Segmentation d’images de microscopie électronique

Introduction :

Le but de la segmentation n'est pas simplement de dessiner des contours marqués mais bien de discerner des objets particuliers dans une image. Ici, nous voulons discerner les parois des cellules sur des photographies de tissus observés via microscope électronique à balayage (MEB).

Un simple travail graphique manuel sur ces images afin de retracer ces contours, par exemple avec un logiciel de traitement d'image en augmentant les contrastes, en gérant les niveaux et en effaçant les noyaux serait possible mais fastidieux. L'automatisation de cette tâche via l'IA est pertinente. Il ne s'agit alors que de tracer les contours à la main une seule et unique fois afin d'avoir les labels, puis, via l'apprentissage machine, on obtient un algorithme qui peut réaliser ce travail de segmentation tout seul sur n'importe quelle nouvelle image de microscopie similaire.

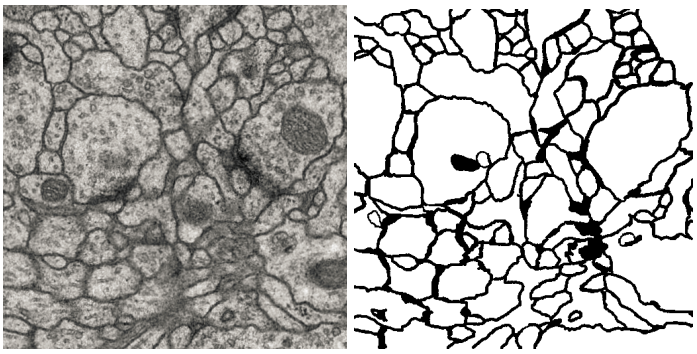
I. Les données :

Le dataset se compose de 30 images de tissus observés au MEB, ce sont les features. Il y a également 30 images de même taille mais où ne sont présents que les contours des cellules, ce sont nos cibles/targets, appelés ici labels. Il y a un x_train et un x_test présents mais le x_test ne me parait pas exploitable dans la mesure où les labels ne valent que pour le x_train. Pour la phase de test, il nous faudra donc créer notre propre test à partir du train.

Il est à noter que les données sont dans un format propriétaire Adobe appelé TIF. Pour les besoins du modèle, il a fallu les convertir en PNG. Pour ce faire, j’ai initialement tenté d’utiliser une API appelée Cloudconvert, seulement, après avoir tout réglé, l’API prétendait fonctionner et avoir réalisé la conversion mais je ne voyais pas comment récupérer l’output de mes fichiers ! J’ai ensuite découvert qu’on pouvait faire la même chose beaucoup plus vite et simplement avec le module PIL...

II. La data-augmentation :

Dans la mesure où il n'y a que 30 images pour les features et que le x_test n'est pas exploitable, la phase de data-augmentation est critique au bon déroulé du projet. En effet, il faut tirer un maximum de ce peu d'images disponibles afin d'avoir tout de même un algorithme généralisable.



Une image d'apprentissage (gauche) et son label (droite)

Pour la data-augmentation, j'ai opté pour un traitement des images avec la bibliothèque open-cv en rédigeant mes propres fonctions. Ces fonctions traitent les images en créant des versions data-augmentées. Les types de data-augmentation sont les suivants :

- flou gaussien,
- augmentation de la netteté,
- luminosité augmentée,
- luminosité diminuée
- rotation,
- miroir horizontal,

Cela nous amène à 210 images au lieu des 30 initiales. Pour avoir les 210 labels associés, il suffit de concaténer les 30 labels dans un array autant de fois que nécessaire jusqu'à en avoir 210 également. A noter que les labels des features ayant subi la rotation et le miroir doivent également subir ces transformations pour que le modèle puisse apprendre correctement.

