Mémoire

Auteur : Borremans, Cadwal
Promoteur(s) : Jonard, François; 25231
Faculté : Faculté des Sciences
Diplôme : Master en sciences géographiques, orientation géomatique, à finalité spécialisée en geodata-expert
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/21531

LIÈGE université
Sciences

Faculty of Sciences

Department of Geography

Tree inventory identification from UAV-mounted LiDAR point cloud using deep learning: application of VoteNet from TorchPoints3D for 3D object detection purposes

Memory presented by : **Cadwal BORREMANS**

For the fulfillment of the degree of

Master in Geographic Sciences,

Geomatics Orientation,

Geo-Data Finality

Academic year:	**2023-2024**
Defense date:	**Septembre 2024**
Committee chair:	**Prof. René WARNANT**
Supervisor:	**Prof. François JONARD**
Co-Supervisor:	**Lect. Romain NEUVILLE**
Reading Committee:	**Dr. Romy SCHLÖGEL**
	Dr. Jean-Paul KASPRZYK

I would like to extend my sincere thanks to the following people who supported me throughout this thesis.

Firstly, I am very grateful to my supervisors, Prof. François Jonard and Lect. Romain Neuville, for their guidance and support. Their expertise and advice were invaluable, and I truly appreciated their suggestion of this thesis topic, which I found particularly interesting

I also want to thank my family for their constant support throughout my studies. Their encouragement has meant a lot to me.

A special thanks to Cassandra Hardy for being there for me every day and for her unwavering support during this past years.

Finally, I want to thank my fellow Geography students for these past five years.

Thank you all for your support!

Abstract

An accurate and efficient tree inventory is a major first step in optimal forest management. Many studies have focused on this topic. One efficient method to acquire data about forest horizontal and vertical structure in a time saving and scalable way is to use light detection and ranging (LiDAR) sensors mounted on UAVs. Various methods based on these data have been explored in the literature. Traditional methods rely either on rule-based algorithms or on machine learning techniques. However, the drawbacks of these methods are their dependency on the studied area, the need to adjust parameters based on the type of forest and their difficulties with heterogeneous forests. A recent solution is the development of deep learning object detection models. These methods are increasingly studied in the literature across various fields of research. Their advantage is that, given a sufficiently large and varied training dataset, an object detection model can learn to accurately detect objects in various contexts. One of their major drawbacks is their time-consuming training process, which requires wide and varied datasets along with high computational resources. This field has been extensively explored for 2D object detection and segmentation. Pretrained models are now available that can detect various classes and be applied to different tasks without needing fine-tuning or only requiring a short training, such as the You Only Look Once (YOLO) model. However, to the best of our knowledge, no studies have applied 3D object detection model directly to 3D aerial point clouds for forest tree detection.

Therefore, the aim of this thesis is to perform forest tree detection in a LiDAR point cloud using a 3D object detection model. The chosen model is the VoteNet model, presented by Qi et al. in 2019. A UAV-mounted LiDAR sensor was used to acquire an aerial forest point cloud of a mixed forest located in Sart-Tilman, within the city of Liège, Belgium. These data were manually labelled using the CloudCompare software and subsequently used as training dataset for the model. The model and dataset were implemented and tested in the TorchPoints3D Python framework. Familiarization with this framework constituted a significant portion of the work undertaken for this master thesis.

Although this study led to the implementation of a deep learning algorithm for 3D object detection, its applicability to tree detection in heterogeneous environments could not be demonstrated. One of the major aspects that can be improved to enhance the results is the amount and quality of training data. Due to a time-consuming and laborious labeling method, only a part of the dataset could be labeled. Having a large dataset is imperative for obtaining effective models in deep learning. Thus, before changing the model's architecture or adjusting additional parameters, it is crucial to label more data.

Additionally, gaps in this research field have been identified through this work. One issue is the absence of open-source point cloud labeling software capable of creating bounding boxes for large point clouds. Hence, none of this type of software was found, thus necessitating the use of the software CloudCompare for labeling point clouds, which was tedious and time-consuming. Another issue, as far as we know, is the absence of open point cloud datasets of forest environments for 3D object detection, making it necessary to acquire and label data.

Therefore, this work does not provides enough confidence in the applicability of deep learning for 3D tree detection in point clouds. However it offers a first approach to aerial 3D tree detection using deep learning, an up-to-date and detailed installation procedure of TorchPoints3D framework, an heterogeneous forest point cloud dataset and a labeling methodology to label point clouds for object detection. It also includes an extension of TorchPoints3D to incorporate this dataset and use it to train the VoteNet model.

Résumé

Réaliser de manière efficace un inventaire forestier précis est une étape importante pour une gestion optimale des forêts. De nombreuses études se sont intéressées à ce sujet. Une méthode efficace pour acquérir des données sur la structure horizontale et verticale des forêts sur de larges zones est d'utiliser un capteur light detection and ranging(LiDAR) monté sur un drone. Diverses méthodes basées sur ces données ont été explorées dans la littérature. Les méthodes traditionnelles utilisent souvent des algorithmes basés sur des règles ou des techniques de machine learning. Cependant, le point faible de ces méthodes est leur dépendance au site d'étude, nécessitant que leurs paramètres soient déterminés en fonction du type de forêt étudié. Elles sont plus difficilement applicables en contexte forestier hétérogène. Une solution récente est le développement des modèles de détection d'objets par deep learning. Ces méthodes sont de plus en plus étudiées dans la littérature, dans divers domaines de recherche. Leur avantage est que, grâce à un jeu de données d'entrainement suffisamment large et varié, un modèle de détection d'objets peut apprendre à détecter avec précision des objets de différents types et dans des contextes divers. L'un de leurs principaux inconvénients est que leur processus d'entraînement est chronophage et nécessite des ensembles de données vastes et variés, ainsi que des ressources de calcul importantes. Ce domaine a été largement exploré pour la détection et la segmentation d'objets en 2D, avec des modèles pré-entraînés désormais disponibles, capables de détecter diverses classes et d'être appliqués à différentes tâches sans nécessiter de ré-entraînement, ou nécessitant seulement quelques ajustements, comme c'est le cas avec le modèle You Only Look Once (YOLO). Cependant, à notre connaissance, aucune étude n'a appliqué ce type de modèles de détection d'objets directement aux nuages de points 3D pour la détection des arbres en contexte forestier.

De ce fait, le but de cette étude est de réaliser de la détection d'arbres en contexte forestier, en utilisant un modèle de deep learning. Le modèle choisi est le modèle VoteNet présenté par Qi et al. en 2019. Un capteur LiDAR embarqué sur un drone a été utilisé pour obtenir un nuage de points au sein d'une forêt mélangée au Sart-Tilman, situé dans la ville de Liège en Belgique. Ces données ont été annotées manuellement à l'aide du logiciel CloudCompare, et utilisées en tant que jeu de données d'entrainement pour le modèle. Le modèle et le jeu de données ont été implémentés et testés dans le framework Python TorchPoints3D. La prise en main de ce framework a constitué un partie importante du travail réalisé pour cette étude.

Bien que cette étude ait permis le développement et l'implémentation d'un algorithme de deep learning pour réaliser de la détection d'objets, son applicabilité en contexte forestier mixte n'a pas pu être démontrée. Un des aspects majeurs de cette étude qui pourrait être amélioré pour obtenir de meilleurs résultats est le manque de données d'entrainement. En raison du temp requis et de la difficulté pour annoter les données, seulement une partie du jeu de données a pu être annoté. Or, pour obtenir un modèle de deep learning efficace, il est important d'avoir un large jeu de données. Ainsi, avant de modifier l'architecture du modèle ou d'ajuster davantage de paramètres, il est important de labelliser une plus grande quantité de données.

De plus, des lacunes dans ce domaine ont été mises en évidence par ce travail. L'une d'entre elles est l'absence de logiciels open-source pour la labellisation de nu-

ages de points, permettant de créer des bounding boxes dans de grands nuages de points. Par conséquent le logiciel CloudCompare a été utilisé pour annoter les nuages de points, ce qui s'est avéré chronophage et compliqué. Un autre problème est l'absence, à notre connaissance, de jeux de données publiques de nuages de points d'environnements forestiers annotés pour de la détection d'objets 3D, ce qui rend nécessaire l'acquisition et l'annotation de données.

Par conséquent, ce travail ne fournit pas de résultats permettant de prouver l' applicabilité du deep learning pour la détection d'arbres dans des nuages de points. Cependant, il propose une procédure d'installation complète et à jour du framework TorchPoints3D, un jeu de données de nuages de points aériens de milieux forestiers hétérogènes, ainsi qu'une méthode de labellisation de nuages de points pour la détection d'objets. Il inclut également une extension de TorchPoints3D pour intégrer ce jeu de données et l'utiliser pour entrainer le modèle VoteNet

Contents

List of Figures

Chapter 1

Introduction

Forests hold significant importance in various major areas. They act as carbon sinks, providing a crucial ecosystem service to mitigate global climate change (Jonard et al., 2011; Le Noë et al., 2021). Additionally, forests are vital for biodiversity. However, their degradation, caused by climate change and human activity, poses a serious threat (Liang et al., 2016). The current state and dynamics of forests are influenced by a combination of short-term events, human activities, and long-term variations linked to climate change. To aid in forest management and policy-making, it is essential to understand and monitor these dynamics (Banskota et al., 2014). Efficient forest monitoring is crucial for assessing the economic value of forests and for tracking disease and fire risks (Hershey et al., 2022; Windrim & Bryson, 2020).

Accurate forest inventories are essential for the efficient management and sustainable development of forests (Latella et al., 2021). With the growing need for precise and accurate forest inventories, remote sensing is playing a critical role. In many countries these inventories have traditionally been based on ground samples or human interpretation of aerial photography, supplemented with ground plots. However, these methods are time-consuming and costly, with low scalability. Remote sensing offers an interesting alternative by allowing the coverage of larger areas with economies of scale (White et al., 2016). It can provide complementary data (e.g., thermal, structural, density, health status information) through various remote sensing technologies that have emerged, such as airborne and terrestrial laser scanning (ALS and TLS), thermal infrared technologies, high spatial resolution (HSR) satellite imagery and multi- and hyperspectral imagery. In the scope of this thesis, the focus is on ALS as it provides insights into the 3D structure of forests. Similar information can also be obtained from TLS, with a higher point density under the canopy. However, TLS does not yet allow to cover large areas as efficiently as ALS (Wallace, Musk, & Lucieer, 2014).

ALS uses a LiDAR sensor mounted on an aircraft, such as an unmanned aerial vehicle (UAV). LiDAR stands for light detection and ranging, it is an active remote sensing technology that enables the creation of a point cloud by acquiring range measurements. This scanner emits nanosecond infrared pulses that are scattered back to the sensor by the scanned objects. By computing the time of flight of the signal, the distance between the object and the receiver can be determined (Hall et al., 2011; Lim et al., 2003). It is useful in forestry applications as it can penetrate the canopy given sufficient laser energy, generating multiple returns and allowing the modeling of trees structures (Bates et al., 2021; Hall et al., 2011; Neuville et al.,

2021).

A first step to conduct a precise forest inventory is to detect individual trees. This task is valuable on its own and also serves as the initial step for further applications, such as species identification, health monitoring, and more (Diez et al., 2021; Wang et al., 2023). Based on ALS, there are multiple possibilities for tree detection. These methods can be classified into three categories based on the type of data they use: raster-based methods, point-based methods, and hybrid methods (Li et al., 2023). The raster-based methods require converting the 3D point cloud into a 2D raster and then performing tree segmentation on this raster. The raster is often a canopy height model (CHM), which is a model of tree elevation normalized with respect to the ground (Eva & Johan, 2017). Segmentation techniques used on 2D rasters usually rely on local maxima detection followed by crown segmentation using computer vision algorithms such as region growing, watershed, and mean-shift clustering algorithms. The main issue with these methods is the loss of information due to the conversion of the 3D point cloud into a 2D raster (Li et al., 2023; Zhen et al., 2016). Point-based methods perform tree segmentation directly on the 3D point cloud or on a voxelized point cloud. Traditionally, two popular methods in this group are K-means clustering and voxel-based single tree segmentation (Zhen et al., 2016). The K-means clustering technique is an unsupervised machine learning technique that divides the dataset into clusters with high similarity within the same cluster and high dissimilarity between different clusters. These measures of similarity are based on the distance between points in the d-dimensional Euclidean space formed by their variables (with d being the number of variables) (Sinaga & Yang, 2020). This technique requires a priori knowledge of the number of clusters. In the context of tree segmentation, the number of local maxima computed is often used for this purpose. Local maxima are then used as seed points for the model (points around which the clusters will be formed) (Zhen et al., 2016). Consequently, this detection method is dependent on these seed points. The voxel-based method involves creating volumetric pixels (voxels) to form a voxel layer, which is a dimensionally regular grid of cubes, with each cell containing data. A voxel layer can be seen as the 3D equivalent of a raster (Hussein, 2011). Based on these voxel layers, many operations can be performed to segment trees and extract their attributes. Among these is the K-means technique explained above, applied here in 3D and using either additional data or local maxima as seed points (Zhen et al., 2016). Hybrid methods combine the two previous types of methods. An example of an hybrid method is the one presented by Reitberger et al. (2009), where a watershed segmentation is first applied to obtain an initial segmentation. Hierarchical clustering is then used to extract stem positions based on this segmentation. Finally, a normalized cut segmentation is employed to segment the trees more precisely using these stem positions. These three types of methods, raster-based, point-based and hybrid, have limitations when it comes to identifying non-dominant trees (Wang et al., 2023). However, in recent years, there have been significant advances in 3D data processing, notably in deep learning. These advances have been driven by increasing demands in applications such as robotics and self-driving cars and are now being incorporated into remote sensing and LiDAR forestry applications (Windrim & Bryson, 2020). However, to the best of our knowledge, studies applying these promising techniques to forestry are still few.

Chapter 2

Theoretical concepts

2.1 LiDAR data

The following section decribes the basic principles of LiDAR, it is based on Guo et al. (2023).

2.1.1 Ranging measurments

Light detection and ranging (LiDAR) is an active remote sensing technology that relies on the emission of laser beams and the computation of distances to objects. It measures the reflection of these beams by objects and computes their times of flight (ToF). Thus, distance between the object and the sensor can be computed using the following equation:

$$R = \frac{1}{2} * c * t \tag{2.1}$$

with

- R the distance between the UAV and the object
- c the speed of light in air
- t the time of flight of the laser beam

LiDAR measurements consists in recording a time interval between the emission of a pulse and its reception. It consists of four steps:

- Laser emission: emission of a narrow laser pulse from a laser transmitter through a scanning prism.
- Laser detection: detection of the back-scattered pulse by the laser receiver and its conversion to an electric signal.
- Time delay estimation: analysis the echoed signal compared to the reference signal to compute the time delay of the range measurement. The most common method is cross-correlation (Li et al., 2013) (see Figure 2.1).
- Measurements: utilization of a precision instrument controlled by an atomic clock to accurately measure the emission and reception times.

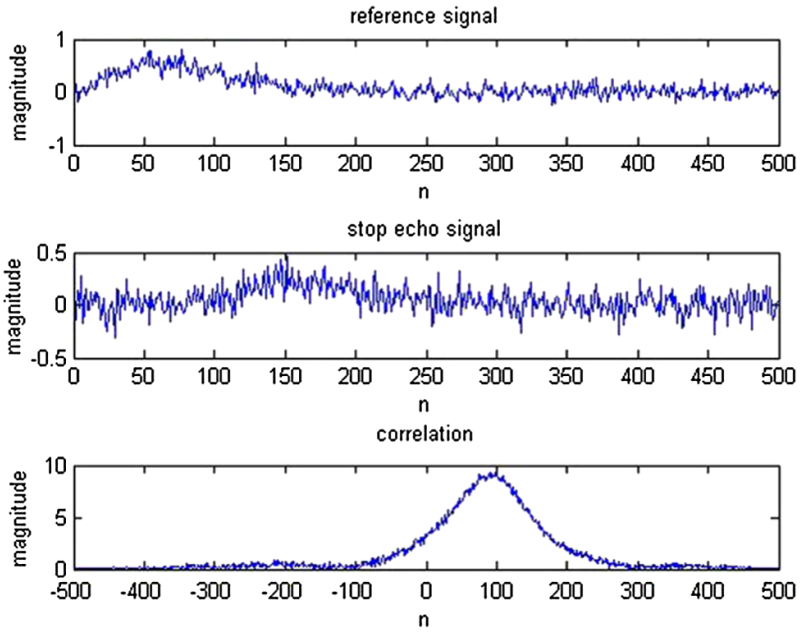


Figure 2.1: Cross-correlation time delay evaluation (Li et al., 2013).

The ranging resolution is the smallest distance required between two objects for them to be distinguishable by the system. It is directly dependent on the time resolution, and it is computed using the following formula:

$$\Delta R = \frac{1}{2} * c * \Delta t \tag{2.2}$$

2.1.2 Ranging accuracy

The ranging accuracy is proportional to the square root of the signal-to-noise ratio (SNR). The SNR is influenced by various parameters (see Figure 2.2). If the received signal is sufficiently weak that the thermal noise is dominant, the ranging accuracy for pulse ranging can be approximated as follows:

$$\sigma_{R_{pulse}} \sim \frac{c}{2} t_{rise} \frac{\sqrt{B_{pulse}}}{P_{R_{peak}}} \tag{2.3}$$

Where

- $\sigma_{R_{pulse}}$ is the ranging accuracy of pulse ranging method.
- t_{rise} is the rise time. The time taken by a pulse to go from a lower specified value, near the lowest possible (usually 10% of the signal amplitude), to a higher specified value, near the peak signal value (usually 90% of the signal amplitude) (Weik, 2001).
- B_{pulse} is the noise bandwidth [Hz].
- $P_{R_{peak}}$ is the peak power of the received pulse [W].

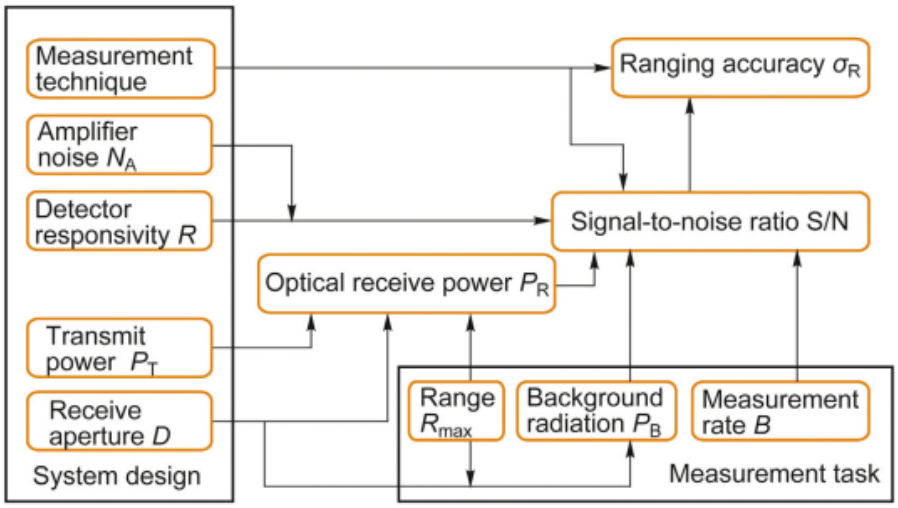


Figure 2.2: Influence of various parameters of LiDAR on the SNR and ranging accuracy (Guo et al., 2023).

Moreover, the maximum detection distance is determined by the pulse frequency. It is limited due to mutual interference between emitted and received pulses. As a consequence, a new pulse is generally emitted only after the previous one has been received. Therefore, for a given pulse emission frequency, a maximum interference-free distance can be computed as follows:

$$R_{max} = \frac{1}{2} * c * PRF^{-1} \tag{2.4}$$

Where

- R_{max} is the maximum interference free distance.
- PRF is the pulse repetition frequency.

Thus, to avoid pulse interference, the PRF cannot be too high. This results in a lower point density. A solution to increase this density is to use multi-frequency LiDAR sensor. Another solution is to increase the frequency and use data processing algorithm to determine the period interval between transmitted and emitted pulse.

2.1.3 Radiation principle

The LiDAR equation allows to quantify the change of laser energy during the emission-reflection-reception phases. It computes the received power(P_r) based on the energy density of the scattered pulse (see Equation 2.5), the scattering power of the object (P_s) (see Equation 2.6) and the energy density of the returned signal at the receiver (d_r) (see Equation 2.7).

$$d = \frac{P_e}{A_s} = \frac{4P_e}{\pi\alpha^2 S^2} \tag{2.5}$$

$$P_s = d * A * \rho \tag{2.6}$$

$$d_r = \frac{P_s}{\beta * S^2} \tag{2.7}$$

Thus

$$P_r = d_r \cdot \pi \cdot \left(\frac{D}{2}\right)^2 = \frac{d \cdot A \cdot \rho}{\beta \cdot S^2} \cdot \pi \cdot \left(\frac{D}{2}\right)^2 = P_e \cdot \frac{A \cdot \rho \cdot D^2}{\pi \cdot S^2} \tag{2.8}$$

Where :

- P_e is the energy of the emitted laser.
- A_s is the area of the target object illuminated by the laser pulse. It can be computed using the object-emitter distance (S) and the angle of the receptive field of the emitter (α) (see Figure 2.3):

$$A_s = \frac{\pi \alpha^2 S^2}{4} \tag{2.9}$$

- A is the projected surface of the target object in the direction of the laser sensor.
- ρ is the reflectivity of the object.
- β is the angle in which the energy back-scattered by the target object is concentrated (see Figure 2.3).
- D is the diameter of the laser receiver (see Figure 2.3.

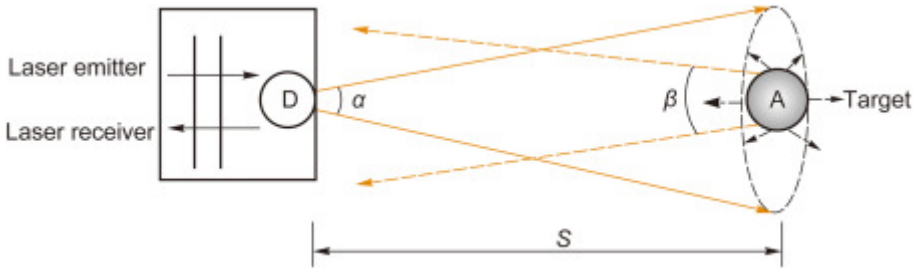


Figure 2.3: Laser pulse detection with α the angle of the receptive field of the laser emitter, β the solid angle where the energy back-scattered by target A is concentrated, S the emitter-target distance and D the diameter of the laser receiver (Guo et al., 2023).

This equation is the basis of all LiDAR systems, as it describes the change of energy of an emitted pulse before being received. It is mainly used to derive information about the medium of propagation or the scattering objects (Ni-Meister et al., 2010; She & Friedman, 2022).

LiDAR systems can be divided into two groups: discrete-return systems and full-waveform systems.

The discrete-return system collect back-scattered pulse as discrete returns. A back-scattered pulse is considered as a return if its energy is above a given threshold. A same pulse can generate multiple returns (see Figure 2.4). Those different returns offers information of ground occupation along with the elevation and distribution of the different reflective surfaces of the sensed area. The thresholds are set to prevent the detection of unnecessary returns, thereby reducing storage requirements. However, this may affect the accuracy of the 3D reconstruction.

On the other side, the full-waveform method record the complete waveform of the back-scattered signals (see Figure 2.4). It offers more detailed information about the 3D structure of the target. However this method requires more storage capacity and more complex data processing. The first step to process such data is generally to convert it into discrete point cloud.

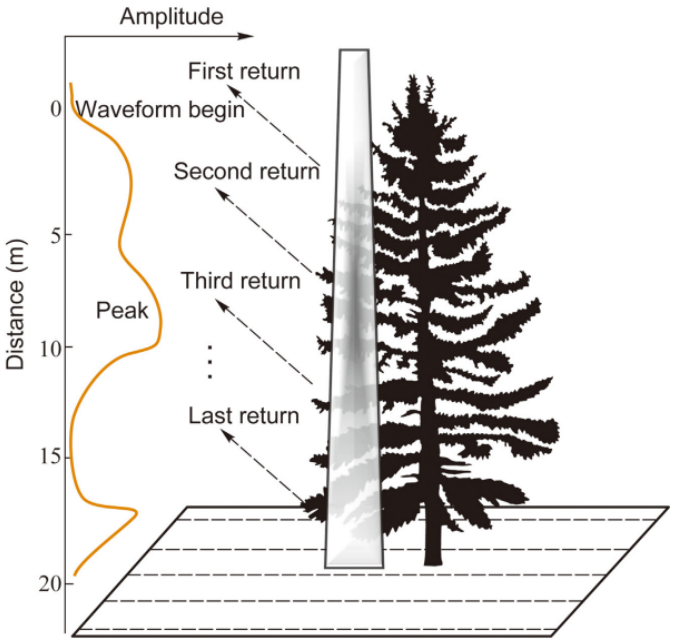


Figure 2.4: Discrete-return and full-waveform LiDAR systems (Guo et al., 2023).

2.1.4 Aerial LiDAR

Aerial LiDAR refers to LiDAR sensors onboard of a flying platform such as a UAV or manned aircrafts.

Systems components

Airborne LiDAR systems include three parts: a laser ranging system, a positioning and operating system (POS) and a data synchronization system.

The laser ranging system is responsible for acquiring the LiDAR data. It is itself composed of three parts:

- Ranging unit: This part is composed of a laser transmitter and a laser receiver. It is responsible for emitting and receiving the laser signal.
- Optical scanning unit: This part consists of a prism and a rotating mechanism to control the direction of the emitted laser. This mechanism can be of four types (see Figure 2.5):
    1. Oscillating mirror: The laser angle is controlled by a mirror oscillating around the central axis at a constant frequency. It produces a periodically changing footprints distribution on the ground (see Figure 2.6). Thus resulting in a point cloud with an non-homogeneous point distribution, with lower density on the edges along the flying direction.
    2. Rotating polygon: The laser angle is controlled by a reflecting polygon rotating around the horizontal axis. In this case the footprints are evenly distributed. However, unlike the rotating mirror, the scan angle cannot be changed.
    3. Palmer scan: The laser angle is controlled by a prism rotating around an axis of rotation but not perpendicular to it. This results in an elliptical scan pattern (see Figure 2.6). The footprints of this pattern have a degree

of overlap along the flight direction. Thus, the point density is increased in these overlapping areas and can reduce the number of occluded regions.

4. Fiber scanner: The laser angle is controlled by transmitting and receiving arrays of fibers along with transmitting and receiving lenses. The arrays are mounted in the focal planes of the lenses. Rotating mirrors, one for each array, relay signal from the central fiber to one of the fibers in the array mounted in a circle around the central fiber. The optical signal from the transmitting fiber is linked to the corresponding fiber in the receiving path (see Appendix a.1). This system scans lines on the ground parallel to the flight direction (see Figure 2.6). Due to the small aperture of the fibers the scanning speed is high, resulting in a high point density.

- Control and processing unit: This part is responsible for coordinating the other two components and processing the data collected by them.

The POS system is composed of an inertial navigation system (INS) and of a differential global navigation satellite system (DGNSS). It is responsible for computing the aircraft's location. The INS records the position, speed, altitude and heading of the aircraft. The DGNSS also computes the position of the aircraft. Thus the DGNSS position is integrated with the INS position estimation to make it more stable.

The data synchronization control system is responsible for acquiring, synchronizing and recording data from the above components. It employs a clock control system to synchronize the INS receiver, the GNSS receiver and the laser scanners.

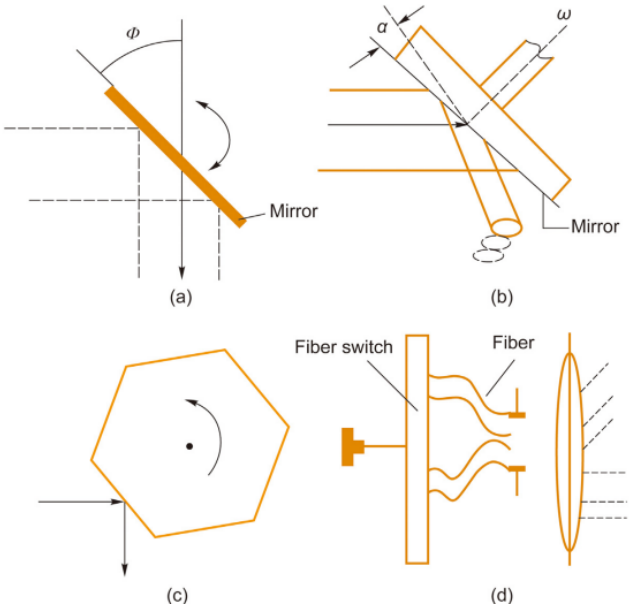


Figure 2.5: Four scanning modes of airborne LiDAR. (a) Oscillating mirror, (b) palmer scan, (c) rotating polygon and (d) fiber scanner. ϕ is the inclination angle of the mirror; α is the angle between the normal of the mirror and the rotation axis ω (Guo et al., 2023).

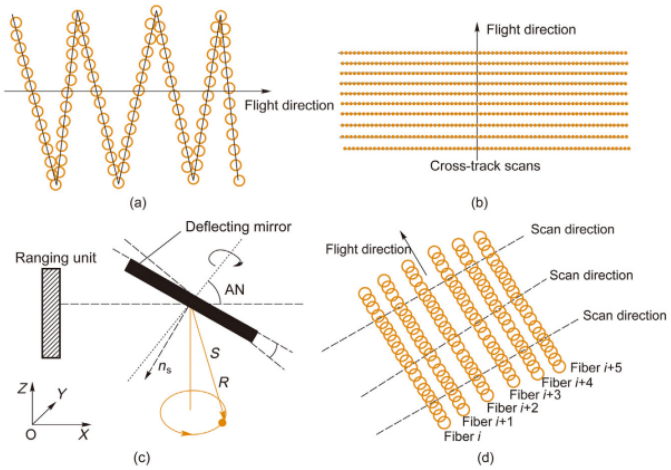


Figure 2.6: Distribution of footprints produced by airborne LiDAR with the mechanism of (a) oscillating mirror, (b) rotating polygon, (c) palmer scan and (d) fiber scanner. AN is the angle between the laser pulse and the normal of the rotation axis; S is the distance from the center of the mirror to the ground; R is the radius of the circle formed by footprints on the ground; and n_s is the normal of the mirror (Guo et al., 2023).

