Note technique

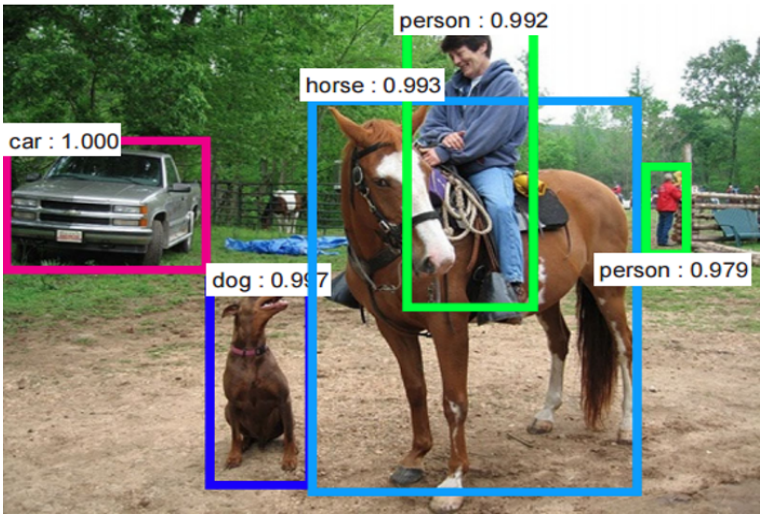
Segmentation :

Le principe de la segmentation d'image est de classifier chaque pixel d'une image en lui attribuant une des classes qu'on définit préalablement.
On cherche par exemple, à savoir quels pixels représentent une personne, un bâtiment, de la verdure, etc...

Input

Semantic Labels

Ce concept est différent de la détection d'objet, qui utilise des zone de prédictions (bounding box).



La segmentation est notamment utilisée sur des images satellites pour de la détection de bâtiments, de la surveillance ou du

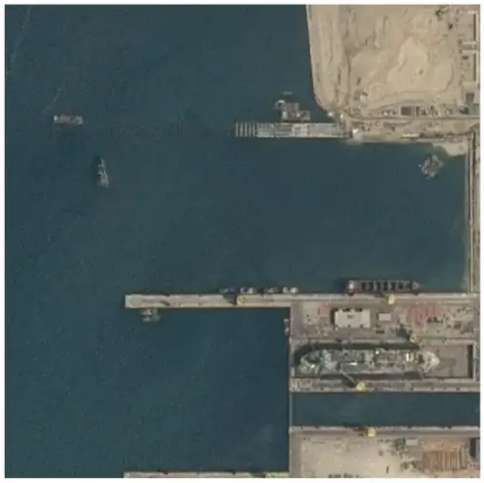
répertoriage du territoire.

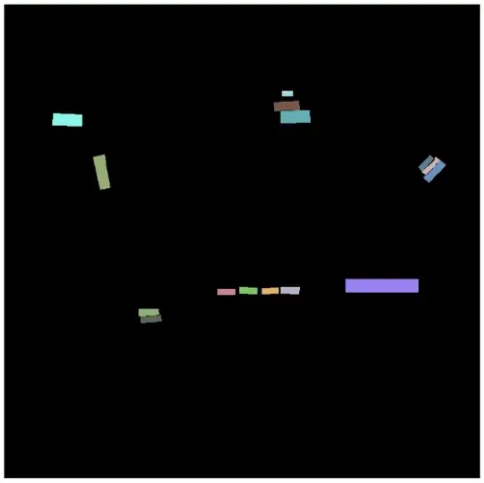
Dans notre cas, l'application sera la voiture autonome.
Le but est de créer un modèle qui génère une version simplifiée de l'image qui lui est donnée.
L'entrée du modèle est une image classique RVB, la sortie sera un masque de l'image où on associe chaque pixel à une route, un véhicule, un piéton, etc..

