RWTH AACHEN UNIVERSITY

Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Informatik 8 (Computer Vision)
Fakultät für Mathematik, Informatik und Naturwissenschaften
Prof. Dr. Bastian Leibe

Master Thesis

Neighborhood Pooling in Graph Neural Networks for 3D and 4D Semantic Segmentation

vorgelegt von

Kushal Sanjay Sharma
Matrikelnummer: 384295
2020-01-31

Erstgutachter: Prof. Dr. Bastian Leibe
Zweitgutachter: Prof. Dr. rer. nat Benjamin Berkels

Eidesstattliche Versicherung

Kushal Sanjay Sharma	384295
Name	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Neighborhood Pooling in Graph Neural Networks for 3D and 4D Semantic Segmentation

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 2020-01-31	
Ort, Datum	Unterschrift

Belehrung:

§ **156 StGB: Falsche Versicherung an Eides Statt**
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ **161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt**

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten dementsprechend.

Die vorstehende Belehrung habe ich zur Kentnis genommen:

Aachen, 2020-01-31	
Ort, Datum	Unterschrift

Abstract

In this thesis I explore how to leverage spatial geometry of point-sets to improve performance of graph neural networks using nearest neighbor graphs. I demonstrate how the proposed approach is able to beat previous graph-based approaches on S3DIS dataset for the task of 3D semantic segmentation. I then generalize the proposed approach so it can also leverage temporal information in a series of point-sets such sequential outdoor laser-scans obtained by a LiDAR scanner for the task of semantic segmentation. This type of segmentation is best classified as 4D semantic segmentation and I demonstrate how my proposed generalization to 4D is able to improve performance on 3D baseline models for this task.

Acknowledgements

I would like to thank my supervisor Francis Engelmann for supervision and feedback through the thesis and also for offering me a student job that eventually led to this work. I would like to thank Prof. Bastian Leibe for providing the opportunity to work on this topic at the chair. I would like to thank Mark Weber for coding advice. I would like to thank my mom and dad who pushed me to pursue a Masters degree in the first place. And lastly, thank you Kilkenny for not hanging on me.

Contents

1 Introduction

Deep learning has led to a paradigm shift in the field of computer vision. After the success of AlexNet [KSH12] in 2012 ILSVRC challenge, deep learning has been successfully applied to several tasks in 2D computer vision such as image classification, object detection and semantic segmentation. Increasing computational power and availability of easy-to-use frameworks such as PyTorch [PGM+19] and TensorFlow [AAB+15] have further helped accelerate the adoption of deep learning. 3D computer vision is an important subfield of computer vision where the aim is to understand 3D data such as LiDAR and Radar scans. Understanding 3D data is particularly important for self-driving cars since a LiDAR scanner is one of the sensors in most self-driving cars. In this thesis, I focus on the task of 3D semantic segmentation on indoor and outdoor scenes. Semantic segmentation is the task of assigning meaningful labels to each point in an input point set. The set of labels used is application dependent. For example, the set of labels for indoor semantic segmentation will include chair, table, floor, ceiling etc whereas for self-driving cars this set will include categories such as pedestrians, small vehicles, cars, road, sidewalk and so on. In the context of autonomous cars, semantic segmentation is best classified as 4D semantic segmentation since it involves performing semantic segmentation on a sequence of laser scans and leveraging temporal information. In this thesis, I propose a generalization of my approach to 3D semantic segmentation that is also applicable to 4D.

1.1 Contributions

First, I will first give an overview of deep learning in the context of 3D vision and the challenges posed. Then I will review the current deep learning methods for 3D semantic segmentation. Then I will propose an approach to 3D semantic segmentation using graph neural networks which allows us to leverage progress made in 2D vision and generalize it to 3D. I demonstrate how the proposed method is competitive with other methods in this direction on both computational

time and performance on indoor 3D data. Then I propose a further generalization of my approach applicable to 4D data and present the results on outdoor LiDAR scans on a newly released dataset.

1.2 Overview

In Chapter 2, I present an quick overview of deep learning with an emphasis on the problem of applying deep learning to 3D data. To understand these concepts and challenges is a pre-requisite for understanding the necessity of designing new algorithms for 3D data and also the methods outlined in this thesis.

In Chapter 3, I review the methods that have been proposed for the challenge of 3D semantic segmentation. In this chapter, I also present a broad classification of 3D deep learning methods and specify the direction I will be taking and my motivation for doing so.

In Chapter 4, I present a detailed explanation of my proposed method. In this chapter, I show how my method leverages existing work and I generalize it to 3D and 4D.

In Chapter 5, I present a quantitative and qualitative overview of the results obtained from my experimentation. I examine the performance of the proposed method on 3D data first and then move to 4D. Chapter 6 concludes this thesis and outlines further directions for future work that could further boost inference speed and performance.

2 Background

In this section I will introduce the basics of deep learning. I will go over the fundamental operations that are used to make deep neural networks as used in 2D vision and explain graph neural networks. One of the first architectures to be applied successfully in computer vision was LeNet [LBB+98] which used a convolutional architecture. Convolutional Neural Networks (CNNs) saw heavy use in the 90s but then fell out of favour with the introduction of support vector machines. The core method used to train deep neural networks is back propagation. Interest in deep learning was revived when AlexNet [KSH12] won the 2012 ILSVRC contest with a huge margin. One of the key contributions of their work was using GPUs for speeding up computation. With increasing computational power and availability of developer-friendly deep learning framework such as TensorFlow [AAB+15] and PyTorch [PGM+19], neural networks have now become one of the key parts of several computer vision algorithms.

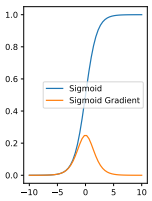
2.1 Fully Connected Neural Networks

The simplest type of neural network is a single layer fully connected network which can mathematically be expressed as follows

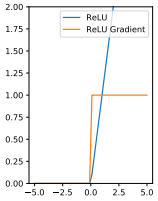
$$\mathbf{z}^{i+1} = g\left(\mathbf{W} \cdot \mathbf{z}^{i} + \mathbf{b}\right) \tag{2.1}$$

where $\mathbf{z}^{i} \in \mathbb{R}^{m}$ is the input, $\mathbf{z}^{i+1} \in \mathbb{R}^{n}$ is the output, $\mathbf{W} \in \mathbb{R}^{nxm}$ is the weight, $\mathbf{b} \in \mathbb{R}^{n}$ is the bias and g is some non-linearity applied to the input after a doing a linear transformation. This type of network is called as such because every component in the output $\mathbf{z}^{i+1}$ depends upon every component of the input $\mathbf{z}^{i}$. The earliest idea of a fully connected network could be traced to the perceptron [Ros58] by Rosenblatt. This network has a single layer and used a step-function as the non-linearity.

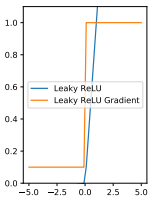
$$f\left(x\right) = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{2.2}$$



(a) Behaviour of the sigmoid function. Notice that the gradient reaches a maximum value at 0 and drops to zero at the ends.



(b) Behaviour of ReLU non-linearity. The gradient is zero for negative values.



(c) The leaky ReLU non-linearity allows for a constant finite gradient even at values less than zero and avoids the problem of dead neurons.

Modern neural networks are usually several layers deep and employ other non-linearities most common of which are the ReLU (Equation 2.3) and sigmoid (Equation 2.4).

$$\text{ReLU}(x) = \begin{cases} 1, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{2.4}$$

The sigmoid function saturates at the ends of its domain as shown in Figure 2.1a whereas the ReLU is an unbounded non-liearity with non-zero derivative in one half of its domain (Figure 2.1b). ReLU non-linearity is usually considered better for convergence because of its non-saturating nature. However, once the value of ReLU goes below zero, it stops receiving further gradient updates and the neuron is effectively "dead". More non-linearities have been proposed in the literature such as the Leaky ReLU which prevents the problem of dead neurons (Figure Figure 2.1c).

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) utilize a convolution operation followed by subsequent down-sampling. A discrete convolution is defined as in Equation 2.5.

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n) \tag{2.5}$$

where S is the output, K is the convolution kernel and I is the input. However, most machine learning libraries implement a related operation called correlation (Equation 2.6).

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{2.6}$$

Fully connected neural networks as defined in Equation 2.1 do not scale very well as the dimensionality of the input increases. Consider a single layer fully connected neural network as in Equation 2.1. This layer will have nx$m + n$ number of parameters while counting the bias b. Thus if we have a L layered fully connected network with feature dimensions $d_{\text{in}}, d_2, d_3..$, then the total number of parameters in the network while not counting the bias parameters will be d_1xd_2 + d_2xd_3... In the case of images the inputs are very high dimensional and hence directly using fully connected network will lead to a model with very large number of parameters. Convolutional neural networks solve this problem by utilizing a convolution operation followed by down-sampling. In this case the number of parameters is not dependent on the size of the input but of the number of input and output feature channels. Consider a convolutional neural network with an RGB image as input and output feature channels f_1. In this case the number of parameters of the network will be 3xf_1. Typically $f_1 << d_1$ which reduces the number of parameters that CNNs require. The reason that even with a reduced set of parameters CNNs are effective is because in images neighborhood information is usually sufficient to derive good features about points in the input.

2.3 Graph Convolutional Neural Networks

A graph is a tuple $G = (V, A, E)$ where V is the set of vertices, E is the set of edges and A is the adjacency matrix associates each edge $e \in E$ with a weight. Each node is the graph $v_i \in V$ can be associated with a feature $f_i \in \mathbb{R}^D$. A graph convolutional neural network or GCN takes as input a graph G and learns corresponding features f_i for each for each vertex.
A spectral GCN utilizes the spectral decomposition of the normalized graph Laplacian matrix [WPC+19] $\mathbf{L} = \mathbf{I}_n - \mathbf{D}^{\frac{1}{2}}\mathbf{A}\mathbf{D}^{\frac{1}{2}}$ where $\mathbf{D}$ is a diagonal matrix of node degrees $\mathbf{D}_{ii} = \sum_j (\mathbf{A}_{i,j})$. $\mathbf{L}$ is symmetric positive definite so it can be expressed in spectral form $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathbf{T}}$ [WPC+19] where $\mathbf{U}$ is a unitary matrix. A graph signal $x \in \mathbb{R}^N$ is a feature vector over the nodes of the graph G. The graph Fourier transform of $\mathbf{x}$ is defined as $\hat{x} = \mathcal{F}(x) = \mathbf{U}x$ and the inverse transform is defined by $\mathcal{F}^{-1}(\hat{x}) = \mathbf{U}^T\hat{x}$. The graph convolution with a filter $g \in \mathbb{R}^N$ [WPC+19] is defined as

$$x \star g = \mathcal{F}^{-1}(\mathcal{F}(x) \odot \mathcal{F}(g)) \tag{2.7}$$

$$= \mathbf{U}^T(\mathbf{U}x \odot \mathbf{U}g) \tag{2.8}$$

Let $g_\theta = diag\,(\mathbf{U}g)$, where $diag\,()$ denotes a function that creates a diagonal matrix, then graph convolution can be simplified as [WPC+19]

$$x \star g_\theta = \mathbf{U}^T g_\theta \mathbf{U} x \tag{2.9}$$

Spectral graph convolutions differ in the choice g_θ but follow the same formulation. Spectral graph convolution networks rely on the eigen-decomposition of the Laplacian matrix meaning the Fourier basis needs to be recomputed for any changes to the graph. Since eigen-decomposition requires a $\mathcal{O}\,(N^3)$, spectral graph convolutions are unsuitable for computer vision since graphs typically need to be computed as run time meaning every inference step will require performing a new eigen-decomposition.
The second approach to graph learning is spatial graph convolution. In its most general form a spatial graph convolution over a graph $G = (V, E)$ is defined as follows [WSL+19] -

$$h\,(x_i) = \square_{j:(i,j)\in E} g\,(x_i, x_j) \tag{2.10}$$

where x_i is an input features associated with vertex $v_i \in V$, h denotes a feature transform of x_i $\square_j$ is some aggregation function like maximum or mean and g is a neural network. The type that I use in my experiments is edge convolutions where

$$g\,(x_i, x_j) = g_{ec}\,([x_i, x_i - x_j]) \tag{2.11}$$

where $[\cdot]$ represents a feature concatenation and g_{ec} is a neural network which acts on this concatenated feature set. Spatial graph convolutions are simpler than spectral graph convolution networks and do not require an expensive decomposition that needs to be re-computed with every update to the graph. This makes spatial graph convolutions more generally applicable and more computationally efficient. Furthermore, frameworks such as [FL19] and [Fey] make spatial graph convolution networks easier to implement.

2.4 Loss functions

In a supervised learning setting, the final output of a neural network is a probability distribution of predictions. To train these networks the problem is converted into a minimization problem with a loss function that needs to be minimized. The most common loss functions are listed below.

Squared Loss

Let $\hat{y}$ be the output of a model and y be the ground truth. Then the squared loss defined by

$$L_{mse}\,(y, \hat{y})_{mse} = \sum_j (y_j - \hat{y}_j)^2 \tag{2.12}$$

Cross Entropy Loss

For classification problems the most common loss that is used is cross-entropy loss which is defined by

$$L_{ce}\left(y, \hat{y}\right) = \sum_{j} \hat{y}_j \ln\left(y_j\right) \tag{2.13}$$

The above loss terms are often paired with regularizers such as L2 where a penalty is placed on the L2 norm of the weights. An example of pairing cross-entropy loss with the L2 regularizer would be the

$$L = L_{ce} + \lambda L_{L2}, \tag{2.14}$$

$$= \sum_{j} \hat{y}_j \ln\left(y_j\right) + \lambda \|\mathbf{w}\|^2 \tag{2.15}$$

Another type of penalty on the parameters would be the L1 loss. But since the L1 is not differentiable at the origin, a modification called the smooth L1 is often used. The smooth L1 function is defined as

$$\text{smooth}_{L1}\left(x\right) = \begin{cases} |x| - 0.5 & \text{if } |x| \geq 1 \\ 0.5x^2, \text{otherwise} \end{cases} \tag{2.16}$$

One problem with cross-entropy loss is that it does not penalize situations where the prediction of one point in the input set is different from others in its neighborhood i.e it does not enforce neighborhood consistency. Due to this limitation for dense prediction problems such as semantic segmentation, a neural network trained using the cross-entropy loss requires a locality enforcing component such as conditional random fields.

2.5 Back-propagation

With the help of loss functions, prediction or classification problems in machine learning can be posed as an optimization problem or more specifically as minimization problem. As long as all operations being used in the model are differentiable, gradient based optimization could be used to minimize the loss. This made possible by the application of chain rule of differential calculus and the method is called back-propagation. The use of back-propagation for learning representations on a neural network was first show in [RHW86]. The chain rule in differential calculus could be stated as

$$\frac{\partial y}{\partial x} = \sum_{j} \frac{\partial y}{\partial z_j} \frac{\partial z_j}{\partial x} \tag{2.17}$$

As long as both $\frac{\partial z_j}{\partial x}$ and $\frac{\partial y}{\partial z_j}$ exist, the chain rule is applicable. Back-propagation provides an efficient means of computing derivatives. Consider the following example from [NW]

$$y = \left(x_1 x_2 \sin x_3 + \mathrm{e}^{x_1 x_2} / x_3\right) \tag{2.18}$$

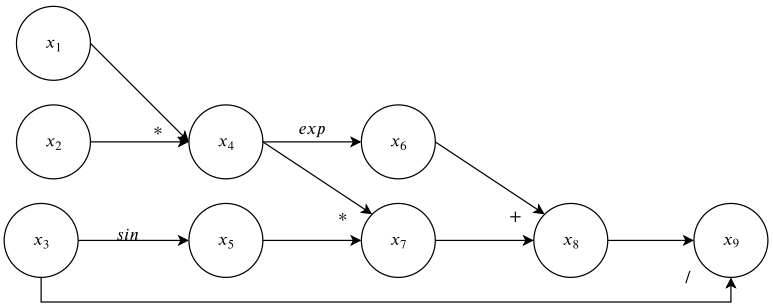


Figure 2.2: Computational graph of a function of several variables. Example and figure from [NW].

This computation could be encoded as a graph as shown in Figure 2.2 where the intermediate variables are given by

$$x_4 = x_1 \cdot x_2 \tag{2.19}$$
$$x_5 = \sin x_3 \tag{2.20}$$
$$x_6 = \mathrm{e}^{x_4} \tag{2.21}$$
$$x_7 = x_4 \cdot x_5 \tag{2.22}$$
$$x_8 = x_6 + x_7 \tag{2.23}$$
$$y = x_8 / x_3 \tag{2.24}$$

To compute $\frac{\partial y}{\partial x_6}$, we can first compute $\frac{\partial y}{\partial x_8}$ and then use the chain rule

$$\frac{\partial y}{\partial x_6} = \frac{\partial y}{\partial x_8}\frac{\partial x_8}{\partial x_6} \tag{2.25}$$
$$= \frac{1}{x_3} \cdot 1 \tag{2.26}$$

In the case of x_6, there is only one path from x_6 to y. If there are multiple paths from any node in the computational graph to the target variable, then the derivatives obtained along each path are added. For example in the case of x_4 ,

$$\frac{\partial y}{\partial x_4} = \frac{\partial y}{\partial x_6}\frac{\partial x_6}{\partial x_4} + \frac{\partial y}{\partial x_7}\frac{\partial x_7}{\partial x_4} \tag{2.27}$$
$$= \frac{1}{x_3} \cdot x_5 + \frac{1}{x_3} \cdot \mathrm{e}^{x_4} \tag{2.28}$$

Thus back-propagation allows us to re-use intermediate derivative computations. In automatic differentiation parlance, this is known as adjoint mode and is the predominant mode used in deep learning software.

