Segmentation de données LiDAR

POM 2022

C. Croz, encadré par J. Digne[a]

[a] *Université Claude Bernard Lyon 1, Lyon, France.*

8 juin 2022

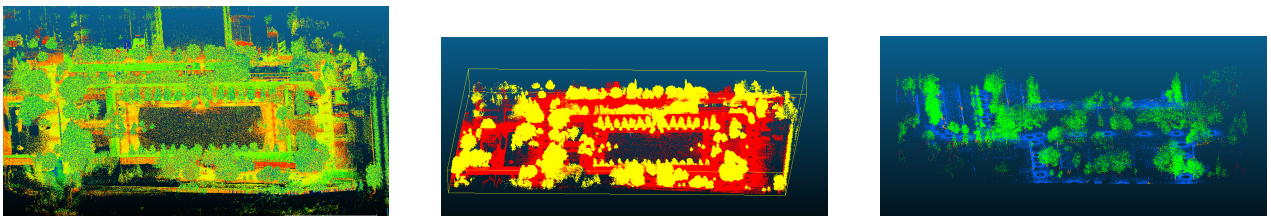
Abstract

L'objectif de ce projet a été de tester la capacité de l'algorithme PointNet à segmenter automatiquement des données LiDAR. Les données sont issues d'un campagne de numérisation d'un campus et représentent une scène contenant différents types d'objets (sol, bâtiments, véhicules, mobilier urbain, arbres), et l'algorithme a été testé sur sa capactié à différencier les arbres du reste de la scène.

1. Introduction

Le LiDAR est une technologie efficace pour la numérisation d'objets en 3D. Néanmoins, les points obtenus forment un nuage de points sans structure et leur analyse n'est pas triviale. PointNet(*7*) est un algorithme d'apprentissage profond pour la classification et la segmentation qui permet de traiter de telles données.

PointNet étant un algorithme d'apprentissage, il faut disposer de données d'entraînement étiquetées, entraîner l'algorithme, puis enfin lui fournir des données à segmenter.



(a) Données brutes (b) Données étiquetées (c) Résultat de l'évaluation

Figure 1 – État du nuage de point à différentes étapes clés du projet

2. Principe du LiDAR

LiDAR (*5*) est l'acronyme de l'expression anglaise « light detection and ranging », en français « détection et estimation de la distance par la lumière ». Le système sóriente dans une direction, envoie une impulsion laser, et mesure le retard de la réponse. Connaissant la vitesse de propagation du signal (ici la vitesse de la lumière), le capteur détermine alors la distance entre la cellule et le premier obstacle dans la direction considérée, et ajoute le point correspondant au nuage. Le capteur oriente ensuite sa cellule dans une autre direction et recommence l'opération.

Cette technologie offre plusieurs avantages :

— Le matériel est portable et relativement facile à déployer (fig. 2(a)).

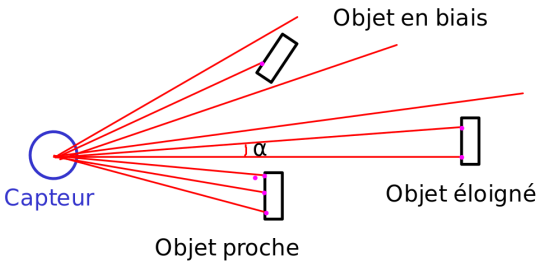
— Le niveau de détail est élevé. Pour le capteur utilisé (Riegl VZ-400), la précision annoncée par le constructeur est de l'ordre de $5mm$ et le pas angulaire est paramétrable avec un minimum de 0.0024 deg (fig. 2(d)).

Elle vient néanmoins avec plusieurs limites :

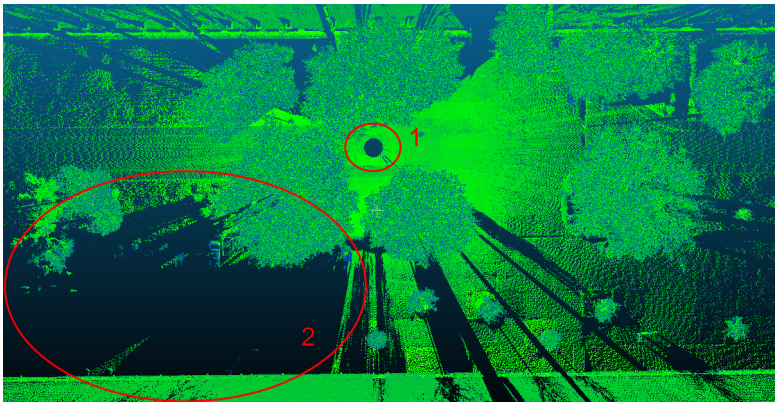
— Lorsque le capteur est fixe, il ne peut scanner que la partie de la scène visible depuis le capteur, ce qui génère des occlusions (fig. 2(c)).

