Ecole Polytechnique Fédérale de Lausanne

Master in Electrical and Electronics Engineering

Master thesis

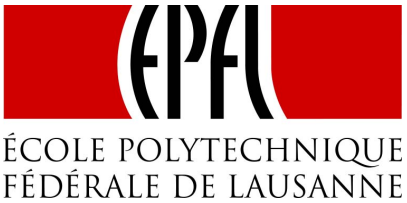
Deep learning on graph for semantic segmentation of point cloud

Author:
Alexandre Cherqui

Supervisors:
Michaël Defferrard, LTS2, EPFL
Frank De Morsier, Picterra

Carried out in the Signal Processing Laboratory 2 (LTS2), at EPFL

July 2, 2018



Contents

1 Introduction

1.1 Motivation

In order to study the environment, geospatial data need to be acquired. These latter can be used to survey the territory and thus prevent deforestation or fight some dangers such as fires for instance. To do so, one can collect data directly on chosen places. But doing so is not efficient and can technically be done only in a reduced number of spots.

Thanks to the development of technology, we can now also collect these data in an automatic fashion: some aerial images can be taken from satellites or drones since recently. Then, all objects in these data have to be identified and labelled. Because this task is not always obvious, photogrammetry can be used to combine images taken from different points of view and thus create a 3D representation of the environnement. Figure 1.1 illustrates this process. Because the resulting 3D map contains more information (heights of objects, 3D structures, hidden objects, ...), it can then help to perform the task.

Figure 1.1: Illustration of the photogrammetry principle. Different images taken from different points of view enable to build a 3D map.

But considering the amount of data, it remains a very fastidious task. Therefore, it seems that finding a more efficient and automatic way to perform the labelling is needed. That's why the startup Picterra asked for a collaboration with the LTS2 at Ecole Polytechnique Fédérale de Lausanne (EPFL). They would like to develop automatic methods in order to label each sample of a 3D point cloud obtained by photogrammetry so that points from power lines and pylons can be separated from the vegetation and the ground.

1.2 The semantic segmentation problem

Machine learning can be used for the automatic classification of data. Indeed, from well chosen extracted features, classifiers can be trained to perform the labelling task. Deep learning refers to the set of machine learning methods which are based on artificial neural networks. Unlike other methods, these latters can learn to extract relevant features from row data in addition to the task of data classification. Because deep learning has been able to provide better results than other machine learning methods so far, many deep models have been developped to tackle more and more complex problems.

Thus, deep architectures were first used to solve the problem of image classification. The latter consists in labelling a whole image with respect to what is inside. Indeed, the model learns some relevant features which well describe the introduced objects so that it can tell if an image is showing a dog or a cat for instance.

But when the image introduces both a dog and a cat, the problem becomes more complex. That's why other deep architectures were then developped to solve the finer task of object detection. Detection consists in localizing different kinds of objects in an image. Thus, in this problem, we identify both the locations of the objects and their classes but it is still done at a coarse level as it is shown in figure 1.2a.

On the contrary, semantic segmentation performs a dense labelling: it consists in identifying for each pixel the kind of object it belongs to, in other words to do the classification at a very fine level (at the pixel level). For illustration purposes, in figure 1.2b, we could like to label all pixels of cat(s) as 'cat', and the others as 'ground', 'trees' or 'sky'.

(a) Illustration of detection (b) Illustration of semantic segmentation

Figure 1.2: Illustrations of two problems which can be tackled with deep learning methods.

1.3 Prior art on images

Many models were developped in order to perform semantic segmentation on images. They are based on the convolutional neural network (CNN [1]) which is the standard architecture for image classification. Figure 1.3 illustrates this architecture. It is built on two main parts. The first one is composed of convolutional layers which aim at extracting the features. The convolutions enable weights sharing and thus give the property of translation equivariance. This part is also composed of pooling layers which enable to capture a larger field of view with a reduced amount of memory and which enable to add some fine translation invariance since it captures information at a coarser level. After this downsampling part, one can find fully connected layers which aim at doing the classification from the extracted features.

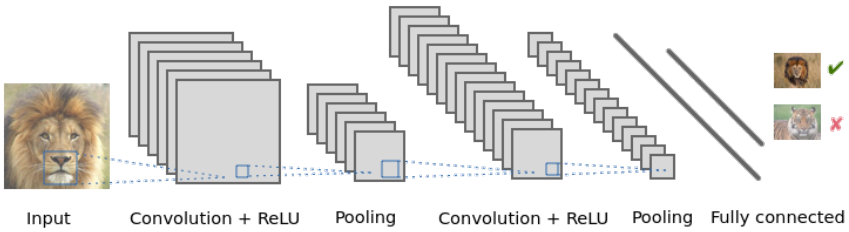


Figure 1.3: CNN architecture.

In order to perform semantic segmentation, one needs to classify each pixel. To do so, we can use a patch based approach: it consists in cropping the image in patches centered in each point. Each resulting image, which contains the pixel we want to label and some context around it (also called receptive field), is then fed to a CNN which performs the classification task.

To make it more efficient, the FCN architecture [2] introduced the idea of replacing the fully connected layers of the CNN by convolutionnal ones with filters of kernel size 1×1 so that the process can be parallelized. Even if we get then an image at the output, this one is of a reduced size since it has been downsampled by the pooling operations. Thus, to get the image back to its original resolution and thus a dense labelling, an uspampling of the resulting map can be performed with a simple bilinear interpolation. Figure 1.4 illustrates this process. Let's note that since this fully convolutionnal network is composed only of convolutionnal layers, as its name indicates, it allows images of any sizes at the input.

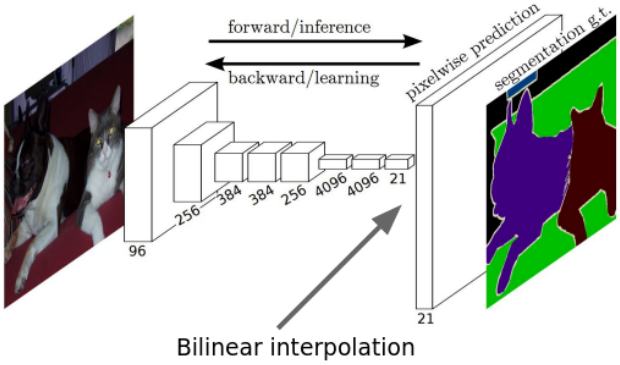


Figure 1.4: FCN architecture.

Then, because the bilinear interpolation is not a very precise process to perform classification at the pixel level, the upsampling can instead be learned. This can be composed of one or many layers. A possibility is to have an upsampling layer for each dowsampling one so that they each learn the "reverse" operation. It thus results in a symmetric network composed of a downsampling part and an upsampling part which mirror each other: for each convolution and each pooling operation in the downsampling part, there will be a convolution and an upsampling operation with the same parameters in the upsampling one (same numbers of filters, same kernel sizes, same strides, same paddings). The upsampling operation can consist in a simple padding based on the repetition of the values in their neighborhoods.

Since the upsampling layers are often followed by convolutional ones in order to make

the upsampling learnable, the DeconvNet architecture [3] introduced the idea of using transposed convolutions to combine these two kinds of layers and replace them for efficiency purposes. Figure 1.5 illustrates this architecture which uses the previously mentionned principles to perform semantic segmentation.

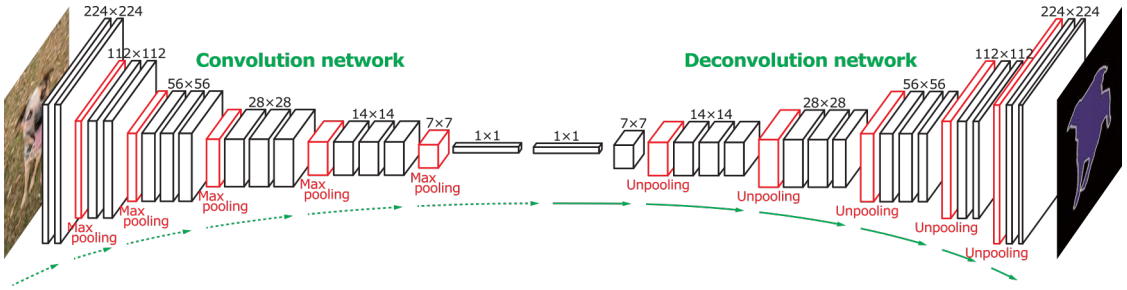


Figure 1.5: DeconvNet architecture.

In order to help more the network learn the upsampling and so label better each pixel, the segnet architecture [4], which uses max pooling layers, shares information from the dowsampling operations to the upsampling ones. Thus, as figure 1.6 illustrates, the upsampling itself consists in placing the input values at the recorded locations of the maxima in the corresponding max pooling layer. The other values are simply set to 0. Thus, it enables to keep some information from the downsampling part and so to better learn the upsampling.

