**Note technique : Segmentation d’Images**

I – Introduction

La segmentation d’images est un sous-domaine du traitement d’images. Il s’agit d’une discipline mêlant informatique et mathématiques appliquées (en particulier traitement du signal, les statistiques, et les probabilités).

Le principe de base est de rassembler des pixels selon des classes suivant des critères prédéfinis (donc d’associer chaque pixel à une *classe*). Les pixels connexes appartenant à une même classe constituent une *région* de l’image. La séparation entre deux régions est appelée *frontière*.

L’objectif des méthodes de segmentation d’images est donc d’identifier certains objets ou zones d’intérêt sur des images.

Ces méthodes sont nombreuses et le but de cette note n’est pas de traiter le sujet de manière exhaustive.

II - Méthodes « traditionnelles »

Avant la popularisation du Deep Learning dans l’industrie, aux alentours de 2015, les méthodes de segmentation d’images principales étaient réparties selon trois familles. On note qu’il existe des modèles combinant des méthodes de différentes familles.

A) Segmentation par approche régions

La segmentation par approche région est la famille d’algorithmes de segmentation qui considèrent les régions de l’image. Deux sous-approches sont possibles.

Soit on part d’une image partitionnée arbitrairement, puis on regroupe et on divise itérativement ses régions (*split and merge*), jusqu’à ce qu’une condition prédéfinie soit remplie (par exemple un nombre de régions, ou encore la différence de luminosité ou de contraste entre les couples de régions).

Soit on part de petites régions qui vont croître en incorporant progressivement de nouveaux pixels (*croissance de régions*). Les régions initiales peuvent être calculées automatiquement (par exemple en utilisant les minimas de l’image), ou fournie par un utilisateur. Les régions grandissent ensuite en incorporant les pixels les plus similaires suivant un critère donné, comme la différence de niveaux de gris.

Il existe également des méthodes plus avancées qui permettent de prendre en compte la régularité des régions (c’est-à-dire l’apparence globale de la segmentation) en plus des paramètres mentionnés au-dessus. La plupart d’entre elles s’appuient sur le Modèle de Markov Caché, un modèle statistique stochastique permettant d’associer les pixels selon des contraintes multiples.

1. Segmentation par approche frontières

Les algorithmes de segmentation performants qui considèrent les frontières au sein des images utilisent principalement des modèles de contour actif. Leur principe de base consiste à épouser le contour des formes à l’aide de courbes paramétriques (par exemple polynomiales, comme les courbes de Bézier), ou de polygones. L’initialisation du processus se fait souvent par détection de points d’intérêts dans l’image, comme la jonction de plusieurs segments.

Ces algorithmes sont principalement utilisés dans le traitement de la vidéo, car on peut restreindre la complexité des nouvelles prédictions en utilisant la prédiction de l’image précédente comme point de départ. D’une manière générale ils sont efficaces dans le traitement consécutif d’images semblables.

1. Segmentation par classification

Cette famille de méthode considère le rapport entre les caractéristiques d’un pixel et les statistiques descriptives de l’ensemble de l’image (par exemple la moyenne ou la médiane des niveaux de gris).

On peut ensuite utiliser des méthodes comme la minimisation de la variance intra-classe ou K-means pour partitionner l’ensemble des pixels d’une image.

III – Deep Learning

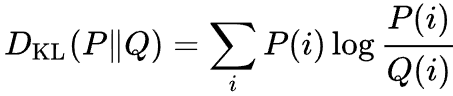
Très récemment, de nouvelles méthodes de segmentation d’images basées sur les réseaux de neurones profonds ont fait leur apparition. Ces réseaux sont composés de couches de neurones successives. Chaque neurone est une cellule indépendante, connectée à un certain nombre de neurones de la couche précédente (inférieure) et de la couche suivante (supérieure). Ils mettent en œuvre des principes variés mais l’approche de base est commune : la couche d’entrée est constituée d’une représentation des images sous la forme de vecteurs, de matrices, ou de tenseurs. La couche de sortie est constituée d’une distribution de probabilité des classes attribuées à chaque pixel. Entre ces deux couches, un modèle possède une succession de couches fonctionnelles paramétriques cachées. Ces paramètres sont appelés *poids* et sont appris par les modèles lors de l’entrainement. Un ensemble de couches est appelé *bloc*.