LiDAR sensor parameters

Different metrics are used to asses airborne LiDAR systems (see Figure 2.7). The point density, which is the number of points per square meter, is commonly used to represent the quality of the point cloud. It is generally inversely correlated with flight height and speed and positively correlated with the laser pulse frequency. Additionally there is the field of view (FOV), which refers to the angular range of acquisition of the LiDAR. Another metric is the flight height (H), it depends on the

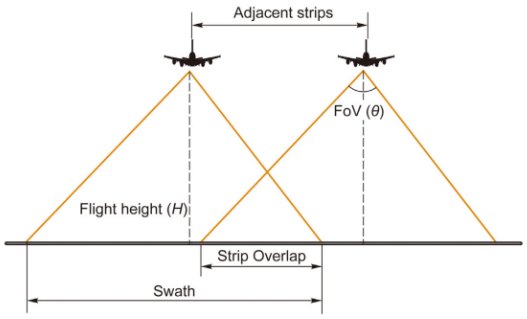


Figure 2.7: Schematic diagram of the parameters of airborne LiDAR systems (Guo et al., 2023).

studied zone, the laser pulse frequency, the FOV, the platform type and the flight campaign (purpose and budget). An additional metric is the swath width (SW) which represents the width of the area covered by a single pass of a LiDAR sensor. It depends on the flight height and the FOV (θ) and can be computed as follows:

$$SW = 2H \tan \frac{\theta}{2} \tag{2.10}$$

The lateral overlap ratio (ξ) is also used. It represents the portion of an acquisition strip that overlaps with an adjacent strip. It depends on SW and the distance

between two strips (e) and is computed as follows:

$$\xi = 1 - \frac{e}{SW} \tag{2.11}$$

There is also the laser pulse frequency and power, which are respectively the number of pulses emitted per second and the energy of each emitted pulse. Furthermore, there is the number of returns, it depends on the performance of the laser scanner and the occupation of the acquisition zone. Another important parameter of LiDAR sensors are the wavelength of the beams they are emitting. This wavelength can go from near ultra-violet ($\sim$ 300 nm) to far infrared (1 mm) (Angevine et al., 2003). In the context of forest applications the chosen wavelength generally range from 900 to 1064 nm, due to the high reluctance of vegetation in these wavelengths (Lefsky et al., 2002). Finally, there is the range resolution, which is the shortest distance required between two objects for them to be individually detected by the LiDAR.

2.2 Deep learning

The following section presents the basics of deep learning along with the concept of object detection within deep learning. Except for the part discussing object detection, the content of the following sections are based on Razavi (2021), and on Ravikumar et al. (2024).

2.2.1 Multi layer perceptron

The primary element of a deep learning network is the perceptron (also called a neuron) (see Figure 2.8a). It is the smallest computational unit in deep learning and performs the following operation:

$$y = f(\sum_{i=1}^{D} w_i x_i + b) \tag{2.12}$$

where

- D is the dimension of the input space
- **x** is the D-dimensional input vector
- **w** is the D-dimensional weight vector
- b is the bias
- f is the activation function.

The weights and the bias are the trainable parameters of the perceptron. The activation function is a non-linear function responsible for introducing non-linearity into the model. The capacity of a single perceptron, representing the complexity of the functions it can approximate, is limited. To increase this capacity the solution is to stack perceptrons both in parallel and in series, forming a multi-layer perceptron (MLP) (see Figure 2.8b). This configuration makes the model more complex and increases the number of trainable parameters: the first "hidden" layer, with n_1 neurons, has D $\cdot$ n_1 weights and n_1 biases; the second, would have $n_1 \cdot n_2$ weights

and n_2 bias; etc. Each neuron in a layer uses the same activation function, but different layers can use different activation functions.The choice of these functions is an important aspect of defining the model.

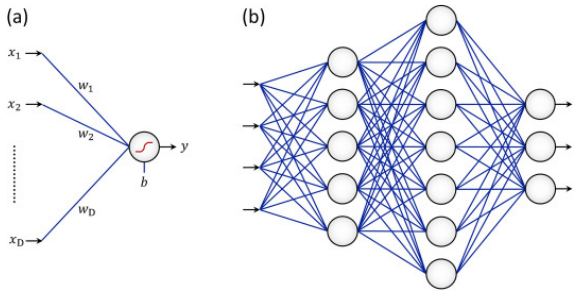


Figure 2.8: A perceptron (a) and a MLP (b) (Razavi, 2021).

Hence a MLP with two layers, the input and output layers, essentially represents a multivariate linear regression operation. The values of the different neurons before the application of the activation function, can be represented as follows :

$$f(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{w}_0 \tag{2.13}$$

Where $\mathbf{x}$ is the D-dimensional vector of the input features, representing one sample. $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{w}_0\}$ are the trainable parameters of the model, where $\mathbf{W}$ is a M $\times$ D matrix and $\mathbf{w}_0$ is a M-dimensional vector and y is the output M-dimensional vector. This formulation can be compacted by incorporating $\mathbf{w}_0$ as a column in $\mathbf{W}$, resulting in:

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_M \end{bmatrix} = \begin{bmatrix} w_{10} & w_{11} & \dots & w_{1D} \\ w_{20} & w_{21} & \dots & w_{2D} \\ \dots & & & \\ w_{M0} & w_{M1} & \dots & w_{MD} \end{bmatrix} \times \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \dots \\ x_D \end{bmatrix} \tag{2.14}$$

This equation represents the forward pass through the 2-layer MLP and produces a prediction $\hat{\mathbf{y}}$ of the target vector $\mathbf{y}$ based on the input sample $\mathbf{x}$. The layer's activation function is then applied to these results. If an additional layer is added after the second one, the output of the activation function serves as input to this new layer.

2.2.2 Activation functions

Activation functions are crucial components of deep learning models, as they introduce non-linearity into the outputs of neurons. They allows model to approximate complex, non-linear functions between inputs and outputs.

One of the simplest activation functions is the sign function:

$$sign(x) = \begin{cases} 1 & \text{if x} > 0 \\ 0 & \text{Otherwise} \end{cases} \tag{2.15}$$

Since then, many other activation functions have been proposed. Among them, two have been widely used: the sigmoïd function (see Equation 2.16) and the hyperbolic tangent (tanh) (see Equation 2.17). The sigmoïd function transforms the input into a value between 0 and 1 (see Figure 2.9), while the tanh function has a similar shape

to the sigmoïd (see Figure 2.9) but is bounded between -1 and 1.

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.16}$$

$$tanh(x) = \frac{2^{2x} - 1}{e^{2x} + 1} \tag{2.17}$$

However, a problem associated with these functions is that they are bounded. This results in their gradients being close to zero. Consequently, during the back-propagation phase (see Section 2.2.4), gradients from layers close to the output layer will be small and these small gradients will be further diminished as they propagate back through the network. As a result, for models with high number of layers, the gradient will not reach the earlier layers, preventing their parameters from being updated. This effect is known as the vanishing gradients problem.

A solution to this problem is proposed by the rectified linear unit (ReLU) activation function, which is now one of the most widely used. It is defined as follows:

$$ReLU(x) = max(x, 0) = \begin{cases} 0 & \text{x} < 0 \\ x & \text{x} \geq 0 \end{cases} \tag{2.18}$$

As its outputs are unbounded, it helps address the problem of vanishing gradients. Additionally, because the output of a neuron is zero if its input value is below 0, ReLU introduces sparsity into the model, which can accelerate the convergence of the network. However, a drawback of the ReLU function is the "dead ReLU" problem. A dead ReLU refers to a neuron whose weights have been updated such that it always produces a negative value for all inputs, blocking the gradient descent for that neuron. To address this issue several variants have been proposed, such as leaky ReLU (LReLU) defined as $LReLU(x) = max(x, \alpha x)$ where α is a small positive constant (e.g. $\alpha = 0.1$).

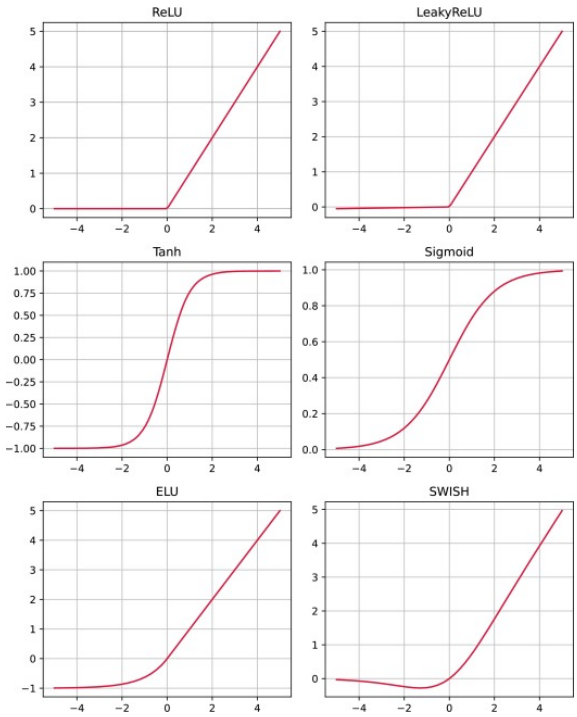


Figure 2.9: Activation functions shapes (Ravikumar et al., 2024).

Finally, there is the softmax activation function which is commonly used as the activation function for the output layer of a classification model. This activation function produces a vector of probabilities (values between 0 and 1, with their sum equal to one) based on a vector of scores (vector of C values z). The probabilities produced depends on the relative magnitude of the values in the input vector (Ren & Wang, 2023). It is computed as follows:

$$softmax(z_c) = \frac{e^{z_c}}{\sum_{j=1}^{C} e^{z_j}} \tag{2.19}$$

Once a model's architecture is defined, it needs to be trained on a training dataset. This training is an iterative optimisation process during which the trainable parameters are gradually adjusted to better fit the problem.

2.2.3 Loss function

The first step in training a model is to define a loss function. This function quantifies the discrepancies between the model's predictions and the expected outputs. For example, a common loss function for regression problems is the mean square error (MSE) loss:

$$L_{MSE} = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{y}_n - \boldsymbol{f}(\boldsymbol{x}_n))^2 \tag{2.20}$$

where

- N is the size of the training dataset.
- $\boldsymbol{y}_n$ is the target value for the nth sample of the dataset.
- $\boldsymbol{f}(\boldsymbol{x}_n)$ is the prediction of the model for the features of the nth sample of the dataset.

For classification problems a widely used loss function is the cross-entropy (CE) loss:

$$L_{CE} = \frac{1}{N}\sum_{n=1}^{N}(-\sum_{c=1}^{C} y_c log(p_c)) \tag{2.21}$$

Where

- C is the number of classes in the classification problem.
- y_c is the binary ground truth label for class c. It is either 0 if a sample does not belong to the class c, and 1 if it does.
- p_c is the probability output by the model that the sample belongs to class c.

This CE loss is computed for each sample, then its mean over the dataset is used to compute the loss for the forward pass.
The goal of training a deep learning model is to find the optimal parameters $\boldsymbol{W}_*^d$ of the model that minimize the loss function L on the dataset d:

$$\boldsymbol{W}_*^d = arg\min_{\boldsymbol{W}} L(\boldsymbol{W}) \tag{2.22}$$

2.2.4 Backward pass

The training of a model is an iterative process composed of two steps: the forward and the backward pass. During the forward pass, predictions are made for each sample in the training dataset, and the loss function is computed for these predictions. Afterward, in the backward pass, the different parameters of the model are updated using a back propagation algorithm. This update is determined by the gradient of the loss function with respect to the different trainable parameters $\mathbf{W}$ of the model. This gradient $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}}$ consists of the partial derivatives of the loss with respect to each parameter. This gradient indicates the direction in which the loss increases in the parameter space. Therefore, the model parameters are adjusted in the opposite direction of this gradient to minimize the loss. This increment added to each parameter is based on this gradient and determined by an optimisation algorithm.

2.2.5 Optimisation algorithm

As previously explained, to train a deep learning model the gradient is used to update the parameter values through an optimization algorithm. The simplest of these algorithms is gradient descent which updates the weights as follows:

$$\boldsymbol{W}^{t+1} = \boldsymbol{W}^{t} - \eta \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \boldsymbol{l}(\boldsymbol{x_i}), \boldsymbol{W^t}}{\partial \boldsymbol{W}^{t}} \tag{2.23}$$

where

- N is the number of samples in the dataset.
- t is the current training iteration (epoch).
- η is the learning rate which controls the magnitude of the changes.

If the gradients used are computed with the entire training dataset, it is known as the batch gradient descent. However, this method is generally not used because it requires significant computational resources and can lead to overfitting of the training dataset by the model. Thus the network would fit the noise in the training dataset, reducing its generalisation to unseen data. To monitor this overfitting a validation dataset is introduced. At each epoch a loss is computed for this validation dataset, but the gradient is never computed on this dataset, thus it does not impact the model's training. It is used solely to assess the model's performance.

Another solution to reduce overfitting is to use subsets of the dataset to compute the gradients. This leads to the stochastic gradient descent (SGD) method, which, unlike batch gradient descent, computes the gradient after each sample. This method introduces higher variance in the training steps. The mini-batch SGD was proposed to mitigate this variance. It divides the training dataset into subsets of few samples (mini-batches) and computes the gradient for each mini-batch. This approach combines the benefits of SGD with reduced variance. It is computed as follows:

$$\boldsymbol{W}^{t+1} = \boldsymbol{W}^{t} - \eta \frac{1}{N_b} \sum_{i \in B_t} \frac{\partial \boldsymbol{l}(\boldsymbol{x_i}), \boldsymbol{W^t}}{\partial \boldsymbol{W}^{t}} \tag{2.24}$$

Where

- N_b is the number of samples in each mini-batch.

- B_t is the tth mini-batch.

In this configuration, the samples are randomly split into mini-batches without repetition.

Another widely used optimization algorithm is the adaptive moment estimation (ADAM). This algorithm add a momentum to the gradient to enable faster convergence and reduce the noise of the gradient. This momentum is implemented by incorporating a fraction of the gradient from the previous iteration into the current gradient. Furthermore Adam utilizes two moments: the first one (V) represents the exponentially weighted moving average of past gradients (see Equation 2.25) while the second one (U) represents the exponentially weighted moving average of past squared gradients (see Equation 2.26):

$$\boldsymbol{V}_t = \beta_1 \boldsymbol{V}_{t-1} + (1 - \beta_1)\boldsymbol{G}_t \tag{2.25}$$

$$\boldsymbol{U}_t = \beta_2 \boldsymbol{U}_{t-1} + (1 - \beta_2)\boldsymbol{G}_t^2 \tag{2.26}$$

where

- $\boldsymbol{G}_t$ is the mini-batch gradient computed for the tth iteration
- β_1 and β_2 are non-negative hyperparameters that control the decay rates of these moving averages.

Both of these moments are initially set to zero. During the initial training phase, both moments tend towards zero, which can cause biased estimates. To correct for this bias, a bias correction term is introduced as follows:

$$\hat{\boldsymbol{V}}_t = \frac{\boldsymbol{V}_t}{1 - \beta_1^t} \tag{2.27}$$

$$\hat{\boldsymbol{U}}_t = \frac{\boldsymbol{U}_t}{1 - \beta_2^t} \tag{2.28}$$

Based on these corrected moments, the update applied to the weights is computed as follows:

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta \frac{\hat{\boldsymbol{V}}_t}{\epsilon + \sqrt{\hat{\boldsymbol{U}}_t}} \tag{2.29}$$

Where ϵ is a small value (generally $\sim 10^{-6}$), it is included to prevent dividing by zero.

2.2.6 Normalization

Furthermore, in addition to the previous steps, another fundamental concept in deep learning for efficiently training a model is normalization. These techniques enhance and accelerate model convergence by ensuring that the data maintain certain statistical properties as they pass through the network. They are commonly used to stabilize training and improve generalization. Normalization encompasses a range of techniques, with batch normalization being the most widely used.

Batch normalization (BN) is used to reduce the internal variance of the features distribution between the mini-batches. It is applied to one, or all, layers of a model, and it is generally done before applying the activation function to the layer. It computes the mean and standard deviation of the outputs of a layer for the current

batch. The outputs of the neurons are then adjusted by subtracting the mean and dividing by the standard deviation. It ensures that the outputs of a layer are mean-centered and have a unit variance. Therefore, the BN for a given layer is a function of its inputs ($\mathbf{x}$) and can be expressed as:

$$BN(\boldsymbol{x}) = \gamma \cdot \frac{\boldsymbol{x} - \boldsymbol{\mu}_B}{\boldsymbol{\sigma}_B + \epsilon} + \beta \tag{2.30}$$

where

- $\boldsymbol{\mu}_B$ is the mean of the layer's neurons for the batch.
- $\boldsymbol{\sigma}_B$ is the variance of the layer's neurons across the batch.
- γ and β are trainable scaling and shift parameters.

This technique stabilizes the outputs, thereby reducing gradient noise and accelerating model convergence.

2.2.7 Regularization

Finally, there are the regularization techniques which improve the generalization of a model. These techniques can be categorized as follows:

- Data-based regularization: This category encompasses data augmentation techniques. These techniques rely on stochastic transformations of the input data to artificially increase the size of the dataset. The transformations applied depend on the data being used. For example, Gaussian noise can be added to the data, or, in the case of images, transformations such as rotation, scaling, and translation can be applied.
- Network based regularization: These techniques imply incorporating specific properties directly into the model's architecture. The two main techniques in this category are the "dropout" and the "early stopping". Dropout is a stochastic method that randomly deactivates neurons of specified layers during each forward pass. Varying the combination of active neurons helps prevent neurons from "co-adapting" to the training set, thereby reducing the risk of overfitting. The early stopping consists in defining a stop criterion based on the validation loss. Generally it consists in monitoring whether the validation loss has decreased by a certain value over a specified number of epochs: if it is the case the training continues, otherwise it stops. It helps to stop the training once the validation loss has converged while the training loss is still decreasing.
- Loss-based regularization: In these techniques a regularization term is added to the loss function to penalize the model's trainable parameters. Two popular regularization terms are the L_1 and L_2 norm regularizers:

$$L_1 = \lambda ||\boldsymbol{W}|| \tag{2.31}$$

$$L_1 = \lambda ||\boldsymbol{W}||^2 \tag{2.32}$$

  In these equations the term λ is used to control the importance of the regularization parameter with respect to the loss function. The first one tends to drive some weight towards zero, inducing sparsity in the model, while the second one tends to reduce evenly the weights.

2.2.8 Object detection

Finally this section discusses about the object detection in deep learning. It is based on Zhao et al. (2019).

The goal of object detection is to localize objects in images, videos or point clouds and classify them. There are two main types of object detection approaches: two-step methods, which first generate object proposals and then classify them, and one-step methods directly addressing both problems in parallel.

The two-step approaches first scan the entire input then focus on regions of interest. Many models have been proposed to address this problem, here will be presented some of the major ones.
One of the earliest major model is the region-based convolutional neural network (R-CNN) (see Figure 2.10). It divides the problem into three stages:

1. Region proposal generation: In this stage 2000 box propositions are generated. This part does not rely on deep learning, the box proposals are generated using the selective search method. This method groups the different pixels together using various grouping strategies each relying on different features of the pixels (Uijlings et al., 2013).

2. CNN based feature extraction: Each region proposal produced at the previous stage is resized or cropped to a determined resolution and passed through a convolutional neural network(CNN) to extract 4096-dimensional feature representation for each region proposal.

3. Classification and localization: The region proposals are then passed through pretrained support vector machines (SVMs) to classify the regions into defined classes and to determine an objectness score. Thus for each region the SVMs output the most probable class and an objectness score. Regions with sufficiently high scores are retained. The proposals are then fed to a linear regression model to predict the bounding boxes more precisely. And finally the boxes are filtered with an non-maximum suppression (NMS) algorithm. This algorithm removes overlapping bounding boxes (those exceeding a certain overlap threshold) by keeping the one with the highest score.

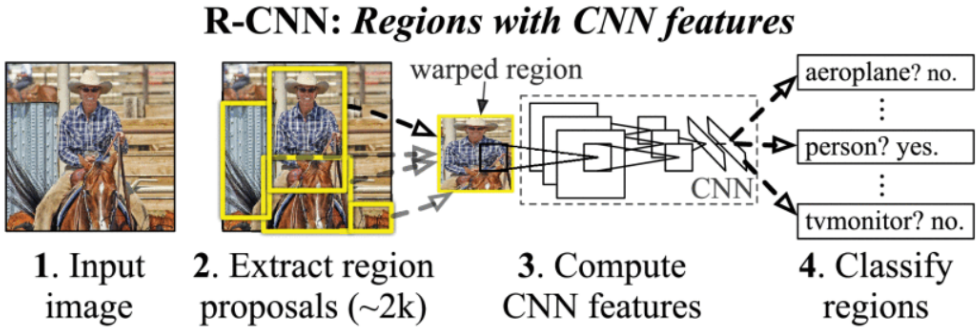


Figure 2.10: R-CNN architecture (Zhao et al., 2019).

The main issues with this model is that it needs a multi-stage training: first, training the CNN for feature extraction, and then training the bounding box regressor. Furthermore, it is memory-intensive, since high dimensional feature space is computed

for each box proposal and stored on disk. The region proposals obtained after the first stage are often redundant and their generation is time-consuming.

Further versions of this model were developed to enhance its performance. For example, the Fast-RCNN introduced a region of interest (ROI) pooling layer. Instead of extracting features from each proposed region independently, the entire image is passed through the CNN, generating the feature map of the whole image. Then the ROI pooling layer is then applied to extract a fixed-size feature vector of each region proposal. It also replaced the SVMs with a fully connected output layer of size $C + 1$ (where C represents the number of classes plus one background class) and uses a softmax activation function. Afterward the Faster-RCNN was proposed, its goal was to replace the selective search method for region proposals by a neural network. This led to the development of the region proposal network (RPN) (see Figure 2.11). This RPN uses a sliding window, implemented as a convolutional layer, which moves over the feature map produced by the initial CNN. For each position of this window, a 256-dimensional feature vector is produced and simultaneously k box proposals are generated. This k proposals correspond to k anchor boxes of different size and heading which are manually defined at the creation of the model. The 256-dimensional feature vector for each window is processed by two parallel fully connected layers. The first one predicts whether each of the k object proposals for that window contains an object, producing 2k objectness scores for each window. The second layer is a regression layer that outputs corrections for the dimensions of each object proposal to better fit the potential object, generating 4k values for each window. The object proposals are then filtered based on their objectness scores, followed by non-maximum suppression (NMS). The remaining proposals are used for ROI pooling with the initial feature map and the subsequent Fast-RCNN architecture.

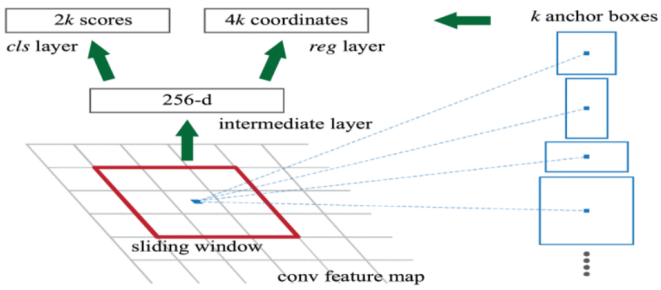


Figure 2.11: RPN construction (Zhao et al., 2019).

The problem with such frameworks is that they are composed of multiple stages, each requiring multiple training phases. To reduce training time and complexity, one-step networks were proposed. These networks predict bounding box coordinates and class probabilities from the image pixels. One of the main architectures developed in this scope is the "You Only Look Once" (YOLO) model.

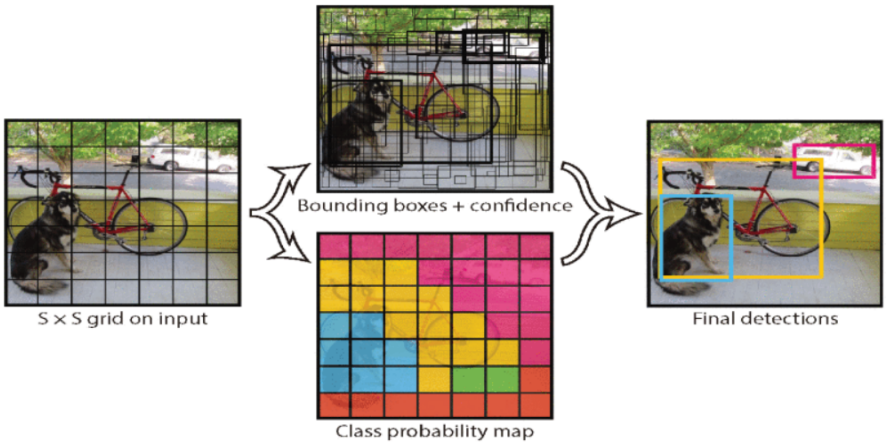


Figure 2.12: YOLO main idea (Zhao et al., 2019).

This model divides the input images into an $S \times S$ grid, where each cell predicts the bounding box coordinates for B potential objects centered on it along with the confidence score of each bounding box. In parallel each cell outputs a class probability vector. Thus, the output for each cell is B bounding box coordinates, B confidence scores and C class probabilities. The confidence score for each bounding box proposal can be seen as : $Pr(Object) \cdot IoU_{pred}^{truth}$ where Pr(object) is the probability of having an object and $I0U_{pred}^{truth}$ is the intersection over union of predicted boxes and ground-truth ones (see Figure 2.13). The class probabilities can be written as the probability of class i given that an object is present: $Pr(C_i|Object)$. Therefore, the output layer can produce a class-specific confidence score for each predicted box, which is computed as follows:

$$Pr(Object) \cdot IoU_{pred}^{truth} \cdot Pr(C_i|Object) = Pr(C_i) \cdot IoU_{pred}^{truth} \tag{2.33}$$

Finally bounding boxes are filtered based on their confidence scores, and a NMS filtering is applied, resulting in classified bounding boxes detecting the different objects in the image (see Figure 2.12). This network consists of 24 convolutional layers and 2 fully connected layers.

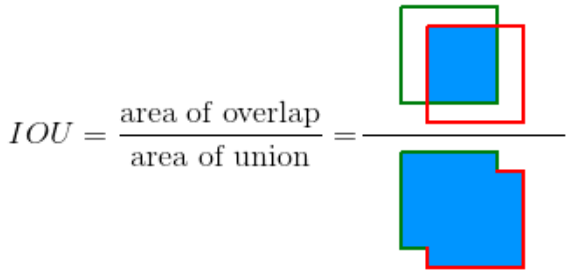


Figure 2.13: Intersection over Union (Padilla et al., 2020).

Chapter 3

State of the art

In this section, our aim is to highlight the main studies in the field of tree detection using remote sensing. These studies are categorized into two groups based on their approach: hypothesis-driven approaches and data-driven approaches. The first group includes the algorithms that rely on predefined rules on input data attributes to solve a problem. In contrast, the second group involves algorithms that learn from the data itself to identify patterns and trends.

3.1 Hypothesis-driven approach

Various approaches have been proposed to detect trees in LiDAR data. One of the most common approaches is the hypothesis-driven approach. This type of methods detect trees by verifying a set of predefined rules on the attributes of the points in the point cloud.

3.1.1 CHM based methods

One of the most common approaches involves performing local maxima detection to identify the tree tops and then delimit their crowns. Solberg et al. (2006) proposed a technique based on the creation of a Canopy Height Model (CHM) derived from the point cloud. This CHM is a 2D raster representing the height of the canopy normalised with respect to the ground elevation. This CHM is then smoothed with a Gaussian low-pass filter and a sliding window is passed through the smoothed CHM. For each position of the sliding window, a tree is identified as the pixel with the highest value in the neighborhood defined by the window. Finally, tree crowns are delimited using a region-growing algorithm with local maxima as seed points. From these seed points, regions are grown by evaluating growing windows around the seed points and adding pixels to them based on two rules: firstly, the pixel's neighbor with the steepest up-slope must be part of the region; secondly, the regions must be star-shaped, i.e., any connection between any pixel of the region and the seed point must be entirely contained within the region.

Dalponte and Coomes (2016) proposed a similar method, but with different conditions for adding pixels to the growing region. The condition are as follows: first, the height of the pixel must be above a given percentage of the tree top's height; second, the horizontal distance from the tree top must be below a given threshold.

Wallace, Musk, and Lucieer (2014) presented a method also based on CHM tree top

detection. However, for crown delimitation, they divide the detected trees into two groups based on their elevation. The first group consists of the tree tops with an above ground height (AGH) value above the 65th percentile of the AGH distribution in the point cloud. The second group is composed of the tree tops with heights below this percentile. For the first group, tree crowns are delimited using a watershed algorithm applied to the complement[1] of the CHM, where each tree crown is supposed to delimit a watershed that can be delimited using a flooding simulator (Chen et al., 2006). For the second group a normalized cut segmentation is used, which involves iteratively splitting the point cloud into two parts to minimize the variations inside each part while maximizing the variation between the parts. These variations are controlled by the following formula:

$$N_{cut}(AB) = \frac{CUT(A,B)}{ASSOC(A,V)} + \frac{CUT(A,B)}{ASSOC(B,V)} \tag{3.1}$$

where

- CUT(A,B) = $\sum_{i \in A, j \in B} w_{ij}$
- ASSOC (A,V) = $\sum_{i \in A, j \in V} w_{ij}$
- A and B are the two segments to be determined
- V is the whole set of points
- $w_{ij} = e^{-3 \times X_{i,j}} \times e^{-Z_{i,j}} \times e^{-S_{i,j}}$ is the similarity measure between point i and j, with
  - $X_{i,j}$: The normalised horizontal distance between point i and j.
  - $Z_{i,j}$: The vertical distance.
  - $S_{i,j}$: The maximum distance between the two comparison points and the nearest point of a region determined by the watershed algorithm.

This is applied recursively to the parts until they meet the termination criteria (i.e., either a segment contains less than 100 points or Ncut < 0.16). The solution of these two methods are merged based on an horizontal overlap threshold, if the overlap exceeds it the two segments are merged, otherwise they remain separate.

The underlying problem with these methods is that they rely on prior knowledge of the studied forest (such as structure, species and topography) to determine various parameters (e.g., the degree of smoothing of the filters applied, the maximum distance between tree crown's points and the summit). In addition to the knowledge required about the studied forest, these methods also necessitate a time-consuming configuration stage (Monnet et al., 2010). Moreover, while these methods can achieve reasonable detection rates (above 65%) in homogeneous coniferous forests, their accuracy significantly drops in deciduous or heterogeneous forests. A trade-off must be made between the false negative rate (the portion of trees not detected) and the false positive rate (the portion of incorrectly detected trees) (Hershey et al., 2022). The challenges in deciduous forests can be attributed to the fact that these detection methods rely mainly on information provided by the CHM, and thus do not take into account the vertical structure of the forest. To overcome these challenges, incorporating structural information of trees from LiDAR data is a promising solution (Hershey et al., 2022).