2.6 Optimizers

The basic rule for gradient based optimization is

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha \cdot \nabla_{\mathbf{w}} L \tag{2.29}$$

where L is the loss function that needs to be minimized. α is called the learning rate and controls the step-size. The different optimizers differ in their method of estimating $\nabla_{\mathbf{w}} L$.

One of the simplest ways of estimating $\nabla_{\mathbf{w}} L$ is stochastic gradient descent or SGD [RM51] where the parameter update rule is

$$w^{t+1} = w^t - \alpha \cdot g_t \tag{2.30}$$

where g_t is the gradient at time t, w_t is the vector of model weights at current time step and w^{t+1} is the updated weight. SGD is sensitive to learning rate α - too high a learning rate can cause instability during training and too low can lead to slower convergence. Several alternative optimizers have been proposed in literature. I give the basic idea behind Adam [KB15] below since this is the optimizer that I use for all of my experiments. Adam computes an unbiased estimate of the gradient and the second moment of the gradient using exponential moving averages. Parameters $\beta_1, \beta_2 \in [0, 1)$, control the weights in the window. The update rules are as follows [KB15]

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{2.31}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{2.32}$$

$$\hat{m}_t = \frac{m_t}{(1 - \beta_1^t)} \tag{2.33}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.34}$$

$$w_t = w_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.35}$$

where m_t is the second moment of the gradient, g_t is the present value of the gradient and w denotes the model parameters. The inital values of m_t and v_t are set to 0, so the update rules for v_t and m_t correspond to doing an unbaised estimate of the first and second moment using weights $1, \beta_1, \beta_1^2, \ldots$ and $1, \beta_2, \beta_2^2, \ldots$ respectively with 1 being the weight given to the current estimate and decresing exponentially afterwards.

3 Related Work

In this section, I give an overview of all the current approaches to 3D semantic segmentation. Point-sets have the following three properties that must be taken into account by any method that intends to learn meaningful features.

- Unordered [QSMG] - Unlike 2D image data, 3D data typically lacks a regular structure. A point-set is an uordered collection of points and is not localized like 2D data. This means that 2D image signal is limited to a fixed grid whereas point-sets typically consist of points that are more scattered especially outdoor point-sets.

- Concept of neighborhood [QSMG] - Point clouds are subspaces of $\mathbb{R}^3$ with a distance metric attached. A collection of points forms a meaningful subset and feature representations should reflect this property. For example, features corresponding to different point-sets should not be too "different" as measured by some metric and feature-learning methods must take neighborhood information into account. This is one property that is shared by 2D data as well.

- Affine invariance [QSMG] - Feature representations should be invariant to non-deforming affine transformations (rotation and translation) of the whole point-set in certain. This may not be the case for shearing deformations. Although this is not always true for example translating a floor can convert it into a roof however, for a scene which has both a roof and floor any feature learning method should be able to learn different representations for points belonging to the two categories which are invariant to rotation and translation. Depending upon use-case, this is a property expected of 2D feature learning methods too.

- Sparsity - This is by far the most distinguishing property of point-sets when compared with 2D data. The 3D volume inside which the point cloud resides has is not densely packed. This does not mean that point clouds do not have

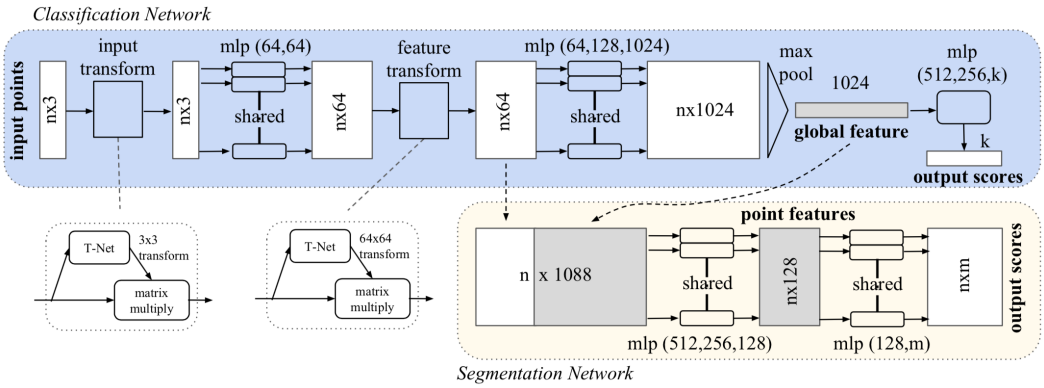


Figure 3.1: PointNet architecture used in [QSMG]. Ablation studies conducted by the author suggest that the intermediate T-Nets used are important for achieving a better score on S3DIS and other benchmarks. Figure taken from [QSMG]

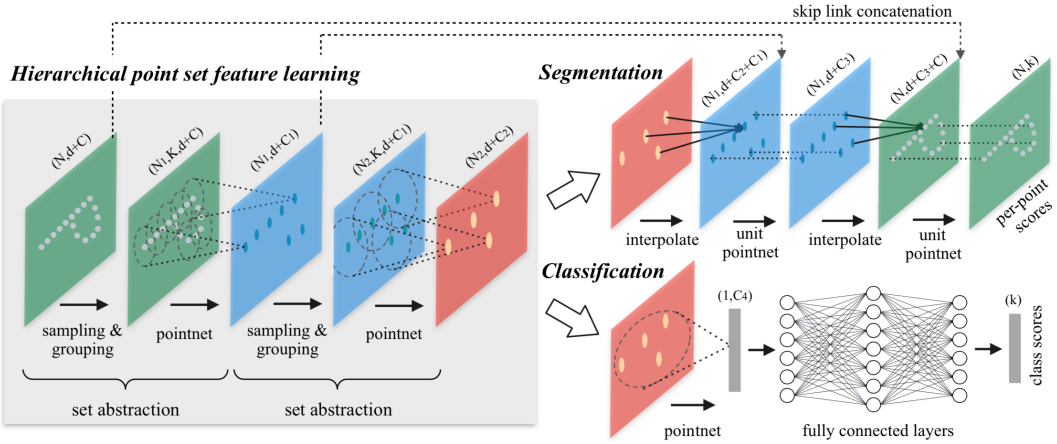


Figure 3.2: PoinetNet++ architecture introduced in [QYSG17]. The difference here wrt PoinetNet is that pooling is introduced gradually rather than having a single global layer of pooling as in PointNet. The authors also define nearest neighbor based interpolation scheme for upsampling and down-sampling. Figure from [QYSG17].

high local density. It is possible that point clouds have high local density but the overall volume occupied by the points is very low. This means that naively imposing a grid on a point cloud and using 3D convolutions is computationally inefficient since most of the time the method will be computing on empty voxels.

3.1 PointNet

PointNet [QSMG] and PointNet++ [QYSG17] represents a milestone in deep learning for 3D data. The idea behind PointNet was to use a shared MLP as a feature learner for local representations followed by a global pooling layer to learn a context vector which is the concatenated with local features to learn a global representation for each point. The concatentated features are then used for tasks like 3D semantic segmentation, shape classification etc. PointNet consists of two key modules: the max pooling layer as a symmetric function to aggregate information from all the points, a local and global information combination structure, and two joint alignment networks that align both input points and point features as shown in Figure 3.1. PointNet could be formally written as in Equation 3.1 [QSMG] .

$$f\left(\{x_1, .., x_n\}\right) \approx g\left(h\left(x_1\right), .., \left(x_2\right)\right) \tag{3.1}$$

where $f : 2^{\mathbb{R}^N} \rightarrow \mathbb{R}$, $h : \mathbb{R}^N \rightarrow \mathbb{R}^K$ and $g : \mathbb{R}^K x..\mathbb{R}^K \rightarrow \mathbb{R}$ is some symmetric function. PointNet++ [QYSG17] improved upon this idea by introducing feature learning at multiple scales. Unlike PointNet which consists of a single global level of pooling, PointNet++ is composed by a number of down-sampling and up-sampling layers which the authors refer to as "abstraction levels"(see Figure 3.2). After each down-sampling layer the down-sampled features are passed through a feed-forward neural network similar to PointNet. For down-sampling the author use furthest point sampling algorithm is listed in 1 [WS10].

Algorithm 1 Furthest point sampling

1. Pick arbitrary $x \in P$.
2. $S \leftarrow \{i\}$
3. **while** $|S| < k$
   $j \leftarrow armax_{j \in P} d\left(x_j, S\right)$
   $S \leftarrow S \cup \{j\}$

As presented above Algorithm 1 is a 2-approximate solution to the k-center problem [WS10] which deals finding a set of centroid points such that the maximum of a point to the nearest cluster is minimized. Feature up-sampling in PointNet++ consists of a inverse distance based interpolation kernel as shown in Equation 3.2 [QSMG] where features from level l with N_l points are propagated to level $l-1$ with N_{l-1} points. This propagation is done until one reaches the top-most level i.e. reach N_1 points.

$$f^{(j)}\left(x\right) = \frac{\sum_{i=1}^{k} w_i\left(x\right) f_i^{(j)}}{\Sigma_{i=1}^{k} w_i\left(x\right)} \text{ where } w_i = \frac{1}{d\left(x_i, x_j\right)} \; j = 1, .., C \tag{3.2}$$

While PointNet computes a single global feature vector which is then broadcasted to all points in the point set, PointNet++ computes coarser representations

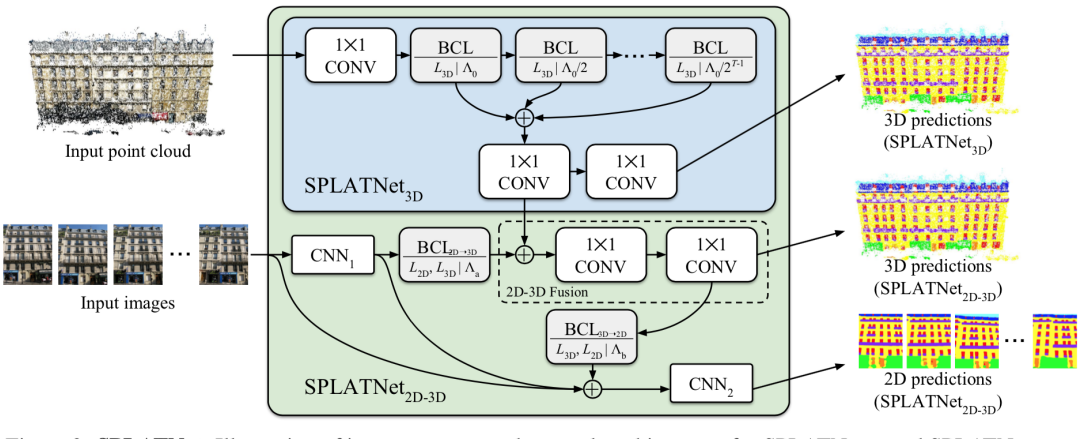


Figure 3.3: SplatNet architecture maps the input point cloud into an *nd*-lattice. It then used convolutions over this intermediate lattice to learn lattice level features which are then projected back to the input pionts. Figure from [SJS+18].

step-by-step utilizing subsequent sampling layers. The input to each sampled point set is an K-nearest neighbors interpolated feature vector from the point set in the previous step. Using the farthest point sampler does however introduce some stochasticity in the feature computation since the output of the sampler depends upon the first point that is selected.

3.2 SplatNet/OctNet

OctNet [RUG17] maps the input 3D point-set onto an octree and then defines convolution operation using the constructed octree as an intermediate representation. Each point in the input point-set is mapped to some node of an octree and feature representations are either down-sampled or up-sampled between the points mapped to the same octant. Convolution operations require frequent access to neighboring points so the authors propose a modified version of an octree called a grid-octree which allows to serialize an octree as a sequence of 0s and 1s. This sequence is obtained by doing a level-order traversal of the underlying octree and adding a 0 to the serialized sequence for a empty octant and 1 for an octant that is occupied by point(s). For a sufficiently shallow octree this sequence could be packed into an integer which makes it possible to traverse the octree using bit-operations(see Figure 3.5). This representation make memory accesses more efficient than the usual implementations where octree is implemented using pointers leading to random memory accesses which are significantly slower. Furthermore such a representation is more suitable for GPGPU (general purpose GPU) computations.

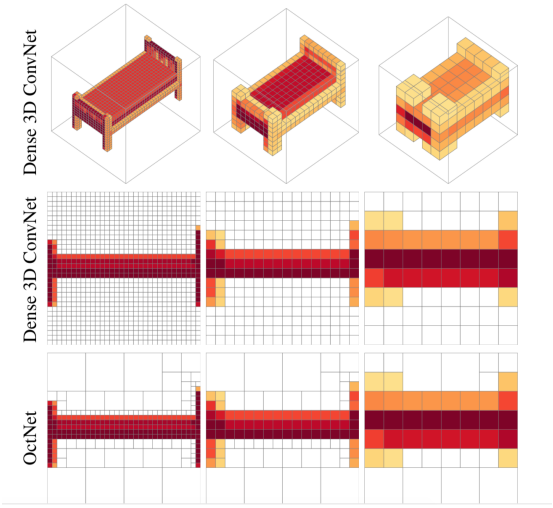


Figure 3.4: Illustration of the OctNet method. OctNet like SplatNet also maps the input point cloud into some intermediate representation. SplatNet uses a lattice while OctNet maps the input into a OctNet and defines convolutions over this OctNet. Figure from [RUG17].

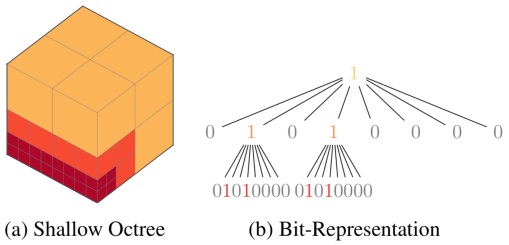


Figure 3.5: OctNet authors use a memory efficient representation of an octree called a grid octree. Since their idea is to pack an octree into a 64 bit integer, the authors only use an octnet of depth 3. Figure from [RUG17].

If $T_{(i,j,k)}$ [RUG17] denotes the value of a tensor at location I, j, k and $O\,(i, j, k)$ is the value in the smallest cell comprising voxel i, j, k, the mapping from grid-octree O to a tensor T [RUG17] is computed by [RUG17]

$$\text{oct2ten} : T_{i,j,k} = O\,[i, j, k]$$

and the reverse mapping can be computed by [RUG17]

$$\text{ten2oc} : O\,[i, j, k] = \text{pool\_voxels}\left(T_{\bar{i},\bar{j},\bar{k}}\right)$$

where $pool_voxels$ is some pooling function. Given these operations it is possible to map the convolution operation over a 3D tensor to corresponding operations

over the octree nodes. For a 3D tensor T convolution with a 3D convolution kernel $W \in \mathbb{R}^{LxMxN}$ [RUG17] can be written as [RUG17]

$$T^{out}_{i,j,k} = \sum_{l=0}^{L-1} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{l,m,n} \cdot T^{in}_{\bar{i},\bar{j},\bar{k}} \tag{3.3}$$

with $\bar{i} = i - l + \lfloor L/2 \rfloor$, $\bar{j} = j - m + \lfloor M/2 \rfloor$, $\bar{k} = k - n + \lfloor N/2 \rfloor$. The corresponding operation over the grid-octree data structure can then be defined as [RUG17]

$$T_{i,j,k} = \sum_{l=0}^{L-1} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{l,m,n} \cdot O^{in} \left[\bar{i}, \bar{j}, \bar{k}\right] \tag{3.4}$$

Similarly pooling and un-pooling can be defined as per Equation 3.5 [RUG17] and Equation 3.6 [RUG17].

$$O^{out}[i,j,k] = \begin{cases} O^{in}[2i, 2j, 2k] \text{ if } \mathrm{vxd}(2i, 2j, 2k) \leq 3 \\ P \text{ else} \end{cases} \tag{3.5}$$

$$O^{out} = O^{in}[\lfloor i/2 \rfloor, \lfloor j/2 \rfloor, \lfloor k/2 \rfloor] \tag{3.6}$$

where *vxd* computes the depth of the indexed voxel in the shallow octree. The limitation of OctNet is that the authors that the octree constructed on the points should be a shallow octree so that it can be serialized and fit into some given number of bits. The authors use an octree that is only 3 layers deep. In contrast to this method, I demonstrate that it is possible to stack multiple layers of grid-pooling and achieve better performance on 3D semantic segmentation tasks.

Permutohedral CNNs were proposed in [KJG15] where the authors propose a convolution operation of a d-dimensional input space that entirely works on a lattice. Input data is a tuple (f_i, v_i) of feature locations $f_i \in \mathbb{R}^d$ and corresponding signal values $v_i \in \mathbb{R}$. The input is then mapped to a permutohedral lattice. A convolution then operates on the constructed lattice and the result is mapped back to the output space. Hence, the entire operation consists of three stages [KJG15]

- splat (the mapping to the lattice space)
- convolution
- slice (the mapping back from the lattice)

The splat and slice operations take the role of an interpolation between the different signal representations. Input samples are first projected into the lattice

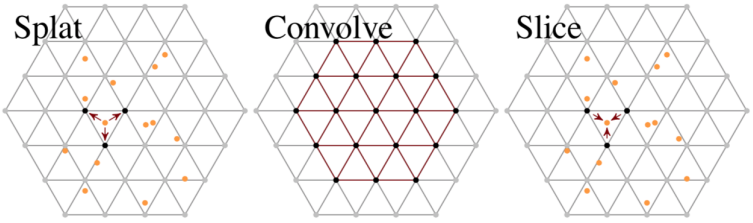


Figure 3.6: The permutohedral CNN pipeline as in [KJG15]. Figure from same source.

using barycentric interpolation in the splatting operation. For lattice point j, all input points that belong to a cell adjacent to are summed up as below [KJG15]

$$l_j = \sum_{i \in C(j)} b_{i,j} v_j \tag{3.7}$$

where $C(j)$ indicate cells adjacent to lattice point j. The reverse operation is slicing which maps signals in the lattice back to the input space again using barycentric interpolation [KJG15].

$$v'_k = \sum_{j \in C(k)} b_{k,j} l'_j \tag{3.8}$$

where $C(k)$ indicates the lattice points neighboring point k from the input set. As formulated above permutohedral convolutions can perform lattice sampling at multiple scales depending upon the scaling factor. This is analogous to dilated 3D convolutions. Using the operations defined above SplatNet [SJS+18], stacks several bilateral convolution layers together with different scales.The authors also features computed from a multi-view 2D images using 2D CNN based methods. In contrast to SplatNet, my method does not rely on any additional computed features other point-geometry.

