The Iby and Aladar Fleischman
Faculty of Engineering
Tel Aviv University

QuadMeshCNN

A Project
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science

by
Omri Levi
I.D. XXXXXXXXX

TEL AVIV UNIVERSITY אוניברסיטת תל אביב

December 2021

Table of contents

Abstract

In recent years, convolutional neural networks (CNNs) have shown an outstanding performance in various of tasks in computer vision. Most of CNNs use 2D images as an input, and usually combine convolution, non-linearity, and pooling layers in order to achieve a robust performance to input variations. However, these CNN operations are suitable to the discrete regular grid of the 2D images and adjusting them to an unstructured data would be a complex task.

Mesh is an efficient and flexible representation for 3D shapes, which takes an advantage of its non-uniform structure to represent both large and flat regions and small and sharp regions. This non-uniform structure is also a drawback, where classic CNNs are hard to adjust for mesh processing. In this context, MeshCNN [Hanocka 2019], which is a novel approach for processing 3D shapes, solves this specific issue by incorporating specialized convolution and pooling layers directly on the unstructured data.

MeshCNN provides a great solution for triangular meshes, which are one of the most commonly used polygon mesh types for surface representation. However, it does not provide a solution for quadrilateral meshes, another commonly used polygon mesh types, which are usually used in many graphic applications and have some advantages over triangular meshes.

In this work we present our method QuadMeshCNN for pure-quad mesh data structure, which is an extension of MeshCNN and works directly on the quad-mesh structure. Specifically, we propose an algorithm for quadratic mesh pooling, based on local quad-mesh simplification operations. Moreover, we present our new quad-mesh classification dataset to demonstrate our method performance, achieving 95% average accuracy rate.

Chapter 1: Introduction

The world around us is a three dimensions world and 3D objects can be found everywhere. These objects play an important role in the field of computer graphics, and also act as a major research topic in computer vision and computational geometry.

3D objects can be represented in many ways; A representation that relies on a discrete approximation of the 3D shape is mostly used for data processing and computational reasons. The polygonal mesh is an important representation with numerous applications in geometric modeling, computer graphics, mechanical engineering, simulation, and architecture. Using a set of 2D polygons in the 3D space, the mesh represents surfaces, which are the foundation of 3D shapes [Mario Botsch, Leif Kobbelt, Mark Pauly]. Meshes create efficient, non-uniform, and flexible representations of shapes: for large and simple surfaces, they require a small number of polygons, whereas for small and complex surfaces, they maintain fine details by allowing higher resolution of polygons.

Triangles and quadrilaterals are the most common polygon types used for surface representation. A pure triangle mesh and a pure quad mesh are meshes composed entirely of triangles or quadrilaterals, respectively. Despite the fact that triangle meshes are more common and the majority of research in geometry processing, the advantage of quad meshes is also well understood in many graphic applications.

Convolutional neural networks (CNNs) have shown outstanding performance in recent years for a variety of tasks in computer vision [Sermanet et al. 2014; Simonyan and Zisserman 2015; Chen et al. 2018]. These CNNs are usually used on 2D images, which have a discrete regular grid representation, however extending these CNN models to work with 3D shapes, which have an irregular structure, is not a trivial task.

MeshCNN [Hanocka 2019] is a convolutional neural network designed for triangular meshes. It is similar to classic CNNs, though it incorporates specialized convolution and pooling layers that exploit the mesh edges geometric properties.

This work aims to extend MeshCNN to process quad meshes inputs. Since the method is a direct derivative of MeshCNN, we call it QuadMeshCNN. In this introduction we will give some basic information regarding mesh representation, explain the motivation of this work and summarize our main contributions.

1.1. Terminology

Here we present a short and basic glossary of terms related to mesh representations and to quad mesh in particular.

1.1.1. Mesh

- **Mesh** - A polygonal mesh, or mesh for short, is a representation of 2D surfaces which build a 3D shape. A mesh can be defined by the pair (V, F) where $V = \{v_1, v_2 \dots\}$ is a set of vertex positions in $\mathbb{R}^3$, and F defines the connectivity, which is a set of sets of N vertices. Given (V, F), the mesh connectivity is also defined using edges E, a set of pairs of vertices.
- **Triangle Mesh** – a pure triangle mesh, is a mesh composed entirely of triangular polygons: as stated above it is defined by the pair (V, F), while F is a set of triplets of vertices.
- **Quad Mesh** – a pure quadrilateral mesh, is a mesh composed entirely of quadrilateral polygons: as stated above it is defined by pair (V, F), while F is a set of quartets of vertices.

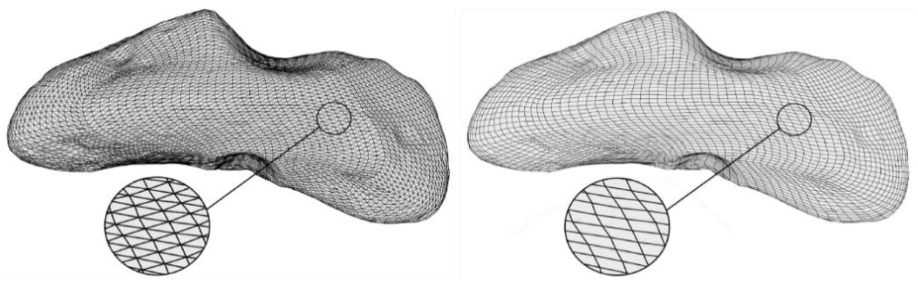

Figure 1 *- Triangle mesh vs. Quad mesh. Image from* [Thibeault et al. 2019]

1.1.2. Vertex, edge, and face

- **Vertex** – a position in the 3D space: $v = (x, y, z)$.
- **Edge** – a connection between two vertices: $e = (v_i, v_j)$
- **Face** – A closed set of edges: $f = (e_i, e_j, e_k \dots)$ or a closed set of vertices $f = (v_i, v_j, v_k \dots)$. A triangle face has three edges, and a quad face has four edges.

