Maxime RICHAUDEAU
Yorik NYSSEN

Modèle de segmentation d'image - Place de parking

1. Origine du projet

Nous utilisons souvent l'application de navigation GPS Waze et avons remarqué que, une fois arrivés à destination, la recherche d'une place de stationnement libre peut être longue et fastidieuse. Nous avons donc décidé de créer un modèle de segmentation d'image qui repère, à partir d'images satellites, les places de parking libres aux alentours de la destination. Bien qu'il n'existe pas, à notre connaissance, de source de données d'image satellite en temps réel facilement accessible, nous avons néanmoins décidé d'entreprendre ce projet.

2. Création de la base de données

Le premier gros chantier de ce projet consistait à créer notre propre jeu de données labellisé via les différentes étapes suivantes :

2.1. Récupération d'images satellites

En utilisant *GoogleMaps*, nous avons récupéré 120 images satellites vierges (i.e; sans aucun label) de tailles quelconques. Ces images proviennent de différents endroits d'europe (e.g. France, Belgique, Allemagne, Suisse, etc.). Parmi ces 120 images, 100 ont été utilisées pour l'entraînement et la validation du modèle, tandis que les 20 dernières ont été réservées pour tester notre modèle. Nous les stockons respectivement dans les dossiers /data/images et data/test/images.

2.2. Récupération d'images satellites

En utilisant les logiciels *LabelBox* et *Label Studio*, nous avons annoté manuellement les places de parking libres sur chacune des 120 images satellites. Nous avons pour cela utilisé la méthode de segmentation sémantique avec des polygons (proposé par les deux logiciels) pour créer un masque contenant un seul label (i.e. parking), où chaque polygone correspond à une surface de parking libre sur l'image satellite.

2.3. Création des masques

Une fois les masques encodés pour chaque image satellite présentes dans notre base de données, nous avons exporté des deux logiciels utilisés l'information relative à ces masques dans des fichiers *json*. Au total, trois fichiers *json* ont été créés : un via l'application *LabelBox* (i.e. mask_maxime.ndjson) et deux via l'application *Label Studio* (i.e. mask.json et mask_test.json). Tous sont stockés dans le dossier /json.

A partir de ces fichiers *json*, nous avons converti l'information des masques au format *json* en image avec le notebook create_masks_from_json.ipynb. Les masques sous forme d'image sont stockés dans le dossier data/masks pour les deux premiers fichiers *json* (i.e. les données d'entraînement et de validation) et dans le dossier data/test/masks pour le troisième fichier *json* (i.e. les données de test).

Chacun des masques prend le même nom que son image correspondante, auquel nous avons ajouté le suffix "*_mask.png*".

2.4. Data augmentation

Pour entraîner, valider et tester notre modèle, il nous faut un nombre conséquent de données. Or, labelliser manuellement nos images satellites prend beaucoup de temps. Nous avons donc agrandi notre base de données initiale de 120 images annotées comme suite :

2.4.1. Redimensionnement des images et masques

Comme mentionné ci-dessus, nous avons récupéré les images satellites dans des tailles différentes. Pour aider l'entraînement du modèle, mais surtout pour démultiplier notre nombre d'images satellites, nous avons redimensionné celles-ci. L'objectif de la classe présente dans le notebook resize_images_masks.ipynb est le suivant : redécouper chaque image satellite initiale ainsi que son masque correspondant en plusieurs images carrées de taille uniforme (i.e. 256px x 256px), et ce en parcourant la totalité de l'image satellite initiale et en minimisant les chevauchement des nouvelles images créées. Une fois les sous-images et sous-masques créés, la classe sélectionne les 4 sous-images ayant la surface de parking la plus grande. Ainsi, pour chaque image satellite initiale, nous obtenons 4 sous-images en gardant un maximum d'informations. Les images sont enregistrées dans les dossiers /data/train/images, /data/val/images et /data/test/images et les masques dans les dossiers /data/train/masks, /data/val/masks et data/test/masks avec un suffixe *"_{num}"* ajouté derrière leur numéro d'image initial.

2.4.2. Transformation des images/masks

Une fois nos nouvelles images satellites et masques tronqués récupérés, nous avons continué notre procédé de data augmentation en appliquant une combinaison de transformations pixels et spatiales grâce la librairie *Albumentations.*

Les images en entrée sont en couleurs et ont donc 3 canaux RGB. Ainsi, les transformations pixels utilisées correspondent à soit une modification légère des intensités ou à un inversement l'ordre de ces 3 canaux.

