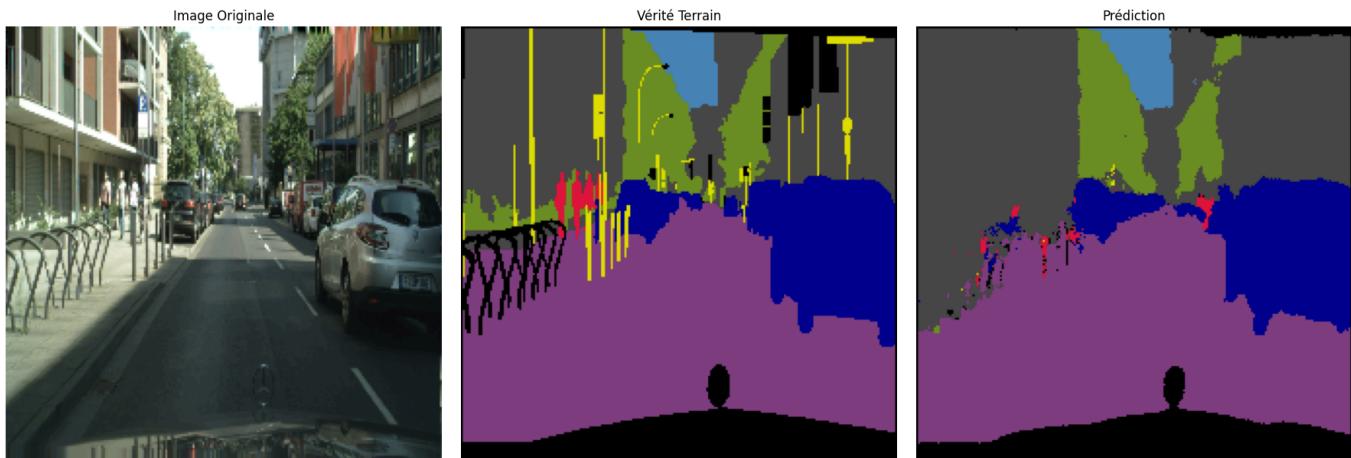
Distribution Statistique des Classes : Pour chaque prédiction, nous générerons automatiquement un tableau statistique détaillant la répartition des classes :

Python

```
📊 Classes détectées:  
flat:      19536 pixels ( 38.9%) # Routes et surfaces de circulation  
construction: 14510 pixels ( 28.9%) # Bâtiments et infrastructures  
nature:     8285 pixels ( 16.5%) # Végétation et terrain naturel  
void:       5459 pixels ( 10.9%) # Zones non classifiées  
vehicle:    1630 pixels (  3.2%) # Véhicules en circulation  
human:      594 pixels (  1.2%) # Piétons et cyclistes  
sky:        98 pixels (  0.2%) # Ciel visible  
object:     64 pixels (  0.1%) # Signalisation et mobilier urbain
```

Visualisation Tripartite : Notre système génère systématiquement trois vues :

- Image originale
- Vérité terrain (quand disponible)
- Prédiction colorisée selon notre palette de couleurs spécifique



4.3 Analyse qualitative des performances

L'évaluation sur des images réelles révèle des **performances encourageantes** avec des points forts et des axes d'amélioration clairement identifiés :

Excellente Détection des Structures Dominantes : Notre modèle démontre une capacité remarquable à identifier les éléments structurants de la scène urbaine. La segmentation des surfaces planes (routes, trottoirs) atteint 38.9% de couverture dans l'exemple analysé, avec des contours nets et une classification cohérente. Les bâtiments et constructions (28.9%) sont également très bien délimités, ce qui est crucial pour la navigation autonome.

Robustesse sur la végétation et l'environnement : La détection de la végétation (16.5%) montre une bonne capacité de généralisation du modèle. Les arbres, buissons et espaces verts sont correctement identifiés malgré leur variabilité de texture et de forme, témoignant de l'**efficacité du transfer learning depuis ImageNet**.

Défis sur les objets de petite taille : L'analyse révèle des difficultés attendues sur les éléments fins et de petite taille. Les objets de signalisation ne représentent que 0.1% des pixels détectés, ce qui correspond à la fois à leur **faible représentation spatiale réelle** et aux **limitations du modèle sur ces éléments critiques**.

