Note Technique

Contexte et portée du projet :

Future Vision Transport est une entreprise pionnière dans le développement de systèmes embarqués de vision par ordinateur pour les véhicules autonomes. Au sein de l'équipe R&D, divers ingénieurs travaillent sur des composantes distinctes de ce système, dont l'acquisition d'images en temps réel, le traitement des images, la segmentation des images, et le système de décision. Mon rôle se concentre sur la segmentation des images, un maillon crucial qui s'intercale entre le traitement et le système de décision
Le projet vise à concevoir un modèle de segmentation d'images s'intégrant harmonieusement dans la chaîne complète du système embarqué. Cette intégration nécessite une attention particulière aux interfaces avec les autres composantes, en particulier l'alimentation depuis le bloc de traitement des images et la fourniture de données au système de décision.

Interaction avec l'équipe :

- Franck (Traitement des Images) : Elles se composent d'images segmentées et annotées issues de caméras embarquées, composées de 32 sous-catégories. Dans le cadre de notre projet, les annotations des images devront être remappées avec une concentration sur 8 catégories principales.
- Laura (Système de Décision) : Laura requiert une API simple pour utiliser les résultats de la segmentation. L'API doit prendre en entrée une image et retourner la segmentation correspondante.

Cityscapes fournit des images urbaines diversifiées capturées dans des villes européennes, offrant un large éventail de scénarios urbains. Il est composé d'images segmentées et annotées, ce qui est essentiel pour l'entraînement de modèles de segmentation d'images précis.

Présentation des librairies

Segmentation_models est une bibliothèque Python conçue spécifiquement pour la segmentation d'images en deep learning. Elle offre une interface simplifiée pour construire des modèles de segmentation.Elle inclut des architectures populaires comme U-Net, FPN, et LinkNet avec des poids pré-entraînés sur des ensembles de données communs. Elle est conçue pour être utilisée avec Keras (et par extension TensorFlow), ce qui facilite son intégration dans les flux de travail.

Albumentations est une bibliothèque de traitement d'images et d'augmentation d'images rapide et flexible. Elle est optimisée pour le deep learning. Elle est conçue pour être rapide et efficace, ce qui est crucial lors du traitement de grands ensembles de données. Elle offre une large gamme de techniques d'augmentation d'images, y compris les changements de couleur, les distorsions spatiales, et les cropings.

TensorFlow est une bibliothèque open-source développée par Google pour le calcul numérique et le machine learning. Elle est particulièrement puissante pour le deep learning. Permet la construction et l'entraînement de modèles complexes de deep learning. Supporte le calcul sur CPU, GPU, et TPU. Elle est utilisée dans un large éventail de domaines, allant de la vision par ordinateur à l'analyse de séries temporelles, en passant par le traitement du langage naturel (NLP).

Ensemble, ces bibliothèques offrent un ensemble d'outils puissants permettant une construction et une expérimentation efficaces de modèles, en particulier dans le domaine de la vision par ordinateur et de la segmentation d'images.

Prétraitement des Données

La classe CityscapesGenerator, dérivée de tf.keras.utils.Sequence, joue un rôle essentiel dans le prétraitement des données du dataset Cityscapes. Cette classe est conçue pour gérer le mappage des 32 sous-catégories en 8 catégories principales, la taille des images et des lots, le mélange des données, et l'activation facultative des techniques d'augmentation des données. Les images et les labels sont chargés à partir de chemins définis, avec des listes d'identifiants et des dictionnaires de labels créés pour faciliter l'accès aux images et aux labels pendant la génération des lots.

L'augmentation des données, activée selon les besoins, inclut le retournement horizontal, les ajustements de couleur, et le flou. Ces techniques aident à renforcer la robustesse du modèle en augmentant la variété des données d'entraînement, essentielle pour la généralisation du modèle et la réduction du surajustement. L'activation facultative permettra de comparer les performances des différents modèles avec et sans augmentation de données.

La gestion des lots de données est assurée par la méthode `__getitem__` qui réduit l'utilisation de la mémoire en permettant d'extraire uniquement les informations essentielles au lieu de charger la totalité des données dans la mémoire. De plus, le mélange des données, réalisé à chaque fin d'époque, est crucial pour assurer que le modèle ne s'adapte pas à un ordre spécifique de données et pour améliorer la généralisation.

En résumé, la classe CityscapesGenerator incarne une composante clé du pipeline de traitement des données, influençant directement la qualité et l'efficacité de la segmentation d'images.
NB : La classe intègre également une fonction qui retourne et sauvegarde un échantillon d'images qui sera utile lors de la mise en

production du modèle.