Métriques :

Pour juger de la qualité d'une prédiction en machine learning, on utilise une métrique adaptée au problème qui permet de facilement visualiser la performance de notre modèle.
On pourrait utiliser comme métrique la fonction de perte du modèle en question (entropie croisée), mais étant simplement une quantité à minimiser, cette méthode n'est pas très pertinente : sans autre point de référence, comment savoir si une perte donnée (par exemple 0.0568) est une performance satisfaisant ?
On peut seulement comparer ce résultat au précédents pour constater la différence.

La précision par pixel est une métrique plus intuitive : il s'agit du pourcentage de pixels bien classifiés. Le score est donc compris entre 0 et 1 (de 0% à 100%).
Cependant cette métrique a un défaut majeur, elle ne prend pas compte de la répartition des classes sur l'image.
Prenons en exemple une image et un masque qui représente en les bateaux en couleur et le reste en noir :

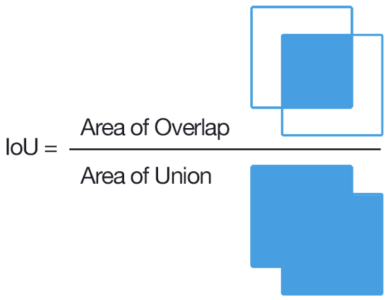




Si on segmente cette image avec un modèle et qu'il prédit un masque entière noir, il obtiendra une très bonne précision par pixel (environ 0,95) car le vrai masque est presque complètement noir.

Il existe des métriques plus adaptées à la segmentation comme par exemples l'IoU (Intersection over Union) et le coefficient Dice (ou l'indice de Sørensen-Dice).

L'IoU permet de quantifier, pour une classes, à quel point la région prédite chevauche la région véritable.
En définissant sur une image une zone A où une classe est présente et une zone de prédiction B (qui tente de copier la zone A), alors on peut décrire l'équation de l'IoU comme étant l'intersection des zones A et B divisée par l'union des zones A et B.

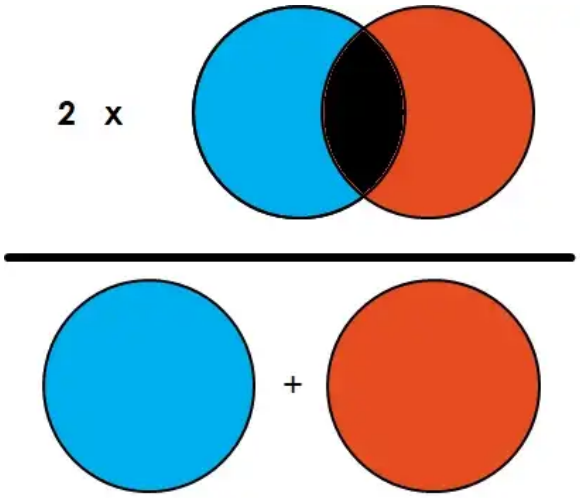


Sous forme d'équation :
$IoU = A \cap B / A \cup B$

Ou en termes de matrice de confusion :
IoU = TP / (TP + FP + FN)

Pour un problème avec plusieurs classes, on calcule le score de chaque classe, ce qui permet de suivre précisément les performance du modèle. On peut ensuite agréger ces scores en un seul pour notre métrique finale. Prendre la moyennes des scores permet de donner la même valeur à toutes les classes, on peut aussi pondérer chaque classe en fonction du problème (par exemple donner plus d'importance aux piétons et moins à la végétation).

Le coefficient de Dice est assez semblable à l'IoU :



Dice = 2 * A ∩B / (A + B)
ou
Dice = 2*TP / (2*TP + FP + FN)

Le concept reste le même et ces deux métriques sont positivement corrélées. La différence principale est dans la notation : comme point de comparaison, le coefficient Dice est plus semblable à une pénalité L1 alors que l'IoU à une pénalité L2.