La plupart de ces nouvelles méthodes relèvent de l’apprentissage automatique supervisé, c’est-à-dire que les modèles sont entraînés en se basant sur des données correctement labelisées. Seules ces méthodes seront traitées ici.

A) Fonctions de perte

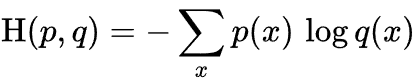
Lors de l’apprentissage, les modèles évaluent une *fonction de perte*, qui calcule la distance entre les prédictions du modèle à un instant donné, et les données de référence fournies au modèle. La fonction que l’on utilise dépend du type de problème auquel on fait face. Parmi les plus utilisées on peut trouver :

- La *Divergence de Kullback-Leibler* ou *entropie relative*, qui mesure la différence entre deux distributions de probabilités. Elle est calculée selon la formule suivante pour les variables discrètes :



Où P est la distribution de référence et Q la distribution prédite.

- L’*entropie croisée*, est une mesure analogue. Elle est calculée selon la formule suivante pour les variables discrètes :

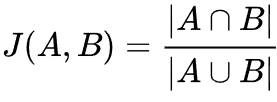


Où H est l’entropie croisée, p la distribution de référence, et q la distribution prédite par le modèle.

En pratique, ces fonctions évaluent le degré de biais d’une distribution : une distribution biaisée aura une forte tendance à prédire les évènements les plus probables, ce qui résultera en une faible entropie. A l’inverse, les distributions balancées auront une forte entropie. Dans un contexte d’entraînement supervisé la seule différence sera la valeur de convergence de ces fonctions lors de l’apprentissage (ce qui ne serait pas le cas si la distribution de référence n’était pas connue à tout instant).

- La *focal loss*, qui repose sur l’entropie croisée, mais où les valeurs sont pondérées de sorte à minimiser l’influence des exemples où le modèle à de bonnes performances, de manière à concentrer l’apprentissage sur les exemples plus difficile, donc moins bien classifiés. Le terme additionnel est (1 – p(x)) ɣ, et dépend de ɣ, un hyperparamètre fixé par l’utilisateur qui permet de contrôler le degré de pondération.

- Le *coefficient* *de Jaccard*, ou *Intersection over Union (IoU),* qui est défini par la formule suivante :



Où, pour une classe donnée, A est l’ensemble des pixels appartenant à cette classe dans les données de référence, et B l’ensemble des pixels appartenant à cette classe dans les prédictions du modèle.

Ce coefficient mesure la similarité entre la prédiction du modèle et la labellisation d’origine. Dans le but de pouvoir minimiser la fonction (voir B), on utilisera plutôt 1 – J (*fonction de perte de Jaccard*) lors de l’entraînement.

B) Apprentissage

Comme le problème de segmentation d’image possède un grand nombre de paramètres et n’est pas linéaire, on ne peut pas minimiser analytiquement la fonction de perte.

Le processus d’apprentissage consiste à utiliser un algorithme d’optimisation pour la minimiser itérativement. Le plus connu d’entre eux est la *descente de gradient*. Son principe est le suivant :

Comme les fonctions de perte sont différentiables, on peut calculer leur *gradient*, un vecteur comportant autant de composantes que de poids dans le modèle et indiquant la direction de la pente descendante la plus forte (il est constitué des dérivées partielles de la fonction de perte par rapport à chacun des poids). A chaque itération, les poids du modèle sont mis à jour, proportionnellement à leur valeur associée dans le gradient. L’entraînement est alors achevé lorsque le gradient est presque nul. Ce processus se nomme rétropropagation.

