Université LAVAL

Apprentissage et reconnaissance GIF-4101/GIF-7005

Projet : Détection automatique de prolongements neuronaux

Auteurs :
Baptiste AMATO
Arnoud VANHUELE
Alexandre CHAVENON

1 Introduction

Présentation du projet

Le projet est proposé par le centre de recherche CERVO. Il consiste à reconnaître des axones et des dendrites sur des images d'une protéine (actine), en étiquetant ces images n'ayant pas de marqueurs axonaux et dendritiques. Nous disposons d'une banque de données d'images déjà marquées : il s'agit donc un problème d'apprentissage supervisé.

Jeu de données

Le jeu de données initial comprend 1024 images au format *.tiff*, ayant chacune trois canaux : un pour l'actine (la protéine d'intérêt), un pour les axones, et un pour les dendrites.
Ce jeu de données étant relativement petit pour un apprentissage par réseau neuronal, nous allons utiliser des méthodes d'augmentation comme des symétries et des découpes de sous-parties des images.

Etat de l'art

Il s'agit ici de détecter différents objets dans une image (axones et dendrites à partir d'une image globale d'actine) : c'est un problème de détection particulier, car il n'est pas possible d'encadrer les objets par des "bounding boxes", utilisées par exemple pour la détection de visage, de personnes ou de voitures ; on cherche alors à détecter le contour des objets. Un article de recherche assez récent a démontré une capacité de détection de contour impressionnante : *Object Contour Detection with a Fully Convolutional Encoder-Decoder Network*, par **Yang *et al.***. Nous nous sommes donc orientés vers un réseau de neurones profond avec une architecture *Encoder-Decoder* ; cette architecture est aussi utilisée dans les traductions de textes (séquences en entrée et sortie). Des résultats probants concernant de la segmentation d'images sont présentés dans l'article *Iterative Deep Convolutional Encoder-Decoder Network for Medical Image Segmentation*, par **Jung Uk Kim, Hak Gu Kim, et Yong Man Ro**, suivant une architecture similaire (*Encoder-Decoder*).
Les principes de segmentation d'image sont clairement expliqués dans l'article **Fully Convolutional Networks for Semantic Segmentation**, par **Shelhamer *et al.***.

2 Pré-traitements

Masques

Une image du jeu de données fournit contient trois canaux, pour l'actine, les axones et les dendrites. Nous séparons donc le *.tiff* afin d'obtenir trois images, avec l'actine en vert, les axones en rouge et les dendrites en bleu. Après traitements, on a des images comme suit :

