Université de Bretagne Occidentale

Faculté des Sciences et Techniques

Département Informatique

Master 2 - Systèmes Intelligents, Interactifs et Autonomes

Rapport de Stage
-
Complétion de nuages de points représentant des scènes industrielles

Auteur :
Pierre Cornen

Encadrants :
Cédric Buche
Romain Cazorla

13 septembre 2023

Remerciements

Je voudrais remercier en premier Cédric BUCHE ainsi que Romain CAZORLA pour m'avoir accepté dans ce stage, ainsi que de m'avoir accompagné lors de la réalisation du projet.
Je voudrais remercier aussi l'équipe `SMARI` de `Segula Technologies` pour leur accueil et les différentes réunions organisées.
Je remercie l'équipe du `CERV` pour leur accueil et pour m'avoir montré plus en détails le monde de la recherche.
Et je remercie Erell COTTOUR avec qui j'ai travaillé pendant les 5 mois du stage et qui m'a aidé et soutenu pour la réalisation de ce projet.

1 Résumé :

Les nuages de points sont devenus de plus en plus prisés par les chercheurs pour représenter des environnements 3D, de part leur précision et leur légèreté en terme de mémoire. Cependant, des trous peuvent rendre ces derniers incompréhensibles et inutilisables. C'est pourquoi les chercheurs ont proposé plusieurs méthodes de complétion de nuages de points. Mais, étant un domaine encore très récent, certains points ne sont pas encore traités tels que : la complétion de scènes entières ou encore la non-modification des points du nuages d'entrée. Ainsi, ce papier propose une analyse de certaines méthodes de la littérature, ainsi qu'une nouvelle approche composée de 2 étapes : la détection de trous via un réseau de segmentation binaire, puis l'adaptation d'une méthode de la littérature afin de prendre en compte la détection.

mots clés : *point cloud, voxel grid, completion, binary segmentation, machine learning, deep learning, hole detection*

Table des matières

1 Résumé : 1

2 Introduction : 1

3 État de l'art : 1
- 3.1 Résumé . . . 1
- 3.2 Introduction . . . 2
- 3.3 Algorithmes Conventionnels . . . 4
  - 3.3.1 La transformation du nuage de points . . . 5
  - 3.3.2 La détection de trous . . . 5
  - 3.3.3 La complétion . . . 6
  - 3.3.4 Avantages / Désavantages . . . 7
- 3.4 Algorithmes Deep Learning . . . 8
  - 3.4.1 Architectures Proposées . . . 9
  - 3.4.2 Fonctions d'objectif utilisées . . . 12
  - 3.4.3 Jeux de données utilisés . . . 13
  - 3.4.4 Résultats . . . 14
- 3.5 Bilan . . . 16

4 Méthode de travail : 17

5 Analyses des méthodes existantes : 19
- 5.1 `PCN`[2] . . . 20
  - 5.1.1 Installation . . . 20
  - 5.1.2 Entraînement . . . 21
  - 5.1.3 Résultats . . . 23
- 5.2 `PoinTr`[11] . . . 26
  - 5.2.1 Installation . . . 26
  - 5.2.2 Entraînement . . . 26
  - 5.2.3 Résultats . . . 27
- 5.3 `LLD`[12] . . . 28
  - 5.3.1 Installation . . . 29
- 5.4 Traitement des données réelles . . . 29
- 5.5 Analyse de la courbure . . . 30
- 5.6 Conclusion . . . 32

6 Conception d'un détecteur de trous : 33
- 6.1 Génération de grille de voxels . . . 34
  - 6.1.1 Lecture d'un nuage de points . . . 34
  - 6.1.2 Transformation en grille de voxels . . . 35

6.1.3 Possibilité d'ajouter des trous 37
6.1.4 Récupération d'un tenseur représentant le nuage de points 39
6.1.5 Optimisations 39
6.2 Implémentation d'un réseau de neurones pour détecter les trous 40
6.2.1 Reprise de l'architecture `U-Net`[20] 40
6.2.2 Création des jeux de données 42
6.2.3 Entraînements 43
6.2.4 Résultats 45

7 Conclusion : 46

Références 48

8 Annexes : 51
8.1 Guide d'installation de l'environnement de travail 51
8.2 Guide d'installation de CloudComPy (API python de CloudCompare) 54

Table des figures

1 Exemple d'un nuage de points de Kitti [1] avec un zoom sur des voitures à compléter, tiré de PCN [2] 3
2 Exemple d'une complétion par PConv, tiré de l'article [5] . . . 4
3 Schéma représentant les étapes d'**algorithmes conventionnels** 5
4 Exemple du problème avec l'inpainting par profondeur, tiré de l'article [4] 6
5 Exemple concret du MaxPooling 8
6 Visualisation de l'encodeur de PCN, tiré de l'article [2] 10
7 Méthodologie utilisée lors du stage 18
8 L'architecture d'un "Encodeur / Décodeur" 20
9 Évolution de `CD` sur les 121 dernières epochs lors de l'entraînement de `PCN`[2] 23
10 Exemple de complétions de catégories vues(a) et non-vues(b) par `PCN` 25
11 Évolution des métriques lors de l'entraînement de `PoinTr`[11] . 27
12 Exemple de complétions par PoinTr 28
13 Exemple d'un nuage de points du jeu de données `KITTI`[1] . . 30
14 Visualisation de la courbure de nuages de points via CloudCompare 31
15 Schéma de l'architecture globale du projet 33
16 Diagramme UML de la classe `PointCloud` 34
17 Diagramme UML des classes `Voxel` et `VoxelGrid` 35
18 Exemple de voxélisation avec plusieurs dimensions 36
19 Diagramme UML de la classe `SphericalHole` 37
20 Exemple de grilles de voxels avec des trous 38
21 Schéma de l'architecture `U-Net`[20] 41
22 Évolution de l'`accuracy` et du `loss` pendant l'entraînement final de 15 epochs 44
23 Exemple de détections de trous sur des objets du jeu de données de test 45
24 Exemple de détections de trous sur des tuyaux réels 45
25 Exemple d'une détection de trous sur une scène 46
26 Diagramme UML de l'architecture du projet 56

Liste des tableaux

1 Évaluation de la complétion d'objets sur le PCN benchmark [2] avec la métrique **CD**, tirée de LLD [12] 15

2 Évaluation de la complétion d'objets sur le MVP benchmark [10] avec la métrique **F1-Score**, tirée de LLD [12] 15
3 Composition de `PCN-BenchMark`[2] 22
4 Évaluation du modèle PCN entraîné avec la métrique `CD` $(1e-3)$ 23
5 Évaluation du modèle PCN avec la métrique `EMD` $(1e-3)$ . . . 24
6 Évaluation du modèle `PoinTr` entraîné 27
7 Moyenne des différences de courbures des nuages de points . . 31
8 Comparaison du temps d'exécution moyen (en sec) de la voxélisation entre des fichiers `ASCII` et `Binaire` 40
9 Résultats obtenus par notre modèle final sur les jeux de données de test 44

2 Introduction :

Ce stage s'introduit dans la réalisation du projet `SMARI`, proposé par `SEGULA Technologies`. Le but derrière ce projet est d'aider les ingénieurs qui modélisent les plans et les modèles 3D de zones industrielles en diminuant leur charge de travail. Ainsi `SEGULA Technologies` propose, avec le projet `SMARI`, d'utiliser les nuages de points. L'idée est de scanner les zones industrielles via des caméras LiDar afin d'acquérir des nuages de points. Ces derniers sont ensuite segmentés puis utilisés pour produire des modèles `CAO` (**C**onception **A**ssistée par **O**rdinateur) des zones industrielles.
Cependant, les phases d'acquisition peuvent générer des nuages de points incomplets. En effet des trous peuvent apparaître lors du scan des zones industrielles, de part les caractéristiques de la caméra mais aussi de part les objets de la scène qui peuvent obstruer la caméra. Ce défaut diminue les performances des traitements appliqués sur les nuages de points, tel que la segmentation. C'est pourquoi il serait intéressant d'appliquer une méthode de complétion sur les nuages de points en amont et de voir si cela a un impact sur les résultats de la segmentation.

Le sujet du stage porte, ainsi, sur la complétion de nuages de points. Dans la littérature, la plupart des méthodes de complétion ne traite que de nuages de points représentant qu'un objet unique et non des scènes entières et encore moins des zones industrielles, étant donné que cela représente des données difficiles d'accès. Le but du stage est alors d'analyser les méthodes existantes afin de voir s'il serait possible d'en adapter une pour compléter des nuages de points représentant d'abord des objets industriels et ensuite des zones.
Le stage a été proposé par le `CERV` (**C**entre **E**uropéen de **R**éalité **V**irtuelle) et est en collaboration avec `SEGULA Technologies`.

3 État de l'art :

3.1 Résumé

Les nuages de points sont devenus de plus en plus prisés par les chercheurs pour représenter des environnements 3D, de part leur précision et leur légèreté en terme de mémoire. Cependant, des trous peuvent rendre ces derniers incompréhensibles et inutilisables. C'est pourquoi il est nécessaire d'avoir des méthodes de complétion de nuages de points. Ainsi, ce papier détaille les différentes approches utilisées par les chercheurs afin d'obtenir des méthodes

de plus en plus performantes, jusqu'à atteindre la méthode de l'état de l'art proposant une complétion d'objets et de scènes. Ce papier met en évidence certains points non traités par les chercheurs : les objets "fantômes" et les nuages de points atypiques, afin de proposer d'éventuelles perspectives qui seront étudiées lors d'un stage.

mots clés : *point cloud, completion, inpainting, network, deep learning*

3.2 Introduction

Cela fait maintenant plusieurs années que nous pouvons constater un engouement pour les nuages de points. De part leur efficacité à représenter l'environnement de manière précise tout en étant peu demandant en mémoire. Étant donné que ces derniers sont composés d'un ensemble de points ayant comme attributs leur position dans l'espace, leur couleur et autres informations.

Pour pouvoir générer un nuage de points, il est nécessaire d'utiliser des dispositifs spécifiques que nous pouvons séparer en 2 types. Le premier sont les caméras RGB-D avec l'arrivée de la Kinect en 2008. Ces caméras sont composés de 2 parties : l'une est une caméra RGB classique et permet de capturer la couleur des points et la deuxième est un laser infra-rouge permettant de récupérer la distance / profondeur des points par rapport à la caméra. Le second type sont les caméras laser, tel que le LiDar. Le but ici est de parcourir la scène avec un laser et, en connaissant la vitesse de la lumière, calculer la distance des points par rapport à la caméra (exemple avec fig. 1).

Il existe plusieurs applications utilisant les nuages de points. L'une d'entre elles est en lien avec le stage : Segula Technologies veut utiliser les nuages de points pour modéliser des installations industrielles, afin de faciliter la tâche de suivi et de modification de ces dites installations en ayant une représentation fidèle grâce aux nuages de points.

Une autre application est celle de la computer vision et plus précisément la conduite autonome. Pour pouvoir analyser leur environnement et réagir rapidement, les voitures autonomes sont dotées de LiDars générant des nuages de points. Comme nous pouvons le voir avec le jeu de données Kitti [1].

Pour le bon fonctionnement de ces applications, il est nécessaire d'avoir un nuage de points représentant parfaitement l'environnement. Cependant cela est rarement le cas, des trous peuvent être plus ou moins présents et rendent la scène 3D incomplète. Cela est dû à plusieurs facteurs : la précision

de la caméra, le positionnement de la caméra, l'occlusion des objets ou encore l'opacité des objets. Pour y remédier, l'une des solutions envisageables est de capturer l'environnement plusieurs fois avec différents points de vue. Mais cela ajoute un défaut car, avec des petites erreurs de callibrations, nous pouvons nous retrouver avec des objets "fantômes" dans le nuage de point (des objets capturés 2 fois). De plus, il restera toujours des trous ne pouvant pas être comblés. Ainsi, l'utilisation d'algorithmes de complétion est nécessaire (exemple avec fig. 1).

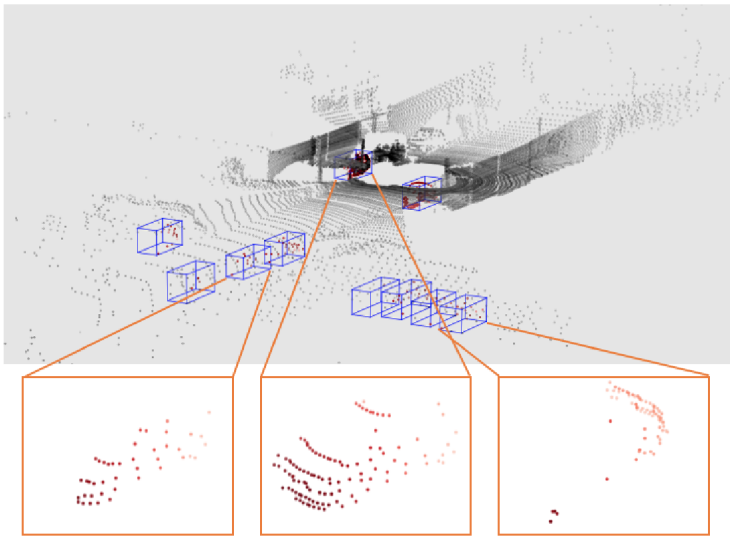

FIGURE 1 – Exemple d'un nuage de points de Kitti [1] avec un zoom sur des voitures à compléter, tiré de PCN [2]

Le but de ce papier, de ce fait, est de recenser les différentes méthodes de complétion de nuages de points à ce jour. Afin de voir : Quelles ont été les approches utilisées par les chercheurs ? Quelle méthode est l'état de l'art de la complétion de nuages de points ? Et quelles seraient les pistes d'améliorations ?

Pour cela, cette bibliographie suivra un axe historique en montrant l'évolution des méthodes de complétion de nuages de points. Ainsi dans un premier temps, nous parlerons des **algorithmes conventionnels** n'utilisant aucun réseau de neurones profonds. Puis, dans un second temps, nous verrons que grâce à PointNet [3] des **algorithmes de deep learning** ont vu le jour et sont, à l'heure actuelle, les méthodes principalement étudiées.

3.3 Algorithmes Conventionnels

L'une des premières approches pour la complétion de nuages de points a été de s'inspirer de ce qui se faisait déjà au niveau des images 2D : l'inpainting.

L'inpainting est une méthode permettant de combler les trous se trouvant sur une image. Le but derrière est de pouvoir soit reconstituer une image détériorée, soit de supprimer un élément d'une image. Pour ce faire, il faut appliquer un masque sur l'image indiquant les zones à combler et ensuite y appliquer l'algorithme d'inpainting. Ce dernier est basé sur des patches, l'image est séparée en une grille de cellules de taille fixe. Une fois cela fait il faut parcourir les cellules correspondant à la frontière entre les trous et l'image et trouver les cellules d'image les plus similaires pour ensuite compléter le bord des trous, puis ainsi de suite jusqu'à ce qu'il n'y ai plus de trous. [4]

Maintenant, l'inpainting d'image 2D utilise des réseaux de neurones profonds et permet d'obtenir des résultats extrêmement convaincants. Comme nous pouvons le voir avec les derniers travaux réalisés par Nvidia présentant leur algorithme : PConv [5] (fig. 2).

(a) Image avec Masque (b) Image complétée par PConv (c) Image réelle

FIGURE 2 – Exemple d'une complétion par PConv, tiré de l'article [5]

Ainsi les auteurs des papiers [4] et [6] ont proposé des **algorithmes conventionnels** d'inpainting sur les nuages de points, n'utilisant pas de réseau de neurones. Ces derniers peuvent être séparés en 3 étapes décrites par le schéma suivant (fig. 3) :

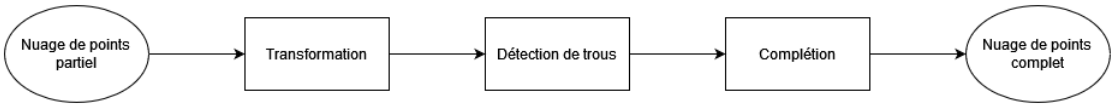

FIGURE 3 – Schéma représentant les étapes d'**algorithmes conventionnels**

3.3.1 La transformation du nuage de points

Implémenter un tel algorithme sur les nuages de points ne se fait pas sans compromis. Car de part la nature des nuages de points à être non-ordonnés, il est difficile d'y appliquer des **algorithmes conventionnels**. C'est pourquoi il faut appliquer une transformation du nuage de points avant tout autre traitement.