[1] a raster computed by subtracting, for each pixel, its value from the maximum value of the original raster

3.1.2 Voxel based methods

Another type of methods is voxel based methods. In these approaches, the point cloud is voxelized, meaning that the space occupied by the point cloud is discretized into 3D bins (voxels), and points are assigned an index indicating the voxel to which they belong (Hershey et al., 2022). In their paper, Hershey et al. (2022) proposed a method based on the voxelization of the point cloud. Firstly, they determined the optimal horizontal section of the point cloud to detect tree stems, which helps eliminate understory vegetation and tree canopy. This section was determined as the one between 2 and 7 meters above ground. The voxels in this section can then be used to identify vertical elements that may be associated with tree stems. From this voxelised layer, they analyze the columns of voxels to determine the locations of the stems. These columns consist of 10 voxels, each with a size of 0.5 meter in x, y and z axis. This is done by evaluating for each voxel column the point density and the point distribution. A column is considered a stem if it meets the following conditions:

- A column should at least contain twelve points (point density)
- A column should at least have eight of its voxels occupied by at least one point (point distribution)

A second pass was then performed for the layers between three to six meters above ground, with a threshold value of eleven points for point density and five voxels for point distribution. Only the columns detected by both passes were identified as tree stems. Finally, duplicate predictions are merged by applying a horizontal buffer to each detected stem. A union operation is applied to overlapping buffers and the centers of these polygons are considered as tree locations (Hershey et al., 2022). A method similar to this one was proposed by Kuželka et al. (2020), they proposed a method based on the creation of a raster with values computed from a voxelised version of the point cloud. Firstly, as Hershey et al. (2022), they keep only sub-canopy elements by filtering out points below 0.5 meters above ground or above 9.5 meters. Then it is voxelized and a 2D raster is built based on the horizontal grid of the voxelization. Each cell of this raster is attributed a value of stem presence indicator (SPI) computed with the following formula :

$$SPI_{i,j} = \sum_{k=1}^{m-1} \sum_{l=k+1}^{m} n_{i,j,k} * n_{i,j,l} \tag{3.2}$$

where

- i, j are the indices of the raster cells along the x and y axes.
- m is the number of voxels in vertical direction.
- $n_{i,j,k}$ is the number of points in the i,j,k-th voxel.

High values for this indicator indicate cells with a high point density from ground to summit. From this, an SPI_{max} raster is created. Each cell of this raster contains the maximum value within a determined radius around the corresponding cell in the SPI raster.Finally, trees are located as pixels in SPI raster where the value matches the value of the corresponding cell in SPI_{max} raster. Then, trees are delineated by partitioning the raster using a Voronoï diagram around the detected tree maxima

(Kuželka et al., 2020).

For these techniques, the voxel size needs to be set to ensure sufficient granularity while remaining larger than the typical stem size of the region. Additionally, the thresholds must be tuned according to the specific study case and the quality of the acquisition materials. These methods have limitations, particularly in identifying trees with irregular geometries, such as banded trees or multi-stem trees (Hershey et al., 2022).

3.1.3 Point cloud based methods

There are also algorithms directly based on the point cloud, such as the method presented by Li et al. (2012). They introduced a method similar to the region-growing technique mentioned earlier. The highest point in the point cloud is selected as the first tree. From this point, all other points will be classified as belonging to the tree if they follow these rules:

- The horizontal distance to the summit is below a given threshold.
- The point is closer to a point belonging to the current tree than to any classified point of another tree.

Once no more points are added to the tree, the process is repeated for a new tree, identified as the highest unclassified point. They also suggested adding additional rules to detect branches by computing the shape index of the 2D convex hulls of the trees (Li et al., 2012).

Another point cloud based method is the one presented by Tiede et al. (2005). They subsampled their point cloud using a 1 m $\times$ 1 m grid, and retained only the point with the highest z value for each cell. They used the following regression model to link crown width to summit height:

$$CD = a + b * TH \tag{3.3}$$

Where CD is the maximum crown distance, TH is the tree height, and a and b are parameters to be determined based on the type of forest studied. Based on this regression model they designed an algorithm that, for each point of the subsampled point cloud, evaluates a neighborhood defined by the CD computed using the regression model and the point's height. If the point is the maximum of this neighborhood it is considered as a tree top.Then they used these local maxima as seed points for a region-growing algorithm, as Solberg et al. (2006). They started with the lower seed point and end with the higher, to avoid assigning points of lower trees to higher ones. So a point in the neighborhood of a seed point is added to its region if the following criteria are met:

- It has a lower z value than the seed point.
- It is, at least, 5 meters above the ground.
- Its height is above 75% of the height of the seed point.
- Its distance to the tree top is below 10 meters.

These criteria are evaluated for every point in the neighborhood of a tree top until no more point is added to the region, then it passes to the next tree top until they are all evaluated (Tiede et al., 2005).

However, these methods still require prior knowledge of the forest type and structure to determine parameters such as the spacing between trees, the choice of shape rules, and the minimum height.
An hybrid approach was proposed by Jaskierniak et al. (2021). They first filter out the understory vegetation. Then, the point cloud is sliced horizontally into layers of 0.4 m of height. These slices are then transformed into rasters with cells size of 0.2 meters. The cell values represent the density of points within them. A watershed algorithm is then applied to this density raster to perform clustering. For each of these clusters, the cell of highest density is assumed as its center and its height is set as the mean height of the points in that cell. The cluster centers are then grouped across slices, using an iterative algorithm beginning with the lowest slice. Each cluster center in the lowest slice is assigned a group index. A cluster center in the slice above is linked to a cluster center in the slice below if it is within a horizontal Euclidean distance threshold of that center, it is then assigned the same group index, otherwise, it is given a new index. Cluster centers in a given slice can have multiple groups within the threshold distance, the closest one is retained. To compute this distance, two methods are possible. If the group architecture includes fewer than four slices below the considered slice, the higehst point's XY coordinates are used to compute this distance. If the architecture has more than four slices, principal component analysis (PCA) is applied to the points composing the groups to find the first principal component directional vector, the projection of this vector onto the considered slice is then used. Afterward, to limit computational complexity, the point cloud was voxelized. The voxels containing clusters from the previous part are clustered together if they are within a range of 0.4 m of each other. These clusters of voxels are then classified as stems or canopy. Finally the clusters of canopy are linked to clusters representing stems closest to their 3D bounding box. A major drawback of this algorithm is the high level of configuration needed, making it highly linked to its studied case (Jaskierniak et al., 2021).

As seen through the different methods presented in this section, a major drawback of the hypothesis-driven methods is the need for prior information on the forest type. This results in poor generalization of these models as a new parametrization is required whenever the forest type changes, and they often lack efficiency in heterogeneous forests.

3.2 Data-driven approach

To overcome these problems, a solution is the data-driven approach. For this approach, instead of defining the link between the input and the output, using series of parameters to fit the studied case, the goal is to design a method that will analyse the input data to find trends in it. It allows to find hidden relations in the data, by exploiting a large amount of input data (Demchenko et al., 2017). It is defined by Hong et al. (2023) as follows: "As a branch of computer science, data-driven technology is an intelligent system that can simulate human thinking, recognize complex situations, acquire learning ability and knowledge, and solve problems".

3.2.1 Machine learning

Machine learning is a part of computer science that uses algorithms to find trends in data instead of explicitly solving problems (Suriyan & Ramalingam, 2023). It can be divided into two main approaches: supervised learning and unsupervised learning. In supervised learning, the input data includes the desired labels, which are used to train and fit the model (e.g., random forest, k-nearest neighbors, regression). In unsupervised learning, the input data of the model (e.g., k-means clustering, principal component analysis) does not require labels. In tree detection the algorithms used are mainly unsupervised clustering techniques (Sun et al., 2024).

Gupta et al. (2010) presented a method that uses machine learning for tree detection.They employed the k-means clustering algorithm, which is an unsupervised machine learning algorithm aimed at creating k clusters. In this algorithm, the elements within a cluster are as similar as possible, while the differences between elements in different clusters are maximized (Rawal et al., 2021). It requires the selection of k seed points from the dataset. In their study, Gupta et al. (2010) tested two different methods for selecting these seed points: first, by selecting them randomly; second, by selecting the k local maxima. This k-means algorithm is trained by iteratively minimizing the sum of the Euclidean distance between each point and its closest cluster center. In the first case, where the seed points are chosen randomly, the model tends to converge to a local optimum. This issue is solved by using local maxima as seed points. They also tested hierarchical clustering, which is a clustering algorithm that starts with each point as its own cluster and merges clusters based on a similarity measure. However, this method didn't achieved satisfactory results. A major drawback of k-means clustering, highlighted in this work, is its dependence on the initial set of seed points. The quality of the final clusters depends on the initialization of the seed points (otherwise, there is a risk of converging to a local optimum), and the number of trees detected is directly related to the number of seed points provided (Gupta et al., 2010). Wallace, Lucieer, and Watson (2014) compared this method with rule-based methods similar to those presented above, and found that k-means clustering achieved higher accuracy.

Tan et al. (2023) proposed an algorithm for tree detection using the Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) method. This algorithm is an improvement of the DBSCAN algorithm. DBSCAN takes as input a radius and a density threshold, and clusters the points in the point cloud based on the density of points in their neighborhood defined by the radius. A cluster is formed if the density of points in the neighborhood is above the given threshold (Ferrara et al., 2018). The advantage of HDBSCAN is that it computes different clusters based on varying density thresholds, which are then organized hirarchically according to the size of the neighborhood radius, with the highest level of the hierarchy being the cluster containing all the points. Then the algorithm extracts the more stable clusters by defining a minimum number of points in a cluster. The algorithm then goes through the cluster hierarchy in decreasing order of cluster size. If a cluster, resulting from the division of a higher-order one, contains more points than the threshold, the higher-order cluster is dropped and the new one is considered. Otherwise, the parent cluster is retained (Campello et al., 2013; McInnes et al., 2016). After applying this, clusters are grouped based on the following criteria :

- They are less than 2 m apart.

- The area of the projection in the xy plane of their convex hull is lower than 1.5 m^2.
- Their geometric centers are parallel to the z-axis.

These conditions are checked until the closest clusters pair is separated by more than 2 meters. Finally, PCA is applied to these clusters to isolate points aligned with the z-axis, and clusters with a small number of points are excluded. Trunks are then fitted by lines based on principal component attributes (Tan et al., 2023). HDBSCAN can also be applied to forestry applications beyond tree detection. For instance, Neuville et al. (2021) used HDBSCAN for point classification at the tree level to detect tree stems, and subsequently applied PCA to these stems to compute the tree diameter at breast height (DBH).

A drawback of these techniques is that they still rely on input parameters specific to the case study (e.g., number of seed points, minimum distance between clusters, minimum density of points within a cluster). Another limitation is that they focus on only one aspect of the information provided by the point cloud, clustering pixels based solely on density or the mean distance to the cluster center.

3.2.2 Deep learning

A recent solution to overcome these problems is the use of deep learning. Deep learning is a subfield of machine learning that maps an input vector space to an output vector space by passing the input through multiple layers. These layers consist of multiple cells called neurons. The value of each neuron is obtained by performing a linear combination of the neurons (or a subset of neurons) from the previous layer, followed by the application of a non-linear function, known as the activation function. The first layer is the input layer and consist of the input to the model. The parameters of each layer are fitted using an optimization algorithm based on the performance of the model evaluated by a defined loss function. This loss function compares the results of the model to the expected values. The model is trained iteratively on the dataset, until the loss converges to a minimum value. To train such a model, large amount of data is required (Misra & Li, 2020). The aspect of deep learning used in this study is the object detection models. The goal of those models is to detect instances of objects of predefined classes in images, videos or point clouds (Zaidi et al., 2022). Firstly, these models were developed for 2D object detection in images or videos. Among the principal 2D object detection networks are RCNN, Mask-RCNN, Faster-RCNN and YOLO. For 3D object detection, the task is more complex due to the sparsity of the data (Wu et al., 2021). Some approaches use 2D data derived from point clouds (such as front view and bird's-eye view) and apply 2D object detection models, while others use voxelized point clouds, a major model developed for this approach is VoxelNet. The final approach directly takes the point cloud as input, with VoteNet being a promising model developed for this approach. VoteNet is a point-wise deep learning model for 3D object detection. It is divided into four modules: first, a sampling module that extracts points from the input and computes additional features for each point; next, a voting module (the core of the model) that generates votes for potential object centers based on each sampled point; then, a clustering module that groups the votes; finally, a regression module that computes the box dimensions for each cluster of votes (Qi et al., 2019).

Deep learning models for processing 3D point clouds were initially developed in the field of self-driving cars. However, they have also been applied to forest applications

in some research studies to process LiDAR point clouds, especially TLS, which benefits from a high point density on the stems compared to ALS (Windrim & Bryson, 2020).

A significant portion of this research focuses on 2D tree detection, which relies on the conversion of the point cloud into a 2D raster. Some approaches replicate the preprocessing steps of previous non-deep learning techniques by merging CHMs and RGB images, then applying deep learning object detection models to these rasters (Diez et al., 2021; Zhong et al., 2024). These studies apply existing object detection models, such as the YOLO model or the Mask R-CNN. Diez et al. (2021) emphasize in their review that these models have achieved encouraging results but still have room for improvement. One key factor for improvement is increasing the amount of available data to train complex networks such as Mask R-CNN, as publicly available labeled data are currently limited (Diez et al., 2021). However, these solutions do not take into account the vertical structure of the forest. Windrim and Bryson (2020) proposed a workflow to account for this verticality. They apply a bird's eye view (BEV) projection to the point cloud (i.e., projection onto the xy-plane). This process creates 3-channel images based on intensity, where the intensity of each pixel represents the point density of the point cloud along the vertical axis for the 2D ground surface of that pixel. These images are then used as input for a Faster R-CNN model to perform 2D object detection (Windrim & Bryson, 2020). Ying et al. (2021) developed a method that complements the local maxima detection method. They first apply local maxima detection to the CHM, as described earlier. Then, they extract boxes of various predefined sizes centered on these detected local maxima from the point cloud to serve as tree samples. These samples are then passed through a 3D-CNN model to classify whether or not they represent a single tree. Finally, the seed points filtered by the 3D-CNN are evaluated using a rule-based algorithm to determine if they belong to the same tree. This workflow enhance the versatility of the method in complex tree context and reduces omissions (Ying et al., 2021).

There are few studies that have applied object detection models directly to the point cloud for individual tree detection (ITD) rather than on a feature map of the point cloud. Schmohl et al. (2022) presented a workflow for ITD in an urban context using deep learning. They developed a three-stage model. The first stage is a U-Net-shaped Sparse Convolutional Network that takes the voxelized point cloud as input and extracts useful features from the point cloud. The output of this first layer is the voxelized point cloud, with each voxel having 32 features computed by the previous block. This output is then passed in parallel through two different blocks. One is a semantic segmentation head, which computes the class for each voxel. The other is a detection head, which computes the bounding box of each tree.They achieve an average precision of 83% for ITD (Schmohl et al., 2022).

Chapter 4

Research question

Through these different sections, an overview of the use of LiDAR remote sensing for forest inventories has been provided, including the various methods applied and their limitations. It has been shown that a promising solution is the use of recently developed deep learning models for object detection. Furthermore, to the best of our knowledge, no study has yet applied 3D object detection models to aerial point clouds in mixed forest context.

Thus, the research objective of this work is to conduct a preliminary investigation of object detection on aerial point clouds in a mixed forest context and assess their potential for forest inventories. The goal is to answer the following research question:

"Is point-wise deep learning approach a suitable solution for tree detection purposes in mixed forest stands? "

In this research project, the attention is focused on the application of VoteNet model in a Belgian mixed forest stand.

To answer this question, the following steps has to be taken:

- Acquisition of a UAV point cloud dataset of mixed forests, along with the design of a preprocessing and labeling methodology.
- Implementation of the model algorithm.
- Training and testing on the model using the acquired dataset.
- Evaluation of the quality of the objects detected by the model.

Chapter 5

Methodology

This section explains the approach used in this paper. It provides details about the data capture, the preprocessing stages, the VoteNet architecture, its implementation in TorchPoints3D, and the comprehensive methodology for training and evaluating the model performance.

5.1 Data capture

5.1.1 Studied area

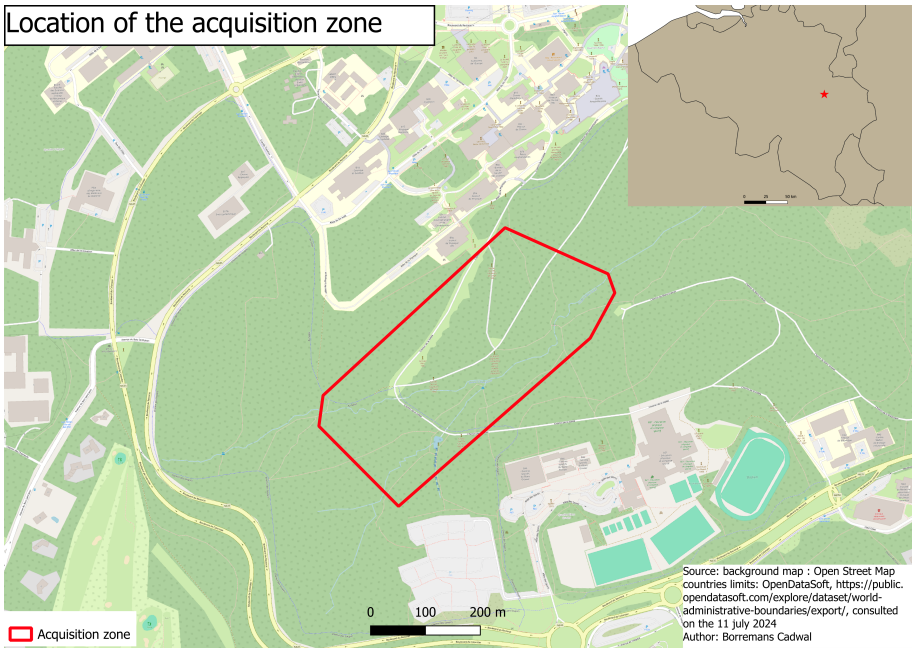
The studied area is located in the Sart-Tilman woods in the city of Liège in Belgium (see Figure 5.1). This area covers 13.15 hectares and is characterized by a dense forest of heterogeneous trees, with a road running through its center.

5.1.2 Acquisition device and protocol

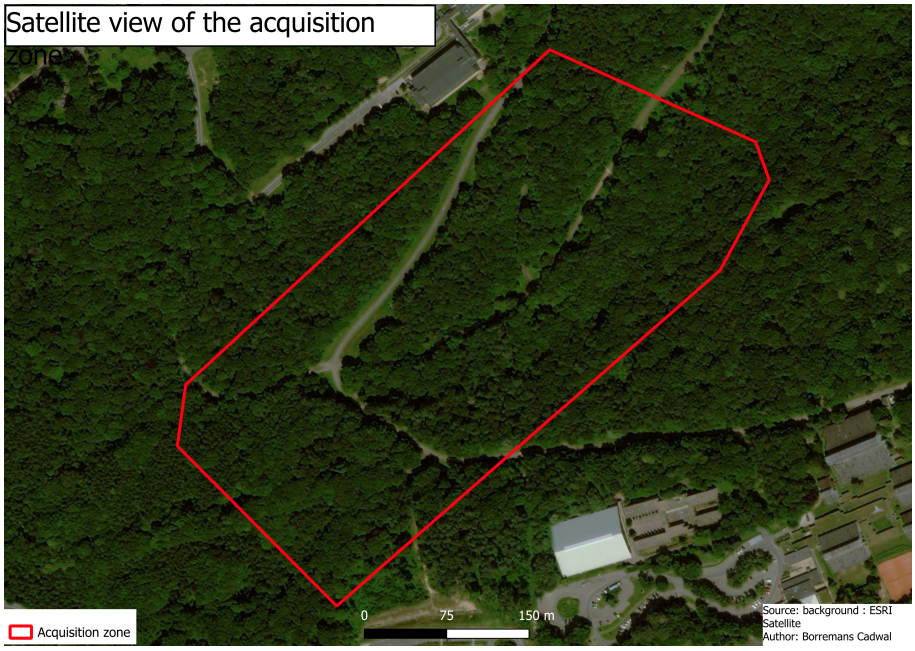
The point cloud was acquired using a Zenmuse L1 sensor (see Figure 5.2a), produced by DJI. It combines a LiDAR sensor and a RGB camera, resulting in a fully colorized point cloud. It uses rotating mirrors to control the orientation of the laser beams (Coptrz, 2021; DJI, n.d.-b). The announced characteristics of the sensor are as follows (DJI, n.d.-b):

- Pulse repetition frequency: 240 000 points/seconds
- Maximum number of returns: 3
- Ranging accuracy : 3 centimeters at a distance of 100 meters (computed using RMS with 1 σ)
- FOV: 70.4° (horizontal) $\times$ 77.2° (vertical)

This sensor was mounted on a Matrice 300 RTK drone (see Figure 5.2b), also produced by DJI. The Matrice 300 RTK can carry various payloads, with a maximum capacity of 3 payloads and up to 2.7 kg, and it provides RTK positioning for enhanced accuracy. The acquisition took place on April 12, 2024, with a flying height of 35 meters above the ground. A complete list of the flight parameters can be found in Appendix A.2.



(a)



(b)

Figure 5.1: Localization (a) and satellite view (b) of the acquisition zone

(a) Zenmuse L1 (DJI, n.d.-b). (b) Matrice 300 RTK (DJI, n.d.-a).

Figure 5.2: Sensor (a) and UAV (b) used for the data acquisition

5.1.3 Preprocessing

The acquired data was processed using DJI Terra, the 3D reconstruction software developed by DJI. Among the various features offered by this software, the ground point classification option was applied. This option adds a new attribute to the points in the output point cloud, containing a binary value that indicates whether each point is classified as ground or not. Based on this classification, the ground points can be removed from the point cloud. This helps reduce the size of the point cloud, thereby accelerating subsequent processing tasks and enhancing the detection of individual trees (Windrim & Bryson, 2019). As a result, we obtained a

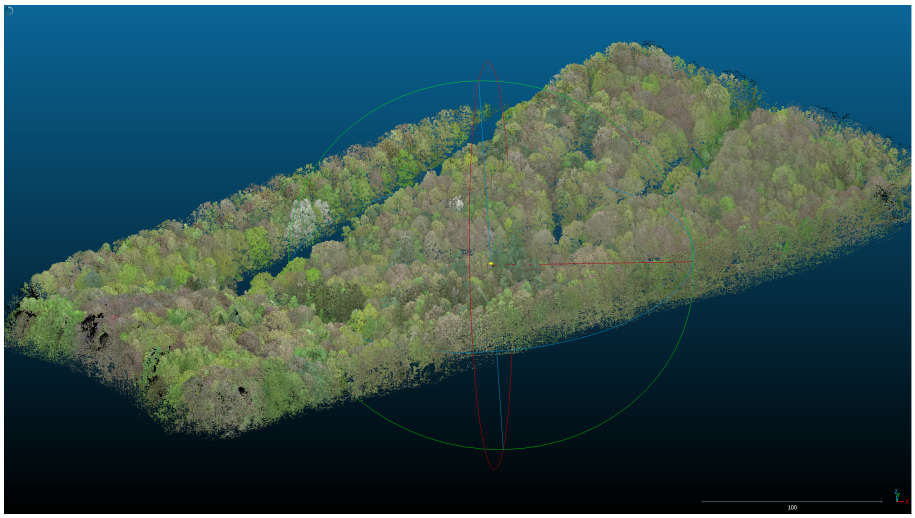


Figure 5.3: Colorized point cloud acquired in the woods of the Sart-Tilman, displayed in CloudCompare.

georeferenced, colorized point cloud of the studied area with ground points removed (see Figure 5.3).

It is important to note that, for subsequent use by VoteNet, the original point cloud required spatial sub-sampling. A regular grid with a 30-meter cell resolution was employed for this purpose. The chosen cell size was selected to ensure full coverage

of trees with more than ten trees per cells. For the rest of this paper, these sub-samples will be referred to as "scenes". This division was performed using a Python script (see Appendix A.4), and resulted in a total of 162 scenes.

5.1.4 Labelling

To train an object detection model, the dataset needed to be labeled with ground truth bounding boxes delimiting the trees. No suitable free software was found for labeling bounding boxes directly on the point cloud. Therefore, the "segment" function in CloudCompare was used to create them. This tool allows for manual cutting and extraction of parts of a point cloud. Each scene was stored in a separate folder, and using the segment tool, each tree in every scene was cut out and saved as a separate point cloud file within the corresponding scene's directory (see Figure 5.4). Subsequently, a Python script was created (see Appendix A.5) to process each scene folder and generate a JSON file containing the bounding box parameters for each extracted tree. The bounding boxes were defined by their center coordinates, width, height, and length.

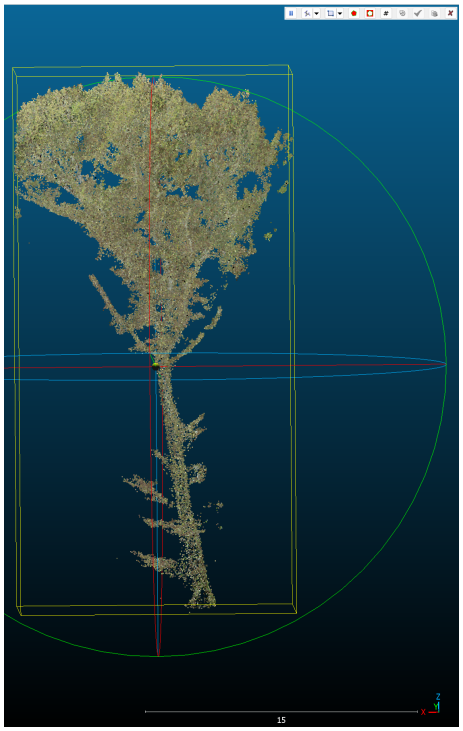


Figure 5.4: Segmented tree in cloud compare

However, this methodology is not optimal, and is time consuming. As a result, only 30 scenes could be labelled in the scope of this work.

5.2 Pointwise Deep Learning-based 3D Object Detection Algorithm: VoteNet

The chosen model is VoteNet (see Figure 5.5), a 3D object detection model developed by Qi et al. (2019). The advantage of this model, and the reason for its selection, is its ability to perform detection directly on the point cloud, without requiring extensive data preprocessing, transformations, or projections, unlike models such as MV3D and VoxelNet (Qi et al., 2019).

VoteNet processes raw point clouds directly. Its approach involves first extracting a sub-sample of points (seed points) from the input point cloud and computing additional features for each seed point. Each of these seed points then generates votes for the center of the potential object to which it belongs. Subsequently, these votes are clustered, and object proposals are generated from the resulting clusters.

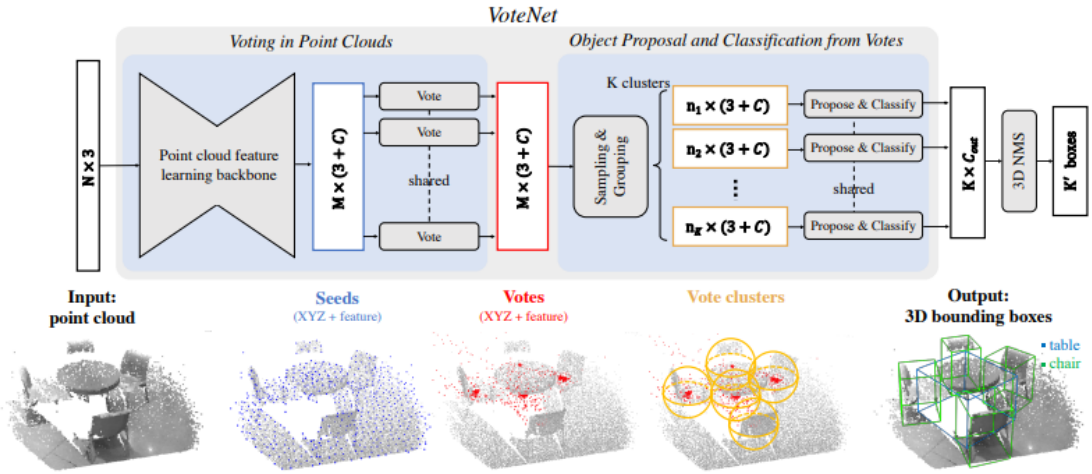


Figure 5.5: VoteNet model architecture (Qi et al., 2019).

5.2.1 Architecture

Seed points extraction

The extraction of seed points and the computation of their additional features are performed using a backbone model. In this case, the PointNet++ model is employed, which is designed for point cloud learning. PointNet++ consists of four set abstraction (SA) layers, responsible for subsampling the point cloud to a subset of 256 points. This is followed by two feature propagation (FP) layers, which upsample the output of the last SA layer to 1024 points, each with 256-dimensional features (plus their 3D coordinates).

Vote generation

Each of these seed points generates a vote for the object center it belongs to. This is achieved using a MLP with three convolutional layers, each with a kernel size of 1, a ReLU activation function, and output dimensions of 256, 256, and 259 (for features plus coordinates). Batch normalization is applied to each layer. The final layer outputs the difference between the features of each vote and its corresponding seed point, as well as the difference between the vote's coordinates and the seed point's coordinates.

Votes clustering

Afterward, the votes are grouped into clusters. To perform this clustering, a subset of K votes is initially selected using farthest point sampling. This algorithm chooses K farthest points from the point cloud by first selecting one point randomly and then iteratively selecting the point that is farthest from all previously selected points (Li et al., 2022). Clusters of votes are then formed around these K selected points using the following formula:

$$C_k = \{v_i^{(k)} \mid ||v_i - v_{ik}|| \leq r\} \quad for \quad k = 1, ..., K \tag{5.1}$$

where

- C_k is a cluster formed around the k-th vote of the sub-sample drawn earlier.
- $v_i^{(k)}$ is the i-th vote belonging to the cluster k.

- v_{ik} is the vote around which the clusters are made.
- r is a given radius.

Object proposal

After that, a box is predicted for each of these clusters. This is done by passing the votes, essentially a set of high-dimensional points, through a point set learning network to create an object proposal. This proposal contains an objectness score, bounding box parameters (center coordinates, dimensions, heading), and a semantic classification score. This model consists of two MLPs. The votes of each cluster are first passed through the first MLP independently. This first MLP is a SA layer with leaky ReLU activation with a batch normalization. It is responsible for extracting further information from each points in this cluster, based on their surrounding points. And they are combined using a max pooling operation for each feature, resulting in a single features vector of the same size as the number of features of the votes plus 3 (x, y, z coordinates) for each cluster. These vectors are then passed through a second MLP composed of three 1 dimensional convolutional layers, with batch normalization and ReLU activation. This final MLP produces the object proposal.