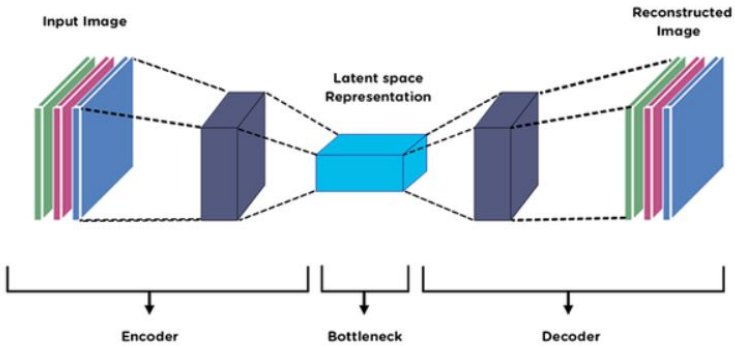
C'est-à-dire que Dice pénalise toutes les erreurs de manière linéaire, alors que l'IoU pénalise plus fortement les pires cas.

Le coefficient Dice privilégie donc un modèle avec l'erreur moyenne la plus basse, même si il produit parfois de très mauvaises prédictions.
L'IoU privilégie un modèle qui ne commet pas d'erreurs graves, même si il est légèrement moins performant en moyenne.

J'ai choisi d'utiliser l'IoU moyenne comme métrique pour ce projet.
Le sujet étant les voitures autonomes, je préfère un modèle plus "prudent" car je considère qu'une prédiction radicalement différente de la réalité pourrait avoir des conséquences graves en pratique.

Architecture des modèles :

Les modèles de segmentation son généralement des réseaux de neurones convolutifs (CNN) présentant une architecture de type encodeur / décodeur.



L'encodeur est la partie du réseau qui applique des filtres sur l'image d'entrée pour en extraire les caractéristiques (sous forme de vecteurs incompréhensibles pour un humain). Plus l'image

avance dans l'encodeur, plus sa taille se réduit (par les filtres ou des couches de max-pooling) et plus elle gagne en profondeur.

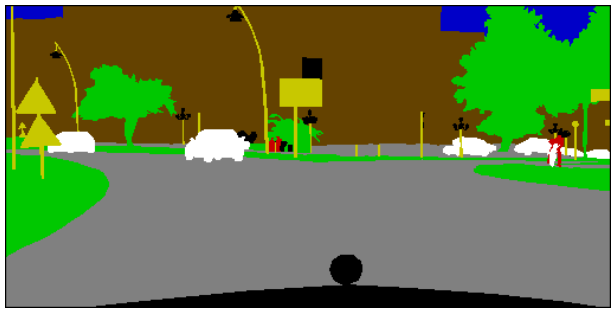
Une image en couleur a une profondeur de 3 pour représenter les canaux RVB, une image en niveaux de gris a une profondeur de 1. Une image en couleur de taille 500 x 500 pixels est représenté comme une matrice de dimensions (500, 500, 3).
En sortie d'un encodeur, cette image pourrait donner, par exemple, une matrice de dimensions (20, 20, 256).

Cette matrice constitue l'entrée de la deuxième partie du réseau : le décodeur.
Le but du décodeur, dans notre cas, est de transformer la matrice de sortie de l'encodeur en masque de prédiction, c'est-à-dire une matrice de la même taille que l'image originelle qui associe à chaque pixel une prédiction de classe.

Cette prédiction peu se présenter sous la forme d'un seul chiffre qui est le numéro de la classe sélectionnée, ou comme une liste qui contient une prédiction pour chaque classe. J'ai opté pour la seconde méthode

Ce projet comporte 8 classes :
void, flat, construction, object, nature, sky, human, vehicle.
La matrice de sortie est donc de taille (500, 500, 8) qui contient des valeur de prédiction (entre 0 et 1).
Cela permet une analyse plus fine, et il suffit de sélectionner sur chaque pixel la plus grande valeur pour obtenir la classe prédite (on obtient donc une matrice (500, 500, 1)
En appliquant ce procédé et en colorant chaque classe, on obtient un visuel facilement interprétable :