— Les mesures sont effectuées avec un pas angulaire fixe. Ainsi, une surface proche du capteur sera plus densément échantillonnée qu'une surface plus éloignée. De même, une surface vue de face sera plus densément échantillonnée qu'une surface vue "de biais" (fig. 2(b)).

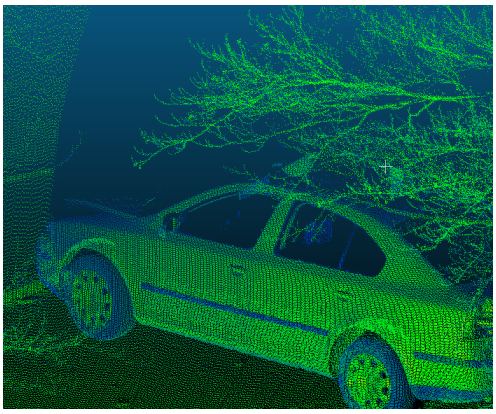
(a) Capteur utilisé pour obtenir les données de cette étude



(b) Différence d'échantillonnage



(c) Extrait d'un fichier obtenu. 1 : angle mort sous le capteur. 2 : occlusion causée par les objets de la scène



(d) Détails des branchages

Figure 2 – Description de la technologie LiDAR

3. Préparation des données

Pour cette étude, plusieurs sources ont été considérées, et c'est un scan du campus de la Jacobs University, à Bremen en Allemagne(*4*) qui a été retenu.

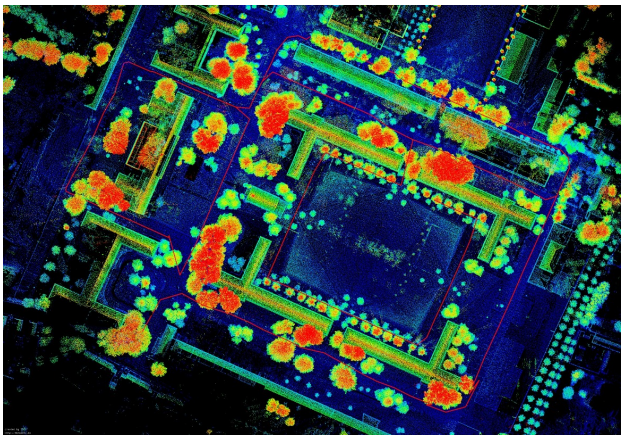


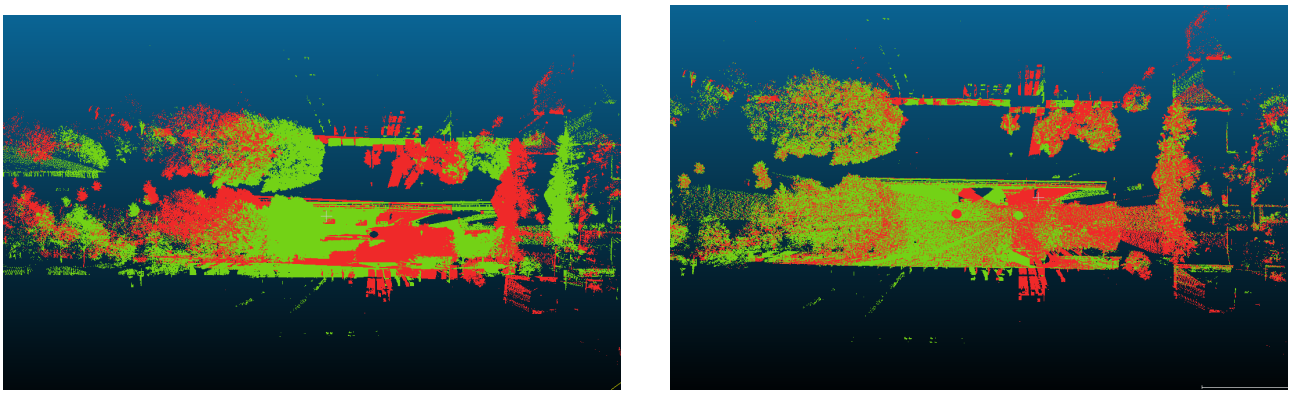
Figure 3 – Vue aérienne du scan du Campus de Bremen