5.2.2 Outputs

The outputs of the model consists of the following predicted information:

- Objectness score: A tensor of size B $\times$ M $\times$ 2, where B is the batch size and M is the maximum number of objects. For each predicted object in each element of the batch, it contains the probability that it is not an object (dimension 1) belonging to a desired class and the probability that it is an object (dimension 2) belonging to a desired class.
- Center coordinates: A tensor of size B $\times$ M $\times$ 3, which contains the center coordinates of the predicted object.
- Heading score: A tensor of size B $\times$ M $\times$ NH, where NH is the number of heading bins (classes). It is used if the dataset divides data into classes of similar headings, each with a defined mean heading. This tensor contains, for each predicted object, the probability that it belongs to each heading class.
- Heading residuals: A tensor of size B $\times$ M $\times$ NH, which contains the difference between the predicted box headings and the mean heading of each heading class.
- Size score: A tensor of size B $\times$ M $\times$ NS, where NS is the number of size bins (classes). It is used if the dataset divides data into classes of similar sizes (this separation can be based on semantic classes), each with a defined mean box size. This tensor contains, for each predicted object, the probability that it belongs to each size class.
- Size residuals: A tensor of size B $\times$ M $\times$ NS $\times$ 3, which contains the difference between the predicted box dimensions and the mean size of each class.
- Semantic class: A tensor of size B $\times$ M $\times$ 1, which contains the predicted semantic class of each object.

Based on these results, it is possible to reconstruct the box of the detected objects using the predicted object center, the mean size of their highest probable size class, the associated size residuals, the mean heading of their highest probable heading class and the associated heading residuals. While retaining only objects with sufficiently high objectness values.

5.2.3 Loss function

The model evaluation and updating is driven by the following loss function :

$$L = L_{vote-reg} + \lambda_1 L_{obj-cls} + \lambda_2 L_{box} + \lambda_3 L_{sem-cls} \tag{5.2}$$

where

- $L_{vote-reg}$ evaluates the quality of the votes made by the seed points. This is done by evaluating the mean difference between the distance from a seed point to the vote point and the ground truth distance from the seed point to the object center. It is computed as follows:

$$L_{vote-reg} = \frac{1}{M_{pos}} \sum_i ||\Delta x_i - \Delta x_i^*|| \mathbb{1}[s_i \text{ on object}] \tag{5.3}$$

  where

  - $\mathbb{1}[s_i \text{ on object}]$ is equal to one if the seed point is on an object, zero otherwise.
  - M_{pos} is the total number of seed points on the object surface.
  - Δx_i is the computed 3D distance between the seed point and the object center.
  - Δx_i^* is the ground truth distance between the seed point and the object center.

- $L_{obj-cls}$ is the objectness loss, which evaluates the predicted objectness scores for object centers based on their distance to the ground truth object center. The predicted object centers are classified as either "near" (below a manually determined "near" threshold) or "far" (above a manually determined "far" threshold). This loss is a CE loss (see Section 2.2.3). More precisely, a binary cross-entropy loss (BCE). In case of a binary classification, where the class labels become 0 and 1 (in this case 0 = "does not belong to an object" and 1 = "belong to an object"), the equation of the CE loss becomes:

$$BCE = -y log(\hat{y}) + (1 - y) log(1 - \hat{y}) \tag{5.4}$$

  where

  - y is the ground truth label of the evaluated element.
  - $\hat{y}$ is the predicted probability that the element belongs to the class 1.

  Thus the objectness loss is computed as the mean BCE loss for the votes below a given near threshold or above a given far threshold.

- L_{box} is the box loss, which measures the accuracy of the predicted bounding boxes for the objects. It is computed as follows:

$$L_{box} = L_{center-reg} + 0.1 L_{angle-cls} + L_{angle-reg} + 0.1 L_{size-cls} + L_{size-reg} \tag{5.5}$$

  where

– $L_{center-reg}$ evaluates the predicted center coordinates by computing the sum of two distances: the mean distance between the detected object centers and their nearest ground truth object centers, and the mean distance between the ground truth object centers and their nearest detected object centers.

– $L_{angle-cls}$ evaluates the predicted heading class by applying the CE loss.

– $L_{angle-reg}$ evaluates the heading residuals predicted by applying the Huber loss. The Huber loss is a regression loss function that combines quadratic loss and absolute error loss, making it less sensitive to outliers (Girshick, 2015). It is defined as follows:

$$L_{huber} = \begin{cases} \frac{1}{2}|x|^2 & \text{if |x| } \leq \text{ d} \\ \frac{1}{2}d^2 + d * (|x| - d) & \text{if |x| } > \text{ d} \end{cases} \tag{5.6}$$

where

- x is the difference between the predicted heading residual and the ground truth heading residual.
- d is a constant of the model, set to 1.

Thus the angle regression loss, is the mean of the Huber loss computed for each object.

– $L_{size-cls}$ evaluates the predicted size class by applying the CE loss.

– $L_{size-reg}$ evaluates the size residuals in the same way as the heading regression loss.

– $L_{sem-cls}$ is the semantic class loss, which evaluates the semantic class assigned to each object. This loss is computed using CE loss. In this case, since there are only two classes (tree and non-tree), it simplifies to BCE loss.

– λ_i with i = 1,2 or 3, are model-defined constants, set to 0.5, 1, and 0.1, respectively.

5.3 Deep Learning framework for Point Cloud: TorchPoints 3D

The solution to implement, train, and assess VoteNet is the open-source Python framework "TorchPoints3D". It was developed to offer a modularly designed framework that eases the use and development of 3D deep learning models. It includes pre-implemented models for various deep learning tasks, such as classification, segmentation, object detection, panoptic segmentation, and registration. Additionally, it includes public datasets to train these models (Chaton et al., 2020).

5.3.1 Installation

This framework relies on several Python libraries that need to be installed beforehand. The main ones include:

- PyTorch: A Python library designed for deep learning on GPUs and CPUs (PyTorch, n.d.-a).
- OmegaConf: A hierarchical configuration system based on YAML files. YAML stands for 'YAML Ain't Markup Language,' is a data serialization language used for creating configuration files (döt Net et al., 2021). It can take information from different sources (e.g. YAML files, command-line arguments, environment variable) (Yadan, 2022).
- Hydra: A Python framework that facilitates dynamic creation of hierarchical configurations (Meta Platforms, n.d.).

The installation process was conducted using an Ubuntu interface set up on Windows through the second version of Windows Subsystem for Linux (WSL2). This setup streamlined the installation of various Python libraries and the cloning of GitHub repositories, which were managed through command-line instructions. However, due to the specific versions of libraries required and potential compatibility issues, the installation process may encounter challenges and can be time-consuming. Therefore, detailed installation steps are provided in Appendix A.3.

5.3.2 Creation of the dataset

An important aspect of this work was designing a dataset class to handle the raw acquired data. This class needs to be able to preprocess the data to fit the labels and format required by the tested model, and pass it through the model.

Among the various datasets available in TorchPoints3D, one suitable dataset for object detection is the ScanNet dataset. This open-source dataset contains over 2.5 million views from more than 1500 scans of indoor environments. Each scan is annotated with instance-level semantic segmentation, from which object detection labels can be derived (Dai et al., 2017). This dataset served as an example for the construction of the new dataset class used in this work.

Datasets used in this framework should inherit from the "InMemoryDataset" class of PyTorch Geometric, which is designed to create graph datasets that can be loaded into CPU memory (PyTorch, n.d.-b).

To pass a dataset to a model, it must contain the labels and parameters required by the model, in the correct formats. For the VoteNet model, the dataset must include the parameter "position", which is a tensor [1] of size n $\times$ 3, with n the number of points in the point cloud. This tensor contains the X,Y and Z coordinates of each point of the point cloud in float data format. Additionally, the dataset must provide the following labels:

- Center label: A tensor of size m $\times$ 3, where m is the maximum number of objects in a point cloud of the dataset (defined by the user at the creation of the dataset). This tensor contains the X,Y and Z coordinates of the center of each object's bounding box in the point cloud. These coordinates should be encoded as floats. If there are fewer than m objects in the point cloud, the remaining rows of the tensor should contain null coordinates.
- Heading class label: A tensor of size m $\times$ 1, which contains the heading class of each object. It can be used to classify objects based on their headings. In

[1] it is "a multi-dimensional matrix containing elements of a single data type" (PyTorch, n.d.-c).

this case, the whole vector is set to zero, assuming that all bounding boxes are oriented in the same direction.

- Semantic class label: A tensor of size m $\times$ 1 that contains the semantic class of each object in the point cloud. These classes are integer values stored as int.
- Box label mask: A tensor of size m $\times$ 1, which is a boolean tensor indicating whether each object has (1) or does not have (0) a class label handled by the dataset.
- Vote label: A tensor of size n $\times$ 9, where n is the number of points in the point cloud. This tensor contains, for each point, the difference between its coordinates and those of the center of the bounding box containing it. Since a single point can belong to multiple bounding boxes, this tensor can have up to three different votes. For points that only belong to one box, the same vote is repeated three times.
- Vote label mask: A tensor of size n $\times$ 1 that is a boolean tensor indicating whether each point belongs (1) or does not belong (0) to an object. This tensor is used to determine if a vote was generated from a seed point that belongs to an object or not.
- Heading residual label: A tensor of size m $\times$ 1, which contains, for each object, the difference between its heading and the mean heading of its heading class.
- Size class label: A tensor of size m $\times$ 1 containing the size class for each object. This label is used to group objects based on their size, with similar-sized objects being assigned the same class. The mean size of each group is then used to compute the bounding box size. In the ScanNet dataset, size classes were set to match semantic classes. In this case, since there is only one semantic class, 'tree', only one size class and one mean size are defined.
- Size residual label: A tensor of size m $\times$ 1, which contains, for each object, the difference between its size and the mean size of its size class.

To construct a dataset, one must define the directory structure, the format of data files, and the labels to be extracted. For this work, these were defined as follows:

- Directory structure: Data is stored in the "data" directory of the working environment. Inside this directory, a sub-directory is created named after the dataset. Within this dataset directory, there are two sub-directories:
  - The 'raw' directory, where point clouds are stored in 'las' format along with their associated bounding boxes in 'json' format. Each point clouds and its associated bounding boxes have their own directory named 'scene_X', where X ranges from 01 to the total number of point clouds.
  - The 'process' directory, where the data is stored after the dataset has been initialized.
- Data files formats: The point clouds are stored in 'las' format, while the bounding boxes are stored in 'json' format. Each 'json' file contains the bounding boxes for a single scene.

- Labels: The labels required depend on the task realised and the model used. For the VoteNet model, the required labels are the one described earlier in this subsection.

After defining these, a Python code file needs to be created and stored in the appropriate folder for the task (in this case, object detection) within the dataset directory of the TorchPoints3D library. This code file should define two classes:

- The dataset class: Manages the training, validation, and test datasets.
- The data class: Handles the raw data processing for all three datasets (training, validation, and test).

The first class needs to define three functions:

- __init__: Initializes the training, validation, and test datasets.
- mean_size_array: Returns an array of size NC $\times$ 3, where NC is the number of size classes. This array contains the mean bounding box size for each size class.
- get_tracker: Defines the metrics to be tracked during the model's training.

The second class needs to inherit from the InMemoryDataset class of PyTorchGeometric and to define the following functions:

- __init__: The initialization function called during dataset creation. It takes as input the root directory where the data is stored, the transformations to apply to the data and whether the dataset will be used for training, validation or testing. It calls the initialization of the InMemoryDataset class, which call the process function defined in this dataset. Finally, it loads the result of the process function into a class variable named 'data'.
- process: A processing function called during initialization. It reads the raw data, extracts the desired information and stores it in a '.pt' file in the "process" directory. It also creates a '.npy' file containing the mean sizes of the different size class bounding boxes
- _getitem_: Returns an item from the dataset, including a point cloud and its labels.
- raw_file_names: Returns the folders containing the raw files.
- processed_file_names: Checks if the processed files exist. This function is called during initialization to determine whether the process function needs to be called.

The code for the dataset used in this work can be found in Appendix A.6.

A configuration file must also be created, detailing general information about the dataset (e.g., name, task, and root directory for data storage). This file also defines pre-transformations to be applied to the data as well as specific transformations for the training, testing, and validation datasets. It facilitates the configuration of data augmentation through transformation parameters, such as rotation and random noise. This file should be a YAML file, stored in the 'conf' directory of the working environment. The one created for this dataset can be found in Appendix A.7.

5.4 Model training

To train and evaluate models in deep learning the dataset is usually randomly split in three parts :

- Training: The part of the dataset on which the model is trained, i.e., used to update the weights of the dataset to minimize the loss function. It constitutes 70% of the data.
- Validation: The part of the data on which the model is not trained. At each training epoch the loss is computed on this dataset to evaluate the model on data it has not seen, but the gradients are not computed for this loss. It is useful to find the best configuration of the model, and ensures it is not over-fitting the training data. It constitutes 20% of the data.
- Testing: The part of the data used to evaluate the performance of the model. This dataset is not seen by the model at any training epoch. It is passed through the trained model, to evaluate its real performance on data it has not encountered during its training period. It constitutes 10% of the dataset.

However, in this specific case, due to the limited availability of labeled data, tests were also conducted using all but one scene for training, with the remaining scene was used for validation.

To monitor the model's performance throughout training and across various experiments, the Weights & Biases platform (WandB) was utilized. WandB is a machine learning tool designed to track and analyze experimental results. It integrates seamlessly with Python scripts via the "wandb" library, which facilitates connecting a Python script to the platform. This allows for real-time tracking of the model's training progress and storage of results in a WandB account. Additionally, it provides graphical representations of chosen metrics and enables comparison across different experiments (see Figure 5.6).

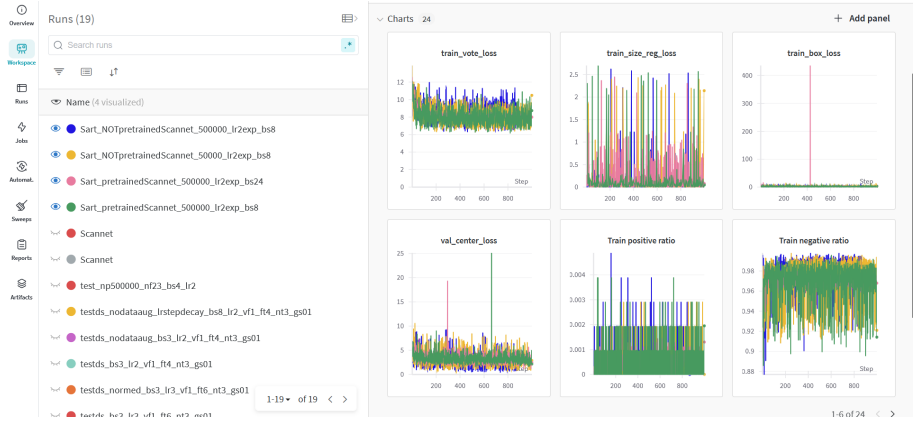


Figure 5.6: Screenshot of WandB interface

The different metrics tracked during the training of the model are as follows:

- Train losses: The various losses explained in Section 5.2.3, computed on the training set at each epoch.

- Validation losses: The different losses computed on the validation set at each epoch.
- Positive ratio: The proportion of predicted boxes whose centers are within a distance to a ground truth center that is below the near threshold.

Multiple parameters of the model can be tuned to find its optimal configuration. These parameters can be categorized into the following classes:

- Data processing: The different preprocessing applied to the data before it is passed through the model. Key aspects include:
  - Number of points: The input layer of the model requires a fixed number of points. Consequently, each sample must be subsampled or upsampled to this fixed number. This involves making a trade-off between the computational complexity and the data degradation.
  - Data augmentation: Involves applying random transformations to the data to artificially increase its diversity.
- Model architecture: The number of neurons per layers and the number of layers of the model can be adjusted to increase the capacity of the model. While this has already been done at the creation of the model by Qi et al. (2019), further modification could be made to enhance its performance on the dataset. However, addressing this aspect is out of the scope of this research project.
- Training configuration: Includes parameters that control the training process:
  - Batch size: The number of samples processed by the model at each iteration (each update).
  - Learning rate: The magnitude of the updates applied at each iteration.
  - Optimization algorithm: The method used to adjust the model's parameters during training.
  - Weights initialization: Refers to how the model's weights are initialized, which can be random, using a specific algorithm, or by using weights from a pretrained model on another dataset.

5.5 Evaluation Methods

To evaluate the quality of the model, standard deep learning metrics (Wu et al., 2021) can be used. To assess the quality of the results, the first step is to define what constitutes correct and incorrect detections. This is typically done using the IoU metric (see Figure 2.13).

Based on the IoU values, a threshold can be established to determine whether a detected box corresponds to a ground truth object. The same thresholds as the ones set by Qi et al. (2019) were used: 0.25 and 0.5. A detected box is considered a false positive (FP) if its IoU with the closest ground truth box is below the threshold, and a true positive (TP) if the IoU is above the threshold.

Using these thresholds, the following metrics can be computed:

- Recall (R): Measures the proportion of ground truth trees correctly identified by the model. It is computed as follows:

$$R = \frac{TP}{N_{AllGroundTruth}} \tag{5.7}$$

  Where $N_{AllGroundTruth}$ is the number of ground truth trees in the scene.

- Precision (P): Indicates the proportion of correctly detected trees among all detected trees. It is calculated as:

$$P = \frac{TP}{N_{AllDetection}} \tag{5.8}$$

  where $N_{AllDetection}$ denotes the total number of trees detected by the model in the scene.

- Mean average precision (mAP): Reflects the mean precision across all classes of the model. In this scenario, with only one class, the mAP is equivalent to the average precision (AP). The AP is determined by first sorting the predicted boxes in descending order of their confidence scores. Next, a precision-recall curve is plotted based on these predictions. The AP is then computed as the area under this curve (AUC).

5.6 Experiments

The initial objective is to train the model using the default parameters. If the results are not satisfactory, adjustments should be made to optimize the model's performance. The aim is to achieve the lowest possible loss value while maximizing key metrics such as recall, precision, and mAP.

The first test was done using the default parameters of the model already defined in the framework for the ScanNet dataset. The parameters were the following:

- Data augmentation: The following transformations are applied:
  - Random symmetry along the x axis: This technique randomly applies symmetry along the x-axis to the data samples.
  - Random scale anisotropic: This technique applies a random scaling transformation to the input data. The scale factor is randomly chosen between 0.9 and 1.1.
  - Random axis rotation: This technique randomly applies a rotation of a given angle around an axis. The angles in degrees for the x, y, and z axes are 2, 2, and 180, respectively.
- Number of points: 50 000
- Batch size: 8
- Learning rate: 10^{-2}
- Weights initialization: Random

After the first inconclusive tests conducted with this parametrization it was also decided to use only one scene as the validation sample and the remaining scenes as training samples. This was done in order to compensate the lack of data. However, this introduces a bias in the results, likely leading to an overestimation of the evaluation metrics. Therefore, these values should be viewed as optimistic indicators of the model's potential performance rather than an accurate assessment.

After the initial testing phase, various experiments were designed to analyze the impact of key parameters on the model's performance and to identify the optimal configuration. Given that it was not possible to evaluate all possible parameter configurations, specific parameters were selected for modifications while others were set to their fixed values. The following parameters were tested:

- Number of points: This parameter was selected to determine whether the value used for ScanNet is sufficient or if it reduces the amount of information passed to the model too much. Increasing the number of points enhances the information available but also increases the computational and storage complexity. A maximum value of 500 000 points was tested.
- Batch size: Increasing the batch size raises computational complexity but can reduce the variance in the model's updates, potentially leading to more stable learning. A higher batch size of 24 was tested to assess whether it could help the model better estimate the underlying function it attempts to learn.
- Weights initialization: This parameter influences the model's convergence and its speed and can be especially beneficial when labeled data is limited. By initializing the model with weights pre-trained on the ScanNet dataset, it can leverage prior knowledge gained from performing a similar task on a different dataset. This approach allows the model to make better use of the limited labeled data by building on the experience acquired from the pre-trained model.

The other parameters were configured as follows:

- Data augmentation: Given the limited amount of labeled data, the data augmentation techniques implemented for the ScanNet dataset were utilized to enhance the training set.
- Learning rate: Since the magnitude of the modifications applied to the model parameters at each step is already optimized by the chosen optimization algorithm, it was hypothesized that the learning rate could be fixed. Therefore, it was set to 10^{-2}.
- Optimization algorithm: The Adam algorithm was selected for this parameter, as it is widely regarded as one of the most effective optimization algorithms in the literature.

The number of epochs for the tests was set to 1000, to determine whether the model's losses had already converged. If not, the training could be resumed using the best model weights obtained from the previous training. After each experiment, the hyperparameter value that achieved the best result was retained for subsequent experiments. Finally, the combination of hyperparameters that yielded the best results was evaluated on the validation dataset, following the evaluation methodology presented in Section 5.5.

Chapter 6

Results

In this chapter, the training losses resulting from the main parameter modifications are presented and compared. The analysis begins with the test results using the basic configuration, derived from the one used for ScanNet dataset, including an examination of the various loss functions and the positive ratio. Following this, different hyperparameter modifications are introduced. For each hyperparameter, the value achieving the best result is retained for subsequent tests. The evolution of all losses from the various experiments for both the training and validation datasets can be found in Appendix A.8. Finally, the outputs from the model configuration that achieved the best results are presented and evaluated.

6.1 Basic configuration

The basic configuration, derived from the one used for the ScaNet dataset, has the following parameters:

- Number of points: 50 000
- Batch size: 8
- Learning rate: 10^{-2}
- Weights initialization: random
- Optimization algorithm: Adam

For the training losses (see Figure 6.1), it can be observed that most of the loss functions decrease during the initial epochs and then tend to converge relatively quickly toward high values.

The general loss converges to a value of 10, which is considered high for a loss function. Typically, a well-converged loss should be as close to 0 as possible, ideally below 1. For instance, the minimum loss obtained for ScanNet dataset during our tests was 0.4.

The vote loss, which measures the error in the votes cast by the seed points, converges toward 8 meters. This indicates that the mean difference between the distance of a seed point to the ground truth center of the object it belongs to and the distance of that seed point to a predicted box center is 8 meters.

The objectness loss diverges from this pattern, directly converging toward 0.

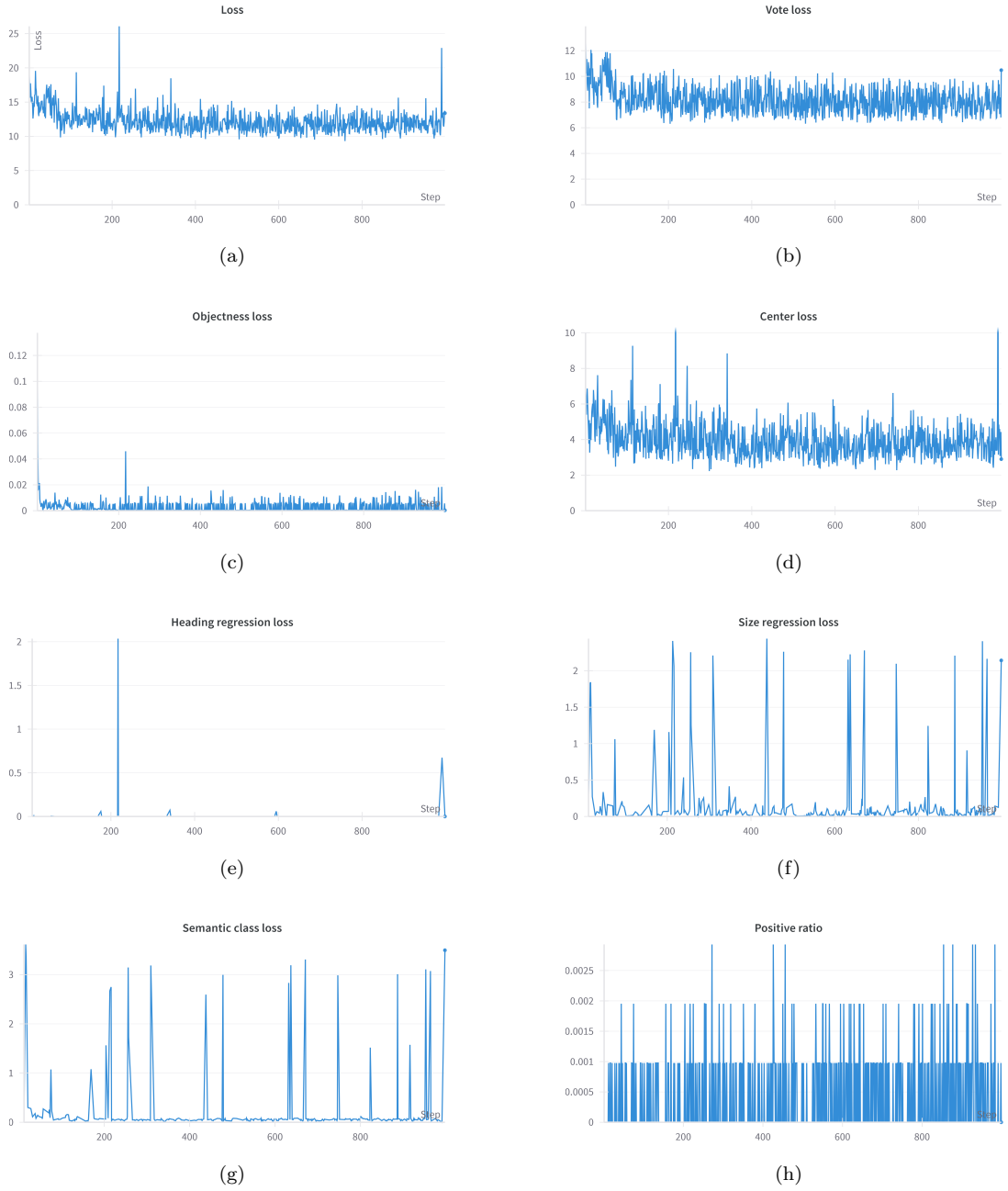


Figure 6.1: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed over the training dataset

The center loss, quantifying the distance between ground truth object centers and predicted centers, fluctuates between epochs but generally tends toward 3 meters.

The heading regression loss converges directly to 0, with occasional high peaks.

The size regression loss, which evaluates the discrepancy between the predicted box size and the ground truth box size, oscillates near zero with occasional peaks reaching up to 2.5 meters.

The semantic class loss also tends toward zero but exhibits similar high peaks as the size regression loss.

Finally, the positive ratio, which represents the proportion of predicted boxes with centers within the near threshold, remains below 0.4% throughout the training process. This implies that no predicted box has a center within the near threshold, defined at 0.3 meters.

For the validation loss (see Figure 6.2), the observations are similar to those for the training losses, but with greater variability from one epoch to the next.

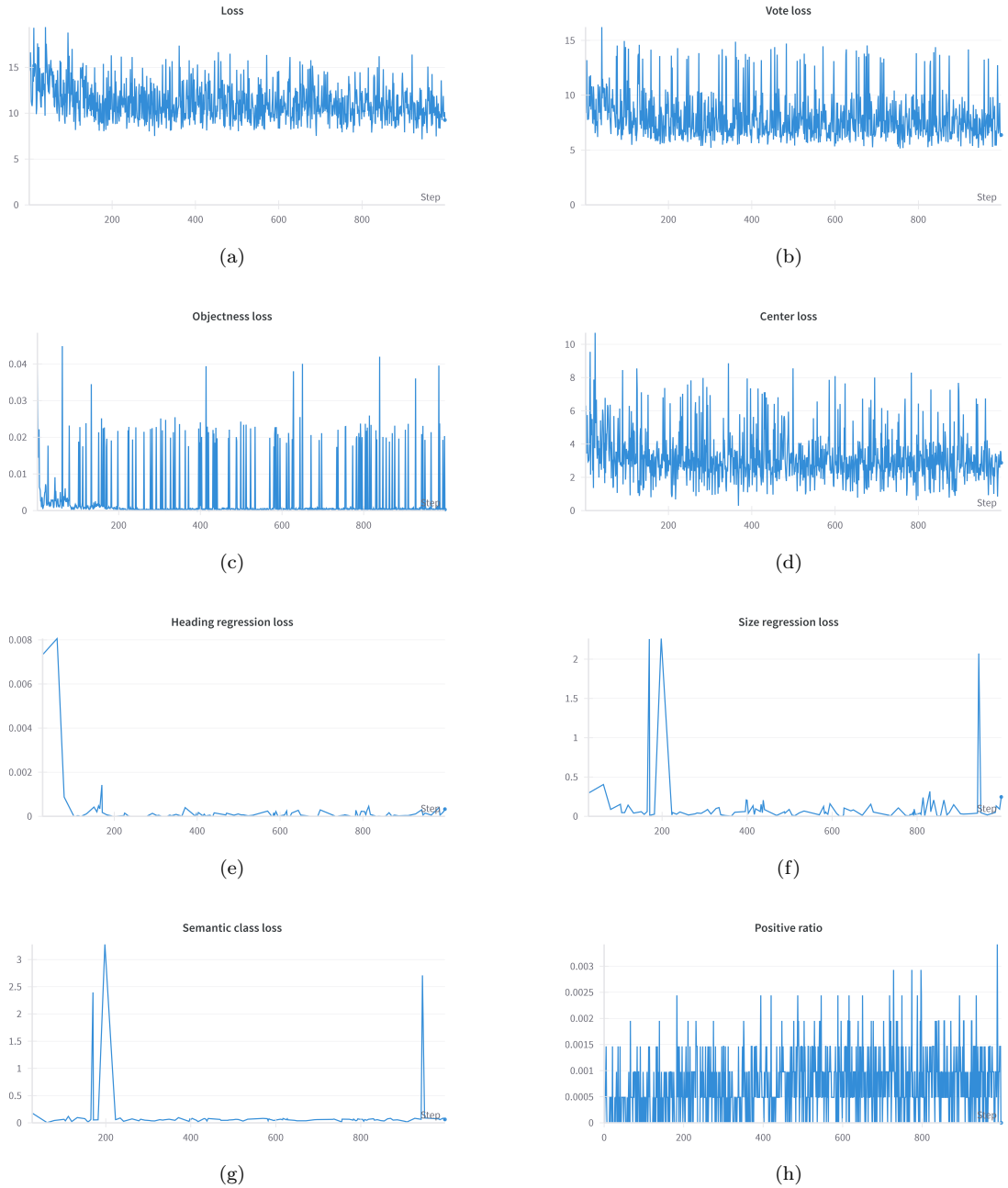


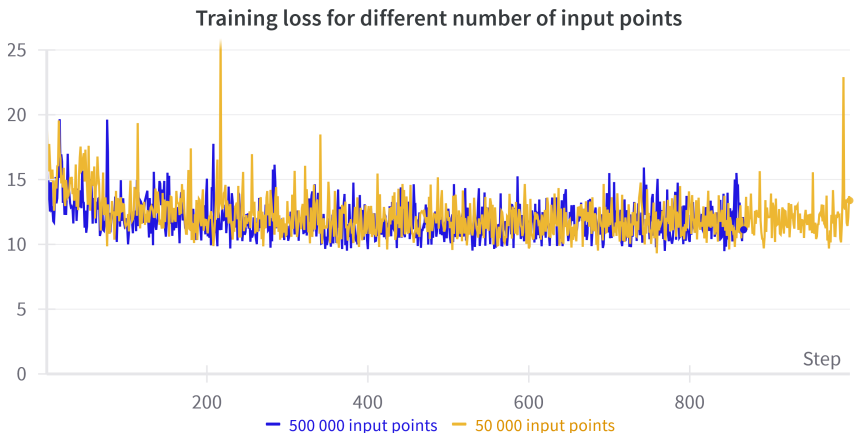
Figure 6.2: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed over the validation dataset

6.2 Experiments

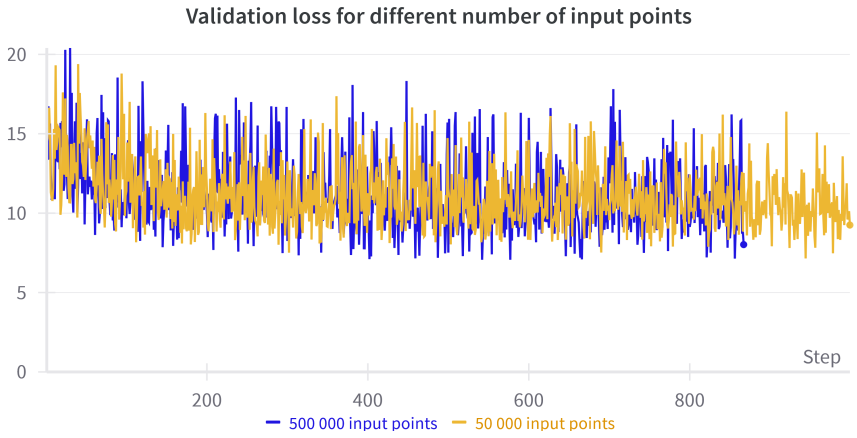
6.2.1 Number of points

For this test, the number of point inputs was modified to 500 000. The aim was to see if using 50 000 did not decrease too much the information passed to the model. Thus the test was done using the following parameters:

- Number of points: 500 000
- Batch size: 8
- Learning rate: 10^{-2}
- Weights initialization: random
- Optimization algorithm: Adam



(a)



(b)

Figure 6.3: Training (a) and validation (b) losses obtained for the VoteNet model with 50 000 and 500 000 input points.