3.3 Monte-Carlo Convolutions

Monte-Carlo Convolutions or MctConv starts with the definition of continuous convolution in 3D space and applies Monte-Carlo approximation on this integral. In continuous space convolution of a function f with a kernel g is given by

$$f \star g = \int f(y)\, g(x - y)\, dy \tag{3.9}$$

A Monte Carlo estimate for the integral Equation 3.9 is given by [HRV+]

$$(f \star g) = \frac{1}{|N(x)|} \sum_{j \in N(x)} \frac{f(y_j)\, g\left(\frac{x - y_j}{r}\right)}{p(y_j|x)} \tag{3.10}$$

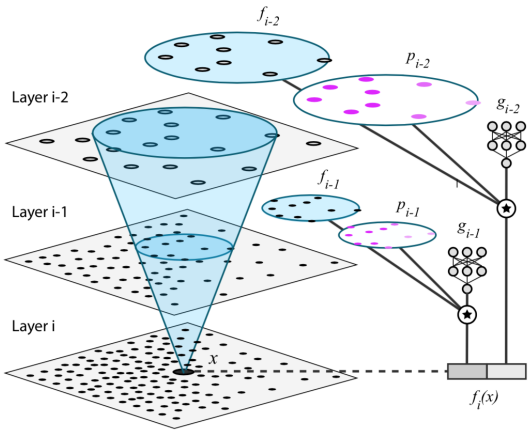


Figure 3.7: Multi scale feature learning as presented in [HRV+]. The receptive field content f_{i-1} and f_{i-2} along with their corresponding densities p_{i-1} and p_{i-2}. Each consists of a Monte Carlo convolution layer for feature learning followed by a concatenation to obtain point level features.Figure from [HRV+].

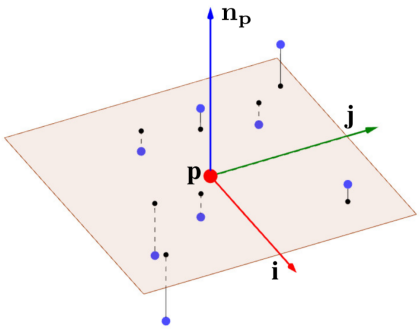
where $N(x)$ denotes the set of points within a sphere of radius r centered at x and $p(y_j|x)$ is the value of the probability density function (PDF) at point y_j when x is fixed. Since the input data points are non-uniformly distributed,each point y_j will have a different value for $p(y_j|x)$. Also, the PDF depends not only on the sample position y_j but also on x. Such convolution operation defined in equation Equation 3.9 is differentiable as shown below [HRV+]

$$\frac{\partial f \star g}{\partial \theta} = \frac{1}{|N(x)|} \sum_{j \in N(x)} \frac{f(y_j)}{p(y_j|x)} \frac{\partial g\left(\frac{x-y_j}{r}\right)}{\partial \theta} \tag{3.11}$$

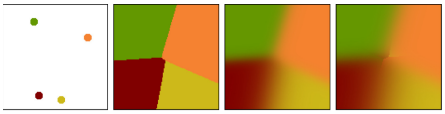
where θ is some model parameter. For estimating the probability distribution $p(y_j|x)$, the authors use kernel density estimation [Par62] [Ros56] as shown below

$$p(y_j|x) = \frac{1}{|N(x)|\,\sigma^3} \sum_{k \in N(x)} \left\{ \prod_{d=1}^{3} h\left(\frac{y_{j,d} - y_{k,d}}{\sigma}\right) \right\} \tag{3.12}$$

where σ determines the smoothing of the resulting sampling density function, h is the density estimation kernel i.e a non-negative function whose integral equals 1. In their experiments the authors use a Gaussian Kernel. In contrast to PointNet and PointNet++, the authors favor using Poisson Disk Sampling [Coo86] and combine several such sampling layers to achieve multi-resolution feature learning as shown in Figure 3.7.



(a) Tangent convolutions projects the neighborhood around an input point cloud onto a tangent plane defined by the surface normal at that point and does convolution on the projected points. The idea here is to simulate sliding a convolution kernel over the surface defined by the point clouds. More sophisticated methods use parallel transport to simulate kernel shifting but tangent convolution stands out for its simplicity. Figure from [TPKZ18].



(b) Effect of using different interpolation kernels for computing the tangent image. a) projected points b) nearest neighbors interpolation c) Gaussian interpolation d) Gaussian mixture interpolation with 3 nearest neighbors. Figure from [TPKZ18].

3.4 Tangent Convolution and surface based methods

Tangent convolution [TPKZ18] starts with defining a continuous tangent convolution at point $p \in P$ as follows [TPKZ18]

$$X(p) = \int_{\pi_p} c(u)\, S(u)\, du \tag{3.13}$$

where $c(u)$ is the convolution kernel, $S(u)$ [TPKZ18] is a tangent image π_p is a tangent plane and $u \in \mathbb{R}^2$ is a point on π_p. This is similar to introducing a orthogonal camera at point p which observes p along the surface normal n_p. The simplest procedure to estimate surface is done my computing the eigen-decomposition of the covariance matrix over the set of points $q : \|q - p\| \leq R$, $C = \sum_q rr^T$ [TPKZ18] where $r = q - p$ and R is some hyper-parameter. The eigen-vector corresponding to the smallest eigen-value is the surface normal n_p [TPKZ18] which also determines the orientation of the tangent plane. The other two eigen-vectors n_i, n_j determine the 2D image axis of the tangent image.

To estimate $S(u)$, the authors project the neighboring points q, onto the tangent image, which yields a set of projection points [TPKZ18]

$$v = \left(r^T n_i, r^T n_j\right) \tag{3.14}$$

$$S(v) = F_q \tag{3.15}$$

where F_q are the features associated with point q. This can be coordinates, colors, hand-crafted geometry features or learned features using an MLP. However, this only provides features for fixed set of points in the tangent image. To estimate the full tangent image requires computing the signal values at the intermediate points in the tangent image which the authors do using an interpolation kernel [TPKZ18].

$$S_u = \sum_v w(u, v) S(v) \tag{3.16}$$

where $u \in \mathbb{R}^2$ indicates coordinates of a point in the tangent image S and w is an interpolation kernel such that $\sum_v w = 1$. The authors consider two interpolation kernels - nearest neighbors interpolation [TPKZ18]

$$w(u, v) = \begin{cases} 1 \text{ if } v \text{ is } u\text{'s NN} \\ 0 \text{ otherwise} \end{cases} \tag{3.17}$$

and Gaussian interpolation [TPKZ18]

$$w(u, v) = \frac{1}{A} \exp\left(-\frac{\|u - v\|^2}{\sigma^2}\right) \tag{3.18}$$

where σ determines the smoothness of the Gaussian kernel. Plugging all these values into Equation Equation 3.13 gives the formula for tangent convolution [TPKZ18]

$$X(p) = \int_{\pi_p} c(u) S(u) \, du \tag{3.19}$$

$$= \int_{\pi_p} c(u) \cdot \sum_v \left(w(x, v) \cdot F(q)\right) du \tag{3.20}$$

The effects of different interpolation kernels can be seen in Figure 3.8b. For efficient computation, the authors flatten the tangent image into a $1D$ grid and use $1x1$ convolution to compute the feature vectors. Due to the fact that this method requires computing surface normals and it is sensitive to errors in the surface normal computation step.

Transport based feature learning method are similar to tangent convolution. They use parallel transport over a manifold to define shifting of kernels over a manifold and thus aim to achieve translation invariance which is a hallmark

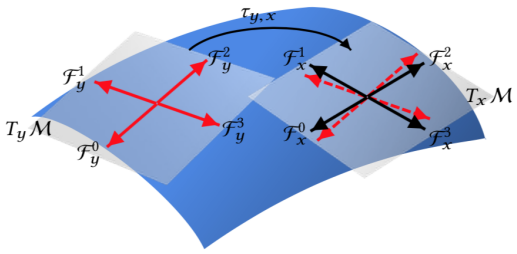


Figure 3.9: Translating a 4-direction frame using parallel transport from $\tau_{x,y}$ from y to x. A smooth frame field will minimize the orientation difference between each pair of frames and thus defines corresponding orientation directions $\mathcal{F}_i$ and $\mathcal{F}_j$. Figure from [PLLT18].

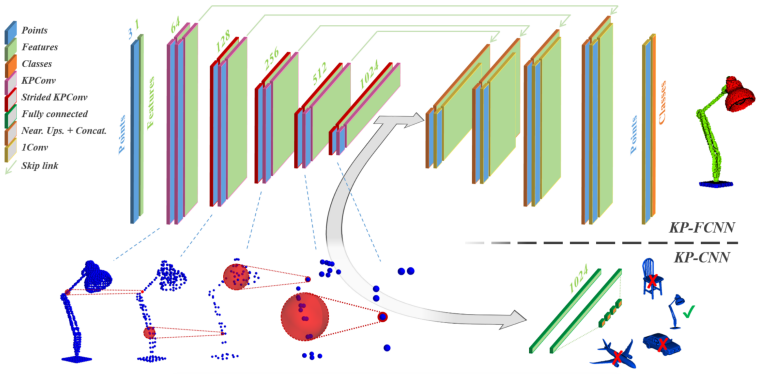


Figure 3.10: Illustration of the KPConv method. The authors define a novel kernel interpolation and achieve state-of-the-art results on S3DIS. Figure from [TQD+19].

property of 2D convolutions. Parallel frame convolutions [PLLT18] divide the directions between 0 to 360degrees into a N_{fields}-directional fields $\mathcal{F}_i$ with $i = 1, 2, .., N_{fields}$. The feature maps are also duplicated N_{fields} times with different directions taken as the x-axis but with shared convolution weights. Between two neighboring surface points, the orientation match $\mathcal{F}_i$ and $\mathcal{F}_j$ (Figure 3.9)is computed by solving an optimization problem. However, this method assumes the construction of a mesh first and does not directly operate on points.

3.5 Kernel Point Convolutions

Kernel Point Convolutions or KPConv is a point-based method where the authors define a novel kernel formulation that learns features directly over points without needing any intermediate representations of the input point-set. Formally the method can be framed as follows. Let $x_i \in P \subset \mathbb{R}^3$ for $i = 1, 2, 3, .., N_{points}$ be the input points of a point set P and the associated features be denoted by

$f_i \in \mathbb{R}^{N \times D}$. The general convolution by a kernel g of the feature set F can be denoted by [TQD+19]

$$F \star g(x) = \sum_{i \in \mathcal{N}(x)} g(x - x_i) f_i \tag{3.21}$$

In KPConv, the domain of the function g is limited to the ball $\mathcal{B}_r = \{y \in \mathbb{R}^3 | \|y\| \leq r\}$ [TQD+19] since the authors consider the radius neighborhood of point x $\mathcal{N}(x) = \{x_j | \|x_j - x\| \leq r\}$ [TQD+19]. Then K kernel points are selected $\{\widetilde{x_k} | k < K\} \subset \mathcal{B}_r$ [TQD+19] and a learnable kernel weight $W_k \in \mathbb{R}^{D_{in} \times D_{out}}$ where D_{in} is the input feature dimension and D_{out} is output feature dimension. The kernel function at a point $y_i \in \mathcal{B}_r$ is defined as [TQD+19]

$$g(y_i) = \sum_{k<K} h(y_i; \widetilde{x_k}) W_k \tag{3.22}$$

where h is the correlation between $\widetilde{x_k}$ and y_i [TQD+19]. The authors make use of a linear correlation

$$h(y_i; \widetilde{x_k}) = \max\left(0, 1 - \frac{\|y_i - \widetilde{x_k}\|}{\sigma}\right) \tag{3.23}$$

where σ is chosen according to input density. Compared to Gaussian correlation, the authors claim that linear correlation is better for gradient based optimization [TQD+19]. Limiting the domain of the function g to $\mathcal{B}_r$ helps the network learn meaningful feature representations [TQD+19]. The positions of the K kernel points is decided by solving an optimization problem where the authors aim to find a configuration of points after assigning to each point an attractive and a repulsive force as defined in Equation 3.25 and Equation 3.24 [TQD+19]

$$\forall x \in \mathbb{R}^3, E_k^{rep}(x) = \frac{1}{\|x - \widetilde{x_k}\|} \tag{3.24}$$

$$\forall x \in \mathbb{R}^3, E^{att}(x) = \|x\|^2 \tag{3.25}$$

The optimization problem then is given by Equation 3.26 [TQD+19]

$$E^{tot} = \sum_{k<K} \left(E^{att}(\widetilde{x_k}) + \sum_{l \neq k} E_k^{rep}(\widetilde{x_l}) \right) \tag{3.26}$$

The optimization is carried by gradient descent with random initialization and using an additional constraint that one of the points is the center of the sphere. For certain values of K, the points converge to stable regular polyhedron configuration (Figure 3.11. The authors also offer a generalization of this approach where the kernel point placement is not defined but is rather learned making the kernel "deformable". The kernel function g is differentiable with respect to the

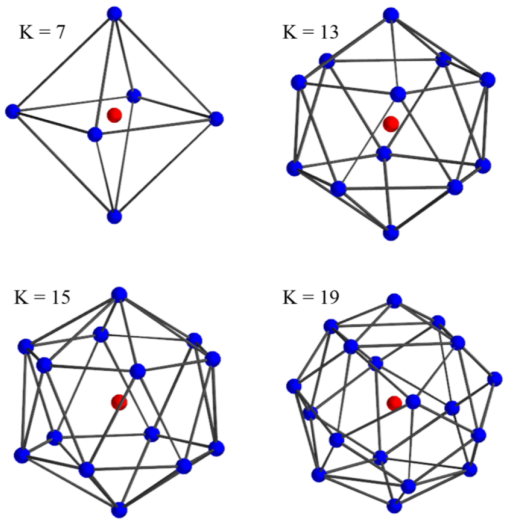


Figure 3.11: Various stable polyhedral configurations obtained by solving the optimization problem Equation 3.26. Figure from [TQD+19].

kernel points $\widetilde{x_k}$, hence the $\widetilde{x_k}$ can be treated a model parameter and the kernel points could then be learned. The authors define a generalization of KPConv for a deformable kernel case where the networks learns shifts Δx which are learned per input point in the point set. The deformable KPConv can be formally written as (from [TQD+19]) -

$$F \star g = \sum_{x_i \in \mathcal{N}(x)} g_{deform}\left(x - x_i, \Delta\left(x\right)\right) f_i \tag{3.27}$$

$$g_{deform}\left(y_i, \Delta\left(x\right)\right) = \sum_{k<K} h\left(y_i, \widetilde{x_k} + \Delta_k\left(x\right)\right) W_k \tag{3.28}$$

However, to make this deformable version work requires a regularization term to be added to the loss otherwise the kernel points end up begin pulled away from the input points. The regularized loss is as per Equation 3.29 [TQD+19].

$$\mathcal{L}_{reg} = \sum_{x} \mathcal{L}_{fit}\left(x\right) + \mathcal{L}_{rep}\left(x\right) \tag{3.29}$$

$$\mathcal{L}_{fit} = \sum_{k<K} \min_{y_i} \left(\frac{\|y_i - (\widetilde{x_k} + \Delta_k\left(x\right))\|^2}{\sigma} \right) \tag{3.30}$$

$$\mathcal{L}_{rep} = \sum_{k<K} \sum_{l \neq k} h\left(\widetilde{x_k} + \Delta_k\left(x\right), \widetilde{x_l} + \Delta_l\left(x\right)\right)^2 \tag{3.31}$$

My proposed method differs from this as I only define a k-NN graph and let the edge level filters learn the most optimum kernel weights without hand-coding the kernel function. In KPConv, the authors use grid-based pooling to learn multi-scale feature representations of the input point-set but the kernel function g is

Figure 3.12: Normal convolutions remove the sparsity of the data due to diffusion. Sparse convolutions aim to preserve this sparsity by doing convolutions only at active sites meaning points where the voxel has non-zero activation. Figure taken from [GvdM17]

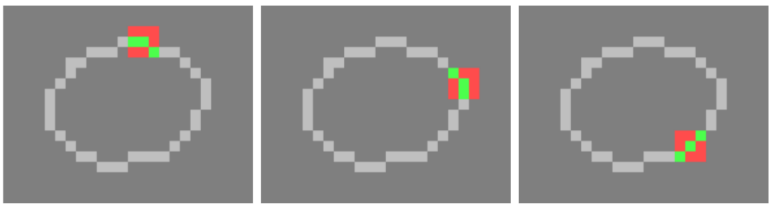


Figure 3.13: The locations shown in red are ignored in sparse convolutions. Thus for each convolution layer, the active sites in the output are the same as that in the input. Figure from [GEvdM18].

fixed. Although the deformable KPConv makes some leeway in the placement of kernel points, the basic form of the function g is fixed for each layer. Compared to this, my method learns kernel functions using edge-convolutions over a pyramid of voxelized representations of the input point-set as edge-level weights which are learned per-layer.