Ainsi, Zequing Fu et al. [6] transforment le nuage de points en une grille volumétrique (Voxélisation). Le but étant de générer une grille en 3D où chaque cellule peut contenir un point du nuage. Cela permet, ainsi, d'ordonner le nuage de points au détriment d'une perte de précision et ajoute une surcharge de travail.

De l'autre côté, David Doria et al. [4] n'utilisent pas directement le nuage de points et proposent de traiter l'image RGB-D correspondante.

3.3.2 La détection de trous

Une fois la transformation effectuée la prochaine étape est de détecter les trous à compléter.

En ce qui concerne [4], les auteurs n'ont pas implémenté de détection automatique de trous. De ce fait, comme pour l'inpainting sur image 2D, c'est à l'utilisateur de préciser les zones à combler en appliquant un masque.

Pour ce qui est de [6], les auteurs proposent une détection automatique des trous. Pour cela, ils partent de la voxélisation du nuage de points et projettent ce dernier vers une image 2D indiquant la profondeur des points. La projection se fait suivant un axe ayant été calculé via les normales des points. Sur cette image, les pixels ne correspondant pas à un point du nuage correspondent donc aux trous. La totalité d'un trous est, donc, représentée par l'ensemble des pixels voisins correspondant à des trous.
Cependant, certains trous détectés sont dûs à la faible densité du nuage de

points. C'est pourquoi les auteurs ont rajouté un critère de densité pour les éliminer de la liste.

3.3.3 La complétion

Après avoir détecter les trous, il faut les compléter. Ici les 2 papiers [4] et [6] proposent d'implémenter une approche basée sur des patches.

Étant donné que [4] travaille sur la représentation en image RGB-D du nuage de points, l'algorithme reste très semblable à ce que nous avons décrit en section **3**. Cependant cette fois-ci l'image est une image RGB-**D**, la profondeur a une importance car elle permet de représenter la position des points et ainsi la structure du nuage de points. Pour trouver le patch le plus similaire à celui à compléter, [4] propose de comparer la profondeur directement. Mais cela peut engendrer plusieurs problèmes :
Imaginons qu'il faut compléter le milieu d'un mur en biais, les zones similaires en profondeur seront la zone avant le milieu et la zone après le milieu du mur. Or l'objectif serait d'avoir une profondeur ayant la moyenne de ces 2 zones pour compléter le milieu (fig. 4).

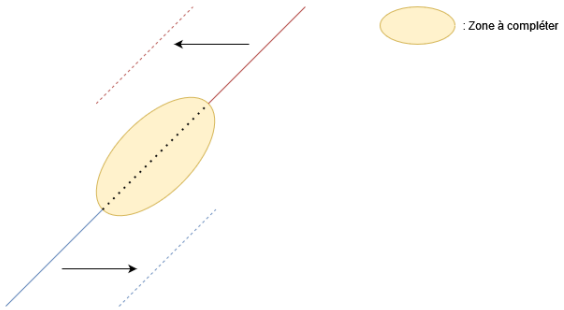

FIGURE 4 – Exemple du problème avec l'inpainting par profondeur, tiré de l'article [4]

C'est pourquoi, les auteurs de [4] ont finalement comparé avec le gradient de la profondeur. Le gradient d'une image étant la moyenne des valeurs voisines.

Pour ce qui est de la méthode de Zeqing Fu et al. [6], elle applique le même procédé que l'inpainting 2D, mais transposé sur la grille volumétrique générée. Ainsi cette dernière est séparée en cubes, correspondant aux patches. Une fois cela fait il faut chercher les cubes correspondant le plus aux cubes à compléter, la comparaison se fait en se basant sur la théorie spectrale des

graphes. Pour ensuite, fusionner les cubes afin de combler les trous et ainsi avoir un nuage de points complet.

3.3.4 Avantages / Désavantages

Pour conclure cette première partie sur les **algorithmes conventionnels**, nous allons aborder les avantages et les désavantages de ces méthodes.

Tout d'abord, l'avantage premier de ces méthodes inspirées par l'inpainting est de combler les trous via les données existantes du nuage de points. Cela permet d'avoir un résultat proche de ce que pourrait proposer le dispositif utilisé lors de la capture du nuage de points. Cependant c'est aussi l'un de ces désavantages, car ce procédé ne peut pas fonctionner pour des zones uniques à compléter. De plus, le défaut le plus important est l'étape de transformation. Le fait qu'il faut transformer le nuage de points en une grille volumétrique ne permet pas de profiter des points forts du nuage de points : la précision et le faible usage de mémoire. Au contraire, les grilles volumétriques, étant très gourmandes en mémoire, limitent le nombre de points pouvant être traités. En continuant sur les désavantages, nous avons pu voir que la méthode [6] utilise des critères fixes pour la détection de trous, ce qui rend ce genre de méthodes moins généralistes.

En ce qui concerne le temps d'exécution, seul David Doria et al. [4] présentent leurs résultats et obtiennent donc des temps aux alentours de 1-2 minutes. Ce qui rend l'utilisation de ce genre de méthodes en temps réel impossible.

De ce fait, Ramesh Ashok Tabib et al. [7] proposent de garder les mêmes étapes présentées dans cette partie, hormis la transformation, et d'utiliser un réseau de neurones pour la détection de trous. Ainsi cela permet de se libérer des critères fixes.

Cependant ce papier ne rentre pas assez dans les détails sur l'implémentation, mais il reste intéressant car il montre vers quoi les chercheurs tendent à aller : le **Deep Learning**. Mais contrairement à [7] (qui est un cas atypique), les chercheurs ne gardent pas les étapes des algorithmes conventionnels et partent sur une approche 100% **Deep Learning**.

3.4 Algorithmes Deep Learning

Une autre approche envisagée par les chercheurs a été d'utiliser les réseaux de neurones profonds, analysant le nuage de points tel un humain, afin de compléter les zones manquantes. Dans un premier temps, l'idée était d'utiliser des réseaux de neurones convolutifs, c'est dernier étant utiliser pour la compréhension et le traitement d'images. Cependant ces réseaux de neurones demandent d'avoir des données extrêmement ordonnées, ce qui n'est pas le cas des nuages de points. Ainsi, la première solution apportée a été la même que pour les **algorithmes conventionnels** : la transformation du nuage de points en un ensemble de données régulier et ordonné. Cela implique aussi les mêmes défauts : trop demandant en mémoire et une perte de précision lors de la transformation.

La deuxième solution nous a été apportée en 2017 par Charles R. Qi et al. grâce à PointNet [3], cette méthode nous présente une approche pour traiter un nuage de points directement par un réseau de neurones profonds. Étant donné qu'un nuage de points est représenté par un ensemble de points non-ordonné, il est donc invariant aux permutations. Ainsi il faut utiliser des opérations symétriques respectant cette condition. De ce fait, le point clé de PointNet [3] a été l'utilisation de l'opération *MaxPooling* : Elle prend en entrée un ensemble de données et retourne une version réduite contenant seulement les caractéristiques essentiels (fig. 5).

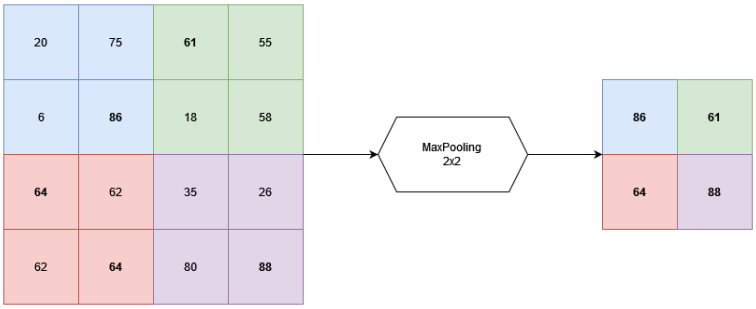

FIGURE 5 – Exemple concret du MaxPooling

Les auteurs de PointNet [3] ont pu analyser ce que récupérer le *MaxPooling* et ont constaté qu'il gardait, comme caractéristiques, les points clés correspondant au squelette de l'objet / de la scène.

Ainsi, PointNet [3] est un réseau de neurones permettant de classifier des objets, de segmenter et d'analyser sémantiquement une scène. Et propose des résultats égales, voir surpassant, les méthodes antérieures. C'est avec cet

exploit que PointNet [3] marque un nouvel âge sur l'utilisation des réseaux de neurones profonds pour traiter les nuages de points.

De ce fait, à partir de ce moment, la quasi-totalité des méthodes de complétion se sont inspirées de PointNet [3]. Chacune proposant une approche différente, nous allons donc en présenter 6 d'entre elles : PCN [2], MSN [8], PF-Net [9], VRCNet [10], PoinTr [11] et LLD [12].

Pour ce faire, nous allons détailler les points importants de ces méthodes. Dans un premier temps, nous allons voir les architectures proposées par ces papiers. Dans un deuxième temps, nous verrons les fonctions d'objectif utilisées ainsi que les métriques. Puis, nous finirons par voir les jeux de données utilisées et nous comparerons les résultats des méthodes pour voir laquelle est la plus performante.

3.4.1 Architectures Proposées

Encodeur & Décodeur

L'une des premières méthodes à avoir vue le jour est PCN [2]. Elle a permis de poser les bases en proposant une architecture basée sur un Encodeur et un Décodeur.

Le but de l'encodeur est de prendre le nuage de points partiel pour en tirer un vecteur de caractéristiques. Pour cela, l'encodeur de PCN [2] est composé de 2 couches PointNet [3] :

La première prend en entrée une matrice $\mathbf{P}$ de $n * 3$, où chaque ligne correspond aux coordonnées (x, y, z) des n points du nuage. Cette matrice $\mathbf{P}$ rentre ensuite dans un **M**ulti**L**ayer **P**erceptron (**MLP**) de 2 couches linéaires, avec une activation ReLU, pour transformer chaque point p_i en un point de caractéristique f_i dans le vecteur $\mathbf{F}$. Puis le vecteur $\mathbf{F}$ est ensuite traité dans un *MaxPooling* afin de générer un vecteur de caractéristiques globales, $\mathbf{g}$.

La deuxième couche prend en entrée la concaténation de $\mathbf{F}$ et $\mathbf{g}$ et passe cette matrice dans un **MLP** et un *MaxPooling* identiques à la première couche pour obtenir un vecteur des caractéristiques final, $\mathbf{v}$.

Pour une meilleure compréhension, Wentao Yuan et al. [2] ont proposé une visualisation de ce que récupère l'encodeur (fig. 6). Et nous pouvons constater qu'il récupère un squelette détaillé de l'objet.

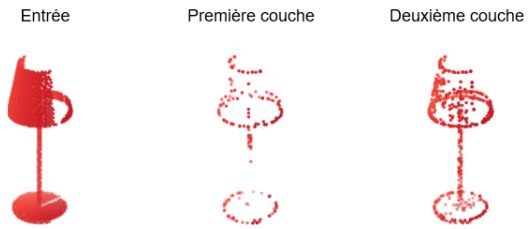

FIGURE 6 – Visualisation de l'encodeur de PCN, tiré de l'article [2]

Le but du décodeur est de prendre le vecteur **v** calculé par l'encodeur est de générer un nuage de points complet. Les auteurs de PCN [2] ont pu étudier 2 décodeurs : un décodeur entièrement connecté et un décodeur basé sur le pliage. Et ils ont constaté que le premier permet de prédire un nuage de points représentant la forme globale de l'objet (un nuage de points grossier), tandis que le deuxième permet de prédire un nuage de points plus détaillé.

Ainsi PCN [2] combine ces 2 décodeurs pour bénéficier de leurs avantages. Pour cela il traite le vecteur **v** en le faisant passer dans un décodeur entièrement connecté pour générer un nuage de points grossier. Une fois cela fait, il passe chaque point du nuage dans un décodeur basé sur le pliage pour prédire un nuage de points détaillé, représentant le nuage de point complet.

Par la suite, MSN [8], PF-Net [9] et VRCNet [10] ont continué sur cette architecture Encodeur / Décodeur, tout en ajoutant, modifiant certains détails :

MSN [8] utilise un décodeur se basant sur le *Morphing*. Ce qui permet de prédire des formes continues, au lieu d'un nuage de points. Ces formes sont ensuite transformées en un nuage de points. Cela permet, ainsi, d'avoir une meilleure géométrie de l'objet compléter.

PF-Net [9] travaille sur différentes échelles de précision du nuage de points. Ainsi l'encodeur est un encodeur multi-résolution prenant en entrée 3 versions du nuage de points plus ou moins détaillées. Puis le décodeur génère lui aussi 3 versions du nuage de points complété. De plus, contrairement aux autres méthodes, PF-Net [9] ne génère que la partie manquante du nuage de points. Ce qui permet de ne pas modifier les points déjà présents dans le nuage.

Pour finir, contrairement aux autres méthodes, VRCNet [10] utilise un encodeur probabiliste ce qui permet d'avoir une incertitude à la prédiction des points manquants.

Cependant le défaut de l'architecture de base Encodeur / Décodeur est

qu'il ne permet pas de prendre en entrée des nuage de points de taille diverse. De ce fait des chercheurs ont pensé à utiliser une autre architecture : Transformeur.

Transformeur

L'architecture Transformeur est majoritairement utilisée pour le traitement de langage naturel (**NLP**). Elle a permis de faire des avancées dans différents domaines tels que : la traduction automatique ou bien la génération de texte. Pour cela, l'architecture fonctionne via un système d'attention et permet de donner plus ou moins de poids aux différentes parties de l'entrée, en fonction de leur pertinence.

Ainsi Xumin Yu et al., les auteurs de PoinTr [11], ont voulu utiliser cette architecture. Cependant, il y a 2 points à traiter avant de pouvoir utiliser un Transformeur sur les nuages de points :

Tout d'abord, le Transformeur prend en entrée une séquence de mots à une dimension. Il faut donc représenter le nuage de points en une séquence à une dimension. Une première solution naïve serait de faire une séquence des coordonnées des points. Mais, le Transformeur est extrêmement gourmand. De ce fait, PoinTr [11] représente le nuage de points en un ensemble de points proxies, c'est-à-dire, une liste de points clés récupérés grâce à un algorithme : **F**urthest **P**oint **S**ample (**FPS**).

Et le deuxième point est de faire en sorte que le Transformeur soit adapté à la géométrie, ce dernier étant dédié avant tout aux **NLP**. Pour ce faire, PointNet [3] utilise les travaux de PointNet [3] que nous avons décrit précédemment.

Avec ceci, PoinTr [11] a réussi à obtenir des résultats dépassant les méthodes précédentes.

En continuant sur cette lancée, LLD [12] propose 2 architectures : une directe à base d'Encodeur / Décodeur et une via des Transformeurs. De plus le point clé de LLD [12] est l'utilisation du *Neighbor Pooling*, au lieu du *Max-Pooling*. Au lieu de prendre la caractéristique maximale, le *Neighbor Pooling* concatène les caractéristiques des points voisins pour en produire une seule. Cela permet de perdre moins de données. Pour finir, un autre point important de LLD [12] est qu'il propose une complétion de scène, contrairement aux autres méthodes ne traitant que des objets. Pour cela, chaque point du nuage prédit à une catégorie d'objet associée afin de traiter sémantiquement la scène.

3.4.2 Fonctions d'objectif utilisées

Une fois l'architecture réalisée, il faut entraîner les modèles. Pour cela, il est important de définir des fonctions d'objectif.

Dans le cadre de la complétion de nuage de points, il faudra comparer les nuages de points prédits et les nuages de points complets. Il existe 2 métriques pouvant réaliser cela :

Chamfer **D**istance (**CD**) calcule la distance entre 2 nuages de points. Pour ce faire, on effectue la somme des distances entre chaque point du premier nuage et son point le plus proche du deuxième nuage, ainsi que la même somme dans l'autre sens. Cela permet, ainsi donc, de mesurer la proximité entre 2 nuages de points. **CD** est moins sensible aux différences mais peu gourmand et peut s'appliquer même si les 2 nuages ont une taille différente.