Gestion Intelligente des Zones Ambiguës : La catégorie "void" (10.9%) capture efficacement les zones d'incertitude et les éléments non classifiables, évitant les fausses classifications qui pourraient être dangereuses. Cette approche conservative est préférable dans un contexte automobile.

Cette analyse confirme que **notre architecture MobileNetV2-UNet offre un bon équilibre** pour les contraintes d'un système de vision embarqué, avec des performances robustes sur les éléments critiques pour la prise de décision de navigation tout en maintenant une efficacité computationnelle compatible avec les ressources limitées d'un véhicule autonome.

1.7 Innovations et optimisations futures

1.7.1 Pistes d'amélioration identifiées

Entraînement Progressif Stratégie en deux phases :

1. **Phase 1** : Encoder gelé (configuration actuelle)
2. **Phase 2** : Fine-tuning avec encoder dégelé et taux d'apprentissage réduit

Augmentation de données avancée

- Simulation de conditions météorologiques (pluie, brouillard)
- Variations d'éclairage plus poussées
- Transformations géométriques adaptées au contexte automobile

Architecture Hybride Exploration de techniques comme :

- **Attention mechanisms** pour améliorer la détection des petits objets
- **Ensembling de modèles** pour améliorer la robustesse
- **Multi-scale training** pour robustesse aux variations d'échelle

1.8 Conclusion de la partie modélisation

Le développement de notre **modèle de segmentation MobileNetV2-UNet** représente un équilibre réussi entre performance et efficacité computationnelle. Avec un **Mean IoU de 63.25 %** et une **précision globale de 87.95 %**, notre modèle fournit une base solide pour le système de vision du véhicule autonome. Les pistes d'amélioration identifiées offrent un **roadmap clair pour les développements futurs**. La mise en place d'un **pipeline MLOps avec MLflow** et notre organisation modulaire garantissent la reproductibilité et facilitent l'**itération continue** sur les performances du modèle.

2. Développement du Backend FastAPI - API de Segmentation Sémantique

Le backend de l'application est une API REST développée avec [FastAPI](#). Cette API **expose notre modèle de deep learning (MobileNetV2-UNet)** entraîné pour la **segmentation sémantique d'images urbaines** du dataset Cityscapes.

Architecture technique

```
None
app/backend/
├── main.py           # Point d'entrée FastAPI
├── config.py         # Configuration et variables d'environnement
└── models/
    └── predictor.py   # Logique de prédiction et chargement du modèle
└── routers/
    └── segmentation.py # Endpoints de l'API
└── schemas/
    └── prediction.py  # Modèles Pydantic pour validation
└── utils/
    └── image_processing.py # Utilitaires de traitement d'image
└── requirements.txt   # Dépendances Python
```

Fonctionnalités principales

1. Chargement du modèle depuis MLflow

Le modèle est hébergé sur **MLflow** et chargé dynamiquement au démarrage de l'API :

- **Modèle : MobileNetV2-UNet** (format Keras 3.x)
- **Poids** : ~25MB
- **Classes** : 8 catégories (flat, human, vehicle, construction, object, nature, sky, void)

2. Prédiction avec génération d'artefacts

L'API génère pour chaque prédiction :

- **Masque de segmentation** : Classification pixel par pixel
- **Visualisations** : Images côte à côte et superposition
- **Statistiques** : Distribution des classes détectée

- **Sauvegarde** : Artefacts de prédiction sauvegardés dans le dossier `/predictions`

Endpoints de l'API

POST `/api/v1/segmentation/predict`

Notre endpoint principal pour la segmentation d'images.