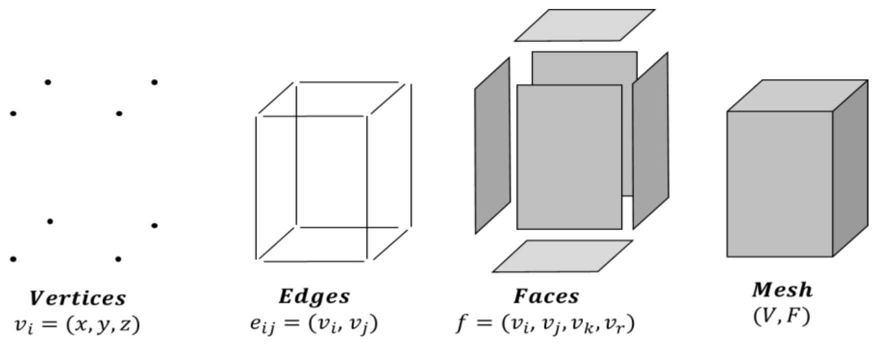

Figure 2 *– Illustration of quad mesh basics*

1.1.3. Normal vector

- **Surface normal** – for a convex polygon, a surface normal can be calculated as the vector cross product of two (non-parallel) edges of the polygon. For a plane given by the equation $ax + by + cz + d = 0$ the vector $n = (a, b, c)$ is normal.
- **Vertex normal** – this term is used in geometry of computer graphics. A normal attached to a vertex is computed as the normalized average of the surface normals of the faces that contain that vertex.

In this work we refer to normal as a surface normal.

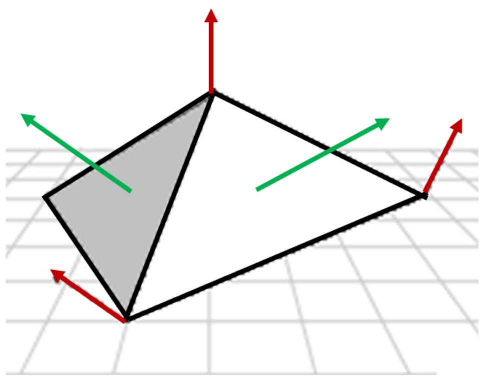

Figure 3 *- Surface normals (green) vs. vertex normals (red)*

1.1.4. Mesh geometry, topology, and validity

- **Mesh geometry -** specifies the position and other geometric characteristics of each vertex.
- **Mesh topology (also connectivity)** – describes the incidence relations among mesh elements, or in other words, how the mesh polygons are connected, ignoring the positions entirely.
- **Validity of polygonal mesh** – a set of constraints on the mesh structure in order to fulfill assumptions of algorithms which will operate on the mesh.
- **Manifold mesh** – a mesh is manifold if:
  (1) Each edge is incident to only one or two faces.
  (2) The faces incident to a vertex form a closed or an open fan.
- **Non-manifold mesh** – a mesh which is not fulfill one of the above conditions is considered as non-manifold.

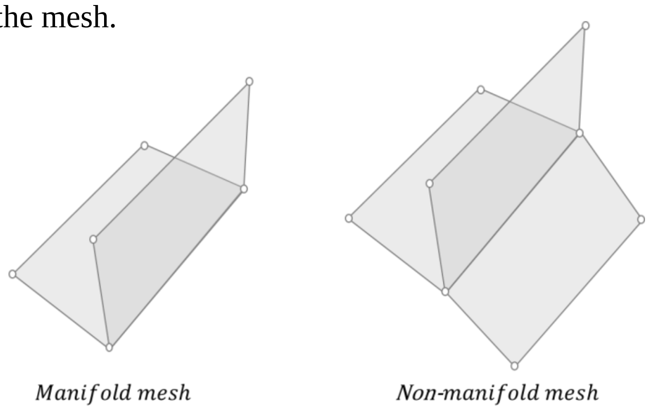

Figure 4 *- Manifold mesh vs. Non-manifold mesh examples*

- **Surface orientation** – this defines the order of the face vertices.

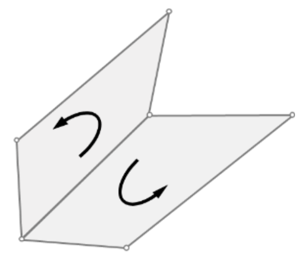

Figure 5 *- Mesh orientation illustration*

In this work we refer to pure quad mesh with the validity conditions as below:

1. A manifold mesh, possibly with boundary edges.
2. Consistent orientation of the surface: face faces ordered in counterclockwise order (figure [5]).
3. Non-self-intersecting surface.

1.1.5. Quad mesh classes

For quad meshes, an internal vertex in is called **regular** if it has valence 4 (**valence** of a vertex is the number of its incident edges). A vertex that is not regular is called irregular.

Quad meshes can be loosely divided into several classes depending on their regularity degree [Bommes et al. 2013]:

- **Regular mesh / geometry image** – a mesh which can be globally mapped to a rectangular subset of a square tiling.
- **Semi regular** – a mesh, which is obtained by gluing, in a conforming way, several regular 2D arrays of quads side to side. Each such regular sub-mesh is called a patch, and the number of patches is assumed to be much smaller than the total number of faces.
- **Valence semi regular** – a mesh which most of its vertices have valence 4.
- **Unstructured** – mesh which has a large fraction of irregular vertices.