In [ ]:
```python
class CityscapesGenerator(tf.keras.utils.Sequence):
    """
    Générateur de données pour le jeu de données Cityscapes, utilisé pour l'entraînement des modèles.

    Args:
        root_dir (str): Chemin du répertoire racine où se trouvent les données Cityscapes.
        batch_size (int): Taille du batch pour le traitement des données.
        img_size (tuple, optional): Dimensions des images (largeur, hauteur). Par défaut à (256, 256).
        data_cat (str, optional): Catégorie des données à utiliser ('train', 'val' ou 'test'). Par défaut à 'tra
        shuffle (bool, optional): Indique si les données doivent être mélangées à chaque époque. Par défaut à T
        data_augmentation (bool, optional): Indique si l'augmentation des données doit être appliquée. Par défa

    Attributes:
        img_size (tuple): Dimensions des images.
        batch_size (int): Taille du batch.
        shuffle (bool): Indicateur de mélange des données.
        img_root (str): Chemin du répertoire contenant les images.
        label_root (str): Chemin du répertoire contenant les étiquettes.
        list_IDs (list): Liste des chemins vers les images.
        labels (dict): Dictionnaire associant les chemins des images à leurs étiquettes.
        categories (dict): Dictionnaire des catégories d'objets dans les étiquettes.
        data_augmentation (bool): Indicateur d'augmentation des données.
        augmentor (albumentations.Compose): Pipeline d'augmentation des données (si activé).
        indexes (np.ndarray): Index des images à utiliser pour chaque batch.

    Return:
        Objet de la classe CityscapesGenerator.
    """
    def __init__(
        self,
        root_dir,
        batch_size,
        img_size=(256, 256),
        data_cat="train",
        shuffle=True,
        data_augmentation=False,
    ):
        self.img_size = img_size
        self.batch_size = batch_size
        self.shuffle = shuffle

        self.img_root = os.path.join(
            root_dir, f"P8_Cityscapes_leftImg8bit_trainvaltest/leftImg8bit/{data_cat}"
        )
        self.label_root = os.path.join(
            root_dir, f"P8_Cityscapes_gtFine_trainvaltest/gtFine/{data_cat}"
        )

        cities = os.listdir(self.img_root)
        self.list_IDs = []
        self.labels = {}

        for city in cities:
            img_dir = os.path.join(self.img_root, city)
            label_dir = os.path.join(self.label_root, city)

            img_names = sorted(
                [name for name in os.listdir(img_dir) if "_leftImg8bit.png" in name]
            )
            label_names = sorted(
                [
                    name.replace("_leftImg8bit.png", "_gtFine_labelIds.png")
                    for name in img_names
                ]
            )

            self.list_IDs.extend([os.path.join(img_dir, name) for name in img_names])
            for i, name in enumerate(img_names):
                self.labels[os.path.join(img_dir, name)] = os.path.join(
                    label_dir, label_names[i]
                )

        self.categories = {
            "void": [0, 1, 2, 3, 4, 5, 6],
            "flat": [7, 8, 9, 10],
            "construction": [11, 12, 13, 14, 15, 16],
            "object": [17, 18, 19, 20],
            "nature": [21, 22],
            "sky": [23],
            "human": [24, 25],
            "vehicle": [26, 27, 28, 29, 30, 31, 32, 33, -1],
```

```
        }

        self.data_augmentation = data_augmentation
        if self.data_augmentation:
            self.augmentor = A.Compose(
                [
                    A.HorizontalFlip(p=0.5),
                    A.ColorJitter(
                        brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2, p=0.5
                    ),
                    A.Blur(blur_limit=2, p=0.3),
                ]
            )

        self.on_epoch_end()

    def __len__(self):
        return int(np.floor(len(self.list_IDs) / self.batch_size))

    def __getitem__(self, index):
        indexes = self.indexes[index * self.batch_size : (index + 1) * self.batch_size]
        list_IDs_temp = [self.list_IDs[k] for k in indexes]
        X, y = self.__data_generation(list_IDs_temp)
        return X, y

    def on_epoch_end(self):
        self.indexes = np.arange(len(self.list_IDs))
        if self.shuffle:
            np.random.shuffle(self.indexes)

    def __data_generation(self, list_IDs_temp):
        X = np.empty((self.batch_size, *self.img_size, 3))
        y = np.empty((self.batch_size, *self.img_size, len(self.categories)))

        for i, ID in enumerate(list_IDs_temp):
            img = Image.open(ID).resize(self.img_size)
            label_img = Image.open(self.labels[ID]).resize(
                self.img_size, resample=Image.NEAREST
            )

            if self.data_augmentation:
                augmented = self.augmentor(
                    image=np.array(img), mask=np.array(label_img)
                )
                img = Image.fromarray((augmented["image"]).astype(np.uint8))
                label_img = Image.fromarray(augmented["mask"])

            label = np.array(label_img)
            X[i,] = np.array(img) / 255.0

            mask = np.zeros((*label.shape, len(self.categories)))
            for k, (cat_name, cat_ids) in enumerate(self.categories.items()):
                for cat_id in cat_ids:
                    mask[:, :, k] = np.logical_or(mask[:, :, k], (label == cat_id))

            y[i] = mask

        return X, y

    def compute_class_weights(self):
        pixel_counts = np.zeros(len(self.categories))
        total_pixels = 0

        for ID in self.list_IDs:
            label_img = Image.open(self.labels[ID]).resize(
                self.img_size, resample=Image.NEAREST
            )
            label = np.array(label_img)

            for k, (cat_name, cat_ids) in enumerate(self.categories.items()):
                for cat_id in cat_ids:
                    pixel_counts[k] += np.sum(label == cat_id)
            total_pixels += label.size

        class_weights = total_pixels / (len(self.categories) * pixel_counts)
        class_weights = class_weights / np.sum(class_weights)

        return class_weights

    def sample_data(self, num_samples=20, output_dir="static/api_sample"):
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)
        sample_images = random.sample(self.list_IDs, num_samples)
```

```
        for img_path in sample_images:
            label_path = self.labels[img_path]
            img = Image.open(img_path)
            label_img = Image.open(label_path)
            label = np.array(label_img)

            remapped_label = np.zeros(label.shape)
            for k, (cat_name, cat_ids) in enumerate(self.categories.items()):
                for cat_id in cat_ids:
                    remapped_label = np.where(label == cat_id, k, remapped_label)

            remapped_label_img = Image.fromarray(remapped_label.astype(np.uint8))
            img_save_path = os.path.join(output_dir, os.path.basename(img_path))
            label_save_path = os.path.join(output_dir, os.path.basename(label_path))

            img.save(img_save_path)
            remapped_label_img.save(label_save_path)
```

