UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Graph convolutional neural networks for 3D data analysis

Albert Mosella-Montoro

Graph Convolutional Neural Networks for 3D Data Analysis

by
Albert Mosella-Montoro

Supervised by
Javier Ruiz-Hidalgo

At the
Image Processing Group
Signal Theory and Communications Department
Universitat Politècnica de Catalunya

This dissertation is submitted on 24 November 2022 for the degree of Doctor of Philosophy (PhD)

Abstract

Deep Learning allows the extraction of complex features directly from raw input data, eliminating the need for hand-crafted features from the classical Machine Learning pipeline. This new paradigm brought a boost in the performance across several domains, including computer vision, natural language processing and audio processing. However, there are still challenges when dealing with unorganized structures. This thesis addresses this challenge using Graph Convolutional Neural Networks, a new set of techniques capable of managing graph structures that can be used for processing 3D data.

The first part of the thesis focuses on the Graph Analysis task, in which we study the capabilities of Graph Convolutional Neural Networks to capture the intrinsic geometric information of 3D data. We propose the Attention Graph Convolution layer that learns to infer the kernel used during the convolution, taking into account the particularities of each neighbourhood of the graph. We explore two variants of the Attention Graph Convolution layer, one that explores a residual approach and another one that allows the convolution to combine different neighbourhood domains. Furthermore, we propose a set of 3D pooling layers that mimics the behaviour of the pooling layers found in common 2D Convolutional Neural Networks architectures. Finally, we present a 2D-3D Fusion block capable of merging the 3D geometric information that we get from a Graph Convolutional Neural Network with the texture information obtained by a 2D Convolutional Neural Network. We evaluate the presented contributions on the RGB-D Scene Classification task.

The second part of this thesis focuses on the Node Analysis task, which consists of extracting features on a node level, taking into account the neighbourhood structure. We present the Multi-Aggregator Graph Convolution layer that uses a multiple aggregator approach to better generalize for unseen topologies and learn better local representations. In addition, it reduces the memory footprint with respect to the Attention Graph Convolution layer. Finally, we analyze the capabilities of our proposed Graph Convolution layers to deal with heterogeneous graphs where the nodes of the graph may belong to different modalities. We evaluate the presented contributions with the Computer Graphics process of skinning a character mesh. Specifically, we propose a Two-Stream Graph Neural Network capable of predicting the skinning weights of a 3D character.

Acknowledgements

I would like to thank my supervisor, Javier Ruiz-Hidalgo, for his valuable guidance and support. I would also like to express my most sincere gratitude to Iain Matthews and Denis Tomé. They did welcome me to work with them for several months in what became a fruitful research stay where I had the opportunity to extend and create an application of my research. Finally, I would like to thank my family for the support provided throughout my entire life.

Contents

List of Figures

List of Tables

Acronyms

1D One Dimensional

2D Two Dimensional

3D Three Dimensional

AGC Attention Graph Convolution

CNN Convolutional Neural Network

DNN Deep Neural Network

EdgeConv Edge Convolution

FC Fully-Connected layer

FeaStConv Feature-Steered Graph Convolution

GAT Graph Attention Network

GCN Graph Convolutional Neural Network

GNN Graph Neural Network

GT Ground truth

ICA Independent Component Analysis

kNN k Nearest Neighbours

LBS Linear Blend Skinning

MAE Mean absolute error

MAGC Multi-Aggregator Graph Convolution

MLP Multilayer Perceptron

MSE Mean square error

MUNEGC Multi-Neighbourhood Graph Convolution

NVP Nearest Voxel Pooling

RAGC Residual Attention Graph Convolution

RGB Red, Green, Blue

RGB-D Red, Green, Blue, Depth

RNN Recurrent Neural Network

SVM Support Vector Machine

VP Voxel Pooling

Mathematical Notation

s	Scalar
$\vec{v}$	Vector
$\tilde{v}$	Vector of homogenous coordinates
M	Matrix
M^T	Transposed matrix M
$\mathbb{R}$	The set of real numbers
(a, b)	The open interval from a to b
$[a, b]$	The closed interval from a to b
$f(x)$	f applied to input(s) x
$\log x$	The natural logarithm of x
$\| \vec{v} \|$	The L_2 norm of $\vec{v}$
v_i	Element i of vector $\vec{v}$
$\vec{m}_i$	The i-th row vector of matrix M
M_{ij}	Element (i, j) of matrix M
$\vec{a} \| \vec{b}$	Concatenation of $\vec{a}$ and $\vec{b}$

$\nabla_\theta y|_{\theta=\theta'}$ Gradient of y with respect to θ, evaluated at $\theta = \theta'$

$x \sim \mathbb{P}(x)$ Random variable x sampled from $\mathbb{P}$

$\mathrm{Var}(f(x))$ The variance of $f(x)$

$\mathcal{U}(a, b)$ The uniform real distribution on the range $[a, b]$

$\mathcal{N}(\mu, \sigma)$ The normal distribution with mean μ and standard deviation σ

$\deg(i)$ The degree of the node i of a graph

CHAPTER 1

INTRODUCTION

Computer Vision is an interdisciplinary scientific field that explores techniques to mimic the human visual perception system [74]. This system is in charge of capturing and processing the visual information of our world. Humans are capable of identifying objects and learning relationships between them. To mimic this behaviour, Computer Vision borrows techniques from Artificial Intelligence and Image Processing to try to identify relevant information from vision signals.

Early algorithms in Computer Science addressed complex problems by breaking them down into simpler ones, such that the steps to solve those problems were defined as a sequence of encoded instructions that a computer could execute. Artificial Intelligence generalized this idea and tried to represent the knowledge about the world as a base of axioms, along with simple rules for deriving new knowledge [79].

However, the community realised that the symbolic approach to Artificial Intelligence becomes intractable since, in many problems of interest, it was extremely difficult to manually specify the axioms and rules that a computer should use to reach a decision. This phenomenon is known as the symbolic grounding problem [38], which states that relying on handcrafted features or mathematical elements can hardly be reconciled with the richness of a real-world signal.

Machine Learning is a discipline within Artificial Intelligence that, motivated by how human beings learn from past experiences, proposes a set of techniques that can learn from data. However, Machine Learning still relies on handcrafted features to learn. In the 21^{st} century, Deep Learning, a sub-field of the Machine Learning discipline, has attracted much research interest. This set of techniques addresses the problem from a representation learning perspective [6]. Deep Learning studies general-purpose systems, usually known as Deep Neural Networks (DNNs), that can automatically learn hierarchical representations from data.

DNNs are composed of a stack of layers, with the first layer processing raw data and each following layer receiving the output of the previous one. DNNs learn a more complex representation of the input data on

each layer, creating a hierarchical representation of the input signal and eliminating the need for handcrafted features. Given a large training set of input and target examples, the network iteratively optimizes its parameters using a first-order method [91] such as gradient descent. The network gradually learns better data representations, where the shallower layers specialise in detecting simple features and the deeper layers more complex ones.

In 2012 *Krizhevsky* et al. [55] proposed, for the first time in Computer Vision, the use of DNNs to solve the ImageNet [18] classification benchmark. The proposed DNN named AlexNet reduced the top-5 error from 25% to 15.3%. AlexNet guided the change in paradigm in Computer Vision to move away from handcrafted representations towards learned ones. The use of DNNs yielded remarkable improvements in different Computer Vision applications such as image classification [41, 98], object detection [29, 30, 89], image segmentation [39, 70, 126], image captioning [113, 118] or text-conditioned image generation [7, 123], among others. While there are a lot of DNN architectures for dealing with two-dimensional images, it is still not clear how to deal with three-dimensional (3D) captures. Main DNN architectures assume organized structures, and most of the sensors that capture 3D information are unorganized. This thesis addresses the aforementioned challenge using Graph Convolutional Neural Networks (GCNs), a new set of techniques that allows DNNs to work with graph structures that can be used for processing 3D data structures.

1.1 Objectives and Contributions

The main goal of this thesis is to study and improve current Graph Convolutional Neural Networks to analyze 3D data on different kinds of tasks. We focus our research on studying the capabilities of Graph Convolutional Neural Networks to capture the geometric characteristics of 3D data, as well as the best way to use other multi-modal information. We divide the research into two major parts: Graph and Node Analysis. Graph Analysis consists of finding characteristics that define the whole graph structure, whereas node analysis consists of predicting characteristics on a node level, taking into account the whole graph structure. Now, we are going to present the contributions of this thesis for each of the two parts.

Graph Analysis Contributions:

The Graph Analysis part is divided into two blocks: In the first block, we first analyze the capabilities of GCN to capture the geometric characteristics of 3D data. To this end, we propose the Attention Graph Convolution layer, which learns to infer the kernel used during the convolution, taking into account the neighbourhood configuration, allowing it to better capture the neighbourhood characteristics. Inspired by the well-known residual blocks of 2D Convolutional Neural Networks (CNNs), we extend the Attention Graph Convolution layer into a residual approach.

In the second block, we first analyze the best way to introduce other modal information. To this end, we propose a 2D-3D Fusion Network capable of fusing multi-modal features from sensors with different resolutions. Furthermore, we extend the previous Attention Graph Convolution into a multi-neighbourhood approach which allows learning richer representations by exploiting the information from multiple neighbourhoods. Finally, to be able to mimic the traditional CNNs architectures, we present the Nearest Voxel Pooling layer.

The results of this first part have been published in the following articles:

- **Albert Mosella-Montoro** and Javier Ruiz-Hidalgo. Residual Attention Graph Convolutional Network for Geometric 3D Scene Classification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshop*, 2019

- **Albert Mosella-Montoro** and Javier Ruiz-Hidalgo. 2D–3D Geometric Fusion network using Multi-Neighbourhood Graph Convolution for RGB-D Indoor Scene Classification. *Information Fusion*, 76:46–54, 2021

Node Analysis Contributions:

In the Node Analysis part, we present the Multi-Aggregator Graph Convolution that uses a multiple aggregator approach to better generalize for unseen topologies and learn better local representations. Furthermore, we study the performance of previously proposed methods to infer characteristics at a node level of a heterogenous graph. To this end, we propose a variant of the Multi-Aggregator Graph Convolution that better handles heterogeneous graphs.

The results of this second part have been published in the following article:

- **Albert Mosella-Montoro** and Javier Ruiz-Hidalgo. SkinningNet: Two-Stream Graph Convolutional Neural Network for Skinning Prediction of Synthetic Characters. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022

1.2 Outline

This dissertation is organized into six chapters:

- **Chapter 1: Introduction.** Introduces the motivation, the main objectives and an overview of the contributions of this dissertation.

- **Chapter 2: Deep Neural Networks and 3D Data Structures**. Presents the basic knowledge of the techniques used in this dissertation.

- **Chapter 3: Residual Attention Graph Convolutional Network.** Chapter devoted to the study of the capabilities of a GCN to capture the geometric characteristics of 3D data and presents the Attention Graph Convolution and its residual extension. Furthermore, it proposes the Voxel Pooling layer to mimic the traditional CNNs architectures.

- **Chapter 4: 2D-3D Fusion Network using Multi-Neighbourhood Graph Convolutional Network.** Presents a two-branch approach to fuse multi-modal data. Furthermore, it introduces the extension of Graph Convolutional Neural Networks to work in a multi-neighbourhood fashion. Finally,

it proposes the Nearest Voxel Pooling layer, an improved version of the Voxel Pooling layer that better handles outliers.

- **Chapter 5: Heterogeneous Graph Convolutional Neural Network using A Multi-Aggregator approach.** This chapter studies the performance of previously proposed methods to infer characteristics at a node level of a heterogenous graph. Moreover, it presents the Multi-Aggregator Graph Convolution.

- **Chapter 6**: **Conclusions.** Summarizes and discusses the contributions of this dissertation. It also gives a high-level description of future research lines.

Chapters 3, 4, 5 are divided into two research parts, aforementioned and depicted in Table 1.1.

Part I **Graph Analysis**	**Part II** **Node Analysis**
Chapter 3	Chapter 5
Chapter 4	-

Table 1.1: Thesis structure.

CHAPTER 2

Deep Neural Networks and 3D Data Structures

The contributions of this dissertation are centred around *Deep Neural Network* architectures applied to 3D data structures. In order to provide the contextual information for interpreting their significance, we will survey the most important background methods and introduce the 3D data structures relevant to this thesis. For the sake of clarity throughout the document, as well as establishing a common notational framework, we have defined a consistent notation inspired by the one used in [32].

2.1 Essentials

2.1.1 Perceptron

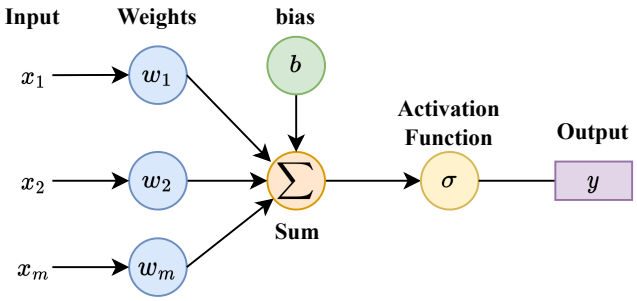


Figure 2.1: The Perceptron model with input vector $\vec{x}$, weights $\vec{w}$, bias b and activation function σ.

The *perceptron* or *neuron* [90] (illustrated by Figure 2.1) is the basic unit of a neural network. This unit receives several inputs, $\vec{x} \in \mathbb{R}^n$, and computes a linear combination of them, producing a single output. The linear combination is guided by a set of weights, $\vec{w} \in \mathbb{R}^n$, and a bias value, b. A non-linear activation function, σ, is potentially applied to the result to obtain the final output value, $y \in \mathbb{R}$, as described in Equation 2.1.

$$y = \sigma\left(b + \sum_{i=1}^{n} w_i x_i\right) = \sigma\left(b + \vec{w}^T \vec{x}\right) \tag{2.1}$$

The most common activation functions and the ones of interest for this thesis are the following:

- The *logistic Sigmoid* function defined in Equation 2.2:

$$\sigma(x_i) = \frac{1}{1 + e^{-x_i}} \tag{2.2}$$

  which maps real numbers to the $(0, 1)$ range as depicted in Figure 2.2. This function is commonly used for binary classification problems due to its capability of modelling Bernoulli random variables.

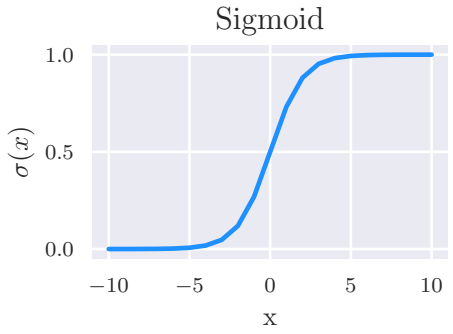


Figure 2.2: Sigmoid activation function.

- The *Hyperbolic Tangent (tanh)* function defined in Equation 2.3:

$$\sigma(x_i) = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \tag{2.3}$$

  which maps real numbers to the $(-1, 1)$ range and has a sigmoidal shape, depicted in Figure 2.3.

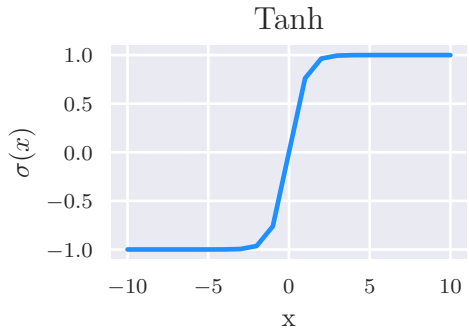


Figure 2.3: Hyperbolic tangent (tanh) activation function.

- The *Softmax* function defined in Equation 2.4:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.4}$$

which is a generalization of the logistic Sigmoid function to a multiple-dimension scenario that maps a vector of real numbers to a probability distribution. This function is commonly used for multi-class classification problems.

- The *Rectified Linear Unit (ReLU)* function:

$$\sigma(x_i) = max(x_i, 0) \tag{2.5}$$

which is one of the most used activation functions due to its biological plausibility, lack of vanishing gradients and computational efficiency. The ReLU function suppresses the negative values of a function as depicted in Figure 2.4.

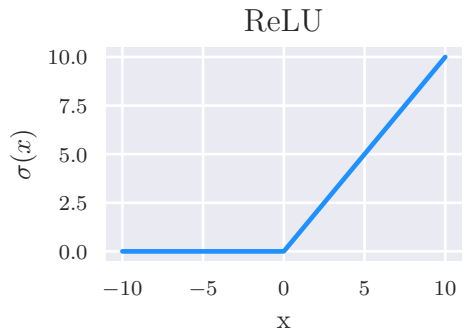


Figure 2.4: Rectified Linear Unit (ReLU) activation function.

- The *Leaky Rectified Linear Unit (Leaky ReLU)* function:

$$\sigma(x_i) = \begin{cases} x_i & x_i \geq 0 \\ \alpha x_i & x_i < 0 \end{cases} \tag{2.6}$$

which attenuates the negative values (depicted in Figure 2.5) using the slope, α, instead of completely suppressing the values as ReLU does.

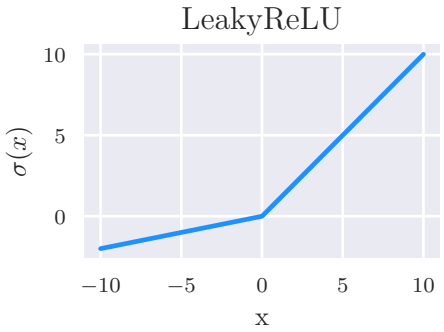


Figure 2.5: Leaky Rectified Linear Unit (Leaky ReLU) activation function.

2.1.2 Multilayer perceptron

Feedforward neural networks organize their neurons in a stack of layers, with each layer receiving as input the output of the previous layer only. The simplest feedforward example is the multilayer perceptron (MLP). Each layer is composed of several perceptrons, which are fully connected with the next layer's perceptrons, as depicted in Figure 2.6. Layers that fulfill this property are also called *fully-connected layers* (FC). Equation 2.7 defines the computation of the j-th perceptron of this layer, defining the input to an MLP layer as $\vec{x}$, the weights and biases of the perceptron j as $\vec{w}_j$ and b_j respectively. We can also represent the computation of an MLP layer in *matrix* form as defined in Equation 2.8, where W and $\vec{b}$ are the *weight matrix* and *bias vector* of an MLP layer. This representation is advantageous when leveraging *Deep Neural Networks*. Increasing the number of layers in a neural network is also referred to as increasing its *depth*, which is the reason we refer to these models as *Deep Neural Networks*.

$$y_j = \sigma(b_j + \vec{w}_j^T \vec{x}) \tag{2.7}$$

$$\vec{y} = \sigma(\vec{b} + W\vec{x}) \tag{2.8}$$

The matrix form of Equation 2.8 makes it easy to represent MLP networks by stacking multiple layers with their own parameters. Equation 2.9 shows an example of a two-layer MLP.

$$\vec{y} = \sigma_2(W_2\sigma_1(W_1\vec{x} + \vec{b}_1) + \vec{b}_2) \tag{2.9}$$

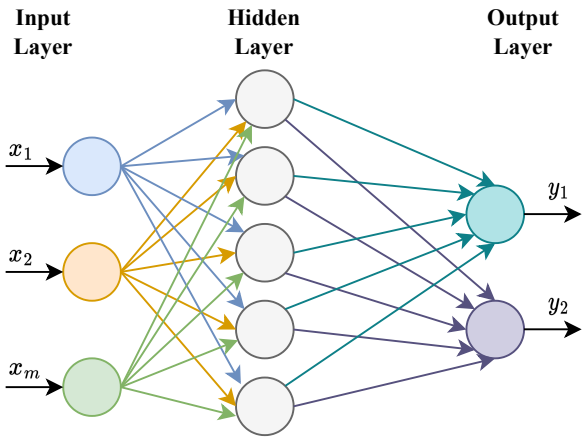


Figure 2.6: Schema of a multilayer perceptron with a single hidden layer.

2.1.3 Training Neural Networks

The learning procedure of neural networks is in charge of finding the best set of weights, W, and biases, $\vec{b}$, that approximate the target problem. In this thesis, the tasks we target belong to the *Supervised Learning paradigm*, which needs a *training* set, $\vec{s} = (\vec{x}_i, \vec{y}_i)_{i=1}^{n}$, of n input-output pairs. Feeding the input, $\vec{x}_i$, into the neural network, we obtain the prediction of the network for this input, $\vec{\hat{y}}_i$, as the output of its final layer.

The training procedure can be divided into three main steps. First, we have to choose the proper loss function to evaluate the network's performance. In Section 2.1.3.1, we review in detail the main characteristics of this loss function. Once we have defined the function that evaluates the network's performance, we have to choose the proper initial parameters. The straightforward solution would be using a random initialization of them. However, using clever strategies discussed in Section 2.1.3.2 allows the optimization algorithm to find the best set of weights, W, and biases, $\vec{b}$, faster. Finally, we apply the chosen optimization algorithm to train the network, which is explained in Section 2.1.3.3.

Deep Neural Networks are models composed of a vast number of parameters. They are normally optimized on training sets of smaller sizes than the number of parameters the network has, causing the model to overfit the training set and fail to generalize outside of it. In Section 2.1.3.4, we discuss the techniques to fight the overfitting used in this thesis.

2.1.3.1 Loss functions

In a supervised machine learning context, the loss function, also known as the error function, measures the quality of a particular set of weights, W, and biases, $\vec{b}$, based on how well the predictions match with the ground truth (GT) of the training data. The loss function guides the training process during the optimization to find the best set of parameters for the defined task. These functions must satisfy the following properties:

- They must be differentiable.
- They should be as convex as possible. When the prediction is equal to the ground truth, the error must be the minimum one, and it must increase when the prediction and the ground truth differ.

In the *Supervised Learning paradigm*, we can identify two major groups of loss functions: *Regression* and *Classification* losses.

The *Regression* losses are used on tasks where the goal is to predict continuous numeric values (regression tasks). For instance, the length of an object in an image, the temperature for a specific day, and a stock price evolution are common tasks where regression losses are used. The most standard regression losses are the *L1* and *L2* errors.

The *L1 loss* function, also known as the mean absolute error (MAE), is defined in Equation 2.10, where n is the number of samples.

$$\mathcal{L}_{mae} = \frac{1}{n}\sum_{i=1}^{n} |\vec{y}_i - \vec{\hat{y}}_i| \tag{2.10}$$

L1 loss function penalizes more small errors than higher ones, as depicted in Figure 2.7. One of the advantages of the *L1 loss* function is its robustness to the outliers.

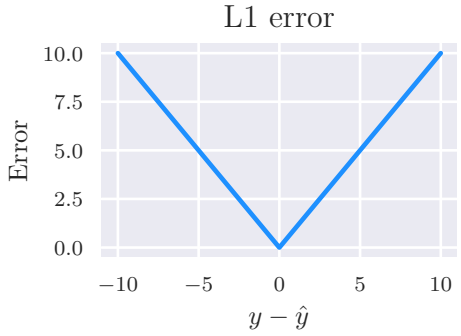


Figure 2.7: L1 loss function.

The *L2 loss*, also known as the mean square error (MSE), is defined in Equation 2.11, where n is the number of samples.

$$\mathcal{L}_{mse} = \frac{1}{n}\sum_{i=1}^{n}(\vec{y}_i - \vec{\hat{y}}_i)^2 \tag{2.11}$$

L2 loss function penalizes more big errors than smaller ones, as depicted in Figure 2.8. One of the advantages of the *L2 loss* is that the gradients are smaller near 0, making it easier for the optimizer to find the best solution.

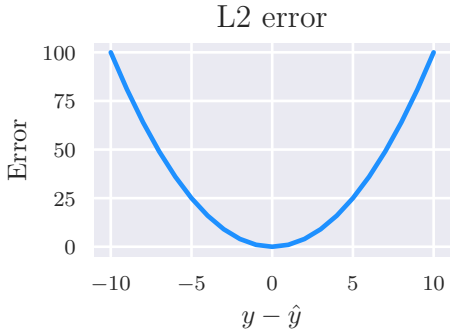


Figure 2.8: L2 loss function.

Usually, for regression problems, *L2 loss* is preferred. However, if the dataset has outliers, the *L1 loss* might perform better.