Earth **M**over's **D**istance (**EMD**) mesure la quantité d'étapes nécessaires pour transformer un nuage de points en un autre. Cela est fait en essayant de regrouper les points des 2 nuages en couples où chaque couple est composé d'un point de chaque nuage ayant la distance la plus courte. **EMD** est plus sensible aux différences mais très gourmand, car il essaye d'optimiser un problème, et ne peut s'appliquer que sur des nuages de points de même taille (de part le fait qu'il faut réaliser des couples).

PCN [2], PF-Net [9], VRCNet [10], PoinTr [11] et LLD [12] utilisent **CD** pour leurs fonctions d'objectif afin de comparer soit les nuages de points grossiers générés avec leurs versions réelles, soit les nuages de points complets générés avec les nuages de points complets réels.

En ce qui concerne **EMD**, PCN [2] l'utilise avec **CD** lors de la comparaison des nuages de points grossiers. Tandis que MSN [8] l'utilise pour les 2 comparaisons.

Certaines méthodes utilisent d'autres fonctions d'objectif pour obtenir de meilleurs résultats, comme par exemple LLD [12] qui proposent 2 fonctions d'objectif supplémentaires :

La première permet au modèle de générer les points dans un certains ordres, du point le plus visible au point le plus obstrué. Rien qu'avec cette nouvelle fonction, les auteurs de LLD [12] ont pu constaté une amélioration sur la précision de la complétion.

La deuxième permet au modèle de traiter sémantiquement la complétion, comme nous l'avons vu précédemment LLD [12] permet la complétion de scène en associant aux points du nuages générés une catégorie afin de savoir quel objet est représenté par les points. Ainsi ils mettent en place une fonction

d'objectif pour traiter cette partie.

3.4.3 Jeux de données utilisés

L'un des points le plus importants dans la réalisation de réseaux de neurones profonds est la sélection de jeux de données pertinents pour pouvoir entraîner les modèles, ainsi que de les évaluer.

En ce qui concerne l'entraînement, nous avons vu que les fonctions d'objectif comparent les nuages de points prédits avec les nuage de points complets, on dit que l'entraînement est supervisé. C'est pourquoi, il faut avoir un jeux de données proposant des nuages de points complets. Malheureusement, nous ne pouvons pas utiliser des données tirées de captures réelles, car elles ne peuvent pas proposer des nuages de points entièrement complets. Ainsi, les chercheurs se sont tournés vers des données virtuelles proposées par :

ShapeNet [13], une base de données composées de plus de 51 000 modèles d'objet 3D de hautes qualités, annotés et séparés en 55 catégories. ShapeNet [13] a été spécifiquement conçu pour l'apprentissage de réseaux de neurones profonds.

Cependant, il faut transformer ces modèles 3D en un ensemble de nuages de points partiels et complets. Étant donné que PCN [2] a été la première méthode, elle a proposé un jeu de données utilisant ShapeNet [13]. Ainsi le jeu de données est composé de 30 974 modèles tirés de 8 catégories : avion, armoire, voiture, chaise, lampe, canapé, table et navire. Ensuite un nuage de points complets est généré en prenant uniformément 16 384 points sur le mesh des modèles. Pour ce qui est des nuages de points partiels, ils sont générés en utilisant une rétro-projection d'une image de profondeur en 2.5D. Grâce à cela, nous obtenons un résultat pouvant se rapprocher de données réelles capturées. 8 nuages de points partiels sont ainsi générés pour chaque modèle. Par la suite, ce jeu de données présenté par PCN [2] est utilisé par les autres méthodes pour l'entraînement, ainsi qu'en tant que benchmark pour pouvoir les évaluer entre elles.

D'autres chercheurs ont, eux aussi, proposé leurs jeux de données. Tels que Liang Pan et al., les auteurs de VRCNet [10], qui ont proposé le jeu de données **M**ulti-**V**iew **P**artial (**MVP**). Ce dernier se base aussi sur ShapeNet [13] et est composé de plus de 100 000 nuages de points partiels et complets. Il y a 3 différences notoires avec le jeu de données PCN [2] :

La première est le placement des caméras virtuelles pour la génération du nuage de point partiel, **MVP** [10] utilise 26 caméras fixes se trouvant sur

un cercle englobant le modèle 3D. Une rotation aléatoire est effectuée pour avoir un placement différents sur les 26 caméras. En fixant les caméras, cela permet d'être sûr d'avoir des points de vues différents et en utiliser 26 permet de couvrir la plupart des parties du modèle 3D.

La deuxième différence est sur le nombre de catégories, **MVP** [10] utilise 16 catégories : les 8 proposées par le jeu de données PCN [2], ainsi que 8 autres (lit, banc, bibliothèque, bus, guitare, moto, pistolet et skateboard). Cela rend l'entraînement et l'évaluation plus difficile.

La dernière différence est que **MVP** [10] propose différentes résolutions de nuages de points : 2 048, 4 096, 8 192, 16 384 points. Afin de tester l'efficacité des méthodes sur différentes échelles.

Une fois l'entraînement réalisée, il est intéressant de pouvoir tester les réseaux de neurones profonds sur des données réelles. Ainsi, les chercheurs utilisent d'autres jeux de données tels que : Kitti [1] qui est une collection de données capturées pour de la conduite autonome et contient des données LiDar pouvant être utilisées. Ou bien ScanNet [14] proposant plus de 2.5 millions de scènes 3D représentées par des nuages de points. Les scènes sont majoritairement des espaces d'intérieurs tels que des maisons, des bureaux ou des appartements.

3.4.4 Résultats

Pour pouvoir comparer les différentes méthodes présentées, nous avons repris les tableaux comparatifs présentés dans le papier de LLD [12]. Grâce à ces tableaux, nous allons pouvoir évaluer les méthodes présentés, hormis PF-Net [9].

Le premier tableau (table 1) utilise le jeu de données PCN [2], ainsi que la métrique **C**hamfer **D**istance (**CD**) pour évaluer : PCN [2], MSN [8], PoinTr [11] et LLD (directe / Transformeur) [12]. Étant donné que **CD** mesure la distance entre 2 nuages de points, le but est d'avoir la plus petite valeur.

Le deuxième tableau (table 2) utilise le jeu de données **M**ulti-**V**iew **P**artial (**MVP**) [10], ainsi que la métrique **F1-Score** pour évaluer : PCN [2], MSN [8], VRCNet [10], PoinTr [11] et LLD (directe / Transformeur) [12]. Le **F1-Score** est une métrique couramment utilisée pour évaluer les réseaux de neurones profonds, il représente le taux de positifs correctement prédits.

Méthodes	Avion	Armoire	Voiture	Chaise	Lampe	Canapé	Table	Navire	Moy.
PCN[2]	5.50	10.63	8.70	11.0	11.34	11.68	8.59	9.67	9.64
MSN[8]	5.60	11.96	10.78	10.62	10.71	11.90	8.70	9.49	9.97
PoinTr[11]	4.75	10.47	8.68	9.39	7.75	10.93	7.78	7.29	8.38
LLD (directe)[12]	5.34	**9.20**	**8.26**	8.96	9.40	**10.46**	7.54	8.56	8.47
LLD (Transformeur)[12]	**4.43**	10.03	8.28	**8.96**	**7.29**	10.55	**7.31**	**6.85**	**7.96**

TABLE 1 – Évaluation de la complétion d'objets sur le PCN benchmark [2] avec la métrique **CD**, tirée de LLD [12]

Méthodes	Avion	Armoire	Voiture	Chaise	Lampe	Canapé	Table	Navire	Moy.
PCN[2]	0.816	0.614	0.686	0.517	0.455	0.552	0.646	0.628	0.614
MSN[8]	0.879	0.692	0.693	0.599	0.604	0.627	0.730	0.696	0.690
VRCNet[10]	0.928	0.721	0.756	0.743	0.789	0.696	0.813	0.800	0.781
PoinTr[11]	0.888	0.681	0.716	0.703	0.749	0.656	0.773	0.760	0.741
LLD (directe)[12]	0.926	0.738	0.766	0.783	0.837	0.709	0.829	0.821	0.801
LLD (Transformeur)[12]	**0.942**	**0.753**	**0.780**	**0.799**	**0.851**	**0.725**	**0.844**	**0.836**	**0.816**

TABLE 2 – Évaluation de la complétion d'objets sur le MVP benchmark [10] avec la métrique **F1-Score**, tirée de LLD [12]

Avec ces tableaux, nous pouvons constater que LLD [12], plus précisément sa version Transformeur, s'en sort extrêmement bien sur les 2 évaluations. Cela démontre la pertinence du choix d'une architecture Transformeur, ainsi que de l'utilisation de *Neighbor Pooling* au lieu de *MaxPooling*.

3.5 Bilan

Avec cette bibliographie, nous avons pu voir que toutes les approches se sont inspirées de méthodes utilisées dans d'autres domaines tel que l'analyse et le traitement d'images 2D, avec l'inpainting et les réseaux de neurones convolutifs, voire même le traitement de langage naturel (**NLP**). Nous pouvons cités 3 approches majeures :

La première fut de retranscrire l'inpainting à base de patches sur les nuages de points. Cependant, de part la nécessité de transformer les nuages en des données ordonnées, telles que les grilles volumétriques, ainsi que la détection de trous utilisant des critères fixes rendent ces méthodes très gourmandes en ressources (mémoire / temps) et peu généralistes.

La deuxième approche fut d'utiliser des réseaux de neurones convolutifs. Cependant, au début le problème de transformation était toujours présent car ces réseaux nécéssitent aussi des données très régulières et ordonnées. Mais grâce aux travaux de PointNet [3], cela est devenu possible de traiter directement des nuages de points via des réseaux de neurones profonds en utilisant des couches **MLP** et le *MaxPooling*. Ainsi plusieurs méthodes (PCN [2], MSN [8], PF-Net [9] et VRCNet [10]) ont vu le jour en suivant une architecture d'Encodeur / Décodeur et ont proposé des petites modifications pour essayer d'améliorer leur précision et leur performance.

Puis PoinTr [11] est arrivé en proposant d'utiliser l'architecture Transformer, normalement utilisée pour le **NLP**, et à montrer des résultats dépassant les autres méthodes. De ce fait, d'autres ont continué sur cette lancée comme LLD [12] qui propose d'utiliser un *Neighbor Pooling* au lieu d'un *MaxPooling* pour perdre moins d'informations lors de la récupération des caractéristiques. De plus LLD [12] est la seule méthode à proposer de compléter des scènes et des objets, au lieu de ne compléter que des objets. Et, à ce jour, LLD [12] présente les meilleurs résultats ce qui en fait la méthode d'état de l'art pour la complétion de nuages de points.

Cependant plusieurs points n'ont pas été abordés :

- Nous avons vu lors de l'introduction que des objets "fantômes" peuvent apparaître lors d'une capture à multi-vues d'un nuage de points. Hors, aucune des méthodes présentées n'a étudié ce problème.
- Les jeux de données utilisées ne contiennent quasiment que des objets et des scènes du quotidien. Cependant, le but du stage est de travailler sur des données représentant des installations industrielles, des données assez atypiques en sortes.

Ainsi, pour conclure, nous avons pu voir quelles ont été les approches prises par les chercheurs pour trouver des méthodes de complétion de nuages

de points de plus en plus performantes. En passant par l'inpainting, puis par les réseaux de neurones convolutifs grâce à PointNet [3], les auteurs de LLD [12] proposent une méthode performante pour compléter des objets, ainsi que des scènes, utilisant une architecture de Transformeur.

Pour les perspectives liées au stage, il serait intéressant de travailler sur le traitement des objets "fantômes" pouvant apparaître lors de capture multi-vues. De plus, un point important à étudier sera d'évaluer l'efficacité de ces méthodes avec des données atypiques, tels que des nuages de points d'installations industrielles. Pour finir, les auteurs de [7] ont proposé une approche assez particulière, s'inspirant des étapes des algorithmes conventionnels en y ajoutant un réseau de neurones pour la détection de trous. Il serait très intéressant de continuer sur cette approche et fusionner avec les réseaux de neurones profonds. De plus, il serait intéressant de reprendre les travaux de LLD [10] en ce qui concerne la récupération des caractéristiques via *Neighbor Pooling* et de l'implémenter sur les autres méthodes pour voir s'il y a des améliorations notoires.

4 Méthode de travail :

L'objectif du stage est donc de proposer une solution envisageable pour la complétion de nuages de points représentant des scènes industrielles. Le stage ne se déroulant que sur 5 mois, ce qui est assez court pour ce genre de projet. Il est nécessaire de bien s'organiser en suivant une méthodologie réfléchie en amont, afin de ne pas se perdre et d'optimiser son temps. Ainsi nous avons suivi la méthodologie suivante :

Méthodologie

Dans un premier temps, nous avons mis en place les différents objectifs nécessaires à la réalisation du projet. Nous en avons principalement 3 :

- La détection de trous, afin de savoir quelles zones d'une scène sont à compléter ou pas
- La complétion de trous, pour compléter les zones détectées sans affecter les autres zones
- Appliquer la méthode sur une scène entière

Avec ces objectifs en tête, dans un deuxième temps, nous avons analyser les méthodes de complétion déjà existantes dans la littérature (PCN[2], PoinTr[11] et LLD[12]). Le but étant de voir quelle méthode pourrait être utilisée dans notre projet, en regardant si elle respecte les objectifs précé-

demment cités.

Une fois les méthodes analysées, dans un troisième temps, nous allons concevoir notre méthode. Puis il faudra la tester et la valider via différentes expériences :

— Tester sur des jeux de données synthétiques
— Tester sur des objets industriels réels
— Tester sur une scène entière

Ainsi cette méthodologie peut se résumer sous la forme du schéma suivant (fig. 7).

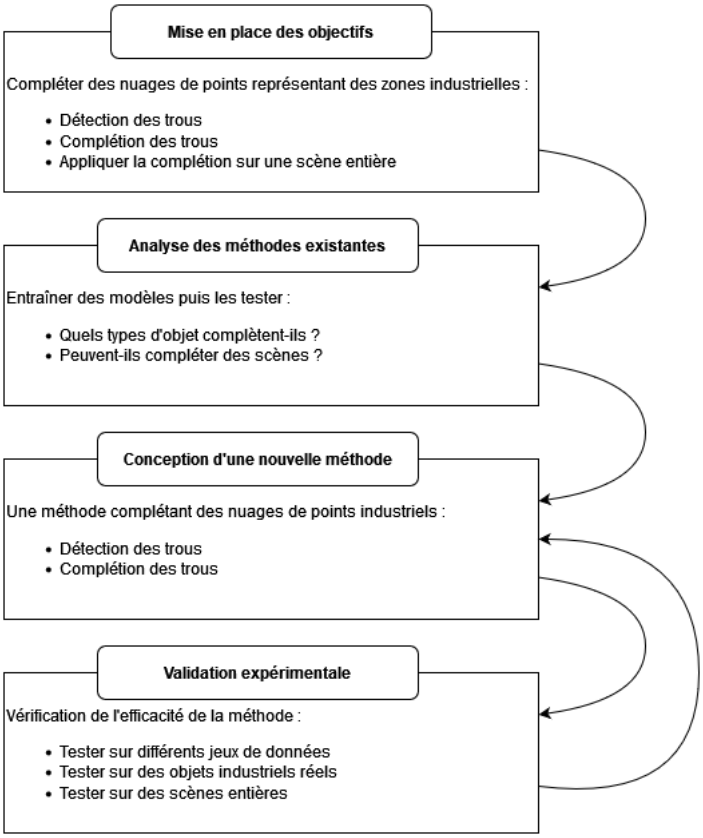

FIGURE 7 – Méthodologie utilisée lors du stage

Caractéristiques de l'environnement de travail

Nous avons travaillé sur un ordinateur prêté par le `CERV`, composé d'un Intel Core I5 7ème générations et d'une carte graphique Geforce GTX 1080 (8Go VRAM). La totalité du projet a été réalisé et exécuté sur le coeur Linux de Windows (WSL2), avec :

- Ubuntu 20.04
- Python 3.10
- PyTorch 1.12.1
- CUDA 11.3
- Open3D

Un guide d'installation se trouve en annexe 8.1

5 Analyses des méthodes existantes :

Notre première mission a été donc d'analyser les méthodes de complétion de nuage de points référencées dans la littérature. Lors de l'état de l'art, nous avons pu voir qu'il existe une multitude de méthodes plus ou moins anciennes. Ne pouvant pas essayer et analyser l'intégralité de ces dernières, il est nécessaire de faire un choix. Pour ce faire nous nous sommes appuyés sur plusieurs critères :

Le premier est l'utilisation de réseaux de neurones, car nous avons pu voir que depuis 2017 avec `PoinNet`[3] la totalité des méthodes utilisent des réseaux de neurones profonds ; Le deuxième critère est l'architecture des réseaux utilisés, il serait intéressant de pouvoir analyser différents types d'architecture ; Et le troisième critère est l'éventail des données pouvant être traitées par la méthode, en effet nous avons vu que la quasi-totalité des méthodes ne traitent que des nuages de points représentant un seul objet. Mais dernièrement nous avons pu voir que `LLD`[12] arrive à traiter des petites scènes.