3.6 Sparse Convolutions

Sparse convolutions [GEvdM18] [GvdM17] are similar to OctNet with respect to their underlying approach that they take towards feature learning - making 3D convolutions efficient. However, sparse convolutions tend to take better advantage of the sparsity of 3D point clouds while simultaneously managing to preserve the sparsity of the input point-set. One of the problems that the authors of sparse convolutions try to tackle is that of “manifold dilation” which could be explained as follows. Define a d-dimensional convolutional network as a network that takes as input that is a $(d+1)$-dimensional tensor: the input tensor contains d spatio-temporal dimensions (such as length, width, height, time, etc.) and one additional feature space dimension for instance RGB color channels, surface normal vectors, etc [GEvdM18]. A sparse input corresponds to a d dimensional grid of sites that is associated with a feature vector. A site is defined to be ”active” if the feature corresponding to that site is not in ground state [GvdM17]. The ground state for a a feature could be, for instance, a value of zero. If the input

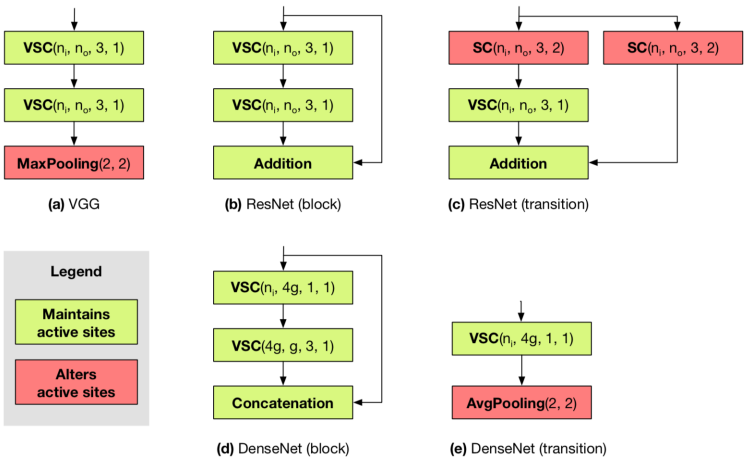


Figure 3.14: Implementation of various deep learning architectures using sparse convolutions. Figure from [GvdM17].

data contains a single active site, then after applying a 3^d convolution there will be 3^d active sites. After applying a second convolution of the same size will yield 5^d active sites and so on [GvdM17]. This rapid growth in the number of sites represents a problem since it loses the sparsity of 3D data and limits the depth of deep-learning architectures that can be implemented due to memory constraints.

The authors define sparse generalizations of three key deep learning components - convolutions, strided covolutions, pooling and unpooling. There are two modifications made to the convolution operation. The authors prune the number of active sites to include only those for which the central point has non-zero activation. The ground-state activation of all points is assumed to be zero. This means that the potential active sites in the output are exactly the ones that are active in the input. While the first modification makes implementation of the networks simpler, the second modification helps preserve the sparsity of the input.

MinkowskiNet [CGS19] operates along the same lines. The key difference between sparse convolutions and Minkowski engine is how inference is done. Sparse convolutions uses no neighborhood enforcing penalty to the loss function and uses vanilla cross-entropy loss but MinkowskiNet authors use a CRF-like layer on 7D inputs (obtained by doing mean-field approximation over the output probabilities).
The problem with sparse tensor based methods is that it requires re-defining and re-implementing most common operations in deep learning and thus are not the easiest approaches to extend. Furthermore, there is a voxelization step that is necessitated by voxel-hashing which could lead to potential loss of information.

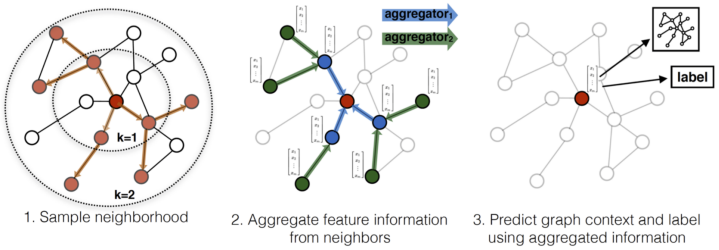


Figure 3.15: Illustration of the GraphSage method. I uses a sampled k-hop neighborhood to compute subgraphs, however this is not suitable for computer vision applications because of increasing subsample size. The authors use a hops size of 2. Figure from [HYL17].

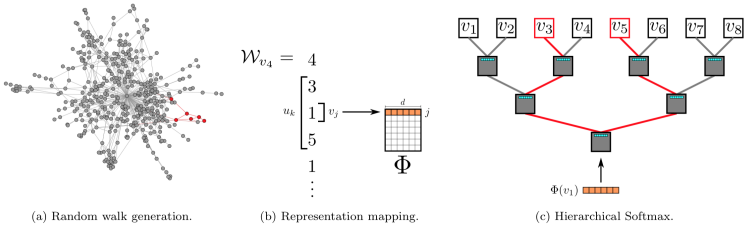


Figure 3.16: DeepWalk solves the problem of short receptive fields by sampling walks in a graphs using depth first search as shown here. The walks are then fed into a word2vec model like Glove which helps to learn node level features. However, these representations need to be recomputed once the underlying graph changes making it suitable only for static graphs. Figure from [PARS].

3.7 Graph Based Methods

Graph based methods rely on creating a graph - either KNN or neighborhood graph and then using graph convolutions on this graph to learn vertex level features. This class of methods requires extending the convolution operator on graphs and also on down-sampling. There are two ways to define convolutions on graphs - spectral and spatial.

Spectral methods rely on the convolution property

$$g \star f = \mathcal{F} * \mathcal{G} \tag{3.32}$$

Spectral convolutions rely on computing the eigen-decomposition of the graph Laplacian and then using the eigenvectors for computing convolution in frequency space and then back-projection. Naively using spectral convolutions is expensive since eigen-decompositions require quadratic computation. Methods based on special kind of polynomial kernels such as Chebyshev polynomials remove this

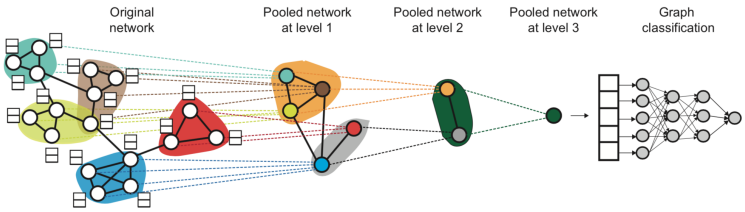


Figure 3.17: DiffPool [YYM+] tries to learn a subgraph by learning an assignment matrix. Due to the $\mathcal{O}(n^2)$ computations involved, this type of learned pooling is not suitable for 3D data. Figure from [YYM+].

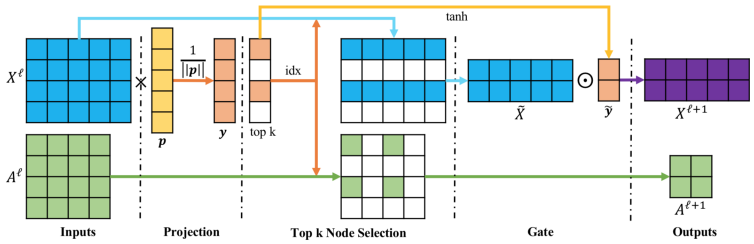


Figure 3.18: gPool was proposed as a part of a GraphUnet architecture. It differs from DiffPool in that it computes a vector of scores which avoids the quadratic assignment matrix computation. However, this method does not places guarantees on the local geometry of the subgraphs learned and in case the graphs become disconnected, further convolutions will be rendered ineffective. Figure from [GJ].

requirement. The problem with spectral convolutions is that kernels learned on graph cannot be applied to another since the graph Laplacian changes resulting in a change of the eigen-basis. Hence most methods use spatial graph convolutions, the general form [WSL+19] of which can be expressed as

$$h(x_i) = \square_{j:(i,j)\in E} g(x_i, x_j) \tag{3.33}$$

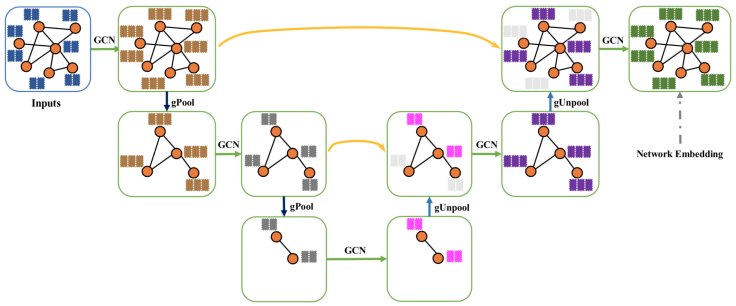


Figure 3.19: Graph UNet architecture which uses gPool. Figure from [GJ].

where $\square$ is some aggregation operator such as min, max or $mean$. This type convolution is computationally efficient to compute and furthermore kernels learned on one kind of graph can be generalized to another. The most commonly use convolution is edge convolution where $g = G([x_i, x_i - x_j])$ where G is represented by a neural network. The key however is to define sub-sampling methods. Various methods have been tried.

Super-point graphs computes a partition of the KNN graph using a minimization of the Potts energy function using the cut-pursuit algorithm. For each point $x_i \in P$ where P is a point-set, the authors compute d_g number of geometric features for characterizing the shape of its local neighborhood. These features are linearity, planarity and scattering and verticality feature. These features were introduced in [DMDV11] and [GL17]. For a point $x \in P$ and points $x_1, x_2, \ldots, x_k \in \mathcal{N}(x)$, the local covariance matrix is defined as [GL17]

$$\mu(x) = \frac{1}{k}\sum_{i=1}^{k} x_i \tag{3.34}$$

$$C = \sum_{i=1}^{k}(x_i - \mu)(x_i - \mu)^T \qquad = R\Lambda R^T \tag{3.35}$$

where Λ is a diagonal matrix of eigen-values and R is rotation matrix. The eigen-decomposition will exist since C is a symmetric positive definite matrix. If the three eigen-values are $\lambda_1 \geq \lambda_2 \geq \lambda_3$ without loss of generality and let $\sigma_i = \sqrt{\lambda_i}$ for $i = 1, 2, 3$ [GL17], then [DMDV11] describes the following three features for describing the local geometry of a point

$$\text{Linearity} = a_{1D} \qquad = \frac{\sigma_1 - \sigma_2}{\sigma_1} \tag{3.36}$$

$$\text{Planarity} = a_{2D} \qquad = \frac{\sigma_2 - \sigma_3}{\sigma_1} \tag{3.37}$$

$$\text{Scattering} = a_{3D} \qquad = \frac{\sigma_3}{\sigma_1} \tag{3.38}$$

The linearity describes how elongated a neighborhood is, planarity assesses how well it is fitted to a plane and a high scattering value denotes a spherical neighborhood. Since $a_{1D} + a_{2D} + a_{3D} = 1$, these feature values can also be interpreted as probabilites [DMDV11] of a point neighborhood being labelled as 1D, 2D or 3D. In [GL17] the authors define a fourth feature verticality which the authors claim proves crucial for discriminating roads and facades, and between poles and electric wires, as they share similar dimensionality. The verticality feature is the sum fo absolute values of the eigen-vectors weighted by the absolute values of $\sigma_1, \sigma_2, \sigma_3$

$$\hat{u_i} = \sum_{j=1}^{3} \left|\sigma_i u_i^j\right| Verticality = \hat{u_i}^{(3)} \tag{3.39}$$

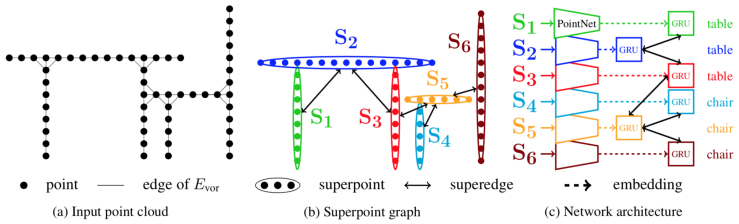


Figure 3.20: Overall schema of the Super point graph method. Figure from [LS].

where we without loss of generality it is assumed that the z-axis is the vertical axis. The verticality feature reaches a maximum value of 1 for a linear vertical neighborhood and a minimum value of 0 for a horizontal neighborhood. In [LS], the authors use these 4 features along with the normalized z-component of a point as the features associated with the points. The authors then compute a coarsened graph over a 10-nn graph $G_{nn} = (P, E_{nn})$ of the input point cloud by minimizing the Potts energy [LS]

$$\arg_{g \in \mathbb{R}^{d_g}} \sigma_{i \in P} \|g_i - f_i\|^2 + \mu \sum_{i,j \in E_n n} w_{i,j} \left[g_i - g_j \neq 0 \right] \tag{3.40}$$

where $[\cdot]$ denotes the Iversion bracket function Equation 3.41

$$[P] = \begin{cases} 1 \text{ if } P \text{ is true} \\ 0 \text{ otherwise} \end{cases} \tag{3.41}$$

This minimization problem is intractable due to the functional being non-convex and non-continuous. The authors utilize L_0 cut-pursuit algorithm [LO17] to compute an approximate solution. Advantage of using the cut-pursuit algorithm is that the number of partitions does not need to be set beforehand and furthermore the granularity of the partitions can be controlled. The regularization parameter μ in Equation 3.40 above controls the trade-off between coarsity and number of partitions since it introduces a boundary penalty. Smaller values of μ (0.01) lead to too-many smaller partitions and large values of μ (0.5-1) lead to coarser partitions. After computing this decomposition of the original graph into "super-points" i.e clusters of vertices given by the partition algorithm, the authors recompute a subgraph among these vertex partitions. The final segmentation is carried on the partition level using a GRU based message passing network Equation 3.42 [LS]. The complete pipeline is shown in Figure 3.20.

$$h_i^{t+1} = \left(1 - u_i^t\right) \odot q_i^t + u_i^t \odot h_i^t \tag{3.42}$$

$$q_i^{t+1} = \tanh\left(x_{1,i}^t + r_i^t \odot h_{1,i}^t\right) \tag{3.43}$$

$$u_i^t = \sigma\left(x_{2,i}^t + h_{2,i}^t\right), r_i^t = \sigma\left(x_{3,i}^t + h_{3,i}^t\right) \tag{3.44}$$

where $\odot$ denotes element-wise product, $\sigma(\cdot)$ [DMDV11] denotes the sigmoid activation function. Note that the receptive field is determined by both the memory of the GRU network as well as the number of time-steps for which message passing is carried out. This method however is sensitive the quality of partitions that are found out using the cut-pursuit algorithm.
In the graph learning literature various other methods of graph coarsening have been proposed. GraphSage [HYL17] uses a sampled k-hop neighborhood as show in Figure 3.15 for graph coarsening along with edge conditioned convolutions for feature-learning. The authors use a log-divergence between the learned representations as the loss function [HYL17]

$$J = -\log(\sigma f_u \cdot f_v) - Q \cdot E_{v_n \sim P_n v} \log(\sigma(-z_u \cdot z_v)) \tag{3.45}$$

where v is a vertex that appears in a sampled neighborhood of u. The method involves looking at the k-hop neighborhood of the input graph and sampling a certain number of points for pooling. The number of samples is a hyper-parameter and must be set. However, this method does not allow the network to have a very high receptive field since the number of points in the sample grows as $\mathcal{O}(s1 \cdot s2 \cdot s3 \cdot s4 \ldots \cdot s_k)$ where s_1 is the number of points sampled in the first hop, s_2 in the second hop and so on until s_k. In the original paper, the authors use a hop size of 2 which is not a sufficiently high receptive field for computer vision applications.

DeepWalk [PARS] solves this problem by sampling paths on an input graph using depth-first search, then uses a language model learning method such as Glove [PSM] for learning vertex level features. The objective here is model the probability of observing a vertex v after having visited other vertices on a path [PARS]

$$P(v|v_1, v_2, ..v_{n-1})$$

By sampling paths using depth-first search, it is possible to reach very high receptive fields. However, these representations are no longer valid once the underlying graph changes i.e. the method only works for graphs which are static. Furthermore, the method is not order-invariant meaning that if the label for a graph vertex changes, the learned representations are no longer valid. Due to these limitations, DeepWalk is only applicable to static graphs but not for graphs that are constructed at runtime. Methods such as DiffPool [YYM$^+$] try to learn the pooling instead of defining it. DiffPool attempts to learn an assignment matrix which is similar χ-conv discussed above. In the same manner as χ-conv DiffPool requires computing a cluster assignment matrix which quadratic time and space. Improvement over DiffPool is gPool [GJ] which learns a scoring function and then relies on top-k pooling to compute the graph vertices for the next subsample. The problem with DiffPool is that it does not place any guarantees on the properties of the coarsened graph. For example, it is possible that gPool computes a disconnected sub-graph. In such a scenario neighborhood information is lost and the

quality of the learned representations will be adversely affected.

Dynamic graph methods on the other do not rely on subgraph computation. These approaches typically use graph convolutions to learn vertex level representations and then apply some type of sub-sampling (eg. grid pooling) and then re-compute a graph from the sub-sampled representation. Examples include DGCNN [WSL+19] which computes KNN graphs in the feature space instead of the point space. Another method is [EKL] which samples alternate points in the adjacency matrix of the KNN graph. My approach goes in the direction of dynamic graphs. However, I rely on grid pooling for down-sampling but preserve the relationship between points and voxels which is used for later upsampling. Similar to DGCNN, I also re-compute graphs after each level of grid-pooling however, these graphs are computed only in the point-space unlike in DGCNN where re-computation of graphs is done in the feature space. In my experiments, we can see that this simplistic approach is better. Another major difference is the number of neighbors used for computing the k-NN graphs. DGCNN uses a graph computed using 20-40 nearest neighbors. Contrary to this, my method uses 5-10 nearest neighbors and is able to achieve competitive scores on S3DIS dataset.