- **Entrée** : Image (PNG, JPEG) via `multipart/form-data`
- **Sortie** : JSON contenant :

```
JSON
{
  "class_statistics": [...],
  "image_size": [largeur, hauteur],
  "dominant_class": "flat",
  "dominant_class_percentage": 42.5,
  "images": {
    "original": "data:image/png;base64,...",
    "prediction_mask": "data:image/png;base64,...",
    "overlay": "data:image/png;base64,...",
    "side_by_side": "data:image/png;base64,..."
  }
}
```

GET `/api/v1/segmentation/health`

Ce endpoint vérifie l'état de santé de l'API.

```
JSON
{
  "status": "healthy",
  "model_loaded": true
}
```

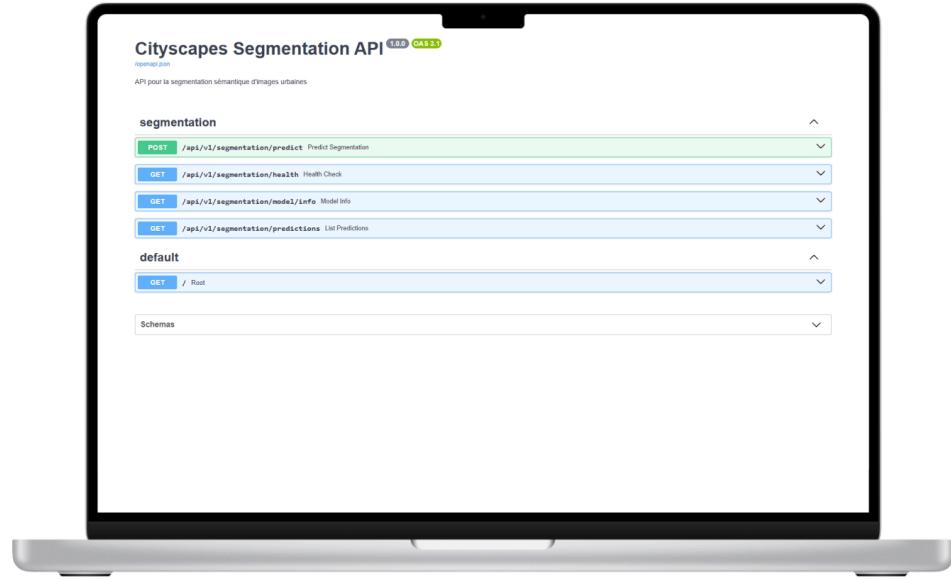
GET `/api/v1/segmentation/model/info`

Ce endpoint renvoie des informations détaillées sur le modèle chargé.

```
JSON
{
  "model_name": "MobileNetV2-UNet",
  "input_shape": [null, 224, 224, 3],
  "num_classes": 8,
  "class_names": ["flat", "human", "vehicle", ...],
  "tensorflow_version": "2.18.0"
}
```

GET /docs

Documentation interactive **Swagger UI** générée automatiquement par FastAPI. Ce endpoint est très utile pour effectuer des requêtes de tests vers notre API, en attendant de développer le frontend.



Technologies utilisées

- **FastAPI** : Framework web asynchrone haute performance
- **TensorFlow 2.18 (CPU)** : Inférence du modèle
- **Gunicorn + Uvicorn** : Serveur ASGI pour la production
- **Pillow** : Manipulation d'images
- **MLflow** : Gestion et versioning du modèle

Optimisations pour la production

1. **TensorFlow CPU** : Version allégée sans GPU (250MB vs 500MB)
2. **Workers limités** : 1 worker pour économiser la RAM
3. **Timeout ajusté** : 120s pour les prédictions complexes
4. **Stockage temporaire** : Utilisation de `/tmp` sur Railway

Déploiement sur Railway

Notre backend FastAPI est déployée sur [Railway](#) qui offre des ressources suffisantes pour déployer un modèle de segmentation d'image, gratuitement.

Configuration Railway

Le déploiement sur Railway utilise un fichier `railway.json` :

```
JSON
{
```

```
"$schema": "https://railway.app/railway.schema.json",
"build": {
  "builder": "NIXPACKS",
  "buildCommand": "pip install -r requirements.txt"
},
"deploy": {
  "startCommand": "uvicorn main:app --host 0.0.0.0 --port $PORT",
  "restartPolicyType": "ON_FAILURE",
  "restartPolicyMaxRetries": 10
}
}
```

Variables d'environnement

Les "credentials" sont configurés via le **dashboard Railway** :

- `MLFLOW_TRACKING_URI` : URL du serveur MLflow
- `AWS_ACCESS_KEY_ID` / `AWS_SECRET_ACCESS_KEY` : Accès S3 pour MLflow
- `RUN_ID` : Identifiant de l'expérience MLflow
- `PORT` : Port d'écoute (géré automatiquement)
- `FRONTEND_URL` : URL de l'interface frontend (pour éviter les erreurs CORS)