Ainsi, en suivant ces critères, nous avons retenu 3 méthodes :

- `PCN`[2] : première méthode de complétion utilisant une architecture "Encodeur / Décodeur"
- `PoinTr`[11] : première méthode utilisant des "Transformeurs"
- `LLD`[12] : une méthode utilisant des "Transformeurs" et traitant des petites scènes tel qu'un bureau rempli de plusieurs objets

L'objectif, maintenant, est d'installer, entraîner et tester ces 3 méthodes. Cela aura pour but, dans un premier temps, de nous familiariser avec l'environnement de travail en apprenant comment fonctionnent les différentes

technologies utilisées et plus précisément le framework `PyTorch` utilisé pour implémenter les réseaux de neurones. Puis, dans un second temps, cela nous permettra de trouver quelle serait la méthode la plus prometteuse à la complétion de zones industrielles.

5.1 PCN[2]

`PCN`[2] (**P**oint **C**ompletion **N**etwork) a été la première méthode de complétion de nuages de points basée sur les travaux de PointNet[3], de ce fait elle prend directement en entrée un nuage de points brute (sans transformation). Elle est basée sur une architecture "Encodeur / Décodeur"(8), le but étant de prendre un nuage de points en entrée et de le condenser en un `Tenseur` V regroupant les caractéristiques importantes pour, ensuite, en générer un nuage de points complet.

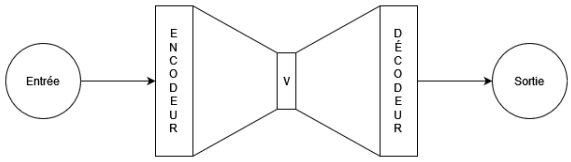

FIGURE 8 – L'architecture d'un "Encodeur / Décodeur"

Avec une telle architecture, nous pouvons apercevoir quelques défauts : Le premier est dû au manque de "skip connections" qui permettent normalement de relier les étages de l'encodeur aux étages du décodeur pour ne pas perdre d'informations. Ainsi, durant la phase d'encodage, où les dimensions de la donnée d'entrée rétrécissent au fur et a mesure des étages, nous perdons forcément des informations et donc en précision. Le deuxième défaut étant que les données d'entrée se retrouvent dans la sortie et donc ces dernières peuvent être altérées lors du processus.

5.1.1 Installation

Les auteurs de `PCN`[2] proposent une implémentation de la méthode via `Tensorflow`. Cependant, nous nous sommes mis comme contrainte d'utiliser `PyTorch` étant plus stable, ainsi que le framework utilisé par `SEGULA Technologies`.

Ainsi nous avons cherché et trouvé une version fidèle de `PCN`[2] basée sur `PyTorch`[1]. Les résultats obtenus par cette version sont équivalents aux résultats présentés dans l'article (tab.4 et 5).

1. https ://github.com/qinglew/PCN-PyTorch

5.1.2 Entraînement

Étant donné que `PCN`[2] est la première méthode de complétion utilisant un réseau de neurones profonds, les auteurs ont proposé un jeu de données synthétiques basé sur `ShapeNet`[13] : `PCN-BenchMark`.

Ce dernier est composé de 16 catégories d'objets (8 vues lors de l'entraînement, 8 non-vues). Chaque catégorie contient des nuages de points (partiels / complets) ayant une taille entre 600 et 900 points pour les partiels et précisément 16384 points pour les complets (tab.3).

Entraînement / Validation		**Test**		**Test (non-vues)**	
Catégorie	nombre de NdP [2]	Catégorie	nombre de NdP	Catégorie	nombre de NdP
Avion	3795 / 100	Avion	150	Bus	150
Armoire	1322 / 100	Armoire	150	Lit	150
Voiture	5677 / 100	Voiture	150	Bibliothèque	150
Chaise	5750 / 100	Chaise	150	Banc	150
Lampe	2068 / 100	Lampe	150	Guitare	150
Canapé	2923 / 100	Canapé	150	Moto	150
Table	5750 / 100	Table	150	Skateboard	150
Navire	1689 / 100	Navire	150	Pistolet	150

TABLE 3 – Composition de `PCN-BenchMark`[2]

Les auteurs de `PCN-Pytorch` fournissent une version de ce jeu de données utilisant des fichiers sour le format `Polygon File Format` (.ply). Ils sont utilisés pour stocker les nuages de points et peuvent être lus par différents logiciel tels que `CloudCompare` ou encore `Open3D`[15].

En ce qui concerne l'entraînement, pour pouvoir le lancer il faut fournir plusieurs paramètres :

- **taux d'apprentisage** : une constante pour gérer la taux d'apprentissage
- **Nombre d'epochs** : le nombre d'itérations sur le jeu de données d'entraînement
- **Taille du batch** : le nombre de données traitées en parallèle avant d'évaluer le modèle
- **La fonction de loss** : fonction évaluant l'apprentissage du modèle
- **Le nombre de workers** : le nombre de threads utilisés pour générer le jeu de données

Les paramètres par défaut qui ont été utilisés pour l'entraînement du modèle de l'article et de la version `PyTorch` dans l'ordre :

taux d'apprentissage	nb epochs	batch size	fonction loss	nb workers
0.0001	400	32	`CD`	8

N'ayant que 8Go de VRam, nous avons dû adapter ces paramètres pour pouvoir lancer l'entraînement sur notre machine. Ainsi, en faisant quelques tests, nous avons trouvé les paramètres suivants :

taux d'apprentissage	nb epochs	batch size	fonction loss	nb workers
0.0001	400	**16**	`CD`	**4**

Avec ceci, nous avons pu lancer l'entraînement du modèle pendant 7 jours. À la fin de cette semaine, l'entraînement aura fait 186 sur 400 epochs et a atteint un score de `CD` de 0.01091. Pendant ce dernier, nous avons pu observer l'évolution de `CD` :

Nous pouvons voir, avec cette courbe(fig.9), qu'en effet `CD` diminue au fur et à mesure des epochs, mais cela reste minime : En 120 epochs, il est passé de 0.012 à 0.011.

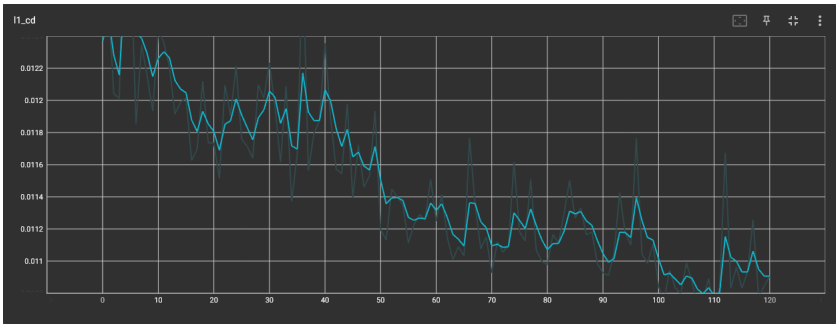

FIGURE 9 – Évolution de `CD` sur les 121 dernières epochs lors de l'entraînement de `PCN`[2]

Certes, il est important d'avoir la meilleure métrique, mais il faut aussi faire attention que cela reste rentable par rapport au temps investi. Et il faut faire attention que le modèle ne se sur-entraîne pas. C'est pourquoi l'entraînement a été arrêté après 7 jours.

5.1.3 Résultats

Maintenant que nous avons notre modèle entraîné, nous pouvons générer des résultats et les comparer par rapport aux modèles du papier et de `PCN-PyTorch`. Pour ce faire, nous pouvons calculer la moyenne des métriques (`CD` et `EMD`) obtenues lors de la phase de test. Ce qui nous permet d'obtenir les résultats suivants (tab. 4 et 5).

/	Catégories vues							
	avion	armoire	voiture	chaise	lampe	canapé	table	navire
notre modèle	6,16	11,62	9,27	12,09	13,05	12,77	9,97	11,10
`PCN-PyTorch`	6,00	11,20	9,13	12,03	12,67	12,82	9,88	10,16
`PCN`[2]	**5,50**	**10,60**	**8,69**	**10,99**	**11,33**	**11,67**	**8,59**	**9,66**
/	Catégories non-vues							
	bus	lit	bibliothèque	banc	guitare	moto	skateboard	pistolet
notre modèle	11,07	25,68	16,11	12,13	13,93	15,81	12,01	16,18
`PCN-PyTorch`	10,51	24,93	15,81	12,13	11,49	15,34	13,19	17,48
`PCN`[2]	**9,46**	**21,63**	**14,79**	**11,02**	**10,40**	**14,75**	**12,04**	**14,23**

TABLE 4 – Évaluation du modèle PCN entraîné avec la métrique `CD` ($1e-3$)

/	Catégories vues							
	avion	armoire	voiture	chaise	lampe	canapé	table	navire
notre modèle	10,78	28,49	12,85	23,64	65,46	21,51	24,89	20,03
`PCN-PyTorch`	10,59	27,15	14,36	22,49	58,77	19,28	23,76	17,97
`PCN`[2]	46,63	97,69	57,17	86,78	169,54	83,42	80,78	75,09
/	Catégories non-vues							
	bus	lit	bibliothèque	banc	guitare	moto	skateboard	pistolet
notre modèle	17,03	47,17	30,24	13,76	72,14	22,59	19,06	31,32
`PCN-PyTorch`	17,02	42,79	28,56	12,74	28,42	21,86	17,99	33,89
`PCN`[2]	62,70	135,56	103,32	61,61	168,34	92,06	84,64	107,02

TABLE 5 – Évaluation du modèle PCN avec la métrique `EMD` ($1e-3$)

Avec ces 2 tableaux, nous pouvons analyser plusieurs choses : En ce qui concerne la partie sur la métrique `CD`(tab.4), nous pouvons voir que notre modèle et celui de `PCN-PyTorch` sont extrêmement proches, la différence s'explique de par le nombre d'epochs réalisées (186 dans notre cas, contre 400 pour `PCN-PyTorch`). `PCN`[2], le modèle de l'article, reste le meilleur mais nous restons dans le même ordre de grandeur.

Cependant ce n'est pas le cas pour la partie sur la métrique `EMD`(tab.5), où nous pouvons voir que notre modèle ainsi que `PCN-PyTorch` ont des résultats totalement différents de ceux de `PCN`[2]. Cela peut s'expliquer par une erreur de l'implémentation du calcul de `EMD`, car les résultats obtenus par notre modèle et ceux de `PCN-PyTorch` sont très semblables.

Un autre point intéressant à analyser est, tout simplement, le rendu des nuages de points complétés par notre modèle entraîné (fig.10).

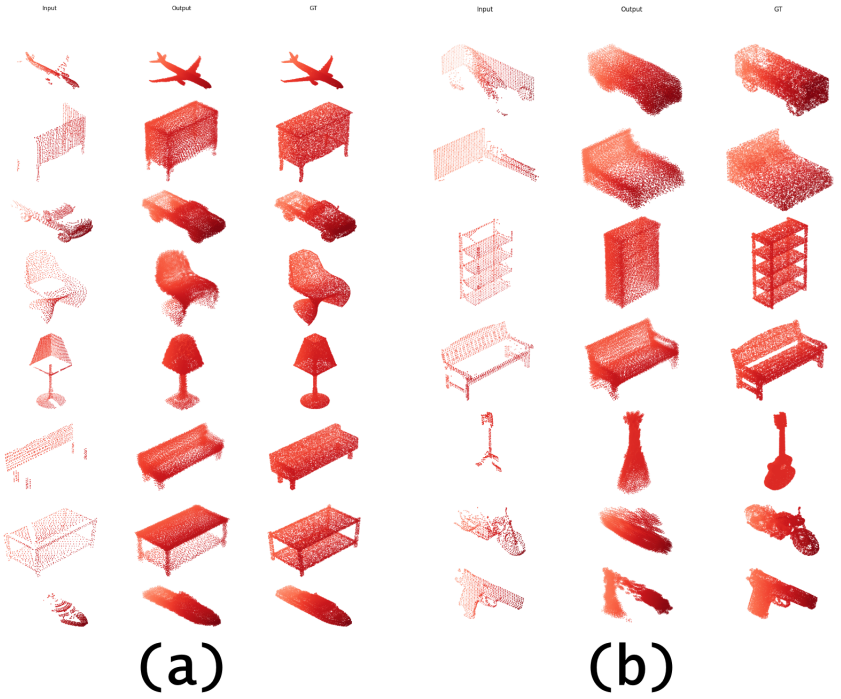

FIGURE 10 – Exemple de complétions de catégories vues(a) et non-vues(b) par `PCN`

Avec ces exemples(fig.10), nous pouvons voir que même si notre modèle ne s'est entraîné que sur 186 epochs, il nous propose des résultats très satisfaisants sur les catégories vues lors de l'entraînement. Cependant, en ce qui concerne les catégories non-vues, nous pouvons apercevoir un phénomène particulier apparaître. En effet, le modèle transforme le type de l'objet en un type vu lors de l'entraînement afin de pouvoir compléter le nuage et retrouver la forme globale de l'objet (un bus devient une voiture, un lit devient un canapé ou encore une bibliothèque devient une armoire). De plus, les catégories "guitare" et "pistolet" étant des objets assez atypiques, le modèle a vraiment du mal à les compléter et essaye tant bien que mal à refaire la forme globale de l'objet.

Si nous reportons cela à une complétion d'une zone industrielle, cela pourrait avoir un impact néfaste sur la retranscription de la scène, car la sémantique même des objets est altérée par le modèle.

5.2 PoinTr[11]

La deuxième méthode que nous avons décidée d'analyser est PoinTr[11]. Elle est aussi basée sur une architecture "Encodeur / Décodeur", cependant elle innove en utilisant des Transformeurs. Ces derniers ont été démocratisés pour le traitement du langage naturel (NLP), mais depuis peu ces derniers sont utilisés dans la Computer Vision car n'étant pas basé sur des convolutions mais sur un système d'attention, les Transformeurs permettent de ne pas perdre d'informations. Cependant nous retrouvons le même défaut que PCN[2], les données de l'entrée sont modifiées en sortie.

5.2.1 Installation

L'implémentation proposée par les auteurs de PoinTr[11] utilisant déjà PyTorch, il a été facile de trouver le github [3].

5.2.2 Entraînement

Pour l'entraînement, les auteurs de PoinTr[11] nous proposent 3 jeux de données : PCN-BenchMark, ShapeNet-34 et ShapeNet-55.