3.1. Description des données originales

L'archive en téléchargement, annoncée comme contenant 122 fichiers de 1.2Gb, fait 173Go et contient 146 fichiers. Chaque fichier contient environ 14 millions de points, pour un total d'environ deux milliards de points, et correspond à une mesure, c'est à dire un "point de vue".

Pour chaque fichier, les points sont repérés dans un repère local lié au capteur. Les fichiers sont donc accompagnés de la description de la transformation qui permet de les replacer dans un repère global. Cette opération, appellée recalage (ou en anglais *registration*) permet de reconstruire a scène globale et est illustrée en figure 4.

Les données utiles au recalage n'étaient pas toujours correctes et une étape d'affinage du recalage a été nécessaire.



(a) Avant recalage (b) Après recalage

Figure 4 – Étape de recalage illustrée sur deux fichiers

Plusieurs étapes de sous échantillonnage ont aussi été nécessaires, pour obtenir une scène finale contenant environ 9 millions de points.

3.2. Établissement de la vérité terrain

La vérité terrain est la représentation de la réalité, traduite dans le formalisme du modèle étudié. Ici, la question étudiée est de savoir, pour chaque point, s'il fait partie d'un arbre ou non. L'établissement de la vérité terrain consiste donc à étiqueter l'ensemble des données, c'est-à-dire spécifier, pour chaque point, la catégorie à laquelle il appartient, ici "**tree**" (pour les arbres) et "**not_tree**" (pour le reste de la scène).

Cette opération n'est évidemment pas réalisée point par point, mais en sélectionnant des volumes et en étiquetant tous les points contenus dans le volume, et a été effectuée en utilisant le logiciel CloudCompare(*2*).

Cette étape reste très fastidieuse, certaines zones peuvent être délicates à définir (limite tronc/sol, végétation/toiture) et certains objets ont été par moment mal étiquetés (lampadaires, poteaux électriques. Voir fig 5).

3.3. Fichiers d'entrée PointNet

Une fois les données ainsi préparées, il a fallu les mettre dans le format requis par PointNet (voir 4), c'est-à-dire pour chaque point lister ses $n = 1023$ plus proches voisins et sauvegarder ces listes dans des fichiers séparés. Les fichiers sont ensuite placés dans 4 dossiers différents :

— points **tree** utilisés pour l'entraînement ;
— points **tree** utilisés pour l'évaluation ;
— points **not_tree** utilisés pour l'entraînement ;
— points **not_tree** utilisés pour l'évaluation.

À l'issue de la vérité terrain, les données ne sont pas structurées et il s'agit simplement d'une liste de points. La recherche de plus proches voisins en temps raisonnable a donc nécessité de stocker les points sous la forme d'un KD-Tree et a été réalisée en utilisant la bibliothèque nanoflann(*1*).

Certains points ont aussi été écartés : les *outliers*, c'est à dire les points isolés, ainsi que les surdensités qui sont surtour localisées autour du capteur.