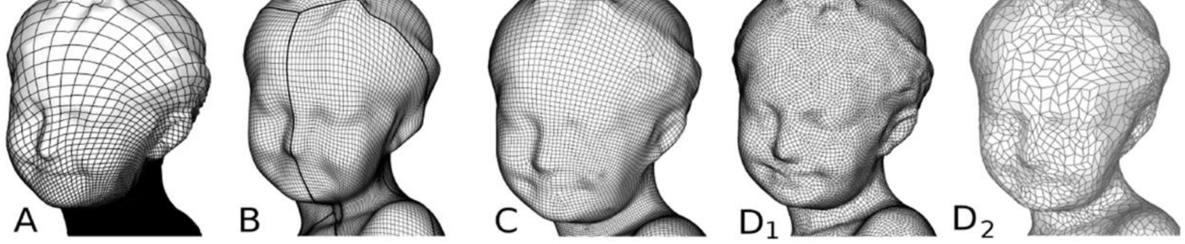

Figure 6 *- Quad meshes types: A - regular, B - semi-regular, C - valence semi-regular, D1-D2 - unstructured. Image from [Bommes et al. 2013].*

1.2. Motivation

Computer graphics has largely been dominated by triangle meshes, and there are large number of methods for preparing and processing triangle meshes in the literature. At the same time, quadrilateral meshes, particularly semi-regular ones, are advantageous for a variety of applications, and a significant progress has been made in quadrilateral mesh generation and processing in recent years.

In this section we will list briefly the advantageous of using quad mesh in the context of computer graphic application. We encourage the reader to review full explanations in [Bommes et al. 2013].

1.2.1. Quad mesh properties

The reasons to prefer quad meshes in many applications are:

a. **Two dominant local directions** – Geometry typically has two dominant local directions either associated with principal curvature directions or sharp features. While quad meshes can be easily aligned with these directions, triangle meshes will require a third edge direction to be chosen arbitrary.
b. **Surface representations** – Tensor-product high-order bases, associated with quad meshes are preferred for surface representations.
c. **Sampling pattern of textures** – Patches of semi-regular quad meshes naturally match the sampling pattern of all types of textures.

1.2.2. Quad mesh applications

The properties stated above are important in many common computer graphics applications:

a. **Polygonal modeling** – polygon meshes are used to represent objects and characters in real-time rendering. A carefully designed mesh must follow the line features of the represented shape in order to adjust to deformations caused by animation. In this context, quad meshes are more straightforward and easier to manipulate than triangle meshes.

b. **High-order surface modeling -** Semi-regular quad meshes are very useful as base meshes for fitting tensor-product techniques which are dominate some industrial applications e.g., CAD/CAM.
c. **Texturing** - Using semi-regular quad meshes for texturing is an excellent option, since each patch can be mapped to a rectangular texture.
d. **Finite element simulation** – As compared to triangles, quad meshes perform better in some numerical analysis, such as finite element modeling within highly elastic and plastic domains, because they reduce both the approximation error and the number of elements.
e. **Compression -** The ability to store high resolution shapes as a coarse quad mesh, with compressed displacement details on regular grids attached to quads, could greatly reduce storage space and transmission time.

1.3. Contributions

Work goal is to extend MeshCNN method to process pure-quad mesh inputs. In this context, our contributions can be summarized as following:

1. We extend MeshCNN to process pure-quad mesh inputs by changing some trivial parts (geometric features, mesh convolution, mesh un-pooling) and some non-trivial parts (mesh augmentations, mesh pooling) of the original MeshCNN method.
2. We created a new dataset, called QuadZoo5, which consist of 5 different animal classes each one with different mesh deformations and orientations.
3. We apply our QuadMeshCNN method for a classification task on the new dataset QuadZoo5 and got an average accuracy of 95%.

Chapter 2: Previous work

Deep learning approach has been used extensively on 2D images for various of computer vision tasks. Convolutional neural networks (CNNs) are used and achieved a great success in tasks such as classification and segmentation of 2D images [Sermanet et al. 2014; Simonyan and Zisserman 2015; Chen et al. 2018]. However, the use of these methods on 3D data is not trivial at all since they are suited to 1D data or 2D images with structured data representation. In this chapter we review some common 3D representations while focusing on the mesh data representation and MeshCNN method for triangle mesh processing. In addition, we review some of the common methods of quad mesh simplification as it is one of the key aspects of QuadMeshCNN method for quad mesh processing.

2.1. <u>3D representations and processing methods</u>

In order for deep learning models to be effective, it is necessary to consider the fundamental properties of 3D data representations, as well as their efficiency, simplicity, and usability. We briefly describe some of these representations here and refer the reader to [Gezawa et al. 2020] for a more comprehensive review.

a. **<u>Multi-view data</u>**

   A natural way to represent 3D structures would be to render a number of images from different points of view, then pile them as an input to a CNN. This kind of basic idea was used for different tasks such as shape classification [Su et al. 2015] and shape segmentation [Kalogerakis et al. 2017]. On the one hand, this kind of data representation has the advantage of being able to handle high-resolution inputs as well as using image based CNNs. On the other hand, such methods have several limitations such as self-occlusions, determining the number of views, which can result in significant computational overhead.

b. **<u>Solids</u> (Octree, Voxels)**

   Solids representations of 3D models are basically a space control information of an object. The two commonly used solids in the deep learning community are voxels and octrees.

   - Voxels – A voxel is a representation of a 3D object, showing how the object is distributed in the 3D space. A binary voxel form is a way to transform 3D object by dividing the 3D space into a 3D grid and represent the object in a grid-based representation, similar to 2D grid of an image. This similarity encourage to extend operations that applied on 2D grid to 3D grid such as 3D convolutions over the sampled data [Wu et al. 2015]. The advantage of this representation is it provides a full volumetric representation of the 3D shape. However, representing both non-occupied and occupied areas results in a huge memory demand.

- Octree – an extension of a 2D quadtree, where each node in it contains 8 children. The advantage of such a data structure is its efficient memory consumption, as well as its ability to generate high resolution voxels. The drawback is the inability of some 3D objects to maintain their geometry under such a data structure.

c. <u>High level structures (3D descriptors and graphs)</u>

A 3D object must be represented in a succinct, yet very rich way in order to serve in 3D shape retrieval and classification. 3D shape descriptors and a graph structure can be used for this representation.