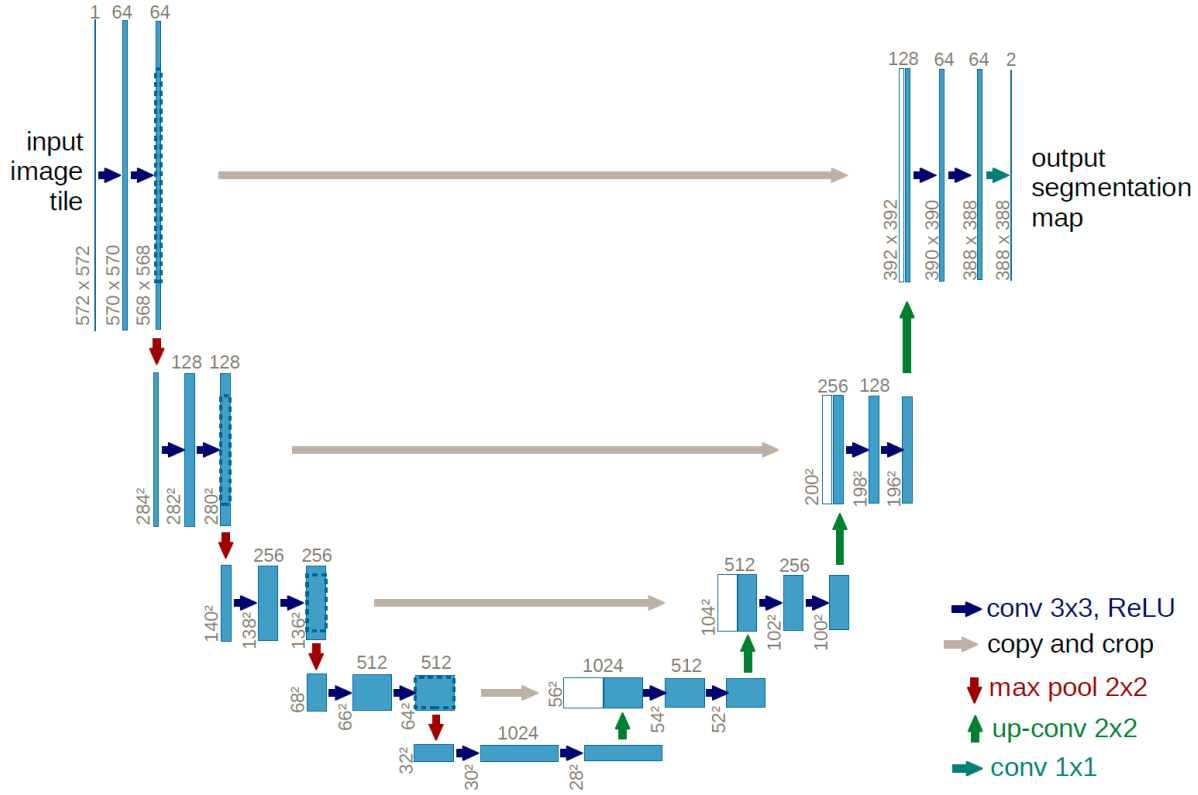
III. Le modèle

1) L'architecture

Le modèle choisi pour ce brief est un réseau de neurones en "U" (U-Net). Ce type de modèle a justement été développé spécifiquement pour la segmentation d'images biomédicales à l'université de Fribourg.

Le modèle est entièrement convolutionnel (il ne comporte pas de couche Dense). Son architecture est censée permettre de travailler sur de "grosses" images de 512 par 512

pixels rapidement sur GPU.



Architecture U-net classique

Comme visible sur l'image ci-dessus, le modèle fonctionne en deux phases, la première branche du U est la phase de contraction par couches de convolution successives. Ensuite, l'image est ramenée à sa taille d'origine par un suréchantillonnage dans la seconde branche du U via des couches de déconvolution.

Notons également que ce type de modèle est prévu pour pouvoir fonctionner avec peu d'images d'entraînement. Sur le papier, il est donc parfaitement adapté à notre problème.

Pour l'implémentation python de cette architecture, j'ai utilisé les bibliothèques Keras et Tensorflow. Il a fallu créer des métriques personnalisées qui soient adaptés au problème de segmentation. En lisant la littérature, j'ai découvert qu'on pouvait se reposer sur le coefficient de Dice pour juger de l'efficacité de notre modèle.