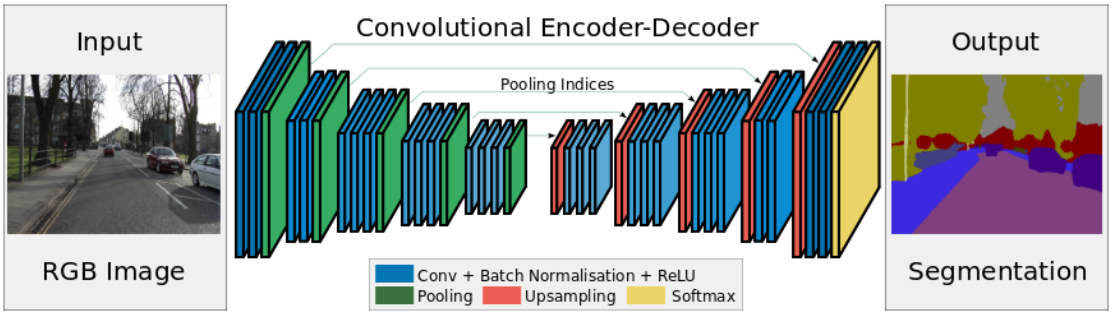


Figure 1.6: Segnet architecture.

In order to further improve the semantic segmentation, more information can be shared between the dowsampling and upsampling parts so that the model can learn at different scales. Indeed, the labels can depend on more or less large and coarse receptive fields. Hence, skip connections can be used such that some features maps of the downsampling part are concatenated together with the corresponding upsampled features maps (same sizes) before convolutions are performed on the result. Figure 1.7 illustrates the U-net architecture [5] which uses this principle. The connections enable the models to have information at different scales since the more the image is downsampled, the coarser the features we capture. Moreover, because information are lost during the pooling operations, these combinations enable to retrieve them in the upsampling part and so to improve the dense labelling. What is more, architectures with more layers are supposed to perform better in theory but since it is not always the case in practice, these skip connections enable the network to learn how deep the model should be for a better semantic segmentation.

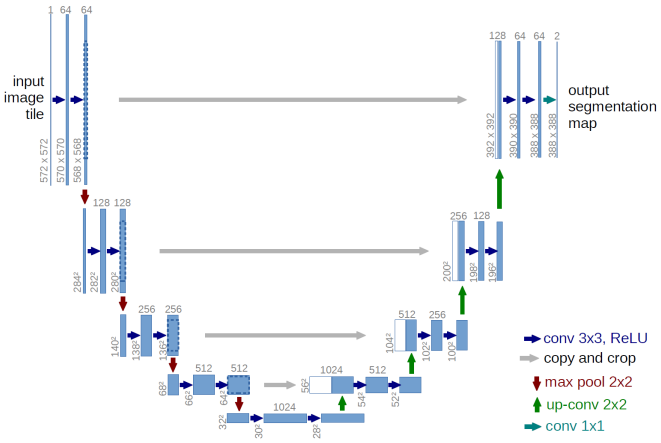


Figure 1.7: U-net architecture.

In order to perform a good classification at the point-wise level, the learning at different scales can also be done by using parallel separated networks which extract more or less coarse features. Figure 1.8 illustrates the PSPNet [6] which uses this principle in the middle of its architecture to extract features at different scales before combining them to perform the dense labelling.

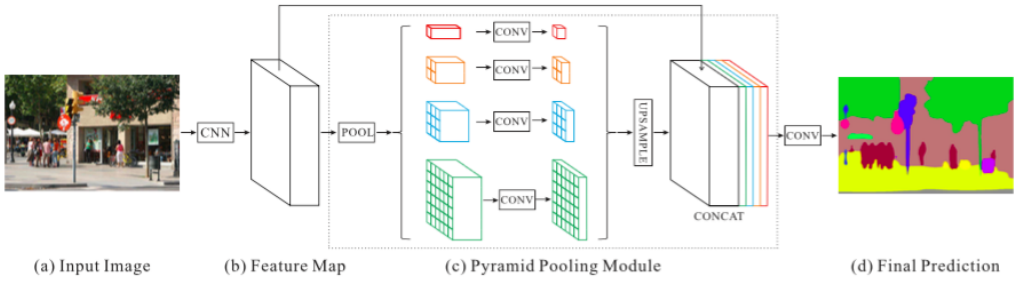


Figure 1.8: PSPNet architecture.

1.4 From images to 3D points clouds

In order to be able to perform semantic segmentation on 3D points clouds, some models directly dealing with the 3D information (without using projections on 2D spaces) have been developped, such as the 3D-CNN [7]. The main idea of such models is to use what exists on images by simply adding an extra dimension to the tensors. Since these kinds of models imply working with sparse grids which cells contain global information, they are not efficient nor well suited if we want to perform a fine labelling of all the 3D points.

Because graphs enable to connect nodes to each other depending on their respective relationships (represented by edges), they can represent data in an efficient way. Therefore, it seems graphs are well suited for these kinds of applications for which the 3D structure is sparse. Thus, in order to perform the task of semantic segmentation on 3D points clouds, we used graphs as support of the data to both develop an accurate and computationally

efficient model but also to efficiently capture the local information around each point, this local context enabling better results in general.

2 Model

2.1 Graph construction

There are many possibilities to build a graph when the dataset is a 3D points cloud. Because the points are in essence related to each other by their relative positions in the 3D space, it seems natural to base the graph construction on this idea. Thus, the nodes will be connected with respect to their distances to each other and the edges between them will represent how close the points are (the information of distance will be encoded in these links). Thus, a first possibility would be to compute the distances between each point and all the others and then apply a gaussian kernel of the following form:

$$W_{i,j} = \exp(-\frac{d_{i,j}^2}{2\sigma^2})$$

where $W_{i,j}$ are the coefficients of the weight matrix representing the graph, $d_{i,j}$ is the euclidian distance between nodes i and j, and σ is the scale hyper-parameter.

Since we do not want to keep the weak links between nodes which are far from each other to reduce the computational cost, we could then apply a simple threshold so that we only keep the strong edges (we want to only keep the connections between the close nodes). Let's note that instead of thresholding the weight matrix coefficients, we can also first keep the distances below a certain threshold and then apply a gaussian kernel on the remaining values, leaving the other weights to 0.

But doing so is not very efficient since we need to compute a huge amount of distances. A better idea is to first subdivide the space and build a tree which leaves contain points and which enables to efficiently do both the distances computations and the thresholding. Thus, KD-tree (subdivision in boxes) and ball-tree (subdivision in balls) strategies can be used to identify for each point its neighbors within a chosen distance (the threshold). The neighborhood being known, the desired distances can then be computed before being passed in the gaussian kernel.

By construction, this method implies having different amounts of points in the neighborhoods defined by a fixed distance. Because we want to have a sufficient amount of neighbors so that we well capture the relationships between points and thus the manifold (argument in favor of a high threshold) but we also want to enforce the sparsity of the weight matrix for computational efficiency (argument in favor of a low threshold), an alternative can be to use one of the two tree-based strategies to extract only the k nearest neighbors of each point, with a well chosen k. From these k neighbors, we can then compute their distances to the concerned node and apply the gaussian kernel. The disadvantage of this kNN method is the following: it can happen we capture a point which is far away from the node of interest and which connection with is thus not relevant. In a worse scenario, it can happen we capture a neighbor which belongs to another part of the cloud so that we create connections between nodes that should not be connected and so we badly represent the manifold. This situtation can happen when two different objects are close, or when some objects are sufficiently small so that connections through them are created. To avoid this problem, we could first think to threshold the distances to the kNN but since this scenario can also happen with the method based on a threshold, we would still have the same problem. A means to avoid this difficulty is to mesh the cloud.

Indeed, a mesh enables to approximate a geometric domain such as a surface by fitting polygones such as triangles to its points (each vertex of the triangles is a point). These polygones do not overlap but they can be adjacent to each other. In the meshing process, also called mesh generation, no point is left alone. Since photogrammetry leads to a 2D manifold, the meshing operation seems to be well suited for our application. From this surface approximation with triangles, we can extract a neighborhood for each point. Thus, the neighbors will be defined as the other vertices of the triangles in which the node of interest is present. Then, we can compute the distances between them and this latter and apply the gaussian kernel. By construction, the mesh enables to get a neighborhood which does not come along the previously mentionned problem. Moreover, it enables to have an accurate representation of the relationships between nodes and to have a sparse graph. Even if there are some mistakes as we can see on the figure 2.1, the erroneous links are weak since the corresponding distances are larger than the others. Therefore, we used this method to build the graph. More precisely, we used the 2.5D Delaunay triangulation (with the best fitting plane option) provided as a plugin of the CloudCompare software [1] to do the meshing operation.

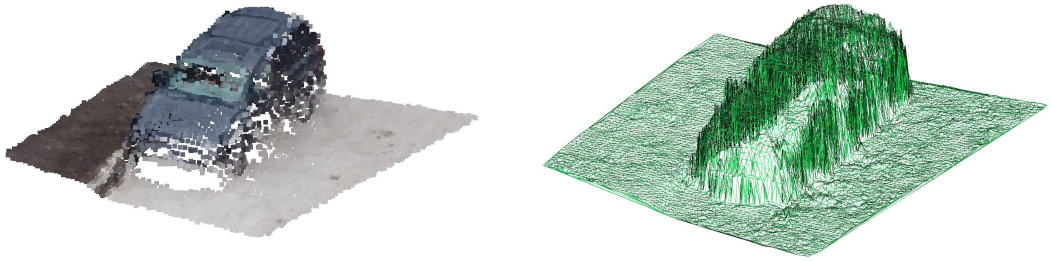


Figure 2.1: Mesh generation.