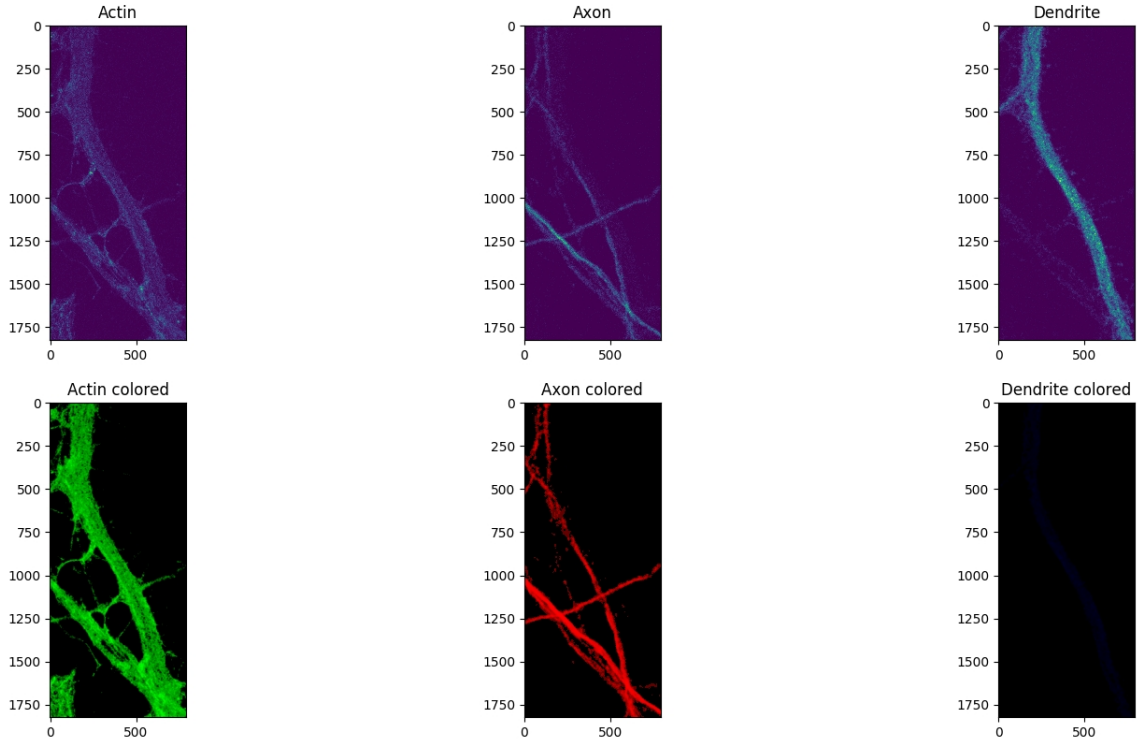

Figure 2.1: En haut, un triplet actine-axone-dendrite et en bas les mêmes images une fois le masque appliqué

Le pré-traitement commence par une égalisation de l'histogramme des intensités des images en niveaux de gris, afin de pousser légèrement le contraste et donc faire ressortir l'information la plus importante. On utilise ensuite deux opérations, une d'érosion afin de mieux délimiter les contours et une de flou gaussien, afin de raffiner les continuités qui auront pu être perdues lors de l'érosion.
On applique ensuite un *thresholding* sur chaque image, permettant de supprimer du bruit, soit les pixels de très faible intensité : on définit un seuil entre 0 et 255 (dans notre cas, 10), et tous les pixels ayant une valeur inférieure à 10 sont ramenés à 0. On convertit ensuite les images en RGB et on conserve des valeurs non nulles que pour un canal par image (vert pour l'actine, rouge pour les axones et bleu pour les dendrites). Toutes les valeurs sont ramenées entre 0 et 1.

Images d'entrée

Taille

Les images en entrée sont toutes de dimensions différentes, mais de même résolution. Il est donc nécessaire de conserver ces résolutions (ayant un sens physique) en n'effectuant aucun redimensionnement. Le problème est que le réseau de neurones attend des images de même taille en entrée : nous avons donc découpé chacune des images en *crops* (petits carrés) de taille 224x224, correspondant à la taille des images en entrée du réseau **VGG16**, mentionné plus bas.

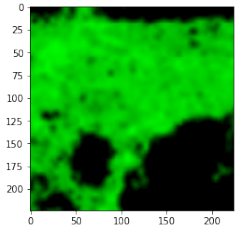

Figure 2.2: Exemple de *crop* d'une image d'entrée (actine seule). La faible résolution n'est pas due à redimensionnement de l'image d'origine mais seulement de la pauvre qualité d'une capture d'écran

Contenu

Notre problème est une détection de contour, mais nous devons inclure la notion d'**intensité** dans nos matrices d'entrée (valeur allant de 0 à 255). Les images sont normalisées entre 0 et 1 ; une matrice d'entraînement sera de taille 224x224x1 (intensités de l'actine), et le label associé sera une matrice de taille 224x224x1, selon si l'on souhaite entraîner le réseau sur les axones ou les dendrites. Nous avions commencé en traitant simultanément les axones et dendrites (donc en ayant des labels de taille 224x224x2), mais il semblait que l'un des canaux est souvent sur-appris alors que l'autre était délaissé. La fonction de perte devait retourner une valeur unique pour la matrice, donc le réseau n'avait aucun moyen de savoir quel canal devait être corrigé.
Nous nous sommes demandés s'il était nécessaire de conserver des valeurs décimales pour les axones en dendrites, ou bien ne garder que des valeurs binaires comme le font les autres problèmes de segmentation d'image. Nous avons donc créé deux jeux de données, l'un binaire, l'autre non, afin de comparer les performances. Il semble finalement qu'il est préférable de garder des valeurs décimales.

Prédiction d'une nouvelle image

Là encore, il est nécessaire de procéder par *crops* pour traiter une nouvelle image. Ainsi, chaque *crop* sera associé à une image de prédiction et l'image résultante sera reconstituée à partir de ces *crops*. Le réseau de neurones n'a donc affaires qu'à des images de taille 224x224x1.

Augmentation des données

Nous avons, pour chaque *crop*, ajouté des *flips* (opérations "miroir") permettant d'avoir quatre nouvelles données en sortie (l'originale, la symmétrie horizontale, verticale et l'enchaînement des deux). Ces opérations ne sont effectuées que lorsque le crop est assez "discriminant", c'est-à-dire si la différence entre l'actine et le masque d'axone ou de dendrite est assez importante.