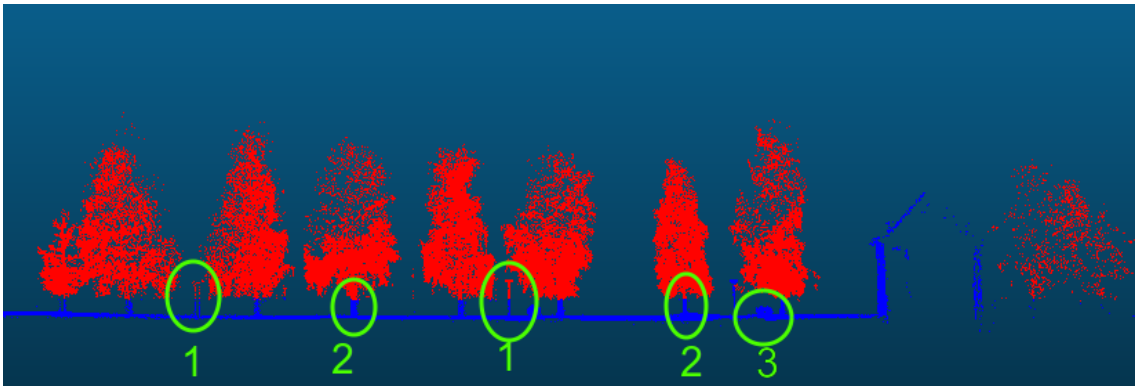


Figure 5 – Détails de la vérité terrain. 1 : mobilier urbain étiqueté "tree". 2 : variabilité de la hauteur du tronc. 3 : buisson étiqueté "not_tree"

Étant donné les capacités limitées de calculs à disposition pour l'exécution de PointNet, il n'a pas été possible d'utiliser l'intégralité des points préparés. Ainsi, seulement 200 000 points d'entrainement ont été sélectionnés aléatoirement pour la génération des fichiers PointNet, et 1 000 000 points pour l'évaluation, pour un total de 1 200 000 fichiers générés. De plus, cette étape génère énormément de redondance, puisque les coordonnées de chaque point sont recopiées 1024 fois.

Les différentes étapes de préparation des données ont modifié la taille des données manipulées. Ces tailles sont visibles dans le tableau 1.

Étape	Taille sur le disque dur	Nombre de points
Données téléchargées	173 Go	2 000 000 000
Sous-échantillonnage	900 Mo	19 000 000
Suppression des points inutiles	320 Mo	9 000 000
Fichiers PointNet (générés)	45 Go	1 200 000
Fichiers PointNet (potentiels)	336 Go	9 000 000

Table 1

4. L'algorithme PointNet

4.1. Principe général

PointNet est un algorithme de classification et de segmentation. L'intérêt de cet algorithme est qu'il s'applique à des ensembles de points sans structure particulère et qu'il respecte entre autre les invariances par rotation et translation, et que l'ordre dans lequel les points sont listés n'influe pas sur le résultat.

L'algorithme repose sur l'architecture illustrée en figure 6, et est constitué de plusieurs perceptrons multi-couche (en anglais, *multilayer perceptron* ou MLP (*6*)). k représente le nombre de catégories considérées. Dans notre cas, $k = 2$ ("tree", "not_tree").

Pour un fichier d'entrée, PointNet effectue d'abord un traitement qui identifie 1024 caractéristiques locales attachées à chacun des n points. Deux sous-réseaux impliqués dans cette tâche, notés T-Net sur la figure 6, rendent l'algorithme robuste aux rotations de l'ensemble de points. On obtient ainsi une matrice $n \times 1024$, qui est réduite en un vecteur de 1024 caractéristiques globales (*global features*) par *max pooling*, ce qui assure l'invariance par permutation des n points du fichier d'entrée. Finalement, à partir de ces caractéristiques, un score pour chacune des k catégories est calculé. Le point est considéré comme appartenant à la catégorie ayant obtenu le meilleur score.

4.2. Implémentation retenue

L'algorithme ayant été choisi, il a fallu séléctionner une implémentation. C'est l'implémentation originale de 2016 par l'auteur de l'algorithme qui a été retenue. Celle-ci est réalisée en Python et s'appuie sur la bibliothèque d'apprentissage TensorFlow.