Processus de déploiement

1. **Initialisation** : `railway init` pour créer le projet
2. **Configuration** : Ajout des variables via le dashboard
3. **Déploiement** : `railway up` pour déployer le code
4. **Domaine** : `railway domain` pour générer l'URL publique

URL de production

L'API est accessible à : <https://future-vision-api-production.up.railway.app/docs>

Cette architecture permet un déploiement rapide et scalable d'un modèle de deep learning, avec une API REST moderne et bien documentée, idéale pour être consommée par une application frontend.

3. Développement de l'interface frontend Next.js

3.1 Architecture et choix techniques

L'interface utilisateur a été développée avec [Next.js 15.3](#) en JavaScript, privilégiant la simplicité et la performance. Le framework [React](#) permet de développer une interface adaptée aux besoins de **visualisation des résultats de segmentation**.

Stack technique

- **Next.js 15.3 : Framework React** pour le développement frontend
- **Bootstrap 5.3 : Framework CSS** pour un design responsive et professionnel
- **Fetch API** : Communication avec l'**API FastAPI de prédiction**
- **Base64** : Gestion des **images encodées** retournées par l'API

3.2 Fonctionnalités principales

L'application propose une interface intuitive pour tester le modèle de segmentation sémantique :

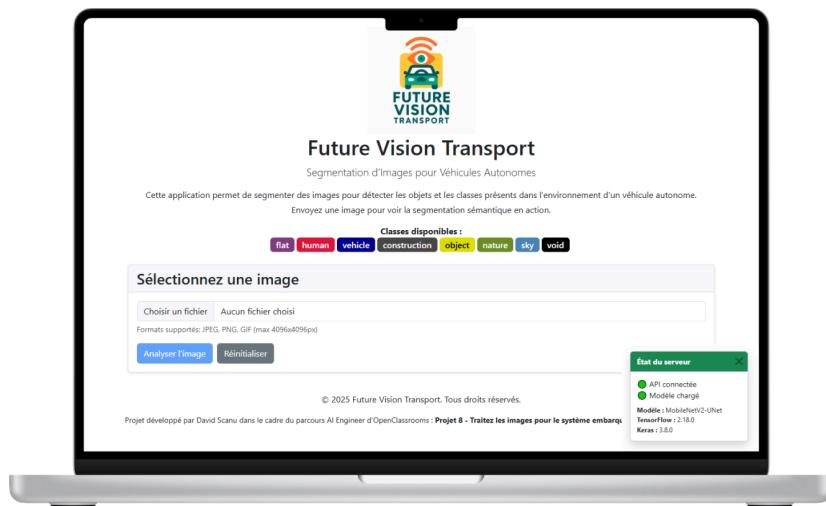
- **Upload d'images** : Sélection et prévisualisation des images (formats JPEG, PNG, GIF)
- **Validation** : Contrôle de la taille maximale (4096x4096px) et du format
- **Prédiction en temps réel** : Appel à l'API avec feedback visuel de progression
- **Visualisation multi-format** :
 - **Image avec overlay** de segmentation semi-transparent
 - **Comparaison côté à côté** (originale vs prédiction)
 - **Masque de segmentation** isolé
- **Statistiques détaillées** : Distribution des classes, classe dominante, métadonnées

3.3 Interface utilisateur

L'interface suit les principes de design UX/UI pour une utilisation professionnelle.