The *classification* losses are used on tasks where the goal is to predict discrete values (classification tasks), where the number of possible outputs is known. For instance, an application capable of classifying images into different categories is an example of a classification task. The ground truth, $\vec{y}_i$, is a one-hot probability distribution over k classes. In order to convert the network predictions to probability distributions, $\vec{\hat{y}}_i$, a *Softmax* activation function is used. To compare both probability distributions, the *cross-entropy* loss function is normally used, which is depicted in Figure 2.9 and defined in Equation 2.12, where n is the number of samples and k is the number of classes.

$$\mathcal{L}_{ce} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} -y_{i,j}log(\hat{y}_{i,j}) \tag{2.12}$$

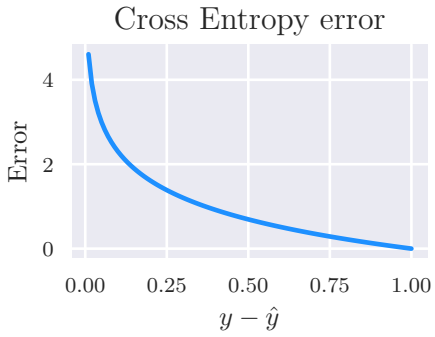


Figure 2.9: Cross-entropy loss function.

2.1.3.2 Initialization

Initializing the weights and biases is an important step that can make the difference between good performance and *not converging at all*. Nowadays, this initialization is done by drawing each weight, independently and identically distributed, from a probability distribution. Frequently, these are either the normal or uniform distributions with zero mean, for each weight, $w_i \sim \mathcal{N}(0, \sigma^2)$ or $w_i \sim \mathcal{U}(-a, a)$, where a is the maximum value that we can get from the uniform distribution. Meaning that if we use the normal distribution, only the *variance*, σ^2, of the weights needs to be specified.

For the sake of simplicity, let us assume a *linear neuron* with zero-mean and uncorrelated inputs, $\vec{x}$, which means that the inputs have been *standarized* ($\mu(\vec{x}) = 0$ and $\sigma(\vec{x}) = 1$), and weights $\vec{w}$. Consider the variance of the output of the neuron defined in Equation 2.13, where n_{in} is the fan-in, the number of inputs to the neuron.

$$\text{Var}\left(\sum_{i=1}^{n_{in}} w_i x_i\right) = \sum_{i=1}^{n_{in}} \text{Var}(w_i x_i) = \sum_{i=1}^{n_{in}} \text{Var}(W)\text{Var}(X) = n_{in}\sigma^2 \text{Var}(X) \tag{2.13}$$

To guarantee consistent gradient updates during the optimization step, explained in Section 2.1.3.3, we need to preserve the *variance of the input*. To do so, we need to set $\sigma^2 = \frac{1}{n_{in}}$ as is done on the *LeCun initialization* [58]. For the same reason, it would be interesting to preserve the variance of the computed weight updates. To achieve this, we need to apply the condition $\sigma^2 = \frac{1}{n_{out}}$, where n_{out} is the fan-out, the number of neurons that directly receive this neuron output. Typically, it is unfeasible to satisfy both conditions at once. To solve that, the *Xavier initialization* [31] proposes to compute the average of both conditions as formalized in Equation 2.14. Nowadays, the *Xavier initialization* is the standard for linear and sigmoidal neurons.

$$\sigma^2 = \frac{2}{n_{in} + n_{out}} \tag{2.14}$$

A similar argument for the ReLU family functions leads to the *He initialization* [40] described in Equation 2.15.

$$\sigma^2 = \frac{2}{n_{in}} \tag{2.15}$$

2.1.3.3 Optimization

Once we have selected the proper loss function to evaluate the network and its parameters have been properly initialized, a learning algorithm is applied iteratively to fine-tune the set of parameters, W and $\vec{b}$. The loss function dictates the suitability of the network's parameters for the defined task and is the primary objective of the optimization algorithm.

The optimization algorithm commonly used in the training of neural networks is the *mini-batch stochastic gradient descent (SGD)* [51]. A mini-batch, $\mathcal{B}$, of n training examples is selected at each iteration. After that, the *backpropagation algorithm* [92] is applied to compute the *gradient* of the loss function, with respect to the network's parameters, $\vec{\theta}$, at time t, which is defined as $\vec{g}_t = \nabla_{\vec{\theta}} \mathcal{L}_{\mathcal{B}}|_{\vec{\theta}=\vec{\theta}_t}$. The backpropagation algorithm applies the *chain rule* for partial derivatives, proceeding *backwards* from the output layer towards the network's input layer. Once we have the gradients, the parameters are updated by doing a step of the gradient descent, described in Equation 2.16 where λ is the *learning rate*.

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \lambda \vec{g}_t \tag{2.16}$$

The choice of the learning rate is important to guarantee stability and a good convergence rate during training. Recent advances in *adaptive optimizers* that dynamically adjust the learning rate based on previous gradient updates have reduced the impact of a wrong choice in the learning rate. Nowadays, *Adam optimizer* [52] is widely adopted by the community. *ADAM* maintains exponential moving averages of both $\vec{g}_t$ and $\vec{g}_t^2$, which are defined in Equation 2.17.

$$\vec{m}_{t+1} = \beta_1 \vec{m}_t + (1-\beta_1)\vec{g}_t \qquad \vec{v}_{t+1} = \beta_2 \vec{v}_t + (1-\beta_2)\vec{g}_t^2 \tag{2.17}$$

The vectors $\vec{m}_t$ and $\vec{v}_t$ are initialized with zeros at time $t = 0$. The mixing constants β_1 and β_2 are hyper-parameters that the user can control. Normally, these parameters have as default values $\beta_1 = 0.9$ and $\beta_2 = 0.999$. This heavily biases the averages towards zero at the beginning. For this reason, the *bias correction* described in Equation 2.18 is applied.

$$\hat{\vec{m}}_t = \frac{\vec{m}_t}{1-\beta_1^t} \qquad \hat{\vec{v}}_t = \frac{\vec{v}_t}{1-\beta_2^t} \tag{2.18}$$

Finally, these averages are used to update the network parameters. The update rule applied is defined in Equation 2.19 where λ is the *learning rate*, and ϵ is a small constant used to protect against division by zero.

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \lambda \frac{\vec{\hat{m}}_t}{\sqrt{\vec{\hat{v}}_t} + \epsilon} \tag{2.19}$$

In practice, the applied optimization algorithm consists of the following steps. First, we shuffle the training set, then sequentially take batches from the shuffled training set and apply the optimizer algorithm for each. Once the entire training set is covered, an epoch of the training is completed, and the training set is reshuffled. This is done to avoid the network memorizing instead of learning.

2.1.3.4 Fighting overfitting

Deep Neural Networks are models composed of a vast number of parameters that are normally optimized on training sets of smaller sizes than the number of parameters that the network has. For this reason, these models tend to overfit the training set, which means that the model fails to generalize outside the training set (the network memorizes the training set). Regularization techniques are usually used to fight against the overfitting problem. Essentially, these techniques introduce additional constraints that somehow restrict the set of parameters available during training, trying to discourage the network from memorizing. In this thesis, the main regularization techniques used are described in the following paragraphs.

L_2 **regularization**: This technique imposes a constraint that makes the parameters not deviate too far from zero. This regularization is implemented as a weight penalty term added to the loss function. The penalty term is based on the L_2 norm as formulated in Equation 2.20, where λ is a constant that controls the importance of penalizing the weights compared to optimizing the loss function.

$$\tilde{\mathcal{L}} = \mathcal{L} + \frac{\lambda}{2} \|\vec{\Theta}\| \tag{2.20}$$

Dropout [104]: This technique tries to overcome the problem of having *highly specialized neurons* that appears during an overfitting scenario. This method proposes randomly killing each neuron in a layer with probability p during training; in each iteration, a different set of neurons is killed. To preserve the expected activation value, the output of the neurons that have not been killed is scaled by $\frac{1}{1-p}$.

Batch Normalization [45]: The initialization techniques explained in Section 2.1.3.2 try to preserve the statistics of the input signal as it propagates through the network. However, during optimization, it is expected that weights are pushed in a direction that will change the intermediate layers' statistics. This effect is known as the *internal covariate shift*. To solve this, batch normalization re-normalizes the network's activations across each training minibatch. Let $\mathcal{B} = \vec{x}_1, ..., \vec{x}_m$ be the outputs of a layer across a minibatch. We apply the renormalization described in Equation 2.21 to obtain the new outputs, $\vec{y}_i$. The γ and β are *learnable* parameters, allowing the network to learn its own normalization scheme. At test time, the values of $\vec{\mu}_{\mathcal{B}}$ and $\vec{\sigma}^{\,2}$ are taken from the entire training set. This is usually computed as an exponential moving average during training.

$$\vec{\mu}_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}\vec{x}_i \qquad \vec{\sigma}^{\,2} = \frac{1}{m}\sum_{i=1}^{m}(\vec{x}_i - \vec{\mu}_{\mathcal{B}})^2 \qquad \vec{\hat{x}}_i = \frac{\vec{x}_i - \vec{\mu}_{\mathcal{B}}}{\sqrt{\vec{\sigma}^{\,2} + \epsilon}} \qquad \vec{y}_i = \gamma\vec{\hat{x}}_i + \beta \tag{2.21}$$

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a particular kind of neural network suitable to process data that can be represented in a grid-like structure, such as images which are represented as tensors of shape $h \times w \times d$, being h the height, w the width and d the channels of an image. Processing grid-like structures with MLPs becomes unfeasible for large structures since the MLP treat each input element independently. In the case of an image, $O(h \times w \times d)$ parameters are introduced per neuron in the first hidden layer. Consequently, the memory requirements increase exponentially, and the neural network is prone to overfit. In order to make a parameter-efficient network, we need to exploit the spatial structure present in such inputs, a candidate for such a layer is the convolutional operator.

2.2.1 Convolutional layers

Let us assume a 2D input, $X \in \mathbb{R}^{h\times w}$, and a small kernel matrix, $K \in \mathbb{R}^{k_h \times k_w}$. The convolution operation used on CNNs overlays in all possible ways the kernel matrix over the input, recording sums of elementwise products that yield a new 2D structure, $Y \in \mathbb{R}^{h\times w}$, as defined in Equation 2.22. To ensure the same input size on the output structure, the input is normally padded as much as needed with zeroes. It should be noted that this approach is actually a *cross-correlation*, which is the same as convolution but without flipping the kernel. However, in Deep Learning, both are called convolution. In this thesis, we follow this convention of calling both operations convolution.

$$Y_{i,j} = b + \sum_{n}^{k_h}\sum_{m}^{k_w} X_{i+n,j+m} \cdot K_{n,m} \tag{2.22}$$

We need a few extensions of the previously defined convolution to define the convolutional layer in order to support multiple input and output channels:

- The number of input channels should match the number of channels of the kernel, meaning that if we have an input $X \in \mathbb{R}^{h\times w\times d}$ we need a kernel matrix $K \in \mathbb{R}^{k_h \times k_w \times d}$. This kernel tensor specifies a single output channel.

- Each output channel requires a separate kernel tensor.

- A channel-specific bias and activation function may be applied.

To process a 2D input $X \in \mathbb{R}^{h \times w \times d}$ using a convolutional layer that computes d' output channels, we require a kernel tensor $K \in \mathbb{R}^{k_h \times k_w \times d \times d'}$ and a bias vector $\vec{b} \in \mathbb{R}^{d'}$ as defined in Equation 2.23.

$$Y_{i,j,c} = \sigma\left(\vec{b}_c + \sum_{n}^{k_h} \sum_{m}^{k_w} \sum_{k}^{d} X_{i+n,j+m,k} \cdot K_{n,m,k,c}\right) \tag{2.23}$$

This kind of layer is widely used to process 2D images since it exploits the structure of the image with neighbouring pixels influencing each other more strongly than the ones further. In addition, this operator is translation invariant, as the same kernel is applied across the image.

2.2.2 Pooling layers

Convolutional Neural Networks consist of a series of interleaved convolution and pooling layers until the input is reduced sufficiently to be processed by an MLP. Applying pooling layers helps the network reduce the amount of memory needed and enables the network to learn richer high-level features.

When a particular pattern is detected in the input, the convolution filter gets activated (returns high values). It makes sense to preserve its maximally activated components when summarizing using pooling layers. This is the reason behind most of the current CNNs for image processing [55, 98] using *max-pooling* layers, which takes $n \times m$ image patches across each channel and summarizes them by taking only the maximal value, as depicted in Figure 2.10. In most of the current architectures $n = m = 2$.

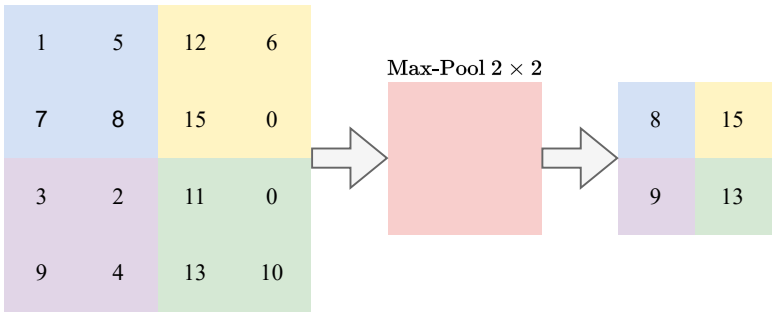


Figure 2.10: An example of applying a 2×2 Max-Pool layer to an input of 4×4.

2.2.3 Residual networks

Training a very Deep Convolutional Network is extremely difficult. As proved in [41], extremely Deep Convolutional Networks do not overfit. Instead, their training and testing performance degrades. This behaviour is counterintuitive since deep convolutional networks have more parameters which, in theory, leads to more overfitting. However, the issue comes with the extra layers added. If the new layers cannot learn good representation, they could add noise to the feature space, making them useless.

Residual skip connections [41] is the intuitive fix for the problem described. These connections *shortcut* a particular block of operations in a neural network. For a particular block of operations inside a Deep Convolutional Network, $\mathcal{F}(\vec{x})$, a residual skip connection allows $\vec{x}$ to go directly to the output, which is added

to the output of this specific block, $\mathcal{F}'(\vec{x}) = \mathcal{F}(\vec{x}) + \vec{x}$. Equation 2.24 generalizes the residual skip connections for the case where a block's input and output dimensionality are not the same. Using a linear projection, $\mathcal{P}$, the input $\vec{x}$ is projected to the same output dimensionality before being added to the block's output (if input and output dimensionality are the same, $\mathcal{P}$ is equal to the identity function).

$$\mathcal{F}'(\vec{x}) = \mathcal{F}(\vec{x}) + \mathcal{P}(\vec{x}) \tag{2.24}$$

2.3 Graph Neural Networks

Convolutional Neural Networks (GNNs) have been successfully applied to address problems such as image classification [41] or machine translation [27], where the underlying data representation has a grid-like structure. However, many tasks involve data that can not be represented in a grid-like structure, such as 3D meshes, social networks, and biological networks, whereas such data can be represented using graphs. A graph $G = (N, E)$ is composed of a set of nodes or vertices, N, and a set of edges, E. Graph structures are a natural generalization of different inputs such as images, point clouds, text and speech. Designing appropriate methodologies for graphs on neural networks is one of the major ongoing challenges in machine learning [5, 8, 37, 53]. This thesis explores how we can use graph structures to deal with 3D data. For now, let us assume that we already have our graph G. For each graph, we have a matrix of *node features*, $F \in \mathbb{R}^{n \times d}$, with a number of nodes n and a number of features d, as well as an *adjacency matrix*, $A \in \mathbb{R}^{n \times n}$. In this thesis, we do both graph and node analysis. The graph analysis consists of summarizing the graph to get a prediction $Y \in \mathbb{R}^{d'}$, with d' output features related to the entire graph. In contrast, the node analysis is a prediction $Y \in \mathbb{R}^{n \times d'}$, with d' output features in each of the n nodes. In both cases, the prediction can be continuous, as it is in the case of regression, or it can be discrete and bounded, as it is in the case of classification.

In the case of node analysis, there are two kinds of learning tasks:

- *Transductive* Learning: The algorithm has access to the whole graph and its features during training, *including the test nodes*. The goal is to propagate the labels from labelled nodes to non-labelled nodes.

- *Inductive* Learning: The algorithm does not have access to all nodes upfront. This is the typical scenario when we are dealing with an *evolving* graph wherein new nodes are incrementally added or when disjoint and unseen graphs exist during the test phase. Inductive learning is more challenging since it needs to generalize across *arbitrary graph structures*.

The intuitive way to deal with 3D data and graphs would be to process each node *indepdendently* using an MLP per node. This setup is used by *PointNet* [84], where each node is processed independently with a combination of max pooling and average pooling layers, used to get global feature descriptors to try to describe the 3D data. However, this approach *drops* the inherent structure of the data.

The first methodologies to consider the graph structure and get features that describe this structure were the methods based on the *Random-walk* algorithm. Such methods decouple the graph structure from the node

features. For each node i, structural features, $\vec{s}_i$, are computed only using the adjacency matrix information. Afterwards, the structural features and the input features are concatenated, $\vec{x}_i||\vec{s}_i$, which can be used as input of an MLP. The first method to successfully exploit the *Random-walk* algorithms is *DeepWalk* [82], which draws inspiration from *skip-gram* methods in natural language processing [75]. *DeepWalk* generalizes *skip-gram* methods for graphs by considering random walks as *sentences* and nodes as *words*. Most of the random-walk-based methods that came after it, such as *node2vec* [34], *LINE* [106] and *Planetoid* [121], focus on improving the way in which the random walks are constructed.

In this thesis, we focus on techniques that leverage the graph structure directly while extracting intermediate feature representations, $\vec{x}_i$, for each node, i, in the graph. In the literature, these techniques may be referred to as *Graph Neural Networks* [33, 64, 94] or *Graph Convolutional Networks* [10, 16, 53]. In this thesis, we use the term *Graph Convolutional Networks*, as it relates them to a generalization of the *convolutional layer* from CNNs to graph-based networks. In general, all *graph convolutional layers* can be reformulated as a particular instance of the *message-passing architecture* [28].

2.3.1 Message-passing architecture

In this section, we introduce the *message-passing architecture* depicted in Figure 2.11, composed of 4 main steps, explained below:

- **Selection**: For each of the nodes i, of the graph, we select the neighbouring nodes, $j \in N(i)$. This selection is made by taking into account the direction of the edges.
- **Message**: The message is a function $\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)$, that depends on the features $\vec{x}^t$, of the nodes i, j and the edge attribute $\vec{e}_{i,j}^t$.
- **Aggregation**: All the messages that the central node of the neighbourhood receives are aggregated, $\underset{j \in N(i)}{Aggr}\left(\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)\right)$.
- **Update**: Each node updates its feature $\vec{x}_i^{t+1}$, as a function of the current node feature $\vec{x}_i^t$, and the resulting aggregation previously done, $\gamma(\vec{x}_i^t, \underset{j \in N(i)}{Aggr}\left(\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)\right))$.

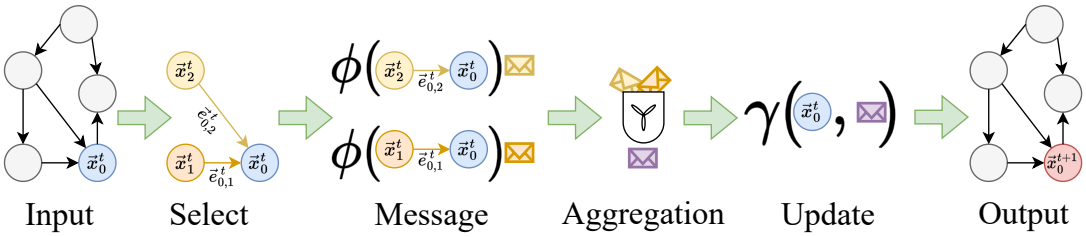


Figure 2.11: Message-passing architecture applied on node $\vec{x}_0^t$, where ϕ is the message function and γ is the update function.

The original *message-passing architecture* can be simplified by adding self-loops, $i \in N(i)$, to the input graph and omitting the explicit update step. This simplification is formalized in Equation 2.25, where σ is the

activation function and b the bias term. In this thesis, we use this simplification of the *message-passing architecture* and assume that all graphs contain self-loops.

$$\vec{x}_i^{t+1} = \sigma\left(\underset{j \in N(i)}{Aggr}\left(\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)\right) + b\right) \tag{2.25}$$

2.3.2 Graph Convolutional Layers

To extend neural networks to deal with graphs, early works proposed to use Recursive Neural Networks (RNNs) to process directed acyclic graphs [26, 103]. Afterwards, a generalization of RNNs that can directly deal with different kinds of graphs, such as cyclic, directed and undirected graphs, was introduced in [33, 103]. These first approaches of Graph Neural Networks consist of an iterative process, which propagates node states until equilibrium. Then a neural network produces an output for each node based on its state. This idea was improved by [64], which proposes to use gated recurrent units [14] in the propagation step. The main issue with these first approaches is the restriction from specifying a true analogue of a convolutional layer with an arbitrary number of filters since the dimensionality of the feature computed at each stage needs to be fixed.

The generalization of convolutions to the graph domain can be categorized into two groups: *spectral* and *spatial* approaches.

Spectral approaches use graph spectral analysis theory where the convolution corresponds to the multiplication of the signal on vertices transformed into the spectral domain, using the Graph Fourier transform. In [10], the Fourier transform is done by computing the eigendecomposition of the Graph Laplacian, provoking intense computations and non-spatial localized filters. These issues were addressed by [43], which proposed the parameterization of the spectral filters with smooth coefficients to make them spatially localized. [16] proposed a parameterization of filters using Chebyshev polynomials that is computationally more efficient. The main drawback of spectral approaches is that a model trained on a specific graph structure can not be directly applied to a graph with a different structure.

Spatial approaches defines the convolution directly on the graph, operating on groups of neighbourhoods. Nowadays, there are different ways to generalize the convolution operator following a spatial approach [1, 21, 36, 76, 80]. This thesis focuses on these methods since they can be generalized for unseen graph structures. We are going to review the most popular graph convolutional layers used nowadays to process 3D data. The *Graph Convolution layer* [53] defined in Equation 2.26 is one of the first spatial approaches. It proposes to learn the weight matrix, $W \in \mathbb{R}^{d \times d'}$, shared between all the nodes, where d are the input channels, and d' are the output channels. The result is aggregated using an average aggregator that considers the neighbourhood degree, $deg(i)$.

$$\vec{x}_i^{t+1} = \sigma\left(b + \sum_{j \in N(i)} \frac{W\vec{x}_j^t}{deg(i)}\right) \tag{2.26}$$

Graph Attention Network (GAT) [111] proposes the graph attention layer which is defined in Equation 2.27. In this layer, an attention coefficient is learnt using the edge attributes, which are the result of the concatenation of $\vec{x}_i^t$ and $\vec{x}_j^t$ as defined in Equation 2.28. For each head of attention, h, we have a linear transformation, $\mathcal{F}^h$, implemented using an MLP. The output of each linear transformation is passed through a softmax layer to guarantee that the sum of the attention coefficients is equal to one. In this formulation, we have two kinds of aggregators, one for the messages and another one for the heads of attention. In the case of the message, the addition aggregator is used. Whereas to aggregate the heads of attention, two aggregators are proposed: the concatenation and the average.

$$\vec{x}_i^{t+1} = \sigma\left(b + \underset{h=1}{\overset{H}{Aggr}}\left(\sum_{j \in N(i)} \alpha_{i,j}^h W^h \vec{x}_j^t\right)\right) \tag{2.27}$$

$$\alpha_{i,j}^h = softmax_j(\mathcal{F}^h(\vec{x}_i^t || \vec{x}_j^t)) \tag{2.28}$$

The *Feature-Steered Graph Convolution* (FeaStConv) proposed in FeaStNet [112], which is defined in Equation 2.29, is quite similar to the formulation proposed in GAT. The main difference resides in how the edge attributes are computed. As described in Equation 2.30, the edge attributes are defined as the difference between $\vec{x}_j^t$ and $\vec{x}_i^t$. The aggregator used to combine the messages is the average aggregator that considers the neighbourhood's degree. The heads of attention are aggregated using the addition aggregator.

$$\vec{x}_i^{t+1} = \sigma(b + \sum_{j \in N(i)} \sum_{h=1}^{H} \frac{q^h(\vec{x}_i^t, \vec{x}_j^t) W^h \vec{x}_j^t}{deg(i)}) \tag{2.29}$$

$$q^h(\vec{x}_i^t, \vec{x}_j^t) = softmax_j(\mathcal{F}^h(\vec{x}_j^t - \vec{x}_i^t)) \tag{2.30}$$

The *Edge Convolution* (EdgeConv) [115] described in Equation 2.31 proposes to define the edge attributes using an asymmetric function. The edge attributes of each neighbourhood are fed into an MLP and aggregated using a maximum aggregator.

$$\vec{x}_i^{t+1} = \sigma\left(b + \underset{j \in N(i)}{MAX}\left(\mathcal{F}(\vec{x}_i^t || (\vec{x}_j^t - \vec{x}_i^t))\right)\right) \tag{2.31}$$

One of the problems detected by Xu [119] is that most of the current Graph Convolutional Neural Networks fail to get rich descriptors to distinct neighbourhoods with different degrees but equal features. This thesis proposes new graph convolution layers that consider this limitation and tries to learn better local representations. In Chapters 3 and 4, we propose the Attention Graph Convolution (AGC), a new layer that infers the W tensor in run-time, helping to find specific weights for each neighbourhood. An improved version of AGC is proposed

in Chapter 5. Motivated by the work of Dehmamy *et al.* [17], which proposes to use multiple aggregators, we present the Multi-Aggregator Graph Convolution (MAGC), an extension of the message-passing architecture using complementary aggregators and learning how to combine them instead of just concatenating or adding the results of each of them.