It is important to note that due to the size of the model using 500 000 input points, the main disk's memory became saturated before reaching 1000 epochs, causing the training to stop after 867 epochs. However, since the model appeared to have converged by that point, this training was retained.

The losses obtained on the training and validation datasets were compared with those from the basic configuration (see Figure 6.3). While some local differences can be observed for the training between the corresponding steps of both models, no significant difference is evident in the overall trend. Both models tend to converge toward the same value with similar variability between steps.

For the subsequent steps, the number of points was set at 500 000, despite not improving the model's results. This decision was made because it provides the model with more information about the scene. It was hypothesized that the lack of improvement might be due to other hyperparameters not being optimally determined, but that this could still be a better value for this particular parameter. To prevent further memory shortages, some files were transferred to a secondary disk to free up additional storage space.

6.2.2 Weight initialization

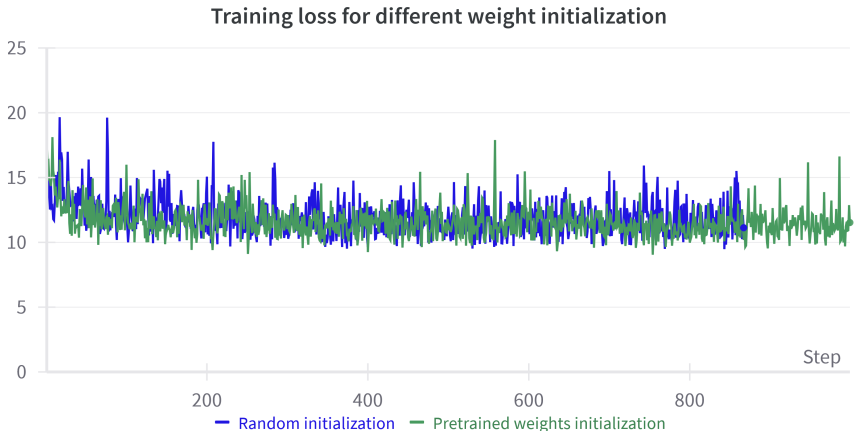
For this test, the weights initialization was modified, in order to see if the training of the model on another dataset could be used to enhance the results of the model. Thus the test was done using the following parameters:

- Number of points: 500 000
- Batch size: 8
- Learning rate: 10^{-2}
- Weights initialization: Pretrained weights
- Optimization algorithm: Adam

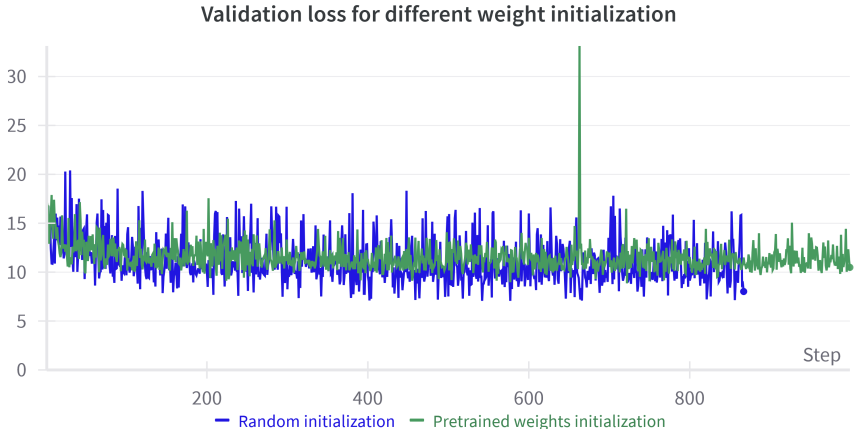
The losses obtained on both the training and validation datasets were compared with those from the current best configuration:

- Number of points: 500 000
- Batch size: 8
- Learning rate: 10^{-2}
- Weights initialization: Random
- Optimization algorithm: Adam

While no significant improvement in training loss was observed (see Figure 6.4(a)), the loss on the validation dataset appears to be more stable (see Figure 6.4(b)). This difference in variation across the validation loss is evident for the different types of losses (see Appendix A.8.3), particularly for the vote loss (see Figure 6.5). Therefore, although this initialization does not directly enhance model performance, it seems to improve generalization. Therefore, this parameter was retained for the subsequent steps.



(a)



(b)

Figure 6.4: Training (a) and validation (b) losses obtained for the VoteNet model using random weight initialization and pretrained weights from the model trained on the ScanNet dataset.

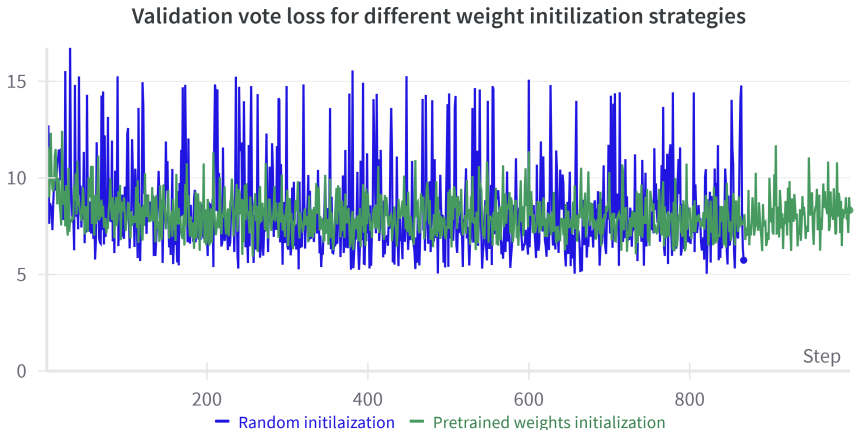


Figure 6.5: Validation vote loss obtained for the VoteNet model using random weight initialization and pretrained weights from the model trained on the ScanNet dataset.

6.2.3 Batch size

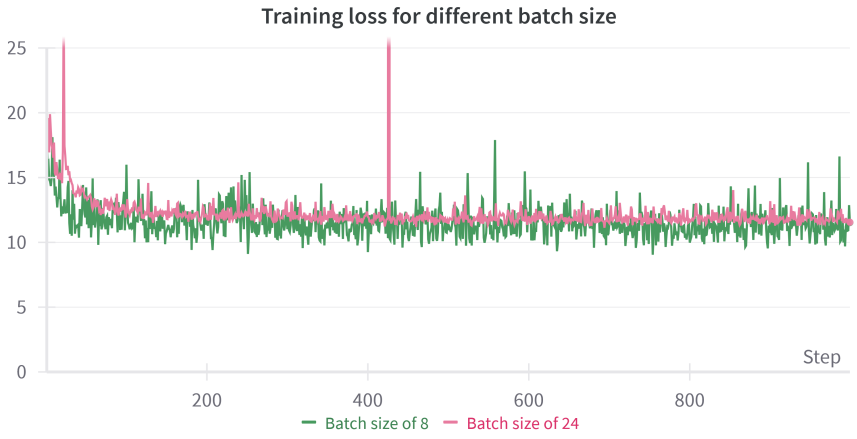
The final test conducted involved modifying the batch size. The objective was to assess whether using more data in each iteration could enable the model to better approximate a suitable function for the dataset. The test was carried out using the following parameters:

- Number of points: 500 000
- Batch size: 24
- Learning rate: 10^{-2}
- Weights initialization: Pretrained weights
- Optimization algorithm: Adam

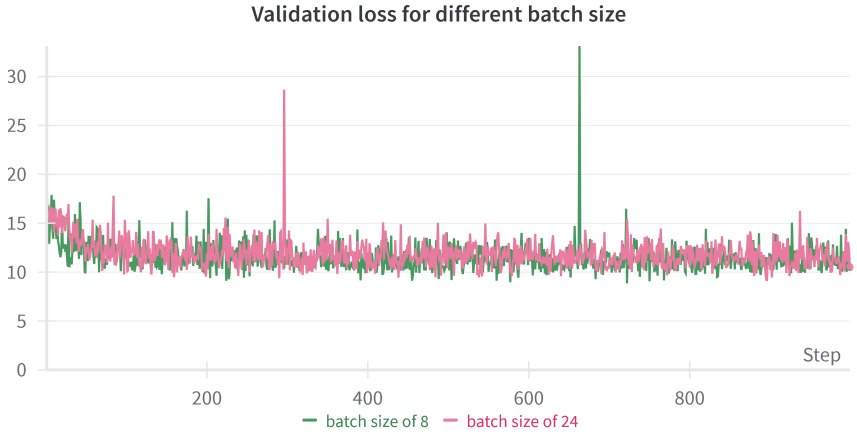
This configuration was compared to the one resulting from the previous experiment:

- Number of points: 500 000
- Batch size: 8
- Learning rate: 10^{-2}
- Weights initialization: Pretrained weights
- Optimization algorithm: Adam

In this configuration, the variation from one epoch to another is reduced for the training dataset (see Figure 6.6). However, no significant change was observed in the validation loss, and the model still converges to the same value for both datasets. As a result, this higher value does not enhance the model. Moreover, since the model seems to have converged to a local minimum of the function it approximates, a more consistent loss function is less desirable than one with more variance. A greater magnitude of steps in the loss function might help escape this local minimum. Therefore, a batch size of 8 was maintained.



(a)



(b)

Figure 6.6: Training (a) and validation (b) losses obtained for the VoteNet model using batch size of 8 and 24.

6.3 Model Assessment

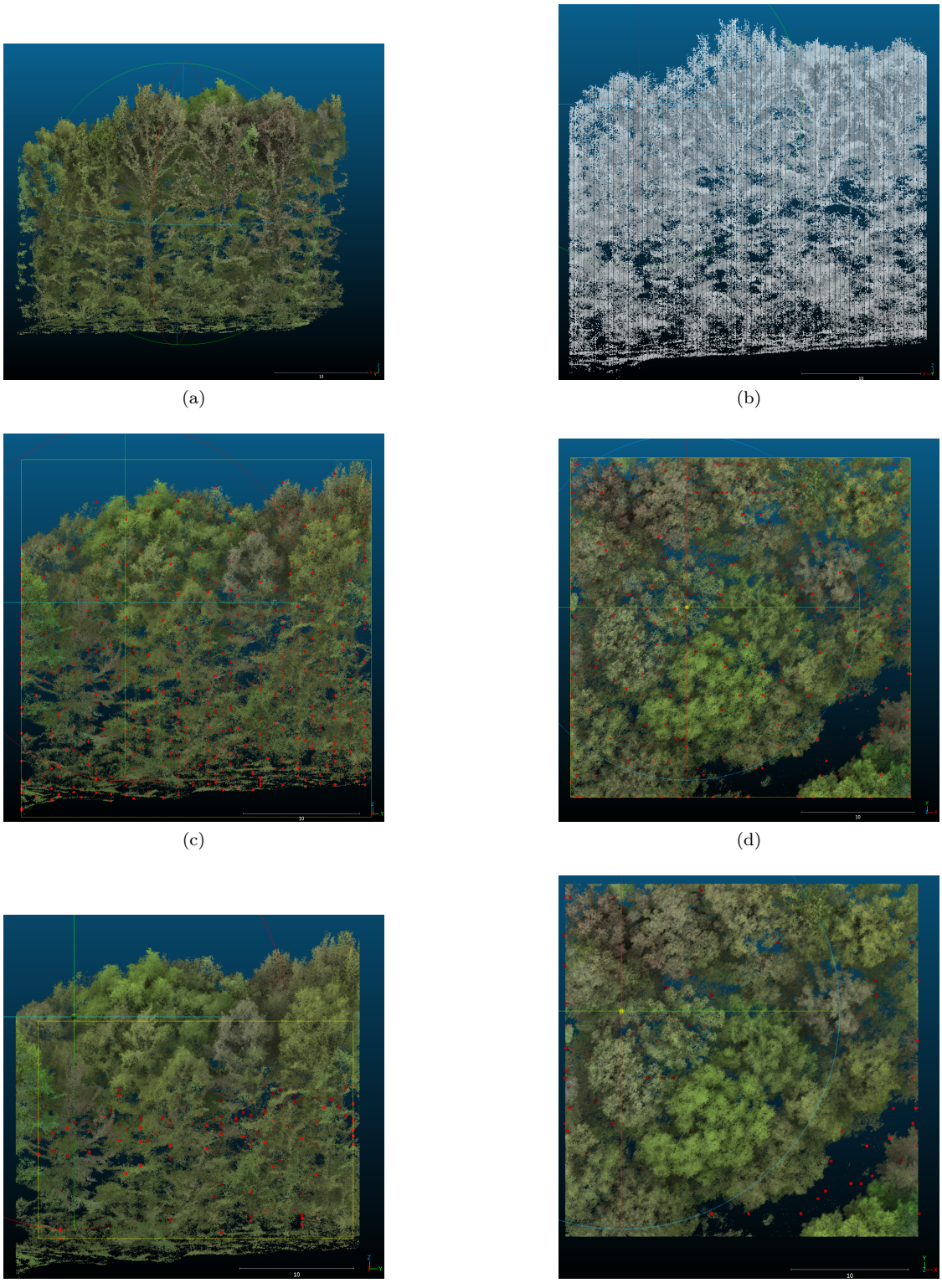
Thus, from the above results, the best configuration was determined as:

- Number of points: 500 000
- Batch size: 8
- Learning rate: 10^{-2}
- Weights initialization: Pretrained weights
- Optimization algorithm: Adam

At the end of the training, the model parameters that resulted in the lowest general loss on the validation set were saved, allowing the trained model to be reused.

However, as discussed in previous sections, none of the configurations led to successful training of the model. Consequently, no tree could be reliably extracted, and the

quality metrics outlined in the methodology could not be computed. Nevertheless, visualizations of the different model outputs were generated to provide insight into the model's performance.



(a) (b) (c) (d) (e) (f)

Figure 6.7: Original point cloud (a), sub-sampled point cloud (b), above (c) and front (d) views of the seed points and above (c) and front (d) views of the votes generated

On Figure 6.7 the original point cloud can be seen, along with the sub-sampled point cloud, the seed points represented in the original point cloud, and the vote points also represented in the original point cloud.

The subsampled point cloud (see Figure 6.7b) still retains sufficient information about tree stems and canopy to allow the trees to be distinguishable by human sight.

The seed points (see Figure 6.7c, d) appear to be evenly distributed across the point cloud.

Regarding the votes (see Figure 6.7e, f), they tend to appear on an elevation near the base of the canopy, which is logical since they are intended to represent the centers of trees bounding boxes. However, the planar distribution of the votes seems to be more random, with some votes even appearing in areas without trees (e.g., the bottom right corner of Figure 6.7f).

Even if no tree could be reliably extracted, the bounding boxes produced by the model were visualized (see Figure 6.9, 6.8). It can be seen that the 2D footprints of the bounding boxes (see Figure 6.8) are large, with boxes going from 0.06 to 45.93 m^2 and a mean of 38.98 m^2 for the box predicted for the validation scenes. And some boxes having a big part outside of the scene. Some examples of the boxes extracted can be seen on Figure 6.9.

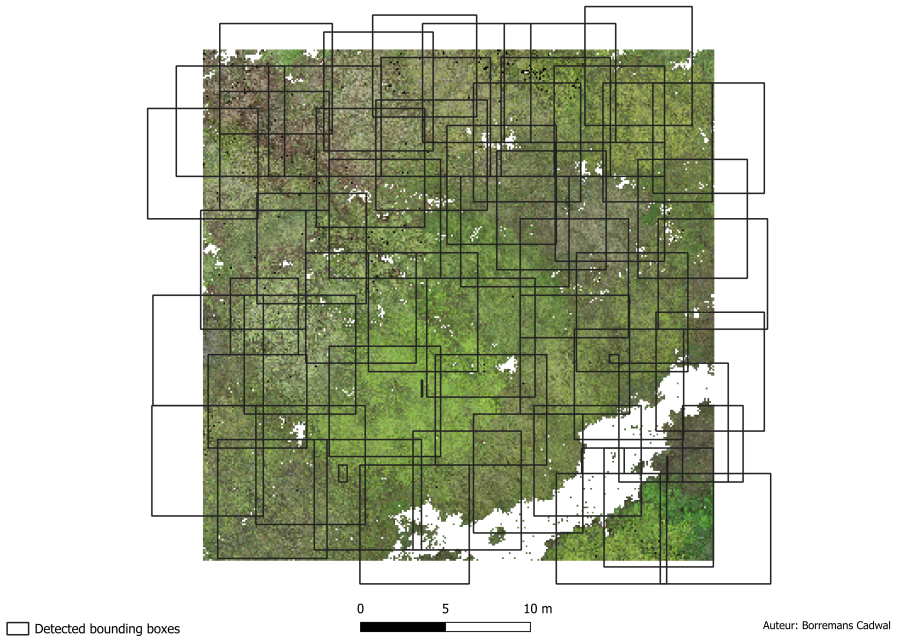
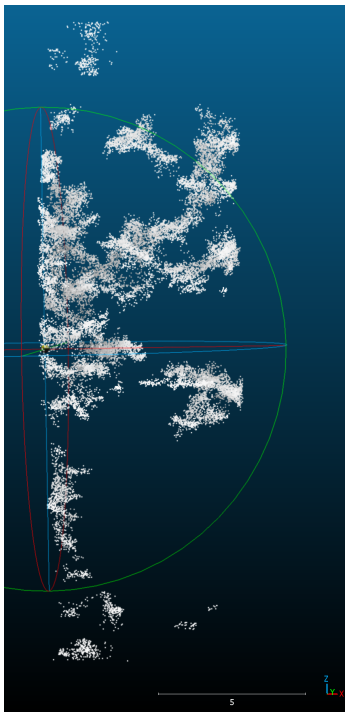
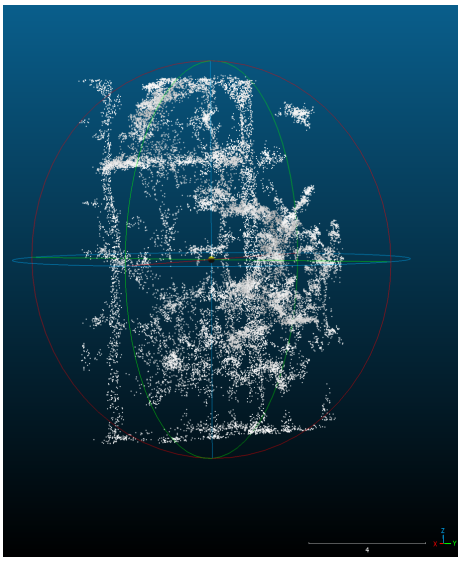


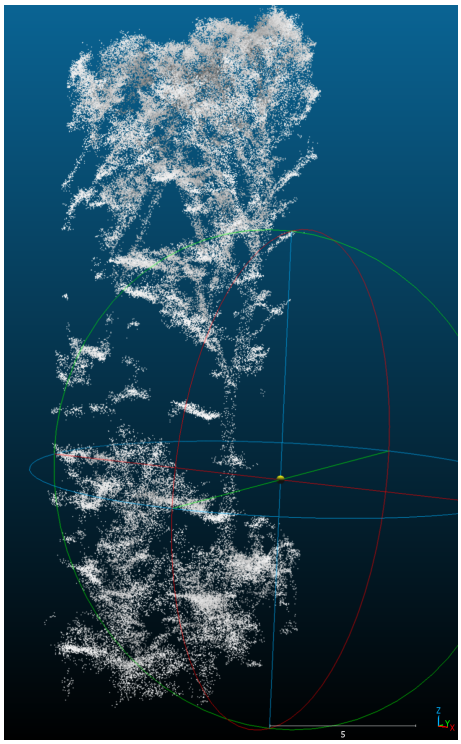
Figure 6.8: 2D footprints of the bounding boxes produced by the VoteNet model



(a)



(b)



(c)

Figure 6.9: Extracted bounding boxes produced by the VoteNet model

Chapter 7

Discussion

In this section, the methodology used and the results obtained will be discussed, along with the limitations of this work and future perspectives.

7.1 Back to the Research Question

While the model could be trained and bounding boxes extracted from the point cloud, the results did not allow to correctly realise tree detection in the acquired dataset. Thus the answer to the research question "Is point-wise deep learning approach a suitable solution for tree detection purposes in mixed forest stands?" is negative.

This result can be explained by the fact that, across the various tests conducted, none of the training attempts allowed the model to sufficiently fit the training dataset to achieve successful tree detection. Indeed, after a few epochs, it converged to high values for most of the loss functions and oscillated around these values. The only losses that were adequately minimized were the semantic class loss and the heading loss. This can be explained by the fact that both of these losses only have one possible outcome, so the model simply outputs the same value for every prediction. Furthermore, it appears that the generated votes tend to cluster at the base of the canopy, which is logical for the location of tree bounding box centers. However, their horizontal dispersion is less consistent, with some votes even falling outside the point cloud.

The various experiments conducted did not lead to an improvement in the model's performance. The first experiment involved increasing the number of points passed to the model. This experiment aimed to determine whether the initial reduction in points was negatively impacting the information provided to the model. However, based on the results obtained, increasing the number of points did not enhance the model's performance. It is possible that this lack of improvement may be attributed to another underlying factor.

The second experiment focused on using the weights from a pretrained model to initialize the weights of the tested model. This approach was expected to reduce the initial loss value in the early epochs and help the model converge toward a lower value.However, the only noticeable impact from this initialization was an increase in the stability of the loss function on the validation dataset, without a decrease in

the value to which it converges. This outcome may be attributed to the significant differences between the scenes in the ScanNet dataset, used to pretrain the model, and the scenes in the dataset created for this thesis. The ScanNet dataset features indoor scenes with an average surface area of 51 m^2, characterized by high and regular point density on the objects it annotates (Dai et al., 2017). In contrast, the scenes used in this work cover an area of 900 m^2, with a high point density in the tree canopies but lower density at the stem level. Additionally, the data can be noisy, making the scenes more complex and possibly limiting the benefits of the pretrained weights. This difference in scene characteristics could explain why the model did not exhibit improved convergence during training. The pretrained weights, optimized for a different type of environment, may not have provided a meaningful starting point for the model in this new context, thereby preventing its performance improvement.

The final test conducted involved varying the batch size, which controls the number of elements provided to the model at each iteration. Increasing the batch size can improve the stability of the model's training process. Although the updates to the model became more stable throughout the training, this adjustment did not enhance the model's performance or help it avoid reaching a local minimum.

This convergence toward a high value can be explained by various reasons. Among these, the principle are the following ones:

- Data Availability: The amount and quality of data required by deep learning models present one of the main challenges in training them effectively. For a model to accurately learn and represent the problem it is designed to solve, a substantial amount of high-quality labeled data is essential (Alzubaidi et al., 2023). In this case, due to the time-consuming and labor-intensive methodology developed for data labeling, only a portion of the dataset could be labeled, resulting in a limited amount of labeled data. Data augmentation was employed to mitigate this issue during the experiments. However, while data augmentation can enhance a model's generalization ability by introducing variations of existing data, it does not generate entirely new data. Therefore, it is more suited to improving a model's generalization rather than assisting it in fitting the training data accurately.

- Hyperparameters optimization: Defining optimal hyperparameters is a crucial aspect of training a deep learning model, significantly impacting the model's ability to accurately approximate the problem it is designed to solve (Ippolito, 2022). In this study, some hyperparameters of the model were adjusted. However, due to the time-consuming implementation and labeling phases, a thorough exploration of the best hyperparameter configurations could not be conducted. Despite the adjustments made to the model's hyperparameters during the experiments, no significant improvement in performance was observed.

- Expressive capacity: The expressive capacity of a deep learning model represents its ability to approximate complex functions. It is directly linked to the hypothesis space of the model, i.e., the family of functions it can approximate. One of the main parameter affecting this expressivity is the model size (Hu et al., 2021). In this study, the issue of model complexity was not directly addressed, as the architecture was left unchanged. Although the model's architecture was originally designed for object detection on point clouds and has

been successfully applied to other datasets, the scenes used in this study were exterior environments containing complex objects, i.e., trees. This contrasts with the previous tests, where the model was applied to indoor scenes with objects of more consistent shapes (e.g., chairs, tables, lamps).

The most likely explanation is the lack of data, as the model has demonstrated successful object detection on other datasets. Although this dataset may be noisier and more complex than traditional indoor datasets. Various tests were conducted with different parameter configurations, but no significant improvement in results was observed. Therefore, the lack of data seems to be the most probable reason for the training issues.

Another possible explanation for these results is the size of the scenes, which is much larger compared to the datasets used in papers presenting VoteNet (Qi et al., 2019) or the scenes in ScanNet dataset. Since those are indoor scenes, their sizes are generally smaller than those in the dataset created for this work.

7.2 Limitations

The major limitation of this study is the lack of well-labeled data. The absence of efficient labeling software necessitated the use of a sub-optimal methodology that was both time-consuming and prone to low quality labels. As a result, the labeling process was imperfect, which could lead to incomplete or inaccurate annotations. Specifically, the tree annotation process involved extracting trees sequentially, which posed issues when trees overlapped. The first tree labeled in an overlapping region might include parts of the second tree, thus its extraction leads to incomplete annotations for subsequent trees. This limitation in both data quality and quantity significantly constrained the training process and was likely a major factor in the model's inability to converge effectively. This limitation is also due to the lack of publicly available data for 3D object detection in forests, thus limiting the possibilities to increase the amount of available data.

Another limitation is the limited hyperparameter tuning performed. In this work, only a few hyperparameters were tested, and for each, only a limited range of values was explored. This approach provided only preliminary insights into their effects on the model. Additionally, many hyperparameters were set to fixed values based on initial assumptions. A more comprehensive evaluation of hyperparameter configurations is necessary to determine the optimal settings for these parameters and fully understand their impact on model performance.

Another constraint related to the previously mentioned issue of data quality was the lack of effective labeling software. Given the absence of publicly available 3D object detection labeling tools capable of handling large datasets, an alternative methodology had to be designed. The developed approach involved using CloudCompare to extract trees individually from each scene. This method was not only time-consuming but also impacted the quality of the labeling process. As a result, the dataset was only partially labeled, with some bounding boxes lacking accuracy.

This study tested only one model to assess the applicability of 3D object detection using deep learning, and the architecture of the model was not modified. The

model architecture could be modified to increase the model capacity, for example by increasing the number of seed points extracted, or the number of votes generated per seed points. Further more, other models exist and could be evaluated to more thoroughly assess the effectiveness and precision of deep learning for such tasks.

Several challenges were encountered throughout this work. Firstly, implementing the framework was complex, requiring an understanding of various sophisticated Python libraries and managing conflicts between different versions of these libraries. Secondly, handling the framework, which involves highly modular code in Python and YAML, required the comprehension of advanced deep learning concepts. Finally the development and realisation of a methodology to label the data was also a highly time-consuming task. These factors slowed down the development process and constrained the testing phase.

7.3 Future perspectives

Based on previously mentioned limits, the further development should firstly focus on the increase of the amount and quality of labeled data, which can be achieved in several ways. Firstly, by completing the labeling of the data acquired in this study. The workflow presented in section 5.1 can be reused for this purpose. Additionally, open access data could be utilized. However, to the best of our knowledge, no publicly available forest point cloud data for object detection has been found. Nonetheless, outdoor urban data that includes a tree class is more likely to be available and could be used for initial training of the model, before fine-tuning it on the acquired dataset.

