INF574

PointNet++, a data-driven Point cloud Segmentation technique

December 2022

Virgile Foussereau, Adrien Ramanana Rahary

Contents

Acknowledgements

We would like to thank Mathieu Desbrun, Professor of INF573, for his lectures and advices. We also want to thank Pooran Memari and Jiayi Wei for their support.

1 Goals, state-of-the-art, and introduction to PointNet++

Point cloud data is a common format for geometric information, but its irregular structure can make it difficult to work with. To overcome this, many techniques rely on converting point cloud data to regular 3D voxel grids or collections of images. Unfortunately, this can make the data unnecessarily large and most often leads to a loss of information about the geometry. To avoid such issues, Charles R. Qi et al. introduced PointNet in 2017 [1], a novel way to process point clouds by using neural networks. This new technique performed better than state of the art methods on various tasks, including shape classification, part segmentation or semantic segmentation. The main limitation of PointNet provided by the authors was the fact that it struggled to capture the fine-grained details of the processed point cloud. To improve upon this technique, the authors came up with a new paper shortly after: PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space [2].

Unlike PointNet, PointNet++ is a hierarchical neural network which is able to process point clouds at multiple scales. Thanks to simple geometry processing steps, PointNet++ manages to capture both local and global features of a point set. Besides, PointNet++ is also efficient on point clouds with varying density of points. Such a feat is particularly interesting considering that most point set acquisitions lead to non-uniform densities of points. As part of our INF574 project, we focused on implementing PointNet++ as a part segmentation tool.

PointNet++ has shown segmentation results above state-of-the-art methods when published in 2017. Since then, several methods have reached higher precision. In January 2021, a new deep learning architecture called PIG-Net reached 90.5% of precision over ShapeNet dataset [3]. In PIG-Net, the local features are extracted from the transformed input points using the proposed inception layers and then aligned by feature transform. These local features are aggregated using the global average pooling layer to obtain the global features. Finally, the concatenated local and global features are fed to the convolution layers for segmenting the 3D point clouds. To this day, no other methods have performed better.

2 Description and implementation

As a neural network, PointNet++ can be decomposed into two main steps: the Set Abstraction (SA) and the Feature Propagation (FP). During the Set Abstraction, PointNet++ computes the features of multiple regions of points at multiple scales. This is done by iteratively grouping neighbouring points and applying a traditional PointNet to those groups. By the end of the set abstractions, because of the iterative grouping of points, we are only left with a few remaining points. Since our goal is to achieve part segmentation, it is necessary to provide each point of the original point set with a feature. The feature propagation step serves this purpose as it propagates the features of the few remaining points to all the points in the original point set. A complete overview of the process is presented in figure 1.

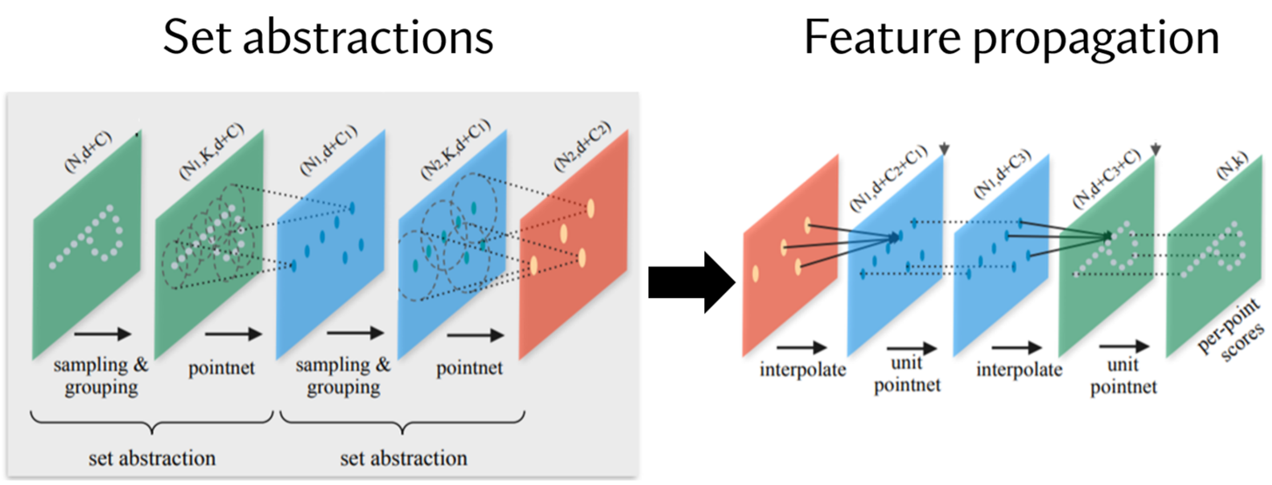

Figure 1: PointNet++ overview for part segmentation

Our implementation takes advantages of numpy arrays and methods which are heavily optimized and therefore lead allow for better time performance.