2.4 3D Data Structures

In this thesis, we will work with two different data structures: 3D point clouds and 3D meshes.

A **point cloud** is a collection of points in a 3D space. Each point has its set of cartesian coordinates (x, y, z) and features, such as colour, as shown in Figure 2.12. Point clouds are normally produced by 3D scanners, some of them directly output a point cloud, such as lidar, and others return a depth map that has to be back-projected into a 3D space. This back-projection is normally done by applying the pin-hole camera model after correcting all the distortions introduced by the lens of the sensor. The back-projection used in this thesis is described in Equation 2.32 being $[x, y, x]$ the 3D coordinates in the camera coordinate system and $[u, v]$ the coordinates in the image. The camera's focal length is represented by $[f_x, f_y]$ and the principal point is represented as $[c_x, c_y]$.

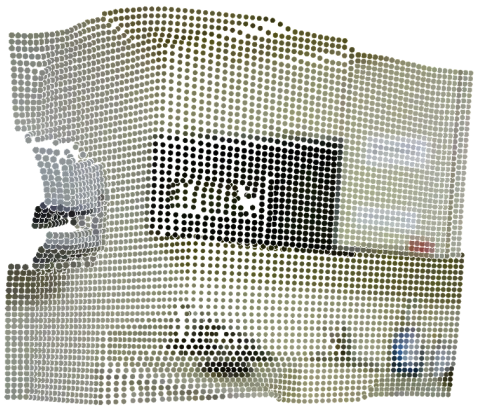


Figure 2.12: Example of a 3D Point Cloud from the NYU-Depth-V2 dataset [78] with RGB values.

$$\left.\begin{aligned} z &= \text{depth} \\ x &= \frac{(u - c_x) \cdot z}{f_x} \\ y &= \frac{(v - c_y) \cdot z}{f_y} \end{aligned}\right\} \tag{2.32}$$

A **mesh** is a collection of *vertices*, *edges* and *faces* that defines the shape of a polyhedral object:

- The *vertices* and the points of the point cloud are equivalent. Each vertex has its set of cartesian coordinates (x, y, z) and a feature such as colour.

- The *edge* defines the connection between two vertices.

- The *face* is a set of closed edges, in which a triangular face has three edges, and a quadrilateral face has four edges.

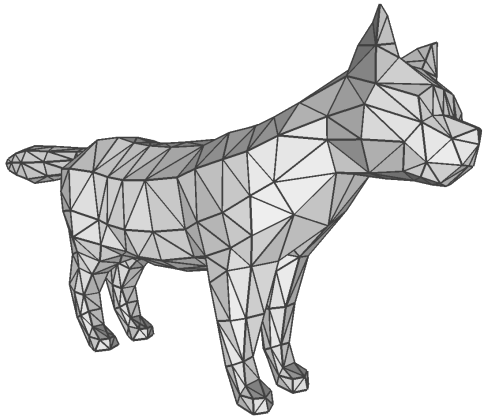
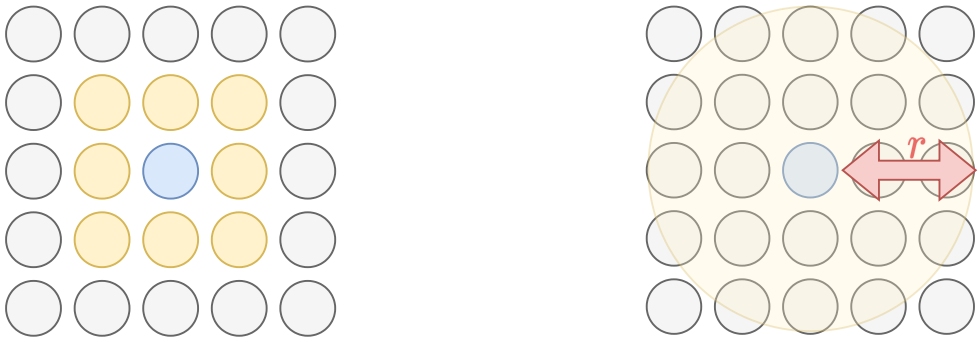


Figure 2.13: Example of a 3D mesh of a cat with triangular faces.

To generate a graph from a mesh or a 3D point cloud, all the vertices and 3D points are considered a node of the graph. The adjacency matrix of the graph can be generated with two different policies: the *k nearest neighbours (kNN)* and *radius proximity* policy, both of them depicted in Figure 2.14. Both policies generate a directed graph and use the euclidean distance to find the neighbouring nodes.

The *kNN* policy consists of selecting the k closest neighbours in the geometric space of each node of the graph and connecting them. k is constant for all the nodes of the graph. This means that each node of the graph has the same number of neighbours. The receptive field of this policy depends on the density of the point cloud.

The *radius proximity* policy consists of selecting all nodes in the graph that are inside of a defined radius, r_g, as neighbours for each node. The radius is constant for each node. Therefore, each node may have a different number of neighbours. However, to limit the memory needed on dense point clouds, the number of possible connections is limited to k_{max}, which means that for all the candidates, k_{max} random nodes are selected as neighbours.



(a) *kNN* policy.

(b) *Radius proximity* policy.

Figure 2.14: Comparison between the *kNN* policy and the *radius proximity* policy with radius r.

In the case of the mesh, we can generate the adjacency matrix, converting the faces of the mesh into undirected edges. This is called the *one-ring* neighbourhood and is depicted in Figure 2.15.

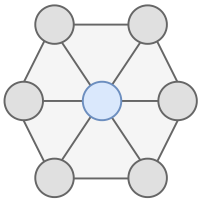


Figure 2.15: Example of a one-ring neighbourhood.

Afterwards, the features of the nodes $X \in \mathbb{R}^{n \times d}$ and the attributes of the edges $E \in \mathbb{R}^{n \times n \times a}$ must be defined, being n the number of nodes of the graph, d the input feature dimension of the node and a the attributes dimension of an edge.

Part I

Graph Analysis

chapter 3

Residual Attention Graph Convolutional Network

Inherently, we have to deal with unstructured data for most of 3D image analysis. Appropriately exploiting the intrinsic information of this data remains a key challenge, especially when the information comes from noisy sensors. Extending current neural network architectures to properly deal with unstructured data is an essential research direction for 3D image analysis; however, it has received low levels of attention until very recently.

In this chapter, we will introduce Residual Attention Graph Convolutional Network (RAGC) [107], a Graph Convolutional Neural Network architecture that extends the residual connections used in Convolutional Neural Networks [41] to unstructured data. In addition, we propose a new graph convolution operation that generates its own kernel conditioned by the edge attributes of the local neighbourhood, learning to pay attention to the important nodes of the current neighbourhood.

To prove the validity of our method, we use 3D point clouds. Specifically, we use captures of different indoor scenes provided by public datasets. To better assess the capabilities of the proposed method to capture the geometric information, in this chapter, we restrict ourselves to the problem of geometric classification using just the geometric information without using any additional features such as colour.

The main contributions of this chapter are:

- The Attention Graph Convolution layer, which learns to predict a set of weights that depends on the neighbourhood configuration.

- The extension of residual connections to the Attention Graph Convolution layer.

- The Voxel Pooling layer, which allows mimicking the traditional CNN architectures and learning richer, higher-level features.

3.1 Related Work

3.1.1 Attention methodologies

Visual attention enables humans to analyze complex scenes and devote their limited perceptual and cognitive resources to the most important of sensory data. Attention models aim to identify the most attractive regions in data like human visual systems do. *Xu et al.* [118] introduced an attention-based model that learns to describe the content of images. Their model can automatically learn to fix its gaze on salient objects. *Ren et al.* [88] proposed an end-to-end Recurrent Neural Network architecture with an attention mechanism that produces detailed instance segmentation. Inspired by these recent works, Graph Attention Convolution (GAT) [111] proposed an extension of these mechanisms into graph structures explained in detail in Section 2.3.2. The main difference between Attention Graph Convolution (AGC) and Graph Attention Convolution (GAT) [111] is that the AGC predicts the weights used on the convolutional kernel depending on the neighbourhood composition. In contrast, GAT learns a set of weights that will be shared with all the neighbourhoods of a graph.

3.1.2 Geometric Scene Classification

Earliest works used traditional handcrafted features such as SIFT [9] and HOG [110] to classify geometrical captures. With the emergence of Deep Learning techniques, better features can be obtained. *Socher* et al. [99] used a combination of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) to learn the geometric information used during the classification. *Cai* et al. [11] proposed a new Convolutional Neural Network that extracts regions of interest to capture local and global structures from the depth channel to classify a scene. Furthermore, they proposed to use *RICA*, a method proposed by *Le* et al. [57], to classify geometric information. This method is an improved version of the *Independent Component Analysis* (ICA) which uses a soft reconstruction cost for ICA that allows the method to learn highly overcomplete sparse features even on unwhitened data. Finally, well-known network architectures, such as ResNet [55] and VGG [98], have been used for geometric classification being pre-trained on a huge RGB dataset as Places [127].

3.2 Residual Attention Graph Convolutional Network

This section presents our Graph Convolutional Network to tackle the geometric scene classification problem. The input of the network is a 3D point cloud that can be obtained from a lidar sensor or using the depth information and the intrinsic parameters of a depth sensor. Each node of the 3D point cloud encodes the depth information using the *HHA* [35] encoding, as done by previous works [11, 35]. *HHA* encodes the depth into a $[0, 255]$ range with three channels. Each channel represents horizontal disparity, height above the ground, and the angle with the inferred gravity direction. In Sections 3.2.1, 3.2.2 and 3.2.3, we introduce the key components of our proposed network depicted in Figure 3.1.

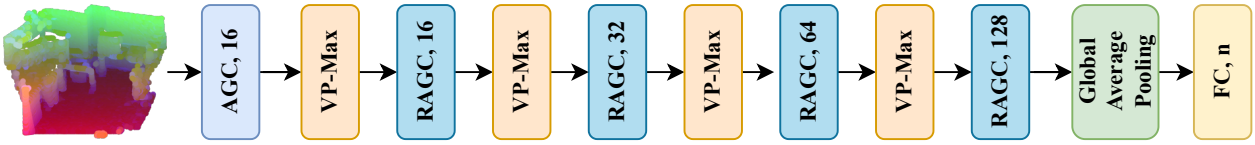


Figure 3.1: Residual Attention Graph Convolutional Network architecture, where n is the number of classes, and FC is a fully-connected layer. The Global Average Pooling layer computes a global descriptor computing the average of all node features.

3.2.1 Attention Graph Convolution

Attention Graph Convolution (AGC) is a graph convolution that performs the convolution over local graph neighbourhoods exploiting the edges and their attributes, E. AGC is formalized in Equation 3.1, where $\vec{x}_i$ is the feature vector of the node i, $N(i)$ the set of neighbours of node i and $\Theta_{i,j}$ is the predicted weight matrix that is used to generate the message for node j. In AGC, the average aggregator that considers the neighbourhood degree, $\deg(i)$, is used to fuse the messages. An MLP network is used to generate the weights, $\Theta \in \mathbb{R}^{n \times n \times d \times d'}$, where n is the number of nodes, d is the number of input features of each node, and d' is the number of expected output features per node. This network uses E as input and outputs for each of the edges, $i \leftarrow j$, a matrix of weights, $\Theta_{i,j} \in \mathbb{R}^{d \times d'}$, formalized in Equation 3.2.

$$\vec{x}_i^{\,t+1} = \frac{1}{deg(i)} \sum_{j \in N(i)} \Theta_{i,j}^t \vec{x}_j^{\,t} \tag{3.1}$$

$$\Theta_{i,j}^t = MLP(\vec{e}_{i,j}^{\,t}) \tag{3.2}$$

In AGC, the attributes of the edges are defined as the positional offset $\vec{e}_{i,j}^{\,t} = \vec{p}_j - \vec{p}_i$, where $\vec{p}$ is the 3D position of the nodes i and j. These offsets can be represented in *cartesian* or *spherical* coordinates. The workflow of the AGC operation is depicted in Figure 3.2. The attention stage of AGC is located in the weight generation stage. Due to the design of this layer, the network learns to infer a weight value that pays attention to the attributes of the edge, specifically in the geometric information encoded as an edge attribute.

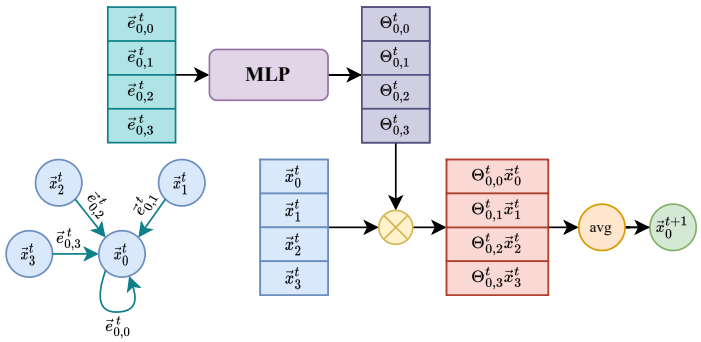


Figure 3.2: Attention Graph Convolution workflow. $\vec{x}_i^{\,t}$ is the input feature vector of the node i, $\vec{e}_{i,j}^{\,t}$ is the edge's attribute vector of the edge $i \leftarrow j$, and $\Theta_{i,j}$ is the predicted weight matrix that is used to generate the message for node j.

3.2.2 Residual Attention Graph Convolution

The previous *AGC* is extended to a *Residual Attention Graph Convolution (RAGC)* following the inspiration of the ResNet [41] architecture. Residual connections tackle the counterintuitive problem that emerges when training *Deep Neural Networks*, which degrades their training and testing performance instead of overfitting. The intuitive fix for this situation is to use residual skip connections, which *shortcut* a particular block of layers in a neural network. Figure 3.3 depicts a RAGC block composed of two AGCs.

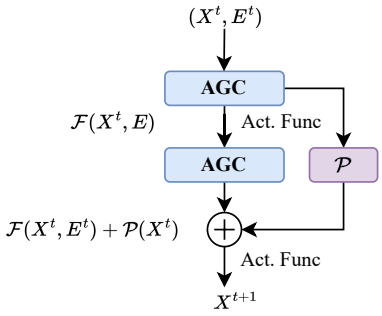


Figure 3.3: Residual Attention Graph Convolution block composed of two AGCs.

The RAGC block requires as input the node features and the edge attributes, (X^t, E), which are passed through both of the AGC layers, $\mathcal{F}(X^t, E)$. In parallel, the node features X^t are passed through a projection function, $\mathcal{P}(X^t)$, that projects the input space to the output space. However, in the case that the input and output spaces have the same dimensionality, $\mathcal{P}(X^t)$ becomes the *identity* function. Finally, we add the output of both branches yielding the new output feature, X^{t+1}, as described in Equation 3.3. The $\mathcal{P}(X^t)$ function is implemented as a 1D convolution. $\mathcal{F}(X^t, E)$ is a stack of s AGC layers, $\{AGC_s\}$.

$$X^{t+1} = \mathcal{F}(X^t, E^t) + \mathcal{P}(X^t) \tag{3.3}$$

3.2.3 Voxel Pooling Layer

In 2D CNN, pooling layers are a key component in learning richer high-level features and reducing the amount of memory needed. In this section, we explain our approach to adopt the 2D pooling layer for 3D sparse data.

The first step is to convert the graph into a 3D *point cloud*, which is a straightforward operation that removes the edges and keeps the nodes as 3D points. Each 3D point contains the 3D coordinates, $\vec{p}_i$, and the feature vector, $\vec{x}_i$. Afterwards, we apply the Voxel Pooling (VP) layer, described in Algorithm 1, which consists in creating voxels of size $\vec{s}$ over the point cloud and replacing all points inside the voxel with their centroid. The position of the centroid is computed as the average of the position of all the points inside the same voxel. The centroid's feature is the aggregation of the features of the points inside the same voxel. Following the same logic as in 2D pooling layers, the aggregation operation is, generally, the maximum to preserve the maximally activated components of the filters, as discussed in Section 2.2.2.

Finally, the downsampled version of the point cloud, composed of all the voxel's centroids, is converted into a graph where the edges and their attributes are re-generated.

Algorithm 1: Voxel Pooling

Input: $\vec{vs} \leftarrow$ voxel's size, $P \leftarrow$ point cloud
Output: $C \leftarrow$ centroid's position, $F \leftarrow$ centroid's feature

Create voxels, V, of size $\vec{vs}$, over the point cloud P
foreach V_i *in* V **do**
    Compute $\vec{c_i} \leftarrow$ average of points' positions inside V_i
    Compute $\vec{f_i} \leftarrow$ aggregation of points' features inside V_i
end foreach

3.3 Experimental Setup

3.3.1 Datasets and Metrics

The SUN RGB-D dataset [100] was captured from different RGB-D sensors, including Asus Xtion, RealSense, Kinect v1 and Kinect v2. Following the settings proposed by the authors, classes with less than 80 samples are discarded, resulting in 9504 captures with 19 different classes. These captures are divided into 4845 for training and 4659 for testing using the split provided by the authors. This dataset is the one used for the ablation studies.

The NYU-Depth-V1 dataset (NYUV1) [97] is split following the division provided by the authors, using 1097 captures for training and 1140 for testing, with 6 classes.

The proposed method is evaluated on both datasets using the *mean accuracy* metric, also known as balanced accuracy score, used to deal with an imbalanced dataset. It is defined as the average of the recall obtained in each class which is formalized in Equation 3.4, where k is the number of classes, tp is a *true positive* classification where the model correctly predicts the positive class, and fn is a false negative classification where the model incorrectly predicts the negative class. In this multi-class scenario, the *positive class* is the class we are currently evaluating, and the *negative class* is the rest of the classes.

$$\text{MeanAcc} = \frac{1}{k} \sum_{i=1}^{k} \frac{tp_i}{tp_i + fn_i} \tag{3.4}$$

3.3.2 Implementation details

3.3.2.1 Pre-processing input data

The input data used in our model comes from depth captures. First, the depth capture is encoded using the HHA encoding [35], downsampled by a factor of 8 and back-projected to a 3D point cloud. The point cloud positions are scaled between $[-1, 1]$ and, during training, the following online data augmentation techniques are applied:

1. Rotation over the vertical axis randomly between $(0, 2\pi)$.

2. Mirroring over horizontal axis randomly with a probability of 0.5.

3. Random removal of points with a probability of 0.2.

4. 3D random crop. A factor, f, is defined to specify the desired number of points inside the crop, which must be in the range of $0 < f < 1$. In this thesis, the values of f are randomly chosen from the range $[0.875, 1]$. The desired number of points (dnp) is defined as $dnp = np \times f$, where np is the number of points of the original point cloud. Then we find a random centroid of a voxel inside the point cloud, where a voxel that satisfies the following condition will be placed: $npi < dnp$, where npi indicates the number of points inside the proposed voxel. The crop is made up of the points inside the voxel.

After applying the data augmentation techniques (only during training), the point cloud is converted into a graph. The proposed network follows the *kNN policy*, where the value of k is 9, and defines the attributes of the edges as the positional offset with the spherical coordinate system, following the procedure explained in Section 3.2.1.

3.3.2.2 Architecture details

The model is implemented with Pytorch [81] and Pytorch Geometric [25]. In terms of the network architecture, the following hyper-parameters are chosen:

- **AGC** layer defines the MLP network used to generate the weights to create the messages as the stack of two MLP layers with output features $(128, d_l \times d_{l-1})$, where d_l is the number of output features of the layer l.

- **RAGC** layer is a stack of two AGCs with a residual connection.

- **VP** layer uses the maximum aggregator with cubic voxels.

The detailed architecture with the number of filters used in each layer and the sizes of the Voxel Pooling layers is shown in Table 3.1.

RAGC Architecture		
Layer	**N. Filters**	**Cubic voxel size (meters)**
AGC	16	-
VP-Max	-	0.05
RAGC	16	-
VP-Max	-	0.08
RAGC	32	-
VP-Max	-	0.12
RAGC	64	-
VP-Max	-	0.24
RAGC	128	-
Global Average Pooling	-	-
FC	n	-

Table 3.1: RAGC architecture details. n is the number of classes, and FC is a fully-connected layer. After each AGC, we have batch normalization and ReLU activation layers. The Global Average Pooling layer computes a global descriptor computing the average of all node features.

3.3.2.3 Training details

The datasets used to train the proposed network are characterized by having an unbalanced number of captures for each category. To handle the imbalance problem, the proposed network is trained using a variation of the cross-entropy named weighted cross-entropy. This loss introduces a weighting factor, $\vec{w}$, that weights each class's contribution to the loss's final value, as formalized in Equation 3.5, where n is the number of 3D point clouds in a batch and k the number of classes. The $\vec{w}$ is computed as the inverse class frequency, $\vec{w}_j = \frac{1}{f(j)}$, where $f(j)$ is the frequency of the class j. Furthermore, to make the total loss be on the same scale when the weight is applied, $\vec{w}$ is normalized so that $\sum_{j=1}^{k} \vec{w}_j = k$.

$$\mathcal{L}_{ce} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} -w_j y_{i,j} log(\hat{y}_{i,j}) \tag{3.5}$$

The network is initialized using the Xavier initialization [31], except for the biases of the last FC, which are initialized as $b = -log((1-\phi)/\phi)$, where $\phi = 1/k$ and k is the number of classes. This initialization aims to avoid the possible training instability that biases $b = 0$ could cause at the beginning of the training, as demonstrated in [15]. The network is trained using the early-stopping criteria with a patience of 20 epochs and a maximum of 200 epochs with a batch size of 32. The optimizer used is the Rectified Adam (RADAM) [67], an improved version of ADAM [52] that rectifies the variance of the adaptive learning rate. The learning rate used is 1×10^{-3}, betas $(0.9, 0.999)$ and a weight decay of 1×10^{-4}. A dropout layer is added before the MLP layer of the *classification network* with a probability $p = 0.2$ to be zeroed.

Moreover, by carrying extensive hyperparameter tuning, we ensure that differences in performance can be attributed to modelling choices rather than incomplete hyperparameter optimization. We use the Hyperband [62] algorithm over the validation split, which is created taking 20% of the samples of the training split. Specifically, we run the hyperparameter optimization over the SUN RGB-D dataset. Once we have selected the hyperparameters, we re-train the network using the complete training split and evaluate the model on the test split.

3.4 Results

3.4.1 Ablation Studies

3.4.1.1 Study of the graph creation

To generate the graph from the point cloud needed to apply a Graph Convolutional Network, two different policies can be used: *radius proximity* and *kNN* policies. To apply the *radius proximity* policy, we need to define a set of radius for each graph convolutional layer. We found that the best set of the radius to be used are $[0.05, 0.08, 0.12, 0.24, 0.48]$, in meters, after applying the hyperparameter search, whereas for the *kNN* policy, we found $k = 9$ works best. As can be observed in Table 3.2, the kNN policy surpasses the mean accuracy obtained with the best configuration of the *radius proximity* policy.