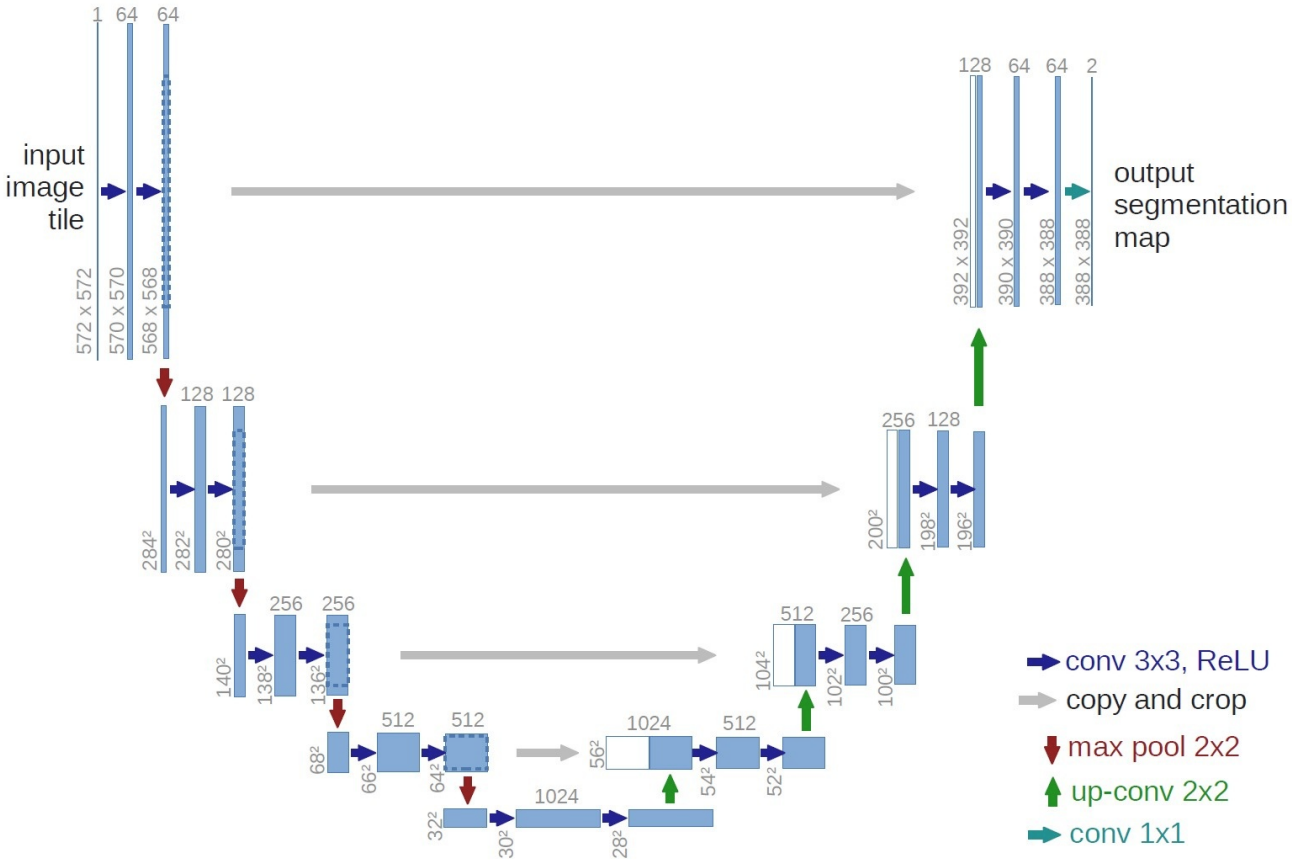
Sélection et configuration des modèles

En plein boom depuis quelques années grâce au deep learning, le domaine du computer vision et de la reconnaissance d'image a de plus en plus d'applications très concrètes. Du déverrouillage de votre iPhone à l'autopilot d'une Tesla, en passant par un logiciel qui classe automatiquement vos photos, les exemples sont nombreux.