2.1 Set Abstraction

The Set Abstraction can be decomposed into 3 steps: point sampling, point grouping and feature computation.

2.1.1 Sampling

Point sampling is performed using a technique called Farthest Point Sampling (FPS). This algorithm consists in randomly picking a first point in the point set and finding the point in the point set which is the farthest to it. Iteratively, new points are added by looking for the farthest point to the set of sampled points being created. The number of points K is a hyperparameter chosen by the user. The final set of sampled points will now be referred to as the centroids. In figure 2, we present a simple example for a better understanding of the method.

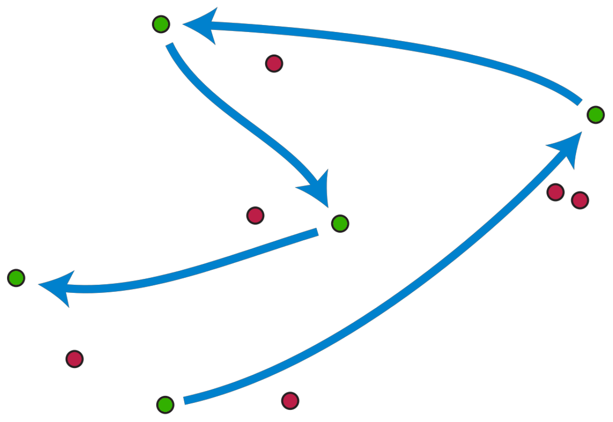

Figure 2: Farthest Point Sampling (FPS)

2.1.2 Grouping

Point grouping is performed using a technique called Ball Query. For each centroid sampled in the previous step, ball query finds the points that lie within a sphere centered around the centroid and of radius r (hyperparameter). The centroid and its neighbours are now considered as a group of point for which features need to be computed. In practice, the number of neighbours retrieved by the ball query is limited to 3. Figure 3 illustrates the ball query method.

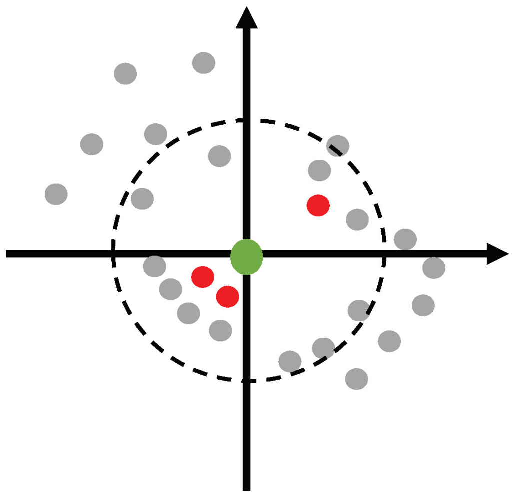

Figure 3: Ball query around a centroid (green) with a maximum of 3 returned points (red)

While an intuitive approach for grouping the points would be to simply use the K-Nearest-Neighbours (KNN), ball query is preferred because it is robust to regions with a low density of points. However, our implementation still draws from the concept of KNN. Indeed, for each centroid we sort all the other points by increasing distance. Such a step is the starting point of KNN. Once that is done, we take the first 3 points *i.e.* the 3 closest to the centroid, and check if they lie within the sphere of radius r. The ones that do are returned as the result of the ball query. Methods that come with the Numpy package were extremely handy to sort the arrays efficiently and easily from a programming standpoint.

2.1.3 PointNet

This layer is based on the PointNet architecture [1]. It is a neural network that takes a set of points, their coordinates and their features if they have any, and compute new features based on these inputs. At the end, a single maxpool is applied to obtain a vector of features that will represent the entire point cloud given as an input to Poinet. A simple version of this architecture is present in 4.