Policy	**Mean Acc. (%)**
kNN	**42.4**
Radius proximity	40.3

Table 3.2: Analysis of kNN and radius proximity as neighbourhood generation policies used on RAGC on SUN RGB-D dataset [100].

Once the graph needed is defined, we need to specify the attribute for each edge. We use the positional offset of the two neighbouring nodes as an attribute of the edge. In this section, we compare the cartesian and the spherical representation of this positional offset. Table 3.3 shows that the spherical offset outperforms the cartesian representation.

Edge Attribute	**Mean Acc. (%)**
Spherical Offset	**42.4**
Cartesian Offset	40.5
Cartesian ‖ Spherical Offset	41.1

Table 3.3: Analysis of the effectiveness of different positional offset representations as edge attributes on SUN RGB-D dataset [100].

3.4.1.2 Study of the MLP used to generate the weights on AGC

To study the influence of the MLP architecture used on AGC to generate the weights, we run several tests using a different number of layers in the MLP, as shown in Table 3.4. We observe that using just one layer yields a loss of 3.3% of the mean accuracy. Adding a second layer that uses half of the optimal found output features yields a loss of 1.4% of the mean accuracy. Finally, adding an extra layer does not help to improve the results; instead, it decreases the mean accuracy.

Configuration	Mean Acc. (%)
$(d_l \times d_{l-1})$	39.1
$(64, d_l \times d_{l-1})$	40.7
$(128, d_l \times d_{l-1})$	**42.4**
$(64, 128, d_l \times d_{l-1})$	41.8

Table 3.4: Comparison of different MLP architectures used on AGC to generate the weights on SUN RGB-D dataset [100].

3.4.1.3 Comparison with different Graph Convolutional Layers

In order to make a fair comparison, we remove all the residual connections from the original architecture and replace the AGC for each of the state-of-the-art graph convolutional layers we previously explained in Section 2.3.2. We observe that the layers that give similar results to AGC are the EdgeConv and the FeaStConv, which AGC outperforms by 1.6% of increment in the mean accuracy, as shown in Table 3.5.

Method	Mean Acc. (%)
AGC	**40.2**
EdgeConv [115]	38.6
FeaStConv [112]	38.3
GAT [111]	37.5
GCN [53]	36.2

Table 3.5: Comparison between AGC and the state-of-the-art graph convolutions on SUN RGB-D dataset [100].

3.4.1.4 Study of the influence of the residual connections on state-of-the-art graph convolutions

This section proves that our extension of residual connections to graph convolution can be used on different state-of-the-art layers. As shown in Table 3.6, using residual connections helps improve each layer's performance, where RAGC outperforms by 1.9% of the mean accuracy, the closest state-of-the-art alternative.

Method	Mean Acc. (%)
RAGC	**42.4**
REdgeConv	40.5
RFeaStConv	39.9
RGAT	39.1
RGCN	37.8

Table 3.6: Performance comparison of state-of-the-art graph convolutions with residual connections on SUN RGB-D dataset [100].

3.4.2 Comparison with state-of-the-art

3.4.2.1 Results on SUN RGB-D dataset

In this section, the proposed method is compared against the previous state-of-the-art using the SUN RGB-D dataset. As proposed by LM-CNN [11], we use a combination of different methods as a baseline. We train from scratch an AlexNet [55] and a VGG [98] using the Places [127] dataset. Then both networks are fine-tuned using the SUN RGB-D dataset using just the HHA codification of the depth channel as input. Table 3.7 shows that our method outperforms previous state-of-the-art methods with an increment of 7.8% of the mean accuracy.

Method	**Mean Acc.(%)**
CNN-RNN [99]	26.1
Places-Alexnet [55, 127]	32.1
Places-VGG [98, 127]	34.7
LM-CNN [11]	34.6
RAGC	**42.4**

Table 3.7: Performance comparison with state-of-the-art methods on SUN RGB-D dataset [100].

3.4.2.2 Results on NYU-Depth-V1 dataset

In this section, the proposed method is compared against the previous state-of-the-art, using the NYU-Depth-V1 dataset. As proposed by LM-CNN [11], we use a combination of different methods as a baseline. We train from scratch an AlexNet [55] and a VGG [98] using the Places [127] dataset. Then both networks are fine-tuned using the SUN RGB-D dataset using just the HHA codification of the depth channel as input. In addition, the Hybrid-CNN uses a similar approach, but first is pre-trained using ImageNet [18], then it is fine-tuned with Places [127] and finally fine-tuned with the NYU-Depth-V1 dataset. Table 3.8 shows that our method outperforms previous state-of-the-art methods with an increment of 7.5% of the mean accuracy.

Method	**Mean Acc.(%)**
CNN-RNN [99]	65.2
RICA [57]	64.7
Places-VGG [98, 127]	66.9
Hybrid-VGG [18, 98, 127]	68.2
LM-CNN [11]	67.8
RAGC	**75.3**

Table 3.8: Performance comparison with state-of-the-art methods on NYU-Depth-V1 dataset [97].

3.5 Conclusions

In this chapter, we presented three components that leverage the state-of-the-art in graph analysis, which effectiveness has been proved in Section 3.4.1, where we tackled the 3D geometric scene classification to prove the validity of our method. The first component is the Attention Graph Convolution layer, which outperforms previous graph convolutional layers, learning to infer a set of weights used to generate the messages of each node of the neighbourhood. The second component is the extension of residual connections to the Attention Graph Convolution layer, which effectively improves the performance of the layer in deep networks. The third component is the Voxel Pooling layer which allows mimicking the traditional CNNs architectures and learning richer, higher-level features. Our proposed method outperforms the previous state-of-the-art, proving that using graph structures to analyze 3D point clouds is effective, as shown in Section 3.4.2. The current method does not exploit the features of the RGB data which could be considered a limitation. In the next chapter, we will explore how we can introduce the RGB data in our architecture. Furthermore, we will study how we can enhance the Attention Graph Convolution using the attributes of the edges to better capture the difference between similar neighbourhoods using the features of the nodes. In addition, we will explore different spaces to generate the graph instead of just using the euclidean one. Finally, we will propose an extension of the current Voxel Pooling layer to reduce the effects of the outliers.

chapter 4

2D-3D Fusion Network using Multi-Neighbourhood Graph Convolutional Network

A great number of important real-world data come together with captures from a multitude of sensors capable of capturing information on different domains: colour, heat, depth, etc. Furthermore, the different input modalities come with different structural biases. Appropriately exploiting these multi-modal data remains unresolved, especially when the corresponding training set sizes are small.

This chapter introduces a two-branch neural network [108] capable of fusing data from 2D and 3D sensors. The architecture is composed of a 3D branch that uses a Graph Convolutional Neural Network that processes the 3D input and a 2D branch that uses a Convolutional Neural Network that processes the 2D input. The proposed architecture revisits the design of the Attention Graph Convolution and extends this operation to work in a multi-neighbourhood fashion. Furthermore, the Nearest Voxel Pooling layer is proposed, a more robust pooling layer for noisy 3D data. Finally, this architecture presents a novel 2D-3D Fusion block capable of fusing multi-modal features with different sensor resolutions.

To prove the validity of our method, we will use 3D point clouds and 2D colour images of different indoor scenes provided by public datasets.

The main contributions of this chapter are:

- The Multi-Neighbourhood Graph Convolution, which allows learning richer representations exploiting information from multiple neighbourhoods.

- The improved version of the AGC, which incorporates the node feature offset as a new edge's attribute,

allowing to learn richer local representations and better capture the neighbourhood structure.

- The Nearest Voxel Pooling layer, which is an improved version of the Voxel Pooling layer that mitigates the effects of outliers.

- The 2D-3D Fusion block, which allows the fusion of multi-modal features from 2D and 3D worlds.

4.1 Related Work

4.1.1 2D-3D fusion networks

The fusion of the features obtained by 2D-3D networks is widely used in multi-view scenarios, where it is possible to obtain different 2D RGB images from a dense point cloud. FusionNet [42] proposed to do a late fusion using the classification scores obtained by the final fully-connected layer of the 2D and 3D networks. SPLATNet [105] presented a different approach where the 2D and 3D features representations are mapped onto the same lattice. This mapping is achieved using an improved version of the Bilateral Convolution Layer [47]. More recently, an extension of PointNet++ [85] for multi-view scenarios with an early fusion strategy was proposed in MVPNet [48]. This strategy consists of concatenating the features learned on a 2D CNN to the geometric point used as input of the PointNet++. The main drawback of this approach is that each point of the point cloud used as input on PointNet++ must have a 2D feature associated with it.

These previous methods of 2D-3D fusion made use of multi-view approaches to obtain different 2D RGB images from a single dense point cloud. In the method proposed in this dissertation, the fusion is done using only one 2D RGB image from each point cloud, and it will not assume that each point of the point cloud will have an RGB feature associated, which makes it possible to work with different sensor resolutions.

4.1.2 Multi-modal scene classification

To address the Multi-modal scene classification, *Zhu* et al. [128] proposed to train a two-branch network to learn features from RGB and depth and then fuse these features using a Support Vector Machine (SVM). *Song* et al. [101] proposed to learn a more effective depth representation using a two-step training approach that directly learns effective depth-specific features using weak supervision via patches. *Li* et al. [65] proposed a novel discriminative fusion network that can learn each modality's correlative and distinctive features. *Cai* et al. [11] proposed a multi-modal CNN that captures local structures from the RGB-D scene images and learns a fusion strategy. Similarly, MAPNet [66] presented two attentive pooling blocks to aggregate semantic cues within and between feature modalities. *Song* et al. [102] proposed to use object-to-object relations obtained with detection techniques. More recently, TRecgNet [20] tackled the RGB-D scene recognition problem as a combination of a translation and a classification problem. Their work proposes to train simultaneously a classifier network that classifies the scene and a translation network that predicts the depth from RGB and the RGB from the depth. Training the network in a multitask manner helps the network learn more generic features that yield an increment in performance. However, these methods use a 2D CNN on depth maps to obtain geometric information that introduces possible errors due to missing local geometric context that the projection to a 2D world can produce. In this dissertation, we address this limitation using a Graph

Convolutional Neural Network capable of exploiting the intrinsic geometric context inside a 3D space using as input 3D point clouds obtained from RGB-D captures. Furthermore, the geometric feature extraction performance is increased using the proposed Multi-neighbourhood Graph Convolution. This convolution fuses two different neighbourhoods, one in the *euclidean* space and the other in the *feature* space, which helps improve the quality of the extracted features.

4.2 2D-3D Geometric Fusion Network with Multi-Neighbourhood Graph Convolution

This section presents our 2D-3D Geometric Fusion Network architecture and the Multi-Neighbourhood Graph Convolution. The architecture is illustrated in Figure 4.1, which is composed of two branches: the 3D Geometric branch and the 2D Texture branch. The input of the 3D Geometric branch is a 3D point cloud that can be obtained directly from a lidar sensor or using the depth information and the intrinsic camera parameters of an RGB-D sensor. Each node of the 3D input point cloud encodes the depth information using the HHA encoding [35]. The input of the 2D Texture branch is a 2D RGB image corresponding to the same capture as the one used on the 3D Geometric branch. After the corresponding branches, the extracted 3D Geometric and 2D Texture features are fused using the 2D–3D Fusion stage and the result of this stage is used by the classification network to predict the corresponding scene class.

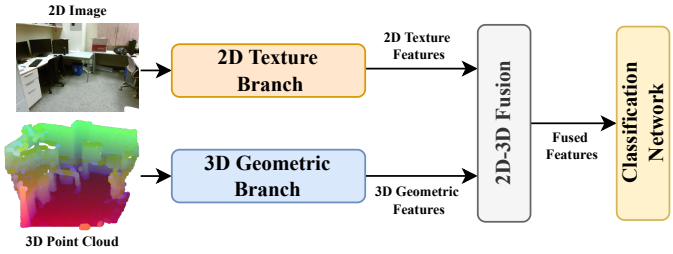


Figure 4.1: 2D-3D Geometric Fusion Network architecture.

The 2D Texture branch uses, as a backbone, the well-known architecture *ResNet-18* [41], which is depicted in Figure 4.2. *ResNet-18* is composed of a combination of residual blocks, convolutional layers, and poolings. The output of the last residual block corresponds to the *2D Texture features* used for the *2D-3D Fusion block*. This branch aims to exploit the power of already proven CNNs to obtain texture information that is aggregated to the geometric information obtained by the 3D Geometric branch.

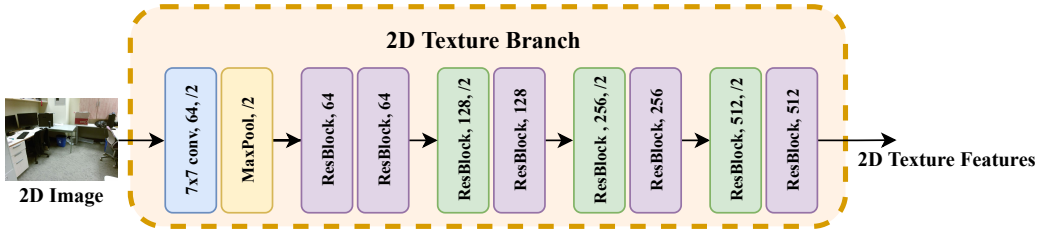


Figure 4.2: 2D Texture branch architecture, where /2 means that the stride has a value of 2. Residual Blocks are composed of two convolution layers with a kernel size of 3×3.

The 3D Geometric branch, depicted in Figure 4.3, is composed of two layers: the *Multi-Neighbourhood Graph Convolution (MUNEGC)* and Nearest *Voxel Pooling (NVP)*, both layers are explained in detail in Sections 4.2.2 and 4.2.3, respectively. The *3D Geometric branch aims* to have the same number of pooling stages as *ResNet-18*.

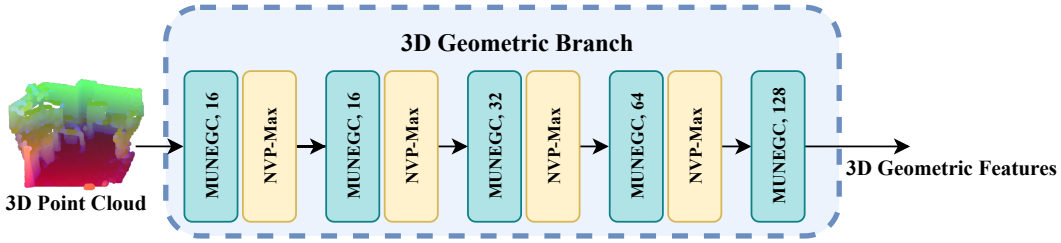


Figure 4.3: 3D Geometric branch architecture.

The 2D–3D Fusion block takes the features generated by the previously 2D and 3D branches and fuses them. Notice that the output resolution and sampling of both branches are different. The reason is that pooling layers of both branches work on different spaces (2D and 3D). As a result, even if 3D point clouds are extracted from RGB-D sensors, the final number of points and their positions are different. The proposed 2D–3D Fusion stage can handle that behaviour and generate a new set of combined features. This stage is explained in detail in Section 4.2.4. The new features are fed to the *classification network*. As depicted in Figure 4.4, the classification architecture is composed of a global average pooling and an $FC(n)$ layer, where FC is a *fully-connected* layer, and n is the number of classes.

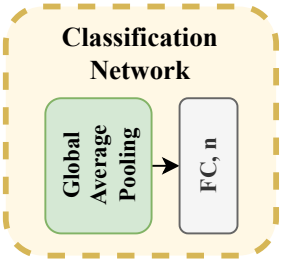


Figure 4.4: Classification network architecture.

4.2.1 Revisiting AGC

The Attention Graph Convolution (AGC) layer has been explained in detail in Section 3.2.1. As a reminder, AGC is a graph convolution that performs the convolution over local graph neighbourhoods exploiting the edges and their attributes, E. The attributes of the edges are used to estimate the weights, $\Theta_{i,j}$, which are used to generate the message of each node, j, of the neighbourhood, $N(i)$, as it was formalized in Equations 3.1 and 3.2.

The attributes of the edge proposed in the AGC were the positional offset, $\vec{s}^{\,t}_{i,j} = \vec{p}_j - \vec{p}_i$, where $\vec{p}$ is the 3D position of the nodes i and j. The attributes were defined during the graph creation step and updated just after a pooling layer was applied (positional offset can be represented in cartesian or spherical coordinates). In this section, we extend the attributes of the edges previously defined, adding a new set of attributes, the node feature offsets. For each edge, $i \leftarrow j$, we compute the offset between the node feature vectors, $\vec{k}^{\,t}_{i,j} = \vec{x}^{\,t}_j - \vec{x}^{\,t}_i$.

The edge's attribute vector is defined as the concatenation of the positional offset and the node feature offset, $\vec{e}^{t}_{i,j} = s^{t}_{i,j} || k^{t}_{i,j}$. The new attributes must be recomputed after each AGC layer since $\vec{k}^{t}_{i,j}$ depends on the previous layer's output. Adding the node features offset into the edge's attribute helps to predict richer features and better capture the difference between similar neighbourhoods.

Furthermore, to prevent the prediction of large weights, we propose to use a *tanh* activation layer in the output of the MLP that generates the weights $\Theta_{i,j}$, as it is formalized in Equation 4.1.

$$\Theta_{i,j} = tanh(MLP(\vec{e}^{t}_{i,j})) \tag{4.1}$$

4.2.2 Multi-Neighbourhood Graph Convolution

Multi-Neighbourhood Graph Convolution (MUNEGC) is a graph operation that estimates the new feature of each node using the combination of the features obtained in two different neighbourhoods, the *euclidean neighbourhood* and the *feature neighbourhood*.

The *euclidean neighbourhood* uses the position, P, of the nodes on the euclidean space to define which nodes are connected. Whereas the *feature neighbourhood* uses each node's feature vector, X, to connect the nodes. Therefore, in the *feature neighbourhood*, nodes with similar features (rather than closeness in space) are connected. Both neighbourhoods share the same original nodes and their features. For both neighbourhoods, edges can be generated following either a *kNN* policy or *radius proximity* policy. In the case of the *euclidean neighbourhood*, the *radius proximity* policy has a geometric meaning and is intuitive to choose. Whereas in the *feature neighbourhood*, the meaning of this radius is unclear and is not recommended to use due to the complexity of its selection as the feature space changes at each iteration of the training phase. For that reason, the *kNN* policy is chosen in the case of the *feature neighbourhood*. Figure 4.5 shows an example of both neighbourhoods used on MUNEGC, where the same input nodes are used to generate the *euclidean neighbourhood* and the *feature neighbourhood*. For simplicity, both neighbourhoods follow a *kNN* policy. On the *euclidean neighbourhood*, nodes 0, 1, 2 and 3 are selected as neighbours of node 0 as they are the closest nodes on the 3D space. Whereas, in the *feature neighbourhood*, nodes 0, 3, 4 and 5 are selected as neighbours of node 0 as the features of the nodes are similar, represented as nodes with the same colour. Once both neighbourhoods are defined, the attributes of the edges are computed following the method explained in Section 4.2.1.

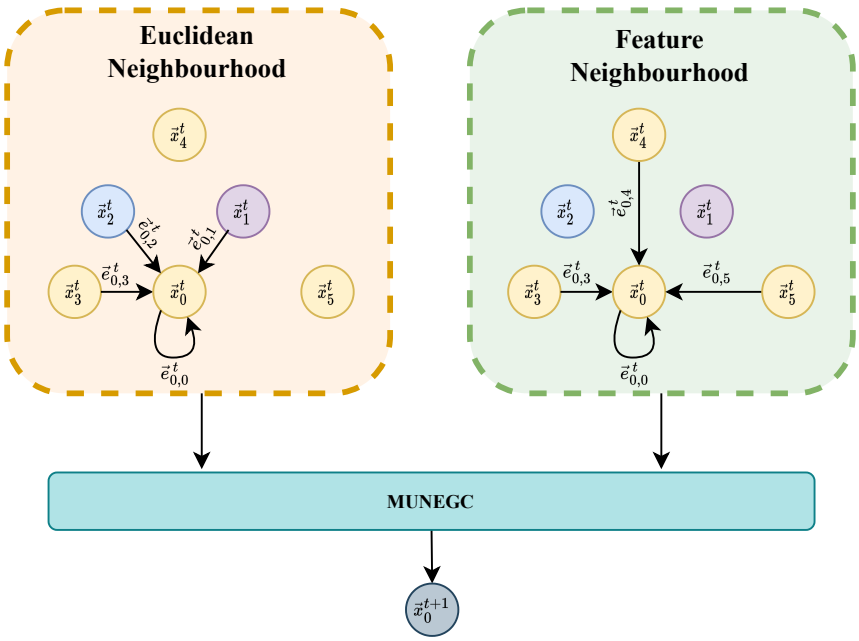


Figure 4.5: Multi-Neighbourhood Graph Convolution, where $\vec{x}_i$ is the node feature vector i, $\vec{e}^{\,t}_{i,j}$ is the edge's attribute vector of the edge $i \leftarrow j$. Nodes with similar features are coloured with the same colour. Both neighbourhoods are created using a *kNN policy* where $k = 4$.

As a result of having two neighbourhoods, each node has two possible feature vectors, one for the *euclidean neighbourhood* and another for the *feature neighbourhood*. By definition, the filter size used in both neighbourhoods is the same. This means that if a MUNEGC of d output features is requested, the filters of both neighbourhoods output d features for each one. These features are going to be combined using an aggregator which must output d features, as formalized in Equation 4.2, where the sub-index e indicates that the neighbourhood belongs to the *euclidean neighbourhood*, whereas the index f indicates that it belongs to the *feature neighbourhood*. The computation of the weights, Θ, has been defined previously in Equation 4.1.

$$\vec{x}_i^{\,t+1} = Aggr\left\{ \frac{1}{deg(N_e(i))} \sum_{j \in N_e(i)} \Theta^e_{i,j} \vec{x}_j^{\,t}, \quad \frac{1}{deg(N_f(i))} \sum_{j \in N_f(i)} \Theta^f_{i,j} \vec{x}_j^{\,t} \right\} \tag{4.2}$$

Using two different neighbourhoods helps to learn more robust node features that consider the characteristics of the regions with similar properties and the regions close in the 3D space.

4.2.3 Nearest Voxel Pooling

The Nearest Voxel Pooling (NVP) layer is based on the previously defined VP layer in Section 3.2.3. The VP layer creates a grid of voxels of size $\vec{s}$ over the input point cloud and replaces all points inside the voxel with their centroid. However, the VP layer can introduce noise when the points of two different voxels are closer than their respective voxel's centroid. As shown in Figure 4.6, NVP solves that issue by grouping the points in respect of how far they are from all the centroids of the voxel grid.

(a) Voxel Pooling. **(b)** Nearest Voxel Pooling

Figure 4.6: Comparison of (a) Voxel Pooling and (b) Nearest Voxel Pooling. Crosses represent the new centroids, and the dots the original points. Each colour indicates the points used to calculate each centroid.

As done in the VP layer, the first step is to convert the graph to a 3D *point cloud*, which is a straightforward operation that removes the edges and keeps the nodes as 3D points in the point cloud. Each 3D point contains the 3D coordinates, $\vec{p}_i$, and the feature vector, $\vec{x}_i$. Afterwards, we apply the NVP layer described in Algorithm 2, which consists of assigning each point in the point cloud to the voxel with the closest centroid. Then, the voxels with no points are removed, whereas the voxels with points assigned re-compute their centroid as the average of all the points assigned to this voxel. The centroid's feature is the aggregation of the points assigned to the correspondent voxel. Finally, the downsampled version of the point cloud, composed of all the voxel's centroids, is converted into a graph where the edges and their attributes are re-generated.