Ce coefficient est un indicateur statistique qui mesure la similarité de deux échantillons, ici entre le label réel et l'output prédit par le modèle. Le Dice est compris entre 0 et 1. Si les deux images n'ont rien à voir, le Dice sera proche de 0, si elles sont des copies conformes ou quasiment, il sera très proche de 1 (c'est notre objectif).

$$s = \frac{2|X \cap Y|}{|X| + |Y|}.$$

Formule générale du coefficient de Dice

L'implémentation python de ce coefficient est proche de la formule ci-dessus à ceci près qu'on ajoute un smooth de 1. au numérateur et au dénominateur afin de ne pas changer le rapport tout en évitant les risques de diviser par zéro. Le X de la formule est le label réel, le Y est le label prédit par le modèle. On obtient l'intersection de ces deux images par la somme du produit des tenseurs.

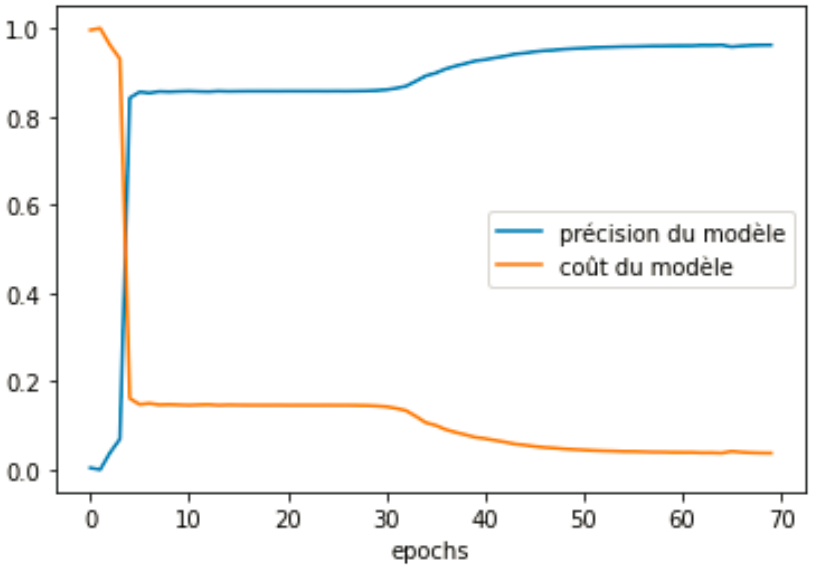
```
[34]  1 def dice_coef(y_true, y_pred):
      2     # Ici on reproduit la formule montrée plus haut pour le coeff Dice.
      3     y_true_f = K.flatten(y_true) # Les arrays à comparer doivent être écrasés donc on utilise flatten()
      4     y_pred_f = K.flatten(y_pred)
      5     intersection = K.sum(y_true_f * y_pred_f)
      6     coeff_dice = (2. * intersection + smooth)/(K.sum(y_true_f) + K.sum(y_pred_f) + smooth)
      7     return coeff_dice
```

La fonction coût à minimiser afin que notre modèle apprenne est tout simplement le Loss du coefficient Dice soit 1 moins le coefficient. Ainsi, si on a un coefficient Dice proche de 1, on aura une fonction coût proche de zéro.

```
[35]  1 def dice_coef_loss(y_true, y_pred):
      2     # La fonction coût à minimiser est l'opposé du coefficient Dice.
      3     # Dit autrement, on veut un Dice le plus haut possible (1).
      4     return 1 - dice_coef(y_true, y_pred)
```

Les performances

Au bout de 70 epochs, le modèle offre un coefficient Dice de 0.95% en test, cela signifie que les images prédites sont 95% similaires à celles des labels. Le modèle peut donner de meilleurs résultats sur plus d'epochs (voir les difficultés dans le paragraphe suivant pour plus de détails).



Courbe d'évolution du coût et du coefficient de Dice en fonction des epochs

On voit bien sur la courbe que le coefficient de Dice et le coût du modèle sont inversement corrélés de façon parfaite comme prévu. On remarque qu'ils montent très rapidement en 5 epochs puis il y a un plateau et le modèle converge plus doucement avec l'aide du callback qui réduit

IV. Difficultés rencontrées

Output médiocre :

J'ai longtemps eu un output très médiocre sorti par la dernière couche à savoir des images parfaitement noires (arrays de 0 uniquement ou presque) ou bien parfaitement blanches (arrays de 1 uniquement ou presque). Mon problème se trouvait dans mon prétraitement sur les features et targets, apparemment inadapté à mon réseau de neurones (ou à ce problème de segmentation en particulier ?). J'avais en effet opté pour une standardisation en divisant mes tenseurs par l'écart-type du x_train et en leur soustrayant la moyenne du x_train. Pour une raison qui m'échappe, le seul moyen d'aboutir à un output passable a été de changer totalement la méthode de prétraitement et de simplement diviser tous les tenseurs par le float 255. (passant ainsi sur des valeurs exclusivement comprises entre 0 et 1 pour chaque pixel des tenseurs).