image d'origine



masque

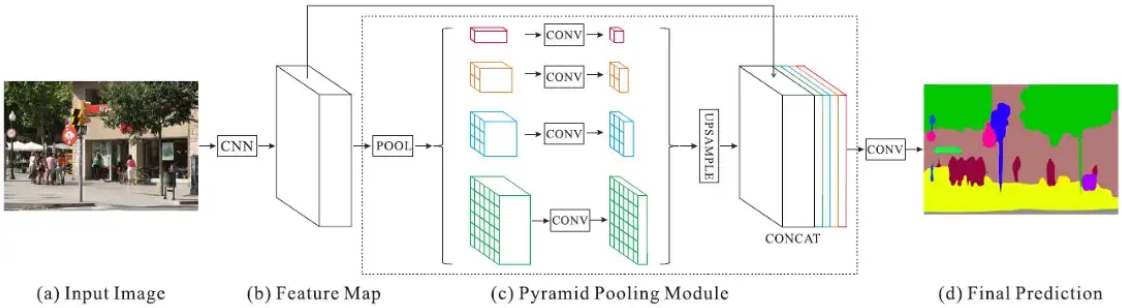
<u>Modèles souvent utilisés :</u>

Certains modèles sont plus fréquemment utilisés, que ce soit parce qu'ils ont été les plus performants au temps de leur publication, qu'ils présentent des avantages particuliers comme une faible complexité, ou par convention.

Parmi les plus connus on peut citer PSPNet, U-Net et Segnet.

PSPNet :

Architecture encodeur classique, décodeur avec "pyramid pooling module".



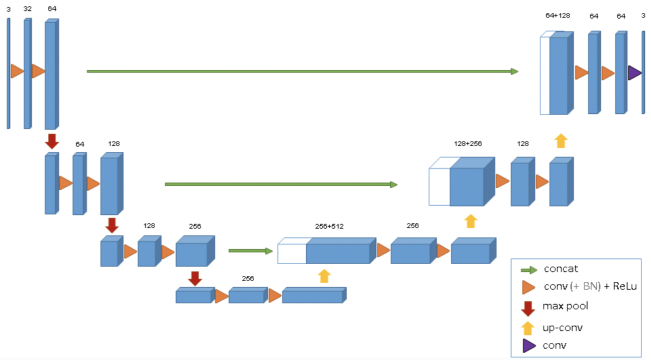
" pyramid pooling module" :

Plusieurs couches convolutives de différentes tailles qui prennent chacune en entrée la sortie de l'encodeur

("feature map" sur l'image ci-dessus).
Ces couches sont ensuite ajoutées à la "feature map" pour constituer la matrice finale qui passe par un dernier bloc convolutif pour donner la prédiction.

U-Net (Modèle utilisé dans ce projet) :

Architecture encodeur / décodeur avec "skip connexions" et "transposed convolution".



"skip connexions" :
Avant chaque couche du décodeur, on combine l'entrée de la couche du décodeur à la sortie de la couche de l'encodeur à ce niveau. Donc en plus de la matrice compressée en sortie de l'encodeur, chaque sortie de couche de l'encodeur est donnée directement au décodeur sans passer par ladite compression.

"transposed convolution" :
Méthode d'upsampling convolutive avec apprentissage de paramètres (en quelque sorte l'inverse d'une couche convolutive classique).

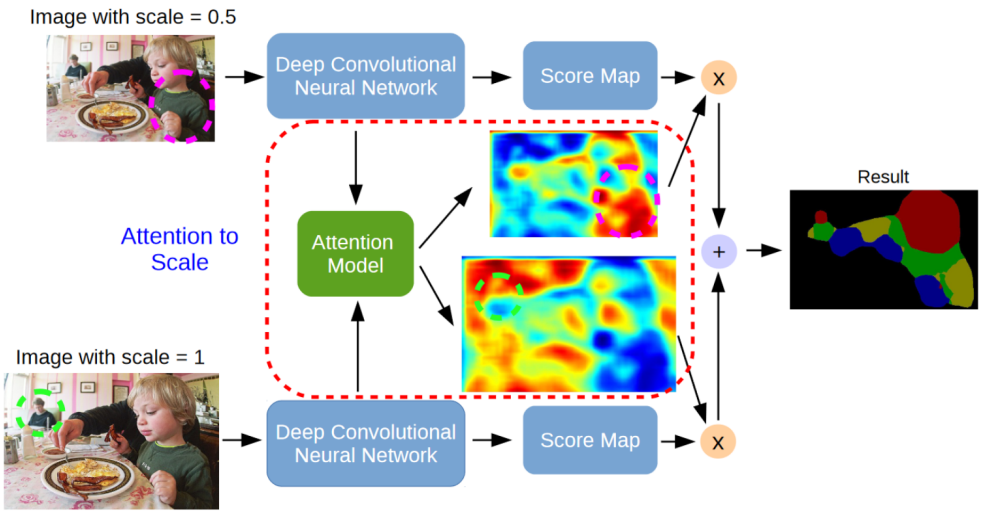
Segnet :