3 Réseau de neurones

Nous utilisons la librairie *Keras* avec un back-end en *TensorFlow*. L'architecture sera celle présentée dans l'article de recherche *Object Contour Detection with a Fully Convolutional Encoder-Decoder Network*, par **Yang *et al.*** : la première partie du réseau est l'encodeur, basé sur l'architecture du réseau **VGG16**, en s'arrêtant juste avant le *Fully Connected layer*. Ensuite, le décodeur est celui décrit par l'article, permettant de reconstituer une image de la taille d'origine avec des opérations de *déconvolution*, qui en *Keras* se font grâce à l'opération *UpSampling* suivie d'une *same convolution* (convolution ne modifiant pas la taille de l'image d'entrée grâce à l'ajout de *padding*).
L'optimisation est faite avec la méthode **Adam** utilisant les avantages des algorithmes **AdaGrad** et **RMSProp** ; celle-ci a fait ses preuves dans le monde du Deep Learning. Nous utilisons la fonction de perte **mean squared error**.

4 Tests

Le jeu de données, après transformations, est très lourd (plusieurs dizaines de Giga Octets). De plus, chaque image n'est pas fournie telle quelle au réseau de neurones, mais est découpée en *crops* qui, selon la taille de l'image, peuvent être au nombre de 30 voire 40 pour une unique image. Ainsi, nos ordinateurs ne pouvaient pas créer des tableaux *numpy* contenant toutes les données. De plus, lors des entraînements du réseau, nous avons été confrontés à des *ResourceExhaustedError*, donc nous devions utiliser des tailles de *mini-batches* très petites (4 au plus), ce qui ne donne pas de bons résultats.
Nous avons alors obtenu l'accès à une machine des laboratoires de l'université ayant un GPU puissant afin d'effectuer des tests sur un grand nombre de données et des tailles de *mini-batches* raisonnables (32). Il a été nécessaire de créer une instance *Docker* pour pouvoir empaqueter notre code dans la machine ; une fois cela fait nous avions accès à distance à notre code et pouvions donc lancer plusieurs tests sur l'ensemble des données.
Chaque exécution de modèle génère deux fichiers, tous deux accompagnés du nom du modèle en question : l'un conserve la structure du modèle en *.json* et l'autre sauvegarde les poids appris en *.hdf5*. On peut ainsi "re-créer" en local le modèle déjà appris et voir ce qu'il donne sur différentes images.
Afin de tester l'efficacité du réseau, nous avons séparé le jeu de données en deux : une partie pour l'entraînement, et une partie pour les tests. Lors de l'apprentissage en tant que tel, 30% des données d'entraînement sont utilisées pour la validation, avec un arrêt prématuré si deux itérations consécutives n'améliorent pas les performances sur le jeu de validation. Enfin, toutes les images de résultats présentées plus bas font partie du jeu de test.

5 Résultats

Post-traitement

Le but est d'obtenir des masques d'axones et dendrites sur une image d'actine. On ne s'intéresse donc qu'aux valeurs non nulles de l'image d'origine, c'est pourquoi nous appliquons un masque sur les prédictions afin de n'avoir que du contenu pouvant être superposé à l'actine d'origine (en effet, dans la plupart des prédictions, le fond est légèrement coloré, mais cela nous importe peu).

Exemples de prédictions

Temps d'entraînement

Il s'avère que les résultats ci-dessus peuvent être obtenus assez rapidement, après seulement quelques *epochs*. En fait, le taux d'erreur est de moins de 25% à la fin du premier *epoch*, puis stagne autour de 17-18% après environ cinq *epochs*.

6 Analyse

7 Pistes d'amélioration

8 Protocole d'utilisation

La première chose à faire est de créer un fichier **config.py** et de spécifier le chemin d'accès vers les images d'origine dans la variable *main_folder_path*. Les images doivent être placées dans un dossier appelé *original_data* (que l'on peut modifier dans le fichier **constants.py**). Il faut ensuite générer le jeu de données traitées, à l'aide de la fonction *save_train_label_images(number_of_images0)* du fichier **dataset.py**. Si l'on veut ré-entraîner le modèle, il faut aussi générer le jeu de données d'entraînement (après augmentation des données), avec *save_dataset(nb_images, channel)* du même fichier (*channel* prend l'une des valeurs "axons" ou "dendrites"). Un exemple d'entraînement est donné dans le fichier **multi_testing.py** ; le modèle doit être défini dans le fichier **models.py**. Afin d'afficher une image provenant du jeu de données traitées, il suffit d'appeler la fonction *get_images_from_train_label(X[i], y[i], channel)*, où *X* et *y* représentent les jeu de données traitées.