In this context, a more efficient data labeling method could also be explored. One possibility is to use an existing tree detection algorithm, as one presented in the state of the art, to apply it to the dataset. This could automate the extraction of a significant portion of the trees from the point cloud, with the remaining trees being manually labeled using the workflow presented in this paper. Another possibility is the development of a new 3D bounding boxes labeling software. This task would require extensive development knowledge, however it would be a significant work since it is one of the lack in the domain underlined in this work.

Testing the impact of scene size could be beneficial to determine if the model handles smaller scenes more effectively, potentially improving the utility of weights pretrained on ScanNet dataset. To evaluate different scene sizes without the need to relabel all the trees each time, a Python script could be developed to classify the bounding boxes based on their location relative to the scenes. However, special attention should be given to handling trees situated on the boundary between two scenes.

Furthermore, a significant way of improvement is a thorough testing of different hyperparameter configurations. Hyperparameter tuning is crucial for optimizing deep learning models. Additionally, exploring changes to the model's complexity (e.g., increasing the number of layers of the modules, increase the number of neurons per layers, increase the number of seed points sampled, increase the number of votes per seed points) would be a valuable research direction.

If the proposed improvements are implemented and the model is retrained, it will be essential to evaluate its performance by comparing it against state-of-the-art methods. This comparison requires the implementation of various methods and testing them on the same dataset. By evaluating their mean Average Precision (mAP) at different Intersection over Union (IoU) thresholds, the differences in performance can be assessed effectively.

As noted in the limitations, exploring different model architectures is also a potential avenue for improvement. Currently, TorchPoints3D does not implement other 3D object detection models. Therefore, applying the methodology outlined in this paper to a different model would require implementing that model within the framework. One promising architecture to consider is transformer-based models. Originally designed for machine translation, transformers have increasingly been applied to various deep learning domains (e.g., computer vision, natural language processing, speech processing). Transformers typically utilize an encoder-decoder architecture. The encoder, which comprises multiple self-attention layers, processes the input data to extract relevant features. The decoder then generates the output data based on the encoder's output and the decoder's previous outputs (Lin et al., 2022). The application of transformer-based architectures to 3D object detection is a growing area of research (Misra et al., 2021; Zhou et al., 2023; Zhu et al., 2024).

Chapter 8

Conclusion

The aim of this study was to make a preliminary attempt at 3D object detection in an aerial forest point cloud using deep learning. This effort was motivated by the need to enhance forest management by developing methods for conducting precise, large-scale forest inventories in a time-efficient manner. To achieve this, an aerial point cloud dataset was acquired, and a labeling methodology was proposed for annotating the dataset. The VoteNet model was selected as the model to be tested in this work. The Python framework for 3D deep learning , TorchPoints3D, was implemented, handled, and extended to include the acquired dataset. Subsequently, tests were conducted to assess the applicability of this model for forest inventories. While the model was successfully trained on the acquired dataset and initial tests were conducted, the applicability of deep learning for 3D tree detection in heterogeneous environments could not be conclusively demonstrated.

Several factors likely contributed to these results. The primary issue was the lack of labeled data. Due to the complex implementation and handling of the framework, combined with a time-consuming labeling process, only a portion of the acquired dataset could be labeled. Additionally, there is considerable room for improvement in the model's parametrization, as only a limited number of configurations were tested, and no modifications were made to the model's architecture. As a result, although this first approach did not confirm the applicability of VoteNet for tree detection in heterogeneous forest environments, there remains significant potential for further development and improvement

Additionally, this work highlighted several challenges in the field that should be addressed in future studies. Firstly, the lack of publicly available data presents a major obstacle. For future research in this area, more data need to be acquired and labeled to enhance model training. Secondly, the absence of efficient labeling software complicates the process, requiring the use of tedious and time-consuming alternatives, which makes data labeling more difficult and lengthy. Although a labeling method was proposed in this work, the method is time-consuming and would not be easily applicable to very large datasets. However, alternative approaches were suggested to address this issue. Finally, this study provides a comprehensive methodology to facilitate the implementation and handling of the TorchPoints3D framework, thus potentially saving valuable time for future research efforts, enabling to focus more on the configuration and testing of the model.

Bibliography

Alzubaidi, L., Bai, J., Al-Sabaawi, A., Santamaría, J., Albahri, A. S., Al-dabbagh, B. S. N., Fadhel, M. A., Manoufali, M., Zhang, J., Al-Timemy, A. H., Duan, Y., Abdullah, A., Farhan, L., Lu, Y., Gupta, A., Albu, F., Abbosh, A., & Gu, Y. (2023). A survey on deep learning tools dealing with data scarcity: Definitions, challenges, solutions, tips, and applications. *Journal of Big Data*, *10*(46). https://doi.org/10.1186/s40537-023-00727-2

Angevine, W., Senff, C., & Westwater, E. (2003). Boundary layers | observational techniques – remote. In J. R. Holton (Ed.), *Encyclopedia of atmospheric sciences* (pp. 271–279). Academic Press. https://doi.org/10.1016/B0-12-227090-8/00089-0

Banskota, A., Kayastha, N., Falkowski, M. J., Wulder, M. A., Froese, R. E., & White, J. C. (2014). Forest monitoring using landsat time series data: A review. *Canadian Journal of Remote Sensing*, *40*(5), 362–384. https://doi.org/10.1080/07038992.2014.987376

Bates, J. S., Montzka, C., Schmidt, M., & Jonard, F. (2021). Estimating canopy density parameters time-series for winter wheat using uas mounted lidar. *Remote Sensing*, *13*(4). https://doi.org/10.3390/rs13040710

Campello, R. J. G. B., Moulavi, D., & Sander, J. (2013). Density-based clustering based on hierarchical density estimates. In J. Pei, V. S. Tseng, L. Cao, H. Motoda, & G. Xu (Eds.), *Advances in knowledge discovery and data mining* (pp. 160–172). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-37456-2_14

Chaton, T., Chaulet, N., Horache, S., & Landrieu, L. (2020). Torch-Points3D: A modular multi-task framework for reproducible deep learning on 3D point clouds. *3DV 2020 - International Conference on 3D Vision*. https://hal.science/hal-03013190

Chen, Q., Baldocchi, D., Gong, P., & Kelly, M. (2006). Isolating individual trees in a savanna woodland using small footprint lidar data. *Photogrammetric Engineering and Remote Sensing*, *72*(8), 923–932. https://doi.org/10.14358/PERS.72.8.923

Coptrz. (2021, July 3). Dji's first lidar solution – zenmuse l1. Retrieved August 10, 2024, from https://coptrz.com/blog/djis-first-lidar-solution-zenmuse-l1/

Dai, A., Chang, A. X., Savva, M., Halber, M., Funkhouser, T., & Nießner, M. (2017). Scannet: Richly-annotated 3d reconstructions of indoor scenes. https://doi.org/10.48550/arXiv.1702.04405

Dalponte, M., & Coomes, D. A. (2016). Tree-centric mapping of forest carbon density from airborne laser scanning and hyperspectral data. *Methods in Ecology and Evolution*, *7*(10), 1236–1245. https://doi.org/10.1111/2041-210X.12575

Demchenko, Y., Turkmen, F., de Laat, C., Hsu, C.-H., Blanchet, C., & Loomis, C. (2017). Chapter 2 - cloud computing infrastructure for data intensive applications. In H.-H. Hsu, C.-Y. Chang, & C.-H. Hsu (Eds.), *Big data analytics for sensor-network collected intelligence* (pp. 21–62). Academic Press. https://doi.org/10.1016/B978-0-12-809393-1.00002-7

Diez, Y., Kentsch, S., Fukuda, M., Caceres, M. L. L., Moritake, K., & Cabezas, M. (2021). Deep learning in forestry using uav-acquired rgb data: A practical review. *Remote Sensing*, *13*(14). https://doi.org/10.3390/rs13142837

DJI. (n.d.-a). Matrice 300 rtk. Retrieved August 10, 2024, from https://enterprise.dji.com/matrice-300

DJI. (n.d.-b). Support for zenmuse l1 - dji. Retrieved August 10, 2024, from https://www.dji.com/be/support/product/zenmuse-l1

döt Net, I., Müller, T., Antoniou, P., Aro, E., & Smith, T. (2021, October 1). Yaml ain't markup language (yaml) version 1.2. Retrieved July 22, 2024, from https://yaml.org/spec/1.2.2/

Eva, L., & Johan, H. (2017). Individual tree crown methods for 3d data from remote sensing. *Current Forestry Reports*, *3*, 19–31. https://doi.org/10.1007/s40725-017-0051-6

Ferrara, R., Virdis, S. G., Ventura, A., Ghisu, T., Duce, P., & Pellizzaro, G. (2018). An automated approach for wood-leaf separation from terrestrial lidar point clouds using the density based clustering algorithm dbscan. *Agricultural and Forest Meteorology*, *262*, 434–444. https://doi.org/10.1016/j.agrformet.2018.04.008

Girshick, R. (2015). Fast r-cnn. *2015 IEEE International Conference on Computer Vision (ICCV)*, 1440–1448. https://doi.org/10.1109/ICCV.2015.169

Guo, Q., Su, Y., & Hu, T. (2023). Chapter 2 - working principles of lidar. In Q. Guo, Y. Su, & T. Hu (Eds.), *Lidar principles, processing and applications in forest ecology* (pp. 23–61). Academic Press. https://doi.org/10.1016/B978-0-12-823894-3.00002-5

Gupta, S., Weinacker, H., & Koch, B. (2010). Comparative analysis of clustering-based approaches for 3-d single tree detection using airborne fullwave lidar data. *Remote Sensing*, *2*(4), 968–989. https://doi.org/10.3390/rs2040968

Hall, F. G., Bergen, K., Blair, J. B., Dubayah, R., Houghton, R., Hurtt, G., Kellndorfer, J., Lefsky, M., Ranson, J., Saatchi, S., Shugart, H., & Wickland, D. (2011). Characterizing 3d vegetation structure from space: Mission requirements [DESDynI VEG-3D Special Issue]. *Remote Sensing of Environment*, *115*(11), 2753–2775. https://doi.org/10.1016/j.rse.2011.01.024

Hershey, J. L., McDill, M. E., Miller, D. A., Holderman, B., & Michael, J. H. (2022). A voxel-based individual tree stem detection method using airborne lidar in mature northeastern u.s. forests. *Remote Sensing*, *14*(3). https://doi.org/10.3390/rs14030806

Hong, Q., Jun, M., Bo, W., Sichao, T., Jiayi, Z., Biao, L., Tong, L., & Ruifeng, T. (2023). Application of data-driven technology in nuclear engineering: Prediction, classification and design optimization. *Annals of Nuclear Energy*, *194*, 110089. https://doi.org/10.1016/j.anucene.2023.110089

Hu, X., Chu, L., Pei, J., Liu, W., & Bian, J. (2021). Model complexity of deep learning: A survey. https://doi.org/10.48550/arXiv.2103.05127

Hussein, E. M. (2011). 3 - measurement models. In E. M. Hussein (Ed.), *Computed radiation imaging* (pp. 27–36). Elsevier. https://doi.org/10.1016/B978-0-12-387777-2.00003-3

Ippolito, P. P. (2022). Hyperparameter tuning. In R. Egger (Ed.), *Applied data science in tourism: Interdisciplinary approaches, methodologies, and applications* (pp. 231–251). Springer International Publishing. https://doi.org/10.1007/978-3-030-88389-8_12

Jaskierniak, D., Lucieer, A., Kuczera, G., Turner, D., Lane, P., Benyon, R., & Haydon, S. (2021). Individual tree detection and crown delineation from unmanned aircraft system (uas) lidar in structurally complex mixed species eucalypt forests. *ISPRS Journal of Photogrammetry and Remote Sensing*, *171*, 171–187. https://doi.org/10.1016/j.isprsjprs.2020.10.016

Jonard, F., André, F., Ponette, Q., Vincke, C., & Jonard, M. (2011). Sap flux density and stomatal conductance of european beech and common oak trees in pure and mixed stands during the summer drought of 2003. *Journal of Hydrology*, *409*(1), 371–381. https://doi.org/10.1016/j.jhydrol.2011.08.032

Kuželka, K., Slavík, M., & Surový, P. (2020). Very high density point clouds from uav laser scanning for automatic tree stem detection and direct diameter measurement. *Remote Sensing*, *12*(8). https://doi.org/10.3390/rs12081236

Latella, M., Sola, F., & Camporeale, C. (2021). A density-based algorithm for the detection of individual trees from lidar data. *Remote Sensing*, *13*(2). https://doi.org/10.3390/rs13020322

Le Noë, J., Erb, K.-H., Matej, S., Magerl, A., Bhan, M., & Gingrich, S. (2021). Altered growth conditions more than reforestation counteracted forest biomass carbon emissions 1990–2020. *Nature communication*, *12*. https://doi.org/10.1038/s41467-021-26398-2

Lefsky, M. A., Cohen, W. B., Parker, G. G., & Harding, D. J. (2002). Lidar Remote Sensing for Ecosystem Studies: Lidar, an emerging remote sensing technology that directly measures the three-dimensional distribution of plant canopies, can accurately estimate vegetation structural attributes and should be of particular interest to forest, landscape, and global ecologists. *BioScience*, *52*(1), 19–30. https://doi.org/10.1641/0006-3568(2002)052[0019:LRSFES]2.0.CO;2

Li, C., Chen, Q., Gu, G., & Qian, W. (2013). Laser time-of-flight measurement based on time-delay estimation and fitting correction. *Optical Engineering*, *52*(7), 076105. https://doi.org/10.1117/1.OE.52.7.076105

Li, J., Zhou, J., Xiong, Y., Chen, X., & Chakrabarti, C. (2022). An adjustable farthest point sampling method for approximately-sorted point cloud data. *2022 IEEE Workshop on Signal Processing Systems (SiPS)*, 1–6. https://doi.org/10.1109/SiPS55645.2022.9919246

Li, W., Guo, Q., Jakubowski, M. K., & Kelly, M. (2012). A new method for segmenting individual trees from the lidar point cloud. *Photogrammetric Engineering And Remote Sensing*, *78*, 75–84. https://doi.org/10.14358/PERS.78.1.75

Li, Y., Xie, D., Wang, Y., Jin, S., Zhou, K., Zhang, Z., Li, W., Zhang, W., Mu, X., & Yan, G. (2023). Individual tree segmentation of airborne and uav lidar point clouds based on the watershed and optimized connection center evolution clustering [e10297 ECE-2023-02-00195.R1]. *Ecology and Evolution*, *13*(7), e10297. https://doi.org/10.1002/ece3.10297

Liang, J., Crowther, T. W., Picard, N., Wiser, S., Zhou, M., Alberti, G., Schulze, E.-D., McGuire, A. D., Bozzato, F., Pretzsch, H., de-Miguel, S., Paquette, A., Hérault, B., Scherer-Lorenzen, M., Barrett, C. B., Glick, H. B., Hengeveld, G. M., Nabuurs, G.-J., Pfautsch, S., ... Reich, P. B. (2016). Positive biodiversity-productivity relationship predominant in global forests. *Science*, *354*(6309), aaf8957. https://doi.org/10.1126/science.aaf8957

Lim, K., Treitz, P., Wulder, M., St-Onge, B., & Flood, M. (2003). Lidar remote sensing of forest structure. *Progress in Physical Geography: Earth and Environment*, *27*(1), 88–106. https://doi.org/10.1191/0309133303pp360ra

Lin, T., Wang, Y., Liu, X., & Qiu, X. (2022). A survey of transformers. *AI Open*, *3*, 111–132. https://doi.org/10.1016/j.aiopen.2022.10.001

McInnes, L., Healy, J., & Astels, S. (2016). How hdbscan works — hdbscan 0.8.1 documentation. Retrieved July 21, 2024, from https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html

Meta Platforms. (n.d.). Getting started | hydra. Retrieved April 28, 2024, from https://hydra.cc/docs/intro/

Misra, I., Girdhar, R., & Joulin, A. (2021). An end-to-end transformer model for 3d object detection. https://doi.org/10.48550/arXiv.2109.08141

Misra, S., & Li, H. (2020). Chapter 7 - deep neural network architectures to approximate the fluid-filled pore size distributions of subsurface geological formations. In S. Misra, H. Li, & J. He (Eds.), *Machine learning for subsurface characterization* (pp. 183–217). Gulf Professional Publishing. https://doi.org/10.1016/B978-0-12-817736-5.00007-7

Monnet, J.-M., Mermin, E., Chanussot, J., & Berger, F. (2010, September). Tree top detection using local maxima filtering: a parameter sensitivity analysis. https://hal.science/hal-00523245/

Neuville, R., Bates, J. S., & Jonard, F. (2021). Estimating forest structure from uav-mounted lidar point cloud using machine learning. *Remote Sensing*, *13*(3). https://doi.org/10.3390/rs13030352

Ni-Meister, W., Lee, S., Strahler, A. H., Woodcock, C. E., Schaaf, C., Yao, T., Ranson, K. J., Sun, G., & Blair, J. B. (2010). Assessing general relationships between aboveground biomass and vegetation structure parameters for improved carbon estimate from lidar remote sensing. *Journal of Geophysical Research: Biogeosciences*, *115*(G2). https://doi.org/10.1029/2009JG000936

Padilla, R., Netto, S. L., & da Silva, E. A. B. (2020). A survey on performance metrics for object-detection algorithms. *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, 237–242. https://doi.org/10.1109/IWSSIP48289.2020.9145130

PyTorch. (n.d.-a). PyTorch documentation — PyTorch 2.4 documentation. Retrieved July 25, 2024, from https://pytorch.org/docs/stable/index.html

PyTorch. (n.d.-b). Torch$_g$*eometric.data.inmemorydataset*. Retrieved April 5, 2024, from https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.data.InMemoryDataset.html

PyTorch. (n.d.-c). Torch.tensor. Retrieved August 14, 2024, from https://pytorch.org/docs/stable/tensors.html

Qi, C. R., Litany, O., He, K., & Guibas, L. J. (2019). Deep hough voting for 3d object detection in point clouds. *CoRR*, *abs/1904.09664*. https://doi.org/10.48550/arXiv.1904.09664

Ravikumar, N., Zakeri, A., Xia, Y., & Frangi, A. F. (2024). Chapter 16 - deep learning fundamentals. In A. F. Frangi, J. L. Prince, & M. Sonka (Eds.), *Medical image analysis* (pp. 415–450). Academic Press. https://doi.org/10.1016/B978-0-12-813657-7.00041-8

Rawal, K., Parthvi, A., Choubey, D. K., & Shukla, V. (2021). Chapter 16 - prediction of leukemia by classification and clustering techniques. In P. Kumar, Y. Kumar, & M. A. Tawhid (Eds.), *Machine learning, big data, and iot for medical informatics* (pp. 275–295). Academic Press. https://doi.org/10.1016/B978-0-12-821777-1.00003-3

Razavi, S. (2021). Deep learning, explained: Fundamentals, explainability, and bridgeability to process-based modelling. *Environmental Modelling and Software*, *144*, 105159. https://doi.org/10.1016/j.envsoft.2021.105159

Reitberger, J., Schnörr, C., Krzystek, P., & Stilla, U. (2009). 3d segmentation of single trees exploiting full waveform lidar data. *ISPRS Journal of Photogrammetry and Remote Sensing*, *64*(6), 561–574. https://doi.org/10.1016/j.isprsjprs.2009.04.002

Ren, J., & Wang, H. (2023). Chapter 3 - calculus and optimization. In J. Ren & H. Wang (Eds.), *Mathematical methods in data science* (pp. 51–89). Elsevier. https://doi.org/10.1016/B978-0-44-318679-0.00009-0

Schmohl, S., Narváez Vallejo, A., & Soergel, U. (2022). Individual tree detection in urban als point clouds with 3d convolutional networks. *Remote Sensing*, *14*(6). https://doi.org/10.3390/rs14061317

She, C.-Y., & Friedman, J. S. (2022). Introduction to lidar remote sensing and the lidar equation. In *Atmospheric lidar fundamentals: Laser light scattering from atoms and linear molecules* (pp. 94–102). Cambridge University Press. https://doi.org/10.1017/9781108968713.007

Sinaga, K. P., & Yang, M.-S. (2020). Unsupervised k-means clustering algorithm. *IEEE Access*, *8*, 80716–80727. https://doi.org/10.1109/ACCESS.2020.2988796

Solberg, S., Naesset, E., & Bollandsas, O. M. (2006). Single Tree Segmentation Using Airborne Laser Scanner Data in a Structurally Heterogeneous Spruce Forest. *Photogrammetric engineering and remote sensing*, *72*(12), 1369–1378. https://doi.org/10.14358/pers.72.12.1369

Sun, K., Roy, A., & Tobin, J. M. (2024). Artificial intelligence and machine learning: Definition of terms and current concepts in critical care research. *Journal of Critical Care*, *82*, 154792. https://doi.org/10.1016/j.jcrc.2024.154792

Suriyan, K., & Ramalingam, N. (2023). Chapter 6 - recent challenges, opportunities, and issues in various data analytics. In A. K. Tyagi & A. Abraham (Eds.), *Data science for genomics* (pp. 99–105). Academic Press. https://doi.org/10.1016/B978-0-323-98352-5.00012-4

Tan, Y., Zhang, Y., Chiu, C.-W., Onda, Y., & Li, Z. (2023). Novel approach for tree detection in japanese plantation forests using drone lidar data. *2023 IEEE 12th Global Conference on Consumer Electronics (GCCE)*, 119–120. https://doi.org/10.1109/GCCE59613.2023.10315267

Tiede, D., Hochleitner, G., & Blaschke, T. (2005). A full gis-based workflow for tree identification and tree crown delineation using laser scanning. *Proceedings of the ISPRS Workshop CMRT*, 9–14. https://api.semanticscholar.org/CorpusID:7161179

Uijlings, J. R. R., van de Sande, K. E. A., Gevers, T., & Smeulders, A. W. M. (2013). Selective search for object recognition. *International Journal of Computer Vision*, *104*, 154–171. https://doi.org/10.1007/s11263-013-0620-5

Wallace, L., Lucieer, A., & Watson, C. S. (2014). Evaluating tree detection and segmentation routines on very high resolution uav lidar data. *IEEE Transactions on Geoscience and Remote Sensing*, *52*(12), 7619–7628. https://doi.org/10.1109/TGRS.2014.2315649

Wallace, L., Musk, R., & Lucieer, A. (2014). An assessment of the repeatability of automatic forest inventory metrics derived from uav-borne laser scanning data. *IEEE Transactions on Geoscience and Remote Sensing*, *52*(11), 7160–7169. https://doi.org/10.1109/TGRS.2014.2308208

Wang, Z., Li, P., Cui, Y., Lei, S., & Kang, Z. (2023). Automatic detection of individual trees in forests based on airborne lidar data with a tree region-based convolutional neural network (rcnn). *Remote Sensing*, *15*(4). https://doi.org/10.3390/rs15041024

Wehr, A., & Lohr, U. (1999). Airborne laser scanning—an introduction and overview. *ISPRS Journal of Photogrammetry and Remote Sensing*, *54*(2), 68–82. https://doi.org/10.1016/S0924-2716(99)00011-8

Weik, M. H. (2001). Rise time. In *Computer science and communications dictionary* (pp. 1498–1498). Springer US. https://doi.org/10.1007/1-4020-0613-6_16429

White, J. C., Coops, N. C., Wulder, M. A., Vastaranta, M., Hilker, T., & Tompalski, P. (2016). Remote sensing technologies for enhancing forest inventories: A review. *Canadian Journal of Remote Sensing*, *42*(5), 619–641. https://doi.org/10.1080/07038992.2016.1207484

Windrim, L., & Bryson, M. (2019). Forest tree detection and segmentation using high resolution airborne lidar. *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3898–3904. https://doi.org/10.1109/IROS40897.2019.8967885

Windrim, L., & Bryson, M. (2020). Detection, segmentation, and model fitting of individual tree stems from airborne laser scanning of forests using deep learning. *Remote Sensing*, *12*(9). https://doi.org/10.3390/rs12091469

Wu, Y., Wang, Y., Zhang, S., & Ogai, H. (2021). Deep 3d object detection networks using lidar data: A review. *IEEE Sensors Journal*, *21*(2), 1152–1171. https://doi.org/10.1109/JSEN.2020.3020626

Yadan, O. (2022). Omegaconf — omegaconf 2.3.0 documentation. Retrieved April 28, 2024, from https://omegaconf.readthedocs.io/en/2.3_branch/

Ying, W., Dong, T., Ding, Z., & Zhang, X. (2021). Pointcnn-based individual tree detection using lidar point clouds [Cited by: 2]. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *13002 LNCS*, 89–100. https://doi.org/10.1007/978-3-030-89029-2_7

Zaidi, S. S. A., Ansari, M. S., Aslam, A., Kanwal, N., Asghar, M., & Lee, B. (2022). A survey of modern deep learning based object detection models. *Digital Signal Processing*, *126*, 103514. https://doi.org/10.1016/j.dsp.2022.103514

Zhao, Z.-Q., Zheng, P., Xu, S.-T., & Wu, X. (2019). Object detection with deep learning: A review. *IEEE Transactions on Neural Networks and Learning Systems*, *30*(11), 3212–3232. https://doi.org/10.1109/TNNLS.2018.2876865

Zhen, Z., Quackenbush, L. J., & Zhang, L. (2016). Trends in automatic individual tree crown detection and delineation—evolution of lidar data. *Remote Sensing*, *8*(4). https://doi.org/10.3390/rs8040333

Zhong, H., Zhang, Z., Liu, H., Wu, J., & Lin, W. (2024). Individual tree species identification for complex coniferous and broad-leaved mixed forests based on deep learning combined with uav lidar data and rgb images [All Open Access, Gold Open Access]. *Forests*, *15*(2). https://doi.org/10.3390/f15020293

Zhou, C., Zhang, Y., Chen, J., & Huang, D. (2023). Octr: Octree-based transformer for 3d object detection. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 5166–5175. https://doi.org/10.48550/arXiv.2303.12621

Zhu, Y., Xu, R., Tao, C., An, H., Wang, H., Sun, Z., & Lu, K. (2024). Ds-trans: A 3d object detection method based on a deformable spatiotemporal transformer for autonomous vehicles. *Remote Sensing*, *16*(9). https://doi.org/10.3390/rs16091621

Appendix

A.1 Fiber Scanner

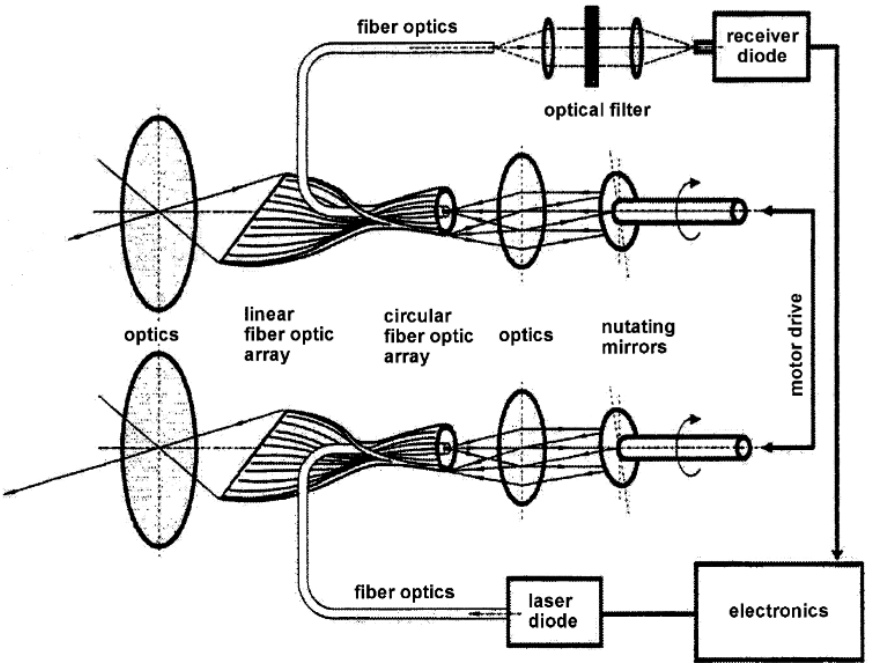


Figure a.1: TopoSys fiber scanner (Wehr & Lohr, 1999).

A.2 Flight and processing parameters report


```
"bad pose time": 0,
"block num": 1,
"colorize time": 23406,
"contour generation": {
    "generate contour": false
},
"dem generation": {
    "generate dem": false
},
"fc pose process time": 9485,
"flight param": [
    {
        "flight end time": 1712928571.1942546,
        "flight start time": 1712927736.4136157,
        "gnss name": "gnss name",
        "gnss sn": "gnss sn",
        "ground beam size": [
            179,
            17
        ],
        "height to ground": 35.8609504699707,
        "imu name": "imu name",
        "imu sn": "imu sn",
        "lidar name": "L1",
        "lidar sn": "3FCDJBJ0042C0E",
        "param between lidar and imu before optimize": {
            "relative angle pitch": 0.0046891579404473305,
            "relative angle roll": 3.123631477355957,
            "relative angle yaw": 0.009083351120352745,
            "relative distance X": 0.03508000075817108,
            "relative distance Y": 0.016940001398324966,
            "relative distance Z": -0.0464399978518486
        },
        "pulse rate": 240,
        "rtk observe interval": 0.2,
        "rtk receive freq": 5,
        "scan rate": 720,
        "speed over ground": 5.119206428527832,
        "uav name": "DJI M300 RTK",
        "uav sn": "uav sn"
    }
],
"generate b3dm": false,
"generate dem tif": false,
"generate dem tile": false,
"generate i3s": false,
"generate las": true,
"generate obj": false,
"generate osgb": false,
"generate pcd": false,
"generate ply": false,
"generate pnts": true,
"generate point ply": false,
"generate point s3mb": false,
"generate s3mb": false,
"generate xml": false,
"georef time": 17760,
"good pose time": 834780,
"ground classification time": 178216,
"ground extraction": {
    "do ground extraction": true,
    "iterator angle": 10.0,
    "iterator angle str": "10",
    "iterator distance": 0.699999988079071,
    "iterator distance str": "0.7",
    "max building size": 20.0,
    "max building size str": "20",
    "scene type": 2
},
"hardware information": {
    "cpu model": "Intel 13th Gen Intel(R) Core(TM) i9-13900KF 32 cores",
    "gpu names": [
        "NVIDIA GeForce RTX 4090"
    ],
    "gpu num": 1,
    "total ram": 130880
},
"imu traj error": {
    "traj d average": 0.005746575361986182,
```

```
    "traj d rmse": 8.422270704287834e-05,
    "traj e average": 0.0064875082495896436,
    "traj e rmse": 0.00010171273799741207,
    "traj n average": 0.005634428301387861,
    "traj n rmse": 7.815509804284694e-05,
    "traj pitch average": 0.00018619072133001113,
    "traj pitch rmse": 7.069039914853951e-06,
    "traj roll average": 0.00017849483858404 67,
    "traj roll rmse": 3.9907136293487175e-06,
    "traj yaw average": 0.0006201133028167113,
    "traj yaw rmse": 9.9150108652343e-05
},
"isGeographic": false,
"lidar point max distance": 300.0,
"lidar point max distance str": "300",
"lidar point min distance": 3.0,
"lidar point optimize": true,
"lpp time": 120047,
"merge point": false,
"output coordinate": "WGS 84 / UTM zone 31N",
"output vertical coordinate": "Default",
"point cloud collection time": 666647,
"point cloud density": {
    "average density": 1617.2244873046875,
    "scale 1 to 1000": {
        "density standard": 4.0,
        "grid size": 0.5,
        "proportion not satisfied": 0.008405305445194244,
        "total grid num": 518720.0
    },
    "scale 1 to 2000": {
        "density standard": 1.0,
        "grid size": 1.0,
        "proportion not satisfied": 0.0092595424503088,
        "total grid num": 130892.0
    },
    "scale 1 to 500": {
        "density standard": 16.0,
        "grid size": 0.25,
        "proportion not satisfied": 0.012360506691038609,
        "total grid num": 2057440.0
    }
},
"pos status": {
    "fixed solution": 1.0,
    "other solution": 0.0
},
"pose data collection time": 834780,
"process point cloud other time": 256421,
"quality level": 1,
"sample distance": 0.0,
"save preprocess results time": 0,
"save result time": 93048,
"scene mode": 0,
"smooth point": false,
"total pc optimize time": 492922,
"total process time": 671138,
"total uav process time": 834780,
"use lidar base station": false
```


A.3 TorchPoints3D installation

The following protocol was used to install TorchPoints3D on a computer running Windows 11 with an Nvidia GPU. The installation was performed using the Ubuntu distribution installed on WSL2 (Windows Subsystem for Linux).