- 3D descriptors – a way to convert 3D shapes into feature descriptors, which will be suited to shape retrieval or classification tasks, describing the topological characteristics of a 3D shape. 3D descriptors are mostly handcrafted and do not learn enough discriminative features which leads to incomplete description.
- Graph – a generalization of grid-based representations. By linking different shape parts, a graph represents the geometric essence of a 3D object.

d. <u>Point cloud</u>

A set of unstructured 3D data points residing on an object's boundary, where the points represented by three coordinates in a Cartesian (or other coordinate system), is called point cloud. Even though creating a point cloud for a 3D object is easy using scanning devices, there are some issues with it such as noise and data incompleteness. Additionally, processing them is challenging due to the lack of connectivity information. Therefore, using neural networks for point clouds processing force the learned model to be permutation invariant. The first work introduced a permutation invariant model for geometric learning of point clouds was PointNet [Qi et al. 2017], which uses 1x1 convolutions followed by global max pooling.

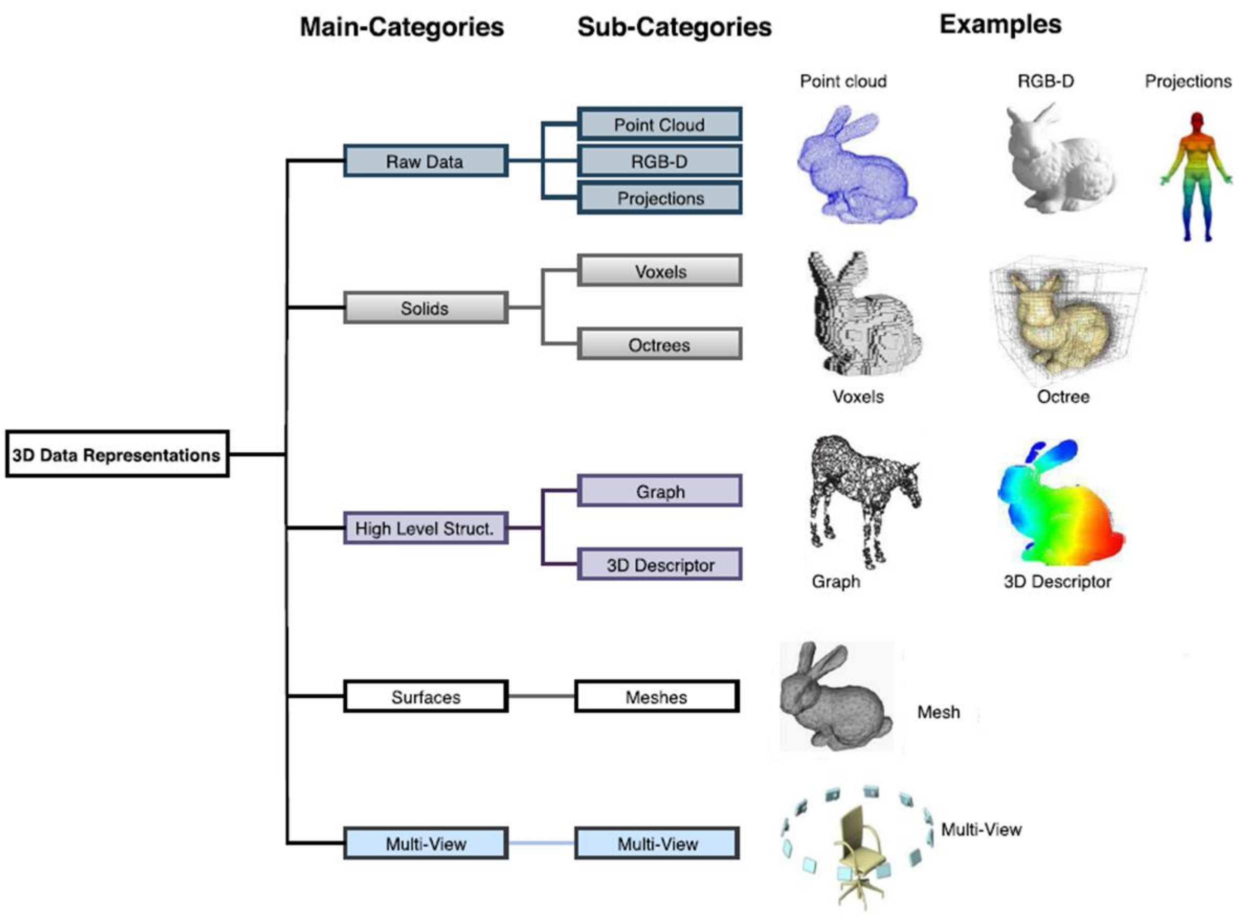

Figure 7 *- different 3D data representations.*
Image from [Gezawa et al. 2020]

e. **Surfaces (mesh)**

Surface polygons are often used to represent the boundary of 3D objects. There are various of methods for surface representations, but polygon mesh is the most popular in the deep learning community.

Polygon meshes can be easily created using 3D software packages (e.g., Maya, Blender, etc.). They consist of a combination of vertices, edges, and faces. On the one hand, since they have a complex and irregular structure, they can represent simple surfaces with a small number of polygons and can also represent small fine-detailed surfaces using a large number of polygons. On the other hand, the irregularity and complexity of this data structure make it difficult to adjust CNNs to handle them.

GCNN was the first method to exploit mesh connectivity structures introduced by [Masci et al. 2015]. Many approaches have been proposed since then to address the irregularity of the mesh structure and sampling rate, such as sampling the neighborhood of each vertex uniformly, or achieving regularity through spectral decomposition [Boscaini et al. 2016; Gong et al. 2019; Lim et al. 2019; Poulenard and Ovsjanikov 2018; Schult et al. 2020; Verma et al. 2018; Zhou et al. 2020]. [Feng et al. 2019] proposed a work called MeshNet that addresses the complexity and irregularity issues in mesh data and successfully performs 3D shape classification and retrieval using mesh data. The innovative work, MeshCNN [Hanocka 2019], proposed novel convolution and pooling operations that exploited the mesh structure to produce a rotation invariant, simple, yet highly accurate shape analysis method. We base our work on MeshCNN and extend it to process pure quad meshes inputs.