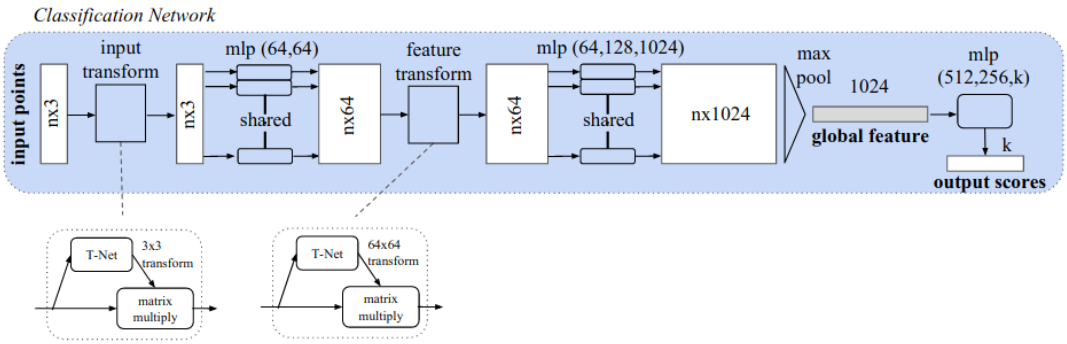


Figure 6 – Description du pipeline de PointNet pour la classification

Le code disponible n'a nécessité que de mineures modifications pour pouvoir charger les fichiers de points. Le *loader*, c'est-à-dire le script de chargement des fichiers afin de fournir les données à PointNet, doit aussi effectuer une normalisation : les points sont translatés pour être centrés en zéro et redimensionnés afin de tenir dans la sphère unité.

L'implémentation a cependant été écrite en Python2.7 (version actuelle 3.9) et TensorFlow 1.0.1 (version actuelle 2.9.1), ce qui implique donc un code sous-optimal, les versions récentes ayant significativement amélioré les performances.

Un autre problème rencontré est l'incompatibilité de TensorFlow avec les carte graphiques ne supportant pas CUDA. Les calculs ont donc été intégralement effectué sur le CPU. Les algorithmes d'apprentissage étant relativement gloutons mais fortement parallélisables, ils sont généralement effectués sur GPU, ce qui permet de traiter beaucoup plus de données et donc d'améliorer les performances.

5. Résultats et discussion

5.1. Entraînement

La phase d'entraînement consiste à fournir à l'algorithme des données étiquetées, et corriger petit à petit les erreurs de prédictions de l'algorithme. Concrètement, un fichier d'entrée est soumis à l'algorithme, qui fournit une prédiction. L'écart entre la prédiction et la valeur attendue est ensuite utilisé pour améliorer la prédition suivante, et ce processus est itéré un certain nombre de fois.

Étant donné la limitation en terme de puissance de calcul, l'entraînement s'est fait sur seulement 20 000 points (10 000 points "tree" et 10 000 points "not_tree"), et a été répété sur 15 *epochs*. Cet entraînement, effectué sur douze cœurs CPU à 2200 MHz, a duré 6h40.

Les points ont été divisés en *batchs* de 2000 points (1000+1000), et des mesures de performances ont été réalisées après chaque batch et epoch.

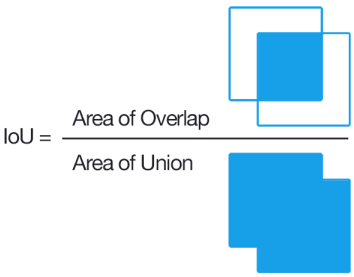


Figure 7 – Principe de l'IoU. Source : *Adrian Rosebrock*, Intersection over Union (IoU) for object detection

Les trois quantités suivies sont :

— La précision (*Accuracy*) : la fraction de points correctement catégorisés par l'algorithme. La valeur est entre 0 et 1, plus la valeur est grande, plus l'algorithme est performant.

— Le IoU (*Intersection over Union*) pour la catégorie "tree" : il s'agit du rapport du nombre de points effectivement "tree" *et* catégorisé "tree" (intersection ou *overlap*) sur le nombre de points effectivement "tree" *ou* catégorisé "tree" (union) (fig. 7). La valeur est aussi entre 0 et 1, plus la valeur est grande, plus l'algorithme est performant.

— L'erreur de prédiction (*loss*) qui représente l'écart entre le score k calculé par l'algorithme et la valeur attendue via la vérité terrain. Plus cette valeur (qui est assimilable à une norme ici, et donc est toujours positive) est basse, meilleur est l'algorithme.