- First, create a new user with administrative privileges. This can be done in the Ubuntu terminal:
  - Enter the command to create the user: `sudo adduser user_name`
  - Restart the terminal.
  - Then, enter the following command to make the repository accessible for reading and writing to every user: `sudo chmod a+rwx /home/user_name`
  - Use the following command to give administrator privileges to the new user: `sudo usermod -aG sudo user_name`
  - Finally, switch to this new user by using the command: `su user_name`
- Update loiciel's packages of Ubuntu with the two folowing command lines:
  - `sudo apt update`

  - `sudo apt upgrade`

- Next, install GCC/G++ versions compatible with the CUDA version being used. In this case, CUDA 11.1 requires GCC/G++ version 9. Use the following commands to install them:
  - `sudo apt install gcc g++` #Install GCC and G++
  - `sudo apt install gcc-9 g++-9` #install the 9th version of GCC and C++
  - `sudo update-alternatives -install /usr/bin/gcc gcc /usr/bin/gcc-9 100` #replace the path to GCC by the one to GCC-9 when GCC is called
  - `sudo update-alternatives -install /usr/bin/g++ g++ /usr/bin/g++-9 100`

- If you have a GPU, download the latest drivers for it. In this case, the drivers were for an NVIDIA GPU. Then, install the required CUDA version for TorchPoints3D, MinkowskiEngine, and TorchSparse (in this case, version 11.1):
  - Download the up-to-date drivers at the following address :
    https://www.nvidia.com/Download/index.aspx
  - Next, download the required CUDA version. For NVIDIA GPUs, instructions for downloading CUDA for your specific operating system can be found on the NVIDIA website. In this case, for CUDA 11.1 on WSL, the following link provides the command-line instructions for Ubuntu:
    https://developer.nvidia.com/cuda-11.1.0-download-archive?target_os=Linux&target_arch=x86_64&target_distro=WSLUbuntu&target_version=20&target_type=runfilelocal
  - add this line in the .bashrc file and restart your terminal :
    `export PATH=/usr/local/cuda/bin:$PATH`
  - It worked if the following command indicate the correct CUDA version :
    `nvcc -version`

- Install Anaconda to make the installation in a virtual environment:
  - wgethttps://repo.continuum.io/archive/Anaconda3-2023.09-0-Linux-x86_64.sh #last part can be changed by the version of Anaconda wanted, better use the latest version available
  - `bash Anaconda3-2023.09-0-Linux-x86_64.sh`
  - restart Ubuntu

- Create an anaconda environment that uses the Python version needed by TorchPoints3D and MinkowskiEngine. In this case Python 3.8:
  - `conda create -n myenv python=3.8`
  - `conda activate myenv`
  - `conda update -n base -c defaults conda`
  - `conda install openblas-devel -c anaconda`
  - `sudo apt-get install libopenblas-dev`

- Install the version of PyTorch needed by Minkowski Engine, TorchSparse and TorchPoints3D, and the versions of TorchVision and TorchAudio needed by this version of PyTorch. It is needed to precise the version of CUDA used. The command to install the PyTorch version wanted along with torch audio and TorchVision and with the CUDA version used, can be found on the site : https://pytorch.org/get-started/previous-versions/
  `pip install torch==1.9.0+cu111 torchvision==0.10.0+cu111`
  `torchaudio==0.9.0`
  `-f https://download.pytorch.org/whl/torch_stable.html`

- Installation of TorchSparse. First download the wheel for the version of
  TorchSparse corresponding to the version of CUDA, Torch and Python (here CUDA 11.1, Torch 1.9 and Python 3.8). This wheel can be downloaded at the following : http://pypi.hanlab.ai/simple/torchsparse/. Then TorchSparse can be installed with : `pip install`
  `torchsparse-2.1.0+torch19cu111-cp38-cp38-linux_x86_64.whl`

- Installation of Minkowski Engine
  `MAX_JOBS=1 CXX=g++ pip install`
  `-U git+https://github.com/NVIDIA/MinkowskiEngine -v`
  `-no-deps -global-option="-blas_include_dirs=`
  `$CONDA_PREFIX/include" -global-option="-blas=openblas"`
  `-no-binary :all`

- Install of torch-points3d
  `pip install torch-points3d`

- There can be some error messages linked to some libraries, if it is the case check the needed versions written in the requirements.txt file in the git repository and install them with the command here under, then reinstall torch-points3d:
  `pip install LIBRARY==VERSION`

- There are some libraries needed and not installed by torch-points3d:
  - `pip install pycuda`
  - `pip install laspy`
  - `pip install h5py`
  - `pip install hydra-core==1.0.7`

- To verify if the installation worked correctly, install the git repository of torch-points3d and try to run the train function:
  - `git clone https://github.com/nicolas-chaulet/torch-points3d.git`
  - `cd torch-points3d/`
  - Rename or suppress the torch-points3d file in it, otherwise it will be used instead of the library installed before and can cause problems
  - `python -m unittest -v`

Once this is done, TorchPoints3D should be correctly installed, and the model can be trained using it. However, training such models can quickly consume a lot of memory. Since this is being done on WSL2, it's important to note that the disk space allocated for WSL2 automatically increases as files are added, but it does not automatically decrease when files are deleted. Therefore, if memory becomes limited and you decide to delete files, the following steps need to be taken once the files on WSL are deleted:

- Launch PowerShell
- Shutdown all WSL instance running by using the command `wsl -shutdown`
- Launch the program Dskpart by using the command `diskpart` in PowerShell
- Search in your main disk for the file "ext4.vhdx". It should be located at the following pass `/Users/valorin/AppData/Local/Packages/`*`folder containing the chosen Linux distribution`*`/LocalState/ext4.vhdx`
- Use the following command in DiskPart: `select vdisk file="Path found at the previous step"`
- Use the following command in DiskPart: `compact vdisk`

A.4 Python code to cut a 'las' file into 30m sub-las

```
import numpy as np
import laspy
import os

def create_grid(x_min, x_max, y_min, y_max, cell_size):
    x_grid = np.arange(x_min, x_max, cell_size)
    y_grid = np.arange(y_min, y_max, cell_size)
    return x_grid, y_grid

def split_pc(las_file, x_grid, y_grid, output_dir):
    pc = laspy.read(las_file)
    scene_nbr = 1
    for i in range(len(x_grid)-1):
        for j in range(len(y_grid)-2):
            x_min, x_max = x_grid[i], x_grid[i+1]
            y_min, y_max = y_grid[j], y_grid[j+1]

            indices = np.where((pc.x >= x_min) & (pc.x < x_max) & (pc.y >= y_min) & (pc.y < y_max))[0]

            #indices = np.where((pc[indices].y >= y_min) & (pc[indices].y < y_max))[0]
            if(len(indices) < 1):
                continue
            scene_dir = os.path.join(output_dir,f"Scene_{scene_nbr}")
            os.makedirs(scene_dir, exist_ok=True)

            outFile = laspy.create(point_format=pc.header.point_format, file_version=pc.header.version )
            outFile.header = pc.header
            outFile.points = pc.points[indices]
            outFile.write(os.path.join(scene_dir, f"subcloud_{i}_{j}.las"))
            scene_nbr += 1

las_file = "data\Sart\without_ground.las"
test = 'test.las'
output_root = "data/Sart/divided"
cell_size = 30

pc = laspy.read(las_file)
"""pc2 = laspy.read(test)
pc2.header =
print(pc2.vlrs)
exit()
outFile = laspy.create(point_format=pc.header.point_format, file_version=pc.header.version )
outFile.points = pc.points
outFile.header = pc.header
outFile.write('test.las')
pc2 = laspy.read(test)
print(pc2.x)
exit()"""
x_min, x_max = pc.x.min(), pc.x.max()
y_min, y_max = pc.y.min(), pc.y.max()

x_grid, y_grid = create_grid(x_min, x_max, y_min, y_max, cell_size)

split_pc(las_file,x_grid, y_grid,output_root)
```

A.5 Creation of "json" file, containing bounding boxes, from point clouds of individual trees

```
import numpy as np
import json
import laspy
import os

bbox_file_name = "bbox.json"
las_directory = "data/Sart/divided"

def get_bbox(pc):
    wx = pc.x.max() - pc.x.min()
    wy = pc.y.max() - pc.y.min()
    h = pc.z.max() - pc.z.min()
    cx = pc.x.min() + (wx/2)
    cy = pc.y.min() + (wy/2)
    cz = pc.z.min() + (h/2)
    return cx, cy, cz, wx, wy ,h

for dir_name in os.listdir(las_directory):
    dir = os.path.join(las_directory,dir_name)
    bbox = {}
    for i, las in enumerate(os.listdir(dir)):
        if las.endswith('.las') and las.startswith('tree'):
            las_file = os.path.join(dir,las)
            pc = laspy.read(las_file)
            cx, cy, cz, wx, wy ,h = get_bbox(pc)
            bbox[i] = {}
            bbox[i]['coords'] = {}
            bbox[i]['coords']['centre_x'] = cx
            bbox[i]['coords']['centre_y'] = cy
            bbox[i]['coords']['centre_z'] = cz
            bbox[i]['coords']['largeur'] = wx
            bbox[i]['coords']['longueur'] = wy
            bbox[i]['coords']['hauteur'] = h
    bbox_dir = os.path.join(dir, bbox_file_name)
    with open(bbox_dir, 'w') as f:
        json.dump(bbox, f)
```

A.6 Dataset class for TorchPoints3D

```
import os
import numpy as np
import torch
import json
import logging
from torch_geometric.data import InMemoryDataset
from torch_points3d.metrics.object_detection_tracker
import ObjectDetectionTracker
from torch_points3d.datasets.base_dataset
import BaseDataset, save_used_properties
from torch_points3d.utils.box_utils import box_corners_from_param
from torch_geometric.data import Data, InMemoryDataset
import torch_points3d.core.data_transform as cT
import laspy
#from torch_points3d.datasets.segmentation import ScannetDataset
import random
from omegaconf import OmegaConf
import hydra
from hydra.core.global_hydra import GlobalHydra
from torch_points3d.models.object_detection.votenet
import VoteNetModel
from torch_points3d.models.model_factory import instantiate_model
from torch_points3d.datasets.object_detection.scannet
import ScannetDataset

class Sart(InMemoryDataset):
    SPLITS = ['train','val','test']
    TRAIN_SCANS = None
    VAL_SCANS = None
    TEST_SCANS = None
    TRAIN_PART = 0.7
    TEST_PART = 0.1
    VAL_PART = 0.2
    MAX_NUM_OBJ = 70
    #pour definir ce dataset il faut definir un certain nombre
    #d hypothese sur mon futu dataset:
    #   -Type de fichier pour le nuage de points: ici on va partir
    #    sur du las => si modification, modif fonction read_point
    #   -Definir si points possedes des valeurs RGB ou juste
    #    les coord x y z
    #   -Type de fichier pour les bounding box: ici on va partir
    #    sur du json => si modification, modif fonction read_bbox
    #   -Maniere d'encoder les bbox : ici on va partir sur 7
    #    valeurs (x centre, y centre, z centre, largeur(x),
    #    longueur (y), hauteur(z), label)

    def __init__(
        self,
        root,
        split="train",
        transform=None,
        pre_transform=None,
        pre_filter=None,
        process_workers=1,
        is_test=False,
        max_num_point = None,
    ):

        #root = racine du dossier contenant les fichiers
        #(separe en "raw_dir" et "processed_dir")
        self.mean_size_array = []
        self.is_test = is_test
        self.split = split
        super(Sart,self).__init__(root,transform,pre_transform)
        if split == "train":
            path = self.processed_paths[0]
        elif split == "val":
            path = self.processed_paths[1]
        elif split == "test":
            path = self.processed_paths[2]#2
        else:
            raise ValueError((f"Split {split} found, but expected
            either " "train, val, or test"))
        self.data, self.slices = torch.load(path)
        self.mean_size_array = np.load(os.path.join(self.processed_dir,
        'mean_size_array.npy'))

    def __getitem__(self, idx):
        """
```

```
    Data object contains:
        pos - points
        x - features
    """
    if not isinstance(idx, int):
        raise ValueError("Only integer indices supported")

    # Get raw data and apply transforms
    data = super().__getitem__(idx)

    # Extract instance and box labels
    self._set_extra_labels(data)
    data.pos = data.pos.float()
    return data

def raw_file_names(self):
    #cherche le fichier/dossier contenant le dataset,
    #si n'existe pas le telecharge (pour dataset
    #en ligne pas pour le notre)
    return ["scans"]
@property
def processed_file_names(self):
    #Si ces fichiers sont trouves, l etape de
    #transformation est skip
    return [
        "{}.pt".format(
            s,
        )
        for s in Sart.SPLITS
    ]

def download(self):
    pass

def set_scan_repartition(self,scan_paths):
    n = len(scan_paths)
    paths = list(range(n))
    if self.split != 'test':
        Sart.TRAIN_SCANS = np.array(random.sample(paths, n-1))
        paths = [path for path in paths if path not in Sart.TRAIN_SCANS]
        Sart.VAL_SCANS = [path for path in paths if path not
        in Sart.TRAIN_SCANS]

"""def set_scan_repartition(self,scan_paths):
    n = len(scan_paths)
    paths = list(range(n))
    if self.split != 'test':
        Sart.TRAIN_SCANS = np.array(random.sample(paths,
        int(Sart.TRAIN_PART * n)))
        paths = [path for path in paths if path not in
        Sart.TRAIN_SCANS]
        #Test.TEST_SCANS = np.array(random.sample(paths,
        #int(Sart.TEST_PART * n)))
        Sart.VAL_SCANS = [path for path in paths if path not in
        Sart.TRAIN_SCANS]
    else:
        Sart.TEST_SCANS = paths"""

"""def get_scan_paths(self):
    path = self.raw_dir
    path = os.path.join(path,'scans')
    scan_paths = []
    for dir in os.listdir(path):
        scan_path = os.path.join(path,dir)
        if os.path.isdir(scan_path) and dir.startswith("Scene_"):
            scan_paths.append(scan_path)
    if len(scan_paths) < 1:
        print("No scan found. Make sure there are files named scene...
        in your raw dataset scans directory (/data/your dataset/raw/scans/)")
        exit()
    #ici on definit les indices des elements qui seront utilise
    #pour l entrainement, test et validation
    if self.split == 'train':
        self.set_scan_repartition(scan_paths)
        train_paths = [scan_paths[i] for i in Sart.TRAIN_SCANS]
        #val_paths = [scan_paths[i] for i in Sart.VAL_SCANS]
        #test_paths = [scan_paths[i] for i in Test.TEST_SCANS]
        self.scan_paths = [train_paths,[],[] ]
    else:
        self.set_scan_repartition(scan_paths)
        val_paths = [scan_paths[i] for i in Sart.VAL_SCANS]
        self.scan_paths = [[],val_paths,[]]"""

def get_scan_paths(self):
    path = self.raw_dir
```

```
    path = os.path.join(path,'scans')
    scan_paths = []
    for dir in os.listdir(path):
        scan_path = os.path.join(path,dir)
        if os.path.isdir(scan_path) and dir.startswith("Scene_"):
            scan_paths.append(scan_path)
    if len(scan_paths) < 1:
        print("No scan found. Make sure there are files named scene...
        in your raw dataset scans directory (/data/your dataset/raw/scans/)")
        exit()
    #ici on definit les indices des elements qui seront utilise pour
    #l entrainement, test et validation
    if self.split != 'test':
        train_paths = []
        val_paths = []
        for path in scan_paths:
            if path.endswith('Scene_18'):
                val_paths.append(path)
            else :
                train_paths.append(path)
        #test_paths = [scan_paths[i] for i in Test.TEST_SCANS]
        self.scan_paths = [train_paths,val_paths,[] ]
    else:
        test_paths = scan_paths
        self.scan_paths = [[],[], test_paths]


def read_points(self, laz_path):
    laz_file = laspy.read(laz_path)
    coords = laz_file.xyz
    r = (laz_file.red)/255
    g = (laz_file.green)/255
    b = (laz_file.blue)/255
    rgb = np.column_stack((r,g,b))/255

    return coords,rgb

def read_bbox(self, bbox_path):
    with open(bbox_path,"r") as bbox_file:
        bbox_json = json.load(bbox_file)
        #il faut stocker les coord de la bbox dans un array
        #de taille (m,6) avec m le nombre d'objet dans le scan
        #egalement stocker l id des objets pour pouvoir infine
        #trouver un moyen d associer a chaque point l id d
        #un objet. Regarder pour faire un fichier
        #json ou txt qui contient comme pour scannet un array
        #dont la longueur = nombre point scan et pour chacun indiquer l objet auquel il #appartient.
        bbox = np.zeros((len(bbox_json),6))
        i=0
        mean_size_array = []
        for key, value in bbox_json.items():
            coords = value["coords"]
            mean_size_array.append([coords["largeur"],coords["longueur"],coords["hauteur"]])
            bbox[i] = [coords["centre_x"],coords["centre_y"],coords["centre_z"],
            coords["largeur"],coords["longueur"],coords["hauteur"]]
            i+=1
        return bbox, mean_size_array
def labelise(self,coords, bboxes):
    instance_labels = np.zeros((len(coords)), dtype=int)
    semantic_labels = np.zeros((len(coords)), dtype = int)
    for i,coord in enumerate(coords):
        for j,bbox in enumerate(bboxes):
            x_max = bbox[0] + (bbox[3]/2)
            x_min = bbox[0] - (bbox[3]/2)
            y_max = bbox[1] + (bbox[4]/2)
            y_min = bbox[1] - (bbox[4]/2)
            z_max = bbox[2] + (bbox[5]/2)
            z_min = bbox[2] - (bbox[5]/2)
            if coord[0]>= x_min and coord[0] <= x_max:
                if coord[1]>= y_min and coord[1] <= y_max:
                    if coord[2]>= z_min and coord[2] <= z_max:
                        instance_labels[i] = j+1
                        semantic_labels[i] = 1
                        break
    return instance_labels, semantic_labels

def norm(self,points, bbox, mean_size_array):
    centroid = np.mean(points, axis=0)
    points -= centroid
    bbox[:,0] = (bbox[:,0] - centroid[0])
    bbox[:,1] = (bbox[:,1] - centroid[1])
    bbox[:,2] = (bbox[:,2] - centroid[2])
    mean_size_array = mean_size_array
    transfo_params = [centroid]
```

```
    return points, bbox, transfo_params, mean_size_array


def read_one_scan(self,scan_path,bbox_path, normalize = True):
    data = {}
    coord, rgb = self.read_points(scan_path)
    bbox, mean_size_array = self.read_bbox(bbox_path)
    instance_labels, y = self.labelise(coord, bbox)
    if normalize:
        coord, bbox, tparams, mean_size_array = self.norm(coord, bbox, mean_size_array)
    self.mean_size_array.extend(mean_size_array)
    data['pos'] = torch.from_numpy(coord[:,:3]).float()
    data['instance_bboxes'] = torch.from_numpy(bbox)
    data['instance_labels'] = torch.from_numpy(instance_labels)
    data['rgb'] = torch.from_numpy(rgb)
    data['y'] = torch.from_numpy(y)
    data['x'] = None

    return Data(**data)

def _download(self):
    return
def process(self):
#Cette implementation se base sur une architecture ou dans le raw_dir, il y a une liste de
#dossier, dans chacun de ces dossiers il y a un (et un
# seul) scan au format laz et les bouding box de ce scan au format json
    self.get_scan_paths()
    for i, (scan_paths, split) in enumerate(zip(self.scan_paths,Sart.SPLITS)):
        if len(scan_paths) == 0:
            continue
        if not os.path.exists(self.processed_paths[i]):
            datas = []
            if isinstance(scan_paths, str):
                scan_paths = [scan_paths]
            for id, path in enumerate(scan_paths):
                files = os.listdir(path)
                laz_file = [file for file in files if file.endswith('.las')]
                bbox_file = [file for file in files if file.endswith('.json')]
                if len(laz_file) != 1 or len(bbox_file) !=1 :
                    print("There must be only ONE laz file and ONE bbox file in a scan directory")
                    return
                laz_path = os.path.join(path,laz_file[0])
                bbox_path = os.path.join(path,bbox_file[0])
                data = self.read_one_scan(laz_path, bbox_path)
                data['id_scan']=torch.tensor([id]).int()
                datas.append(cT.SaveOriginalPosId()(data))

            if self.pre_transform:
                datas = [self.pre_transform(data) for data in datas]
            if self.split != 'test':#A modifier pour que ca soit plus propre
                torch.save(self.collate(datas), self.processed_paths[i])
            else:
                torch.save(self.collate(datas), self.processed_paths[2])
    if not os.path.exists(os.path.join(self.processed_dir, 'mean_size_array.npy')):
        self.mean_size_array = np.mean(self.mean_size_array,axis=0)
        self.mean_size_array = np.vstack((self.mean_size_array, self.mean_size_array))
        np.save(os.path.join(self.processed_dir, 'mean_size_array.npy'), self.mean_size_array)

def _set_extra_labels(self, data):

    num_points = data.pos.shape[0]
    semantic_labels = data.y
    instance_labels = data.instance_labels

    center_label = torch.zeros((Sart.MAX_NUM_OBJ, 3))
    target_bboxes_mask = torch.zeros((Sart.MAX_NUM_OBJ), dtype=torch.bool)
    angle_residuals = torch.zeros((Sart.MAX_NUM_OBJ,))
    size_classes = torch.zeros((Sart.MAX_NUM_OBJ,))
    size_residuals = torch.zeros((Sart.MAX_NUM_OBJ, 3))
    point_votes = torch.zeros([num_points, 3])
    point_votes_mask = torch.zeros(num_points, dtype=torch.bool)
    instance_box_corners = []
    box_sizes = []
    centers = []
    instance_classes = []

    for i_instance in np.unique(instance_labels):
        ind = np.where(instance_labels == i_instance)[0]
        instance_class = semantic_labels[ind[0]].item()
        pos = data.pos[ind, :3]
        max_pox = pos.max(0)[0]
        min_pos = pos.min(0)[0]
        center = 0.5 * (min_pos + max_pox)
        point_votes[ind, :] = center - pos
        point_votes_mask[ind] = True
```

```
            box_size = max_pox - min_pos
            instance_box_corners.append(box_corners_from_param(box_size, 0, center))
            box_sizes.append(box_size)
            centers.append(center)
            instance_classes.append(instance_class)
        point_votes = point_votes.repeat((1, 3))
        instance_classes = torch.tensor(instance_classes).int()

        num_instances = len(instance_classes)
        target_bboxes_mask[0:num_instances] = True
        target_bboxes_semcls = np.zeros((Sart.MAX_NUM_OBJ))
        target_bboxes_semcls[0:num_instances] = instance_classes

        size_classes[0:num_instances] = 1

        if num_instances > 0:
            box_sizes = torch.stack(box_sizes)
            centers = torch.stack(centers)
            size_residuals[0:num_instances, :] = box_sizes - torch.from_numpy(self.mean_size_array[instance_classes, :])
            center_label[0:num_instances, :] = centers

        data.center_label = center_label
        data.heading_class_label = torch.zeros((self.MAX_NUM_OBJ,))
        data.heading_residual_label = angle_residuals.float()
        data.size_class_label = size_classes
        data.size_residual_label = size_residuals.float()
        data.sem_cls_label = torch.from_numpy(target_bboxes_semcls).int()
        data.box_label_mask = target_bboxes_mask
        data.vote_label = point_votes.float()
        data.vote_label_mask = point_votes_mask
        data.instance_box_corners = torch.zeros((self.MAX_NUM_OBJ, 8, 3))
        if len(instance_box_corners):
            data.instance_box_corners[: len(instance_box_corners), :, :] = torch.stack(instance_box_corners)

        delattr(data, "instance_bboxes")
        delattr(data, "instance_labels")
        return data

class SartDataset(BaseDataset):
    SPLITS = ['train','val','test']

    def __init__(self,dataset_opt, is_test = False):
        super().__init__(dataset_opt)
        process_workers: int = dataset_opt.process_workers if dataset_opt.process_workers else 0
        is_test: bool = dataset_opt.is_test if dataset_opt.is_test is not None else False
        max_num_point: int = dataset_opt.get('max_num_point', None)

        self.train_dataset = Sart(
            self._data_path,
            split="train",
            pre_transform=self.pre_transform,
            transform=self.train_transform,
            is_test=is_test,
            max_num_point=max_num_point,
        )
        self.val_dataset = Sart(
            self._data_path,
            split="val",
            pre_transform=self.pre_transform,
            transform=self.val_transform,
            is_test=is_test,
            max_num_point=max_num_point,
        )
        """self.test_dataset = Sart(
            self._data_path+"/test",
            split="test",
            pre_transform=self.pre_transform,
            transform=self.test_transform,
            is_test=is_test,
            max_num_point=max_num_point,
        )"""

    @property  # type: ignore
    @save_used_properties
    def mean_size_arr(self):
        return self.train_dataset.mean_size_array.copy()

    def class2angle(self, pred_cls, residual, to_label_format=True):
        """ Inverse function to angle2class.
        As ScanNet only has axis-alined boxes so angles are always 0. """
        return 0

    @property  # type: ignore
    @save_used_properties
    def num_classes(self):
```

```
    return 2

def class2size(self, pred_cls, residual):
    """ Inverse function to size2class """
    if torch.is_tensor(residual):
        mean = torch.tensor(self.mean_size_arr[pred_cls, :]).to(residual.device)
    else:
        mean = self.mean_size_arr[pred_cls, :]
    return mean + residual

def get_tracker(self, wandb_log: bool, tensorboard_log: bool):
    """Factory method for the tracker

    Arguments:
        wandb_log - Log using weight and biases
    Returns:
        [BaseTracker] -- tracker
    """
    return ObjectDetectionTracker(self, wandb_log=wandb_log, use_tensorboard=tensorboard_log)
```

A.7 YAML file for the dataset class

```
# @package data
defaults:
  - object_detection/default
class: sart_dataset.SartDataset
dataset_name: "sart"
task: object_detection
dataroot: data
use_instance_labels: True
use_instance_bboxes: True
donotcare_class_ids: []
process_workers: 8

test_transform:
  - transform: XYZFeature
    params:
      add_x: True
      add_y: True
      add_z: True
  - transform: FixedPoints
    lparams: [500000]
    params:
      replace: False
      allow_duplicates: True
  - transform: AddFeatsByKeys
    params:
      list_add_to_x: [True, True, True]
      feat_names: ["pos_x", "pos_y", "pos_z"]
      delete_feats: [True, True, True]
train_transform:
  - transform: XYZFeature
    params:
      add_x: True
      add_y: True
      add_z: True
  - transform: FixedPoints
    lparams: [500000]
    params:
      replace: False
      allow_duplicates: True
  - transform: RandomScaleAnisotropic
    params:
      scales: [0.9, 1.1]
  - transform: RandomSymmetry
    params:
      axis: [True, False, False]
```

```
  - transform: Random3AxisRotation
    params:
      rot_x: 2
      rot_y: 2
      rot_z: 0
  - transform: AddFeatsByKeys
    params:
      list_add_to_x: [True, True, True]
      feat_names: ["pos_x", "pos_y", "pos_z"]
      delete_feats: [True, True, True]
val_transform:
  - transform: XYZFeature
    params:
      add_x: True
      add_y: True
      add_z: True
  - transform: FixedPoints
    lparams: [500000]
    params:
      replace: False
      allow_duplicates: True
  - transform: AddFeatsByKeys
    params:
      list_add_to_x: [True, True, True]
      feat_names: ["pos_x", "pos_y", "pos_z"]
      delete_feats: [True, True, True]
```

A.8 Experiments losses

A.8.1 Basic configuration

Configuration

- Number of points: 50 000
- Batch size: 8
- Learning rate: 10^{-2}
- Weights initialization: random
- Optimization algorithm: Adam