Identique à U-Net mais ne possède pas de "skip connexions" ni de "transposed convolution".
Les couches d'upsampling reçoivent les positions des éléments choisis par le max-pooling à la couche d'encodeur correspondante.
Cela réduit la complexité et la taille en mémoire du modèle.

État de l'art :

L'état de l'art est encore en évolution dans ce domaine, comme le montre les performances du modèle HRNet (publication en 2019).

HRNet est un type de modèle qui utilise un mécanisme d'attention.



Au lieu d'une seule image en entrée, le modèle en reçoit trois : l'image originelle, l'image à l'échelle 0.5 et une à l'échelle 2.
Cela permet de choisir entre une vision simplifiée ou précise pour chaque zone de l'image.

Voici un lien pour voir le mécanisme d'attention en pratique :
https://www.youtube.com/watch?v=odAGA7pFBGA

Pour plus d'information sur l'état de l'art actuel, cette page recense les meilleurs performances par date sur le jeu de données que ce projet utilise (Cityscape) :
https://paperswithcode.com/sota/semantic-segmentation-on-cityscapes-val

Transfer learning :

Notre jeu de données comporte 3475 images (2975 pour l'entraînement et 500 pour la validation), ce qui est assez peu pour un projet de deep learning.
J'ai sélectionné deux moyens pour remédier à ce problème :
le transfer learning et l'augmentation de donnée.

Tous les modèles cités précédemment n'ont pas d'encodeurs spécifiques, ce qui veut dire qu'il est possible d'utiliser comme encodeur des modèles pré-entraînés sur d'autres jeux de données. Grâce au pré-entraînement, on gagne en temps d'entraînement et en performance.
Des modèles de type VGG ou ResNet entraînés sur le jeu de données ImageNet sont souvent utilisés, c'est donc ceux que j'ai implémenté.

Augmentation des données :

L'augmentation de donnée consiste à générer des images artificielles à partir de vrais images. On peut donc agrandir le nombre d'image à notre disposition pour l'entraînement, ce qui permet au modèle de mieux généraliser, réduire le sur-apprentissage et d'obtenir de meilleurs résultats à la validation.

Parmi les transformations d'image classiques, on peut citer la translation, la rotation et le recadrage.
Il est aussi possible de modifier la valeur des pixels en rendant l'image floue, en changeant le contraste, la luminosité, etc...
J'ai aussi testé la simulation de pluie et de neige.

En pratique, je paramètre le nombre d'images augmentées par image réelle et la probabilité d'application 'p' de chaque type d'augmentation.
Par exemple, pour chaque image d'origine, une image augmentée sera créée avec comme augmentations potentielles la rotation (p=0.2), le recadrage (p=0.5) et le retournement horizontal (p=1).