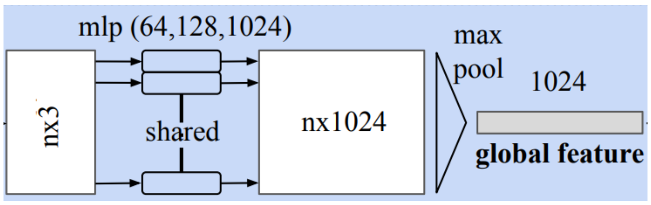

Figure 4: Pointnet Layer

In our case, we apply PointNet to each group computed in the grouping step. Before applying the multi-layer perceptron, we transform our system of coordinates to place the centroid as the origin (0,0,0) of each group. The vector obtained for each group is considered as the features of the centroid of the group. We obtain a new point set with only the centroids and their features. We can then iterate the whole abstraction layer until we get only one point and its features. The last abstraction layer, which takes the last remaining centroids and compute the features of the last single point, is called a *Global Abstraction Layer*. This Global Abstraction Layer is basically tantamount to the classic PointNet, as no sampling and grouping are done.

As said above we used the Pytorch library for our implementation. We created a PointNet class that inherits from the module class of Pytorch. Our PointNet class is adaptive, meaning that we can initialized it with as many layers as needed. For part segmentation, which is our goal, the number of layers will always be 3 but this number may change for other applications of PointNet++. Each layer of neurons is fully connected and followed by a ReLU activation function, except for the last layer.

At the end of PointNet, we take the max of the features obtained as the vector that will represent the pointset given in input.

2.2 Feature Propagation

At the end of the Set Abstraction steps, we are only left with a few points characterized by some features. Since our goal is to classify the whole point cloud, it is necessary to propagate the features back to the original point set so that all the points have their own features. This step is called the Feature Propagation. It consists in two steps which can be summed up as an interpolation step and a unit Pointnet.

2.2.1 Interpolation

To provide all the points in the original point set with features, the technique used by the authors of the paper is to go backwards in the point sets of the Set Abstractions, and to interpolate features from the smaller point set to the one with more points. For each point in the larger point set, a new set of features is concatenated to the already existing features of the point. To do so, the authors interpolate the features from all the points in the smaller point set using weights based on the inverse of the distance squared between the points. Figure 5 provides a schematic view of how features are interpolated using points from the previous the step of the reversed set abstraction.

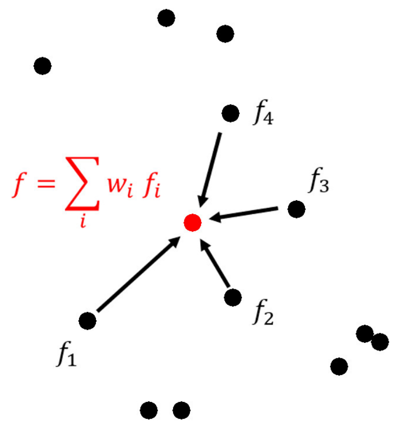

Figure 5: Propagation of the interpolation. Weights rely on the inverse of the distances squared

In their paper, the authors do not mention what happens when one of the distance is 0, *i.e.* when a point belongs to both the smaller and the larger point set. It seemed natural to us that such a point should keep its features. Therefore, we simply replaced the 0 distance by a value really close to 0 in the computation of the weights to avoid a forbidden division. Since the inverse of the distance squared is used for the interpolation, this acts the same as just keeping the features of the point (the weight of that point is extremely large).

2.2.2 Unit PointNet

A unit PointNet is in fact very similar to a PointNet network. However, we do not apply a max pool a the end. We keep the features computed for each point so the number of points does not change. We also add drop out between hidden layers of the unit PointNet as suggested by the article to avoid overfitting.

The last unit PointNet ends with a linear layer (no ReLU activation function or drop out) that gives 50 values, corresponding to the probabilities that the point belongs to the corresponding 50 classes. We take the max probability to assign a class to the point.

3 Personal Contribution and results

To try a new approach, we decided to change the sampling step in the abstraction layer. Instead of using a deterministic Farthest Point Sampling, we developed a similar approach but more randomized. While the Farthest Point Sampling always picks the farthest point from the current selection of points C, our algorithm picks any remaining point i with probability P(C,i) where:

$$P(C,i) = \frac{d(C,i)^2}{\sum\limits_{j \notin C} d(C,j)^2}$$

with d(C,i) the distance between point i and current selection of point C.