2.2. MeshCNN

As already said, MeshCNN is a novel approach which leverage the mesh properties using convolution, pooling and un-pooling operations, similar to 2D-input CNNs, directly on the mesh structure without any conversion to regular or uniform format. Since we base our work on MeshCNN, we will review in this section the MeshCNN operations and its guidelines.

2.2.1. Input features

Each edge in MeshCNN method has a set of geometric features that are invariant to translation, rotation, and uniform scale. In figure [8] you can see the geometric features: the dihedral angle, two inner angles, and two edge-length ratios. By sorting the two face-based features, the order ambiguity is resolved, providing invariance. In Chapter 3 we will define mathematically the geometric features for a quad mesh.

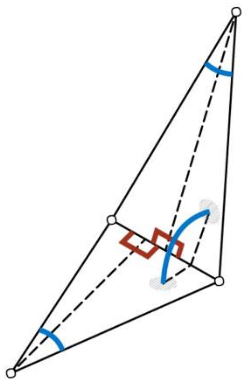

Figure 9 *– Illustration of edge input features. Image from [Hanocka 2019].*

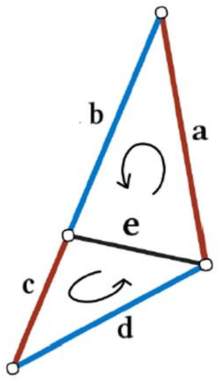

Figure 8 *-Illustration of triangle mesh edges orientation*

2.2.2. Mesh convolution

In a manifold triangle mesh, every edge has exactly four neighbors, so the convolution operation is defined on the mesh edges. It operates on the incident edge features and in order to create an order-invariant operation it uses simple similarity transformations on ambiguous edges (a and c, b and d). Mathematically, the convolution on edge e is defined as follows:

$$f_e \cdot k_0 + \sum_{j=1}^{4} f_e^j \cdot k_j \tag{1}$$

$$(f_e^1, f_e^2, f_e^3, f_e^4) = (|a - c|, a + c, |b - d|, b + d) \tag{2}$$

where f_e^j is the feature of the j^{th} convolutional neighbor of e.

2.2.3. Mesh Pooling

Mesh pooling operator is an important layer in MeshCNN method and is based on classic mesh simplification processing technique called edge collapse [Hoppe 1997]. Edge collapse is illustrated in figure [10]: edge e collapses to a point, resulting in the merging of the four incident edges ((a, b) and (c, d)) into two merged (p and q) edges. The two merged edge features are the average features of the collapsed edges as illustrated in figure [10].

In contrary to mesh simplification methods which aim to reduce the number of mesh elements while minimizing the geometry distortion, in MeshCNN the pooling operator aim to reduce the feature maps of the mesh and prioritize the edge collapses according to edge feature norms (start with smallest norm). By doing so, certain regions that are less significant to the learning process can be collapsed in a non-uniform way. Moreover, it offers flexibility in the output dimensions of the pooling layer, as well as robustness to the initial mesh tessellation.

2.2.4. Mesh Unpooling

Combined with Pooling layer, Mesh Unpooling layer serves as a partial-inverse operation. While the pooling layer decreases the resolution of feature activations, the unpooling layer increases resolution based on the history of merge operations collected by the pooling layer. By expanding back the edges, which were merged during the pooling operation, the unpooling layer can increase the activation of the features. Although it does not have any learnable parameters, it is usually combined with convolutions to recover the lost original resolution during the pooling process. In order to calculate the unpooled edge features they use a retained graph of adjacencies from the paired pooling layer. An unpooled edge feature is a weighted combination of the pooled edge features (see figure [10]).

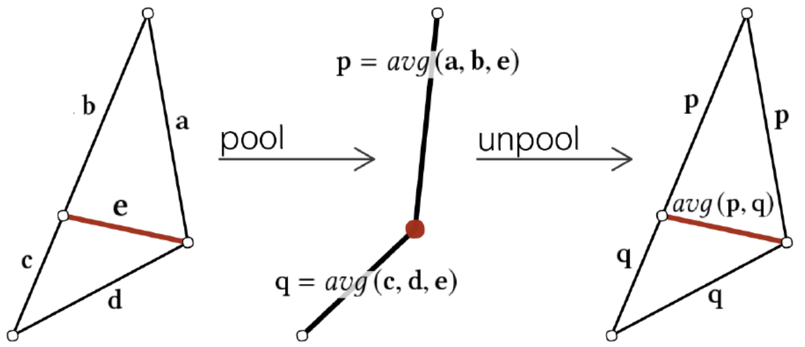

Figure 10 *– Mesh pooling and mesh unpooling. Feature aggregation is illustrated for the pooling and unpooling operations. Image from [Hanocka 2019].*

2.3. <u>Quad mesh simplification methods</u>

Simplification methods strive to reduce the number of elements in an input mesh, while keeping the level of introduced error low and the quality of the mesh high. As previously mentioned, in our work, we seek to minimize the number of features activations of quad meshes, while preserving their purely quad structure, but without considering their structure quality. In this section we will review some quad mesh simplification methods.

Quad mesh simplification is significantly more complex than triangle mesh simplification due to the stronger global connectivity dependency. Moreover, quad meshes are prone to degenerate configurations, meaning that avoiding them requires an extra care. Consequently, quad mesh simplification methods cannot be considered as mature as triangle mesh simplification methods.

Most of the iterative quad mesh simplification methods rely on an atomic set of operations. These operations modifies the mesh connectivity aiming to reduce the number of elements or to improve the individual quad shape.

Quad mesh simplification can be divided into two kinds: local and non-local. In local methods, atomic operations only affect a small area around a quad or a vertex, whereas in non-local methods they can affect a larger number of elements and even the whole mesh. Local methods have the advantage of finer granularity but have a downside of creating more irregular vertices.