Nous avons décidé de lancer l'entraînement sur le jeu de données PCN-BenchMark, dans l'idée de pouvoir comparer les méthodes à "armes égales".
Bien entendu, nous retrouvons les même paramètres à donner pour lancer l'entraînement, dont voici les valeurs par défaut :

taux d'apprentissage	nb epochs	batch size	fonction loss
0.0005	300	**48**	CD

En adaptant ces paramètres à notre machine, nous obtenons les valeurs suivantes :

taux d'apprentissage	nb epochs	batch size	fonction loss
0.0005	300	**16**	CD

Avec ceci, nous avons pu lancer l'entraînement du modèle pendant, là aussi, 7 jours. À la fin de cette semaine, l'entraînement aura fait 146 sur 300 epochs. Cependant, lors de cet entraînement, un problème récurrent venait ralentir l'apprentissage. En effet, pendant la nuit l'entraînement se mettait en pause brutalement sans aucun message d'erreur indiqué, il fallait donc le relancer à chaque fois. De plus, nous ne pouvions rien exécuter d'autre à côté (tel que l'utilisation de tensorboard pour voir l'évolution des métriques) car cela arrêtait brusquement l'entraînement. Ainsi, nous n'avons pu voir l'évolution des métriques qu'à la fin de l'entraînement.

Ces différentes courbes(fig.11) montrent l'évolution des 4 métriques suivantes : CDL1, CDL2, EMD et F1-Score.

3. https ://github.com/yuxumin/PoinTr

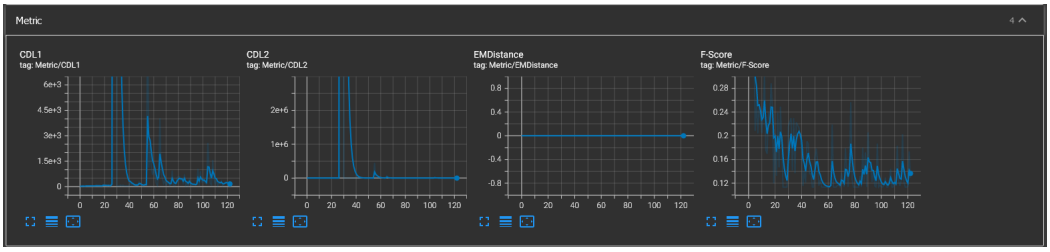

FIGURE 11 – Évolution des métriques lors de l'entraînement de `PoinTr`[11]

L'`EMD` n'évolue pas du tout, car il n'est pas calculé lors de l'entraînement étant trop gourmand en ressources. En regardant le `F1-Score`, nous pouvons voir qu'il diminue au cours du temps signifiant que le modèle s'améliore. Cependant, un point important peut être constaté sur l'évolution de `CDL1` et `CDL2`, nous pouvons apercevoir la présence de pics assez conséquents. Ces pics sont en corrélation avec le problème des pauses présenté plus haut. Et à cause de cela, il est difficile de lire l'évolution de ces métriques et cela a sûrement un impact sur la performance du modèle.

5.2.3 Résultats

Une fois l'entraînement terminé, nous pouvons générer des résultats et les comparer par rapport aux résultats présentés dans l'article. Pour ce faire, nous calculons les différentes métriques sur le jeu de données de test (tab.6).

Catégories	Notre modèle				`PoinTr`[11]			
/	F1-Score	CDL1	CDL2	EMD	F1-Score	CDL1	CDL2	EMD
Avion	**0,615**	13,807	1,314	37,882	0,915	**4,718**	**0,097**	**16,523**
Armoire	**0,253**	27,134	2,622	77,461	0,665	**10,604**	**0,499**	**28,298**
Voiture	**0,443**	15,891	0,951	44,109	0,718	**8,652**	**0,252**	**25,793**
Chaise	**0,345**	23,127	2,535	62,523	0,710	**9,191**	**0,357**	**27,241**
Lampe	**0,316**	24,944	2,801	73,227	0,798	**7,605**	**0,309**	**27,736**
Canapé	**0,325**	23,921	2,870	57,858	0,631	**11,275**	**0,653**	**30,032**
Table	**0,360**	21,339	2,197	54,356	0,796	**7,854**	**0,315**	**28,683**
Navire	**0,473**	16,307	1,180	44,464	0,797	**7,203**	**0,205**	**22,346**
Moyenne	**0,391**	20,808	2.058	56,48485	0,753	**8,387**	**0,335**	**25.831**

TABLE 6 – Évaluation du modèle `PoinTr` entraîné

En analysant les résultats, nous pouvons effectivement voir qu'il y a eu un problème lors de l'entraînement. Car, hormis avec le `F1-Score`, les résultats obtenus pour toutes les métriques sont totalement différents des résultats présentés dans l'article de `PoinTr`[11]. Pour avoir plus d'informations sur d'où pourrait venir le problème, nous pouvons analyser les résultats visuels de notre modèle :

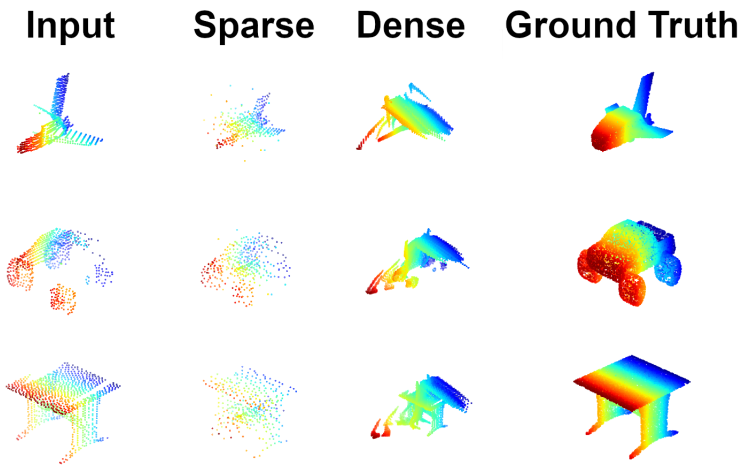

FIGURE 12 – Exemple de complétions par PoinTr

Sur la figure ci-dessus(fig.12), nous pouvons voir les différentes étapes de la complétion du nuage de points par `PoinTr`[11] : L'input, le nuage de points partiel donnée en entrée du modèle ; Le sparse, premier nuage de points générés représentant le squelette de l'objet ; Le dense, le nuage de points complété.

Lors de la génération du sparse, nous apercevons que le modèle arrive à former le squelette de l'objet cependant quelques artefacts sont générés autour. Mais le plus gros du problème vient surtout de la génération du dense. Ce dernier ne ressemble en aucun cas à l'objet en question, ce qui explique les résultats obtenus précédemment.

La transition entre le sparse et le dense se fait grâce à l'utilisation de `FoldingNet`[16]. Ce dernier, ici, est utilisé pour densifier le nuage de points sparse afin d'obtenir un nuage de points complet. L'implémentation de cette méthode est fournie par les auteurs de `PoinTr`[11]. L'erreur peut venir de cette partie, cependant, il est impossible de savoir où exactement.
Ainsi pour la suite, nous avons utilisé un modèle pré-entrainé de `PoinTr`[11], qui est le modèle du papier dont les résultats sont indiqués dans le tableau(tab.6), afin de continuer les analyses.

5.3 `LLD`[12]

La dernière méthode que nous avons analysée est `LLD`[12]. Elle est assez semblable à `PoinTr`[11], la différence se joue sur les possibilités qu'offre cette méthode. En effet, elle permet de compléter des nuages de points représentant des scènes tels que des bureaux avec quelques objets. Cela est rendu possible

grâce au jeu de données utilisé. Ce dernier est composé de nuages de points où chaque point est labellisés, c'est-à-dire, qu'il connaît le type d'objet qu'il représente.

5.3.1 Installation

Encore une fois, l'implémentation proposée par les auteurs de `LLD`[12] est basée sur `PyTorch`, donc le github [4] était facile à trouver.
Cependant, le jeu de données utilisé n'est pas fourni. Ce qui est un problème assez conséquent si nous voulons faire l'entraînement du modèle. Étant donné que `LLD`[12] utilise des nuages de points labellisés, cela aurait été sûrement très chronophage d'adapter nos propres données à cette méthode. Ainsi, cela à couper court à l'analyse de cette méthode.

5.4 Traitement des données réelles

Un point que nous n'avons pas encore abordé lors de l'analyse des différentes méthodes est : **Comment traitent-elles des données réelles ?**

Dans la littérature, la totalité des méthodes utilisent le même jeu de données pour pouvoir faire des tests sur des données réelles : `KITTI`[1] Ce jeu de données recueille plusieurs nuages de points capturés par des voitures autonomes, c'est pourquoi les scènes représentées sont des scènes d'extérieures tels que des quartiers ou des rues. Cependant, les articles ne présentent pas comment ils utilisent ce jeu de données, il ne montrent que les résultats obtenus.
Ainsi, nous sommes allés voir comment est utilisé ce jeu de données. Et nous avons découvert que le jeu de données utilisé par les différentes méthodes ne contient pas de scènes complètes mais contient des nuages de points représentant un objet spécifique (la majorité sont des voitures) :

4. https ://github.com/wangyida/disp3d

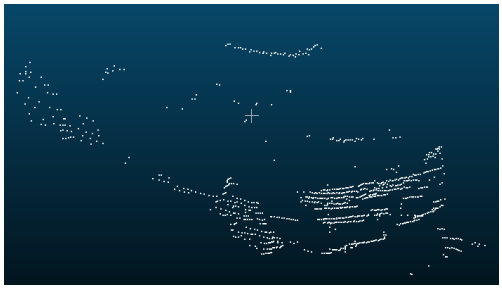

FIGURE 13 – Exemple d'un nuage de points du jeu de données `KITTI`[1]

N'ayant aucune information sur la réalisation de cette segmentation d'objets, nous avons conclu qu'il s'agit d'un pré-traitement, sûrement fait manuellement, pour récupérer les objets à compléter des scènes de `KITTI`[1].

5.5 Analyse de la courbure

Pour rappel, l'un des objectifs de cette première mission est de pouvoir sélectionner, parmi les méthodes `PCN`[2] et `PoinTr`[11], celle qui semble être la plus prometteuse à la complétion de zones industrielles. Ainsi donc, il est nécessaire de trouver un critère pertinent pour analyser les précédentes méthodes. Pour cela, nous avons regardé les zones industrielles scannées par `SEGULA Technologies` et nous avons pu constater que la plupart sont composées majoritairement de tuyaux. Étant donné que ce type d'objets ont une allure plutôt cylindrique, un critère permettant de représenter la justesse de la représentation serait la courbure du nuage de points.
En effet, analyser la courbure des nuages de points complétés par les méthodes permettrait de tester leur fidélité par rapport à la vérité.

Dans un premier temps, il faut choisir un jeu de données pertinent à utiliser. Nous sommes donc partis sur la catégorie "Lampe" de `PCN-BenchMark`, étant donné que les lampes sont souvent composées de cylindres, de sphères et autres formes ayant une forte courbure.
Pour pouvoir calculer la courbure d'un nuage de points nous utilisons `CloudCompare`, un outil très complet pour traiter les nuages de points. Ainsi, nous pouvons obtenir des résultats tels que sur l'image(fig.14) :

- la lampe (a) a une forte courbure au niveau du cylindre central.
- la lampe (b) a une forte courbure au niveau du support et de l'abat-jour, cependant le pied a une faible courbure étant donné qu'il est plat.
- la lampe (c) a une forte courbure sur toute sa structure.

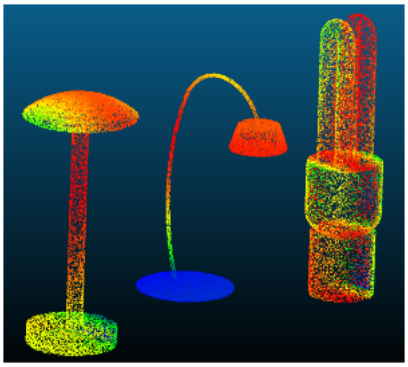

FIGURE 14 – Visualisation de la courbure de nuages de points via CloudCompare

Cependant le jeu de données étant composé de 150 nuages de points, cela prendrait du temps de calculer manuellement la courbure via `CloudCompare`. C'est pourquoi nous avons utilisé `CloudComPy`, l'`API` python de `CloudCompare` (dont un guide d'installation se trouve en annexe8.2). L'`API` est toujours en cours de développement mais elle sera suffisante pour pouvoir automatiser le calcul de la courbure.
Ainsi le principe a été de calculer la courbure moyenne des nuages de points "ground truth" et des nuages de points complétés par `PCN` et `PoinTr`. Ensuite, nous avons calculé la différence entre la courbure moyenne du "ground truth" et celle du nuage de points complété associé. Ansi, plus la valeur se rapproche de 0 plus le nuage de point complété est fidèle à la vérité. Voici un exemple qui montre que `PoinTr` a un meilleur résultat que `PCN` étant donné que la différence obtenue se rapproche plus de 0 :

ID	GT (Ground Truth)	PCN	PoinTr	GT - PCN	GT - PoinTr
2	0.095	0.015	0.049	0.080	0.046

Ainsi avec tous les autres nuages de points, nous obtenons les résultats suivants :

/	moyenne
GT - PCN	0.146
GT - PoinTr	**-0.026**

TABLE 7 – Moyenne des différences de courbures des nuages de points

Nous pouvons donc voir que `PoinTr` donne des résultats plus fidèles que `PCN`. Cependant, nous devons être sûr que cette différence entre les 2 moyennes est liée à la méthode utilisée et non au hasard engendré par le jeu

de données utilisé. Pour le savoir, nous avons fait des tests statistiques en utilisant `RStudio`[17] :
Dans un premier temps, il faut savoir si les résultats obtenus suivent une distribution normale ou non. Pour ce faire, nous utilisons le test de Shapiro-Wilk qui permet de tester l'hypothèse nulle selon laquelle un échantillon suit une distribution normale. Nous obtenons pour `GT - PCN` et `GT - PoinTr` une p-valeur inférieure à $2.2e-16$ et donc inférieure à 0.05. Ce qui veut dire que nous pouvons rejeter l'hypothèse nulle.
Maintenant que nous savons que les résultats obtenus ne suivent pas une distribution normale, nous pouvons utiliser le test de Wilcoxon-Mann-Whitney qui permet de tester l'hypothèse nulle selon laquelle il n'y a pas de différence significative entre 2 échantillons. Nous obtenons donc comme p-valeur de $1.1e-4$ et donc inférieure à 0.05. Ce qui veut dire que nous pouvons rejeter l'hypothèse nulle.

Ainsi, la différence observée (tab.7) est belle est bien significative et donc que `PoinTr` est plus fidèle que `PCN` au niveau de la courbure des nuages de points complétés.

5.6 Conclusion

Pour conclure, dans cette première partie nous avons pu analyser 3 méthodes de la littérature : `PCN`[2], `PoinTr`[11] et `LLD`[12].
Malheureusement, nous avons dû laisser l'analyse de `LLD` de côté n'ayant pas pu récupérer le jeu de données labellisé, un élément important pour l'entraînement et l'utilisation du modèle. En ce qui concerne `PCN` et `PoinTr`, nous avons entraîné des modèles et nous avons obtenu des résultats convaincants pour `PCN` mais erronés pour `PoinTr`, dû à un problème lors de l'entraînement. Pour choisir la méthode la plus adaptée à la complétion de zones industrielles, nous avons analyser la courbure des nuages de points complétés par notre modèle entraîné de `PCN` et un modèle pré-entraîné de `PoinTr`. Ainsi, les résultats obtenus nous ont permis de démontrer que `PoinTr` est la méthode la plus prometteuse.

De plus, lors de nos analyses, nous avons pu mettre en évidence 2 points non traités par les méthodes :
- La complétion de scènes réelles et entières
- La non modification des points du nuage donné en entrée

6 Conception d'un détecteur de trous :

Avec notre analyse sur les différentes méthodes de la littérature, nous avons pu mettre en évidence différents points non traités par ces dernières. En effet, aucune méthodes proposent de traiter une scène entière. Cela peut s'expliquer par la densité de points, la disparité des points ou encore l'hétérogénéité des différents nuages de points. De plus, un autre point que nous avons vu sur les méthodes analysées est que les points du nuage donné en entrée sont altérés par le réseau de neurones et peuvent être différents à la sortie. Dans le cadre du projet `SMARI`, où pour rappel le but est de pouvoir scanner des zones industrielles en nuages de points afin par la suite de générer un modèle `CAD` de la scène, il est nécessaire d'éviter au maximum d'altérer les nuages de points de crainte de perdre des informations cruciales.