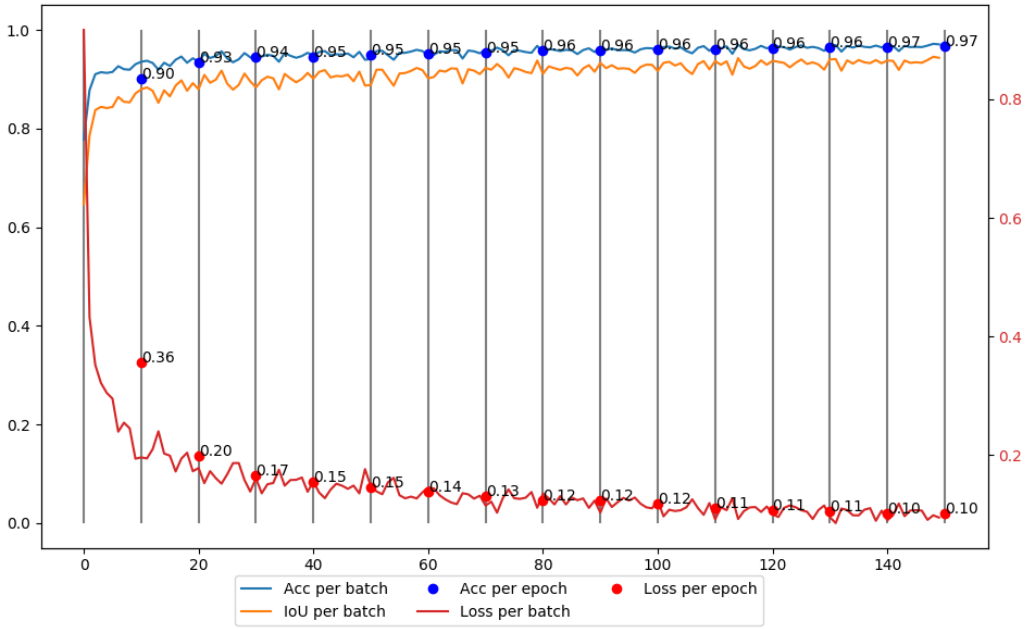


Figure 8 – Performances lors de la phase d'entraînement

5.2. Évaluation

Lors de la phase d'évaluation, les fichiers sont simplement soumis à l'algorithme, et les prédictions sont enregistrées. Ceci nécessite donc moins d'opérations, et a pu être effectué sur 200 000 points dans un premier temps, puis sur 1 000 000 de points, avec les performances compilées dans le tableau 2.

Nombre de points	Durée	Accuracy	Tree IoU	Not_tree IoU
200 000	1h35	0.9612	0.9256	0.9248
1 000 000	7h15	0.9614	0.9261	0.9254

Table 2 – Évaluation de l'algorithme après entraînement sur 20 000 points

Les résultats de l'évaluation de chaque point sont enregistrés, ce qui permet leur visualisation. La figure 9, générée après l'évaluation de l'algorithme sur 1 000 000 de points, permet de faciliter l'interprétation des performances de l'algorithme (voir 5.3).

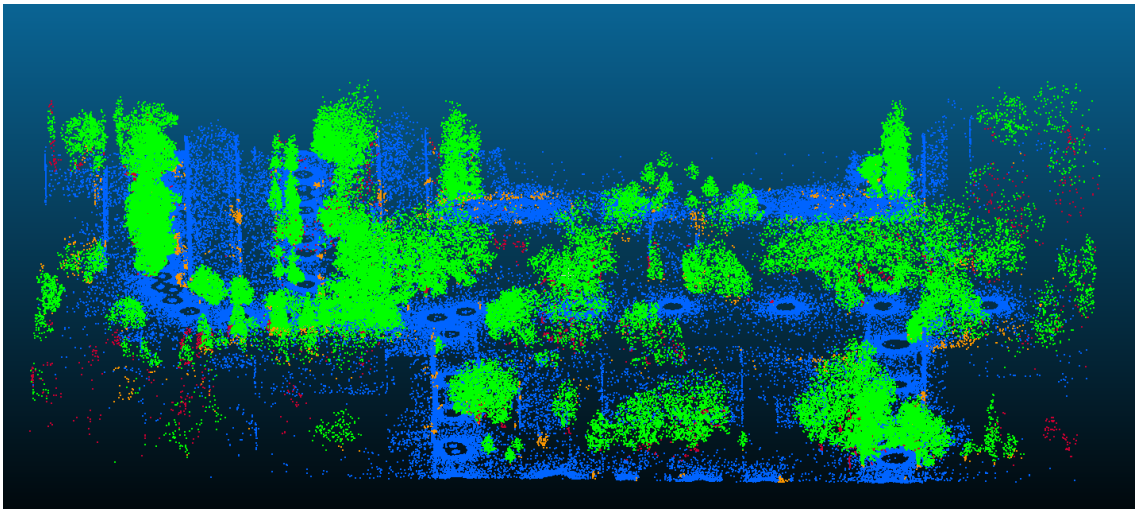


Figure 9 – Résultat de l'évaluation (1 000 000 de points). En **vert**, les points "tree" correctement catégorisés, en **bleu** les points "not_tree" correctement catégorisés, en **rouge**, les points "tree" mal catégorisés, en **orange** les points "not_tree" mal catégorisés.