Une fois ce petit problème résolu, l'output ressemblait enfin à quelque chose et pour l'améliorer, il ne s'agissait plus que d'optimiser le U-Net en visant de meilleures métriques.

Soupçon de surapprentissage :

Je n'ai pas eu de gros problème de surapprentissage à proprement parler mais parfois le Dice général de train augmentait un peu plus vite que le Dice de validation donc dans le doute j'ai tenté des couches de Dropout. J'ai tenté de les placer à des endroits-clé, préférentiellement à la fin de la phase descendante du U-net et avant la couche de batch-normalization afin d'éviter de passer de l'information aux neurones suivants via la normalisation et de "gâcher" le Dropout. Ces essais ont été peu concluants sur ce problème

de segmentation précis. De toute façon, la métrique de validation a bien évolué tout au long de l'apprentissage sur le modèle final.

Temps d'apprentissage et limites de GPU :

Afin d'obtenir un bon coefficient de Dice en validation, il a été plus déterminant de cibler le bon learning rate via un fastidieux fine-tuning. En effet, les fruits du réglage du pas d'apprentissage n'étaient visibles qu'après avoir attendu un grand nombre d'epochs car le modèle ne stagnait pas mais il convergeait assez doucement. Ces epochs étaient d'ailleurs assez lents à cause des problèmes de "Out Of Memory Error" qui m'ont contraint à limiter la taille du batch size à moins de 10. Je suppose que travailler avec des images de 512 par 512 est assez taxant même pour le GPU de Google Colab. A force d'essais, j'ai d'ailleurs fini par être expulsé du GPU Google pendant des heures ce qui m'a gêné dans mon travail.

Etonnamment, même sur la prédiction, le modèle a tendance à provoquer des erreurs OOM. Pour éviter cela, j'ai utilisé un paramètre batch_size personnalisé de 5 dans l'appel à la fonction predict().

En réalité, mon architecture permet d'avoir de meilleurs résultats car le modèle continue à apprendre après 70 epochs. Il converge juste tout doucement. En entraînant sur 200 epochs, je suis parvenu jusqu'à 97% de Dice en test mais j'ai oublié d'enregistrer mon modèle et les ré-entraînements sur colab sont trop coûteux en temps de GPU alloué. Pour être plus à l'aise pour expérimenter, il faudrait peut-être que je configure mon setup en local afin d'utiliser mon propre GPU ou bien que je paie une souscription à Colab Pro.

Conclusion

Les avantages d'utiliser un U-net sont de pouvoir travailler relativement rapidement avec des images de résolution élevée afin de réaliser une tâche de segmentation. Nous avons vu qu'en conditions réelles sur Google Colab, la rapidité était toute relative à cause des problèmes de mémoire limitée. Peut-être que sur notre propre GPU, il aurait été plus simple de travailler à l'optimisation et l'entraînement de ce modèle. Avec un notre propre GPU, un plus gros batch size et donc un entraînement plus rapide seraient sûrement possibles. Ce n'est pas tant l'architecture U-Net qui pose problème que les limitations de Colab en mode gratuit.

Aussi, concernant le choix du modèle, il existe bien entendu des alternatives et/ou compléments aux U-Net pour répondre aux problèmes de segmentation. On peut par exemple imaginer combiner un U-Net à un discriminateur dans une approche GAN pour améliorer la qualité de l'output. On peut aussi utiliser un réseau tel que Mask_RCNN.

La page github suivante liste des modèles concurrents aux U-Net :

https://github.com/osmr/imgclsmob

Je n'ai pas pris le temps de tester ces alternatives par manque de temps et car le U-Net a offert des résultats plutôt satisfaisants mais cela pourrait être intéressant à l'avenir.