Non-local methods preserve the regularity of the mesh, but are less adaptive, and the granularity of simplification is harder to control.

2.3.1. Non-local methods

Poly-chord collapse is the most common non-local operation and used for example in [Daniels et al. 2008; Murdoch et al. 1997; Borden 2001]. Poly-chords are strips of consecutive quads that either self-connect in a loop or has both ends at border edges. In this method the entire poly-chord is collapse (by removing all its elements) and the quad mesh structure is preserved. However, poly-chords can be too long and can produce too coarse mesh. Some non-local methods [Daniels et al. 2008; Shepherd et al. 2010] integrate local operations with poly-chords collapse.

2.3.2. Local methods

In our work, we find the local methods to be more suitable for mesh pooling operation. Each of the local methods define its own atomic operations. [Bommes et al. 2013] group these operations into three classes:

- Coarsening operations – will reduce the number of elements in the mesh structure.
- Optimizing operations – will only change the local connectivity of the mesh structure without changing the number of elements.
- Cleaning operations – will resolve an invalid local configuration and thus will reduce the number of elements and change the local connectivity.

Localized Quadrilateral Coarsening [Daniels et al. 2009] breaks the poly-chord collapse into a sequence of simpler atomic operations where each remove only one element of the poly-chord.

In our work we preferred to adapt the operations defined in Practical Quad Mesh Simplification [Tarini et al. 2010]. In this method they defined more operations from the types listed above, which preserves the quad mesh structure and particularly to maximize homeometry. Homeometry refers to the equal size of all edges and proportional length of all diagonals, which, to some extent, implies most other requirements: flatness, squareness, regularity, isometry, and at least some correspondence between vertex values and gaussian curvature. We will further describe the operations and how did we used them in our pooling layer in chapter 3 section 3.2.2.

Chapter 3: QuadMeshCNN – MeshCNN for quadratic mesh polygons

Our goal in this chapter is to describe in detail how we changed the MeshCNN method in order to process quad meshes, creating QuadMeshCNN method. We will describe what are the input geometric features of a quad mesh, how we changed the basic neural network layers, and what kind of data augmentations can be applied on the mesh structure.

3.1. Geometric features

As in other works, all mesh elements can hold various of features. Following MeshCNN for triangle meshes, we further extended the geometric features for each edge e in the mesh. The geometric features are the initial features of the input quadratic mesh and develop a higher abstraction as they progress through the network layers.

Let A and B be two planes with a common edge e, the input feature of the edge e is a 7-dimensional vector and consists of the following: the dihedral angle between the planes, the four inner opposite angles and the two diagonal connections ratios (figure [11]).

A. Dihedral angle

A dihedral angle is the angle between two intersecting planes A and B. Using Cartesian coordinates, we can define two planes according to the following term:

$$a_A x + b_A y + c_A z + d_A = 0$$

$$a_B x + b_B y + c_B z + d_B = 0 \tag{3}$$

The dihedral angle $0 \leq \varphi_{AB} \leq \pi/2$ between them is given by:

$$\varphi_{AB} = arccos\left(\frac{|a_A a_B + b_A b_B + c_A c_B|}{\sqrt{a_A^2 + b_A^2 + c_A^2}\sqrt{a_B^2 + b_B^2 + c_B^2}}\right) \tag{4}$$

Alternatively, one can calculate the dihedral angle using the normal vectors of the planes:

$$\varphi_{AB} = arccos\left(\frac{|n_A \cdot n_B|}{|n_A||n_B|}\right) \tag{5}$$

while $(\cdot)$ is the dot product between the normal n_A, n_B.

B. Opposite angles

For each plane, the two opposite angles are the angles between two edges from the three uncommon edges of the two planes. For the two planes, we get 4 opposite angles: $\alpha_A, \beta_A, \alpha_B, \beta_B$ as demonstrated in figure [11].

C. Diagonal connections ratio

For each plane, the diagonal connections ratio is the ratio between the length of the two diagonal connections in the quadratic plane d_1, d_2. For the two planes, we get 2 diagonal connections ratios:

$$r_A = \frac{|d_{A,1}|}{|d_{A,2}|},$$
$$r_B = \frac{|d_{B,1}|}{|d_{B,2}|} \tag{6}$$

D. Edge feature vector assembly

Following the above notations, for each edge e we will define the feature vector as a concatenation of the geometric features:

$$f_e = [\varphi_{AB}, \alpha_A, \beta_A, \alpha_B, \beta_B, r_A, r_B\,]^T \tag{7}$$

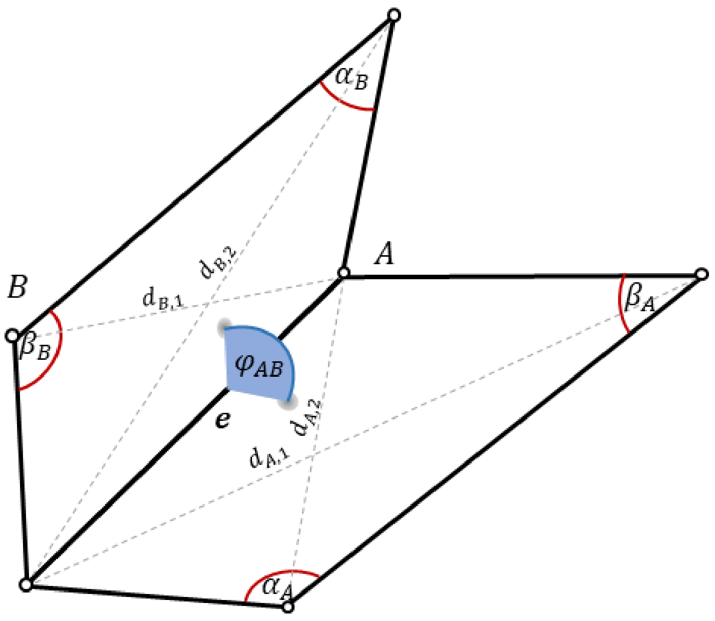

Figure 11 *- Geometric edge features for quad mesh*

3.2. Basic Neural Network layers

We define 3 basic NN layers for quadratic mesh inputs: convolution, pooling and un-pooling layers. The convolution layer and the un-pooling layer defined as an extension to the original definition from MeshCNN [Hanocka 2019] and the pooling layer was designed based on practical quad mesh simplification operations [Tarini et al. 2010].