Algorithm 2: Nearest Voxel Pooling

Input: $\vec{vs} \leftarrow$ voxel's size, $P \leftarrow$ point cloud
Output: $C \leftarrow$ centroid's position, $F \leftarrow$ centroid's feature

Create voxels, V, of size $\vec{vs}$, over the point cloud P
foreach V_i *in* V **do**
  Compute centroid $\vec{c}_i \leftarrow$ average of points' positions inside V_i
end foreach
foreach $\vec{p}_i$ *in* P **do**
  Assign $\vec{p}_i$ to the closest centroid $\vec{c}_i$
end foreach
foreach $\vec{c}_i$ *in* C **do**
  if $\vec{c}_i$ *does not have points assigned* **then**
    Delete $\vec{c}_i$
  else
    Compute centroid's feature $\vec{f}_i \leftarrow$ aggregation of points' features in $\vec{c}_i$
    Re-compute centroid's position $\vec{s}_i \leftarrow$ average of points' positions in $\vec{c}_i$
  end if
end foreach

4.2.4 2D-3D Fusion block

The *2D-3D Fusion block* is in charge of fusing different sets of multi-modal features. In this work, we use this block to fuse the set of *3D Geometric and 2D Texture features* which architecture is shown in Figure 4.7.

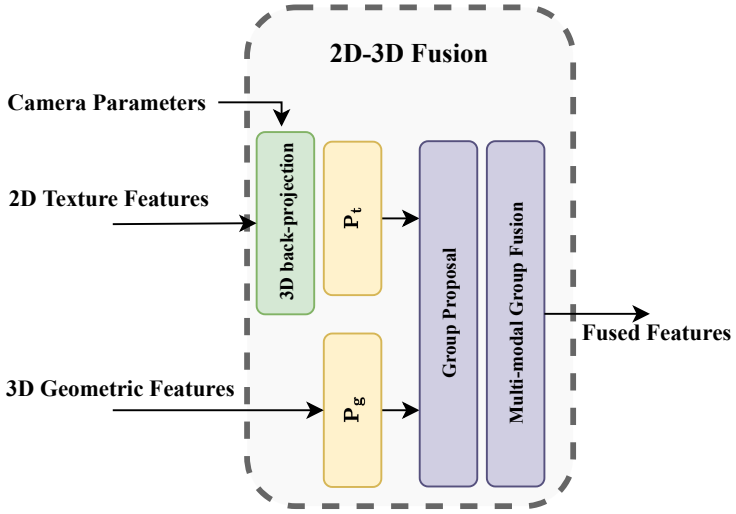


Figure 4.7: 2D–3D fusion block architecture where P_g is the projection function of the 3D Geometric features, and P_t is the projection function of the 2D Texture features.

The first step done by the *2D-3D Fusion block* is to back-project the *2D Texture features* into the 3D space using the camera parameters and the depth channel. Afterwards, a *projection function*, P, is applied to the *3D Geometric and 2D Texture features* that projects both sets of features into a common feature space with the same dimensionality. Each set of features has its own *projection function*, P_g for the *3D Geometric features* and P_t for the *2D Texture features*, which is implemented as a convolution of kernel 1×1 without bias. Then, both sets of projected features are grouped on the *group proposal* step, which is based on the NVP layer and creates groups of points from the *2D Texture* and *3D Geometric features*. For each group of points, a super-point is generated, which position is the average of the positions of the points inside the same group, as depicted in Figure 4.8.

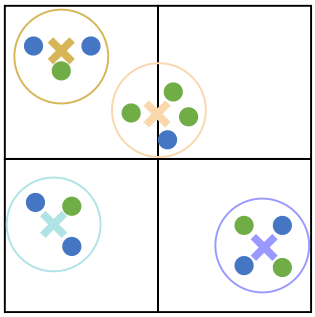


Figure 4.8: Example of the group's creation for the 2D–3D fusion stage. Each dot colour represents a different feature (blue for 2D Texture and green for 3D Geometric), each circle represents a different group, and the crosses represent the super-point of each group.

The feature of each super-point is computed following these two steps. First, we compute the average of the same kind of features (*2D Texture or 3D Geometric feature*) inside the same group. Secondly, we concatenate

the resulting average to generate the fused feature for each super-point. Note that if just one kind of feature appears in the group, we assume that the feature vector for the missing feature group is a vector of 1s. Algorithm 3 formalizes the algorithm used on this block.

Algorithm 3: 2D-3D Fusion

Input: $\vec{vs} \leftarrow$ voxel's size, $CP \leftarrow$ camera parameters and depth,
$G \leftarrow$ 3D Geometric features, $T \leftarrow$ 2D Texture features
Output: $S \leftarrow$ super-point's position, $F \leftarrow$ super-point's feature

Back-project T into a 3D point cloud using CP
Apply P_t to T
Apply P_g to G
Concatenate both point clouds $P \leftarrow (T||G)$
Create voxels, V, of size $\vec{vs}$, over the point cloud P
foreach V_i *in* V **do**
  Compute centroid $\vec{c}_i \leftarrow$ average of points' positions inside V_i
end foreach
foreach $\vec{p}_i$ *in* P **do**
  Assign $\vec{p}_i$ to the closest centroid $\vec{c}_i$
end foreach
foreach $\vec{c}_i$ *in* C **do**
  if $\vec{c}_i$ *does not have points assigned* **then**
    Delete $\vec{c}_i$
  else
    Compute superpoint's texture feature $\vec{ct} \leftarrow$ average of points' texture features in $\vec{c}_i$
    Compute superpoint's geometric feature $\vec{cg} \leftarrow$ average of points' geometric features in $\vec{c}_i$
    Compute superpoint's feature $\vec{f}_i \leftarrow (\vec{ct}||\vec{cg})$
    Compute superpoint's position $\vec{s}_i \leftarrow$ average of points' position in $\vec{c}_i$
  end if
end foreach

4.3 Experimental Setup

4.3.1 Datasets and Metrics

The SUN RGB-D dataset [100] was captured from different RGB-D sensors, including Asus Xtion, RealSense, Kinect v1 and Kinect v2. Following the settings proposed by the authors, classes with less than 80 samples are discarded, resulting in 9504 captures with 19 different classes. These captures are divided into 4845 for training and 4659 for testing using the split provided by the authors. This dataset is the one used for the ablation studies.

The NYU-Depth-V2 dataset (NYUV2) [78], following the configuration proposed by the authors, the categories are grouped into 10, including 9 most common categories and the *Other* category representing the rest. The dataset is split, following the division provided by the authors, using 795 captures for training and 654 for testing.

The proposed method is evaluated on both datasets using the *mean accuracy* metric, which is defined in previous Section 3.3.1 and formalized in Equation 3.4.

4.3.2 Implementation details

4.3.2.1 Pre-processing input data

The input data used in our model comes from RGB-D captures. The RGB images are used as input by the *2D Texture branch*, which are cropped using the center-crop technique using a size of 560×420. In addition, a random horizontal flip is applied during training as a data augmentation technique. The *3D Geometric branch* uses as input a graph based on the depth capture. The depth capture is encoded using the HHA encoding [35], downsampled by a factor of 8 and back-projected into a 3D point cloud. The 3D point cloud positions are scaled between $[-1, 1]$, and during training, the following online data augmentation techniques are applied:

1. Rotation over the vertical axis randomly between $(0, 2\pi)$.
2. Mirroring over horizontal axis randomly with a probability of 0.5.
3. Random removal of points with a probability of 0.2.
4. 3D random crop explained in detail in Section 3.3.2.1.

4.3.2.2 Architecture details

The proposed method is implemented with Pytorch [81] and Pytorch Geometric [25]. The *2D texture branch* is implemented using as a backbone the well-known *ResNet-18*. For the proposed 3D Geometric branch, the hyper-parameters used are:

- **MUNEGC** layer creates both neighbourhoods using the *kNN* policy. As attributes of the edge, both

neighbourhoods use the *positional offset*, represented in spherical coordinates and the *feature offset*. The MLP used to generate the weights to create the messages is composed of two MLP layers with output features $(128, d_l \times d_{l-1})$, where d_l is the number of output features of the layer l. The average aggregator is used for the combination of both neighbourhoods.

- **NVP** layer uses the maximum aggregator with cubic voxels.

The detailed architecture is shown in Table 4.1.

2D Texture Branch			
Layer	**N. Filters**	**Kernel**	**Stride**
Conv	64	7×7	1
MaxPool	-	3×3	2
ResBlock	64	3×3	1
ResBlock	64	3×3	1
ResBlock	128	3×3	2
ResBlock	128	3×3	-
ResBlock	256	3×3	2
ResBlock	256	3×3	-
ResBlock	512	3×3	2
ResBlock	512	3×3	-
3D Geometric Branch			
Layer	**N. Filters**	**Cubic Voxel Size (meters)**	
MUNEGC	16	-	
VP-Max	-	0.05	
MUNEGC	16	-	
VP-Max	-	0.08	
MUNEGC	32	-	
VP-Max	-	0.12	
MUNEGC	64	-	
VP-Max	-	0.24	
MUNEGC	128	-	
2D-3D Fusion Stage			
Layer	**N. Filters**	**Cubic Voxel Size (meters)**	**P. Filters**
2D-3D Fusion Block	512	0.24	256
Global Average Pooling	-	-	-
FC	n	-	-

Table 4.1: 2D-3D Geometric Fusion Network architecture details. P. filters define the number of filters used on the Projection functions used to project the geometric and texture features. n is the number of classes, and FC is a fully-connected layer. After each convolution and MUNEGC layer, we have batch normalization and ReLU activation layers. The Global Average Pooling layer computes a global descriptor computing the average of all node features.

4.3.2.3 Training details

Due to GPU memory constraints, each branch of the proposed network is trained in an isolated manner adding an independent *classification network* for each branch. Following the same setup that we used in Chapter 3, the *classification network* is initialized using the Xavier initialization [31], where biases are initialized as $b = -log((1-\phi)/\phi)$, where $\phi = 1/k$ and k is the number of classes, and the loss used is the weighted cross-entropy formalized in Equation 3.5 which has been explained in Section 3.3.2.

2D Texture branch: Similar to previous works [20, 66], when training the branch in the *SUN RGB-D* datasets, weights are initialized using a pre-trained version of the network on the Places dataset [127]. For the smaller *NYUV2 dataset*, the branch is initialized using the weights obtained on the training done in the *SUN RGB-D* dataset. In both datasets, the branch is trained using the early-stopping criteria with a patience of 10 epochs and a maximum of 100 epochs with a batch size of 16. The optimizer used for this training is *SGD with momentum* [86]. The learning rate is 1×10^{-3} with a momentum of 0.9 and a weight decay of 1×10^{-4}.

3D Geometric branch: This branch is initialized using the Xavier initialization [31] for the *SUN RGB-D dataset* as there is no bigger RGB-D dataset to perform a pre-training of the network. In the case of the *NYUV2* dataset, as done in the *2D Texture branch*, weights obtained on *SUN RGB-D* are used to initialize the branch. The motivation for this is to demonstrate the ability of the *3D Geometric branch* to learn generalized representations that can be used on other datasets. In both datasets, the network is trained using the early-stopping criteria with a patience of 20 epochs and a maximum of 200 epochs with a batch size of 32. The optimizer used for this training is the Rectified Adam (RADAM) [67], an improved version of ADAM [52] that rectifies the variance of the adaptive learning rate. The learning rate used is 1×10^{-3}, betas $(0.9, 0.999)$ and a weight decay of 1×10^{-4}. A dropout layer is added before the MLP layer of the *classification network* with a probability $p = 0.2$ to be zeroed.

2D–3D Fusion stage: The *2D–3D fusion block* and the *classification network* are considered the last branch and are trained together. In both datasets, weights are initialized using the Xavier initialization [31]. The input of this branch is the camera parameters and the output features of both previous branches, without their corresponding classification networks, as can be seen in Figure 4.1. The network is trained using the early-stopping criteria with a patience of 5 epochs and a maximum of 20 with a batch size of 32. The Rectified Adam (RADAM) [67] optimizer is used with a learning rate of 1×10^{-3}, betas $(0.9, 0.999)$ and a weight decay of 1×10^{-4}. A dropout layer is added before the MLP layer of the *classification network* with a probability $p = 0.5$ to be zeroed.

Moreover, by carrying extensive hyperparameter tuning, we ensure that differences in performance can be attributed to modelling choices rather than incomplete hyperparameter optimization. We used the Hyperband [62] algorithm over the validation split, which is created taking 20% of the samples of the training split. Specifically, we run the hyperparameter optimization over the SUN RGB-D dataset. Once we have selected the hyperparameters, we re-train the network using the complete training split and evaluate the model on the test split.

4.4 Results

4.4.1 Ablation Studies

4.4.1.1 Study of the neighbourhood definition on MUNEGC

To generate the neighbourhoods needed by MUNEGC, two different policies can be used: kNN and *radius proximity* policies. However, the radius proximity policy cannot be applied in the *feature neighbourhood* as features are still being defined during the training phase, which complicates the choice of a radius. For this reason, the two policies are studied in the *euclidean neighbourhood*. In the case of the *radius proximity* policy, we need to define a radius for each MUNEGC layer. We found that the best radius for each layer are $[0.05, 0.08, 0.12, 0.24, 0.48]$, in meters, whereas for the kNN policy, we found a $k = 9$ works best. As can be observed in Table 4.2, the kNN policy surpasses the mean accuracy obtained with the best configuration of the *radius proximity* policy, which matches the behaviour observed during Chapter 3.

Policy	**Mean Acc. (%)**
kNN	**44.1**
Radius proximity	42.4

Table 4.2: Analysis of kNN and radius proximity as neighbourhood generation policies for the euclidean neighbourhood of MUNEGC on SUN RGB-D dataset [100].

Once both graphs needed by MUNEGC are defined, we need to define an attribute for each edge. This section explores the combination of different edge attributes defined previously in Section 4.2.1. We combine the positional offset with the node feature offsets. The positional offset is represented in spherical coordinates, which outperforms euclidean coordinates as shown in Chapter 3. The node feature offset is represented directly using the offsets and using the L_2 norm of the offsets. Table 4.3 shows the results of combining different kinds of edge attributes, proving that the best results are achieved using, as attributes of the edge, the spherical offset and the node feature offset in both neighbourhoods.

Euclidean Neighbourhood	**Feature Neighbourhood**	**Mean Acc. (%)**
Spherical offset + Feature offset	**Spherical offset + Feature offset**	**44.1**
Spherical offset + L_2 Feature offset	Spherical + L_2 Feature offset	40.4
Spherical offset	Feature offset	40.2

Table 4.3: Analysis of the effectiveness of different edge attributes on each kind of neighbourhood. L_2 offset is the L_2 distance between the feature vector of two neighbours on SUN RGB-D dataset [100].

4.4.1.2 Study of the MUNEGC design

This section analyzes the benefits of the two proposed extensions to the AGC and the improvements provided by MUNEGC. As a reminder, the two extensions presented were: the addition of the node feature offset as an edge attribute and the use of the *tanh* activation function on the MLP to predict the weights used on the convolution, which were explained in detail in Section 4.2.1. In Table 4.4, we show the improvements provided by the two proposed extensions to the AGC and their impact on the final result. In addition, we compare the result we get with the AGC versus the residual extension of it and the multi-neighbourhood extension. To guarantee that, in all cases, we have the same number of layers, we replaced each MUNEGC layer from the

architecture proposed in Section 4.2 with two AGC with the same number of output features. We observe that the residual extension still helps improve the network's performance. However, the MUNEGC extension outperforms the AGC and its residual extension.

layer	Spherical offset	Feature offset	tanh	Mean Acc.(%)
AGC	Yes	No	No	40.2
AGC	Yes	Yes	No	41.5
AGC	Yes	Yes	Yes	42.3
RAGC	Yes	Yes	Yes	43.2
MUNEGC	**Yes**	**Yes**	**Yes**	**44.1**

Table 4.4: Comparison of AGC and its residual and multi-neighbourhood extensions on SUN RGB-D dataset [100].

Furthermore, we analyze the aggregation operator used to fuse the feature we get for the *euclidean* and the *feature* neighbourhoods. As observed in Table 4.5, the average aggregator outperforms the maximum aggregator.

Method	Mean Acc.(%)
Average	**44.1**
Maximum	40.7

Table 4.5: Comparison between maximum and average aggregators to fuse the *euclidean* and *feature* neighbourhoods of MUNEGC on SUN RGB-D dataset [100].

4.4.1.3 Study of the MUNEGC design on different state-of-the-art graph convolutions

This section proves that our multi-neighbourhood approach can be used on different state-of-the-art layers. As shown in Table 4.6, the multi-neighbourhood approach helps to improve each layer's performance, where AGC using the MUNEGC approach outperforms the closest state-of-the-art alternative by 2.8% of the mean accuracy.

	Mean Acc. (%)	
Method	**Vanilla**	**MUNEGC**
AGC	42.3	**44.1**
EdgeConv	38.6	41.3
FeaStConv	38.3	40.7
GAT	37.5	40.1
GCN	36.2	39.3

Table 4.6: Performance comparison of state-of-the-art graph convolutions with the multi-neighbourhood approach on SUN RGB-D dataset [100].

4.4.1.4 Comparison between Nearest Voxel Pooling and Voxel Pooling layers

The NVP layer is an improved version of the VP that solves the drawback of VP when the points inside of two different voxels are closer than their respective voxel's centroid. NVP layers are replaced with VP layers to analyse the performance of the proposed NVP in the 3D Geometric branch. Table 4.7 shows that the NVP outperforms VP.

Layer	**Mean Acc.(%)**
NVP	**44.1**
VP	42.5

Table 4.7: Comparison between Nearest Voxel Pooling and Voxel Pooling layers on SUN RGB-D dataset [100].

4.4.1.5 Study of the fusion block strategy

Using a geometric proximity approach, the proposed fusion blocks fuse the features extracted from the *2D Texture* and *3D Geometric* branches. The 2D Texture features are projected into the 3D world using the depth and camera parameters. Then, they are grouped using the algorithm described in Section 4.2.4, which is based on the Nearest Voxel Pooling layer. To prove the validity of the 2D-3D Fusion block, we compare the method with a late fusion strategy inspired by FuseNet [42], which concatenates the features obtained from each branch after a global average pooling. Therefore, only a concatenation is used without considering any geometric proximity. Furthermore, we compare the group creation strategy using the NVP and VP to propose groups of points to fuse. In Table 4.8, we observe that the proposed fusion block outperforms the late fusion and the VP alternatives.

Method	**Mean Acc.(%)**
2D-3D Fusion block with NVP	**58.6**
2D-3D Fusion block with VP	57.8
Late fusion	57.2

Table 4.8: Comparison of different strategies for the 2D-3D Fusion block on SUN RGB-D dataset [100].

4.4.1.6 Two-branch approach vs. an RGB Graph Convolution Neural Network

Another way to use the RGB values would be to add them directly on the nodes. In this section, we compare the results that we get using the 2D Texture branch, which uses as backbone a ResNet-18, and the results that we could get using the RGB information directly on the 3D Graph Convolutional Network. As shown in Table 4.9, our RGB 3D Graph Convolution outperforms ResNet-18 when it is trained from scratch; however, when ResNet-18 is pre-trained on the Places dataset and fine-tuned on the SUN-RGB dataset, it outperforms our proposed RGB 3D Graph Convolution, since the 3D datasets are orders of magnitude smaller than the 2D ones. Using a two-branch approach allows us to exploit the benefits of using pre-trained 2D CNNs on bigger datasets and fine-tune them for the specific problem.

Method	Initialization	Mean Acc.(%)
ResNet-18	**Places**	**56.4**
ResNet-18	random	47.8
RGB 3D Graph Convolution	random	50.1

Table 4.9: Comparison of the RGB version of the 3D Graph Convolution vs. ResNet-18 on SUN RGB-D dataset [100].

4.4.2 Comparison with state-of-the-art

4.4.2.1 Results on SUN RGB-D dataset

In this section, the proposed method is compared against the previous state-of-the-art using the SUN RGB-D dataset. Table 4.10 shows that our method outperforms previous state-of-the-art methods with an increment of 1.9% of the mean accuracy.

	Mean Acc.(%)		
Method	**RGB**	**Geometric**	**Fusion**
Multi-modal fusion [128]	40.4	36.5	41.5
Effective RGB-D representations [101]	44.6	42.7	53.8
DF^{2}Net [65]	46.3	39.2	54.6
MAPNet [66]	-	-	56.2
TRecNet [20]	50.6	47.9	56.7
Ours	56.4	44.1	**58.6**

Table 4.10: Performance comparison with state-of-the-art methods on SUN RGB-D dataset [100].

However, if we look in detail at Table 4.10, we observe that our geometric branch is outperformed by the geometric analysis done by TRecNet [20]. The reason for that is that the geometric branch of our method is trained from scratch due to the lack of large RGB-D or 3D datasets. In contrast, since TRecNet uses 2D networks to process the depth channel, they are capable of pre-training their geometric analysis branch using a huge RGB dataset, such as Places [127]. In Table 4.11, we analyze the geometric branch of our method and the geometric branch from TRecNet. We can observe that using pre-trained features provide TRecNet with an improvement of up to 5.4%. However, if we train both methods from scratch, our method outperforms them by a 2% of improvement in the mean accuracy.

Method	Initialization	Mean Acc.(%)
TRecNet [20]	**Places**	**47.6**
TRecNet [20]	Random	42.2
Ours	Random	44.1

Table 4.11: Performance comparison of the 3D Geometric branch with state-of-the-art methods on SUN RGB-D dataset [100].

4.4.2.2 Results on NYU-depth-V2 dataset

In this section, the proposed method is compared against the previous state-of-the-art using the NYUV2 dataset. For this dataset, the 3D Geometric branch is pre-trained using the weights from the training done on SUN RGB-D dataset. Table 4.12 shows that overall our method overcomes state-of-the-art by 6% of the mean accuracy.

	Mean Acc.(%)		
Method	**RGB**	**Geometric**	**Fusion**
Effective RGB-D representations [101]	53.4	56.4	67.5
DF^{2}Net [65]	61.1	54.8	65.4
MAPNet [66]	-	-	67.7
TRecNet [20]	64.8	57.7	69.2
Ours	67.8	59.2	**75.1**

Table 4.12: Performance comparison with state-of-the-art methods on NYU-Depth-V2 dataset [100].

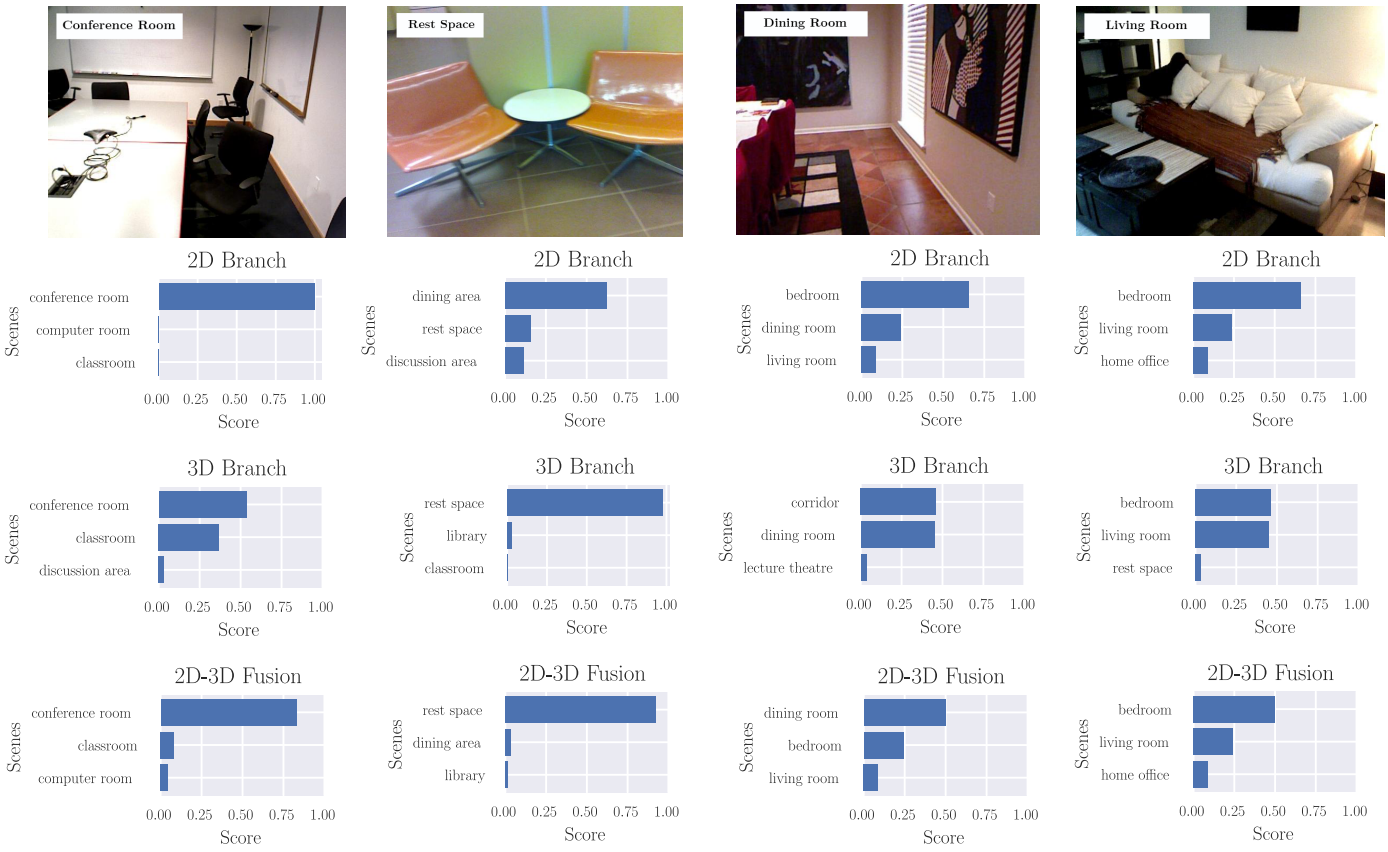
These results reveal that the proposed *3D Geometric* branch has the ability to learn generalized representations that can be used on other datasets, making it possible to apply transfer learning techniques as conventional 2D-CNNs. In Table 4.13, we compare the geometric branch of our method and the geometric branch from TRecNet. In this case, our method trained from scratch outperforms TrecNet when it is initialized with Places. Furthermore, when both methods are initialized using the SUN RGB-D dataset, our method outperforms TrecNet by 2% of improvement in the mean accuracy.