Thus, we got a weight matrix $W \in \mathbb{R}^{n \times n}$ (also called weighted adjacency matrix) which contains 0 everywhere except where there are connections between nodes. The non-zero values obtained from the gaussian kernel applied to the distances between these connected nodes represent thus how close these latters are from each other. By construction, the matrix is sparse, symmetric and real and its diagonal is filled with 0 (no self loop).

2.2 Convolutions on graphs

Now we have seen how to build the support for the computations, we need to define what are graph convolutions. But before doing so, let's define the laplacian L of a graph. This matrix is defined as the following differential operator: $L = D - W$, where D is the diagonal matrix of degrees (matrix which diagonal values are equal to the weighted degrees of the nodes, ie the sum of the values along the rows of the adjacency matrix W: $d_{i,i} = \sum_j w_{i,j}$). Because W and D are symmetric real, L is also symmetric real. Hence, according to the spectral theorem, there exists an orthogonal matrix $U \in \mathbb{R}^{n \times n}$ ($UU^T = U^T U = I_n$) and a diagonal matrix $\Lambda \in \mathbb{R}^{n \times n}$, containing the eigenvalues of the Laplacian L along its diagonal, such that $L = U \Lambda U^T$.

From the orthogonal matrix U, we can then define [8] the so called graph Fourier

[1] http://www.danielgm.net/cc/

transform $\hat{x} \in \mathbb{R}^n$ of a signal $x \in \mathbb{R}^n$ and its inverse $\tilde{x} \in \mathbb{R}^n$ by the following respective operations:

$$\hat{x} = \mathcal{F}_{\mathcal{G}}\{x\} = U^T x$$
$$\tilde{x} = {\mathcal{F}_{\mathcal{G}}}^{-1}\{\hat{x}\} = U\hat{x} = UU^T x = x$$

Let's notice that $\hat{x} = \begin{bmatrix}\hat{x}(\lambda_1) & \cdots & \hat{x}(\lambda_n)\end{bmatrix}^T$ with λ_i the eigenvalues of the Laplacian L (the values of Λ) which are analogous to the frequencies of the "standard" Fourier domain.

Now, we can use the convolution theorem to define the graph convolution of two signals $s \in \mathbb{R}^n$ and $x \in \mathbb{R}^n$:

$$s *_{\mathcal{G}} x = U(U^T s \odot U^T x) = U(U^T x \odot U^T s) = U(\hat{x} \odot U^T s) = U(\text{diag}(\hat{x})U^T s)$$

(with $\odot$ refers to the pointwise Hadamart product)

$$s *_{\mathcal{G}} x = U diag(\hat{x})U^T s = U \begin{bmatrix}\hat{x}(\lambda_1) & & 0 \\ & \ddots & \\ 0 & & \hat{x}(\lambda_n)\end{bmatrix} U^T s$$

Then, "standard" deep learning can learn to extract relevant features from the row input signals by learning filters which are then applied through convolutions. Thus, the idea is to learn functions $\hat{x} : \lambda \mapsto \hat{x}(\lambda)$. To do so, Defferrard et *al.* [9] used Chebychev polynomials T_j to approximate them:

$$\forall i, \hat{x}(\lambda_i) \approx \sum_{j=0}^{K-1} \theta_j T_j(\lambda_i)$$

where K is the polynomial order which refers to the highest degree (K-1) of the polynomials and θ_j are learnable parameters.

Because these polynomials verify the following recurrence relations, they can be built efficiently:

$$\begin{cases} T_0 = 1 \\ T_1 = X \\ \forall p, T_{p+2} = 2XT_{p+1} - T_p \end{cases}$$

But to perform an approximation with such polynomes, the λ_i have to be in [-1,1]. Instead of using the combinatorial Laplacian L, we considered the normalized Laplacian $L_{norm} = D^{-1/2}LD^{-1/2}$ which eigenvalues are in [0,2]. Thus, we just substracted I_n to L_{norm} in order to have the λ_i in [-1,1] (all the previous formula are the same, except we have to replace L by $L_{norm} - I_n = -D^{-1/2}WD^{-1/2}$). Note that if we would have used the combinatorial Laplacian, we would have multiplied it by $\frac{2}{\lambda_{max}}$ before substracting I_n to the result (rescale of the eigenvalues in [-1,1]).

Then, the graph convolution can be written as follows:

$$s *_{\mathcal{G}} x \approx U \begin{bmatrix}\sum_{j=0}^{K-1} \theta_j T_j(\lambda_1) & & 0 \\ & \ddots & \\ 0 & & \sum_{j=0}^{K-1} \theta_j T_j(\lambda_n)\end{bmatrix} U^T s$$

$$s *_{\mathcal{G}} x \approx U(\sum_{j=0}^{K-1} \theta_j T_j(\Lambda))U^T s = (\sum_{j=0}^{K-1} \theta_j U T_j(\Lambda) U^T)s = (\sum_{j=0}^{K-1} \theta_j T_j(U\Lambda U^T))s$$

$$s *_{\mathcal{G}} x \approx (\sum_{j=0}^{K-1} \theta_j T_j(L))s = \sum_{j=0}^{K-1} \theta_j T_j(L)s$$

$$s *_{\mathcal{G}} x \approx \begin{bmatrix} T_0(L)s & \cdots & T_{K-1}(L)s \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_{K-1} \end{bmatrix}$$

Thus, the operations can directly be done in the spatial domain. Hence, even if the graph convolutions were first defined in the spectral domain, we will use a spatial implementation of them. Moreover, we can notice that

$$s *_{\mathcal{G}} x \approx \begin{bmatrix} X_0 & \cdots & X_{K-1} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_{K-1} \end{bmatrix}$$

with $X_i = T_i(L)s \in \mathbb{R}^n$ satisfying the following recurrence relations inherited from the Chebychev polynomials, which enable to limit the number of computations:

$$\begin{cases} X_0 = s \\ X_1 = LX_0 \\ \forall p \in [\![0; K-3]\!], X_{p+2} = 2LX_{p+1} - X_p \end{cases} \tag{1}$$

Now, if look at the product of L by s, we get the following result:

$$Ls = \begin{bmatrix} l_{11} & \cdots & l_{1n} \\ \vdots & \ddots & \vdots \\ l_{n1} & \cdots & l_{nn} \end{bmatrix} \quad \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix} = \begin{bmatrix} \sum_{j \in \mathcal{N}(1)} l_{1j} s_j \\ \vdots \\ \sum_{j \in \mathcal{N}(n)} l_{nj} s_j \end{bmatrix}$$

where $j \in \mathcal{N}(i)$ points out the neighbors j of the node i. Thus, as we can see, this operation enables to get the information of the one hop neighbors of each node. If we go further and look at the product of L^2 and s:

$$L^2 s = L(Ls) = \begin{bmatrix} l_{11} & \cdots & l_{1n} \\ \vdots & \ddots & \vdots \\ l_{n1} & \cdots & l_{nn} \end{bmatrix} \quad \begin{bmatrix} \sum_{j \in \mathcal{N}(1)} l_{1j} s_j \\ \vdots \\ \sum_{j \in \mathcal{N}(n)} l_{nj} s_j \end{bmatrix} = \begin{bmatrix} \sum_{k \in \mathcal{N}(1)} l_{1k} \sum_{j \in \mathcal{N}(k)} l_{kj} s_j \\ \vdots \\ \sum_{k \in \mathcal{N}(n)} l_{nk} \sum_{j \in \mathcal{N}(k)} l_{kj} s_j \end{bmatrix}$$

As we can see, it enables to get the information of the neighbors of the neighbors of each node (so it includes the node itself). Thus, we can get the information until two hops from each node. More generally, $L^K s$ enables to get the information until K hops from each node. That's why the learned filters with a polynomial order $(K-1)$ will be "$(K-1)$ -localized".

Since for each node we weight the information of the datapoint and of its neighbors to assign it a resulting value, we can also interpret the graph convolution in the spatial

domain. Indeed, this operation is similar to what a classical convolution does by considering some context in addition to the data of the sample to assign this latter a value based on a certain weighting of the information (the weights are the coefficients of the filter).