4 Method

In this section, I first present my method for learning point-level features on 3D data and later on generalize this method to learn features on 4D data. Here a 3D data refers to a set of points $P \subset \mathbb{R}^3$, and 4D data refers an ordered sequence of such point-sets $P_{t_1}, P_{t_2}, P_{t_3}, \ldots$ such that $t_1 \leq t_2 \leq t_3 \leq \ldots$

4.1 3D semantic segmentation

Consider a point set $P \subset \mathbb{R}^{N\text{x}3}$ with points $p_i \in P$ for $i = 1, 2, 3..N$. I denote by $G = (V, E)$ the k-nearest neighbor graph over point-set P i.e. $(i, j) \in E$ if

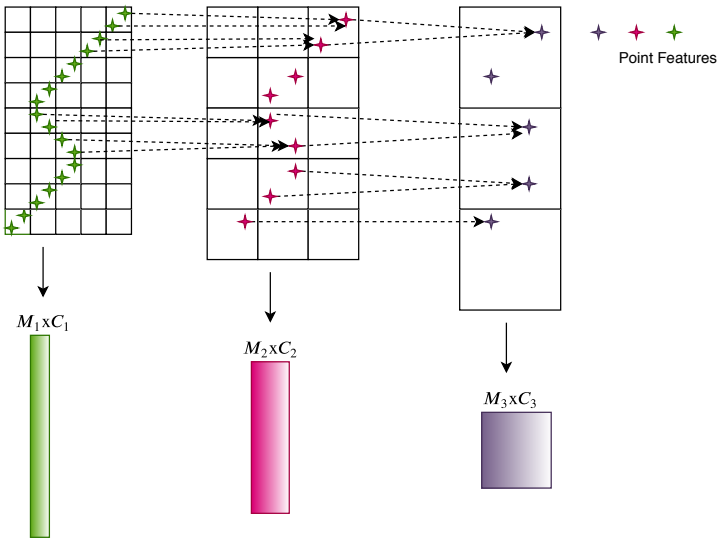


Figure 4.1: Illustration of grid-pooling for a 2D shape. Here $M1 \geq M2 \geq M3$ are the number of points in the corresponding point-sets and $C1 \leq C2 \leq C3$ denote number of feature channels. Arrows indicate which features are pooled to obtain which feature.

p_j is one of the k-nearest neighbors of p_i. A point-set P can be coarsened using consecutive voxelization steps with increasing voxel sizes. Let $P^{(l)} \subset \mathbb{R}^{N^{(l)}\text{x}3}l$ be the result of voxelizing P with voxels of size $r^{(l)}$ with $l \in \{1, 2, \ldots, l_{max}\}$ where l_{max} is the total number of voxelization steps [SK17]. The case $l = 0$ is handled differently. Let $P^{(0)}$ refer to the input point-set P, and $X^{(0)}$ refer to the features associated with $P^{(0)}$. For each $P^{(l)}$, let $G^{(l)} = \left(V^{(l)}, E^{(l)}\right)$ be corresponding KNN graph, $X^{(l)} \in \mathbb{R}^{N^{(l)}\text{x}d^{(l)}}$ denote associated point-level features and for each point $p_i^{(l)} \in P^{(l)}$, let $x_i^{(l)} \in X^{(l)}$ be the associated feature-vector. $P^{(l_1)}$ is said to be at a coarser resolution than $P^{(l_2)}$ if $l_1 > l_2$ and $P^{(l_2)}$ is said to be at a finer resolution than $P^{(l_1)}$. Let $X_D^{(l)}$ denote the point-level features associated with $P^{(l)}$ obtained by down-sampling $X^{(l-1)}$ and let $X_U^{(l)}$ denote point-level features obtained from up-sampling from $X^{(l+1)}$. The special case $X_D^{(0)}$ denotes the output of the first edge-convolution layer and $Z_D^{(0)}$ refers to the input point-level features.

4.1.1 Grid Pooling

Let $X_D^{(l)} \in \mathbb{R}^{N^{(l)}\text{x}d^{(l)}}$ denote the feature matrix associated with point-set $P^{(l)}$ obtained by going from finer resolution to coarser resolution i.e. down-sampling. Let $Z_D^{(l+1)} \subset \mathbb{R}^{N^{(l)}\text{x}d^{(l)}}$ denote the result of pooling $X_D^{(l)}$. To compute $Z_D^{(l+1)}$ from $X_D^{(l)}$, I use grid-pooling by which is formally defined as follows. Let $p_{i_1}^{(l)}, p_{i_1}^{(l)}, p_{i_1}^{(l)}, \ldots, p_{i_m}^{(l)}$ be points in $P^{(l)}$ and let $p_j^{(l+1)} \in P^{(l+1)}$ be obtained by Equation 4.1.

$$p_j^{(l+1)} = \frac{1}{m} \sum_{\alpha=1}^{m} p_{i_\alpha}^{(l)} \tag{4.1}$$

Then grid-pooled feature $z_{D,j}^{(l+1)}$ is given by Equation 4.2.

$$z_{D,j}^{(l+1)} = \square_{\alpha=1}^{m} x_{D,i_\alpha}^{(l)} \tag{4.2}$$

Here $\square$ denotes some aggregation function such as minimum, maximum or mean called the pooling function and the aggregated matrix of pooled features $Z_D^{(l+1)} = \left\{ z_{D,j}^{(l+1)} \right\}$. The corresponding point-level features $x_{D,j}^{(l+1)}$ is obtained by doing an edge-convolution using the KNN graph $G^{(l+1)}$ using $Z_D^{(l+1)}$ as vertex-level input features and is formally given by Equation 4.3.

$$x_{D,j}^{(l+1)} = f_{down}^{(l+1)} \left(x_{D,j}^{(l+1)}, \square_{(i,j)\in E^{(l+1)}} g_{down}^{(l+1)} \left(\left[z_{D,i}^{(l)}, z_{D,i}^{(l)} - z_{D,j}^{(l)} \right] \right) \right) \tag{4.3}$$

where $[\cdot]$ denotes concatenation, $g_{down}^{(l+1)}$ is modeled by a neural network and $f_{down}^{(l+1)}$ is some transformation which is explained later.

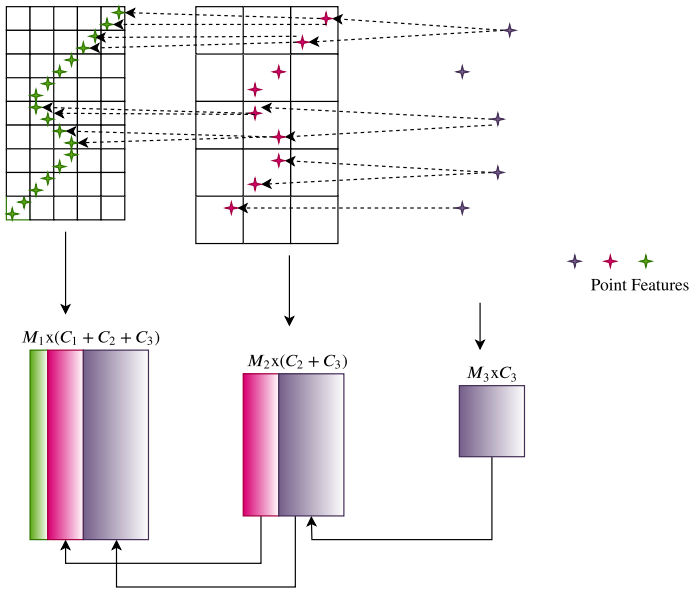


Figure 4.2: Illustation of grid-unpooling. The arrows indicate the direction of feature broad-casting. This information is stored as tuples in $T_{inv}^{(l)}$ for each level l.

4.1.2 Grid Un-pooling

Grid pooling can be inverted by storing the relationship between $z_{D,j}^{(l+1)}$ and $x_{D,i_\alpha}^{(l+1)}$ for $\alpha = 1, 2, \ldots, m$ in the form of index tuples $T_{inv}^{(l+1)} = \{(j, i_\alpha)\}$ for $\alpha = 1, 2, \ldots, m$. Un-pooling is done using feature broad-casting. Let $x_{U,i_\alpha}^{(l)}$ be the un-pooled feature corresponding to $p^{(l)} \in P^{(l)}$. Using the stored relationship T_{inv}, the un-pooled feature is given by Equation 4.4.

$$x_{U,i_\alpha}^{(l)} = x_j^{(l+1)} \tag{4.4}$$

Let $X_{U,i_\alpha}^{(l)} \in \mathbb{R}^{N^{(l)} \mathrm{x} d^{(l+1)}}$ be the matrix of un-pooled feature vectors. Then $X^{(l)}$ is given by concatenating the pooled and un-pooled feature matrices as shown in Equation 4.5 with individual point-features $x_{i_\alpha}^{(l)}$ given by Equation 4.6. Here again $[\cdot]$ denotes feature concatenation.

$$X^{(l)} = \left[X_D^{(l)}, X_U^{(l)}\right] \tag{4.5}$$

$$x_{i_\alpha}^{(l)} = \left[x_{D,i_\alpha}^{(l)}, x_{U,i_\alpha}^{(l)}\right] \tag{4.6}$$

4.1.3 Edge Convolution Layer

The form of the function $f_{down}^{(l+1)}$ still needs to be explained. I use two forms of this function as given by Equation 4.7 and Equation 4.8.

$$f_{down}^{(l+1)}(x, z) = z \tag{4.7}$$

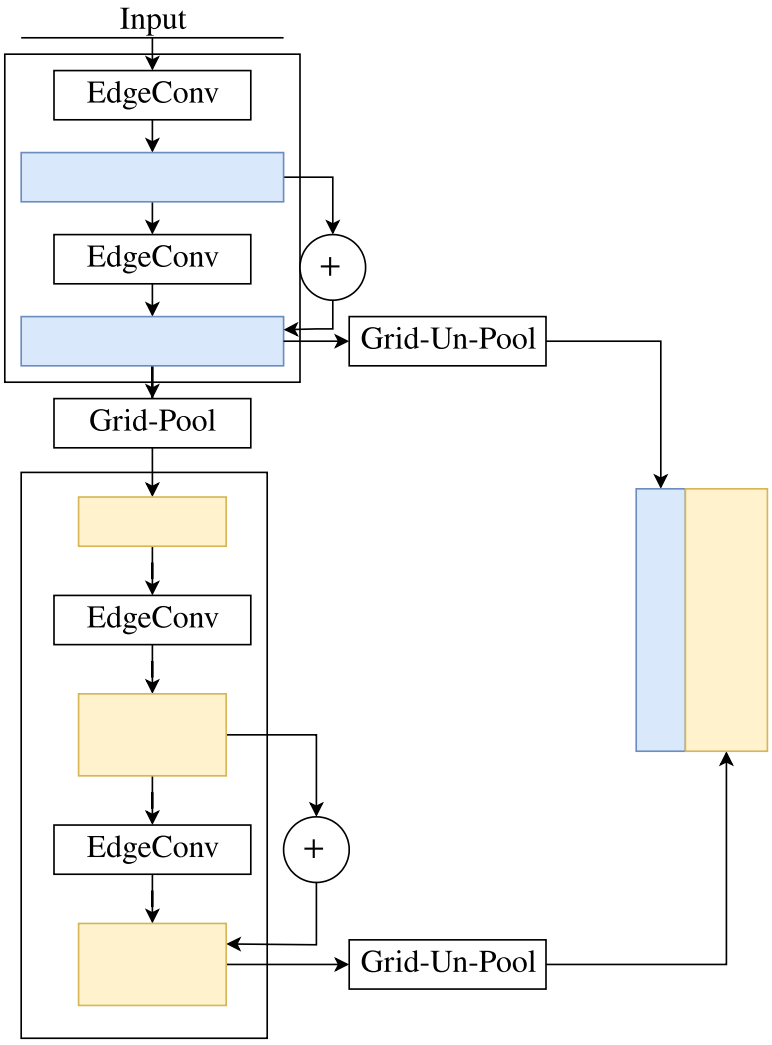


Figure 4.3: Illustration of a two-layer Res-U-Net type architecture with skip connections, grid-pooling and un-pooling.

$$f_{down}^{(l+1)}(x, z) = x + z \tag{4.8}$$

Using Equation 4.7, $x_{D,j}^{(l+1)}$ is given by Equation 4.9 and using Equation 4.8, $x_{D,j}^{(l+1)}$ is given by Equation 4.10. Here Equation 4.10 corresponds to doing convolutions with skip connections as in ResNet [HZRS15] type models. I refer to this type of model as Res-U-Net since this corresponds to an U-Net model with residual connections at each edge convolution layer.

$$x_{D,j}^{(l+1)} = \square_{(i,j)\in E^{(l+1)}} g_{down}^{(l+1)}\left(\left[z_{D,i}^{(l)}, z_{D,i}^{(l)} - z_{D,j}^{(l)}\right]\right) \tag{4.9}$$

$$x_{D,j}^{(l+1)} = x_{D,j}^{(l+1)} + \square_{(i,j)\in E^{(l+1)}} g_{down}^{(l+1)}\left(\left[z_{D,i}^{(l)}, z_{D,i}^{(l)} - z_{D,j}^{(l)}\right]\right) \tag{4.10}$$

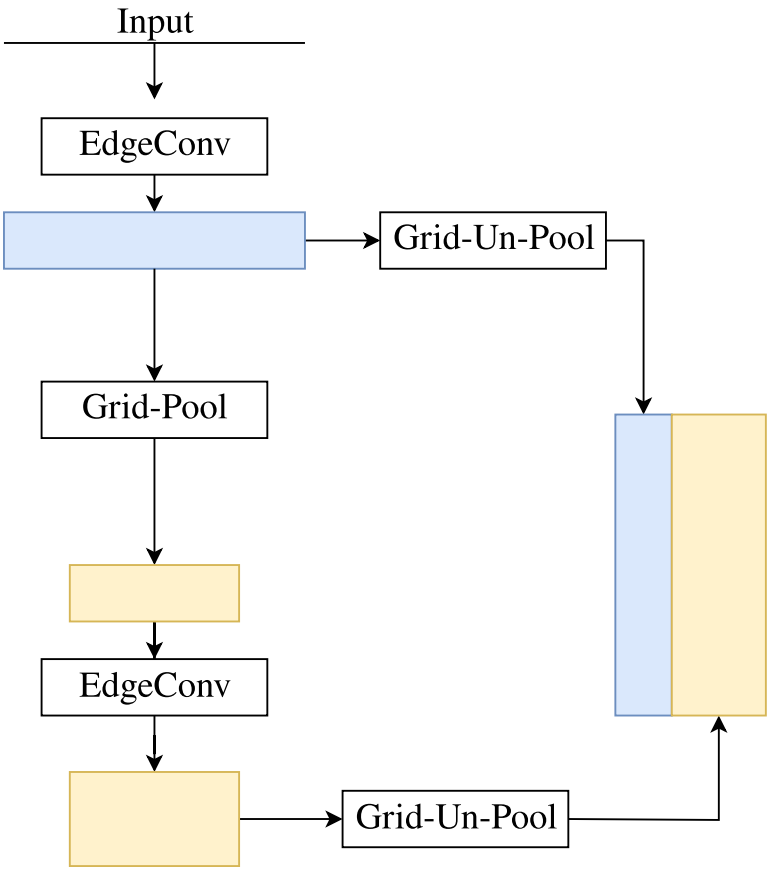


Figure 4.4: Illustration of a two-layer U-Net type architecture with grid-pooling and un-pooling.

Comparison to other methods

Compared to PointNet and PointNet++, my method diverges in how un-pooling is done. PointNet only has a single global layer of pooling followed by broadcasting all features to the input point cloud. The proposed method on the other hand has multiple layers of pooling followed by multiple layers of up-sampling using grid un-pooling. Compared to PointNet++, my up-sampling method is different since I forego the need for having an interpolation kernel for up-sampling Furthermore, both PointNet and PointNet++ utilize furthest point sampling for down-sampling which necessitates the need for doing a nearest neighbor based interpolation strategy whereas I use grid-pooling and just invert the pooling operation for up-sampling. In comparison to SplatNet and OctNet the proposed method learns features directly in the point space bypassing need for an intermediate representation. In comparison to KPConv, the learned kernels are not fixed because the method uses learned edge-convolution based filters. In comparison to sparse tensor based methods, for the proposed method features are learned per point not per voxel and thus there is no risk of losing details. However, sparse

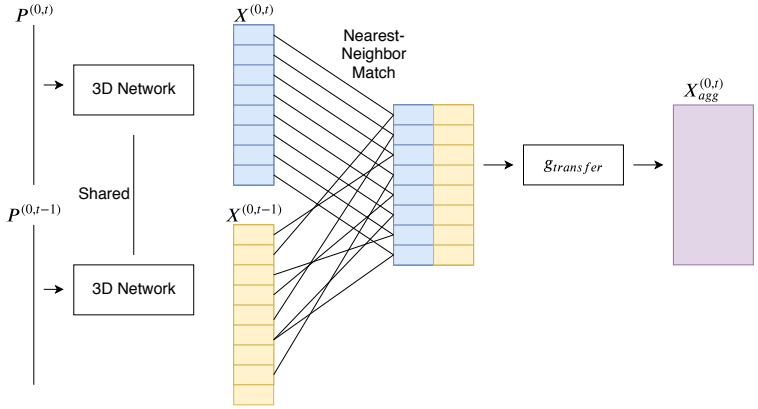


Figure 4.5: Ilustration of temporal integration method.

tensor based methods rely on convolution kernels defined on voxel in a high dimensional space and voxelization is inherently parallelizable making sparse tensor methods such as SparseConv and MinkowskiNet computationally faster.