En ce qui concerne les transformations spatiales, nous n'avons appliqué que des trandformations de type "rotation" mais avec des angles variés. Un des plus grands avantages de la rotation de la librairie *Albumentations* est que les bords de l'image non défini à cause de la rotation sont automatiquement complétés par la symétrie de l'image.

Au total, 7 nouveaux couples images-masques ont été créés pour chaque sous-image ce qui nous a permis de générer, à partir de 100 images satellites initiales, 2800 nouvelles images (100 x 4 x 7).

2.5. Séparation des données d'entraînement, de validation et de test.

Bien que nous ayons essayé d'augmenter la taille de notre jeu de données de la façon la plus optimal possible, le chevauchement des sous-images lors de l'étape de redimensionnement peut s'avérer plus conséquent sur certaines images initiales aux dimensions spécifiques. Pour éviter que deux sous-images qui se chevauchent se retrouvent séparées (i.e. une dans la base données d'entraînement et l'autre dans la base données de validation) et donc d'avoir une validation de notre modèle biaisée, nous avons décidé de séparer les images pour l'entraînement et la validation avant cette étape de redimensionnement dans le notebook resize_images_masks.ipynb. Nous avons donc choisi aléatoirement et sans remise 18 des 100 images satellites initiales pour constituer notre échantillon de validation.

Concernant les données de test, pour éviter tout risque de data leakage, nous avons décidé de tester notre modèle sur 20 images satellites différentes. Ces 20 images ont subi le même procédé de data augmentation que les données d'entraînement et de validation ce qui nous a permis d'avoir au total 560 nouvelles images (20 x 4 x 7) et d'être certain de n'avoir aucun problème de data leakage durant le test de notre modèle.

3. Création du modèle

Après lecture des articles sur le site *Towards Data Science* (réf. 1 à 4), nous avons choisi de suivre le tutoriel (réf. 6) qui implémente un modèle U-Net via la librairie *TernausNet* (réf. 7). En effet, ce modèle a l'avantage d'être adapté à notre projet spécifique.

Initialement, nous avions utilisé un modèle U-Net déjà pré-entraîné sur une tâche de masquage d'animaux. Afin de pouvoir le réutiliser pour notre projet, nous l'avions affiné (fine-tuning) avec quelques dizaines d'époques sur notre tâche de classification des places de parking libres. Lors des tests, les masques prédits étaient entièrement nuls, c'est-à-dire remplis uniquement de zéros.

Face à cette situation, nous avons donc pris la décision d'entraîner le modèle U-Net "from scratch" (i.e. à partir de zéro) sur nos données spécifiques. Bien que ces nouveaux résultats aient été presque nuls, certains pixels étaient tout de même, certes de manière un peu aléatoire, identifiés comme des places de parking libres.

Pour améliorer ces résultats, nous avons ensuite introduit un poids différencié pour la classe 1 (i.e. les places de parking libres), moins représentées par rapport à la classe 0(i.e. Le background). Après plusieurs itérations avec des ajustements de poids (passant de 30 à 25, puis diminuant progressivement jusqu'à 10), les résultats les plus prometteurs et les plus interprétables ont été obtenus avec des poids équilibrés à 10. Cet entraînement a été réalisé sur 60 époques, avec une diminution progressive de la taille des lots (i.e. batch sizes) de 64 à 32 puis à 16, et des taux d'apprentissage (i.e. learning rates) passant de 0.001 à 0.0003, puis à 0.0001.

4. Interprétation et critique des résultats

Les résultats obtenus sur le modèle le plus prometteur indiquent que le modèle commence à segmenter les images fournies. Il parvient à distinguer les routes et les espaces goudronnés des habitations et des espaces verts. Cependant, il ne réussit pas encore à identifier correctement les places de parking libres. Nous avons également observé que le modèle interprète souvent, à tort, les zones ombragées comme classe 0, et ce même si des places de parking peuvent s'y trouver.

De plus, les masques prédits ne correspondent pas parfaitement aux masques labellisés. Cela suggère que le modèle éprouve des difficultés à saisir avec précision les contours et les formes spécifiques des places de parking. Les prédictions apparaissent dispersées et bruitées, caractérisées par des points isolés plutôt que par des formes continues. Il semblerait que le modèle ne se sache pas généralisé correctement ou n'ait pas appris à reconnaître les caractéristiques de façon cohérente.