Now, let's see what the operation looks like if we have N_{in} input signals $s_i \in \mathbb{R}^n$ and if we use N_{out} filters. Since we want to allow the combination of the information of all the input signals, the operation will then be:

$$S_{out} = \begin{bmatrix} T_0(L)S_{in} & \cdots & T_{K-1}(L)S_{in} \end{bmatrix} \begin{bmatrix} \theta^1_{1,0} & & \theta^{N_{out}}_{1,0} \\ | & \cdots & | \\ \theta^1_{N_{in},0} & & \theta^{N_{out}}_{N_{in},0} \\ \vdots & \cdots & \vdots \\ \theta^1_{1,K-1} & & \theta^{N_{out}}_{1,K-1} \\ | & \cdots & | \\ \theta^1_{N_{in},K-1} & & \theta^{N_{out}}_{N_{in},K-1} \end{bmatrix}$$

where $S_{in} = \begin{bmatrix} s_1 & \cdots & s_{N_{in}} \end{bmatrix}$ (and so $T_j(L)S_{in} = \begin{bmatrix} T_j(L)s_1 & \cdots & T_j(L)s_{N_{in}} \end{bmatrix}$), $k \in [\![1; N_{out}]\!]$ refers to the filter, $i \in [\![1; N_{in}]\!]$ refers to the input signal $s_i \in \mathbb{R}^n$ and $j \in [\![0; K-1]\!]$ refers to the order of the polynomial in the learnable weights $\theta^k_{i,j}$. Let's notice that $S_{out}(p,k) = \sum_{i=1}^{N_{in}} \sum_{j=0}^{K-1} \theta^k_{i,j}(T_j(L)s_i)(p)$ where $p \in [\![1; n]\!]$ refers to the datapoint.

To build the matrix on the left, we can use the same trick as before by replacing s by S_{in} in the recurence relations (1) for efficiency purposes ($X_j = T_j(L)S_{in} \in \mathbb{R}^{n \times N_{in}}$ then).

2.3 Coarsening and pooling

In order to perform the pooling operation on the nodes, these latters have to be spatially grouped depending on the part of the cloud they belong to and so on the relationships we pointed out between them during the graph construction. Indeed, the nodes are "randomly" placed in the adjacency matrix and in the data matrix so far (same order in these two matrices) and we would like that from a piece of the 3D cloud, we obtain a downsampled version or a point in the extreme case after the pooling.

Then, instead of picking different amounts of nodes located at different places in the data matrix, we would like both to group them so that the pooling can be done in a moving fashion (pooling on the first amount of points, then pooling on the second one, and so on and so forth...) and we would like to have a fixed amount of points in the groups so that the pooling operation is always on the same number of samples and its stride is constant. Thus, the idea is to reorganize the nodes so that we can use the existing pooling framework on 1D signals (constant size and stride).

To do so, we used the graclus strategy. Starting from the nodes i with the lowest degrees, it consists in pairing them with other vertices of the graph by maximizing the coefficient $w_{i,j}(\frac{1}{d_{ii}} + \frac{1}{d_{jj}})$, where $w_{i,j}$ represents the strength of the connection between the nodes i and j, and d_{kk} is their respective degrees. Thus, the matching of the node i is done so that the connection with its match j is the strongest possible with a favor for the low degree neighbors. It results then in a coarsened graph which nodes are *children'* by comparison with the *parents* vertices of the initial graph. Indeed, these children are obtained by merging the matched parents. Moreover, they inherit of their connections so that if the two matched nodes have both connections with a third vertex, the child will have a stronger connection with this one (equal to the sum of the parents edges). Let's note that when no match is found for the parent, its child is simply equal to it. Then, from this coarsened graph, we can repeat the process to coarse it at a higher level.

At the end of this process, we are given multiple graphs which correspond to different levels of coarsening. Then, the vertices left single in the matching process are paired with fake nodes and couples of fake parents are associated to these latters. Then, the nodes of the different graphs can be reordered so that the parents are next to each other and thus the union of two neighbors from layer to layer forms a binary tree. This latter enables then to easily do the pooling operation since the nodes are grouped with respect to their relationships to each other and since the formed groups are of the same size. Because max pooling operations are used in our model and because these ones are done after ReLU activations, the data at the fake nodes locations were set to 0 to not be taken into consideration.

For illustration purposes, let's look at the figure 2.2. On the left is illustrated the matching procedure. Thus, at the first stage, the blue cells are connected but the dark ones have a stronger connection. So these latters are paired, leaving the light blue single. The two green ones which are connected together are paired and the red one which is not connected to the other nodes is left single. At the second stage, the light blue node is connected to the child of the dark blues cells since parents additively give their relations. The green and red nodes which are isolated are left single. Then, we can locate the parents by assigning them the locations of their children. Thus, in the first coarsening level, 0 will be assigned to the two parents of the child located at the first place, 1 to the parents of the child at the second place and so on and so forth... We do so at each stage so that at the end, we can easily locate the positions of the parents of a given child.

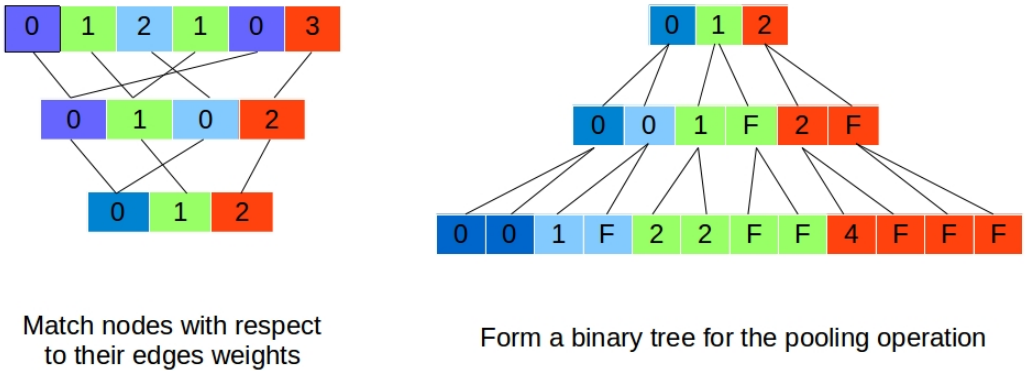


Figure 2.2: Illustration of the coarsening principle.

Then, as it is illustrated on the right part of the figure, fake nodes are added to pair the single vertices and couples of fake parents are associated to the fake children. Starting from the coarsest level, the nodes are then reordered with respect to their paternities. As we can see, it results in a binary tree which make easy the pooling operation.

The figure 2.3 illustrates the new order of the nodes of a car at the non coarse level for a coarsening with six levels. Indeed, our model use three max poolings of size 4 ($2^6 = 4^3$). To obtain this result, the reordered nodes were assigned an increasing number with a step of 4, meaning each group of 4 successive nodes have the same color. Because the binary tree contains different unconnected parts, we get different patterns.

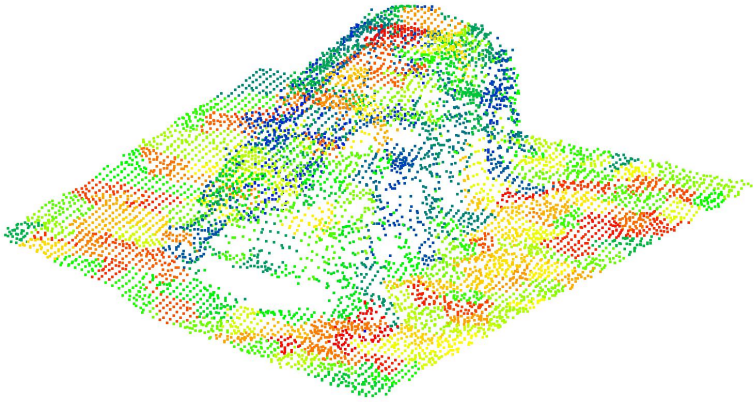
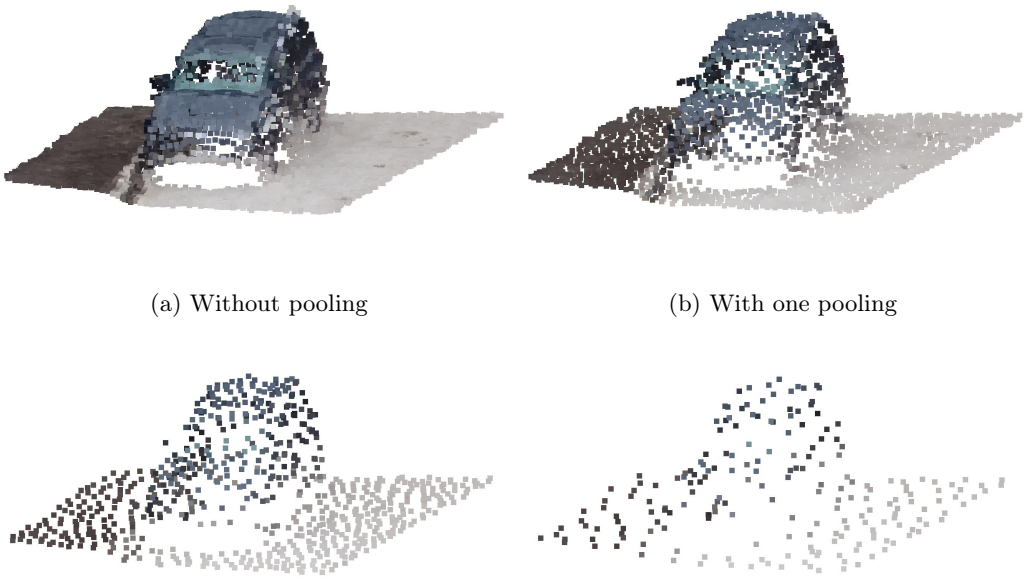


Figure 2.3: Illustration of the nodes ordering.