4.2 4D semantic segmentation

4.2.1 Temporal integration layer

To generalize the 3D approach to 4D we start with a sequence of point sets $P^{(t_1)}, P^{(t_2)}, \ldots, P^{(t_k)} \subset \mathbb{R}^3$ such that $t_1 \leq t_2 \leq \ldots \leq t_k$. Similar to Section 4.1, for l in $\{1, 2, \ldots, l_{max}\}$ where l_{max} is the number of voxelization steps let $P^{(l,t)}$ represent the point-set $P^{(t)}$ voxelized with voxel size $r^{(l,t)}$, $G^{(l,t)}$ represent the KNN graph computed over $P^{(l,t)}$ and $X^{(l,t)}$ be point-level features associated with $P^{(l,t)}$. Let $P^{(0,t)}$ denote the input point-set (this corresponds to the case $l = 0$) with $X^{(0,t)}$ being the features associated with $P^{(0,t)}$.

Denote by $G^{(t)} = \left(V^{(t)}, E^{(t)}\right)$, the nearest neighbor graph that is computed between $P^{(0,t-1)}$ and $P^{(0,t)}$ i.e. $(j,i) \in E^{(t)}$ if $p_i^{(0,t-1)} \in P^{(0,t-1)}$ is the nearest neighbor of $p_j^{(0,t)} \in P^{(0,t)}$ among all points in $P^{(0,t-1)}$. Let $x_j^{(0,t)}$ be an associated feature of point $p_j^{(0,t)}$. Then the temporal feature $x_{temporal,j}^{(0,t)}$ associated with $p_j^{(0,t)}$ is given by Equation 4.11.

$$x_{temporal,j}^{(0,t)} = x_i^{(0,t-1)} \tag{4.11}$$

The aggregated point-level feature $x_{agg,j}^{(0,t)} \in X_{agg}^{(0,t)}$ is then given by Equation 4.12 where $g_{transfer}$ is modeled by neural network.

$$x_{agg,j}^{(0,t)} = g_{transfer}\left(\left[x_j^{(0,t)}, x_{temporal,j}^{(0,t)}\right]\right) \tag{4.12}$$

4.3 Memory complexity

In this section, I go over the memory complexity of my proposed method. Let $N = \left|P^{(0,t)}\right| = N^{(0,t)}$ and k be the number nearest neighbors. I store graphs using their edge representations hence a nearest neighbor graph with k nearest-neighbors requires $\mathcal{O}(Nk)$ storage. Assuming that each subsequent voxelization step down-samples the input point-set so that it retains only some fraction β_l of the input point-set $P^{(0)}$. The memory complexity of given by

$$\mathcal{O}\left(N^{(0)}k + \beta_1 N^{(0)}k + \ldots + \beta_{l_{max}} N^{(0)}k\right) = \mathcal{O}\left(kN^{(0)}\left(1 + \beta_1 + \beta_2 + \ldots + \beta_{l_{max}}\right)\right) \tag{4.13}$$

It is possible to put a closed-form upper-bound on the summation in the inner-brackets. Assume that voxel-sizes are doubled in every subsequent voxelization step which means that each step retains at most $\frac{7}{8}$ of the points. Thus,

$$\beta_{i+1} \leq \frac{7}{8}\beta_i \leq \ldots \leq \left(\frac{7}{8}\right)^i \beta_1 \tag{4.14}$$

Using Equation 4.14, we can bound the sum above as

$$1 + \beta_1 + \beta_2 + \beta_3 + \ldots + \beta_{l_{max}} \tag{4.15}$$

$$\leq 1 + \beta_1 + \frac{7}{8}\beta_1 + \left(\frac{7}{8}\right)^2 \beta_1 + \ldots + \left(\frac{7}{8}\right)^{l_{max}-1} \beta_1 \tag{4.16}$$

$$= 1 + \beta_1 \left(1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \ldots + \left(\frac{7}{8}\right)^{l_{max}-1}\right) \tag{4.17}$$

$$\leq 1 + \beta_1 \left(1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \ldots\right) \tag{4.18}$$

$$\leq 1 + 8\beta_1 \tag{4.19}$$

The final bounds the memory complexity to be $\mathcal{O}\left(N^{(0)}k\left(1 + 8\beta_1\right)\right)$ which is the same complexity as $\mathcal{O}\left(N^{(0)}k\right)$ since $\beta_1 < 1$. Note that we cannot have a tighter bound on β_1 since it is determined by $r^{(0)}$ i.e. voxel-size in the first voxelization step.

5 Experiments

In this chapter, I will discuss my experiments for 3D and 4D semantic segmentation, provide a qualitative and quantitative analysis of the results and compare with other existing methods in the respective areas.

5.1 3D semantic segmentation

For 3D semantic segmentation I use S3DIS dataset [ASZS]. This dataset consists of point-set representations of indoor environments. Corresponding to each point in a sample there is a label and there are a total of 13 semantic classes.

Data Preprocessing

All point-sets in the original S3DIS dataset are voxelized using a voxel size of 3cm. The label for each point in the voxelized point-set is computed using voting i.e for each point in the voxelized point-set, the assigned label is the most-frequent label in its corresponding voxel. Each voxelized point-set is again split into multiple smaller point-sets using Algorithm 2 using a threshold value N_{max} of 50000. The split point-sets are then used for training. Splitting using Algorithm 2 has the advantage that points are roughly uniformly distributed in each bin. A popular form of splitting is to split each point-set using cuboidal bounding volumes of certain dimensions [QSMG] [QYSG17]. Using such an approach has the disadvantage that it is possible to encounter pathological cases with low number of points inside the bounding volume. PointNet [QSMG] and PointNet++ [QYSG17] solve this problem by artificially increasing the number of points inside a bounding volume using sampling with replacement. Using Algorithm 2 avoids this problem in the case of indoor point clouds. Furthermore, Algorithm 2 is equivalent to using bounding volumes with infinite size along two axes and dynamically finding a size along one particular axis such that the variance in the number of points inside the bounding volumes is minimized.

Algorithm 2 Point Cloud Splitting Algorithm

Inputs: Point-set P, axis index $ax \in \{1, 2, 3\}$, threshold value N_{max}
Output: A sequence S consisting of k splits $P_1, P_2 \dots P_k$ such that $|P_i| \leq N_{max}\ \forall i = 1, 2, 3 \dots k$
$k \leftarrow 1$
$S \leftarrow \{P\}$ ▷ $\{\}$ indicates an empty list.
$C_{ax} \leftarrow \left\{x_j^{(ax)} : x_j \in P\right\}$ ▷ $x_j^{(ax)}$ is the coordinate of point x_j along the axis numbered ax.
$x_{min} = min\,(C_{ax})$, $x_{max} = max\,(C_{ax})$
while not $|P_i| \leq N_{max} \forall P_i \in S$ **do**
    $k \leftarrow k + 1$
    $S \leftarrow \{\}$
    $m \leftarrow 1$
    while $m \leq k$ **do**
        $y_l = x_{min} + (m-1) \cdot \frac{x_{max} - x_{min}}{k}$
        $y_u = x_{min} + m \cdot \frac{x_{max} - x_{min}}{k}$
        $P_m = \left\{x_j \in P : y_l \leq x_j^{(ax)} \leq y_u\right\}$
        $S.insert(P_m)$
    end while
end while

Model architecture

I use a modified variant of a U-Net type encoder-decoder architecture as shown in Figure 5.1. Unlike the vanilla U-Net architecture, I remove the decoder arm of the U-Net and use grid un-pooling as described in Section 4.1.2. This helps reduce the number of parameters and computation required. By connecting each edge convolution layer directly to the bottleneck layer it also helps alleviate the problem of vanishing gradients. I use 4 levels of voxelization with each level consisting of an edge convolution layer followed by a subsequent pooling layer. Pooling is done with voxel sizes of 5cm, 10cm, 20cm, 40cm. Finally, the features at each level of voxelization are grid un-pooled to their corresponding points at the first level. The number of feature channels is doubled every two levels and capped at 256.

Training

The model is trained using the Adam optimizer [KB15] with a fixed learning rate of 0.001, batch size of 1 and training is cut-off after 300 epochs. The loss function used is average cross-entropy loss over all points in a point-set as in Equation 5.1 where $|P|$ is the number of points in the point-set, $C = \left\{c_1, c_2, ..c_{|C|}\right\}$ is the indexed set of class labels with $|C| = 13$, $y_i^{(j)}$ is 1 if point x_i belongs to

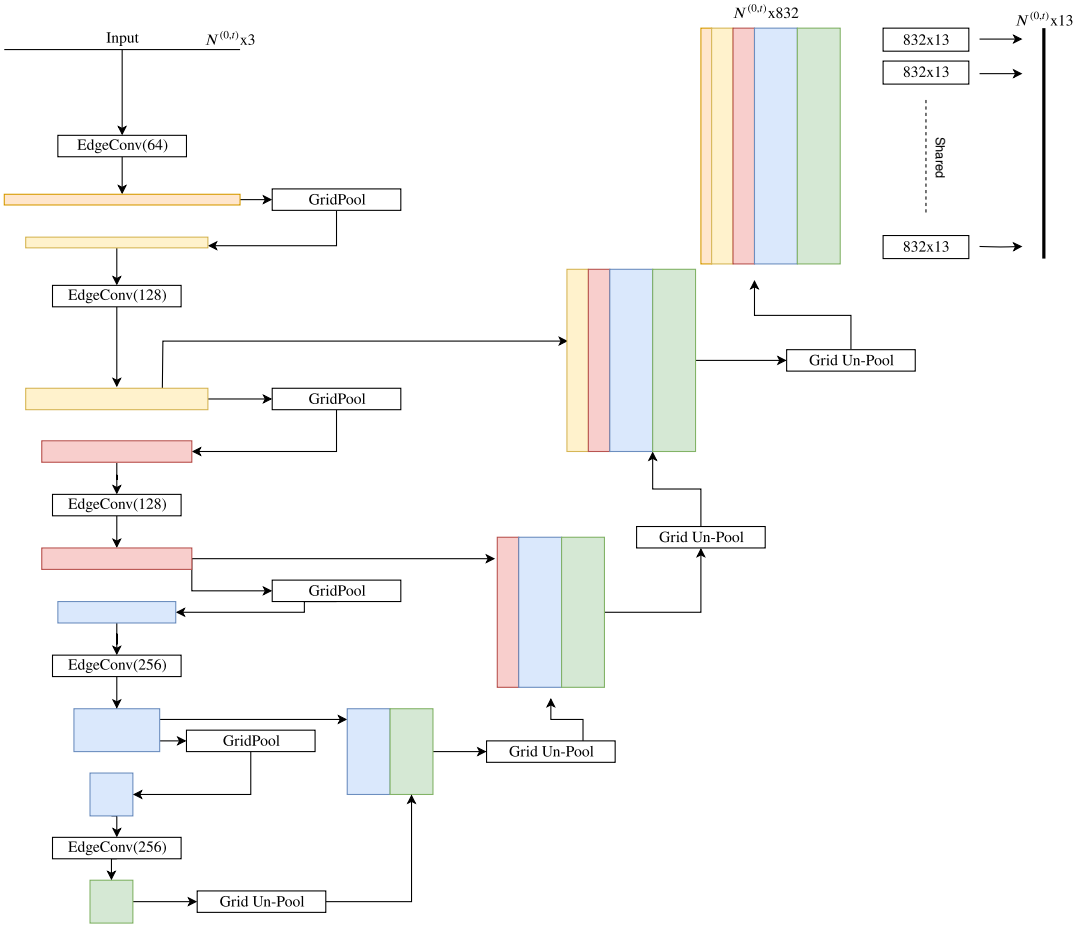


Figure 5.1: Schematic of model architecture used for training on S3DIS.

class $c_j \in C$ and 0 otherwise and $\hat{y}_i^{(j)}$ is probability assigned by the model of point x_i belonging to class c_j. Training is done on a single Nvidia GPU with 12GB of GPU memory. Random rotations around the gravity axis is used for augmentation. Before being fed to the model, each split point-set is translated to the origin. Another method of augmentation is adding a random displacement to each point in the input point-set [QSMG] [QYSG17]. I do not use this since translating the point-set to the origin will remove the effect of adding a random displacement. I report the evaluation scores for models trained using number of nearest neighbors k (number of neighbors used for computing nearest neighbor graphs) from 5 to 10 for hyper-parameter tuning.

$$L = -\frac{1}{|P|} \sum_{x_i \in P} \sum_{j \in C} y_i^{(j)} \lg \hat{y}_i^{(j)} \tag{5.1}$$

Testing

Each point-set in the validation set is voxelized using a voxel-size of 3cm and split into smaller point-sets using Algorithm 2. Inference is done on each of the smaller split point-set. The labels for un-voxelized input test point-set is computed using Algorithm 3 taking the computed labels on the split point-sets as input. I compute the evaluation scores for each model trained using different values of hyper-parameter k and then I select the best model on the basis of mIoU score. For this model, I then report 6-fold cross-validation score and individual scores obtained by testing on Areas 1 to 6 and provide a comparison with other methods.

Algorithm 3 Assembling different split point-sets to compute labels for full test point-set

Inputs : A test point-set P^{test}, a sequence of non-overlapping point-sets $P_1^{split}, P_2^{split}, \ldots, P_m^{split}$ with associated labels $L_1^{split}, L_2^{split}, .., L_m^{split}$, voxel-size h.
Output: A computed label set L^{test} for each of the points in the input point-set P^{test}.

$P^{concat} \leftarrow concatenate(P_1^{split}, P_2^{split}, \ldots, P_m^{split})$
$D^{ref} \leftarrow \Phi$ ▷ Φ is used to indicate an empty dictionary.
for $p_i^{concat} \in P^{concat}$ with $l_i^{concat} \in L^{concat}$ **do**
  $hx, hy, hz \leftarrow \lfloor p_i^{conat}.x/h \rfloor, \lfloor p_i^{conat}.y/h \rfloor, \lfloor p_i^{conat}.z/h \rfloor$
  $D^{ref}\,[hx, hy, hz] \leftarrow l_i^{concat}$ ▷ Put l_i^{concat} in the dictionary at key (hx, hy, hz)
end for
$L^{test} \leftarrow \{\}$ ▷ $\{\}$ indicates an empty list.
for all $p_i^{test} \in P^{test}$ **do**
  $hx^{test}, hy^{test}, hz^{test} \leftarrow \lfloor p_i^{test}.x/h \rfloor, \lfloor p_i^{test}.y/h \rfloor, \lfloor p_i^{test}.z/h \rfloor$
  $l_i^{test} \leftarrow D\,[hx^{test}, hy^{test}, hz^{test}]$
  $L^{test}.append(l_i^{test})$
end for

Results

The scores for different values of k are listed in Table 5.3. The model trained with $k = 10$ has the best mIoU on Area-5. A comparison of the best-model with other methods is presented in Table 5.1.

On S3DIS the best performing model uses a U-Net architecture and achieves a 60.12mIoU on Area-5 and 65.3mIoU with 6-fold cross validation. A comparison with other methods is presented in Table 5.1. A qualitative analysis is presented in section 5.1. A detailed results table with IoU scores over all areas for the best model is presented in Table 5.2.

We can see from Table 5.1, that the proposed method achieves a better performance on S3DIS than super-point graphs [DMDV11]. Results presented are

	OA	mAcc	mIoU	ceiling	floor	wall	beam	col.	win.	door	chair	table	book.	sofa	board	clutter
PointNet		49.88	41.09	88.8	97.33	69.8	0.05	3.92	46.26	10.76	52.61	58.93	40.28	5.85	26.38	33.22
SPG	86.38	66.5	58.04	89.35	96.87	78.12	0	42.81	48.93	61.58	84.66	75.41	69.84	52.6	2.1	52.22
KPConv	-	-	67.1	92.8	97.3	82.4	0	23.9	58	69	91	81.5	75.3	75.4	66.7	58.9
Tangent Convolution	82.5	62.2	52.8	90.5	97.7	74	0	20.7	39	31.3	69.4	77.5	38.5	57.3	48.8	39.8
Unet (mine)	**86.54**	**70**	60.12	92.68	**98.43**	77.86	**1.22**	16.25	40.81	62.46	79.95	**82.92**	49.1	64	60.61	55.3

Table 5.1: Results for best model on S3DIS on Area-5. I report overall accuracy (OA), mean accuracy (mAcc), mean intersection-over-union (mIoU). My method manages obtain a better mIoU than super-point graphs (SPG) [DMDV11] which relies on graph partitioning for coarsening input graphs. TanConv indicates the tangent convolution method [TPKZ18]

.

Area	ceiling	floor	wall	beam	column	window	door	table	chair	sofa	bookcase	board	clutter	mIoU
1	96.77	96.87	76.97	74.24	39.47	60.57	67.7	81.16	74.33	57.02	72.92	50.92	74.41	71.03
2	90.33	95.68	78.56	43.93	20.41	30.24	43.13	66.05	72.66	40.53	56.24	21.9	50.49	54.63
3	96.72	98.51	82.05	84.09	2.54	69.33	65.47	88.03	81.23	73.76	80.47	59.45	70.94	73.28
4	90.17	97.9	77.75	14.64	23.68	8.24	58.02	74.23	65.83	46.2	67.93	63.11	53.63	57.03
5	92.68	98.43	77.86	1.22	16.25	40.81	62.46	79.95	82.92	49.1	64	60.61	55.3	60.12
6	96.64	97.9	83.17	88.8	49.21	62.13	67.29	88.81	79.32	56.97	76.36	63.21	74.8	75.74

Table 5.2: Area-wise mIoU scores of the best-performing UNet model. My method is able to achieve a 6-fold mIoU score of 65.3

without using conditional random-fields (CRFs) and using only cross-entropy loss. With my method I am able to obtain competitive performance without using any neighborhood enforcing loss or post processing. The complete results for all areas is presented in Table 5.2. My model is able to obtain a 6-fold cross-validation mIoU of 65.3.