Pour éviter de dépasser le minimum de la fonction en modifiant trop les poids au cours d’une seule itération, on multiplie les composantes du gradient par une valeur comprise entre 0 et 1 (typiquement 0,01 ou 0,0001). Cette valeur est un hyperparamètre défini par l’utilisateur et se nomme l*earning rate*.

En pratique, ce sont des variantes de cet algorithme qui sont utilisées. Parmi elles on compte :

* La *descente de gradient stochastique*, une variante qui utilise à chaque itération un petit sous-ensemble des données (choisi aléatoirement) au lieu de leur totalité. Cela permet de palier le fait que des jeux d’entraînement volumineux associés à des modèles comportant beaucoup de paramètres peuvent rendre impraticable l’algorithme original (chaque itération serait trop longue à calculer).
* *Adaptive Gradient Algorithm (AdaGrad)*, une variante de la descente de gradient stochastique qui possède un learning rate propre à chaque paramètre au lieu d’un learning rate global
* *Adam*, un des algorithmes les plus populaires. Il combine les idées de plusieurs autres algorithmes (dont AdaGrad). L’idée est de calculer les moments d’ordre 1 et 2 des gradients (c’est-à-dire leur variation moyenne et la variance associée). En pratique, cela revient à calculer la moyenne glissante de l’évolution du gradient (ce qui permet lisser les oscillations pour obtenir sa tendance à plus long terme). Ceci permet de mieux optimiser les learning rate individuels de chaque paramètre, et donc d’accélérer la convergence.

Une autre idée implémentée est l’*élan (Momentum)*. Le principe est d’ajouter une fraction de la mise à jour précédente à chaque mise à jour : de cette manière, les mise à jour successives dans la même direction seront progressivement amplifiées, ce qui a également pour effet d’accélérer la convergence.

L’algorithme possède trois hyperparamètres : le learning rate initial, et beta1 et beta2 qui contrôlent le taux de décroissance de la moyenne glissante (car les oscillations se font moins importantes au fur et à mesure que le modèle progresse).

1. Pratiques courantes

Pour maximiser l’efficacité de l’apprentissage, certaines pratiques seront appliquées à la plupart des modèles.

Lors de la minimisation de la fonction de perte, deux problèmes récurrents peuvent survenir :

* L’explosion du gradient (augmentation très rapide des valeurs des gradients pendant l’apprentissage entraînant un dépassement de la capacité de la représentation interne des nombres et donc l'arrêt de l'apprentissage). Pour pallier ce problème, la plupart des modèles utilisent des fonctions d’activation en entrée de certaines couches, comme par exemple la fonction *ReLU* (ReLU(x) = x si x > 0 et 0 sinon).
* La disparition du gradient (diminution très rapide des valeurs des gradients pendant l’apprentissage entraînant l'annulation du gradient et l'arrêt de l'apprentissage alors que le modèle n’a pas fini d’apprendre). Ceci peut arriver notamment dans le cas de modèles comportant beaucoup de couches, car les multiplications successives par de petites valeurs lors de la rétropropagation du gradient peuvent le rendre infiniment petit.

Les *hyperparamètres* des modèles (comme le learning rate) peuvent être optimisés automatiquement. La méthode la plus courante est le *grid search*, qui consiste à tester différentes combinaisons d’hyperparamètres et de conserver la meilleure pour les entraînements futurs.

Pour surveiller la progression d’un modèle l’utilisateur à recours à des métriques (souvent inversement proportionnelles aux fonctions de pertes, pour obtenir un score qui croit au fur et à mesure que le modèle progresse).

L’entrainement est subdivisé en *époques*. A la fin d’une époque, les performances d’un modèle sont évaluées sur un jeu de données qui n’a pas servi lors de l’entraînement (jeu de *validation*), pour que l’utilisateur puisse suivre la progression du modèle de manière non biaisée, sans attendre la fin de l’entraînement.

Pour améliorer le fonctionnement des algorithmes d’optimisation de la fonction de perte, il est possible d’implémenter des *callbacks*, qui vont par exemple diminuer automatiquement le learning rate au cours de l’entraînement quand le modèle commence à avoir de bonnes performances.