The figure 2.4 illustrates the three levels of downsampling we have with the three levels max pooling operations of size 4 each. The positions and the assigned values of the children are respectively defined by the barycenters of the coordinates and by the weighted sum of the RGB colors of the parents.



(a) Without pooling

(b) With one pooling

(c) With two poolings

(d) With three poolings

Figure 2.4: Coarsening operations at three different levels.

2.4 Model architecture

The data in input are first normalized with respect to their means and their standard deviations. Then, before being fed into the main architecture, a batch normalization on the features is performed so that the model can learn the normalization: $f_{i,j} = \gamma \frac{f_{i,j} - \mu_j}{\sigma_j} + \beta$

where $f_{i,j}$ denotes the feature j of the point i, γ and β are learnable parameters, μ_j and σ_j are respetively the mean and the standard deviation of the feature j on the datapoints.

The main architecture is based on two parts which mirror each other. The downsampling part is composed of three main layers. Each of them performs a graph convolution with a polynomial order $K = 5$, uses a rectified linear unit (ReLU) as activation function to introduce some non linearities, applies a batch normalization on the features and then a max pooling on the results with a pool size of 4. The numbers of filters for the graph convolutions are respectively of 64, 128 and 256. After these three layers, one can find two similar layers except that no max pooling operation is performed in both of them and the second one does not contain ReLU. The numbers of filters used for the 2 graph convolutions are both set to 512. Then begins the upsampling part. This latter is composed of 3 main layers. Each of them is composed of an upsampling by 4 based on a repetition of the values, a graph convolution with a polynomial order $K = 5$ and a batch normalization on the features. The numbers of filters are respectively of 256, 128 and 64.

Then, a last graph convolution with a polynomial order $K = 1$ is used to map the 64 channels to 6 output channels (there are 6 classes). After that, a softmax is applied to compute the probabilities of a point to belong to the different classes.

Figure 2.5 illustrates our model architecture.

The cross entropy was chosen as loss function since this cost function is well suited for classification problems. In order to tackle the problem of class imbalance, the loss was weighted with respect to the class frequencies with the following coefficients: $w_i = \frac{f_{med}}{f_i}$, where f_i is defined by the total number of occurrences of class i there are in the train set divided by the total number of points in the tiles for which the class i is present, and f_{med} is the median of the f_i. Let's note that the computation of the loss does not take into consideration the added fake nodes nor the points in the strips added around the tiles to give more context to the nodes on the borders.

The initializations of the learnable coefficients of the graph convolutions filters were done with random numbers following a normal distribution with means equal to 0 and standard deviation equal to $\sqrt{\frac{2}{K \times NB_{filters}}}$.

The model was trained for 200 epochs. For each of them, we compute the loss and the accuracy on both the whole training set and the whole validation set. For the optimization, we used gradient descent with a learning rate of 10^{-2}.

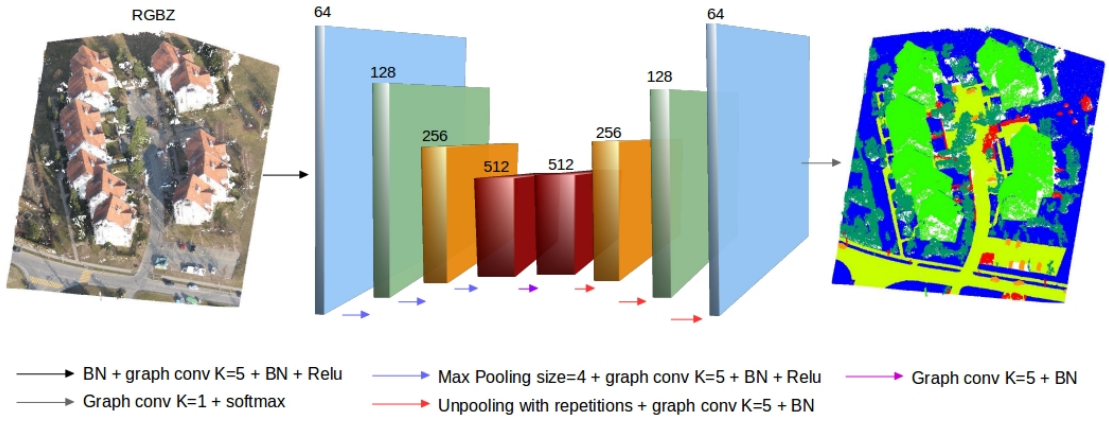
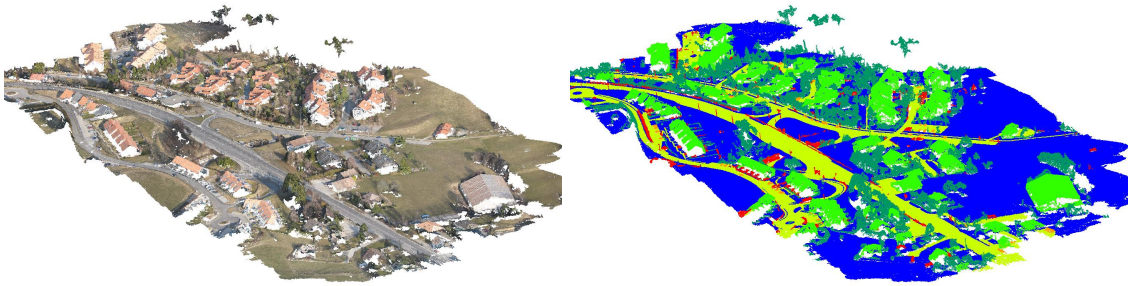


Figure 2.5: Model architecture.

3 Results

3.1 Data: geospatial 3D point cloud

After the photogrammetry process, we are given a point cloud such the one of figure 3.1 which was make available by Pix4D. Since some experiments have already be done on this cadastre, we chose to use this one to develop our model. As we can see, we have for each datapoint its x,y,z coordinates, its R,G,B colors and its label. For this dataset, there are 6 different classes: ground, road, building, high vegetation, car and human made object. As figure 3.2 shows, the dataset is very imbalanced, with a majority class of 51% (the ground) and a minority class of 0,4% (the cars).



(a) Dataset (RGB) (b) Dataset (labelled)

Figure 3.1: Cadastre: dataset provided by Pix4D.

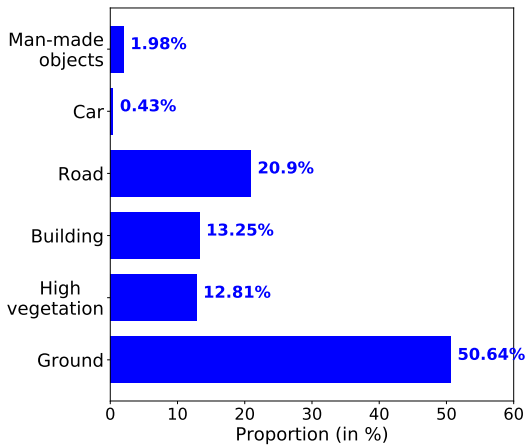


Figure 3.2: Highly imbalanced class distribution of the cadastre dataset.

Then, we also tried to extract additional features from the data. To do so, we used the CANUPO software which enables to extract geometric features at different scales [10]. Indeed, it can compute for each point the three normalized eigenvalues which are obtained

by applying a PCA to all the coordinates in the ball centered on the point of interest and of the chosen diameter. The respective values of these three eigenvalues (which are between 0 and 1) enable to tell if the object the point belongs to looks like more 1D, 2D or a 3D at the given scale. The software also enables to extract for each point the angle between the xy plane and the normal to the local surface in this point. We can also combine the three eigenvalues in order to extract additionnal geometric features such as the ones suggested by Pix4D [11].

As figure 3.3 shows, we used a random forest in order to select the most promising features. To do so, we first considered their relative importance in the task of semantic segmentation and then we kept the features which removals were impacting the performances of the classifier. It appeared that the most promising features were the angle and the third eigenvalues (the one indicating if the local point cloud is more or less 3D) obtained at the scales 0.3, 1.5, 3 and 10 m.