Qualitative Analysis of 3D segmentation network

Here I present a qualitative analysis for the predictions obtained by the 3D segmentation on S3DIS . These examples were selected by ranking the test samples using sample mIoU scores and selecting the top three and bottom three to analyze where the model performs well and identify possible areas of improvement. First I will go over the bottom-3 samples followed by the top-3. All visualizations were generated by a modified version of the PyViz3d package [Eng].

k	ceiling	floor	wall	beam	col.	win.	door	chair	table	book.	sofa	board	clutter	mIoU
5	85.69	98.39	76.04	0.24	**19.73**	38.98	53.39	78.56	73.41	46.22	64.31	41.79	42.18	55.3
6	86.98	98.34	76.81	0.39	14.62	40.89	55.91	76.54	77.15	40.27	60.53	44.71	44.99	55.24
7	87.94	98.35	76.5	0.07	15.07	43.54	53.86	77.74	78.31	43.38	60.08	44.12	46.78	55.83
8	89.8	98.02	76.95	0.42	17.03	**44.76**	52.9	79.1	77.89	40.97	**66.93**	46.66	52.14	57.2
9	89.53	98.22	76.02	0.16	18.68	42.91	57.56	79.01	76.66	34.5	63.01	46.56	50.3	56.39
10	**92.68**	**98.43**	**77.86**	**1.22**	16.25	40.81	**62.46**	**79.95**	**82.92**	**49.1**	64	**60.61**	**55.3**	**60.12**

Table 5.3: IoU of all categories for different values of k.

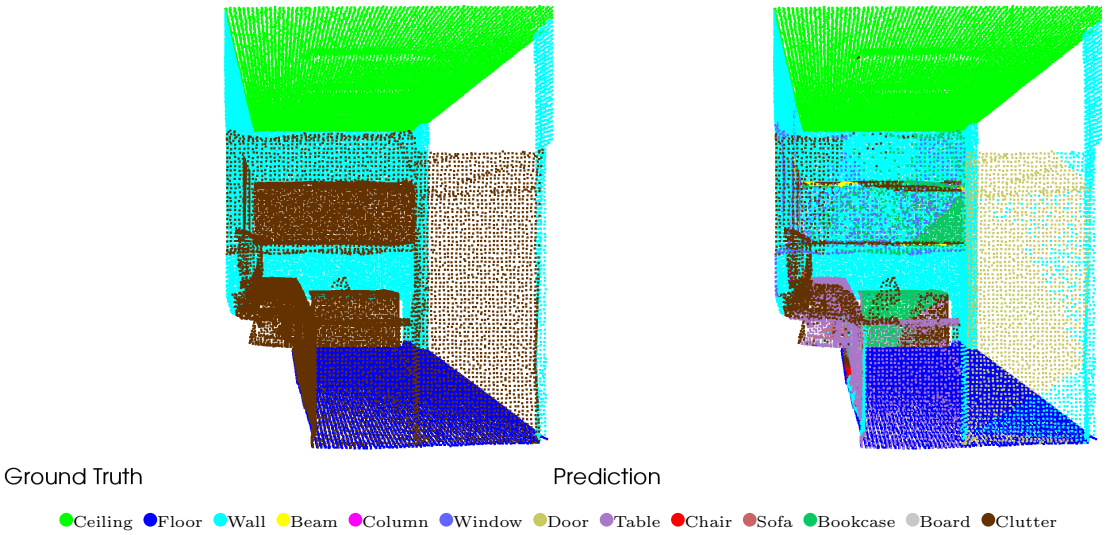


Figure 5.2: In this figure we see that the ceiling and floor has been segmented correctly. The large brown part is clutter and the model fails to identify it correctly. Possible reason for this is the inconsistent geometry of what constitutes clutter in the ground truth. However, one thing to note here is that the predictions are largely homogeneous across a large region even though the model uses no methods for enforcing neighborhood consistency.

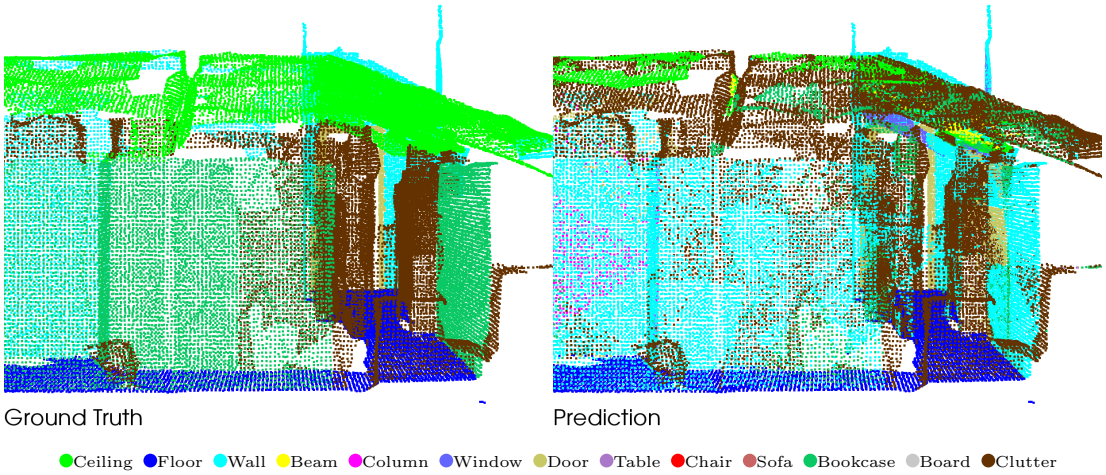


Figure 5.3: This is an example where the model outputs very bad predictions. We see that the scene is quite crowded with objects belonging to several semantic categories being present. This can be improved by using a slower voxel progression i.e. start with very small voxels in grid pooling but at the expense of more computational and memory costs.

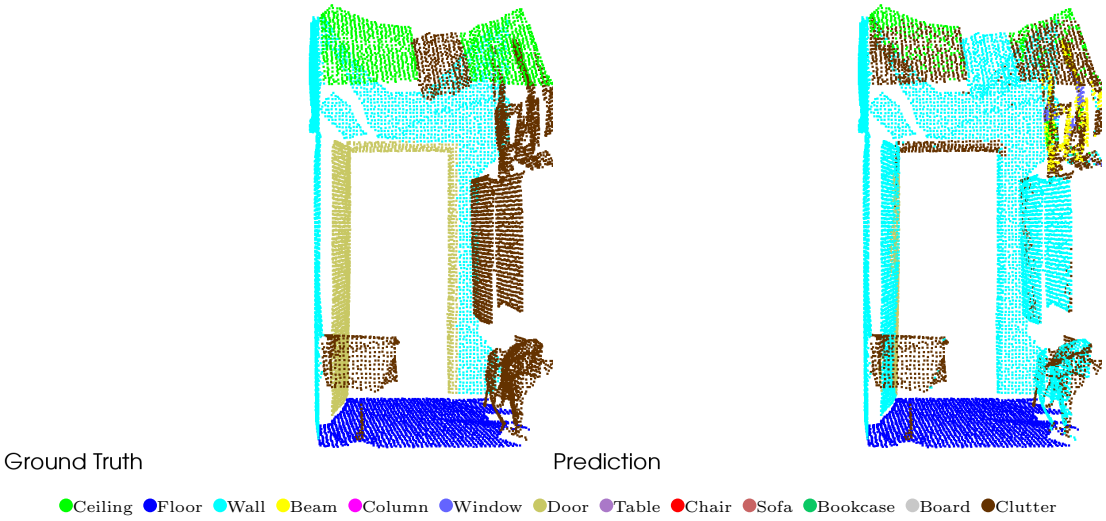


Figure 5.4: This example is similar to Figure 5.2 where in the background clutter is mis-identified. Another major source of error is the door that is mis-identified as wall.

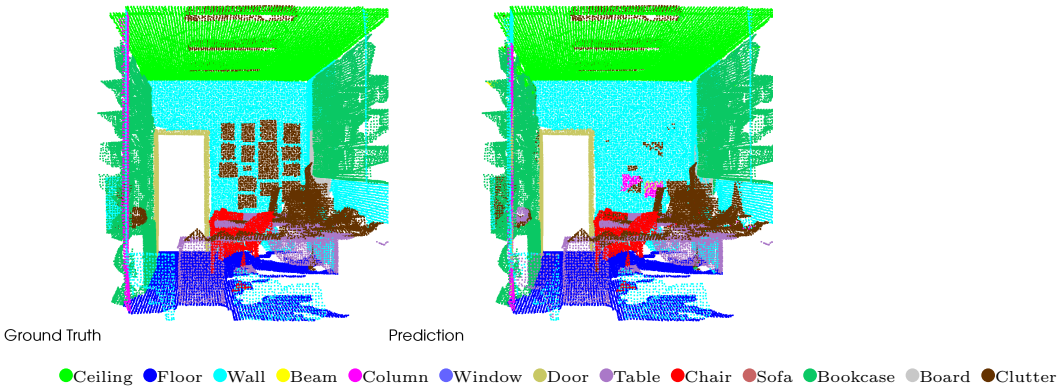


Figure 5.5: This is a relatively crowded scene with objects from multiple categories present. In this case the model is able to output good results. It is also able to identify clutter of irregular geometry correctly. What is missed are the rectangular shaped clutter objects in the background.

5.2 4D semantic segmentation

5.2.1 Dataset for 4D

The dataset used for 4D segmentation tests is a subsample generated from SemanticKITTI [BGM+19] dataset. SemanticKITTI has 19-classes which I reduce to 17 by re-mapping related classes into a common category. The re-mapping is done as per Table 5.4. The final set of labels has the following categories - car, bicycle, motorcycle, truck, other-vehicle (not truck and car), person, bicyclist,

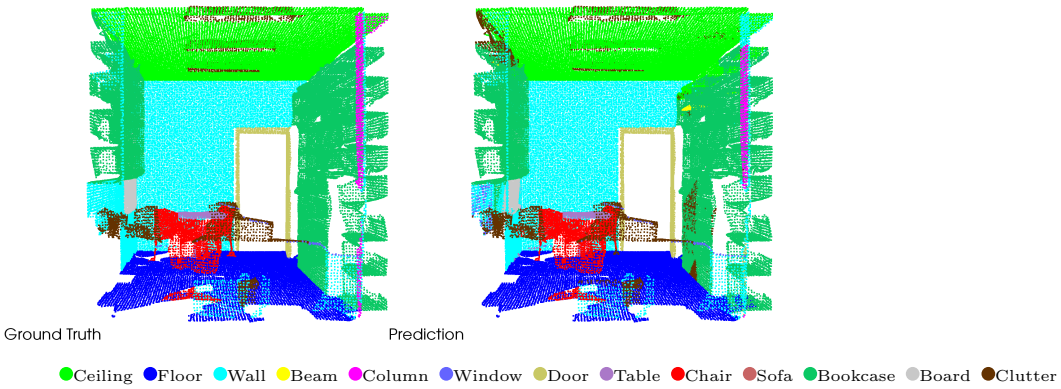


Figure 5.6: In this example, the model performs very well. It is also able to output correct semantic labels for objects which have thin geometry such as the door in the background.

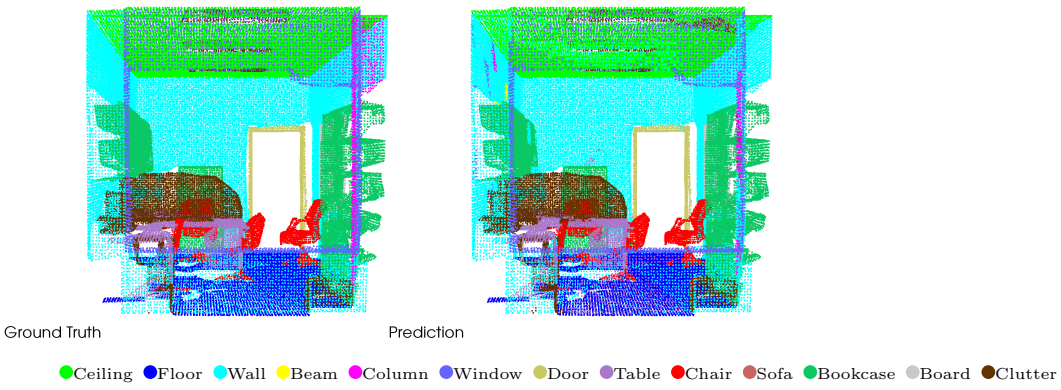


Figure 5.7: This scene also has objects belonging to multiple semantic categories. However, in this example, the method is able to assign correct labels for most points. Points belonging to objects with thin geometry such as the column are also correctly identified along with irregular geometry such as clutter.

Original semantic class	Re-mapped to
Bus	Other-vehicle
On-Rails	Other-vehicle
Lane marking	Road
Moving-car	Car
Moving-bicyclist	Bicyclist
Vegetation	Non-drivable area
Trunk	Non-drivable area
Terrain	Non-drivable area

Table 5.4: Related classes of objects from SemanticKITTI dataset are mapped to a common class.

road, sidewalk, building, fence, pole, traffic-sign, non-drivable area. Non-drivable area is introduced as a new class which refers to area where driving is not possible and the mapping from SemanticKITTI is defined in Table 5.4. The training data consists of the first 2000 laser scans from sequence 1 of SemanticKITTI dataset and test set consists of first 200 scans from sequence 7 of SemanticKITTI. The selection is done by computing label distribution in both sequences and using two sequences that have non-zero number of samples in each semantic class.

Pre-processing

Each point-set in the 4D dataset is split into two parts - front and back of the car. This helps to split the point-sets into two roughly equal splits consisting of around 60000 points. The splitting is done by converting each point in the input-point set from Cartesian to radial coordinates and then splitting using Algorithm 2 on the yaw axis. An advantage of this splitting method instead of using bounding volumes is that it avoids the problem of having unbalanced splits due to radial distribution of LiDAR point-sets. The point-sets are not voxelized and both training and inference is done at the full resolution.

5.2.2 Model and Training

Training is done on a single Nvidia GPU with 12GB memory using the Adam optimizer with fixed learning rate of 0.001 for 100 epochs. The baseline models consist of two variations of the proposed 3D model - one with residual U-Net connections and one without. Another model with temporal integration is then trained for each of the 3D baseline models. Pooling is done for 4 layers with voxel-sizes of 10cm, 20cm, 40cm and 80cm.. The number of feature channels is doubled every two layers from 32 in the first layer to 128. In the temporal integration layers consists of an multi-layer perceptron with output channel size of 288. The final classification is obtained using a bottle-neck layer that computes output logits using point-level features.

Results

We can see from Table 5.5 that for both 3D baselines, using temporal integration improves the model performance. The boost in performance can be seen especially in categories “Non-Drivable Area” and in the category “Building”.

Qualitative Analysis of 4D segmentation network

In this section I will present a qualitative analysis of the results obtained from temporal integration over the 3D baseline network. All visualizations were generated by a modified version of the PyViz3d package [Eng].

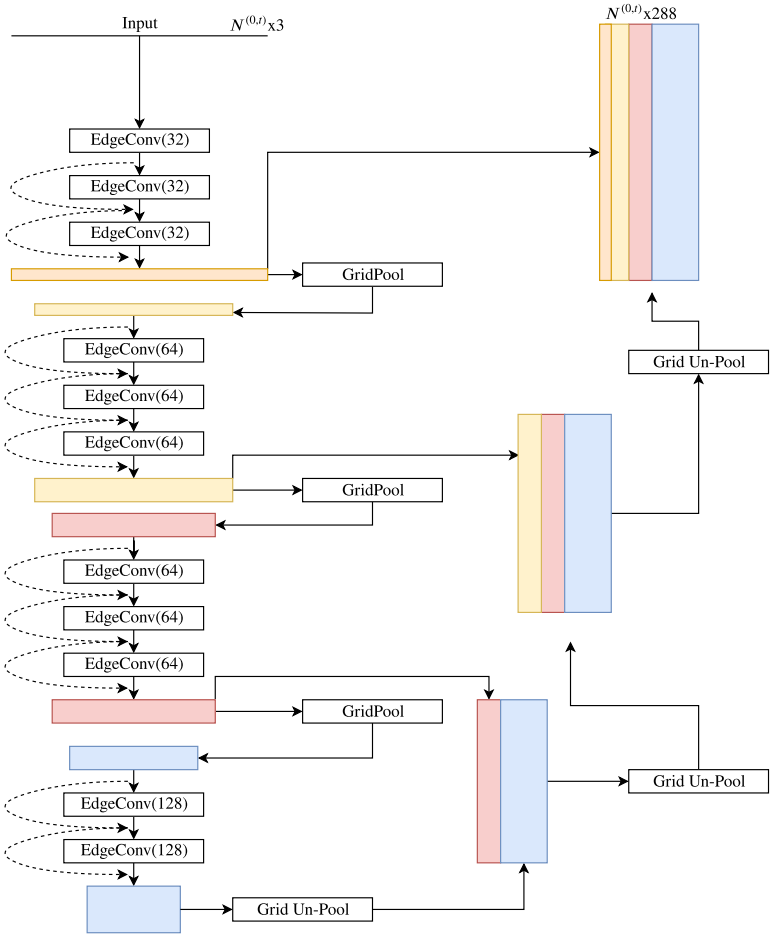


Figure 5.8: Illustration of Res-U-Net model used for 4D semantic segmentation. Dotted arrows indicate skip connections.

	Car	Bicycle	Moto.	Truck	OV	Person	Bicyc	M.cyc	Road	Parking	Sidewalk	OG	Building	Fence	Pole	Traf.	NDA	mIoU
U-Net 3D	42.92	0.03	0	0	0	0	0	0	62.48	0	54.69	0	20.61	13.5	0.06	0	34.59	13.46
Res-U-Net 3D	54.6	0	0	0	0	0	0	0	84.8	0	75.8	0	72.3	19.72	18.74	0	62.3	23.2
U-Net 4D	57.56	0	0.04	0	0	**0.01**	0	0	83.95	0	75.01	0	66.48	16.5	18.64	6.9	52.83	22.23
Res-U-Net 4D	**59.52**	**0.07**	**0.16**	0	0	0	0	0	**90.12**	0	**82.09**	0	**80.34**	**19.81**	**24.02**	**12.79**	**70.21**	**25.83**

Table 5.5: mIoU for each 3D baseline model with and without temporal integration. Suffix 3D means model without temporal integration and suffix 4D means a model with temporal integration. Abbreviation moto. refers to motorcycle, OV to other-vehicle, Bicyc. to bicycle, M.cyc. to motorcyclist, OG to other-ground, Traf. to traffic, NDA to non-drivable-area.