Les premiers modèles de segmentation d'images utilisaient des architectures CNN basiques. Ces réseaux étaient bons pour identifier des caractéristiques visuelles simples mais limités pour comprendre le contexte spatial complexe.
C'est alors que des modèles spécialisés dans la segmentation d'images, chacun avec des caractéristiques et applications uniques, sont apparues.

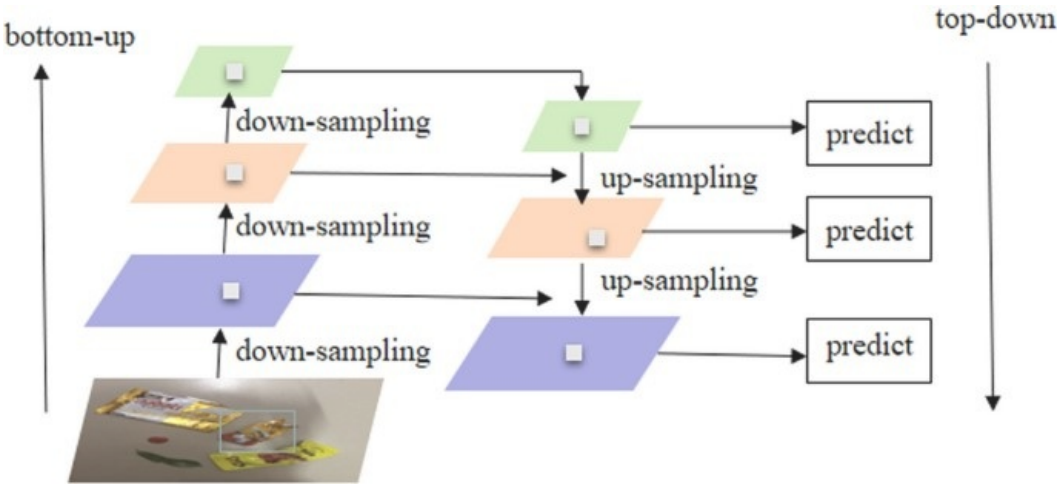
Choix des modèles :

- **U-Net** :
  - Origine et Utilisation: U-Net a été initialement développé pour la segmentation d'images biomédicales. Sa structure est conçue pour fonctionner efficacement avec un nombre limité de données d'entraînement, tout en produisant des résultats de segmentation précis.
  - Architecture: U-Net est caractérisé par sa structure en forme de "U", comprenant un chemin de contraction (encodeur) et un chemin d'expansion (décodeur). Cette conception permet à U-Net de capturer à la fois des informations contextuelles et localisées, ce qui est essentiel pour une segmentation précise.
  - Applications: U-Net excelle dans les tâches où les détails fins et la précision locale sont importants, comme la segmentation de structures cellulaires ou de tumeurs en imagerie médicale.



- **FPN** :
  - Conception pour la Détection d'Objets: FPN est souvent utilisé dans les tâches de détection d'objets et de segmentation

sémantique. Il est conçu pour améliorer la qualité de la segmentation en intégrant des informations à différentes échelles.
- Pyramide de Caractéristiques: L'architecture FPN créé une pyramide de caractéristiques à plusieurs niveaux, ce qui permet de capturer des informations à la fois fines et grossières, rendant le modèle efficace pour détecter des objets de différentes tailles.
- Applications: Les FPN sont largement utilisés dans des tâches telles que la détection d'objets et la segmentation d'instances, où la capacité à gérer des objets de différentes tailles est essentielle.



Les modèles ont été personnalisés pour s'adapter aux spécificités du dataset Cityscapes et aux exigences du projet. J'ai utilisé une version allégée d'U-Net, `Unet_mini`, comme modèle de référence. Cette version utilise des blocs d'encodage et de décodage simplifiés, tout en conservant l'efficacité de la structure U-Net originale. Ce modèle servira de baseline afin d'avoir un point de départ dans la comparaison des performances.

Les modèles U-net et FPN ont été entrainés avec et sans augmentation de données afin de pouvoir faire un comparatif des performances. Des stratégies d'entraînement et d'optimisation, y compris le choix de fonctions de perte et de métriques spécifiques, ont été appliquées pour affiner les modèles pour de meilleures performances. De plus, n'ayant qu'un nombre limité de données à utiliser pour l'entrainement du modèle (2975 images pour le set d'entrainement), nous avons utilisé l'apprentissage par transfert (Transfer Learning) qui permet l'utilisation de modèles pré-entrainés sur de grandes bases de données et leur ajustement pour des tâches spécifiques.

Choix des métriques et fonction de perte

Le choix des métriques et de la fonction de perte dans un projet de segmentation d'images est crucial pour évaluer et optimiser la performance des modèles de deep learning.

La fonction de perte, également appelée fonction coût ou fonction d'erreur, mesure à quel point les prédictions du modèle diffèrent des valeurs réelles (ou valeurs cibles) pendant l'entraînement.
L'objectif principal de la fonction de perte est de guider l'optimisation du modèle.
La perte de Dice est une fonction de perte utilisée pour minimiser la différence entre les prédictions du modèle et les vérités terrain. Elle est particulièrement adaptée à la segmentation d'images, en particulier dans des contextes où il y a un déséquilibre de classe.
J'ai opté pour la **perte de Dice (Dice loss)** en utilisant des poids de classe pour gérer l'inégalité dans la représentation des différentes classes dans le dataset. Cela permet au modèle de prêter plus d'attention aux classes sous-représentées, évitant ainsi que le modèle ne soit biaisé en faveur des classes majoritaires.