3.2.1. Mesh Convolution

A convolution operator is a dot product between a kernel k and neighborhood elements. The mesh convolution is defined for the edges of a mesh, where the spatial support is defined using the edge itself and six incident neighbors. Thus, the convolution for an edge e with a feature attribute f_e is:

$$f_e \cdot k_0 + \sum_{j=1}^{6} f_e^j \cdot k_j \tag{8}$$

where f_e^j is the feature of the j^{th} convolutional neighbor of e.

As mentioned in MeshCNN, there is an ambiguity of the neighbors ordering (figure [12]), i.e., $(f_e^1, f_e^2, f_e^3, f_e^4, f_e^5, f_e^6)$ are either (a, b, c, d, f, g) or (d, f, g, a, b, c). To create an invariant convolution operator to the ordering of the input data we apply a set of symmetric functions. In our setting, the receptive field of an edge e is given by:

$$(f_e^1, f_e^2, f_e^3, f_e^4, f_e^5, f_e^6) = (|a-d|, a+d, |b-f|, b+f, |c-g|, g+c\,) \tag{9}$$

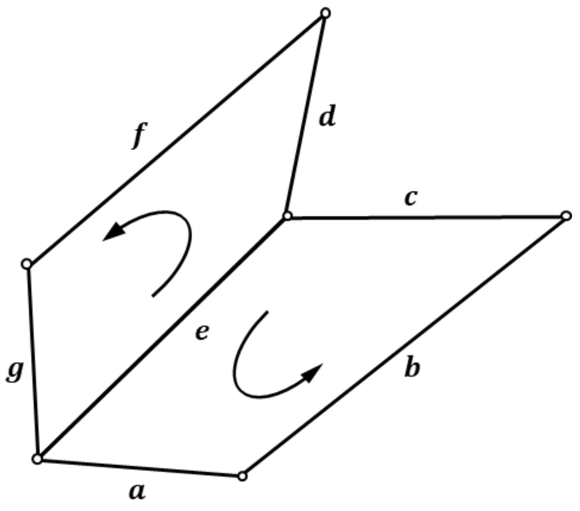

Figure 12 *- Quad mesh orientation. For edge e, there is an ambiguity in neighbors ordering.*

Another implementation of convolution of multi-channel tensor is by expanding the input tensor into a column matrix called general matrix multiplication (GEMM). MeshCNN uses similar method to perform the convolution operation in an efficient manner. In practice we aggregate all edge feature into $n_c x n_e x 7$ feature tensor, where n_c is the number of feature channels, n_e is the number of edges and 7 is the number of edges in the convolution spatial support. Finally, this matrix is multiplied by a matrix of weights of the convolutions.

3.2.2. <u>Mesh Pooling</u>

MeshCNN defines the mesh pooling operator as a series of edge collapse operations, where each such edge collapse converts five edges (a, b, c, d, e) into two edges (p, q) (see Figure [13]). An attempt to extend this method and applying it on a quad mesh will produce non-pure quad mesh, which we consider in this work as an invalid configuration. Figure [14] below illustrates such an attempt:

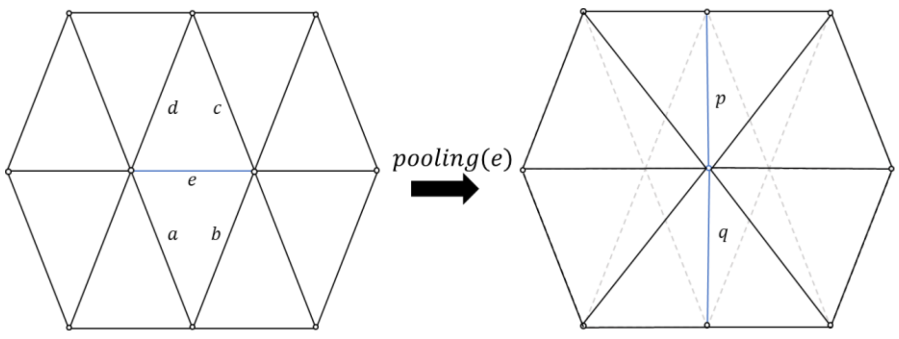

Figure 13 *– edge collapse illustration in the triangle case.*

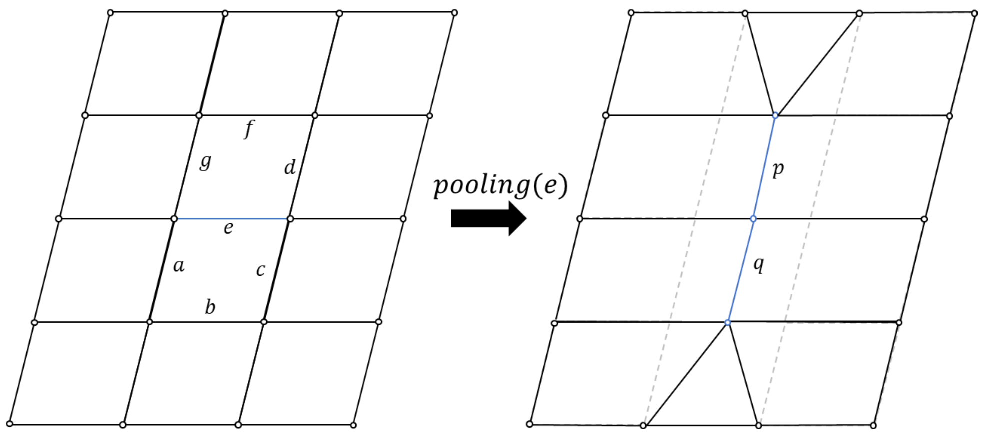

Figure 14 *– An attempt to apply edge collapse method according to the triangle case in the quad case. Such an attempt, results in breaking the pure-quad mesh structure.*

Many other methods which based on the same edge collapse will fail and produce non-pure quadratic mesh.