Il est également possible d’arrêter automatiquement l’entraînement, quand le modèle n’a pas progressé depuis un nombre donné d’époques par exemple.

III - Présentation d’architectures

En quelques années, de nombreux nouveaux modèles toujours plus performants ont été créés, le plus souvent en améliorant certaines idées déjà implémentées ou en les combinant entre elles. Il ne s’agit donc pas d’en faire un exposé exhaustif ici, mais simplement de présenter quelques architectures

1. Réseaux d’extraction de features

L’*extraction de features* consiste à modéliser les points d’intérêt d’une image.

La plupart des réseaux d’extraction de features utilisent des *couches de convolution*. Leur principe est d’appliquer des filtres sur les images. Les filtres sont généralement des matrices 2x2 ou 3x3. Les valeurs des filtres sont apprises lors de l’entraînement. Une couche de convolution utilisera généralement plusieurs filtres, ce qui augmente la dimension des données en sortie d’une couche. La sortie d’une couche de convolution se nomme *feature map*. On note qu’une couche de convolution applique un filtre uniformément sur une image, ce qui rend les réseaux convolutifs (CNN) robustes aux changements de résolution des images en entrée.

Quelques modèles d’extraction de features :

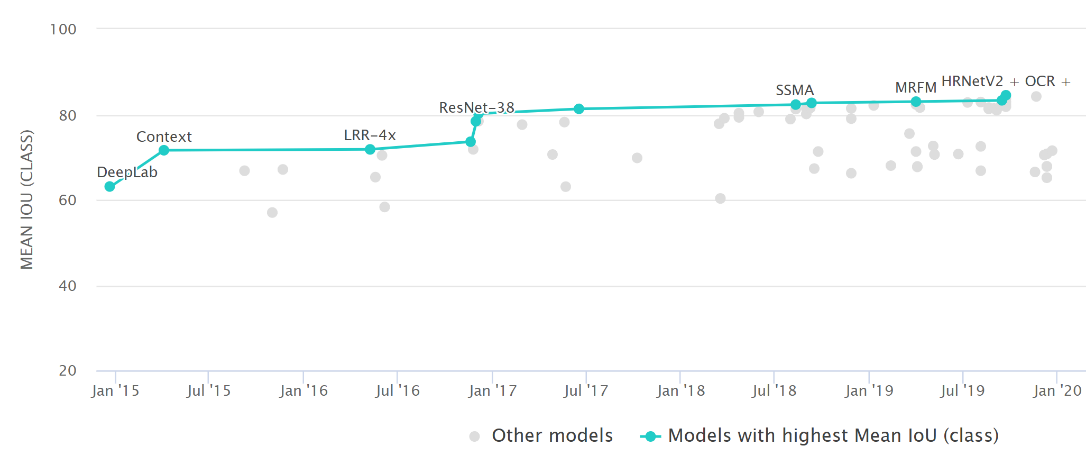
* Les *VGG* sont des réseaux convolutifs constitués de plusieurs blocs, chacun composé de plusieurs couches de convolution, d’une couche de pooling (qui permet de réduire la dimension des données) et d’une couche d’activation ReLU.
* *MobileNet* implémente une nouvelle idée : au lieu de réaliser les convolutions sur les trois canaux (RBG) en même temps, elles sont réalisées indépendamment sur chacun des canaux, puis recombinées lors d’une nouvelle opération de convolution 1x1 (*concaténation*). En pratique cela réduit le nombre de calculs à effectuer à chaque convolution et augmente significativement l’efficacité du modèle.
* Les *ResNet* comptent parmi les réseaux les plus populaires. Ils sont constitués d’un empilement d’un grand nombre de blocs, eux même composés de couches de convolutions (typiquement 34, 101, et plus récemment 1001). Pour éviter le problème de disparition du gradient (qui jusqu’alors était contourné par des astuces comme l’ajout d’une fonction de loss secondaire au milieu du modèle), et pour garder des temps de calculs gérables, le principe est d’utiliser des blocs résiduels. Au lieu de simplement connecter les blocs de convolutions au précédent et au suivant, les blocs apprennent de nouvelles connexions (appelées *fonctions résiduelles*). En pratique, cela permet à une partie de l’information générée dans un bloc de « sauter » un ou plusieurs blocs. L’avantage de cette architecture est d’ignorer les couches qui nuisent potentiellement au réseau dans un certain contexte.
* *ResNeXt* repose sur une extension des concepts implémentés dans ResNet : au lieu que les blocs résiduels soient composés d’un simple empilement de couches de convolution, il est composé d’un ensemble de sous-blocs, traités individuellement puis concaténés (cette méthode est appelée *split-transform-merge*). Cela ajoute une nouvelle dimension par rapport aux ResNet : la *cardinalité* (le nombre de sous-blocs dans chaque bloc résiduel). Il s’agit d’un hyperparamètre du modèle.