Les métriques sont utilisées pour évaluer la performance d'un modèle. Elles sont conçues pour être interprétables et pour refléter les objectifs de la tâche en question. Les métriques ne sont pas utilisées pour l'entraînement du modèle mais pour surveiller et évaluer sa performance. Contrairement aux fonctions de perte, elles ne sont pas directement impliquées dans le processus d'optimisation.
J'ai opté pour **l'indice de Jaccard (IOU Score)** et le **coefficient de Dice (F1 Score)**.
L'indice de Jaccard, également connu sous le nom de IOU (Intersection Over Union), est une mesure statistique de la similarité entre deux ensembles. Dans le contexte de la segmentation d'images, il évalue la précision en comparant la zone de chevauchement entre la prédiction du modèle (segmentation) et la vérité terrain (label). L'IOU est particulièrement adapté pour les projets de segmentation car il fournit une évaluation directe et intuitive de la qualité de la segmentation. Une valeur élevée d'IOU indique que la segmentation prédite par le modèle correspond étroitement à la segmentation réelle.
Le coefficient de Dice, ou F1 Score, est une mesure de la précision qui prend en compte à la fois la précision (proportion des identifications positives correctes) et la sensibilité (capacité à identifier correctement les vrais positifs). Cette métrique est particulièrement utile dans les cas où il y a un déséquilibre entre les classes de pixels (par exemple, une classe beaucoup plus présente que les autres).

Choix des backbones

- **ResNet50** :
    - Conception: ResNet50 est une variante du modèle ResNet (Residual Network) avec 50 couches. L'innovation clé de ResNet est l'introduction de connexions résiduelles ou de "skip connections".

- Skip Connections: Ces connexions permettent au signal de contourner une ou plusieurs couches, ce qui aide à combattre le problème de la disparition du gradient dans les réseaux très profonds, en permettant un flux plus direct du gradient lors de l'apprentissage.
- Utilisations: ResNet50 est largement utilisé pour la classification d'images, la détection d'objets, et la segmentation, en raison de sa capacité à apprendre des caractéristiques riches et complexes sur de grandes profondeurs de réseau.

- **VGG16** :
  - Conception: VGG16, développé par l'Université d'Oxford, est un modèle CNN composé de 16 couches (13 couches convolutives et 3 couches entièrement connectées). Il est connu pour sa structure simple mais efficace.
  - Architecture Uniforme: VGG16 utilise une architecture uniforme avec des couches convolutives de petits filtres (3x3) suivies de couches de pooling. Cette approche uniforme facilite la compréhension et l'entraînement du modèle.
  - Utilisations: Malgré sa simplicité relative, VGG16 est efficace pour l'extraction de caractéristiques en vision par ordinateur, y compris la classification et la détection d'images.
- **EfficientNetB3** :
  - Conception: EfficientNet est une architecture de réseau neuronal convolutionnel et une méthode de mise à l'échelle qui met uniformément à l'échelle toutes les dimensions de profondeur/largeur/résolution à l'aide d'un coefficient composé.
  - Optimisation Multi-Dimensionnelle: Contrairement à la pratique conventionnelle qui échelonne arbitrairement ces facteurs, la méthode d'échelonnement EfficientNet échelonne uniformément la largeur, la profondeur et la résolution du réseau à l'aide d'un ensemble de coefficients d'échelonnement fixes.
  - Utilisations: EfficientNetB3 est utilisé dans diverses applications de vision par ordinateur, en particulier là où l'efficacité en termes de paramètres et de calcul est une priorité, sans sacrifier les performances.

Exemple d'un entrainement de modèle

In [ ]:
```
model_name = "_Unet_resnet50"
num_epochs = 30
model = sm.Unet(
    "resnet50",
    input_shape=(256, 256, 3),
    classes=len(train_gen.categories),
    activation="softmax",
)
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S") + model_name
visualize_callback = VisualizeCallback(
    generator=train_gen, model_name=model_name, log_dir=log_dir
)
checkpoint_callback = ModelCheckpoint(
    "saved_model/Unet_resnet50",
    save_best_only=True,
    monitor="val_iou_score",
    mode="max",
)
tensorboard_callback = TensorBoard(log_dir=log_dir, write_graph=True, write_images=True)
early_stopping = EarlyStopping(monitor="val_iou_score", patience=3, mode="max")
model.compile(optimizer="adam", loss=dice_loss, metrics=[jaccard_index, dice_coef])
history = model.fit(
    train_gen,
    epochs=num_epochs,
    validation_data=val_gen,
    callbacks=[
        checkpoint_callback,
        tensorboard_callback,
        early_stopping,
        visualize_callback,
    ],
)
scores = model.evaluate(val_gen, verbose=0)
print("Loss:", scores[0])
print("IOU score:", scores[1])
print("F1 score:", scores[2])
plot_history(history)
```

Analyse des performances

Tableau comparatif des résultats :