5.3. Discussion

Globalement, les résultats sont bons, avec un taux de succès de reconnaissance de plus de 95%. L'algorithme a un taux de succès sensiblement plus élevé pour les arbres que pour le reste de la scène.

Les points pour lesquels l'algorithme tend à se tromper sont généralement regroupés en "taches". Une partie de ces erreurs semble due à des erreurs lors de la vérité terrain. En effet, certains buissons ont été étiquetés "tree", et d'autres "not_tree" et il semble que l'algorithme a décidé de les considérer comme "tree" (voir fig. 10(a)). On constate aussi des défauts autour des zones ambiguës de la vérité terrain : le bas des troncs, certains lampadaires (voir fig. 10(b) et 10(c)).

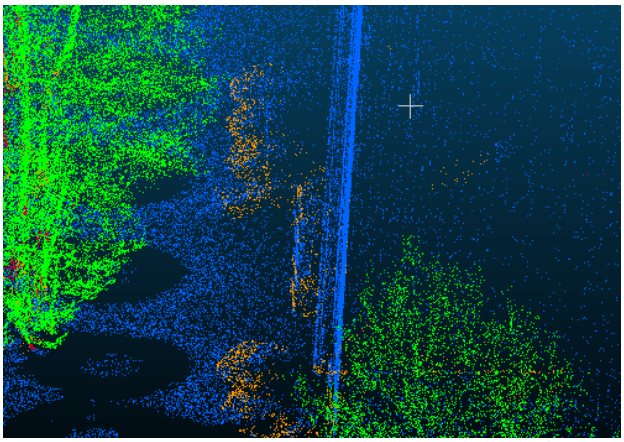
Les angles de certains bâtiments accumulent aussi des défauts(voir fig. 10(d)). Une piste d'amélioration est le recalage. En effet, une différence géométrique importante entre les arbres et les bâtiments est le fait que les arbres ont un profil volumétrique, alors que les bâtiments ont un profil surfacique. Or, les erreurs de recalage peuvent donner une "épaisseur" aux surfaces de bâtiments et induire l'algorithme en erreur.

Deux pistes d'amélioration se dessinent donc :

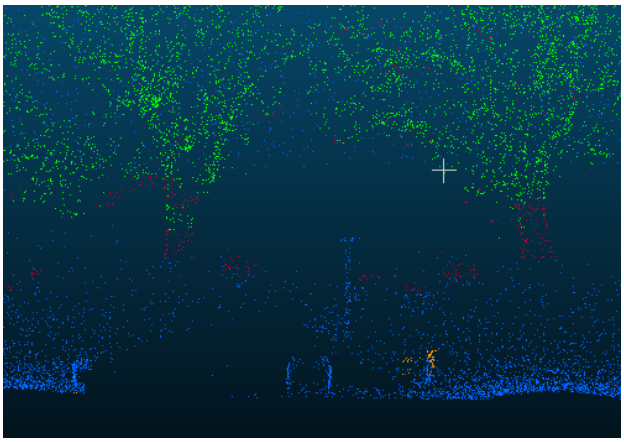
— Le recalage : utiliser des algorithmes de recalage automatique robustes, comme par exemple *Iterative Closest Point* (*3*) ;

— La vérité terrain : expliciter les règles de décision, en particulier sur la définition claire d'un "arbre" (inclusion ou non des buissons, jusqu'où considérer le tronc) et allouer plus de ressource à cette tâche.