Un certain nombre d’architectures modernes reposent sur un réseau d’extraction de features (on appelle alors *backbone* ce dernier).

1. Etat de l’art

Chaque année, de nouveaux modèles toujours plus performants sont publiés. Ils sont comparés en étant entraînés sur les mêmes jeux de données. Entre 2010 et 2017, la compétition principale était ILSVRC, sur le jeu de donnée ImageNet. Ce jeu de données est constitué d’annotation sur la présence d’objets (repartis en un millier de classes) au sein d’images. L’objectif de la compétition était de créer des rectangles délimitant ces objets et de les attribuer à la bonne classe. C’est durant cette période que les réseaux d’extraction de features se sont énormément développés. Après 2015, la communauté s’est portée sur d’autres jeux de données impliquant des problèmes de segmentation plus complexes. CityScapes, le jeu de données utilisé dans ce projet, est l’un des principaux à être utilisé comme référence (*benchmark*) en raison de sa complexité (images 2048x1024, segmentées selon 32 classes pouvant être regroupées en 8 classes principales). Il est composé d’images labelisées de conduite urbaine prises par une caméra sur le capot d’une voiture.

Le diagramme suivant présente les avancées notables réalisée par des modèles évalués sur CityScapes :



Encore une fois l’objectif n’est pas de présenter exhaustivement ces modèles. Je présente ici les idées majeures implémentées dans une suite de modèles, dont le plus récent est emblématique de l’état de l’art actuel.

**DeepLab**

Le modèle DeepLab original présenté en 2015 a été mis à jour plusieurs fois pour être amélioré. Ses performances ont significativement augmenté (passant successivement à 70.4%, 81.3%, et 82.1% de mIoU sur CityScapes pour les versions 2, 3 et 3+ respectivement). Il a également bien performé sur d’autres jeux de données. Ces modèles combinent trois concepts : les Atrous Convolutions, le Atrous Spatial Pyramid Pooling, et les Fully-Connected Conditional Random Fields. Ses premières versions utilisent un ResNet 101 comme backbone.

Les *Atrous Convolutions* (littéralement convolutions à trous), sont des couches qui consistent à n’utiliser qu’une partie des pixels de l’image lors de l’opération de convolution. En pratique, cela revient à espacer les valeurs d’un filtre de convolution (par le biais d’un *facteur de* *dilatation*) : par exemple un filtre 3x3 avec un facteur de dilatation de 2 deviendra un filtre 5x5 creux (avec des valeurs neutres sur les 2e et 4e lignes et colonnes). Cette méthode offre une plus grande robustesse aux variations d’échelle des objets à détecter.

Le *Atrous Spatial Pyramid Pooling* est un bloc du réseau qui consiste à appliquer de multiples convolutions à trous sur les mêmes données, mais avec différents facteurs de dilatation. Les feature maps qui en résultent n’ont donc pas la même dimension. On utilise alors l*’interpolation bilinéaire*, un processus consistant à calculer les valeurs moyennes de tuiles de pixels sur une image pour en réduire la dimension. Cela permet d’uniformiser la dimension des feature maps. Elles sont alors concaténées. Cette extension des convolutions à trous offre une robustesse encore plus grande aux changements d’échelle. Ces opérations sont associées à des couches de pooling pour réduire la dimension des données.