Ainsi, nous nous sommes mis comme objectif de traiter ces 2 points. Pour cela, nous nous sommes inspirés des travaux de [18] et de [19] et nous proposons une méthode ayant l'architecture suivante(fig.15).

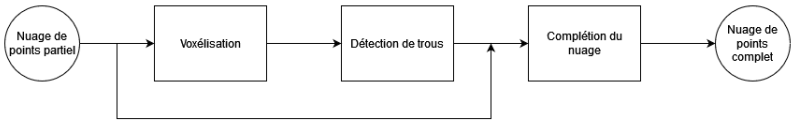

FIGURE 15 – Schéma de l'architecture globale du projet

Un point que nous pouvons relever est l'utilisation de la voxélisation. En effet, nous avons vu plus tôt dans l'état de l'art que cette dernière est gourmande en ressources et nous fait perdre en informations. Cependant, cette transformation ne sera utilisée que pour la partie "Détection de trous" car cela permet de travailler sur des zones au lieu de points. De plus, dans la littérature des méthodes de détection de trous existent mais ces dernières utilisent des informations des points liées à la caméra, tel que le vecteur direction de cette dernière. Malheureusement, les nuages de points capturés par `SEGULA Technologies` sont des concaténations de plusieurs prises de vues, ainsi nous perdons les informations liées à la caméra. C'est pourquoi la voxélisation est intéressante car elle permet de structurer et d'homogénéiser les nuages de points.

Une fois la voxélisation faite, l'idée est d'utiliser un réseau de neurones profonds de segmentation binaire. C'est-à-dire que nous allons séparer les voxels en 2 catégories : les voxels à compléter, les voxels à ne pas compléter. En faisant cela, nous allons pouvoir détecter où se trouvent les zones à

compléter mais cela permettra aussi d'ajouter une sécurité sur les points du nuage, car nous saurons quels sont les points à ne pas modifier.
Avec ces zones détectées, le but sera d'appliquer par la suite une méthode de complétion en lui donnant le nuage de points mais aussi les informations sur les zones et ainsi obtenir un nuage de points complet. Cependant, par manque de temps cette partie n'a pas été traitée lors du stage.

6.1 Génération de grille de voxels

La première étape est donc la voxélisation d'un nuage de points. Pour cela il existe plusieurs framework, tel que `Open3D`[15], qui proposent des méthodes permettant de réaliser cette transformation. Cependant, nous voulons avoir le plus de contrôle sur la manière dont sont représentées nos données. Ainsi donc, nous sommes partis sur notre propre implémentation d'un algorithme de voxélisation.

6.1.1 Lecture d'un nuage de points

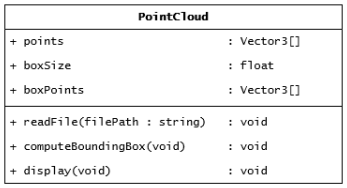

FIGURE 16 – Diagramme UML de la classe `PointCloud`

Tout d'abord, il faut pouvoir lire les fichiers représentant les nuages de points. Pour cela, nous utilisons une fonction proposée par le framework `Open3D`[15] : o3d.io.read_point_cloud(filePath, format = format)

Cette fonction permet de lire le fichier désigné par `filePath` et retourne un objet de type o3d.PointCloud qui comporte la liste des points du nuage. De plus cette fonction lit nativement les fichiers de type `Polygon File Format` (.ply) et `Point Cloud Data` (.pcd). Cependant, il existe d'autres types de fichiers représentant des nuages de points et c'est là que vient jouer la variable `format` qui permet de spécifier comment est agencé le fichier, par exemple le format "xyzrgb" représente un fichier ou chaque ligne correspond à un point ayant comme informations sa positon et sa couleur.

Ce code se trouve dans la méthode readFile(filePath) de la classe `PointCloud`(fig.16). Cette dernière permet de représenter un nuage de points via une liste des

dits-points et permet aussi de calculer la `bounding box` du nuage, grâce à la méthode computeBoundingBox(). Cette `bounding box` est particulière car elle est cubique, chaque côté fait la même longueur, cela permettra par la suite de nous faciliter la transition entre le nuage de points et la grille de voxels.

6.1.2 Transformation en grille de voxels

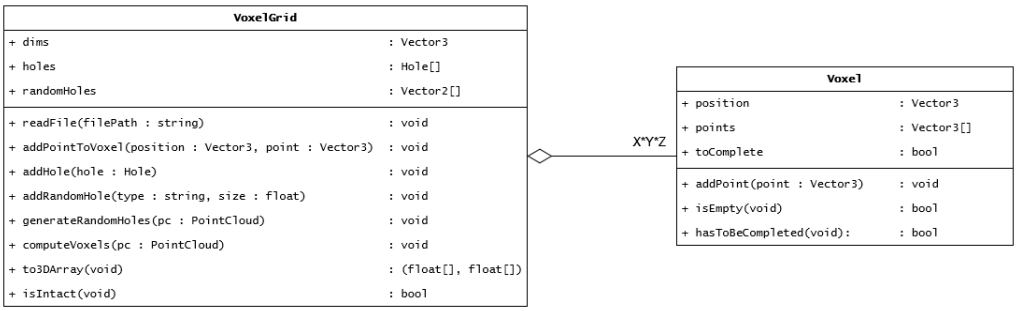

FIGURE 17 – Diagramme UML des classes `Voxel` et `VoxelGrid`

Maintenant que nous avons notre représentation d'un nuage de points par la classe `PointCloud`(fig.16), il faut à présent le transformer en une grille de voxels. Pour ce faire, nous avons créé une classe `Voxel`(fig.17) représentant un voxel avec sa `position`, les `points` associés, ainsi qu'un booléen pour savoir si ce voxel est à compléter ou pas (`toComplete`). Puis nous avons créé la classe `VoxelGrid`(fig.17) représentant une grille de voxels ayant pour dimensions $X * Y * Z$, ainsi nous pouvons avoir des grilles de taille variable. `VoxelGrid` possède une méthode readFile(filePath) permettant de lire le nuage de points associé au chemin puis elle appelle la méthode computeVoxels(pc) qui est le coeur de la classe.

Nous avons vu précédemment que nous calculons la `bounding box cubique` du nuage de points, cela nous permet de transposer les points pour que leurs coordonnées soient entre [0, taille de la `bounding box`]. Ainsi, avec un simple calcul nous pouvons passer des coordonnées du point aux coordonnées du voxel associé :

$$x_{voxel} = floor(x_{point} * \frac{dimensionXdelagrille}{tailledelaboundingbox})$$

$$y_{voxel} = floor(y_{point} * \frac{dimensionYdelagrille}{tailledelaboundingbox})$$

$$z_{voxel} = floor(z_{point} * \frac{dimensionZdelagrille}{tailledelaboundingbox})$$

Et avec ceci, nous pouvons générer la grille de voxels à partir d'un nuage de points. Pour être sûr, que la transformation s'est bien passée nous avons mis au point un `visualiseur`. Ce dernier a été implémenté grâce à l'`Ursina Engine`[5], un moteur de jeu fonctionnant sur Python. Le `visualiseur` affiche les voxels de manière plus ou moins opaques selon le nombre de points associés. Ainsi nous avons testé la voxélisation sur un nuage de points représentant le lapin de Standford :

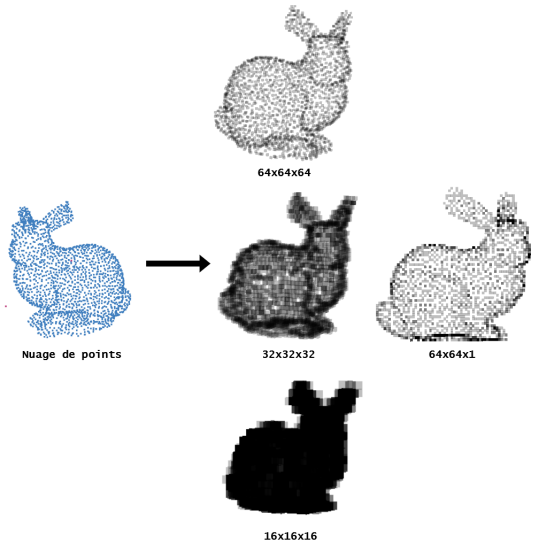

FIGURE 18 – Exemple de voxélisation avec plusieurs dimensions

En voyant cette exemple, nous pouvons nous apercevoir que le lapin n'a pas la même orientation après la transformation. Cela est tout simplement dû à `Open3D` et `Ursina Engine` qui n'ont pas le même repère. Hormis ce detail la voxélisation se déroule correctement, nous pouvons voir que plus la grille est petite plus les voxels sont opaques. En effet la taille des voxels est intriquement liée aux dimensions de la grille, donc plus la grille est petite plus les voxels sont grands.
De plus, nous avons fait un test avec une dimension de $64 * 64 * 1$ par pure curiosité. Le résultat est intéressant car nous obtenons une projection 2D du nuage de points, cela pourrait être utile pour des algorithmes particuliers. Cependant, ce n'est utilisable que si on connaît l'orientation de l'objet.

5. https ://www.ursinaengine.org/

6.1.3 Possibilité d'ajouter des trous

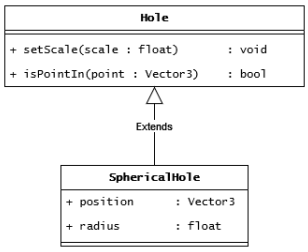

FIGURE 19 – Diagramme UML de la classe `SphericalHole`

La voxélisation est faite, maintenant il faut mettre la possibilité d'ajouter des trous afin de pouvoir commencer à générer nos jeux de données. Pour cela, nous avons créé une interface `Hole`(fig.19) composée de 2 méthodes essentielles : setScale(scale) et isPointIn(point). Pour faciliter l'utilisation de ces trous, nous avons fait en sorte que les valeurs utilisées soient comprises entre [0, 1]. Ainsi il est nécessaire d'avoir la méthode setScale(scale) pour pouvoir adapter le trou au nuage de points, le but étant de multiplier par la taille de la `bounding box`. Puis la méthode maîtresse est isPointIn(point) qui permet de savoir si le point donné en argument se trouve oui ou non dans le trou.

Ainsi, avec cette interface, nous avons la possibilité d'implémenter différents types de trous, tels qu'un trou sphérique via la classe `SphericalHole`(19).

Du côté de la classe `VoxelGrid`(fig.17), il a fallu ajouter 2 nouvelles méthodes : addHole(hole) et addRandomHole(type, size). La première permet d'ajouter un trou créer manuellement, `hole`. La deuxième est plus subtile car elle permet d'ajouter un trou de manière aléatoire dans le nuage de points, mais avec une contrainte le centre du trou doit être égale aux coordonnées d'un point du nuage pris aléatoirement. Mais pourquoi ne pas utiliser des coordonnées aléatoires entre [0, 1] ?
Comme les trous peuvent être placés à n'importe quel endroit de la `bounding box cubique` et que le nuage de points se trouve au centre de ce dernier, si nous utilisons une position complètement aléatoire il y a un risque que le trou placé ne touche aucun point du nuage. C'est pourquoi nous avons mis en place cette deuxième méthode pour être sûr à 100% que le trou placé touchera des points.

Pour appuyer nos propos sur le risque d'avoir un nuage de points intact même si nous plaçons des trous, nous avons généré des grilles de voxels, ayant

2 trous d'une taille variant entre [0, 0.5] afin de brasser sur différentes tailles de trou (petit, moyen, grand), avec le jeu de données de `PCN-BenchMark`(tab.3). Puis nous avons calculé le pourcentage de grilles restant intactes, c'est-à-dire qu'aucun point n'a été touché par un trou. Ainsi nous obtenons un taux de **17.85%**.

Dans la suite du rapport, nous allons parler de ces 2 méthodes de générations de trous avec les noms suivants :

- **Random** : les trous sont générés avec une position complétement aléatoire
- **OnPoint** : les trous sont générés avec comme position un point du nuage pris aléatoirement

Pour finir voici quelques exemples de génération de grilles de voxels avec des trous :

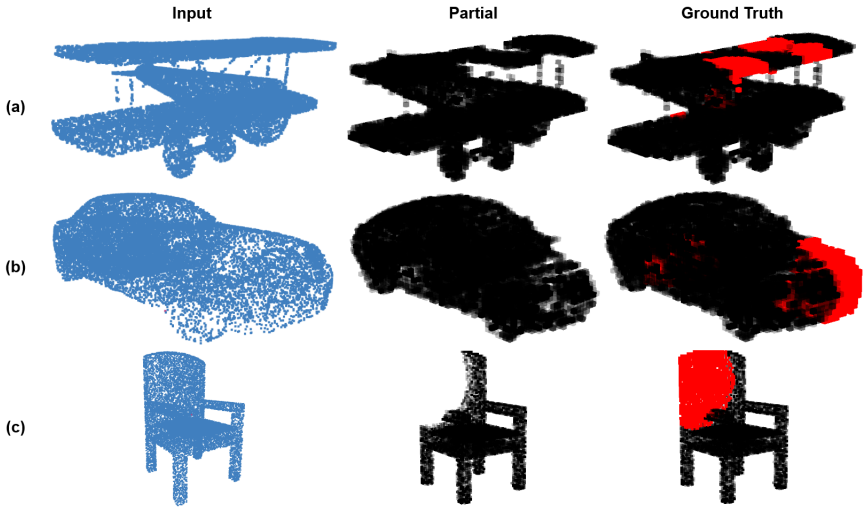

FIGURE 20 – Exemple de grilles de voxels avec des trous

Avec cette exemple(fig.20), nous pouvons voir la génération des trous fonctionne bien. Le `visualiseur` affiche le `ground truth` en colorant les voxels à compléter en rouge.

- Pour la ligne `(a)`, nous avons généré 6 trous d'une taille entre [0.01, 0.1] et nous constatons qu'aillant un assez grand nombre de trous, ces derniers se sont agglutinés faisant un plus gros trou.
- Pour la ligne `(b)`, nous avons généré 3 trous d'une taille entre [0.1, 0.25] et nous constatons qu'il y a 2 trous bien distincts, le troisième a dû fusionner avec l'un des 2 autres.

— Pour la ligne `(c)`, nous avons généré 1 trou d'une taille fixe de 0.35 et nous constatons que ce dernier mange une bonne partie de la chaise.

6.1.4 Récupération d'un tenseur représentant le nuage de points

À présent, nous avons tout ce qu'il faut pour pouvoir transformer un nuage de point complet en une grille de voxels partielle et une grille de voxels complète. Maintenant, le but de ces données est d'être utilisées derrière par un réseau de neurones il faut donc les convertir en `tenseur` utilisable par `PyTorch`. Mais qu'est-ce qu'un `tenseur` exactement ?
Un `tenseur` peut être représenté par une matrice dont les dimensions seraient : [nombre d'information, x, y, ..., n]. Pour que ce soit plus clair, voici un exemple : Nous voulons représenter une image de $128 * 128$ pixels. Un pixel étant composé de 3 couleurs (rgb), il y a donc 3 informations. Ainsi le `tenseur` associé serait de dimension suivante : [3, 128, 128].

Maintenant que nous savons ce qu'est un `tenseur`, comment représenter les grilles de voxels partielles et complètes ? Pour ce qu'il s'agit des dimensions, nous avons choisi d'avoir des grilles de $64 * 64 * 64$. Pour ce qu'il s'agit du nombre d'informations, ici nos pixels de l'exemple sont les voxels. Les voxels contiennent 2 informations : le nombre de points associés et si le voxel est à compléter.

Pour la grille de voxels partielle, ce qui est important c'est le nombre de voxels. Et pour la grille de voxels complète, ce qui est important c'est de savoir si le voxel est à compléter ou non. Donc leur `tenseur` associé serait :

— X (partiel) = [1, 64, 64, 64]
— y (complet) = [1, 64, 64, 64]

L'implémentation de cette conversion se trouve dans la classe `VoxelGrid`(fig.17) avec la méthode to3DArray() qui retourne non pas des `tenseurs` directement mais des `numpy array` (qui peuvent être transformer en `tenseur` très facilement).