Method	**Initialization**	**Mean Acc.(%)**
TRecNet [20]	Places	55.2
TRecNet [20]	SUN RGB-D	57.7
Ours	Random	57.2
Ours	**SUN RGB-D**	**59.2**

Table 4.13: Performance comparison of the 3D Geometric branch with state-of-the-art methods on NYU-Depth-V2 dataset [78].

4.4.3 Qualitative results

In this section, we analyze the qualitative results on the SUN RGB-D dataset shown in Figure 4.9. In the first column, we observe an example of when all the stages of the proposed framework classify the scene correctly. In the second column, we have an example where the 2D branch fails to identify the scene, whereas the 3D branch is capable of it. The network can properly identify the scene when fusing both features with the proposed 2D-3D Fusion network. In the third column, we show an example of when the 2D and 3D branch independently fails to identify the scene. However, when combining the features from both branches, the 2D-3D Fusion network can properly identify the scene. Finally, in the fourth column, we have a failure example where all the stages of the networks fail to identify the scene. However, the correct scene, *Living Room*, is in the top 2 scores on each of the branches. We observe that the system is identifying the scene as a bedroom which might be induced by the presence of many cushions on the sofa, leading the network to a wrong decision.



(a) **(b)** **(c)** **(d)**

Figure 4.9: Qualitative results on SUN RGB-D dataset [100] showing the top 3 classes. Column **(a)** shows an example of when all the stages of the proposed framework classify the scene properly. Column **(b)** shows an example of when the 2D branch fails to classify the scene. Column **(c)** shows an example of when both the 2D and 3D branches fail to classify the scene, but when features of both branches are combined on the Fusion stage, the network is capable of properly classifying the scene. Column **(d)** shows a failure case when all the components of the system fail to distinguish the scene.

4.5 Conclusions

In this chapter, we presented a 2D-3D Fusion Network that combines the intrinsic geometric information of the 3D space obtained by Graph Convolutional Neural Network and the 2D Texture features obtained by a 2D CNN. Using a standard 2D CNN allows us to exploit the benefits of pre-training the network on bigger datasets and fine-tuning the network to a particular problem getting richer and more representative features. In contrast, we can not do the same in the 3D world since the 3D datasets are orders of magnitude smaller than the ones we have in 2D. To fuse the features that we get from the 2D and 3D branches, we proposed a 2D-3D Fusion block which exploits the geometric proximity and allows the fusion of multi-modal features without having a 1-to-1 correspondence between them. The Graph Convolutional Neural Network proposed to get the Geometric features is composed of three novel components that outperform the previous state-of-the-art and also improve the performance of the Graph Convolutional Neural Network presented in Chapter 3. The first one is the improved version of the Attention Graph Convolution layer that allows the addition of new edge attributes, such as the node feature offset, which learns more specific weights for each neighbourhood and outperforms previous state-of-the-art graph convolutions. The second one is the Multi-Neighbourhood Graph Convolution which allows exploiting the use of multiple neighbourhoods generated in different spaces, such as the euclidean and feature spaces, to learn better representations. Finally, we proposed the Nearest Voxel Pooling layer, an improved version of the previous Voxel Pooling layer that mitigates the influence of the outliers. Overall, our proposed method outperforms the previous state-of-the-art. As a drawback, our Attention Graph Convolution requires a lot of memory since it generates the weights used to compute the messages with an MLP that also requires its own parameters. In the next chapter, we will explore a different direction to learn features that properly describe the local neighbourhood reducing the number of parameters needed.

Part II

Node Analysis

chapter 5

Heterogeneous Graph Convolutional Neural Network using a Multi-Aggregator approach

Human beings can analyze different regions of a vision signal independently and learn when to use visual context information or add information coming from other systems, such as the hearing system, to improve the interpretation. The ability to combine heterogeneous information helps to improve the ability of human beings to better understand their surroundings.

In this chapter, we will focus on the node analysis problem, which consists of finding characteristics on a node level instead of a graph level, as we did in Chapters 3 and 4. To do so, we proposed the Multi-Aggregator Graph Convolution (MAGC) [109], a simplification of the Attention Graph Convolution that reduces the amount of memory needed without losing performance. In addition, we presented a variant of MAGC to work with a heterogeneous graph, a special kind of graph composed of different node types.

To better assess the validity of our method, we will focus on the heterogeneous node analysis problem. Specifically, we will address the well-known Computer Graphics process of skinning a character mesh. The data used in this task are synthetic characters generated by 3D artists, which are composed of a mesh and its associated skeleton.

The main contributions of this chapter are:

- The Multi-Aggregator Graph Convolution (MAGC), which extends the message-passing architecture to combine multiple aggregators to better generalize for unseen topologies and learn better local representations.

- The heterogeneous variant of MAGC, which allows working with graphs containing different node features.

- The Two-Stream Graph Neural Network, which allows extracting features from meshes and skeletons with different topologies and without relying on handcrafted features.

- The k unique joints of the closest bones skin binding algorithm, which introduces a natural relationship between the mesh and the skeleton using a joint representation instead of a bone representation which is commonly used in recent literature.

5.1 Preliminaries

5.1.1 Character creation pipeline

Creating 3D characters for video games or high-quality films is a complex and time-consuming process that artists spend years learning to do efficiently. The process of creating a character can be divided into four main stages:

- **Character designing:** Defines how the character should look like.

- **Modelling:** Designs the 3D mesh of the designed character.

- **Texturing:** Assigns the corresponding textures for each part of the 3D mesh.

- **Rigging:** Creates a rig that gives control to the animators to move the character. The rig is composed of a hierarchy of points called *joints*, connected by edges called *bones*, that mimics the human skeleton.

- **Skinning:** Attaches and defines how each joint influences each vertex of the mesh.

5.1.2 Skinning methods

Plenty of different skinning methodologies can be found in the literature, which can be split into three main categories: Geometric [2, 49, 59, 73], Physical-based [61, 68, 83, 87], and Simulation-based [56, 60]. In this thesis, we use the Geometric methodologies, widely used in video game creation due to their simplicity and efficiency when running in real-time scenarios.

The Geometric methodologies deform the position of the vertices of the mesh using only the displacement of the joints of the skeleton. The most popular one, and the one used in this thesis, is the *Linear Blend Skinning (LBS)* [2, 59, 73] algorithm, also known as vertex blending or enveloping. This algorithm assumes the following input data:

- **Binding pose mesh** composed by vertices denoted as $\vec{p}_1, ..., \vec{p}_n \in \mathbb{R}^3$, where n is the number of vertices of the mesh. During the computation of the LBS, the coordinates are assumed to follow the convention

of homogeneous coordinates, which means that $\tilde{p}_n \in \mathbb{R}^4$ with the last coordinate equal to one.

- **Binding pose skeleton** composed of joints which are normally represented as a hierarchy of rigid transformations, and bones that represent the hierarchy of the joints. A bone is composed of two joints the root joint and the child joint. The root joint is the one in charge of the movement of the bone, which influences the child's joint and its bone.

- **Joint transformations** represented as a list of transformation matrices, $T_1, ..., T_m \in \mathbb{R}^{3\times4}$, where m is the number of joints. Each transformation matrix encodes the rotation and translation of each of the skeleton's joints.

- **Skinning weights**. For each vertex p_i, a set of weights, $w_{i,1}, ..., w_{i,m} \in \mathbb{R}$, are defined. Each weight sets the amount of influence of joint j on vertex i. The weights of a vertex must add up to 1, $w_{i,1} + ... + w_{i,m} = 1$.

The binding pose of the mesh and the skeleton refers to the pose that the artist uses to create the character. The binding pose is normally a T-pose or A-pose, which defines the legs and arms position. Figure 5.1 shows an example of a mesh and skeleton in an A-pose. After applying a movement to the joints of the skeleton, the new position of the vertices of the mesh is computed as a weighted linear combination of the corresponding *Joint transformation* as described in Equation 5.1.

$$\tilde{p}_i^t = \left(\sum_{j=1}^{m} w_{i,j} T_j \right) \tilde{p}_i^{t+1} \tag{5.1}$$

In recent years, there has been an effort from the *Computer Graphics* community to develop automatic methods that predict the influence of each joint on each vertex of the mesh to speed up the process of creating 3D animated characters. Figure 5.1 shows the common pipeline used in these kinds of methods.

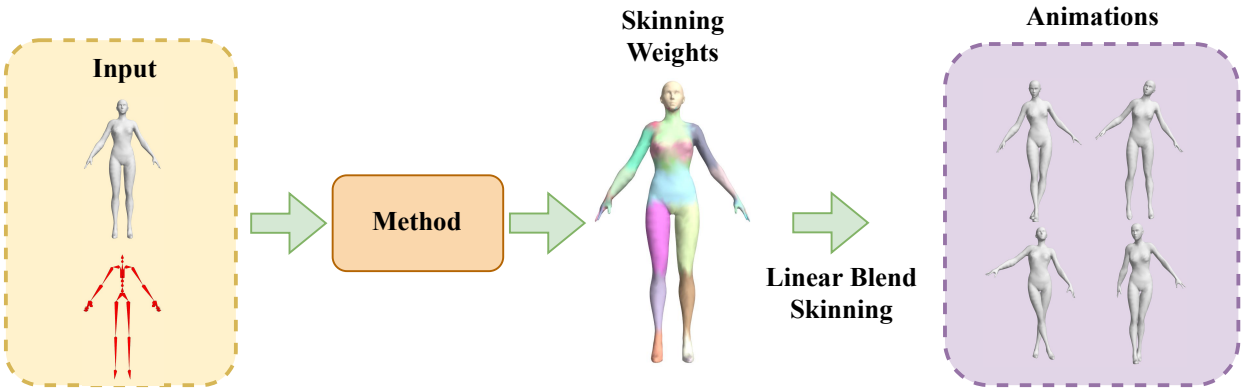


Figure 5.1: Pipeline of a common Automatic Skinning Weight Prediction Method. Asset from Paragon Collection [23].

5.2 Related Work

Automatic skinning weight prediction techniques can be grouped into two different categories: geometric based [3, 19, 46, 50, 54, 77, 96, 116] and data-driven solutions [13, 63, 69, 71, 93, 120].

Geometric methods rely on geometric characteristics between meshes and skeletons. The earliest methods to automatically generate skinning weights proposed to exploit *Heat Diffusion* [4] and *Illumination* [117] models. Alternatively, other methods used energy functions for the estimation, such as *Elastic Energy* [50] or *Laplacian Energy* [46]. Later, *Dionne* et al. [19] proposed Geodesic Voxel Binding to handle non-watertight meshes. All these methods rely on functions that assign the skinning weights depending on the distance between joints and vertices. However, this assumption does not work for high-quality game characters with complex topologies where multiple independent components can intersect.

Data-driven methods typically require multiple poses of a mesh or different meshes as input to learn how to compute the skinning weights. New methods such as [13, 63, 71, 93] estimate skinning weights from Motion Capture data. They focus on finding the skinning weights of humanoids and assume a fixed skeleton topology, making the network unable to work for characters with different skeleton topologies.

NeuroSkinning [69] is one of the earliest proposed data-driven methods to automatically compute the skinning weights for synthetic characters using neural networks. This method uses graph convolutions to compute the skinning weights of a new mesh. *NeuroSkinning* uses a bone representation, meaning that the network predicts the skinning weights per bone instead of per joint. It also relies on creating a super-skeleton that consists of the fusion of all the skeletons that can be found in the training set. This super-skeleton is needed to cope with the fixed output of the network. As a result, this assumption makes the network unsuitable for working with skeleton topologies that can not fit in the super-skeleton structure.

RigNet [120] proposed to overcome this limitation using a kNN approach. In this work, the skin weights assignment is divided into two blocks: First, the *skin binding*, which consists of finding the k bones that will influence each vertex, using a kNN approach. Then, a network that predicts the skinning weights only for the k nearest bones of each vertex. This feature allows the network to work with unseen skeleton topologies, however, a bone representation is still used. By definition, a skeleton-driven mesh's movement comes from a joint's rotation. Both representations are equivalent in an ideal scenario where each of the joints has one associated bone. However, this assumption does not work for complex meshes, where a joint has more than one associated bone. The bone representation used in *RigNet* has problems managing these scenarios, common in stylized characters. To overcome this problem, our proposal directly uses a skeleton joint representation that can manage complex skeleton topologies where each joint can have more than one bone. Furthermore, both *Neuroskinning* and *RigNet* rely on handcrafted features to learn the relation between a mesh and its associated skeleton. Our proposal consists of a Two-Stream Graph Convolutional Neural Network that learns the relation between mesh and skeleton, automatically selecting the best features without relying on selected handcrafted features.

5.3 SkinningNet

SkinningNet is a Two-Stream Graph Convolutional Neural Network that takes as input a mesh and its corresponding skeleton and predicts a set of skinning weights, one for each mesh vertex. It is composed of four different stages, as depicted in Figure 5.2.

Stage 1: *Graph construction* and *skin binding* (explained in detail in Section 5.3.1). The *graph construction* step converts the mesh and the skeleton inputs into two independent graphs. The *skin binding* block decides which joints influence each vertex and creates a graph representing this relationship. Following this approach, we can define the number of predictions that the network has to make, making it possible to work with different kinds of skeleton topologies with different amounts of joints. The graph construction output is fed into *stage 2*, and the output of *skin binding* is used in *stage 3*.

Stage 2: *Mesh and Skeleton branches*. These branches transform the initial node attributes to feature vectors through an input transform implemented using an *MLP*. Each branch is responsible for extracting features independently for the mesh and skeleton. The *mesh branch* is composed of three residual *Multi-Aggregator Graph Convolution MAGC* layers, whereas the skeleton branch is composed of three *MAGC* layers, further details are given in Section 5.3.2 and 5.3.3. This difference is mainly because the skeleton is usually much simpler than the mesh and does not require a deep network to learn the characteristics of its geometry. The output of both branches is combined in *stage 3*.

Stage 3: *Mesh-Skeleton block*, based on a single *Heterogenous MAGC* layer. This block relates the mesh and the skeleton using the output of the *skin binding* block. The output of this block is a single graph where each node can represent the vertices of the mesh or the joints of the skeleton. However, only the nodes representing the mesh's vertices are used in the following stages. Furthermore, to help with the final skinning weight prediction in *stage 4*, a global shape descriptor that encodes the global information of the mesh and skeleton graphs is extracted and concatenated to the mesh nodes.

Stage 4: *Skinning Prediction Network*, composed of three *Multi-Neighbourhood MAGC* layers followed by an *MLP*. Here, the *MAGC* is exploited in a multi-neighbourhood fashion, combining the mesh topology and the local shape information to extract an enriched descriptor which is used to predict the skinning weights. Further details are given in Section 5.3.3.

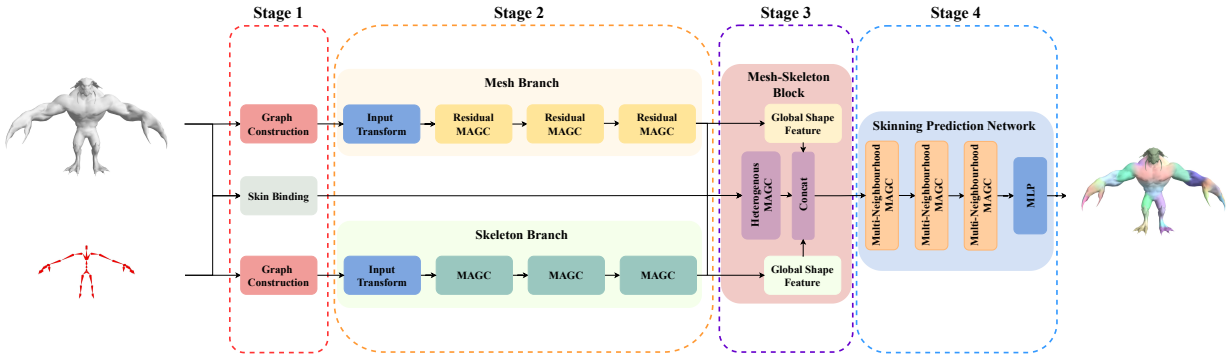


Figure 5.2: SkinningNet architecture is composed of four main stages. Stage 1 is in charge of building the needed graphs from the input mesh and its associated skeleton. Stage 2 is responsible for extracting features independently for the mesh and skeleton. Stage 3 combines the previous mesh and skeleton features to extract a descriptor that relates both structures. Stage 4 predicts the skinning weights. Asset from Paragon Collection [23].

5.3.1 Graph Construction

SkinningNet uses different strategies to create neighbourhoods for each kind of graph: *mesh*, *skeleton* and *mesh-skeleton* graphs. In the *mesh graph*, we use two kinds of neighbourhoods. The first one is the *one-ring* neighbourhood, where the faces of the mesh are converted into undirected edges. The second one follows the *radius proximity* policy, which is used by the *Multi-Neighbourhood MAGC* layer. Both of them are explained in detail in Section 2.4.

The *skeleton graph* is generated by converting the bones to undirected edges. Finally, the relation created by the *skin binding* block is used to define the connections in the *mesh-skeleton* graph.

The *mesh-skeleton* graph is created using the *skin binding* block, which is in charge of assigning which of the joints influence each vertex. For each vertex of the mesh, the closest bones of the skeleton are found, and the associated root joint for each bone is selected. Finally, that selection is refined, leaving only the k unique joints. The entire algorithm is described in Algorithm 4.

Algorithm 4: k unique joints of the closest bones

Input: $V \leftarrow$ vertex positions, $J \leftarrow$ joints positions, $B \leftarrow$ bones
Output: The selected k joints for each vertex

foreach $\vec{v_i}$ *in* V **do**
    Compute distance $\vec{d} \leftarrow d(v_i, B)$
    Sort distances $\vec{d}$
    Replace bones with their associated root joint
    Select the k unique nearest joints
end foreach

5.3.2 Multi-Aggregator Graph Convolution

The *Multi-Aggregator Graph Convolution (MAGC)* is an extension of the *message-passing* architecture [28], where multiple aggregators are used for allowing the graph convolution to better generalize for unseen topologies and learn better local representations. The workflow of the *MAGC* is depicted in Figure 5.3.

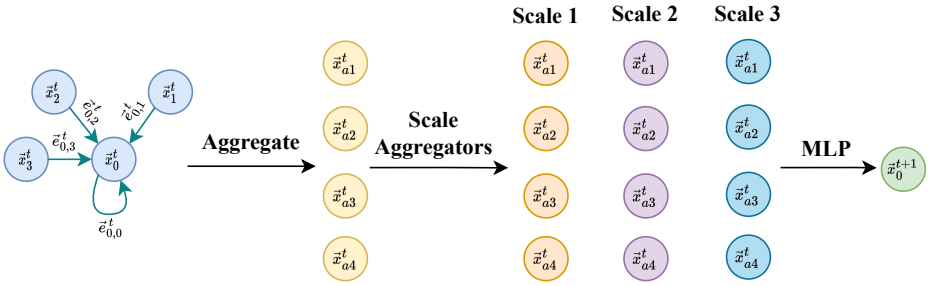


Figure 5.3: Multi-Aggregator Graph Convolution workflow.

The first step is to compute the messages each of the neighbours sends to the neighbourhood's central node. These messages are a function, $\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)$, that depends on the features $\vec{x}^t$ of the nodes i, j and the edge attribute $\vec{e}_{i,j}^t$. In this chapter, the messages are implemented using an *MLP* which takes as input the edge attributes that are an asymmetric function of the features of the nodes, $\vec{e}_{i,j}^t = \vec{x}_i^t || (\vec{x}_j^t - \vec{x}_i^t)$. The computed messages are combined using different aggregators, which are formally defined in Equation 5.2, where $N(i)$ represents the node's neighbourhood i.

$$A = \begin{cases} A_{max} = \underset{j \in N(i)}{max} (\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)) \\ A_{min} = \underset{j \in N(i)}{min} (\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)) \\ A_{mean} = \underset{j \in N(i)}{mean} (\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)) \\ A_{std} = \underset{j \in N(i)}{std} (\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)) \end{cases} \tag{5.2}$$

The results of each aggregator are scaled using a set of logarithmic degree scalers. The proposed scalers are:

1. *Amplification*, the value of the aggregator is amplified.

2. *Attenuation*, the value of each aggregator is attenuated.

3. *Identity*, the value of the aggregator is not changed.

Equation 5.3 formalizes the proposed scalers where d is the degree of the neighbourhood and d_{train} refers to the mean degree of the whole training split. The motivation for using different logarithmic scalers is to improve the generalization of the convolution for unseen topologies, avoiding that the value of each aggregation

explodes when the neighbourhood degree increases.

$$S = \begin{cases} S_{amp} = \dfrac{log(d)}{log(d_{train})} \\ S_{att} = \dfrac{log(d_{train})}{log(d)} \\ S_{iden} = 1 \end{cases} \tag{5.3}$$

Finally, the resulting operation of applying each scaler to each aggregator is fed into an *MLP* that learns how to fuse the information. Equation 5.4 defines the combination of aggregations and scalers, with A being the set of aggregation operations and S the set of scaler operations. The combination of two sets of operations is defined as $\otimes$, so $\mathcal{M}$ is the combination of all the scalers with all the aggregators.

$$\left.\begin{aligned} A = \{A_{max}, A_{min}, A_{mean}, A_{std}\} \\ S = \{S_{iden}, S_{amp}, S_{att}\} \\ \mathcal{M} = S \otimes A \end{aligned}\right\} \tag{5.4}$$

The *MAGC* layer is described in Equation 5.5, where all combinations, $\mathcal{M}$, are fused using an *MLP* network to produce the output feature of node i.

$$\vec{x}_i^{t+1} = \text{MLP}\left(\underset{j \in N(i)}{\mathcal{M}} \left(\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)\right)\right) \tag{5.5}$$

5.3.3 Graph Convolutional Blocks

The proposed architecture extends the previously defined *MAGC*, producing three types of *Graph Convolutional Blocks*: *Residual MAGC*, *Heterogenous MAGC* and *Multi-Neighbourhood MAGC*.

The *Residual MAGC* is the residual extension of graph convolution previously explained in Section 3.2.2. Each *Residual MAGC* is composed of two *MAGCs* stacked together with a residual connection. The residual graph convolution was previously defined in Equation 3.3 where, in this case, $\mathcal{F}(X^t, E)$ represents a stack of 2 MAGC layers.