Training

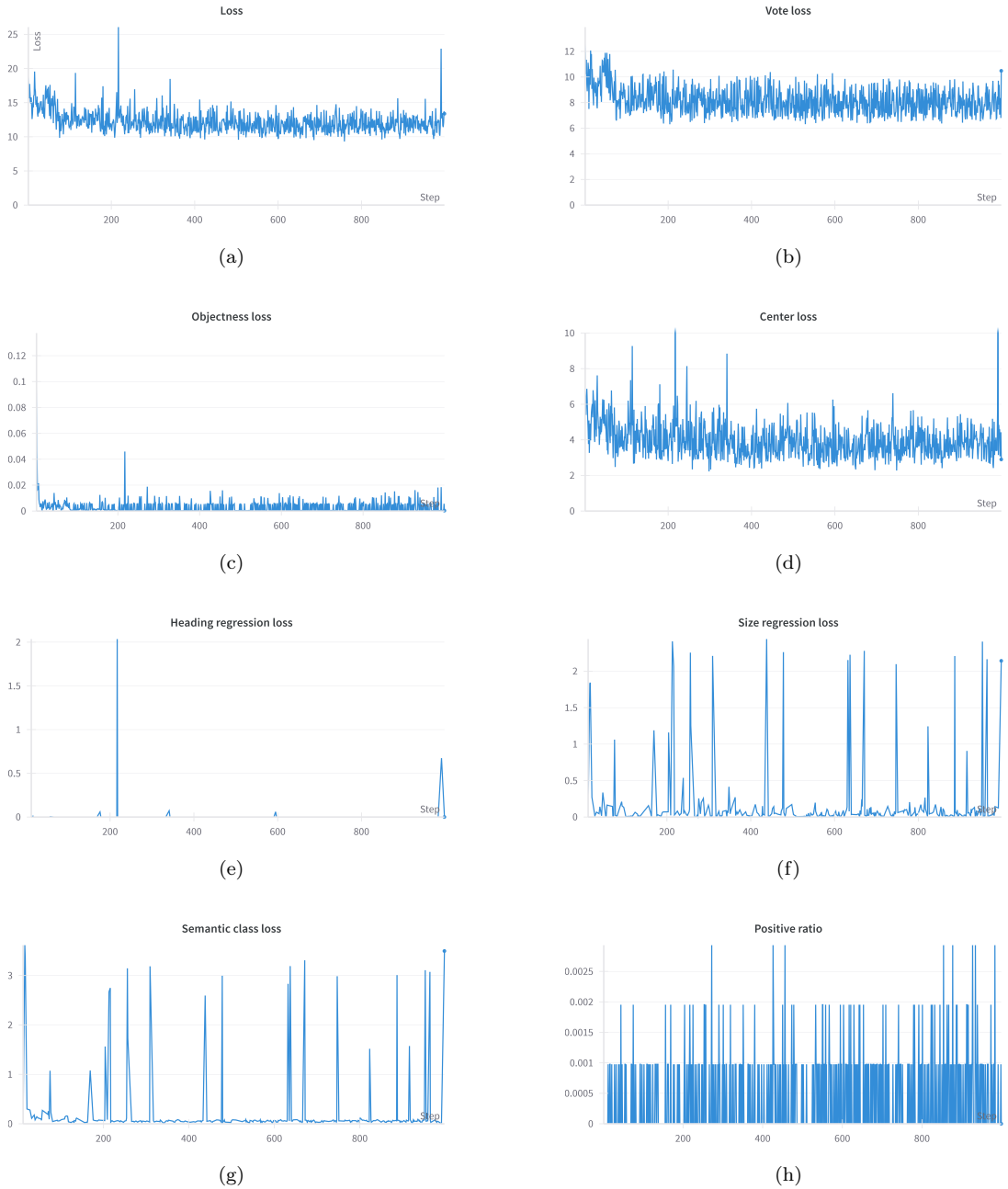


Figure a.2: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed with the basic configuration over the validation dataset

Validation

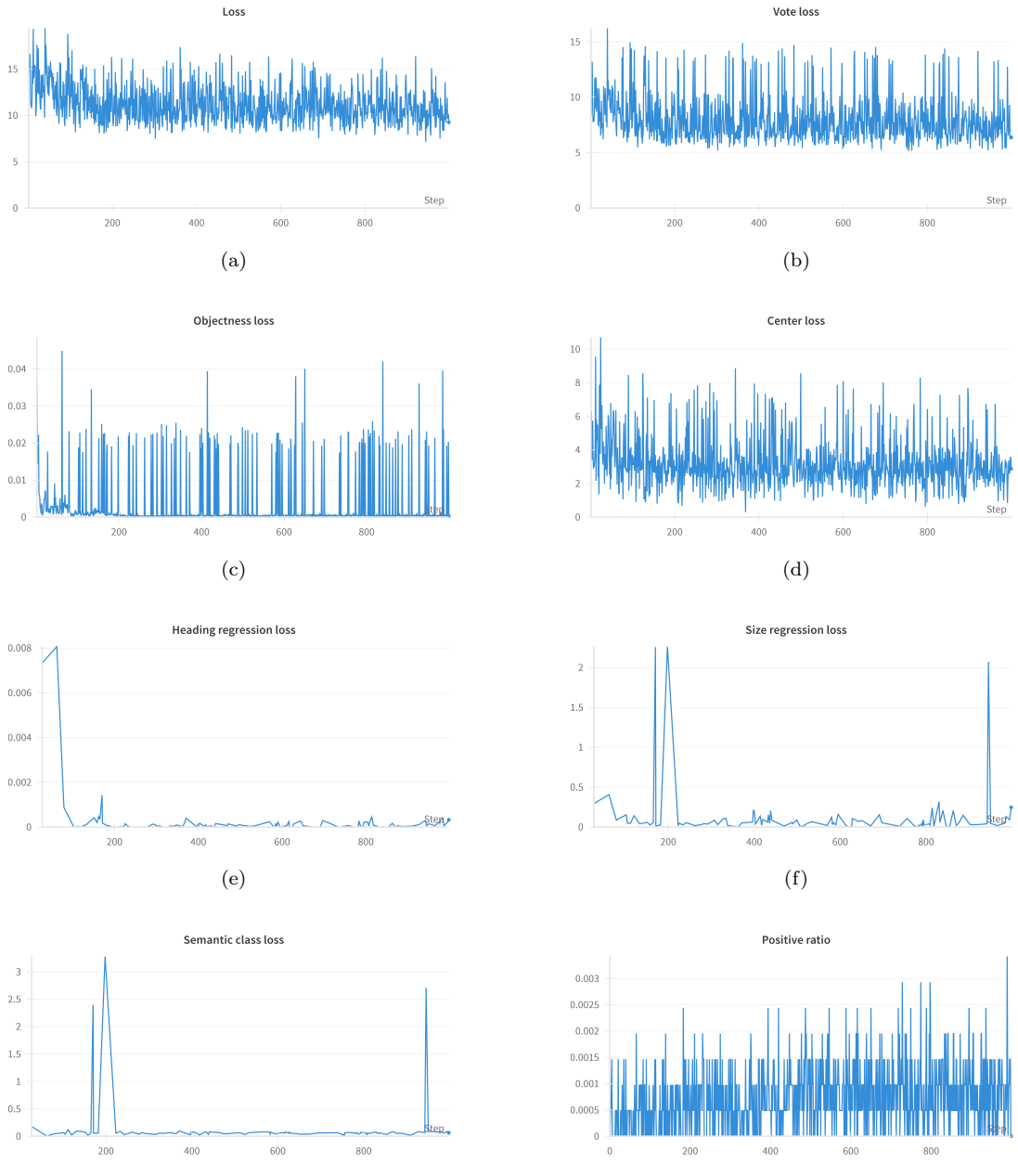


Figure a.3: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed with the basic configuration over the validation dataset

A.8.2 Number of points experiment

Configuration

- Number of points: 500 000
- Batch size: 8
- Learning rate: 10^{-2}

- Weights initialization: random

- Optimization algorithm: Adam

Training

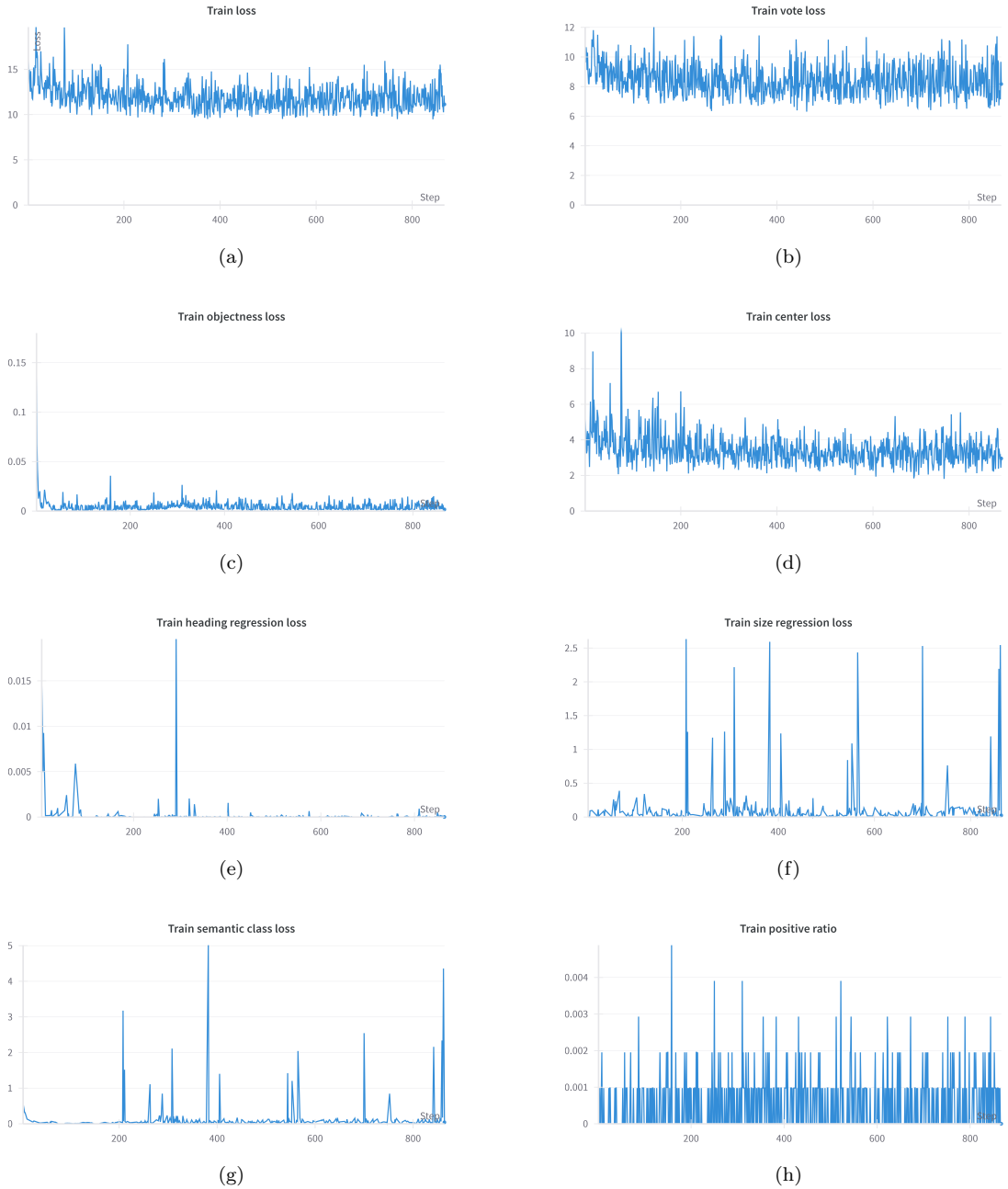


Figure a.4: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed with the basic configuration over the validation dataset for a higher number of input points

Validation

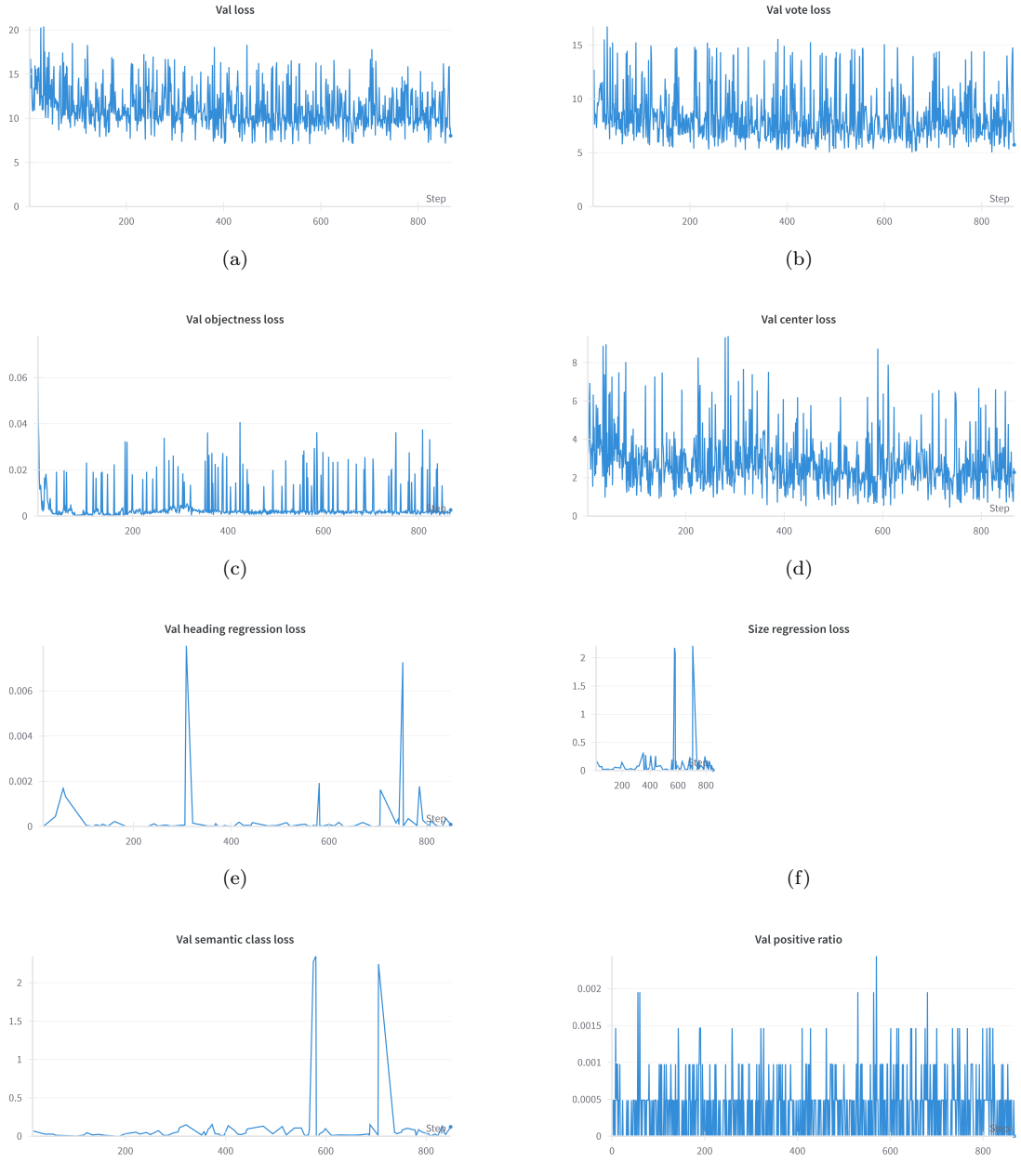


Figure a.5: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed over the validation dataset for a higher number of input points

A.8.3 Initialization experiment

Configuration

- Number of points: 500 000
- Batch size: 8
- Learning rate: 10^{-2}

- Weights initialization: Pretrained weights

- Optimization algorithm: Adam

Training

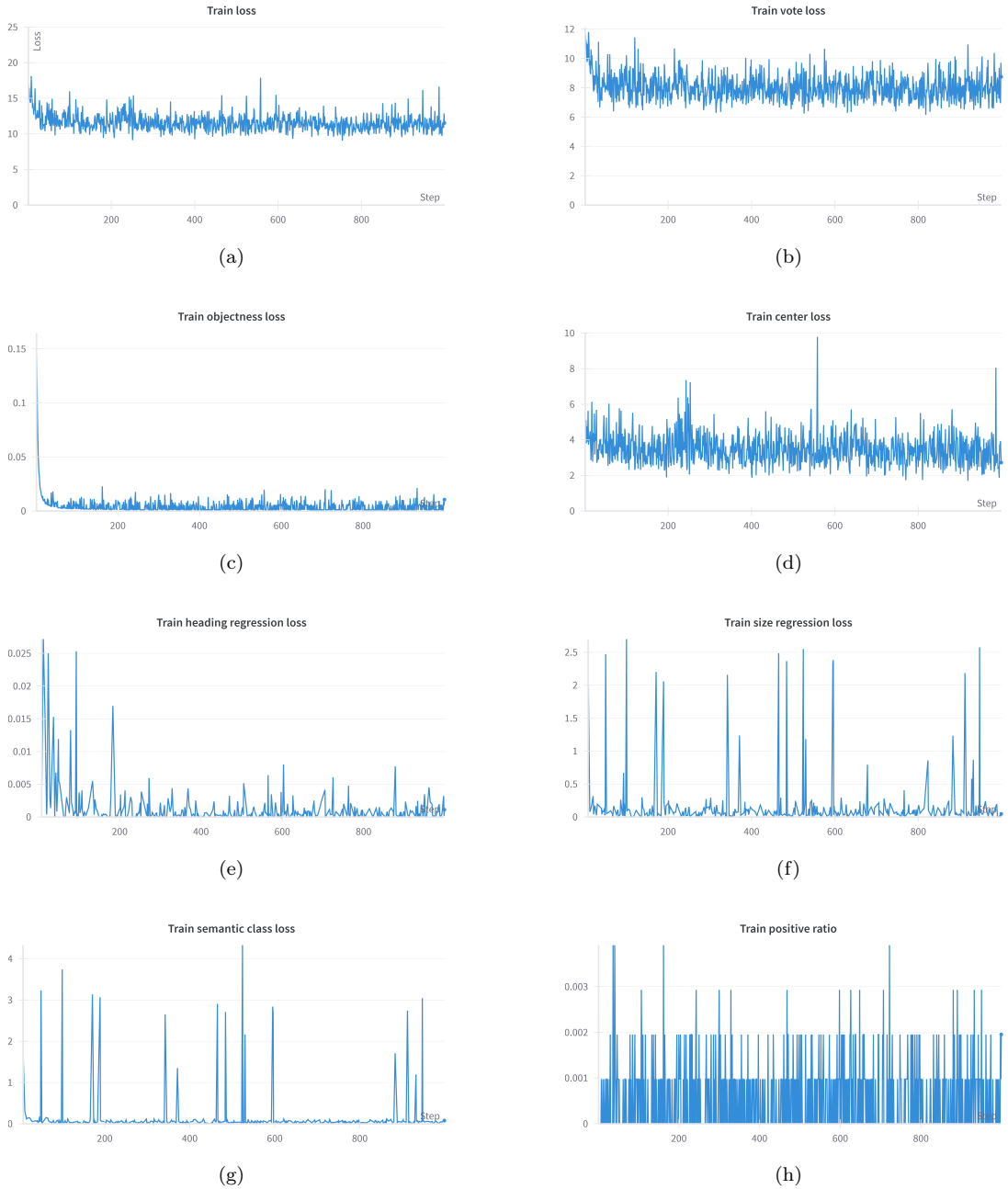


Figure a.6: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed with the basic configuration over the validation dataset for 500 000 input points

Validation

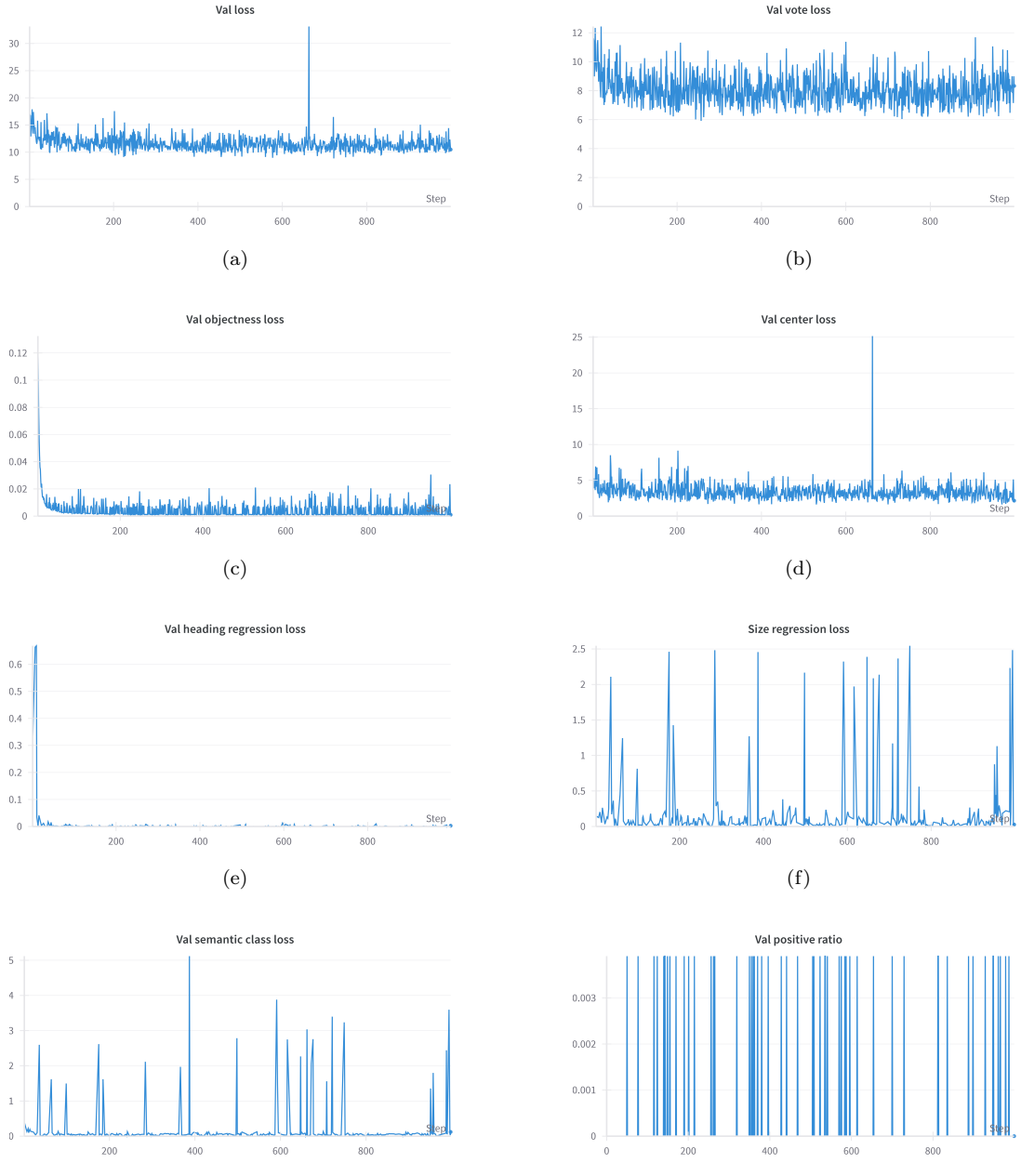


Figure a.7: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed over the validation dataset for 500 000 input points

A.8.4 Batch size experiment

Configuration

- Number of points: 500 000
- Batch size: 24
- Learning rate: 10^{-2}

- Weights initialization: Pretrained weights

- Optimization algorithm: Adam

Training

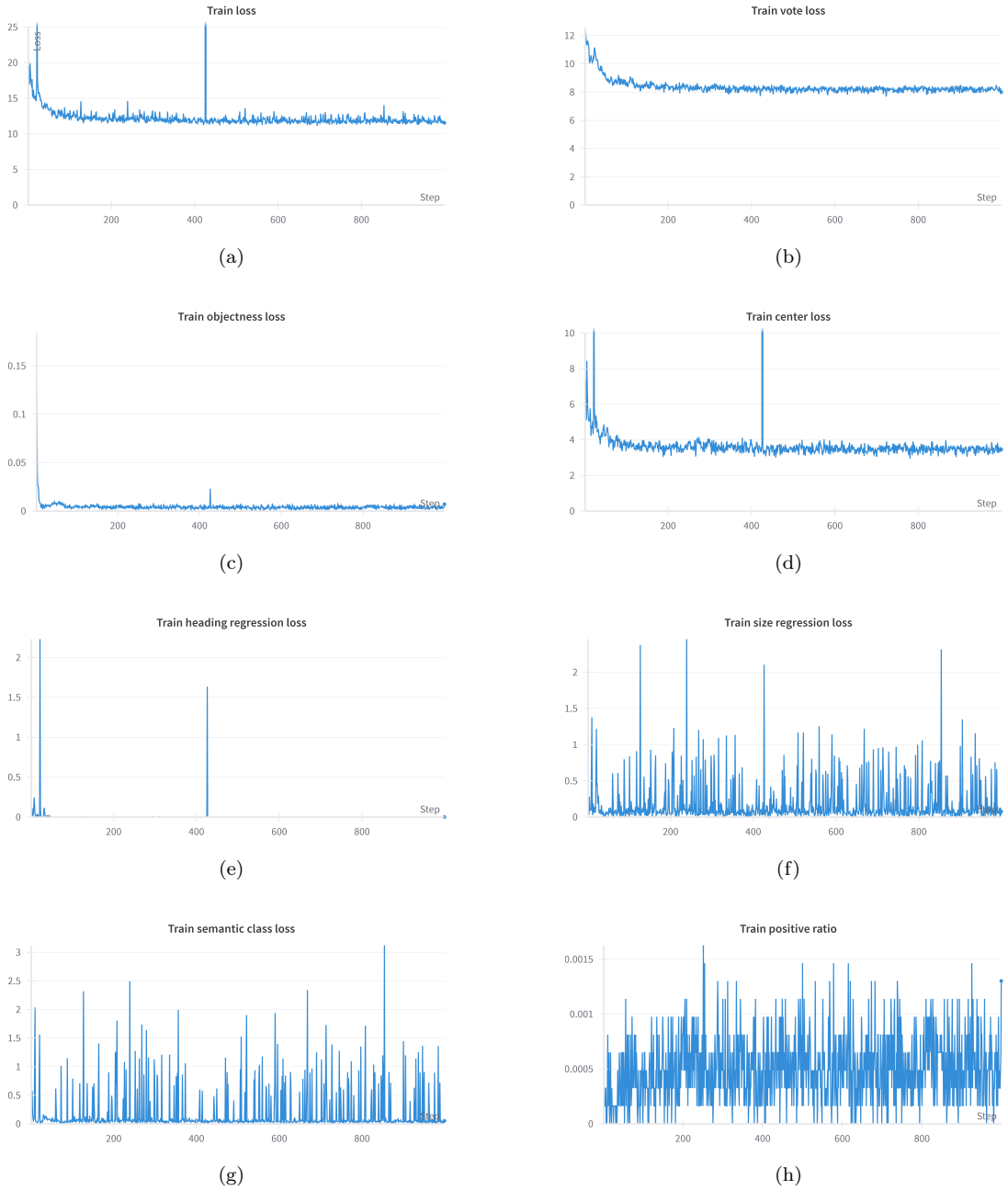


Figure a.8: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed with the basic configuration over the validation dataset for a batch size of 24

Validation

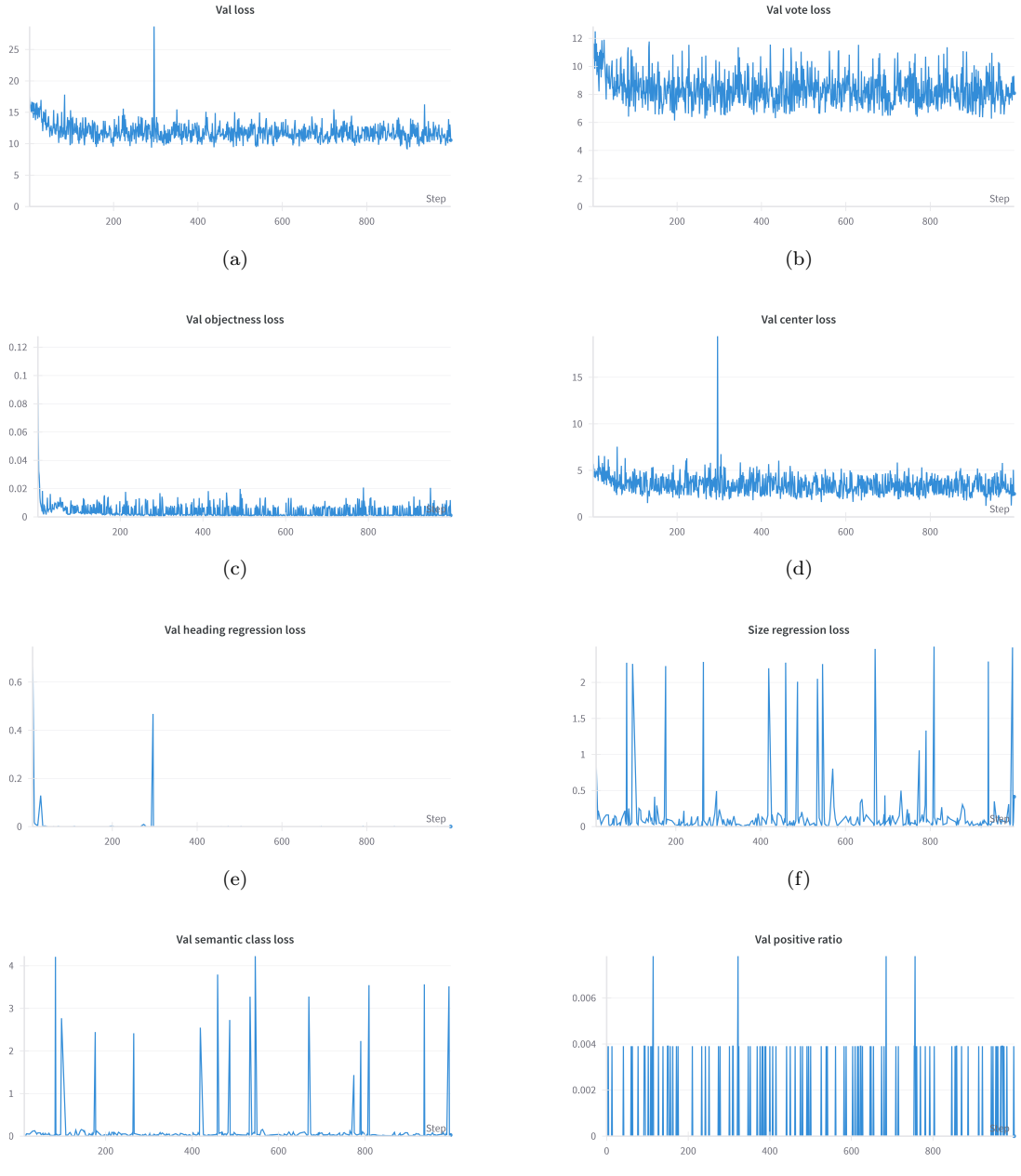


Figure a.9: General loss (a), vote loss (b), objectness loss (c), box center loss (d), heading regression loss (e), size regression loss (f), semantic class loss (g) and positive ratio (h) computed over the validation dataset for a batch size of 24