6.1.5 Optimisations

Un point intéressant à aborder est l'optimisation de la voxélisation. En effet, lors des tests de l'algorithme, nous avons pu constater que la transformation était assez lente (tab.8, partie `ASCII`). Cela peut s'expliquer du fait que les fichiers du jeu de données `PCN-BenchMark` soient sauvegardés en `ASCII`. Du coup, une amélioration envisageable a été de passer ces fichiers en `Binaire`. Ainsi donc, nous obtenons les résultats suivants :

Type de fichier	Avion	Armoire	Voiture	Chaise	Lampe	Canapé	Table	Navire	Total
`ASCII`	0.85	0.86	0.85	0.88	0.85	0.88	0.94	0.84	0.87
`Binaire`	**0.79**	**0.76**	**0.72**	**0.77**	**0.80**	**0.78**	**0.79**	**0.79**	**0.77**

TABLE 8 – Comparaison du temps d'exécution moyen (en sec) de la voxélisation entre des fichiers `ASCII` et `Binaire`

Nous pouvons voir qu'il y a eu une légère amélioration, mais cela reste dans le même ordre de grandeur.

Une autre solution envisagée a été de sauvegarder en amont les grilles de voxels, cela permettra de travailler avec le jeu de données sans se soucier du temps d'exécution. Cependant, nous perdons la force de pouvoir proposer des données variées facilement, or ce point est très important lors d'un entraînement d'un réseau de neurones profonds. Du coup, nous avons écarté cette solution. Et nous avons préféré perdre en temps de calcul mais garder cette notion de jeu de données varié.
D'autres solutions auraient pu être réalisées mais cela aurait demandé de revoir toute la structure du programme : parallélisation, utilisation d'un autre framework pour lire les nuages de points, ou encore utiliser une structure d'arbre pour optimiser la génération des voxels.

6.2 Implémentation d'un réseau de neurones pour détecter les trous

Maintenant que la première partie de l'architecture(fig.15) est faite, il faut maintenant passer à la détection de trous. Ainsi, le but de cette partie est de pouvoir séparer les voxels de la grille en 2 catégories : les voxels à compléter et les voxels à ne pas compléter. Pour ce faire, nous avons décidé naturellement d'utiliser un réseau de neurones profonds de segmentation binaire. Ce dernier aura pour fonctionnalité de prendre en entrée une grille de voxels où chaque voxel a comme information le nombre de points associés, puis de retourner en sortie une grille de voxels où chaque voxel aura un état (0 ou 1) représentant dans quel catégorie il se trouve :

— 0 : le voxel n'est pas à compléter
— 1 : le voxel est à compléter

6.2.1 Reprise de l'architecture `U-Net`[20]

Pour implémenter notre réseau de neurones profonds, nous avons repris l'architecture de `U-Net`[20]. `U-Net` est un réseau très connu dans le domaine de

la `Computer Vision`, car il a été le premier réseau de segmentation à mettre en place une architecture "Encodeur / Décodeur". Une architecture qui est grandement utilisée maintenant par les différents réseaux de segmentation modernes. De plus `U-Net` est facilement modulable pour s'adapter à différents types d'usage, c'est pourquoi notre choix s'est porté sur lui.

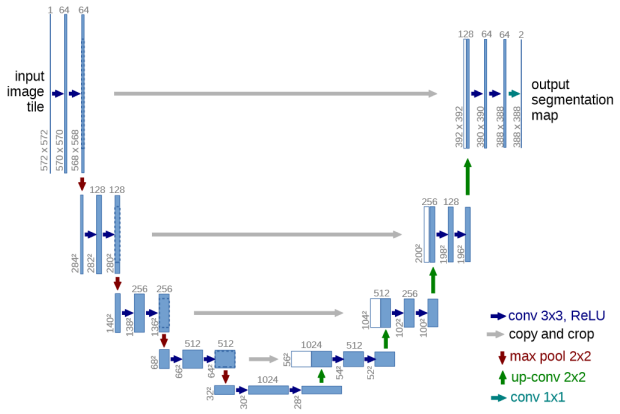

FIGURE 21 – Schéma de l'architecture `U-Net`[20]

Voici un schéma représentant l'architecture de `U-Net`(21), nous pouvons voir que cette dernière a une forme de "U" et qu'elle est composée de 3 parties distinctes :
La première étant l'encodeur, la partie descendante, qui prend en entrée une image d'une taille fixe et réduit les dimensions de cette dernière au fur et à mesure des 4 étages en utilisant des couches de convolution, de pooling ainsi que des ReLU. Cela permet de récupérer les caractéristiques de l'image sur différentes échelles.
La deuxième partie est appelée le goulot d'étranglement, elle permet de faire la transition entre l'encodeur et le décodeur en utilisant des couches de convolution.
Et la dernière partie étant le décodeur, la partie montante, qui prend en entrée la représentation latente donnée par la partie précédente et effectue le travail inverse de l'encodeur. Ainsi, au fur et à mesure des 4 étages, les dimensions de l'image augmentent jusqu'à revenir à sa taille originale tout en combinant les caractéristiques extraites à différentes échelles afin d'obtenir à la fin une segmentation détaillée de l'image. Le décodeur se termine avec une couche de convolution pour adapter la sortie au nombre de catégories à trouver.

Un point important de l'architecture de `U-Net` est l'utilisation de "skip

connections" reliant les différents étages de l'encodeur et du décodeur ensemble. Cela permet de fusionner les informations obtenues aux différentes échelles. Ainsi, contrairement à `PCN`[2], `U-Net` ne perd pas d'informations et aide à avoir une segmentation plus détaillée.

Cependant `U-Net` traite des images 2D, or nous voulons traiter des grilles de voxels 3D. De ce fait, il a fallu adapter l'architecture pour que le réseau puisse prendre en charge des données 3D. Le réseau est composé principalement de 2 couches distinctes de neurones : la convolution et le MaxPooling. `PyTorch` propose une version 2D, ainsi qu'une version 3D de ces couches, ce qui rend le passage de la 2D à la 3D trivial.
Ainsi donc, nous avons implémenté une version 3D de `U-Net` prenant en entrée une matrice de taille $64*64*64$ dont chaque cellule correspond au nombre de points associés au voxel et retourne en sortie une matrice de même taille dont chaque cellule a soit la valeur 0 pour signifier que le voxel n'est pas à compléter, soit la valeur 1 pour signifier que le voxel est à compléter. De plus, nous avons fait en sorte que le réseau puisse prendre en entrée d'autres informations que le nombre de points associés au voxel, ce qui permettra de faire évoluer ce dernier selon nos futures besoins.

Un point crucial de l'implémentation d'un réseau de neurones profonds est la fonction de loss. Étant donné que notre réseau est un réseau de segmentation binaire, nous avons utilisé la **B**inary **C**ross-**E**ntropy Loss, la fonction de loss couramment utilisée pour les classifications binaires.

6.2.2 Création des jeux de données

Nous avons notre modèle, la prochaine étape est de lui fournir une multitude de données afin qu'il puisse s'entraîner. Pour ce faire, nous nous sommes basés sur le jeu de données `PCN-BenchMark`[2] étant donné qu'il est utilisé par la majorité des méthodes de complétion. Pour rappel, les spécifications du jeu de données ont été données dans la figure (fig.3).
Le but étant de prendre les nuages de points "ground truth" est d'utiliser notre générateur de grille de voxels sur ces derniers en ajoutant plus ou moins de trous de taille variable afin d'avoir notre jeu de données personnalisé.

Entraînement et Validation

Pour les phases d'entraînement et de validation, nous faisons en sorte que la génération de grilles de voxels soit dynamique. C'est à dire que nous générons les grilles de voxels à chaque epoch. Cela a pour effet de proposer des

données différentes à chaque fois, étant donné que les trous ajoutés sont positionnés aléatoirement. Cependant, comme nous l'avons vu précédemment, la génération de grilles de voxels est un procédé assez lent ce qui aura un impact considérable sur la durée de l'entraînement.
Ainsi pour jouer sur la génération du jeu de données, nous avons défini plusieurs paramètres à donner lors du lancement de l'entraînement :

- **method** : La méthode de génération de trous (Random ou OnPoint)
- **nbTrous** : Le nombre de trous à générer
- **minRadius** : le rayon minimum des trous
- **maxRadius** : le rayon maximum des trous
- **intact** : le pourcentage de nuages de points intacts

Le dernier paramètre est très intéressant car il permet au modèle d'avoir des exemples où il n'y a rien à compléter, aucun voxel à catégoriser.

Test

Pour la phase de test, il est nécessaire d'avoir un jeu de données fixe. C'est pourquoi, nous avons généré le jeu de données en amont. Nous en avons généré 3 pour représenter différentes situations :

- **smallholes** : chaque grille de voxels contient **6** trous dont le rayon est entre [**0.01, 0.1**]
- **mediumholes** : chaque grille de voxels contient **2** trous dont le rayon est entre [**0.1, 0.25**]
- **largehole** : chaque grille de voxels contient **1** trou dont le rayon vaut **0.35**

Pour illustrer ces jeux de données, nous pouvons reprendre la figure (fig.20).

6.2.3 Entraînements

Maintenant, nous avons tout ce qu'il nous faut pour lancer différents entraînements du modèle. Comme il y a plusieurs paramètres jouant sur les jeux de données d'entraînement et de validation, il est nécessaire de faire plusieurs entraînements dans différentes configurations pour voir quels seraient les meilleurs paramètres.
Il faut donc pouvoir comparer les modèles entraînés entre eux. Pour ce faire, nous calculons l'`accuracy` de ces derniers. Dans notre cas, l'`accuracy` représente le nombre de bonnes prédictions. Par exemple, une `accuracy` de 100%

signifie que le modèle a réussi à prédire exactement les bons voxels à compléter.

Un point notoire à signalé est la durée d'un entraînement, en effet une epoch met 4h pour se réaliser. Cette longue durée s'explique majoritairement par la génération dynamique des grilles de voxels qui, comme nous avons pu le voir précédemment, est assez lente. Ainsi, il fallait prendre en compte ce point lors du choix du nombre d'epochs à effectuer lors d'un entraînement. Nous avons donc choisi arbitrairement de réaliser 3 epochs par entraînement, ce qui est équivaut à 12h d'entraînement. Comme nous pouvons le voir avec la figure (fig.22), avec 3 epochs le modèle arrive à une `accuracy` d'environ 50% et une `loss` d'environ 0.003, ce qui sera suffisant pour pouvoir comparer les modèles entraînés.

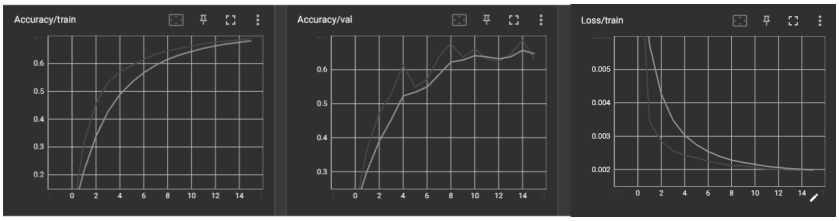

FIGURE 22 – Évolution de l'`accuracy` et du `loss` pendant l'entraînement final de 15 epochs

Au total, nous avons effectué 16 entraînements afin de trouver les meilleurs paramètres suivants :

Paramètres	method	nbTrous	minRadius	maxRadius	intact
Valeur	OnPoint	3	0.01	0.25	10%

Ainsi, nous avons entraîné notre modèle final sur 15 epochs avec ces paramètres et nous avons obtenu les résultats suivants sur les jeux de données de test :

Catégories	smallholes	mediumholes	largehole
Avion	62.91%	60.49%	35.60%
Armoire	52.07%	58.91%	24.89%
Voiture	39.22%	48.72%	19.51%
Chaise	52.24%	58.28%	22.94%
Lampe	49.21%	36.85%	11.29%
Canapé	62.93%	66.95%	35.77%
Table	63.12%	72.47%	33.06%
Bateau	53.47%	50.61%	27.06%
Total	**54.40%**	**56.66%**	**26.26%**

TABLE 9 – Résultats obtenus par notre modèle final sur les jeux de données de test

Ces résultats (tab.9) nous permettent de voir que notre modèle est effi-

cace pour détecter des petits trous ainsi que des trous de tailles moyennes. Cependant, il a du mal à détecter un trou large.

6.2.4 Résultats

Pour terminer, voici différents résultats obtenus avec notre modèle par inférences itératives. C'est à dire que nous appliquons plusieurs fois l'inférence du modèle sur l'entrée afin de détecter le plus de voxels à compléter :

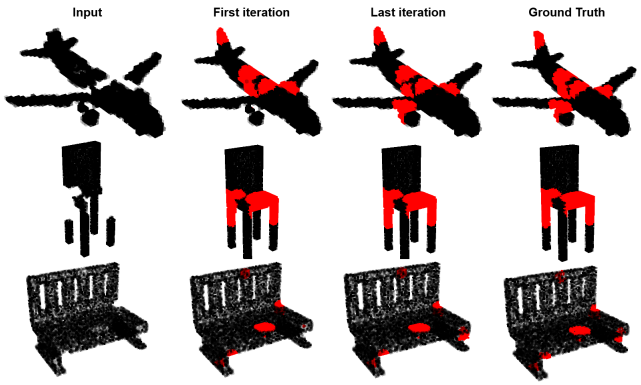

FIGURE 23 – Exemple de détections de trous sur des objets du jeu de données de test

Sur cette figure (fig.23), nous pouvons voir la détection de trous sur des objets faisant partis de catégories vues pendant l'entraînement. Le résultat obtenu est très convaincant, les prédictions obtenues sont quasi-identiques au "ground truth". Nous pouvons même voir que le modèle arrive à comprendre la forme des objets car il reforme le bout de la queue de l'avion ou encore le coin de la chaise.

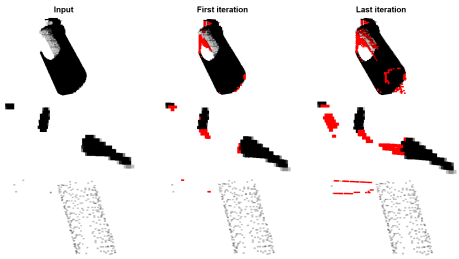

FIGURE 24 – Exemple de détections de trous sur des tuyaux réels

Sur cette figure (fig.24), nous pouvons voir la détection de trous sur des objets réels et plus particulièrement des tuyaux. Le résultat est loin d'être

satisfaisant, cependant nous pouvons voir sur les 2 premiers tuyaux que le modèle est sur la bonne voie. Mais il a du mal à détecter la forme globale des trous. Puis nous avons le troisième tuyau qui montre un cas complètement erroné, car il détecte des trous hors de hors de l'objet.

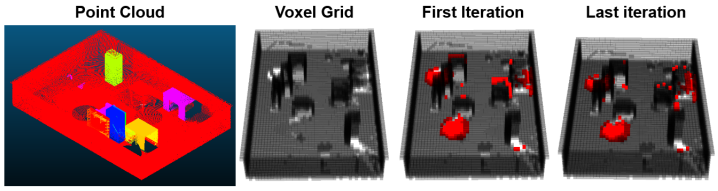

FIGURE 25 – Exemple d'une détection de trous sur une scène

Sur cette figure (fig.25), nous avons testé notre modèle sur une scène entière. Cette scène est synthétique et a été générée par Erell COTTOUR lors de son projet. Nous pouvons voir que même si notre modèle s'est entraîné que sur des objets, il arrive à donner un résultat convenable. Il détecte par exemple les trous se situant au sol et il essaye même de compléter un objet de la scène, la table. Nous pensons que le modèle considère la pièce comme un seul grand objet, une sorte de boîte, et essaye de détecter les trous se situant sur ses côtés.

7 Conclusion :

Pour rappel, le stage avait pour objectif initial de proposer une méthode permettant de compléter des nuages de points représentant des zones industrielles.