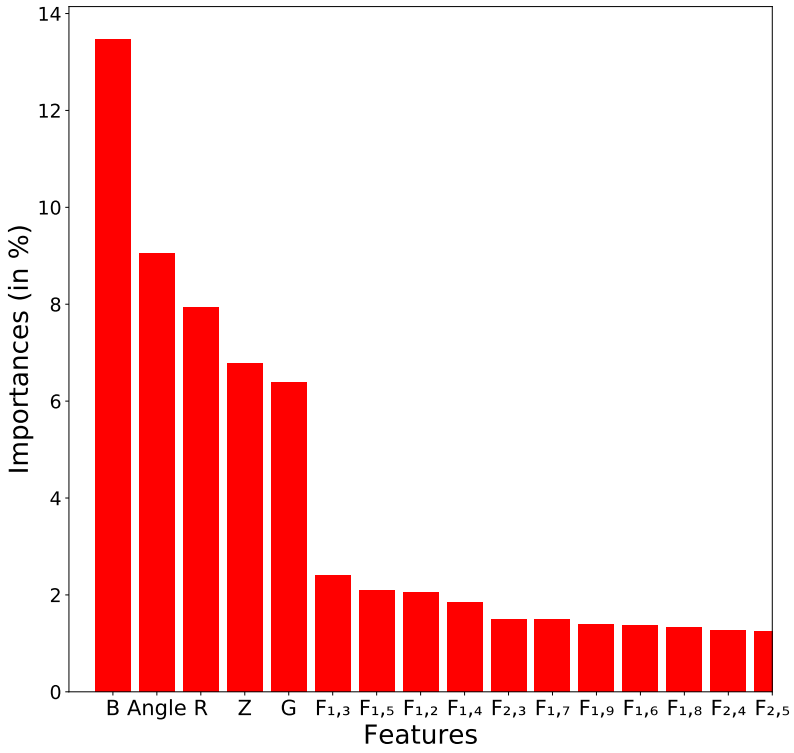


Figure 3.3: Features selection with respect to their importances in the classification task of a random forest.

3.2 Preprocessing of the dataset

Because the amount of data in the whole dataset was too large to fit in the memories of our computers, the dataset was cut in tiles. For simplicity purposes, the tiling was done with respect to the x and y coordinates. Thus, the xy plan was subdivided in rectangles of chosen size ($36m \times 36m$) and each tile was defined as all the points which x and y coordinates fall in one these subspaces. This process resulted in non overlapping point clouds. Then, because our model is supposed to use the information of the neighborhood, extra strips were added to each rectangle in order to give some context to the points on

the borders and they can thus also benefit of neighbors information. Hence, it resulted in overlapping tiles of size $48m \times 48m$ (at most, depending on the tile locations) in the xy plane.

Then, all these tiles were divided in training and test sets with a split of 50-50%. The test set was further divided in the test set and the validation set with a split of respectively 70-30%. The figure 3.4 shows the repartition of the different tiles: the dark green ones correspond to the training set, the dark blue ones to the validation set and the dark red ones to the test set. The other colors correspond to the area where the tiles overlap.

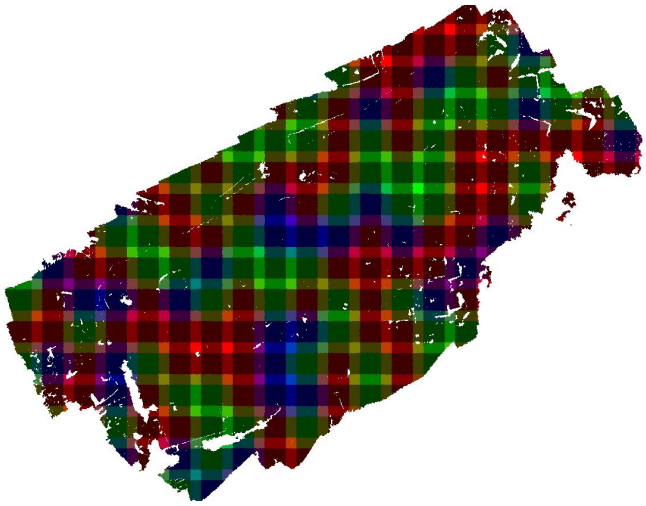


Figure 3.4: Illustration of the tiles split: the dark green tiles correspond to the training set (50%), the dark blue ones to the validation (16%) set and the dark red ones to the test set (34%). The other colors correspond to the area where the tiles overlap.

From the obtained tiles, we can then build their respective graphs and coarsen them with respect to the pooling operations.

Because we split the dataset in tiles looking only at the x and y coordinates, the number of points is rather different from tile to tile. This implies that the different samples have not the same importances in the learning process since they are averaged on the number of datapoints per tile. To avoid this, we concatenate the weights matrices such that the graphs remain disconnected from each other (we create block-diagonal matrices) and such that the number of points of interest per batch is similar. The data were simply concatenated.

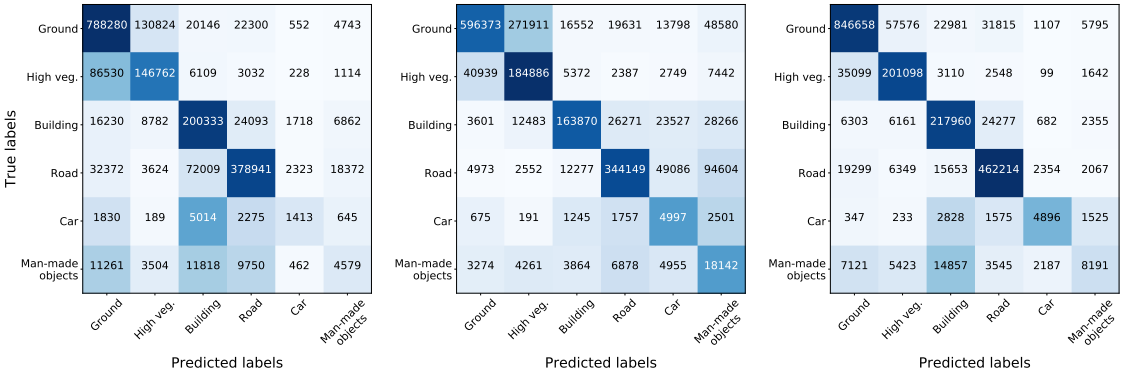
3.3 Experimental performances

In order to be able to asset the performances of our model, we used random forest and XGBoost as baselines. Based on what Pix4D chose for the semantic segmentation of the cadastre [11], we fixed the number of trees to 100 for both baselines, the max depth to 30 and 5 for respectively the random forest and XGBoost, for which we used a learning rate of 0.2. For these two algorithms, the samples were weighted with respect to their class frequency in the whole training set in order to tackle the class imbalance.

From the Z coordinate and the R,G,B colors, we obtained the results depicted in table 3.1 and in figure 3.5 which shows the confusion matrices. Figure 3.9 illustrates some qualitative results obtained from our model.

Performances	Overall accuracy (in %)	Mean accuracy (in %)	Inference time (s)
Random Forest	74.93	52.92	54.87
XGBoost	64.68	59.44	75.81
Our model	85.85	68.09	8.74

Table 3.1: Performances on the test set of the cadastre with RGBZ.



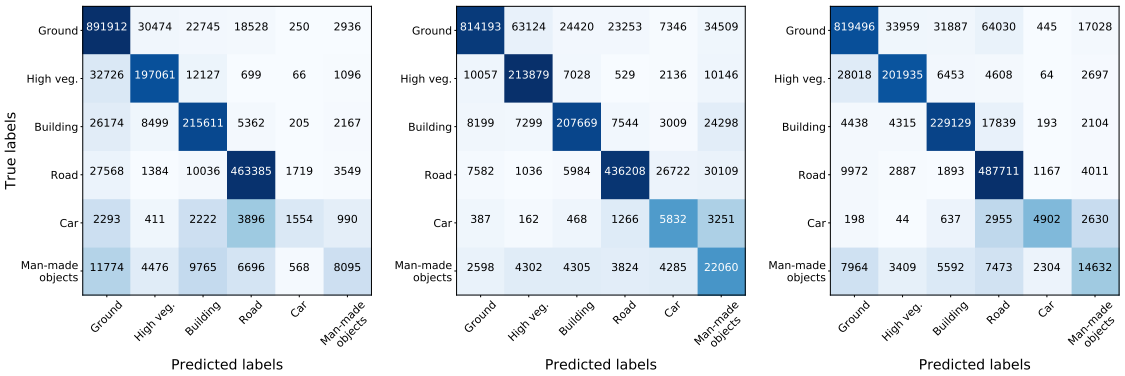
(a) Random Forest (b) XGBoost (c) Our model

Figure 3.5: Confusion matrices computed on the test set of the cadastre with RGBZ.

Then, extra features selected with the random forest were added to the input data. Table 3.2 and figure 3.6 illustrate the results obtained on the cadastre dataset for the three models.

Performances	Overall accuracy (in %)	Mean accuracy (in %)	Inference time (s)
Random Forest	87.61	63.53	41.67
XGBoost	83.78	73.83	74.86
Our model	86.63	71.83	8.82

Table 3.2: Performances on the test set of the cadastre with extra features.



(a) Random Forest (b) XGBoost (c) Our model

Figure 3.6: Confusion matrices computed on the test set of the cadastre with extra features.

Then, in order to asset more generally the performances of our model, we extend the comparison on another dataset provided by Picterra. This one is composed of ground, vegetation, pylons and powerlines and as figure 3.7 shows, it is also very imbalanced. Figure 3.10 illustrates some qualitative results obtained from our model while table 3.3 and figure 3.8 illustrate the quantitative performances of our model and the baselines.