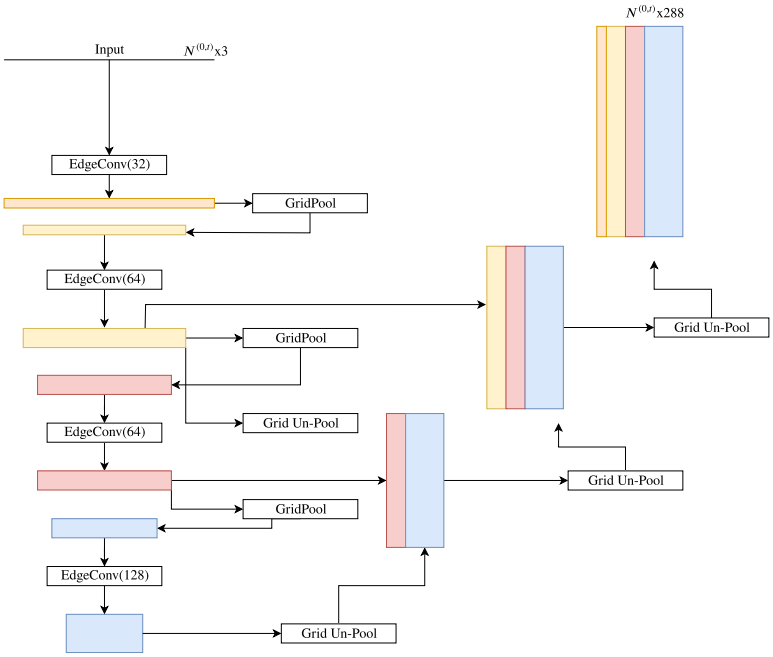


Figure 5.9: Illustration of U-Net based model used for 4D semantic segmentation.

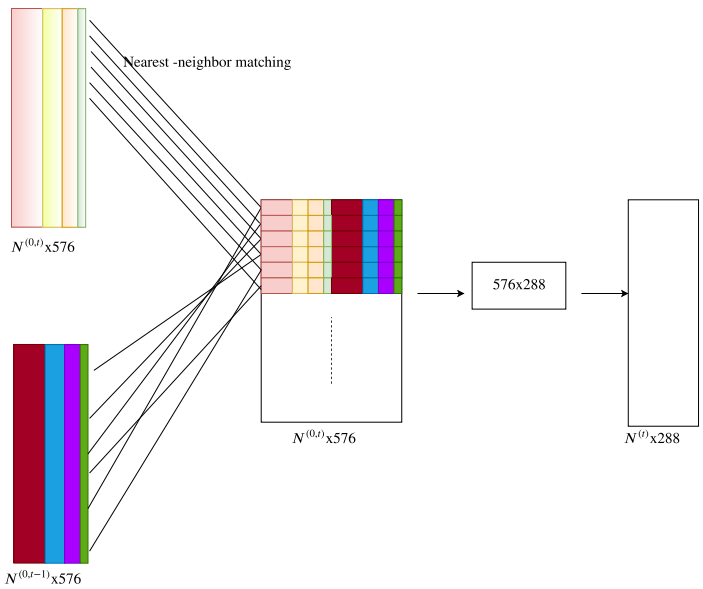


Figure 5.10: Schematic of final bottleneck layer that generates time-aggregated features.

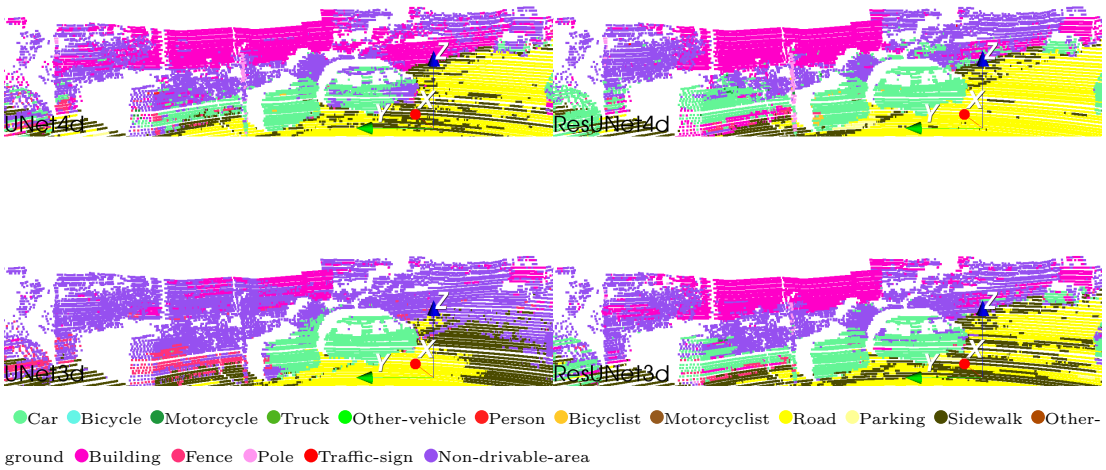


Figure 5.11: In this example, we can see that while the 3D models have trouble distinguishing between road and sidewalk, with temporal integration the errors are removed. The improvement can be starkly seen in the Res-U-Net -4D model where the errors in distinguishing between road and sidewalk are smoothed out. In the original scan, at this point the car is turning to the right, meaning that the area that is seen to the right of the car is unknown and features need to be learned using information from previous time steps. We can also see that the quality of the results obtained using temporal integration is dependent upon the strength of the backbone network where the quality of results obtained by applying temporal integration on Res-U-Net model is the best out of the four.

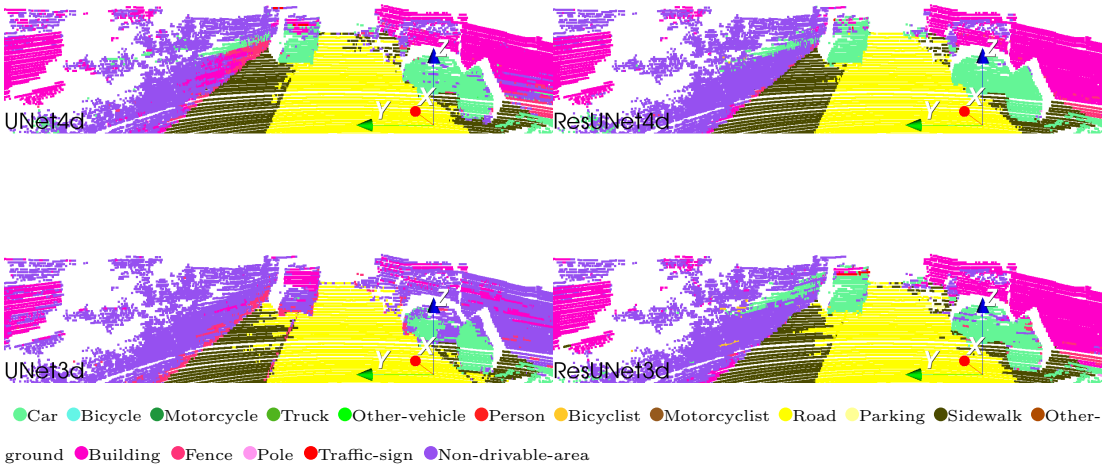


Figure 5.12: In this example, we can see that the 3D U-Net model makes a mistake in segmenting a car some distance away. This error is removed in the 4D U-Net model. The 3D Res-U-Net model does not make this mistake so the result obtained after temporal integration is the same as with the 3D model.

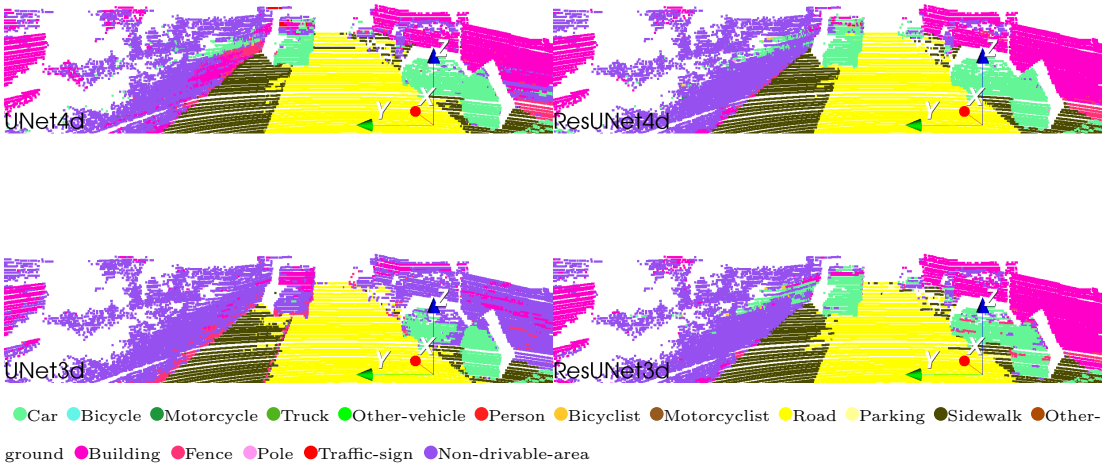


Figure 5.13: In this example we can see that the 4D U-Net model corrects a mistake made by the 3D model. The 3D model wrongly segments a building on the right as non-drivable area. This error is removed in the 4D model.

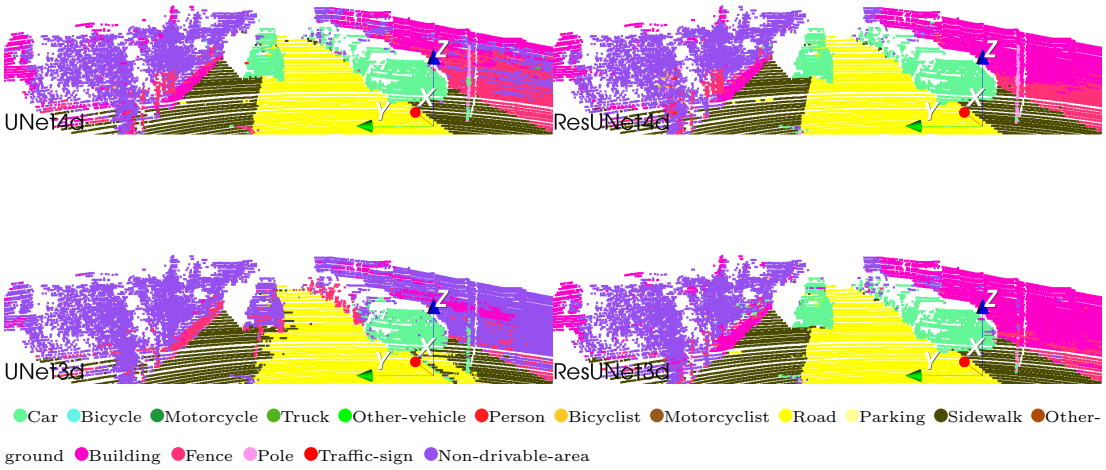


Figure 5.14: We see for both the U-Net model and the Res-U-Net model the corresponding 4D model corrects an error made in segmenting a car at some distance from the origin.

6 Conclusion

In this thesis, I used graph neural networks (GNNs) to build models that can tackle the problem of 3D and 4D segmentation. The key idea here was to use grid pooling to first increase the receptive field of graph neural networks. The current literature on graph neural networks tackles this in different ways but these methods are not suitable for direct applications in 3D. Because 3D data has an inherent "resolution" meaning that the same scene can be rendered at various levels of granularity, it is possible to use grid pooling to enhance the receptive field of graph neural networks. I propose a modification to grid-pooling that allows for feature transfer "upwards" meaning from lower resolution to higher resolution using a U-Net type architecture. The proposed network is able to achieve state-of-the-art result on Stanford 3D dataset beating all existing graph based methods. I then generalized this 3D network to 4D data using nearest neighbor matching across time-steps. Nearest neighbor matching in this case only works because the existing receptive field of the underlying 3D network is enhanced using grid pooling. This means that matching captures a large context from the previous time-step as well. With this generalization I demonstrate the temporal integration method improves performance of the underlying 3D networks.

There are several possible areas of improvement. Apart from the approach tested in the thesis, another possibility of doing temporal integration is to transfer information from previous time steps to the current time step such that features from previous time-steps are stored at a coarser resolution to preserve GPU memory. Another possible area is identify what are other good graph representations that could show improvements either on computation time or performance. Nearest neighbor graphs or radius graphs are not very fast to compute using the existing ball tree or k-d tree methods because of inherent sequential nature of tree construction.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ASZS] Iro Armeni, Alexander Sax, Amir R. Zamir, and Silvio Savarese. Joint 2d-3d-semantic data for indoor scene understanding.

[BGM+19] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *Proc. of the IEEE/CVF International Conf. on Computer Vision (ICCV)*, 2019.

[CGS19] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3075–3084, 2019.

[Coo86] Robert L. Cook. Robert l. cook. In *ACM Trans. Graph.*, 1986.

[DMDV11] Jerome Demantke, Clement Mallet, Nicolas David, and Bruno Vallet. Dimensionality based scale selection in 3d lidar point clouds. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences,*, 2011.

[EKL] Francis Engelmann, Theodora Kontogianni, and Bastian Leibe. Dilated point convolutions: On the receptive field of point convolutions. In *arxiv:1907.12046v1*.

[Eng] Francis Engelmann. Pyviz3d. https://github.com/francisengelmann/pyviz3d.

[Fey] Matthias Fey. Torchscatter. https://github.com/rusty1s/pytorch_scatter.

[FL19] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[GEvdM18] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3d semantic segmentation with submanifold sparse convolutional networks. 2018.

[GJ] Hongyang Gao and Shuiwang Ji. Graph u-net.

[GL17] S. Guinard and L. Landrieu. Weakly supervised segmentation-aided classification of urban scenes from 3d lidar point clouds. In *ISPRS*, 2017.

[GvdM17] Benjamin Graham and Laurens van der Maaten. Submanifold sparse convolutional networks. 2017.

[HLW17] G. Huang, Z. Liu, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[HRV+] P. Hermosilla, T. Ritschel, P-P Vazquez, A. Vinacua, and T. Ropinski. Monte carlo convolution for learning on non-uniformly sampled point clouds. volume 37.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[KB15] Diederik P. Kingma and Jimmy Lei Ba. Adam : A method for stochastic optimization. In *ICLR*, 2015.

[KJG15] Martin Kiefel, Varun Jampani, and Peter V. Gehler. Permutohedral lattice cnns. In *ICLR*, 2015.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, 2012.

[LBB+98] Y. LeCun, L. Bottou, Y. Bengio, , and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

[LO] Loic Landrieu and Guillaume Obozinski. Cut pursuit: fast algorithms to learn piecewise constant functions on general weighted graphs. In *SIAM*.

[LO17] Loic Landrieu and Guillaume Obozinski. Cut pursuit: Fast algorithms to learn piecewise constant functions on general weighted graphs. In *SIAM Journal on Imaging Sciences*, 2017.

[LS] Loic Landrieu and Martin Simonovsky. Large-scale point cloud semantic segmentation with superpoint graphs. In *CVPR2018*.

[NW] Jorge Nocedal and Stephen J Wright. In *Numerical Optimization*.

[Par62] Emanuel Parzen. On estimation of a probability density function and mode. In *Ann. Math. Statist. 33*, 1962.

[PARS] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *KDD 2014*.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. 2019.

[PLLT18] Hao Pan, Shilin Liu, Yang Liu, and Xin Tong. Convolutional neural networks on 3d surfaces using parallel frames. In *arXiv:1808.04952*, 2018.

[PSM] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Stanford Website*.

[QSMG] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Conference on Computer Vision and Pattern Recognition*.

[QYSG17] Charles R Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. 2017.

[RFB] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedicalimage segmentation. In *arXiv:1505.04597*.

[RHW86] David E. Rumelhart, Geoffery Hinton, and Ronald Williams. Learning representations by back-propagating errors. In *Nature Vol 323*, 1986.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. In *The Annals of Mathematical Statistics, Vol. 22,*, 1951.

[Ros56] Murray Rosenblatt. Remarks on some nonparametric estimates of a density function. In *Ann. Math. Statist. 27*, 1956.

[Ros58] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. In *Psycological Review*, 1958.

[RUG17] Gernot Riegler, Ali Osman Ulusoy, and Andreas Geiger. Octnet: Learning deep 3d representations at high resolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[SJS+18] Hang Su, Varun Jampani, Deqing Sun, Subhransu Maji, Evangelos Kalogerakis, Ming-Hsuan Yang, and Jan Kautz. SPLATNet: Sparse lattice networks for point cloud processing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2530–2539, 2018.

[SK17] Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *CVPR*, 2017.

[TPKZ18] Maxim Tatarchenko, Jaesik Park, Vladlen Koltun, and Qian-Yi Zhou. Tangent convolutions for dense prediction in 3D. 2018.

[TQD+19] Hugues Thomas, Charles R. Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, François Goulette, and Leonidas J. Guibas. Kpconv: Flexible and deformable convolution for point clouds. 2019.

[WPC+19] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. In *arxiv:1901.00596*, 2019.

[WS10] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2010.

[WSL+19] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph cnn for learning on point clouds. 2019.

[YYM+] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *arxiv:1806.08804v4*.