Pour ce faire il a fallu, dans un premier temps, analyser 3 méthodes présentes dans la littérature : `PCN`[2], `PoinTr`[11] et `LLD`[12]. Cette analyse nous a permis de savoir laquelle de ces méthodes serait la plus pertinente à utiliser sur des données industrielles. Pour cela, nous avons comparé la courbure des nuages de points générés afin de voir quelle méthode sérait le plus fidèle à la véritable courbure du nuage. Ainsi, `PoinTr`[11] s'est avérée être la meilleure méthode.
De plus, cette analyse nous a permis de mettre en évidence 2 points non traités par ces méthodes :

— La complétion de scène entière
— La non-modification des points du nuage d'entrée

Ces 2 points sont cruciaux pour notre projet, le but étant de traiter des nuages de points de zones industrielles afin de proposer un modèle `CAD` fidèle

à la réalité. La modification des points existants pourrait avoir un impact sur la précision des futurs traitements du nuage.
Ainsi, dans un second temps, nous avons proposé une nouvelle méthode de complétion séparée en 2 parties : une détection de trous, puis une complétion des trous détectés. Malheureusement, nous n'avons pu réaliser que la première partie.

La détection de trous se fait via un réseau de segmentation binaire qui est une adaptation de `U-Net`[20] afin que ce dernier traite, non pas des images 2D, mais des grilles de voxels 3D représentant le nuage de points. De ce fait, nous avons créer notre propre méthode de voxélisation de nuage de points afin d'avoir le contrôle sur tous les paramètres et, ainsi, proposer une multitude de jeux de données à partir d'un seul ensemble de nuages de points. Ainsi, nous avons obtenu un modèle de détection de trous dont les résultats sont plutôt convaincants en ce qui concerne des objets de catégories vues pendant l'entraînement, ainsi que des scène entières synthétiques. Cependant pour des objets réels, tels que des tuyaux, le modèle a du mal mais il est sur une bonne voie.
En améliorations futures, la plus importante est de modifier une méthode de complétion, tel que `PoinTr`[11], afin que cette dernière prenne en compte en compte les résultats de notre modèle de détection de trous. En ayant cette donnée supplémentaire, cela permettra d'éviter de modifier les points déjà présents.
Un autre point d'amélioration serait de remanier la partie voxélisation afin de diminuer son temps d'exécution qui a eu un impact conséquent sur l'entraînement du modèle. Pour ce faire, nous pourrions utiliser un autre framework qu'`Open3D`[15] pour lire les nuages de points ou encore mettre en place une structure d'arbre pour représenter la grille de voxels.

Références

[1] Andreas Geiger, Philip Lenz et Raquel Urtasun. “Are We Ready for Autonomous Driving ? The KITTI Vision Benchmark Suite”. In : *2012 IEEE Conference on Computer Vision and Pattern Recognition*. Juin 2012, p. 3354-3361. doi : 10.1109/CVPR.2012.6248074.

[2] Wentao Yuan et al. “PCN : Point Completion Network”. In : *2018 IEEE 3DV*. arXiv, sept. 2019. doi : 10.48550/arXiv.1808.00671. arXiv : 1808.00671 [cs]. (Visité le 06/03/2023).

[3] Charles R. Qi et al. “PointNet : Deep Learning on Point Sets for 3D Classification and Segmentation”. In : *2017 IEEE Conference on Computer Vision and Pattern Recognition*. arXiv, avr. 2017. doi : 10.48550/arXiv.1612.00593. arXiv : 1612.00593 [cs]. (Visité le 09/02/2023).

[4] David Doria et Richard J. Radke. “Filling Large Holes in LiDAR Data by Inpainting Depth Gradients”. In : *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. Juin 2012, p. 65-72. doi : 10.1109/CVPRW.2012.6238916.

[5] Guilin Liu et al. “Image Inpainting for Irregular Holes Using Partial Convolutions”. In : *2018 ECCV*. arXiv, déc. 2018. doi : 10.48550/arXiv.1804.07723. arXiv : 1804.07723 [cs].

[6] Zeqing Fu, Wei Hu et Zongming Guo. “Local Frequency Interpretation and Non-Local Self-Similarity on Graph for Point Cloud Inpainting”. In : *2018 IEEE Conference on Computer Vision and Pattern Recognition*. arXiv, sept. 2018. doi : 10.48550/arXiv.1810.03973. arXiv : 1810.03973 [cs].

[7] Ramesh Ashok Tabib et al. “Learning-Based Hole Detection in 3D Point Cloud Towards Hole Filling”. In : *Procedia Computer Science*. Third International Conference on Computing and Network Communications (CoCoNet’19) 171 (jan. 2020), p. 475-482. issn : 1877-0509. doi : 10.1016/j.procs.2020.04.050.

[8] Minghua Liu et al. “Morphing and Sampling Network for Dense Point Cloud Completion”. In : *2020 AAAI Conference on Computer Vision and Pattern Recognition*. arXiv, nov. 2019. doi : 10.48550/arXiv.1912.00280. arXiv : 1912.00280 [cs].

[9] Zitian HUANG et al. "PF-Net : Point Fractal Network for 3D Point Cloud Completion". In : *2020 IEEE Conference on Computer Vision and Pattern Recognition*. arXiv, mars 2020. DOI : 10.48550/arXiv.2003.00410. arXiv : 2003.00410 [cs].

[10] Liang PAN et al. "Variational Relational Point Completion Network". In : *2021 IEEE Conference on Computer Vision and Pattern Recognition*. arXiv, avr. 2021. DOI : 10.48550/arXiv.2104.10154. arXiv : 2104.10154 [cs].

[11] Xumin YU et al. "PoinTr : Diverse Point Cloud Completion with Geometry-Aware Transformers". In : *ICCV 2021*. arXiv, août 2021. DOI : 10.48550/arXiv.2108.08839. arXiv : 2108.08839 [cs]. (Visité le 04/04/2023).

[12] Yida WANG et al. "Learning Local Displacements for Point Cloud Completion". In : *2022 IEEE CVPR*. arXiv, mars 2022. DOI : 10.48550/arXiv.2203.16600. arXiv : 2203.16600 [cs]. (Visité le 06/03/2023).

[13] Angel X. CHANG et al. *ShapeNet : An Information-Rich 3D Model Repository*. Déc. 2015. DOI : 10.48550/arXiv.1512.03012. arXiv : arXiv:1512.03012. (Visité le 16/03/2023).

[14] Angela DAI et al. "ScanNet : Richly-annotated 3D Reconstructions of Indoor Scenes". In : *2017 IEEE Conference on Computer Vision and Pattern Recognition*. arXiv, avr. 2017. DOI : 10.48550/arXiv.1702.04405. arXiv : 1702.04405 [cs].

[15] Qian-Yi ZHOU, Jaesik PARK et Vladlen KOLTUN. *Open3D : A Modern Library for 3D Data Processing*. Jan. 2018. DOI : 10.48550/arXiv.1801.09847. arXiv : arXiv:1801.09847. (Visité le 29/03/2023).

[16] Yaoqing YANG et al. "FoldingNet : Point Cloud Auto-encoder via Deep Grid Deformation". In : *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. arXiv, avr. 2018. DOI : 10.48550/arXiv.1712.07262. arXiv : 1712.07262 [cs]. (Visité le 02/08/2023).

[17] R CORE TEAM. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2023. URL : https://www.R-project.org/.

[18] Ramesh Ashok TABIB et al. "Learning-Based Hole Detection in 3D Point Cloud Towards Hole Filling". In : *Procedia Computer Science*. Third International Conference on Computing and Network Communications (CoCoNet'19) 171 (jan. 2020), p. 475-482. ISSN : 1877-0509. DOI : 10.1016/j.procs.2020.04.050. (Visité le 15/05/2023).

[19] Lyne P. Tchapmi et al. "SEGCloud : Semantic Segmentation of 3D Point Clouds". In : *3DV 2017*. arXiv, oct. 2017. doi : 10.48550/arXiv.1710.07563. arXiv : 1710.07563 [cs]. (Visité le 22/05/2023).

[20] Olaf Ronneberger, Philipp Fischer et Thomas Brox. "U-Net : Convolutional Networks for Biomedical Image Segmentation". In : arXiv, mai 2015. doi : 10.48550/arXiv.1505.04597. arXiv : 1505.04597 [cs]. (Visité le 23/06/2023).

8 Annexes :

8.1 Guide d'installation de l'environnement de travail

Installation des derniers drivers Nvidia :

Avant de procéder à l'installation de l'environnement de travail, il faut s'assurer que les drivers Nvidia soient à jour. Pour cela, nous pouvons aller chercher ces derniers sur ce lien. Et voici nos configurations :

Ordinateur
Geforce
Geforce 10 Series
Geforce GTX 1080
Windows 10 64-bit
Game Ready Driver (GRD)
Français

Installation / Mise à jour de WSL :

Nous travaillons via le **W**indows **S**ubsystem **L**inux (**WSL**). Et plus précisément la version **2**, car Nvidia propose une version de Cuda fonctionnelle seulement sur **WSL2**.

L'installation se fait en ouvrant un terminal et en exécutant la commande :

```
wsl --update
```

Installation d'Ubuntu 20.04 :

Une fois **WSL2** installé il faut installer une distribution linux. Pour savoir quelles sont les distributions disponibles pour **WSL**, nous pouvons exécuter la commande suivant :

```
wsl -l -o
```

Nous avons décidé de partir sur Ubuntu-20.04, car en regardant les versions d'Ubuntu prises en compte pour Cuda 11.3 s'arrêtent à 20.04. Ainsi pour l'installer il faut exécuter la commande suivante :

```
wsl --install -d Ubuntu-20.04
```

Pour finaliser l'installation d'Ubuntu-20.04, voici des commandes pour mettre tout à jour (à faire dans **WSL**) :

```
sudo apt update
sudo apt upgrade -y
```

Installation de Cuda 11.3 :

Avant de suivre l'installation de Cuda, il faut supprimer l'ancienne clé GPG :

```
sudo apt-key del 7fa2af80
```

Ensuite il suffit de suivre les explication suivantes pour installer Cuda Toolkits 11.3 :

```
wget https://developer.download.nvidia.com/compute/cuda
    /repos/wsl-ubuntu/x86_64/cuda-wsl-ubuntu.pin
sudo mv cuda-wsl-ubuntu.pin /etc/apt/preferences.d/cuda
    -repository-pin-600
wget https://developer.download.nvidia.com/compute/cuda
    /11.3.1/local_installers/cuda-repo-wsl-ubuntu-11-3-
    local_11.3.1-1_amd64.deb
sudo dpkg -i cuda-repo-wsl-ubuntu-11-3-local_11.3.1-1
    _amd64.deb
sudo apt-key add /var/cuda-repo-wsl-ubuntu-11-3-local/7
    fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

Installation d'Anaconda :

Afin de créer des environnements de travail python facilement, nous avons opté pour `Anaconda`. Pour ce faire, il faut télécharger le `.deb` (ici) et l'exécuter dans **WSL** :

```
bash [nom_du_fichier].deb
```

Création d'un environnement de travail :

Afin de créer l'environnement de travail, on utilise Conda. Il est créé par le biais de la ligne ci-dessous, et il est possible de préciser la version de Python que l'on souhaite qui soit exécutée dessus en la précisant à la fin de la commande (si ce n'est pas fait, il prend la version de Python la plus récente pour Anaconda).

```
conda create -name [env] python=3.X
```

Une fois l'environnement créé, il faut l'activer en exécutant la ligne de commande suivante :

```
conda activate [env]
```

Puis on installe Pytorch sur l'environnement en précisant les bonnes versions pour les installations des différents modules Python. Avec la version de Cuda 11.3, nous avons opté pour la version de Pytorch 1.12.1. Puis nous sommes allés sur ce site pour trouver la bonne ligne à exécuter pour l'installation :

```
conda install pytorch==1.12.1 torchvision==0.13.1
    torchaudio==0.12.1 cudatoolkit=11.3 -c pytorch
```

8.2 Guide d'installation de CloudComPy (API python de CloudCompare)

Le guide officiel peut se trouver ici.

Pré-requis :

- Ubuntu == 20.04 / Debian == 11
- GLIBC >= 2.29*
- Anaconda == 3 / miniconda == 3
- Python == 3.10

** : Pour savoir la version de GLIBS, il faut utiliser la commande suivant :*

```
ldd --version
```

Télécharger les binaires de CloudComPy :

Les binaires se trouvent sur ce site. Il faut ensuite les dezipper avec la commande suivante :

```
tar -xf CloudComPy_Conda310_Linux64_-date-.tgz
```

Créer un environnement conda :

```
conda create --name cloudcompy-dev python=3.10

conda activate cloudcompy-dev

conda config --add channels conda-forge
conda config --set channel_priority strict

conda install "boost=1.74" "cgal=5.4" cmake draco
   ffmpeg "gdal=3.5" jupyterlab laszip "matplotlib=3.5"
    "mysql=8.0" "numpy=1.22" "opencv=4.5" "openmp=8.0"
   "pcl=1.12" "pdal=2.4" "psutil=5.9" pybind11 "qhull
   =2020.2" "qt=5.15.4" "scipy=1.8" sphinx_rtd_theme
   spyder tbb tbb-devel "xerces-c=3.2"
```

Utiliser CloudComPy :

Pour utiliser CloudComPy, il faut lancer un script fourni avec les binaires pour activer l'environnement conda :

```
. <path_CloudComPy>/bin/condaCloud.sh activate
    cloudcompy-dev
```

Il suffit juste ensuite d'exécuter du code python ayant comme import :

```
import cloudComPy as cc
```

Optionnel :

Il est possible qu'il faut installer libomp.so.5 :

```
sudo apt-get install libomp5
```

Et il est intéressant d'installer le module pandas, pour pouvoir traiter / analyser des données :

```
pip install pandas
```

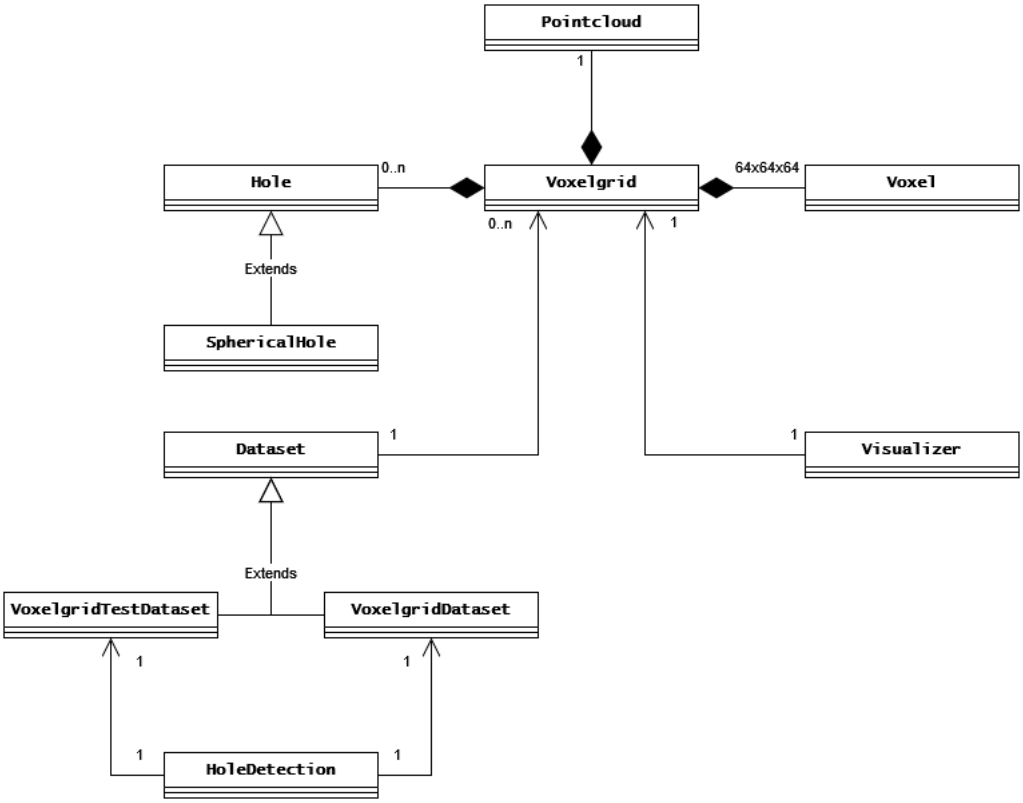

FIGURE 26 – Diagramme UML de l'architecture du projet