The *Heterogenous Graph Convolution* employs a variant of *MAGC* to deal with graphs where the neighbourhood is composed of an *heterogeneous* combination of nodes. This variant concatenates a one-hot vector to the node feature before applying the MAGC that allows differentiating between a node from the mesh and a node from the skeleton defined in Equation 5.6.

$$\vec{x}_i^t = \begin{cases} 0 \| \vec{x}_i^t & \text{if} \quad i \in \text{mesh} \\ 1 \| \vec{x}_i^t & \text{if} \quad i \in \text{skeleton} \end{cases} \tag{5.6}$$

The *Multi-Neighbourhood MAGC* is an extension of *MAGC* into a multi-neighbourhood graph convolution previously defined in Section 4.2.2 where two kinds of neighbourhoods are used to get the new node feature. Equation 5.7 defines the extension of *MAGC* to a multi-neighbourhood approach. Where $\mathcal{K}$ is the set of neighbourhood types.

$$\vec{x}_i^{t+1} = \text{MLP}\left(\underset{k \in \mathcal{K}}{\text{concat}}\left\{\text{MLP}\left(\underset{j \in N^k(i)}{\mathcal{M}}\left(\phi(\vec{x}_i^t, \vec{x}_j^t, \vec{e}_{i,j}^t)\right)\right)\right\}\right) \tag{5.7}$$

In this chapter, instead of using the *maximum* or *average* to aggregate both neighbourhoods as we did in Section 4.2.2, we propose to learn the aggregation operation using an MLP.

5.4 Experimental Setup

5.4.1 Datasets and Metrics

The **RigNetv1** [120] dataset is publicly accessible for non-commercial use. This dataset is composed of 2703 rigged characters of different categories. The original split of the dataset is followed where 2163 assets are used for training, 270 for validation and 270 for testing. All training assets contain between 1k and 5k vertices and a mean of 25 skeleton joints.

The **Paragon Collection** [23] is allowed to be used in non-interactive linear media products under a non-exclusive and non-transferable license. Two assets from this set have been used as test, Aurora and Rampage, to study the generalization capability of the proposed method. Both have been simplified and contain about 6k vertices and 50 skeleton joints.

To evaluate the proposed method, four different metrics have been used:

- *Precision and Recall* of the selection of influencing joints for every vertex, as the wrong selection would cause unexpected deformations. We set the joint selection, y, as defined in Equation 5.8. Where $y_{i,j} = 1$ meaning that the $j-th$ joint is influencing the $i-th$ vertex, and α is a threshold set to $\alpha = 1 \times 10^{-4}$, as done by [69, 120]. We use this metric just to compare our method against the state-of-the-art.

$$y_{i,j} = \begin{cases} 1, & w_{i,j} \geq \alpha \\ 0, & w_{i,j} < \alpha \end{cases} \tag{5.8}$$

To compute the *Precision and Recall*, first, we compute the $\hat{y}$ for the prediction and the y for the ground truth. Then we compute the Precision and the Recall between $\hat{y}$ and y following Equations 5.9 and 5.10.

$$\text{Precision} = \frac{tp}{tp + fp} \tag{5.9}$$

$$\text{Recall} = \frac{tp}{tp + fn} \tag{5.10}$$

- *L1-norm* error between the predicted skinning weights $\hat{Y} \in \mathbb{R}^{v \times j}$ and the ground truth $Y \in \mathbb{R}^{v \times j}$ as described in Equation 5.11, where v is the number of vertices of a mesh and j the number of joints of the associated skeleton. We use this metric to compare our method against the state-of-the-art, using the average of the L1-norm error.

$$\text{L1-norm error} = |Y - \hat{Y}| \tag{5.11}$$

- *Deformation error* that evaluates the deformation quality. We compute the euclidean distance between the position of the vertices deformed by applying the predicted skinning weights and the ground truth. The metric is run over 10 different random poses generated by rotating the skeleton joints randomly within a range of ± 10 degrees. The deformation error is formalized in Equation 5.12, where $\hat{M} \in \mathbb{R}^{p \times v \times j}$ is the deformed mesh applying the predicted skinning weights, and $M \in \mathbb{R}^{p \times v \times j}$ is the deformed mesh applying the ground truth skinning weights, where v is the number of vertices of a mesh, j the number of joints of the associated skeleton and p the number of random poses. In this thesis, we report the average and maximum deformation errors. The average deformation error is computed as the average deformation error of all meshes vertices, and the maximum deformation error is computed as the maximum deformation error of all meshes vertices. We use this metric to compare our method against the state-of-the-art and to run our ablation studies.

$$\text{Def. Error} = \|M - \hat{M}\| \tag{5.12}$$

5.4.2 Implementation details

5.4.2.1 Pre-processing input data

The mesh and skeleton positions are scaled between $[-1, 1]$ and oriented to face the same direction.

The mesh branch's input is the mesh's vertices, which are converted into two different graphs: *mesh topology* and *radius proximity* graph. The radius proximity graph uses a radius $r = 0.06$ with a maximum of 10 nodes. In addition, we use the geodesic distance over the mesh as a distance to compute the neighbourhoods. As a

feature of each mesh's node, we use the 3D coordinates of the vertices, the *geodesic distance* to the $k = 5$ unique joints from the nearest bones, described in Section 5.3.1, the start and end position of the bones of those joints and a boolean value indicating if the joint is an end joint or not.

In the case of the *skeleton* branch, the skeleton is converted into a graph using the bones as undirected edges, as described in Section 5.3.1. As a feature of the nodes, the 3D position of the joints is used.

Finally, the *Mesh-Skeleton block* needs a third graph, the mesh-skeleton graph, which is created using the Algorithm 4, using the volumetric geodesic distance [120] to compute the distances between vertices and joints. This distance consists of computing the shortest path from a vertex to a joint passing through the interior mesh volume.

5.4.2.2 Architecture details

The detailed architecture with the number of filters used in each layer is shown in Table 5.1, where k is the number of joints that influence each vertex. In this thesis, $k = 5$ since it is the maximum value of joints that will influence a vertex of the mesh in the training split of the *RigNetv1* dataset. A dropout layer is added before each MLP of the *Skinning Prediction Network* with a probability of $p = 0.5$ and the activation function *Softmax* is applied to the weights predicted for each vertex.

Mesh Branch	
Layer	**N. Filters**
Input Transform	MLP(64, 128)
Residual MAGC	128
Residual MAGC	256
Residual MAGC	512
Skeleton Branch	
Input Transform	MLP(64)
MAGC	128
MAGC	256
MAGC	512
Mesh - Skeleton Block	
Mesh Global Shape	MLP(256)
Skeleton Global Shape	MLP(256)
Mesh-Skel MAGC	512
Concat	512 + 256 + 256
Skinning Prediction Network	
Multi-Neighbourhood MAGC	256
Multi-Neighbourhood MAGC	128
Multi-Neighbourhood MAGC	64
MLP	(64, 32, k)

Table 5.1: SkinningNet architecture details. k is the number of joints that can influence each vertex.

5.4.2.3 Training details

The network is trained in an end-to-end fashion using the early-stopping criteria with a patience of 20 epochs and a maximum of 200. The Rectified Adam (RAdam) [67] optimizer is used with a learning rate of 1×10^{-4}, a weight decay of 1×10^{-4}, and a batch size of 4. The Kullback-Leibler divergence loss, formalized in Equation 5.13, minimizes the distance between the predicted skinning weight distribution and the ground truth distribution, where $y_{i,j}$ is the ground truth value of the i vertex and the j joint. $\hat{y}_{i,j}$ is the prediction of the network, n is the total number of vertices, and k is the maximum number of joints influencing each vertex.

$$\mathcal{L}_{kl} = \frac{1}{n \times k} \sum_{i=1}^{n} \sum_{j=1}^{k} y_{i,j}(log(y_{i,j}) - log(\hat{y}_{i,j})) \tag{5.13}$$

Moreover, by carrying extensive hyperparameter tuning, we ensure that differences in performance can be attributed to modelling choices rather than incomplete hyperparameter optimization. We used the Hyperband [62] algorithm over the validation split. Once we have selected the hyperparameters, we evaluate the model on the test split.

5.5 Results

5.5.1 Ablation Studies

5.5.1.1 Study of the architecture design

This study aims to understand how each component of our proposed network architecture influences the final result. Specifically, we are going to remove the *global shape feature* for both branches, the *residual connections* from the mesh branch, and the *Multi-Neighbourhood MAGC* will be replaced by two *MAGC* with the same number of output features. The results of these experiments are shown in Table 5.2.

Removing the *Multi-Neighbourhood MAGC* from the network leads to a loss of performance of about 5%. These results show that *Multi-Neighbourhood MAGC* is helping to get an enriched local descriptor for each of the vertices, which enables the network to be aware of the local structure around the vertices. A similar performance loss is obtained when removing the *global shape feature* as it helps the network to be aware of the global shape of the skeleton and the mesh when making the prediction. Finally, adding the residual connections to the *Mesh Network* leads to an improvement of 4%, demonstrating that the residual connections are helping the proposed approach.

Method	Avg. Def	Max. Def
SkinningNet	**0.002288**	**0.1789**
No Global Feature	0.002452	0.1905
No Residual	0.002394	0.1862
No Multi-Neighbourhood MAGC	0.002427	0.2009

Table 5.2: Study of the influence of each of the proposed stages on RigNetv1 dataset [120].

5.5.1.2 Comparison of Joint vs. Bone representation

In this section, the difference between a joint vs. a bone representation is analyzed. The *skin binding* step is modified to use bones instead of joints when creating the relations between the mesh and the skeleton. Table 5.3 shows the results of both approaches. As can be seen, the *skeleton joint* representation gives an improvement of 5% in both average and maximum deformation. This improvement is because the joint representation follows a natural approach where each of the joints represents an articulation which is in charge of defining the movement of each bone.

Method	Avg. Def	Max. Def
Joint	**0.002288**	**0.1789**
Bone	0.002407	0.1893

Table 5.3: Joint vs. Bone representation study, where the influence of each representation is analyzed on RigNetv1 dataset [120].

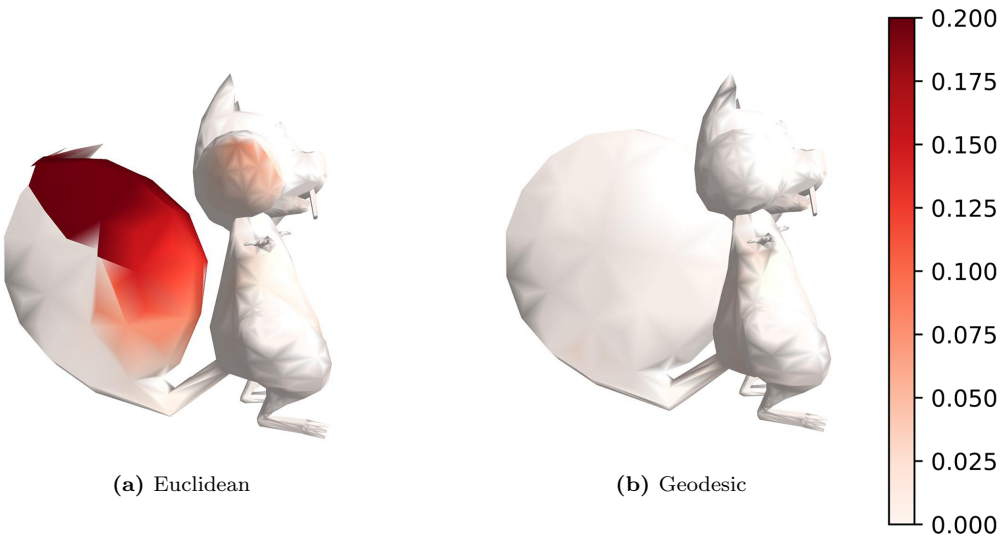
5.5.1.3 Comparison of Euclidean vs. Geodesic distance

Finding the vertex to joint distance is critical for skinning prediction methods. NeuroSkinning [69] proposed using the *euclidean distance*, while RigNet [120] proposed using the *geodesic distance*. Both distances have their advantages and disadvantages. The *geodesic distance* is defined for connected components, with the distance between two non-connected components being infinite. This means that the *geodesic distance* is better suited for watertight meshes, whereas the *euclidean distance* can be used for both watertight and non-watertight meshes. In terms of performance, using *geodesic distance* helps the network to predict better results than using *euclidean distance*, as observed in Table 5.4.

Method	Avg. Def	Max. Def
Euclidean	0.002663	0.4473
Geodesic	**0.002288**	**0.1789**

Table 5.4: Geodesic vs. Euclidean performance comparison on RigNetv1 dataset [120].

An example of the effects of a higher maximum error when using the *euclidean distance* can be observed in Figure 5.4 where the tail of the squirrel is wrongly deformed. The euclidean distance relates the mesh vertices with joints in the back of the squirrel, whereas the geodesic distance avoids this problem.



(a) Euclidean **(b)** Geodesic

Figure 5.4: Euclidean vs. Geodesic qualitative results. It can be observed that the euclidean distance introduces big deformation errors in the tail. Asset from RigNetv1 dataset [120].

5.5.1.4 Comparison with different Graph Convolutional Layers

In this experiment, the MAGC has been replaced with four different state-of-the-art graph convolutions and the AGC to study this operator's influence on the prediction output. The *MLP* needed by the AGC to generate the weights is composed of two MLP layers with output features $(128, d_l \times d_{l-1})$, where $d_l$is the number of output features of the layer l, and the edge attributes are the concatenation of the positional offset and the node feature offset as defined in Section 4.2.1. Table 5.5 shows that AGC achieves the closest results to the proposed MAGC, which is outperformed by the MAGC with an improvement of 2.6% on the average deformation error.

The MAGC needs fewer parameters than the AGC, which requires an MLP that generates the weights used to generate the message of each neighbouring node which needs its own parameters too. Whereas the MAGC uses an MLP of just one layer to directly get the message, which reduces the memory footprint. Specifically, the AGC needs $(edge\_attributes\_channels \times c) + (c \times input\_channels \times output\_channels)$ weights, where c is the number of outputs of the first layer of the MLP that generates the weights. Whereas the MAGC needs $(edge\_attributes\_channels \times output\_channels) + (output\_channels \times number\_aggregators \times number\_scalers \times out\_channels)$ weights. The main difference resides in the last term. The number of parameters of the AGC depends on the product between the number of input and output channels of the layer. In contrast, the MAGC depends on the product between the number of scalers, the number of aggregators and the number of output channels. The product between the number of aggregators and scalers will be considerably lower than the number of inputs which in deeper layers will correspond to high values.

Method	Avg. Def	Max. Def
MAGC	**0.002288**	**0.1789**
AGC	0.002351	0.1873
EdgeConv [115]	0.002381	0.1921
FeaStConv [112]	0.002431	0.2054
GAT [111]	0.002551	0.2098
GCN [53]	0.002765	0.2533

Table 5.5: Graph Convolution study where MAGC has been compared with three different state-of-the-art operators on RigNetv1 dataset [120].

5.5.1.5 Study of the influence of the learned aggregator of the Multi-Neighbourhood Graph Convolution

This work proposes to learn the aggregation operation used on the Multi-Neighbourhood approach instead of using the maximum or average aggregators as done in Chapter 4. We can observe that using a learned approach based on an MLP, we get an improvement of 3.7% as shown in Table 5.6.

Aggregation	Avg. Def	Max. Def
MLP	**0.002288**	**0.1789**
Average	0.002376	0.1874
Maximum	0.002450	0.2023

Table 5.6: Comparison of different aggregation strategies on the Multi-Neighbourhood Graph extension on RigNetv1 dataset [120].

5.5.2 Comparison with state-of-the-art

SkinningNet is compared to the two most recent data-driven approaches: NeuroSkinning [69] and RigNet [120]. Both networks are trained from scratch following the procedure described in their respective papers. In the case of NeuroSkinning, the original *euclidean distance* is replaced with the *geodesic* one, as it has been demonstrated in Section 5.5.1.3, to be a more suitable choice for watertight meshes. Furthermore, the two architectures use the same input data as the one proposed in our work to guarantee a fair comparison.

SkinningNet outperforms the best method of the state-of-the-art with over 5% of improvement on Precision with the same Recall and 15% improvement on average L1-norm. Table 5.7 summarizes the comparison with the state-of-the-art. Figure 5.5 shows the skinning weights predicted for three assets from the test set of the *RigNetv1* dataset using the state-of-the-art methods and the proposed method. To get this representation, a random colour is assigned to each of the joints of the skeleton. Then, colours are blended using the skinning weights associated with each vertex. We can see that our method better predicts the skinning weights with associated colours that are closer to those of the ground truth in the first column.

Method	**Prec.(%)**	**Rec.(%)**	**Avg. L1**
NeuroSkinning [69]	82.3	79.7	0.41
Rignet [120]	82.3	**80.8**	0.39
SkinningNet	**87.0**	**80.8**	**0.33**

Table 5.7: Comparison with the current state-of-the-art techniques on RigNetv1 dataset [120].

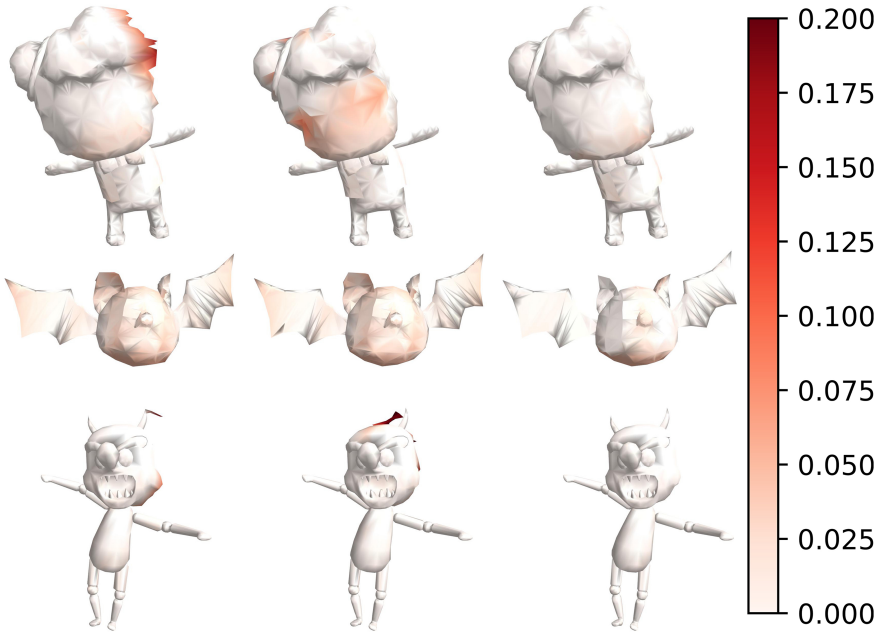
In terms of deformation error, Table 5.8 shows that our method outperforms with a 20% of improvement in the average error and a 17% improvement in the maximum error. The qualitative results of this metric can be observed in Figure 5.6, where SkinningNet can generate reasonable results where previous state-of-the-art methods fail.

Method	**Avg. Def**	**Max. Def**
NeuroSkinning [69]	0.002843	0.2151
Rignet [120]	0.002921	0.2246
SkinningNet	**0.002288**	**0.1789**

Table 5.8: Deformation error comparison with the current state-of-the-art techniques on RigNetv1 dataset [120].

(a) GT **(b)** NeuroSkinning **(c)** RigNet **(d)** SkinningNet

Figure 5.5: Skinning weight prediction results of each of the state-of-the-art methods. Each joint is assigned a random colour, and colours are blended using the skinning weights associated with each vertex. Assets from RigNetv1 dataset [120].



(a) NeuroSkinning **(b)** RigNet **(c)** SkinningNet

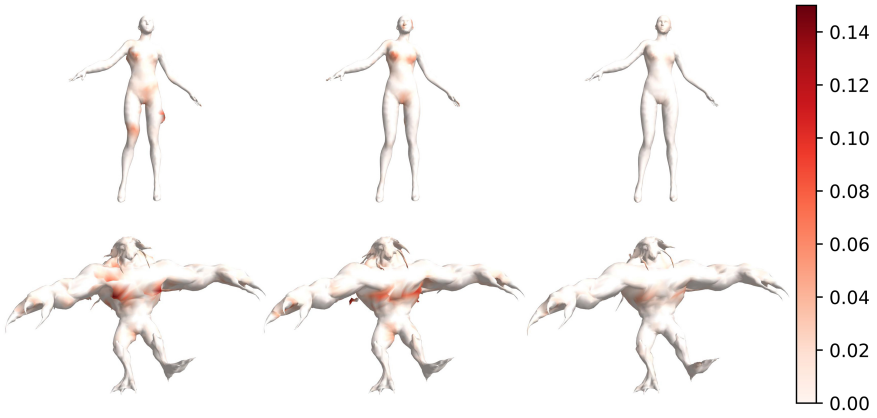
Figure 5.6: Deformation error of three different characters with a randomly generated pose. Assets from RigNetv1 dataset [120].

5.5.2.1 Generalization study

In this section, we are going to demonstrate that our proposed method has a better generalization and is suitable for working with high-quality game characters. The networks are trained using the RigNetv1 [120] dataset and the result of it is applied to the *Aurora* and *Paragon* assets from the Paragon Collection [23]. Results in Table 5.9 show that our method outperforms previous works with over 28% of improvement in average deformation and 36% in maximum deformation error, proving that the proposed method generalizes better than previous methods for unseen complex characters. In Figure 5.7, it can be observed that the proposed method generates good-quality animations without strong errors, whereas the other methods have high errors in both characters.

Method	**Avg. Def**	**Max. Def**
NeuroSkinning	0.003724	0.1213
Rignet	0.003398	0.1051
SkinningNet	**0.002666**	**0.0664**

Table 5.9: Generalization Study of the state-of-the-art methods. The deformation error is computed using a normalized version of the Paragon Assets [23].



(a) NeuroSkinning **(b)** RigNet **(c)** SkinningNet

Figure 5.7: Generalization study with Aurora and Rampage assets from the Paragon collection [23].

5.5.3 Knowledge Transfer

Following the original publication of the *SkinningNet* paper [109], we have been delighted to collaborate with Epic Games to create an application based on *SkinningNet* to help artists speed up their workflow.

SkinningNet is retrained using a new dataset based on non-watertight assets from the Fortnite™ game [24], composed of 767 assets, where 690 of them are used for training and 77 for testing. On average, we have 10k vertices and 101 joints per asset, as shown in Figure 5.8, increasing the complexity that we had in the previous dataset.

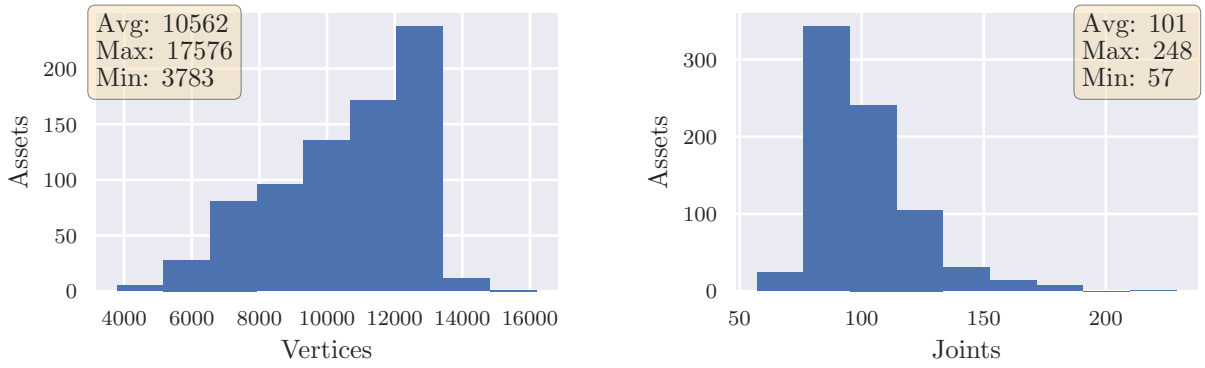


Figure 5.8: Analysis of the statistics in terms of the number of vertices and joints of the Fortnite™ dataset [24].