	Unet-mini	U-net + resnet50	U-net + vgg-16	U-net + efficientnetb3	U-net + efficientnetb3+ albumentation
IOU score	0.4035	0.6543	0.7162	0.7406	0.7416
F1 score	0.5536	0.7709	0.8221	0.8405	0.8402
Dice loss	0.9423	0.9190	0.9089	0.9053	0.9055

Performances des Modèles U-Net :

- **U-Net Mini**: Cette version simplifiée présente des scores IOU et F1 inférieurs et une perte de Dice plus élevée. Cette performance est attendue compte tenu de sa conception simplifiée, ciblant les environnements à ressources limitées, et représente un compromis entre efficacité et performance.
- **U-Net avec ResNet50**: L'incorporation du backbone ResNet50 dans U-Net entraîne une amélioration notable de la précision de segmentation par rapport à U-Net Mini, illustrant l'impact positif d'un backbone plus robuste.
- **U-Net avec VGG-16**: Cette configuration affiche des performances supérieures à ResNet50, bien que VGG-16 soit le modèle le plus ancien, il se montre particulièrement efficace dans la segmentation d'images.
- **U-Net avec EfficientNetB3**: EfficientNetB3 affiche des performances supérieures. Cela met en lumière l'efficacité des backbones modernes pour capturer des caractéristiques complexes, essentielles à une segmentation précise des images.
- **Impact de l'Augmentation des Données**: L'utilisation d'Albumentation avec U-Net et EfficientNetB3 montre une amélioration marginale mais notable des scores IOU et F1, indiquant que l'augmentation des données peut renforcer la généralisation des modèles.

	FPN + efficientnetb3	FPN + efficientnetb3 + albumentation
IOU score	0.7382	0.7441
F1 score	0.8381	0.8419
Dice loss	0.9060	0.9054

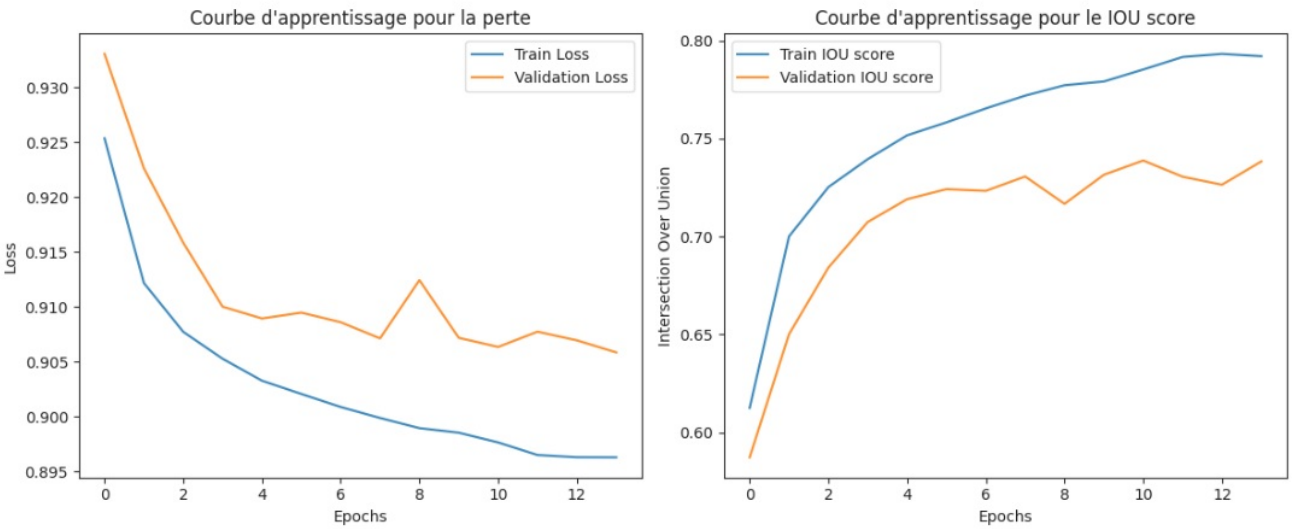
Performances des Modèles FPN

- **FPN avec EfficientNetB3**: Cette configuration offre des résultats compétitifs, légèrement supérieurs à ceux d'U-Net utilisant le même backbone. Cela confirme l'efficacité de l'architecture FPN, qui intègre des informations à différentes échelles, pour la tâche de segmentation.
- **Effet de l'Augmentation des Données sur FPN**: Comme observé avec U-Net, l'ajout d'Albumentation à FPN avec EfficientNetB3 entraîne une légère amélioration des performances, soulignant l'importance de l'augmentation des données pour améliorer la précision et la robustesse des modèles de segmentation.