image d'origine

image augmentée

<u>Résultats :</u>

Voici un tableau qui contient les résultats les plus pertinents :

model_name	img_size	n_augments	img_augments	mean_IoU	class_IoU	epoch_train_time
U-Net pretrained vgg16	(256, 512)	1	{'h_flip': 1, 'crop': 0.2}	0.625	[0.691 0.904 0.718 0.298 0.767 0.667 0.3 0.658]	643.0
U-Net pretrained vgg16	(256, 512)	0	{}	0.621	[0.693 0.904 0.713 0.293 0.782 0.679 0.279 0.621]	547.0
U-Net pretrained vgg16	(256, 512)	1	{'h_flip': 1, 'rotate': 0.2}	0.615	[0.687 0.902 0.711 0.25 0.773 0.66 0.275 0.657]	665.0
U-Net pretrained resnet50	(256, 512)	1	{'h_flip': 1, 'crop': 0.2, 'rotate': 0.2}	0.475	[0.594 0.748 0.542 0.226 0.733 0.276 0.124 0.555]	656.0
U-Net pretrained resnet50	(256, 512)	0	{}	0.455	[0.524 0.779 0.553 0.16 0.658 0.271 0.205 0.49]	546.0
U-Net	(512, 1024)	0	{}	0.592	[0.761 0.928 0.676 0.212 0.72 0.658 0.249 0.532]	1140.0
U-Net	(256, 512)	0	{}	0.473	[0.677 0.867 0.6 0.028 0.663 0.314 0.117 0.515]	500.0
U-Net	(128, 128)	0	{}	0.390	[0.518 0.809 0.52 0. 0.449 0.446 0.021 0.36]	370.0

"class_IoU" désigne le score de chaque classe :

void, flat, construction, object, nature, sky, human, vehicle.

D'après les trois dernières ligne du tableau, le moyen le plus simple d'augmenter les performances est d'utiliser la plus grande taille d'image possible. Si on donne à nos modèles des données d'entrée plus précises, il semble logique que la prédiction soit meilleure.

L'augmentation d'image apporte une légère amélioration du score si utilisé avec parcimonie : une image artificielle par image originelle est bénéfique, mais à partir de deux images artificielles les prédictions sur le jeu de validation deviennent très mauvaises. Le jeu d'entraînement étant très similaire au jeu de validation, trop d'images transformées à l'entraînement ne font que perturber le modèle.
J'émets l'hypothèse qu'avec un jeu de test plus varié (qualité d'image, heure de la journée, saison), l'augmentation d'image serai encore plus utile.

Pour ce qui est du transfer learning, en regardant le tableau on voit rapidement que VGG16 est bien plus performant que RestNet50.

<u>Améliorations :</u>

Pour conclure, voici une liste de différentes aspects du projet qui pourraient être modifiés pour obtenir de meilleurs résultats :

- Résolution des images :
    N'ayant pas accès à assez de mémoire vive, j'ai dû réduire la taille des images ce qui entraîne une perte d'information. L'idéal serait de conserver la taille originelle de l'image.

- Tester d'autres fonctions de perte :

  La seule fonction de perte que j'ai utilisé dans ce projet est l'entropie croisée. J'ai ensuite appris que cette fonction possède analogue à la précision par pixel : elle ne tient pas compte du déséquilibre des classes.
  Cela pourrait expliquer la mauvaise performance de tous mes modèles sur la classe "humain". Je recommande donc des tests avec d'autres pertes (weighted cross-entropy, IoU loss, boudary loss).

- Tester d'autres modèles :

  J'ai choisi de me concentrer sur U-Net, mais des essais avec les modèles que j'ai présentés précédemment (SegNet, PSPNet, HRNet) donneront sûrement une amélioration des résultats.

- Ajouter des canaux supplémentaires aux images :

  Sur une images satellite, chaque pixel peut se voir associé une valeur d'élévation par rapport au sol, ce qui permettrait à un modèle d'identifier un bâtiment plus facilement.
  Dans notre cas, j'aurais pu ajouter à chaque image un canal supplémentaire correspondant à son niveau de gris.

- Gestion de la classe "void" :

  Cette classe englobe beaucoup de choses différentes, que se soit des objets non permanents (poubelles, sac, etc...) ou d'autres objets non couvert par les autres catégories (partie du véhicule égo, dos d'un panneaux, rond point, étendue d'eau).
  Dans ce projet j'ai considéré "void" comme une classe normale, mais sur le site de CityScape il est précisé que "les pixels classifiés void ne contribuent pas au score".
  Retirer cette contrainte pourrait faciliter l'apprentissage.