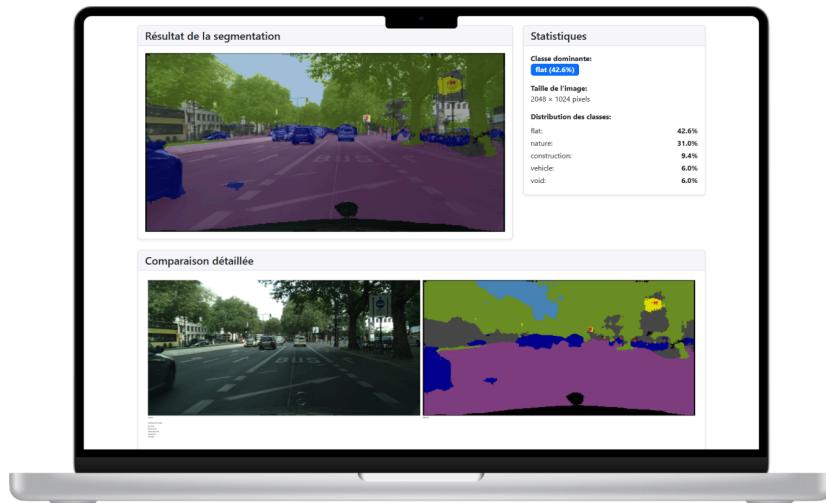
État initial

- **Formulaire d'upload central** avec instructions claires
- Validation en temps réel du fichier sélectionné
- Boutons d'action désactivés tant qu'aucune image n'est sélectionnée



Résultats de prédiction

- **Affichage principal de l'image avec overlay de segmentation**
- **Panneau latéral avec statistiques** et distribution des classes
- Section de comparaison détaillée avec visualisations multiples
- **Design responsive** adapté aux différentes tailles d'écran



3.4 Déploiement et Accessibilité

L'application est déployée sur [Vercel](#) :

URL de production : <https://oc-ai-engineer-p08-images-systeme-v.vercel.app/>

- **Déploiement automatique** : Intégration continue depuis le repository Git
- **Configuration** : Variables d'environnement pour l'URL de l'API backend
 - `NEXT_PUBLIC_API_URL` : URL de l'API REST de prédiction

3.5 Intégration API

L'interface communique avec l'**API FastAPI** de segmentation sémantique d'images via des **requêtes HTTP** :

- **Endpoint** : `POST /predict` pour l'analyse d'images
- **Format** : Multipart/form-data pour l'upload de fichiers
- **Réponse** : JSON contenant les images encodées et statistiques

Cette architecture **facilite la maintenance** et permet une **évolution future** vers des fonctionnalités avancées comme la gestion de lots d'images ou l'historique des prédictions.

Conclusion

Ce projet de développement d'un **système de segmentation d'images pour véhicule autonome** a permis de concevoir une **solution complète**, de la **modélisation au déploiement en production**.

Réalisations techniques

Modélisation IA : Nous avons développé un modèle **MobileNetV2-UNet** atteignant **63.25% de Mean IoU** sur 8 catégories d'objets urbains. L'architecture hybride combine l'**efficacité de MobileNetV2** pour les contraintes embarquées avec la **précision de U-Net** pour la reconstruction spatiale.

Pipeline MLOps : L'intégration de **MLflow** a permis un **suivi rigoureux des expériences** et le versioning des modèles, garantissant la reproductibilité et la traçabilité des développements.

Architecture API : Le **backend FastAPI expose le modèle via une API REST moderne** avec **génération automatique de visualisations et statistiques détaillées**, [déployée sur Railway](#).

Interface utilisateur : L'application **Next.js** offre une **interface intuitive** pour tester le modèle avec visualisations multi-formats (overlay, comparaison, masques) et facilement [déployée sur Vercel](#).

Axes d'Amélioration

Les principales pistes d'optimisation incluent un **entraînement progressif** (encoder dégelé en phase 2), une **augmentation de données avancée** simulant conditions météorologiques et variations d'éclairage, l'intégration d'**attention mechanisms** pour améliorer la détection des petits objets (panneaux, poteaux). Également, l'ajout d'un **système d'authentification API**, d'un **cache de prédictions** et de **métriques de monitoring** en production compléterait l'industrialisation du système.

A propos de ce projet

Ce projet démontre la faisabilité technique d'un **module de segmentation pour Future Vision Transport**, avec une base solide pour l'évolution vers un **système de production robuste**.

Récapitulatif des livrables

- [GitHub - DavidScanu/oc-ai-engineer-p08-images-systeme-voiture-autonome](#)
- [Notebook de modélisation sur Colab](#)
- [Backend API REST \[FastAPI\]](#)
- [Interface utilisateur \[Next.js\]](#)
- [Rapport technique](#)
- [Article complet disponible sur dev.to](#)
- [Support de présentation pour l'équipe de Laura](#)