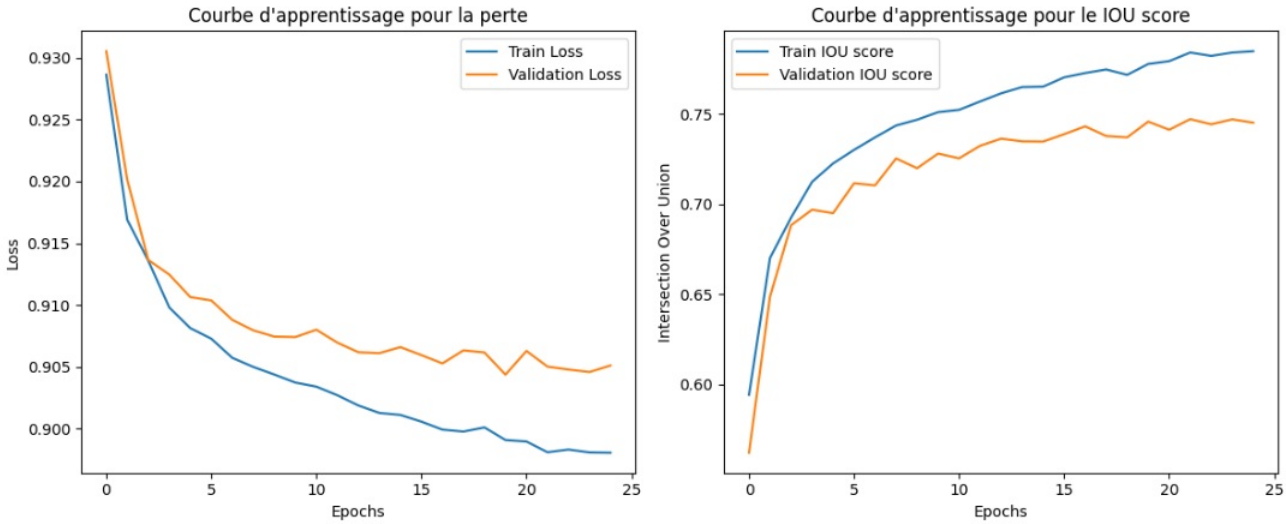
Visualisation de l'impact de l'Augmentation des données

Les résultats des tests de segmentation d'images montrent des tendances intéressantes et significatives en ce qui concerne les performances des différentes configurations de modèles U-Net et FPN. Nous avons observé que les résultats étaient légèrement supérieurs en utilisant l'augmentation de données. Bien que la différence de performance ne soit pas phénoménale, on observe tout de même une meilleure généralisation de l'apprentissage lors de l'utilisation d'albumentations :

FPN - efficientnetb3 :



FPN - efficientnetb3 - Augmentation de données :

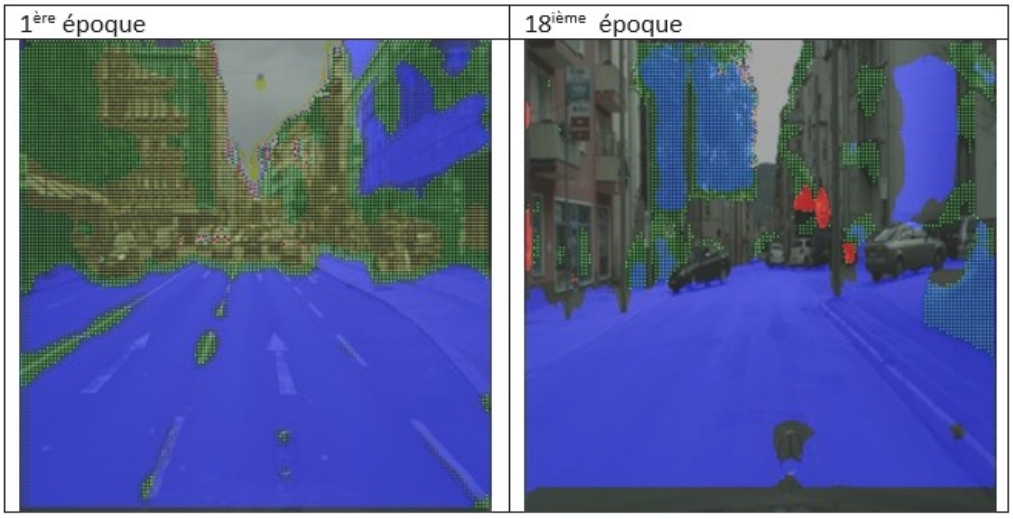


En effet, les courbes d'apprentissage et de validation sont plus proches lors de l'utilisation d'albumentations. Ce qui signifie que le modèle est capable de mieux générer sur des données non vues.

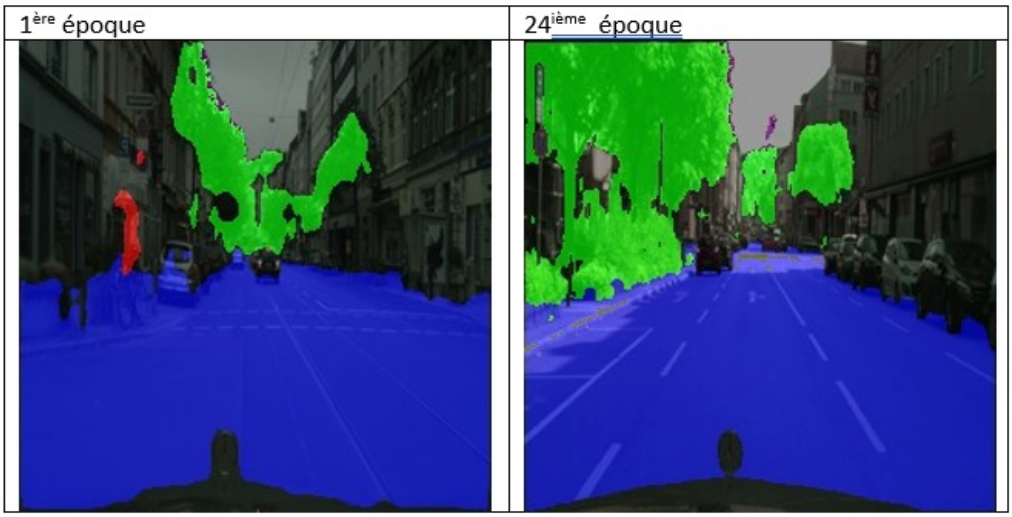
Visualisation de la segmentation d'images du modèle