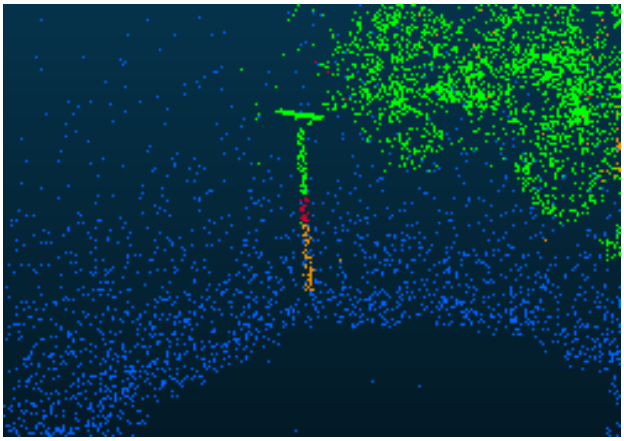
Il pourrait aussi être intéressant de tester les performances en faisant varier la taille de l'ensemble d'apprentissage et le nombre d'epochs, étude qui demanderait cependant un accès à des ressources plus adaptées à l'apprentissage profond (grille de calcul, GPU) pour être réalisée dans de meilleures conditions.



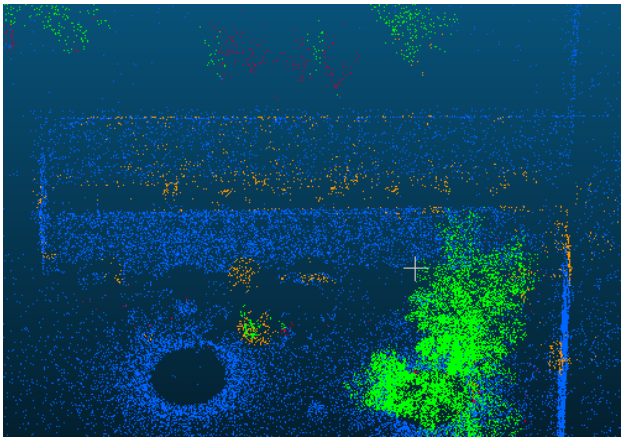
(a) Buissons



(b) Incertitude au niveau du tronc



(c) Mobilier urbain



(d) Toitures

Figure 10 – Défauts présents dans les résultats de l'évaluation

6. Conclusion

L'utilisation de PointNet pour la tâche considérée semble donc possible. Un entraînement sur un nombre raisonnable de points permet en effet des résultats significatifs sur une quantité 500 fois plus grande avec un taux de réussite supérieur à 95%.

Le cahier des charges initial du projet prévoyait l'étude d'un autre algorithme, UNet (*8*), en transformant les données 3D en une carte de hauteur, et la comparaison des performances en terme de précision et de temps d'entraînement et d'évaluation. Cette tâche n'a pas pu être réalisée en raison de la sous-estimation de la quantité de travail nécessaire à l'étude de PointNet.

Ce projet a été l'occasion pour moi de découvrir les problématiques, les outils et les méthodes de la segmentation automatique. Il a aussi été mon premier projet en *machine learning*. C'est aussi le projet qui m'a amené à traiter le plus grand volume de données jusqu'à présent, et à ressentir la dure réalité de la limitation en terme de puissance de calcul.

Remerciements

Je tiens à remercier chaleureusement Julie Digne pour sa disponibilité et ses conseils au cours de ce projet, l'équipe pédagogique encadrant le POM, mes relecteurs, et plus généralement tous mes professeurs dont les enseignements ont contribué directement ou indirectement à la réussite de ce projet.

Références

1. J. L. Blanco, P. K. Rai, *nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees*, `https://github.com/jlblancoc/nanoflann`, 2014.
2. *CloudCompare*, version 2.10.3, (`http://www.cloudcompare.org/`).
3. *Iterative Closest Point*, Wikipedia (2022 ; `https://en.wikipedia.org/wiki/Iterative_closest_point`).
4. P. K.C, D. Borrmann, J. Elseberg, A. Nuchter, *Robotic 3D Scan Repository*, Jacobs University (2022 ; `http://kos.informatik.uni-osnabrueck.de/3Dscans/`).
5. *Lidar*, Wikipedia (2022 ; `https://en.wikipedia.org/wiki/Lidar`).
6. *Multilayer perceptron*, Wikipedia (2022 ; `https://en.wikipedia.org/wiki/Multilayer_perceptron`).
7. C. R. Qi, H. Su, K. Mo, L. J. Guibas, *arXiv preprint arXiv:1612.00593* (2016).
8. O. Ronneberger, P. Fischer, T. Brox, *arXiv:1505.04597* (2015).