Un *Conditional Random Field (CRF)* est un modèle permettant de considérer les caractéristiques des voisins d’un pixel donné. Il est utilisé juste avant la couche de prédiction. En pratique cette couche est capable de rétro-propager deux types de pénalités : une qui concerne les pixels mal labelisés, et une autre qui considère les frontières (autrement dit, la cohérence entre les caractéristiques et la labélisation de voisinages de pixels, ce que l’on peut se représenter comme l’harmonie visuelle de la segmentation). L’aspect Fully-Connected signifie qu’au lieu de considérer seulement les voisins proches d’un pixel, on considère l’ensemble des pixels (ce qui implique l’apparition de la notion de distance). On note qu’il existe des algorithmes permettant de réaliser les calculs avec une complexité linéaire (*mean field inference*).

La dernière version en date de DeepLab est DeepLabv3+ (2018). Elle utilise une idée nouvelle :

Le réseau a été structuré sous la forme d’un *encoder-decoder* (dont le principe est de produire des feature maps de plus en plus petites et nombreuses, avant d’effectuer les transformations de dimensions inverses pour pouvoir effectuer la prédiction). L’encoder est une version modifiée d’*Xception* (un extracteur de features), qui utilise le concept d’*Atrous Separable Convolution* (de manière analogue aux MobileNet, les convolutions sont désormais réalisées indépendamment sur chacun des canaux RGB).

DeepLabv3+ possède un bloc ASPP. Les sorties de ce bloc sont concaténées et *suréchantillonnées* (on parle *d’upsampling*, c’est-à-dire que leur dimension est augmentée en dupliquant leurs valeurs), ce qui permet de les concaténer avec les sorties de l’encoder. Le reste du réseau est constitué d’un enchaînement de deux couches de convolution 3x3 et d’un dernier upsampling pour créer la segmentation finale.

D’autres modèles utilisent certaines des idées implémentées dans la suite DeepLab, et DeepLabv3+ a lui-même été utilisé comme backbone, fin 2021, pour un nouveau type d’application aux CRF : la régularisation de Frank-Wolfe (généralisation du concept de mean field inference).

IV – Approche pratique pour la segmentation

Cette section décrit la méthodologie mise en œuvre pour entraîner des modèles dans le cadre du projet.

1. Augmentation des données

Une pratique courante dans la préparation des données consiste à appliquer des transformations paramétriques aléatoires aux images d’entraînement. Comme le jeu de données ne contient que quelques milliers d’images, cela peut permettre à un modèle d’être plus robuste lorsqu’il rencontrera des contextes différents de ceux présents dans le jeu d’entraînement. Les transformations doivent être des déformations réalistes pour que cette pratique ait un intérêt. Dans le cadre des données de CityScapes, les augmentations que j’ai choisies sont les suivantes :

* Symétrie horizontale
* Agrandissement linéaire (robustesse au changement d’échelle)
* Flou ou bruit gaussien (robustesse à l’effet de la vitesse et aux régions floues)
* Modification de la luminosité / du contraste / de la saturation (robustesse au niveau d’éclairage / à la luminosité naturelle)
* Suppression d’une petite fraction de pixels (robustesse au contexte)

Dans mon implémentation, une augmentation consiste à combiner de 0 à 4 de ces traitements.

1. Format des données

Pour entraîner un modèle volumineux (tel que DeepLab présenté plus haut) avec des images aussi grandes que celles de CityScapes, les ressources informatiques nécessaires pour effectuer les calculs sont prohibitives. En effet, il s’agit de modèles entraînés pendant plusieurs jours sur des machines aux très larges capacités (généralement capable d’utiliser de l’ordre d’une centaine de Go de mémoire de GPU pour paralléliser les calculs), ce qui les rend impraticables dans le cas d’un projet scolaire, que ce soit en termes de temps ou de prix des infrastructures cloud.