Maintenant que nous avons une idée des performances de nos modèles à l'aide de mesures statistiques, nous allons visualiser les prédictions des modèles au fur et à mesure de l'entrainement de ceux-ci. Pour ce faire, j'ai implémenté une classe de rappel (callback) en Python, utilisée avec TensorFlow pour la visualisation des résultats de segmentation d'images à la fin de chaque époque pendant l'entraînement. Cette classe est conçue pour générer des images de sortie à la fin de chaque époque d'entraînement, montrant comment le modèle segmente une image spécifique. La couleur des masques est définie en les assignant à différentes couches (canaux) de l'image RGB. Le masque 'nature' est assigné au canal vert, le masque 'flat' au canal bleu et le masque 'human' au canal rouge. L'image résultante montre donc ces classes en superposant ces couleurs sur l'image originale.

Prédictions de notre moins bon modèle (Unet-mini) :



Prédictions de notre meilleur modèle (FPN - efficientnetb3 - albumentations) :



Bien que les mesures statistiques soient utiles pour déterminer les performances d'un modèle, ici nous pouvons en juger par nous même en comparant les images générées lors de l'entrainement des deux modèles.

Conclusion et Perspectives d'Amélioration

Dans ce projet, nous avons franchi des étapes importantes en développant et optimisant plusieurs modèles de segmentation d'images, spécifiquement adaptés aux systèmes de vision par ordinateur pour véhicules autonomes. En utilisant des architectures telles que U-Net et FPN, combinées à des backbones variés comme ResNet50, VGG-16 et EfficientNetB3, nous avons atteint un haut niveau de précision en segmentation.

Pour s'adapter aux limites des ressources computationnelles, nous avons choisi un batch size de 32 et des résolutions d'images de 256x256 pixels, réduisant ainsi la consommation de mémoire pendant l'entraînement. Les backbones comme EfficientNetB3 ont été sélectionnés pour leur bon équilibre entre taille et performance. Dans la perspective d'une amélioration continue, nous envisageons d'utiliser des images de plus grande taille pour conserver plus d'informations, tout en expérimentant avec des backbones plus volumineux comme EfficientNetB7.

Concernant l'évaluation des modèles, nous avons utilisé le set de validation car le set de test ne disposait pas de masques pour l'évaluation. Idéalement, il faudrait avoir trois sets de données distincts (entraînement, validation et test) pour être dans la norme. Une option aurait été de séparer certaines images du set d'entraînement pour créer notre propre set de test. Cependant, cela aurait réduit notre ensemble de données d'entraînement déjà limité. Cette décision souligne un compromis, où la quantité et la qualité des données disponibles peuvent directement influencer les choix méthodologiques et les résultats.

Déploiement du Modèle

Notre API Flask est structurée pour gérer les requêtes de segmentation d'images. Elle reçoit des images envoyées par les utilisateurs, les traite, et renvoie les résultats de la segmentation. Les images reçues sont redimensionnées et prétraitées avant d'être soumises au modèle de segmentation pour prédiction. Le modèle de segmentation d'images est stocké sur Azure Blob Service, un service de stockage d'objets dans le cloud de Microsoft Azure.
Des mesures sont prises pour sécuriser l'API, notamment en limitant la taille des fichiers reçus et en validant les formats d'image pris en charge. La gestion des erreurs est assurée par des réponses appropriées en cas d'exceptions ou de données inadéquates envoyées à l'API.

L'interface utilisateur de l'application Flask permet aux utilisateurs de sélectionner et d'envoyer des images pour segmentation. Elle affiche ensuite les images originales et les résultats de la segmentation, permettant une comparaison visuelle directe. L'application interagit avec l'API Flask en envoyant des images sélectionnées et en recevant les masques de segmentation prédits. La mise en œuvre de l'API Flask et de l'interface utilisateur reflète une intégration de l'intelligence artificielle dans des applications web pratiques, rendant la technologie de segmentation d'images accessible et utilisable pour le système de décision de Laura.

L'utilisation de GitHub Actions a joué un rôle crucial dans la mise en place d'un pipeline d'Intégration Continue et de Déploiement Continu (CI/CD) pour le développement de l'API et de l'interface utilisateur. Cette approche a facilité une intégration transparente entre le dépôt GitHub et le serveur Cloud Azure, assurant ainsi une coordination efficace entre le développement du code et son déploiement.