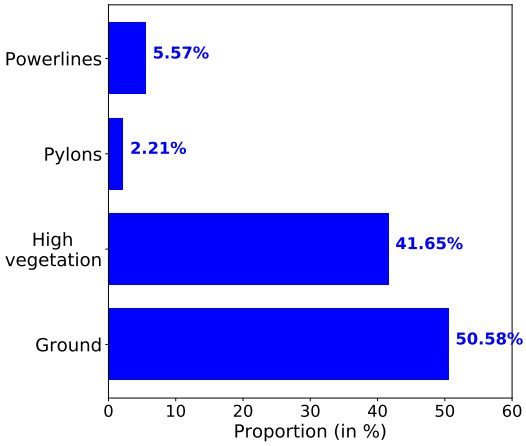
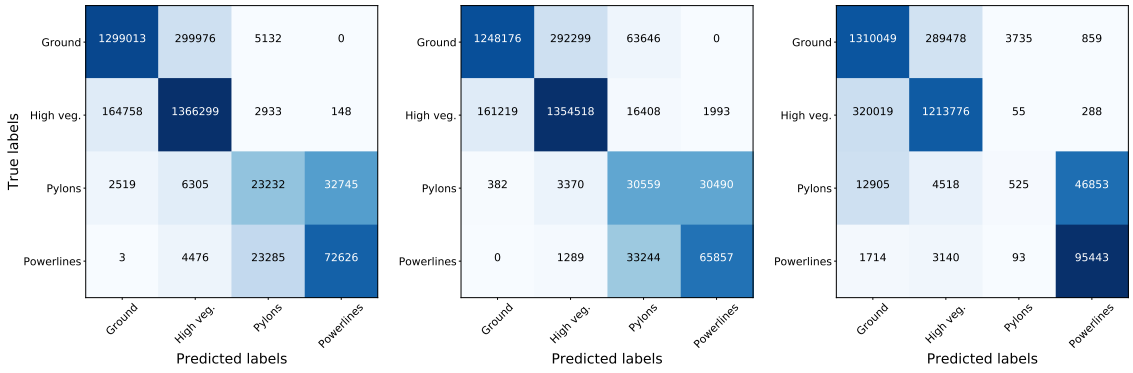


Figure 3.7: Very imbalanced class distributions.

Performances	Overall accuracy (in %)	Mean accuracy (in %)	Inference time (s)
Random Forest	83.58	69.56	99.05
XGBoost	81.71	69.72	61.83
Our model	79.30	64.17	4.99

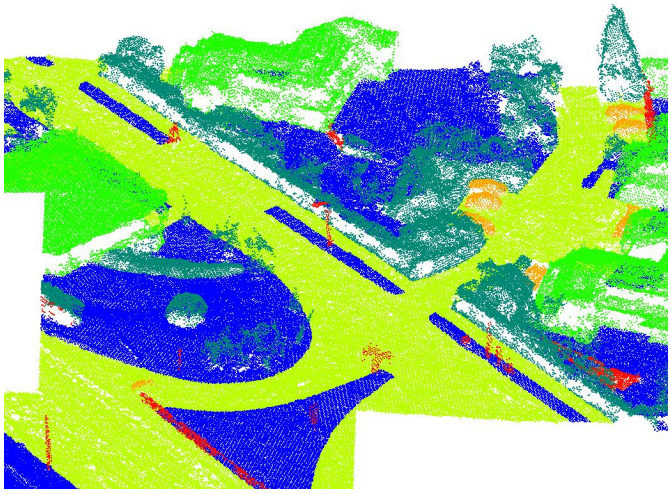
Table 3.3: Performances on the test set of the Picterra's dataset with RGBZ.



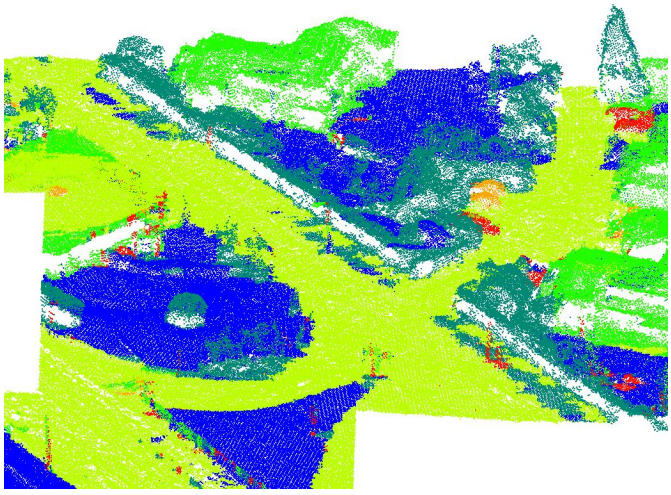
(a) Random Forest (b) XGBoost (c) Our model

Figure 3.8: Confusion matrices computed on the test set from Picterra with RGBZ.

(a) Test set



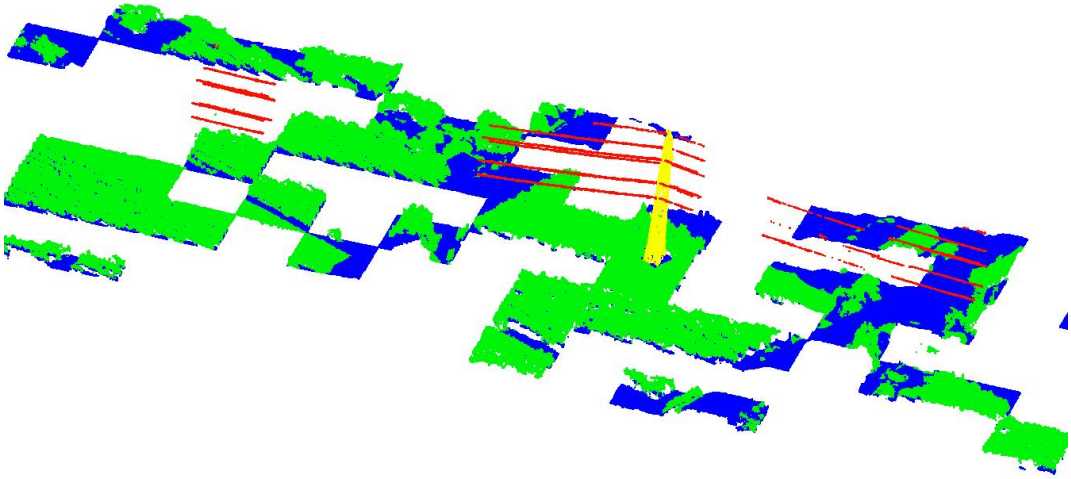
(b) Ground truth



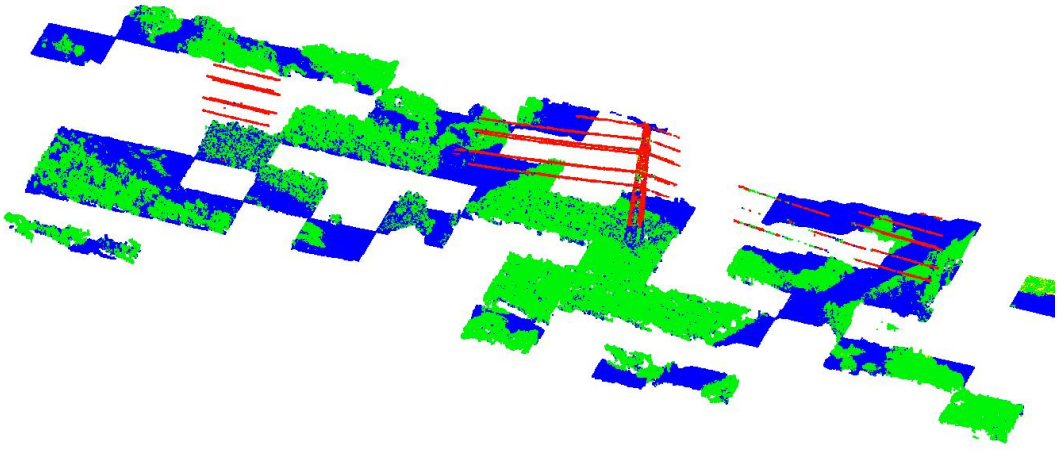
(c) Predictions

Figure 3.9: Qualitative results of our model on the test set of the cadastre.

(a) Test set



(b) Ground truth



(c) Predictions

Figure 3.10: Qualitative results of our model on the test set of the Picterra's dataset.

3.4 Discussion

The previous part enabled to asset our model since it outperforms the baselines with RGBZ as input features. Table 3.2 and figure 3.5 show the model benefits from the slected extra features since they allow a boost of 0.8% and 3.70% in respectively the overall and the mean acuracies. But these extra features also enable to boost more our baselines.

Since we wanted to know how well our model generalizes on other datasets, we run it on the data provided by Picterra. Because the density of points is larger, we considered tiles of $32m \times 32m$ and we divided the number of filters at each layer by 4 to not exceed the available memory. Table 3.2 and figure 3.5 show that despite the reductions, the model is still good, even if we lose some percentages on the accuracies. Moreover, as figure 3.10 shows, if we compare the predictions and the annotations with respect to the data, we can wonder if the model is not sometimes better than the labels. In this setting, the baselines beat our reduced model.