Une solution naïve aurait consisté à redimensionner les images mais cela pose deux problèmes :

* La labellisation des données est assez précise, et comme il aurait fallu réduire les images d’un facteur important, certains objets auraient été réduits à l’état de lignes, voire auraient disparu, ce qui pose un évident problème de qualité des prédictions dans le contexte du projet.
* Une fois qu’un modèle a été entraîné sur les images redimensionnées, il sera impossible d’effectuer des prédictions fiables sur les images d’origine, car les modèles sont sensibles à l’échelle (un modèle qui aura appris par des images ou les voitures font 50 pixels sera incapable de reconnaitre une voiture de 500 pixels).

La solution que j’ai retenue consiste à exploiter la propriété d’invariance à la dimension des données d’entrée des réseaux de neurones convolutifs : les modèles sont donc entraînés à partir de sous-images (*crops*) de petite dimension des images du jeu d’entraînement (224x224 dans mon cas). Le défaut majeur de cette solution est la perte de contexte (le bas des images originales est systématiquement au niveau de la route, ce qui n’est pas le cas dans les crops, par exemple). Les crops possèdent également plus d’objets tronqués, en proportion. Mes attentes n’étaient donc pas d’égaler les performances des benchmarks. Cette solution possède toutefois plusieurs avantages :

* Malgré l’entraînement sur de petites images, les modèles peuvent effectuer des prédictions pertinentes sur les images d’origine.
* L’entraînement et la parallélisation des calculs est désormais possible sur des machines accessibles.
* Elle permet de générer les augmentations de données pendant l’entraînement : habituellement les augmentations des images sont faites en amont, et statiques, car il s’agit d’opération coûteuses à effectuer sur de grandes images. Dans le cas des crops, les augmentations peuvent être faites juste après le découpage, à chaque fois qu’une image est chargée en mémoire. Comme il est possible de précharger les images suivantes pendant l’entraînement d’un groupe d’image, l’impact sur les temps de calcul est négligeable. Le bénéfice est de posséder un jeu de données théoriquement infini, puisqu’il existe un grand nombre de découpages possibles d’une image, et une infinité d’augmentations possibles. Ceci limite considérablement les risques de surapprentissage.

1. Réseau convolutif avec encoder-decoder et saut de blocs

Le modèle de référence que j’ai utilisé dans ce projet est un petit réseau encoder-decoder.

L’encodeur est constitué de deux blocs de convolutions, eux-mêmes composés de deux convolutions 3x3, séparées par un dropout de 20% et suivies d’un pooling 2x2.

Le décodeur est lui aussi constitué de deux blocs contenant des opérations de convolution similaires, mais au lieu du pooling, les données subissent un upsampling de facteur 2 (réciproque au pooling de l’encodeur).

De plus, juste avant l’upsampling, les features maps du décodeur sont concaténées avec les feature maps intermédiaires de l’encoder (celles de dimensions similaires). Cela permet de pallier la perte d’information bas niveau induite par les réductions de dimensions successives de l’encodeur. En pratique, cela se traduit par une plus grande précision dans la détermination des frontières.