This algorithm is, for instance, used in K-means++ initialization [4]. We choose to implement this new method because it can help to avoid choosing outliers. Indeed, in some cases, the farthest point is not the best centroid candidate as it might be too far out, away from any interesting local structure. With our method, outliers are less likely to be picked. Our guess is that this sampling could improve the model performance in some cases.

To test our implementation, we downloaded the dataset ShapeNetPart provided by [5]. ShapeNetPart dataset contains 16,881 pre-aligned shapes from 16 categories, annotated with 50 segmentation parts in total. Most object categories are labeled with two to five segmentation parts. We chose to work on the airplane part segmentation, which is a sufficiently complex shape to assess PointNet++ performance.

After 2 epochs of training, each using close to 2000 airplanes pointsets, we obtain an accuracy of 86.3 % on the validation set which contains 391 pointsets. Using our modified model, we achieve 86.5 % which is indeed a slight improvement. We provide a resulting visualization of part segmentation for each models in figure 6.

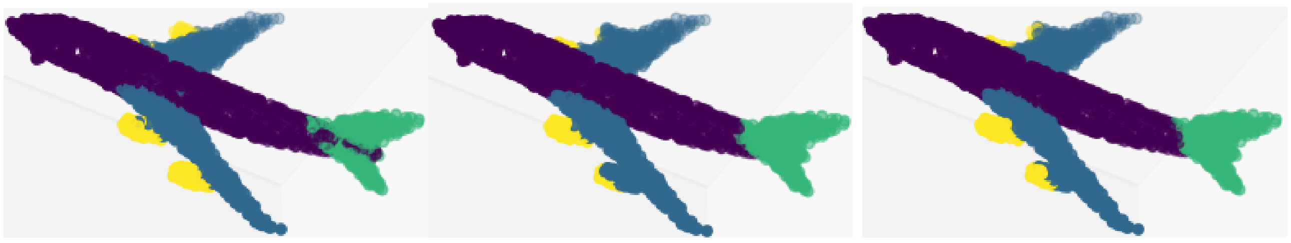
(a) Ground Truth (b) PointNet++ (c) Modified PointNet++

Figure 6: Part Segmentation results

When it comes to the performance of our implementation, a single iteration during the training on the plane model took on average 700ms while the authors' takes about 100ms. Such a comparison might not make much sense though: the specs of the machines used differ. More importantly the computations we did ran on the CPU while those of the authors ran on a GPU. What is clear though, is that had we used standard Python lists rather than Numpy arrays, our implementation would have been much slower (approximately 30 times slower).

Conclusion

PointNet++ is a variation of the original PointNet architecture that improves upon it in several key ways. The main advantage of PointNet++ is its ability to handle point cloud data of varying densities, sizes, and scales. This is achieved through a hierarchical neural network architecture that processes the data at multiple levels of resolution.

Our implementation of PointNet++ in Python, based on Numpy and PyTorch packages, is functional and was able to accurately segment the parts of several complex shapes such as planes for instance. The accuracy of our model compares with that claimed by the authors.

We also explored a small variation of PointNet++ by taking inspiration from a technique called K-Means++. By slightly changing the sampling method used by the authors for a randomized version that avoids outliers, we managed to get a tiny improvement of accuracy on a shapes from the ShapeNet dataset.

References

[1] Charles R. Qi et al. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. 2016. DOI: `10.48550/ARXIV.1612.00593`. URL: `https://arxiv.org/abs/1612.00593`.

[2] Charles R. Qi et al. *PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space*. 2017. DOI: `10.48550/ARXIV.1706.02413`. URL: `https://arxiv.org/abs/1706.02413`.

[3] Sindhu Hegde and Shankar Gangisetty. "PIG-Net: Inception based deep learning architecture for 3D point cloud segmentation". In: *Computers and Graphics* 95 (2021), pp. 13–22. ISSN: 0097-8493. DOI: `https://doi.org/10.1016/j.cag.2021.01.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0097849321000042`.

[4] David Arthur and Sergei Vassilvitskii. "K-Means++: The Advantages of Careful Seeding". In: vol. 8. Jan. 2007, pp. 1027–1035. DOI: `10.1145/1283383.1283494`.

[5] *Point Cloud Datasets*. `https://github.com/AnTao97/PointCloudDatasets`. Accessed: 2022-11-10.