Because the meshes are non-watertight and layered (we have the body and clothes as different layers of vertices), we use the *euclidean distance* instead of the *geodesic distance* to build the mesh-skeleton graph. In addition, the number of joints influencing each vertex is increased to 8 instead of the 5 we had in the previous dataset. Figure 5.9 shows the qualitative results of applying the skinning weights predicted by *SkinningNet* to a skeleton asset from the test split. The first row shows the resulting animations using the predicted skinning weights, and the second row shows the resulting animations using the ground truth skinning weights. Overall, the predicted skinning weights generate reasonable animations; however, if we compare both results in detail, we observe some errors when deforming the hips and other parts of the clothing. Figure 5.10 shows the weight map of the joint *spine_05*, where we can observe that the skinning weights predicted are not smooth on this section, and it might provoke the wrong deformations on the upper part of the skeleton chest, depicted in Figure 5.9.

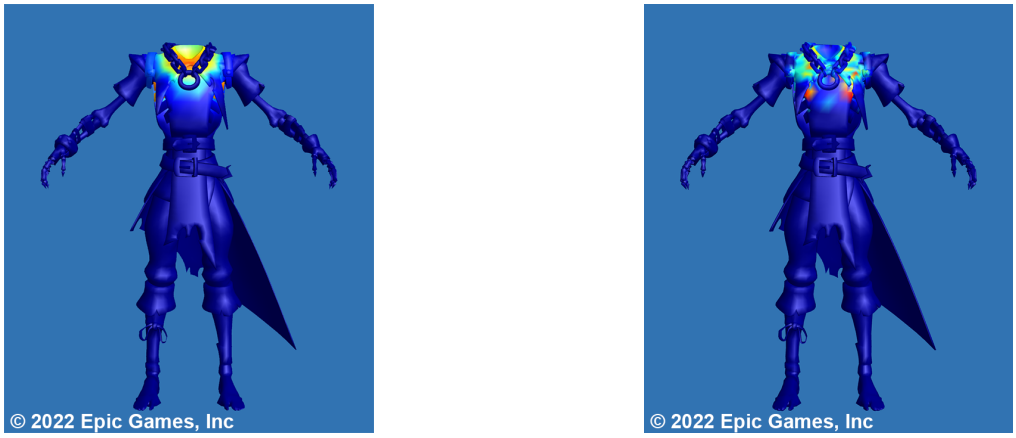
Figure 5.11 shows the qualitative results of applying the skinning weights predicted by *SkinningNet* to a robot asset from the test split. As before, the predicted skinning weights generate reasonable animations; however, when we look closely, we observe that there are deformations on the tubes and in some of the details of the armature. In addition, in frame 2, we observe that in the shoulder, there is an outlier. Figure 5.12 shows the weight map of the joint *pelvis*, where we can observe that the skinning weights predicted are not smooth in this section, and it might provoke the wrong deformations on the pelvis area, depicted in Figure 5.11.

Even though the predictions are not perfect, our tool allows the artist to have, in less than $10s$, a decent first version of the skinning weights that they can quickly refine later, saving many hours of manual work.

(a) Prediction Frame 1 **(b)** Prediction Frame 2 **(c)** Prediction Frame 3

(d) GT Frame 1 **(e)** GT Frame 2 **(f)** GT Frame 3

Figure 5.9: Qualitative results using SkinningNet over a skeleton asset of the Fortnite™ dataset [24].



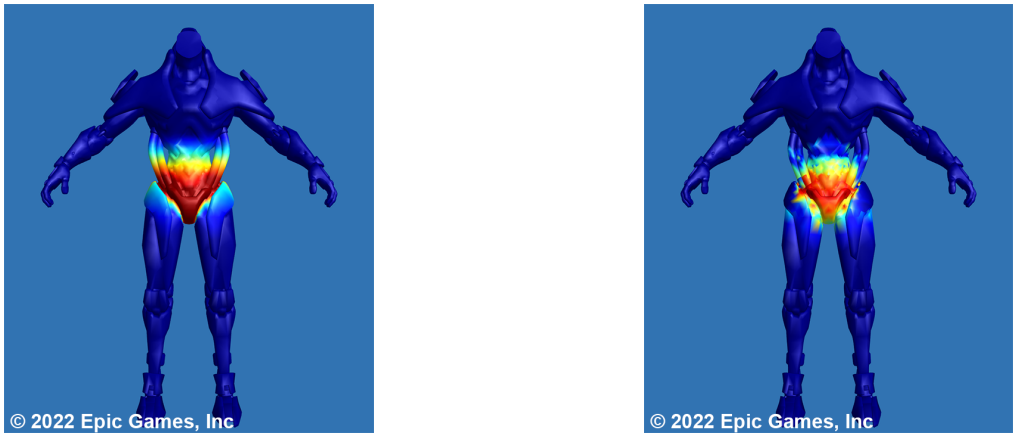
(a) GT **(b)** Prediction

Figure 5.10: Comparison of predicted and ground truth skinning weights of the joint spine05 of the skeleton asset from the Fortnite™ dataset [24].

(a) Prediction Frame 1 **(b)** Prediction Frame 2 **(c)** Prediction Frame 3

(d) GT Frame 1 **(e)** GT Frame 2 **(f)** GT Frame 3

Figure 5.11: Qualitative results using SkinningNet over a robot asset of the Fortnite™ dataset [24].



(a) GT **(b)** Prediction

Figure 5.12: Comparison of predicted and ground truth skinning weights of the joint pelvis of the robot asset from the Fortnite™ dataset [24].

5.6 Conclusions

In this chapter, we presented the Multi-Aggregator Graph Convolution, which extends the message-passing architecture to combine multiple aggregators that help to better generalize for unseen topologies and learn better local representations. The proposed Multi-Aggregator Graph Convolution requires less memory than the Attention Graph Convolution and also gets better results, as is shown in Section 5.5.1.4. In addition, the new Multi-Aggregator Graph Convolution outperforms previous state-of-the-art graph convolutional layers on the task of node analysis of heterogeneous graphs, which effectiveness has been proved on the automatic skinning prediction task. To solve this task, we proposed SkinningNet, a Two-Stream Graph Convolutional Neural Network that automatically generates skinning weights for an input mesh and its associated skeleton. The proposed architecture has two independent branches, composed of different variants of Multi-Aggregator Graph Convolutions, that process the skeleton and the mesh independently. The output of both branches is fused using the heterogeneous variant of the Multi-Aggregator Graph Convolution, which as input, takes a heterogeneous graph that contains nodes from the mesh and from the skeleton and outputs new features for each of them. The output of the heterogeneous graph convolution is used to predict the skinning weights for each of the vertices of the mesh. The proposed architecture outperforms current approaches with over a 20% improvement on mesh deformation error and is also able to better generalize for complex characters of unseen domains.

Even though the results are promising, there are some limitations in the creation of the graphs. For instance, in this specific problem of automatically generating the skinning weights, we created the graph that relates the mesh and the skeleton assigning the joints that will influence each of the vertices based on a kNN approach. However, if the joint that should influence the vertex is not part of the neighbourhood, the network will not be able to find it. As future work, in order to improve the graph creation step, we propose to explore *link prediction* strategies to let the network learn the binding strategy, which will allow the network to learn the best neighbourhood configuration for each scenario.

CHAPTER 6

Conclusions

Throughout this thesis, we prove how Graph Convolutional Neural Networks are currently the way to go when dealing with 3D data. We will conclude the write-up briefly, summarizing the key contributions, followed by an overview of future research lines.

6.1 Summary of contributions

- In Chapter 3, we have designed a Graph Convolutional Neural Network architecture capable of extracting rich geometric features from 3D data. This neural network is composed of three novel components. The first component is the Attention Graph Convolution (AGC) layer, which outperforms previous state-of-the-art graph convolutional layers. The main characteristics of the AGC reside in its ability to infer the kernel used during the convolution, taking into account the neighbourhood configuration, allowing it to better capture the neighbourhood characteristics. The second component is the extension of residual connections to the Attention Graph Convolution, which effectively improves the performance of different graph convolutional layers in deep networks. The third component is the Voxel Pooling layer which allows mimicking the traditional CNNs architectures and learning richer, higher-level features. The results of our experimentation on the Graph Analysis task show that our method is capable of extracting better 3D geometric characteristics than the previous state-of-the-art.

- In Chapter 4, we presented a 2D-3D Fusion Network that combines the 3D geometric information obtained by Graph Convolutional Neural Network and the 2D texture features obtained by a 2D CNN. The proposed network is able to fuse 2D and 3D features at different resolutions/sampling without having a 1-to-1 correspondence between features. Furthermore, we introduced the Multi-Neighbourhood Graph Convolution, which allows exploiting the use of multiple neighbourhoods generated in different spaces, such as the euclidean and feature spaces, to learn richer representations. Finally, we improved the previous version of the Voxel Pooling layer proposing the Nearest Voxel Pooling layer that mitigates the influence of the outliers. The results of our experimentation on the Graph Analysis task show that

our method is better at managing multi-modal characteristics than the previous state-of-the-art.

- In Chapter 5, we evaluated previously proposed methods to infer characteristics at a node level of a heterogenous graph. In addition, we introduced the Multi-Aggregator Graph Convolution (MAGC) that uses a multiple aggregator approach to better generalize for unseen topologies and learn better local representations. Moreover, this new layer reduces the memory footprint with respect to the Attention Graph Convolution. Finally, we show a variant of the MAGC that better handles heterogeneous graphs. The results of our experimentation on the Node Analysis task show that the proposed MAGC outperforms previous state-of-the-art graph convolution layers.

6.2 Future work

- Converting 3D data to a graph and giving control to a neural network to create/modify the graph is still an open problem. Some attempts try to predict links between nodes [124, 125]. However, most of them are done in transductive learning scenarios where the problem is simplified. Exploring methodologies to apply link prediction strategies in transductive learning scenarios is a challenging task that would help to learn better relations between nodes.

- It is fair to say that the predominant architecture underlying current Graph Convolutional Neural Networks is message passing. However, there exist other promising options to deal with graphs that, in the future, could outperform the message-passing, such as diffusion-based methods [12, 22]. Finding new efficient ways to deal with graph data could help to deal with large 3D data.

- Extending current Graph Convolutional Neural Networks to work with heterogenous graphs is still an open challenge. Most of the attempts [44, 114] done previously are made in transductive learning scenarios where the problem is simplified. Exploring methodologies to handle heterogeneous graphs on transductive scenarios is indeed a challenge that must be addressed in the future.

- In the area of Graph Convolutional Networks, interpretability is experiencing rapid development. However, the research focus is on knowledge graphs and molecules [72, 95, 122]. Exploring methodologies to analyze the knowledge learned by a Graph Convolutional Neural network trained on 3D points clouds and meshes would help to better understand current architectures and discover unknown issues.

Bibliography

[1] James Atwood and Don Towsley. Diffusion-convolutional neural networks. *Proceedings the Advances in neural information processing systems Conference*, 29, 2016.

[2] Norman I Badler and Mary Ann Morris. Modelling flexible articulated objects. In *Proceedings Computer Graphics*, pages 305–314, 1982.

[3] Seungbae Bang and Sung-Hee Lee. Spline interface for intuitive skinning weight editing. *ACM Transactions on Graphics*, 37(5), 2018.

[4] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. *ACM Transactions on Graphics*, 26(3), 2007.

[5] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[7] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. In *Proceedings of the International Conference on Learning Representations*, 2019.

[8] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[9] M. Brown and S. Süsstrunk. Multi-spectral sift for scene category recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 177–184, June 2011.

[10] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *Proceedings of the International Conference on Learning Representations*, 2014.

[11] Ziyun Cai and Ling Shao. Rgb-d scene classification via multi-modal feature learning. *Cognitive Computation*, Aug 2018.

[12] Ben Chamberlain, James Rowbottom, Maria I Gorinova, Michael Bronstein, Stefan Webb, and Emanuele Rossi. Grand: Graph neural diffusion. In *Proceedings of the International conference on machine*

learning, volume 139 of *Proceedings of Machine Learning Research*, pages 1407–1418. PMLR, 2021.

[13] Xu Chen, Yufeng Zheng, Michael J Black, Otmar Hilliges, and Andreas Geiger. SNARF: Differentiable Forward Skinning for Animating Non-Rigid Neural Implicit Shapes. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021.

[14] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Empirical Methods in Natural Language Processing Conference*, 2014.

[15] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9268–9277, 2019.

[16] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Proceedings the Advances in neural information processing systems Conference*, 29, 2016.

[17] Nima Dehmamy, Albert-László Barabási, and Rose Yu. Understanding the representation power of graph neural networks in learning graph topology. In *Proceedings of the Neural Information Processing Systems Conference*, 2019.

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[19] O. Dionne and M. de Lasa. Geodesic binding for degenerate character geometry using sparse voxelization. *IEEE Trans. Vis. & Comp. Graphics*, 20(10), 2014.

[20] D Du, L Wang, H Wang, K Zhao, and G Wu. Translate-to-Recognize Networks for RGB-D Scene Recognition. In *Conference on Computer Vision and Pattern Recognition*, pages 11828–11837. IEEE, 2019.

[21] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *Proceedings the Advances in neural information processing systems Conference*, 28, 2015.

[22] Moshe Eliasof, Eldad Haber, and Eran Treister. Pde-gcn: Novel architectures for graph neural networks motivated by partial differential equations. *Proceedings the Advances in neural information processing systems Conference*, 34:3836–3849, 2021.

[23] Epic Games. *Paragon Collection*. Epic Games, 2018. `https://www.epicgames.com/`.

[24] Epic Games. *Fortnite™*. Epic Games, 2022. `https://www.epicgames.com/`.

[25] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *Proceedings of the International Conference on Learning Representations Workshop*, 2019.

[26] Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of

data structures. *IEEE transactions on Neural Networks*, 9(5):768–786, 1998.

[27] Jonas Gehring, Michael Auli, David Grangier, and Yann N. Dauphin. A convolutional encoder model for neural machine translation. In Regina Barzilay and Min-Yen Kan, editors, *Association for Computational Linguistics*, pages 123–135, 2017.

[28] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[29] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1440–1448, 2015.

[30] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2014.

[31] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[33] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *IEEE international joint conference on neural networks*, 2005.

[34] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[35] Saurabh Gupta, Ross Girshick, Pablo Arbeláez, and Jitendra Malik. Learning Rich Features from RGB-D Images for Object Detection and Segmentation. In *European Conference on Computer Vision*, pages 345–360. Springer International Publishing, 2014.

[36] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Proceedings the Advances in neural information processing systems Conference*, 30, 2017.

[37] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.

[38] Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1):335–346, 1990.

[39] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2017.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1026–1034, 2015.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[42] Vishakh Hegde and Reza Zadeh. Fusionnet: 3d object classification using multiple data representations. *arXiv preprint arXiv:1607.05695*, 2016.

[43] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

[44] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of The Web Conference*, pages 2704–2710, 2020.

[45] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International conference on machine learning*, pages 448–456. PMLR, 2015.

[46] Alec Jacobson, Ilya Baran, Jovan Popoviundefined, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Transactions on Graphics*, 30(4), 2011.

[47] Varun Jampani, Martin Kiefel, and Peter V Gehler. Learning sparse high dimensional filters: Image filtering, dense crfs and bilateral neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4452–4461, 2016.

[48] Maximilian Jaritz, Jiayuan Gu, and Hao Su. Multi-view pointnet for 3d scene understanding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshop*, pages 0–0, 2019.

[49] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with dual quaternions. In *Proceedings of the symposium on Interactive 3D graphics and games*, pages 39–46, 2007.

[50] Ladislav Kavan and Olga Sorkine. Elasticity-inspired deformers for character articulation. *ACM Transactions on Graphics*, 31(6), 2012.

[51] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952.

[52] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Proceedings of the International Conference on Learning Representations*, 2015.

[53] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations*, 2017.

[54] Martin Komaritzan and Mario Botsch. Projective skinning. *Proceedings of the ACM in Computer Graphics and Interactive Techniques*, 1(1), 2018.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Proceedings the Advances in neural information processing systems Conference*, 25, 2012.

[56] Binh Huy Le and Zhigang Deng. Smooth skinning decomposition with rigid bones. *ACM Transactions on Graphics*, 31(6):1–10, 2012.

[57] Quoc V. Le, Alexandre Karpenko, Jiquan Ngiam, and Andrew Y. Ng. Ica with reconstruction cost for efficient overcomplete feature learning. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Proceedings the Advances in neural information processing systems Conference*, pages 1017–1025. Curran Associates, Inc., 2011.

[58] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[59] John P Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the conference on Computer graphics and interactive techniques*, pages 165–172, 2000.

[60] John P Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the conference on Computer graphics and interactive techniques*, pages 165–172, 2000.

[61] Duo Li, Shinjiro Sueda, Debanga R Neog, and Dinesh K Pai. Thin skin elastodynamics. *ACM Transactions on Graphics*, 32(4):1–10, 2013.

[62] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[63] Peizhuo Li, Kfir Aberman, Rana Hanocka, Libin Liu, Olga Sorkine-Hornung, and Baoquan Chen. Learning Skeletal Articulations with Neural Blend Shapes. *ACM Transactions on Graphics*, 40, 2021.

[64] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[65] Yabei Li, Junge Zhang, Yanhua Cheng, K. Huang, and T. Tan. Df2net: Discriminative feature learning and fusion network for rgb-d indoor scene classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[66] Yabei Li, Zhang Zhang, Yanhua Cheng, Liang Wang, and Tieniu Tan. MAPNet: Multi-modal attentive pooling network for RGB-D indoor scene classification. *Pattern Recognition*, 90:436–449, 2019.

[67] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. In *Proceedings of the International Conference on Learning Representations*, 2020.

[68] Libin Liu, KangKang Yin, Bin Wang, and Baining Guo. Simulation and control of skeleton-driven soft body characters. *ACM Transactions on Graphics*, 32(6):1–8, 2013.

[69] Lijuan Liu, Youyi Zheng, Di Tang, Yi Yuan, Changjie Fan, and Kun Zhou. Neuroskinning: Automatic skin binding for production characters with deep graph networks. *ACM Transactions on Graphics*, 2019.

[70] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2015.

[71] Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. SMPL: A skinned multi-person linear model. *ACM Transactions on Graphics*, 34(6), 2015.

[72] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network. *Proceedings the Advances in neural information processing systems Conference*, 33:19620–19631, 2020.

[73] N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. Joint-dependent local deformations for hand animation and object grasping. In *Proceedings Graphics Interface*, 1988.

[74] David Marr. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. Henry Holt and Co., Inc., 1982.

[75] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Proceedings the Advances in neural information processing systems Conference*, 26, 2013.

[76] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5115–5124, 2017.

[77] Tomohiko Mukai and Shigeru Kuriyama. Efficient dynamic skinning with low-rank helper bone controllers. *ACM Transactions on Graphics*, 35(4), 2016.

[78] Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *European Conference on Computer Vision*, 2012.

[79] Allen Newell. Physical symbol systems*. *Cognitive Science*, 4(2):135–183, 1980.

[80] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *Proceedings of the International conference on machine learning*, pages 2014–2023. PMLR, 2016.

[81] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *Proceedings the Advances in neural information processing systems Conference Workshop*, 2017.

[82] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[83] Michael Pratscher, Patrick Coleman, Joe Laszlo, and Karan Singh. Outside-in anatomy based character rigging. In *Proceedings of the ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 329–338, 2005.

[84] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 652–660, 2017.

[85] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, page 5105–5114, 2017.

[86] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[87] Olivier Rémillard and Paul G Kry. Embedded thin shells for wrinkle simulation. *ACM Transactions on Graphics*, 32(4):1–8, 2013.

[88] M. Ren and R. S. Zemel. End-to-end instance segmentation with recurrent attention. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, July 2017.

[89] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Proceedings the Advances in neural information processing systems Conference*, 28, 2015.

[90] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[91] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[92] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[93] Shunsuke Saito, Jinlong Yang, Qianli Ma, and Michael J. Black. SCANimate: Weakly supervised learning of skinned clothed avatar networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2021.

[94] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[95] Thomas Schnake, Oliver Eberle, Jonas Lederer, Shinichi Nakajima, Kristof T Schütt, Klaus-Robert Müller, and Grégoire Montavon. Higher-order explanations of graph neural networks via relevant walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7581–7596, 2021.

[96] Weiguang Si, Sung-Hee Lee, Eftychios Sifakis, and Demetri Terzopoulos. Realistic biomechanical simulation and control of human swimming. *ACM Transactions on Graphics*, 34(1), 2015.

[97] N. Silberman and R. Fergus. Indoor scene segmentation using a structured light sensor. In *Proceedings of the International Conference on Computer Vision - Workshop on 3D Representation and Recognition*, 2011.

[98] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image

recognition. In *Proceedings of the International Conference on Learning Representations*, 2015.

[99] Richard Socher, Brody Huval, Bharath Bhat, Christopher D. Manning, and Andrew Y. Ng. Convolutional-recursive deep learning for 3d object classification. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 656–664, USA, 2012. Curran Associates Inc.

[100] S. Song, S. P. Lichtenberg, and J. Xiao. Sun rgb-d: A rgb-d scene understanding benchmark suite. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 567–576, June 2015.

[101] X. Song, S. Jiang, L. Herranz, and C. Chen. Learning effective rgb-d representations for scene recognition. *IEEE Transactions on Image Processing*, 28(2):980–993, 2019.

[102] X. Song, S. Jiang, B. Wang, C. Chen, and G. Chen. Image representations with spatial object-to-object relations for rgb-d scene recognition. *IEEE Transactions on Image Processing*, 29:525–537, 2020.

[103] Alessandro Sperduti and Antonina Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

[104] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[105] 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/CVF Conference on Computer Vision and Pattern Recognition*, pages 2530–2539, 2018.

[106] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the international conference on world wide web*, pages 1067–1077, 2015.

[107] **Albert Mosella-Montoro** and Javier Ruiz-Hidalgo. Residual Attention Graph Convolutional Network for Geometric 3D Scene Classification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshop*, 2019.

[108] **Albert Mosella-Montoro** and Javier Ruiz-Hidalgo. 2D–3D Geometric Fusion network using Multi-Neighbourhood Graph Convolution for RGB-D Indoor Scene Classification. *Information Fusion*, 76:46–54, 2021.

[109] **Albert Mosella-Montoro** and Javier Ruiz-Hidalgo. SkinningNet: Two-Stream Graph Convolutional Neural Network for Skinning Prediction of Synthetic Characters. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

[110] Machbah Uddin. Scene classification using localized histogram of oriented gradients method. *International Journal of Computer (IJC)*, 20:13–18, 01 2016.

[111] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *Proceedings of the International Conference on Learning Representations*,

2018.

[112] Nitika Verma, Edmond Boyer, and Jakob Verbeek. Feastnet: Feature-steered graph convolutions for 3d shape analysis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2598–2606, 2018.

[113] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2015.

[114] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The world wide web conference*, pages 2022–2032, 2019.

[115] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *ACM Trans. Graph.*, 38(5):1–12, 2019.

[116] Rich Wareham and Joan Lasenby. Bone glow: An improved method for the assignment of weights for mesh deformation. In *Proceedings of the International Conference on Articulated Motion and Deformable Objects*, 2008.

[117] Rich Wareham and Joan Lasenby. Bone glow: An improved method for the assignment of weights for mesh deformation. In *Proceedings the International Conference on Articulated Motion and Deformable Objects*, 2008.

[118] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the International conference on machine learning*, 2015.

[119] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[120] Zhan Xu, Yang Zhou, Evangelos Kalogerakis, Chris Landreth, and Karan Singh. RigNet: Neural Rigging for Articulated Characters. *ACM Transactions on Graphics*, 39, 2020.

[121] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the International conference on machine learning*, pages 40–48. PMLR, 2016.

[122] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. *Proceedings the Advances in neural information processing systems Conference*, 32, 2019.

[123] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris N Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2017.

[124] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Proceedings the Advances in neural information processing systems Conference*, 31, 2018.

[125] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. Labeling trick: A theory of using graph neural networks for multi-node representation learning. *Proceedings the Advances in neural information processing systems Conference*, 34:9061–9073, 2021.

[126] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2881–2890, 2017.

[127] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. *Proceedings the Advances in neural information processing systems Conference*, 27, 2014.

[128] H. Zhu, J. Weibel, and S. Lu. Discriminative multi-modal feature fusion for rgbd indoor scene recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2969–2976, 2016.