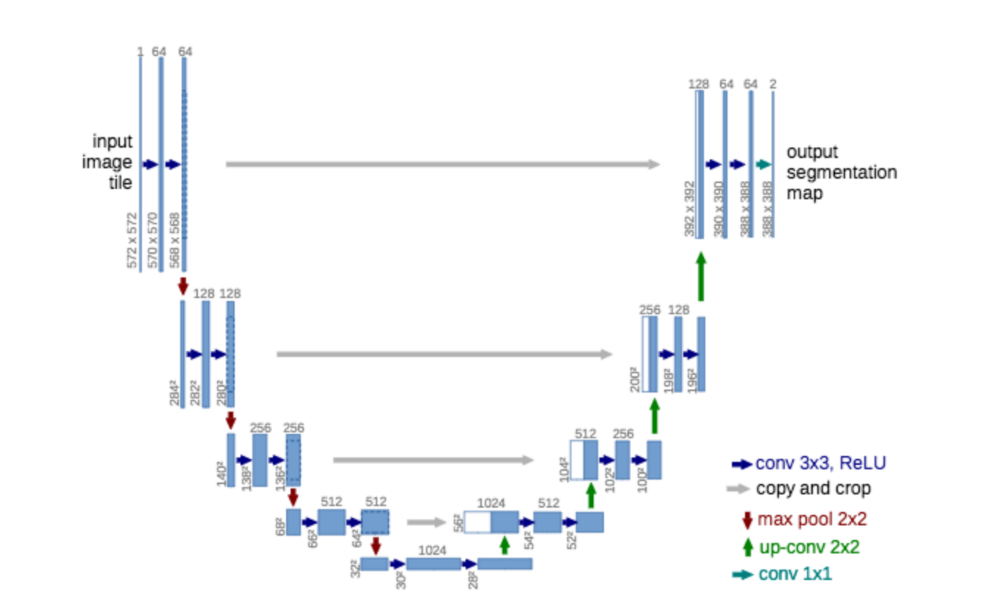
La fonction de perte a été choisie en utilisant du grid search parmi une variété de fonctions. Pour l’ensemble des modèles, celle que j’ai retenu est la somme entre la fonction de perte de Jaccard et l’entropie croisée.

La métrique utilisée est le coefficient de Jaccard (IoU), et l’algorithme d’optimisation est Adam, avec un learning rate initial de 0.0001.

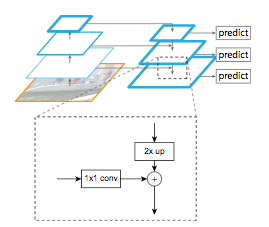
1. UNET et FPN

UNET (pour réseau en U) et FPN (pour Feature Pyramid Network) sont des réseaux de type encoder-decoder avec saut de blocs. UNET est l’un des réseaux qui ont inspiré l’architecture de DeepLab, et FPN est lui-même une extension de UNET.

L’architecture de UNET est similaire à celle de mon réseau de référence, à l’exception qu’elle comporte plus de blocs :



FCN est un réseau assez similaire à UNET. Sa différence majeure est d’effectuer des prédictions en utilisant aussi les couches intermédiaires du décodeur :



Une grande variété de backbones peut servir d’encodeur à ces modèles. J’ai choisi d’entraîner UNET avec resnet34 et FPN avec ResNeXt50 (architectures populaires).

IV – Résultats

|  |  |  |  |
| --- | --- | --- | --- |
| **Modèle** | **IoU score** | **Loss** | **Temps** |
| Baseline | 0.24 | 0.97 | 20 minutes |
| Baseline avec flou et bruit | 0.23 | 1.01 | 27 minutes |
| Baseline avec modifications de de contraste | 0.19 | 1.06 | 24 minutes |
| Baseline avec pixel dropout | 0.26 | 0.94 | 25 minutes |
| Baseline avec les augmentations mélioratives | 0.28 | 0.90 | 37 minutes |
| FPN + ResNeXt50 avec augmentations | 0.50 | 0.57 | 33h |
| UNET + ResNet34 avec augmentations | 0.56 | 0.56 | 13h30 |

Parmi les augmentations, certaines se sont surprenamment révélées contre-productives (toutes n’ont pas été testées individuellement). Ces dernières n’ont donc pas été réalisée lors de l’entraînement des modèles.

Autre surprise : UNET a donné de significativement meilleurs résultats que FPN. Mon hypothèse est que son architecture se comporterait mieux avec les crops.

Une piste d’amélioration envisageable pour ce projet aurait été de diversifier les modèles entraînés, et de me tourner vers des modèles plus complexes à mettre en œuvre, quitte à multiplier les temps de calculs.