L'importante présence de bruit dans les masques prédits peut aussi indiquer un sur-ajustement aux particularités du jeu de données d'entraînement, qui peuvent être différentes de celles des jeux de données de validation et de test. Cette hypothèse est renforcée par le fait que notre jeu de données d'entraînement a été multiplié par 28 (4 x 7). Cela soulève la question de la diversité insuffisante au sein du jeu de données d'entraînement.

5. Pour aller plus loin

Concernant l'entraînement, prolonger la durée de ce dernier et enrichir notre jeu de données pourraient améliorer la capacité du modèle à se généraliser. Cela impliquerait de revenir à l'étape de labellisation manuelle des données pour garantir leur qualité.

Comme mentionné ci-avant, nous avons observé que le modèle éprouve des difficultés à traiter les zones ombragées. Il serait donc judicieux d'enrichir notre jeu de données en veillant à diversifier les images en termes de contraste, d'exposition, et d'ombres. L'objectif est de constituer un jeu d'entraînement représentatif et précisément annoté, afin d'éviter que le modèle n'apprenne à partir de bruits inutiles.

Une autre piste serait d'ajuster les hyperparamètres du modèle, bien que cela requiert des ressources computationnelles importantes – une limite que nous avons déjà atteinte avec *Google Colab* dans notre cas. Nous pourrions également envisager d'explorer d'autres architectures de réseaux de neurones, comme celles présentées dans l'article (réf. 4).

Actuellement, nous utilisons la fonction de perte BCELoss, couramment employée pour les tâches de classification binaire telles que la nôtre. Il pourrait être bénéfique de tester d'autres fonctions de perte ou métriques, telles que l'IOU-Loss mentionnée dans l'article (réf. 1). Ces alternatives pourraient être plus adaptées à notre objectif d'entraînement et d'évaluation, en se concentrant sur des aspects cruciaux de la segmentation, comme la précision des zones d'intérêt prédites, qui représentent une petite portion du masque dans notre cas.

Pour approfondir notre analyse, l'emploi de métriques quantitatives comme l'exactitude (i.e. accuracy), la précision (i.e. precision), le rappel (i.e. recall) et le score F1 serait pertinent. En effet, ces indicateurs permettraient d'évaluer les performances du modèle de manière plus détaillée et exhaustive.

6. Tester notre projet

Pour exécuter notre projet, veuillez suivre les étapes suivantes :

1. Exécution des notebooks préparatoires dans l'ordre suivant :
    - Le notebook create_masks_from_json.ipynb pour créer les masques à partir des fichiers *json*.
    - Le notebook resize_images_masks.ipynb pour redimensionner les images et les masques.
    - Le notebook data_augmentation_w_albumentation.ipynb pour appliquer l'augmentation des données à l'aide de la librairie *Albumentations*.

2. Exécution du notebook d'entraînement et de test du modèle :
    - Si vous souhaitez réentraîner le modèle, utilisez le notebook Train_Test.ipynb en exécutant uniquement la partie dédiée à l'entraînement du modèle.
    - Pour charger et tester le modèle préalablement entraîné et enregistré, ouvrez également le notebook Train_Test.ipynb, mais cette fois-ci, ignorez la partie d'entraînement et procédez directement au chargement et au test du modèle pré-enregistré.

7. Bibliographie

(1) Smith, J. (2020). "Efficient Image Segmentation using PyTorch - Part 1." Towards Data Science. https://towardsdatascience.com/efficient-image-segmentation-using-pytorch-part-1-89e8297a0923
(2) Smith, J. (2020). "Efficient Image Segmentation using PyTorch - Part 2." Towards Data Science. https://towardsdatascience.com/efficient-image-segmentation-using-pytorch-part-2-bed68cadd7c7
(3) Smith, J. (2020). "Efficient Image Segmentation using PyTorch - Part 3." Towards Data Science. https://towardsdatascience.com/efficient-image-segmentation-using-pytorch-part-3-3534cf04fb89
(4) Smith, J. (2020). "Efficient Image Segmentation using PyTorch - Part 4." Towards Data Science. https://towardsdatascience.com/efficient-image-segmentation-using-pytorch-part-4-6c86da083432
(5) Albumentations. https://albumentations.ai/
(6) Albumentations. (Documentation). "PyTorch Semantic Segmentation." https://albumentations.ai/docs/examples/pytorch_semantic_segmentation/
(7) TernausNet GitHub Repository. (GitHub). https://github.com/ternaus/TernausNet/tree/master
(8) PyTorch Tutorials. (Site Web). "Data Loading and Processing Tutorial." https://pytorch.org/tutorials/beginner/data_loading_tutorial.html