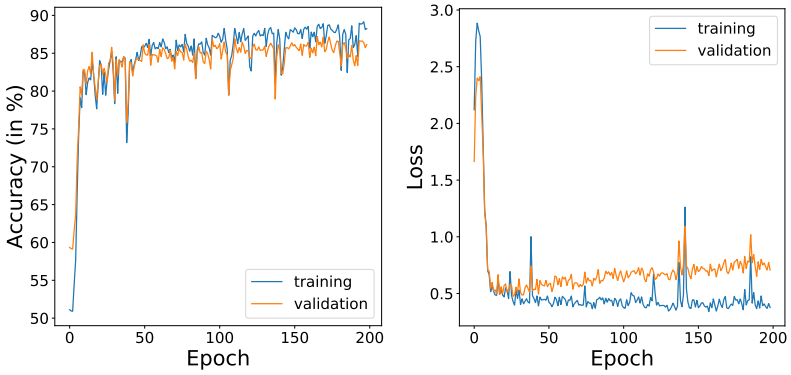
Now, we will examine the key elements of our network and how they affect the performances. First of all, the batch normalizations on the features after the graph convolutions appeared to be an essential element since without them, the network does not learn anything and the accuracy is very low as the model 1 of the table 3.4 illustrates (this one was obtained by removing the batch normalizations from the reference model). Since our model is quite deep (9 main layers), it seems necessary to fight against the problem of the vanishing gradient: since gradients from layers to layers are multiplied during the back-propagation process, we need to boost them so that the model can learn at all layers and the batch normalization is useful for that.

Then, there is the question of the normalization of the data. As we told in the part 3.2, we substract the mean to the data and we divide them by the standard deviation before feeding the network with them. The model 2 of table 3.4 shows it is not sufficient. So we decided to learn the normalization thanks to a batch normalization at the input of the network. As the model 2 shows, it enabled to gain in performance. Indeed, a bad normalization often implies the model does not learn equally from the features (more emphasis on some features) or some directions during the gradient descent are more prioritized. Thus, learning it can be a good way to avoid such problems.

As we explained during the part 3.1, we are dealing with a very imbalanced dataset. Model 3 illustrates the effect of not weighting the cross entropy loss: we can see that the infrequent classes are misclassified resulting in a higher overall accuracy but lower mean accuracy. Since we want to be able to also learn the classes for which the numbers of samples are smaller, we definitely need to weight the loss with respect to the classes frequencies. Indeed, we could also force the balance by keeping a given amount of points for each class but then, the number of points will be so small that the model could not learn anymore. Then, model 4 illustrates the effect of weighting the loss with the frequencies of the 6 classes in the whole training set. A good trade off between learning well in overall and learning well each class has to be found and it seems our weighting performs quite well in doing so.

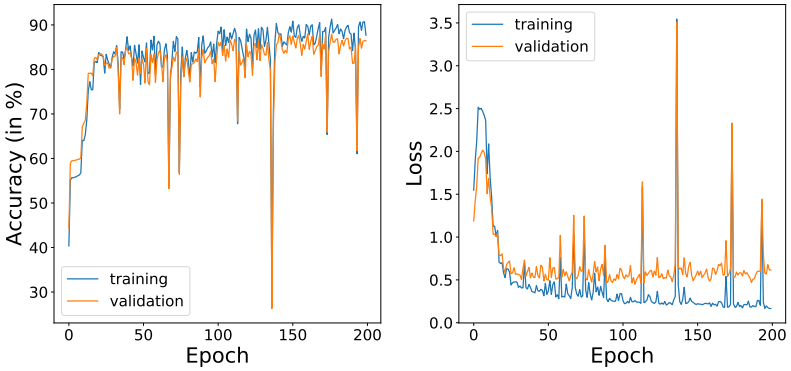
Another problem that araised from our tests is that the number of samples is different for each batch. Indeed, because we split the dataset in tiles looking to only the x and y coordinates, the number of points is rather different from tiles to tiles and so from batch to batch. During the learning process, the datapoints have thus different importances from batch to batch since their gradients are averaged on their number. This could explain the huge peaks in accuracy and loss such as the ones of figure 3.12 we encountered during the training. Indeed, even if the model 5 seems quite good, our best model enabled to cancel

these peaks as figure 3.11 illustrates. To do so, we combined the data so that the number of points from batch to batch is similar. The main idea was to diagonally concatenate the weights matrices (creating thus block-diagonal matrices) to leave the graphs disconnected from each other.



(a) Evolution of the accuracies (b) Evolution of the loss

Figure 3.11: Illustration of nice curves during the learning process.



(a) Evolution of the accuracies (b) Evolution of the loss

Figure 3.12: Illustration of some peaks we met during the learning process.

Model 6 and 7 that were obtained with the same parameters as the best model but which are based on tiles of respectively $22m \times 22m$ (size of $30m \times 30m$ with the surrounding context) and $72m \times 72m$ (size of $85m \times 85m$ with the surrounding context) show how the choices of the sizes of the tiles and the contexts are important. Too small will imply a reduced receptive field since the information cannot propagate from tile to tile. Too large will encourage the high variations in the numbers of points per batch and so could penalize the learning as we saw. Thus, a good trade off has to be found.

Then, the size of the pooling seems also to be an important parameter according to the models 8 and 9 which were obtained for a pooling size of 2 and 8 respectively. Indeed, a larger size enables to have a larger receptive field since the cloud is more downsampled but it is at the cost of losing more information. Thus, dowsampling too much probably leads to the loss of relevant information. Moreover, since we want to recover the cloud from its downsampled version in order to perform a dense labelling, the task of learning a good upsampling is harder when the data are more coarsened.

The model 10 obtained by removing a layer to both the downsampling and upsampling parts seems quite good even if its performances during the training were lower than the reference model. Since we were memory limited, we did not test the addition of extra layers but it could enable to get better scores. Indeed, in theory deeper models should provide better results but it is not always the case in practice.

The model 11 which was obtained by using only the RGB colors and the table 3.2, which illustrates the results with extra geometric features (model 12), show that using more features leads to better scores. But table 3.2 also shows our baselines benefit more from the addition of the geometric features. It can be explained by the fact our network is able to learn any function of the input features and it takes into consideration the information of the neighborhoods, in contrary to the baselines. Moreover, the network could learn from the graph itself which embeds the local structure. Since the extra geometric features are obtained by considering the local geometries of the cloud, it could explain why our model benefits less from them in comparison to the baselines. What is more, we observed that the polynomial order had to be sufficiently large to get good results as model 13 obtained for $K = 1$ shows. In this setting, we do not use the local information and so the graph: we take only into consideration the information of the datapoint to label it. Because the results are similar to those of the baselines with RGBZ as features, it could also be an element in favor of the previous hypothesis.

Performances	Overall accuracy (in %)	Mean accuracy (in %)	Inference time (s)
Ref model	85.85	68.09	8.74
Model 1	12.64	16.60	8.29
Model 2	79.23	52.70	8.72
Model 3	86.11	63.41	8.71
Model 4	85.10	69.14	8.65
Model 5	83.08	67.54	8.56
Model 6	78.42	64.36	6.10
Model 7	73.20	58.46	13.20
Model 8	71.48	58.11	12.47
Model 9	82.37	63.03	5.46
Model 10	85.25	71.58	7.59
Model 11	84.89	70.05	8.73
Model 12	86.63	71.83	8.82
Model 13	65.63	58.97	1.34

Table 3.4: Different performances on the test set of the cadastre for different models. The predictions were done by using a GPU Tesla K40c.

4 Conclusion

During this project, we developed a model for semantic segmentation of aerial photogrammetry points clouds. The core element of our work is to use deep learning on graph to get better results than random forest or XGBoost with a reduced number of features. We first explained what are the main elements of deep architectures to perform a dense labelling. Then, we saw why and how to build and use graphs in such models. Further, we provided a comparison with the two other machine learning methods previously mentionned, and an analysis of our model.

In the future, we intend to implement dilated convolutions to avoid the max pooling operations which are destructive as we saw, but also to add skip connections to allow the model to learn at different scales. Further, we would like to explore the learning on different graphs and in a next step to learn the graph itself.

Bibliography

[1] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. *Object Recognition with Gradient-Based Learning*, pages 319–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[2] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE*, pages 1–12, May 2016.

[3] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. *CoRR*, abs/1505.04366, 2015.

[4] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *CoRR*, abs/1511.00561, 2015.

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[6] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. *CoRR*, abs/1612.01105, 2016.

[7] Jing Huang and Suya You. Point cloud labeling using 3d convolutional neural network. pages 2670–2675, Dec 2016.

[8] David I. Shuman, Sunil K. Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. Signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular data domains. *CoRR*, abs/1211.0053, 2012.

[9] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *CoRR*, abs/1606.09375, 2016.

[10] Nicolas Brodu and Dimitri Lague. 3d terrestrial lidar data classification of complex natural scenes using a multi-scale dimensionality criterion: applications in geomorphology. *CoRR*, abs/1107.0550, 2011.

[11] Carlos Becker, Nicolai Häni, Elena Rosinskaya, Emmanuel d'Angelo, and Christoph Strecha. Classification of aerial photogrammetric 3d point clouds. *CoRR*, abs/1705.08374, 2017.