Practical quad mesh simplification

[Tarini et al. 2010] suggests an approach for quad mesh simplification, i.e., producing a low-resolution quad mesh starting from a high resolution one. Their approach includes local operations only which preserves the quadratic structure.

They define three kinds of local operations: optimizing operations, coarsening operations and cleaning operations (see figure [15]).

- Optimizing operations: These operations will only change the local connectivity without changing the number of elements and includes:
  - *Edge rotate* – let *e* be a non-border edge shared by two quads. Dissolving this edge will produce a hexagonal face, which can be split again into a pair of quads with two possible ways: clockwise and counterclockwise.
  - *Vertex rotate* – let v be a non-border vertex, split each of the k quads sharing v into 2k triangles using their diagonals. Merge the 2k triangles into k quads while the diagonals used to split the original quad will be the edges of the new quads. This operation can be considered as edge rotation around the vertex v.
- Coarsening operations: These operations will reduce the number of elements and includes:
  - *Diagonal collapse* – let *q* be a quad, it can be collapsed on either diagonal by merging the two vertices of the diagonal into one vertex. By applying diagonal collapse the mesh structure reduce one quad, two edges and one vertex, while preserving the quad mesh structure.
  - *Edge collapse* – let *e* be a non-border edge, perform a vertex rotation which turns *e* into a quad diagonal and collapse it.
- Cleaning operations: These operations will resolve an invalid local configuration and thus will reduce the number of elements and change the local connectivity. These operations includes:
  - *Doublet removal* – doublet is a configuration where two adjacent quads shared two consecutive edges. A doublet removal is done by removing the vertex in the middle and merging the two quads into a one single quad.
  - *Singlet removal* – singlet is a configuration where a quad is folded such that two consecutive edges become coincident. A singlet removal is done by removing the degenerate quad and substituting it with an edge.

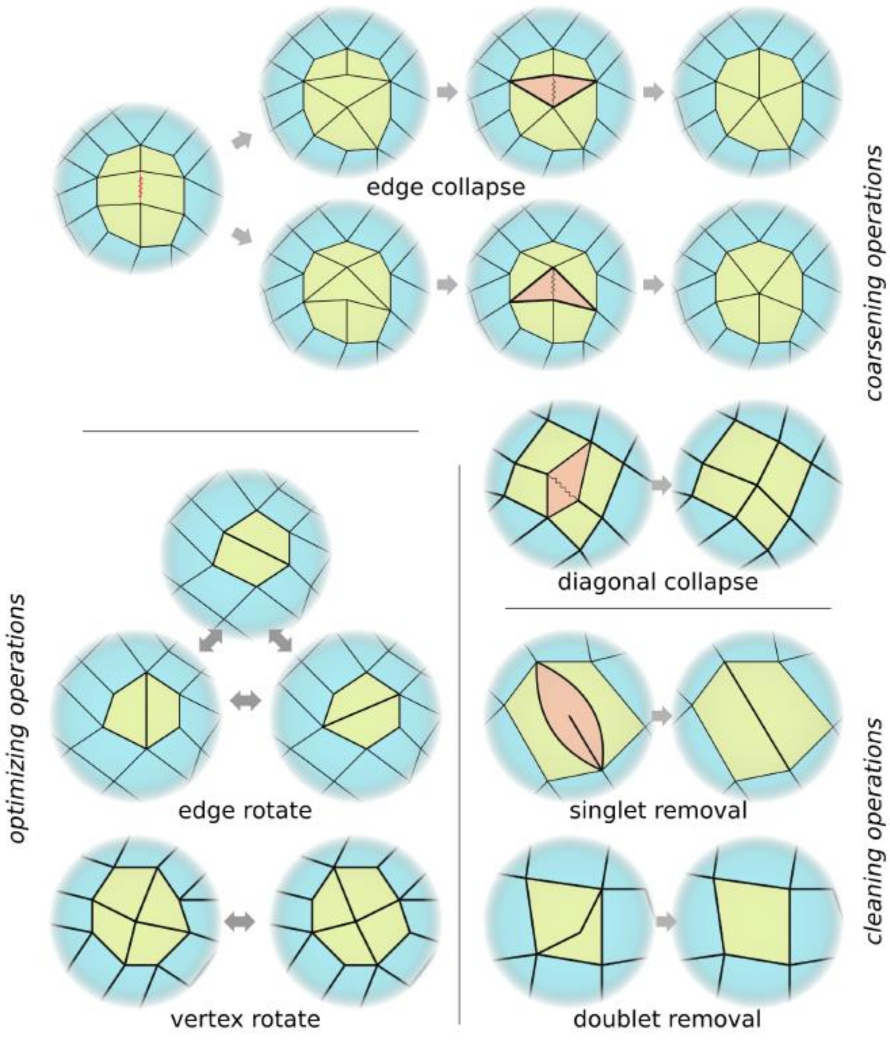

***Figure 15** – Set of local operations.*
Image from [Tarini et al. 2010].

<u>Mesh pooling algorithm</u>

In our work, we developed a mesh pooling algorithm that involves edge-collapse operations and cleaning operations based on [Tarini et al. 2010]. This algorithm preserves the quadratic mesh structure. Similar to MeshCNN for the triangle case, we prioritize the edge-collapse order by the magnitude of the edge features, while edges with smaller magnitude will be collapsed first. Since quad edge-collapse operations are relatively more complex than triangle edge-collapse operations, in some cases more edges will be collapsed during the process of another edge-collapse.

Mesh pooling is implemented as follows:

```
Algorithm 1 – mesh pooling

1. Build edges queue according to edge features magnitude.
2. While pooling_count < resolution_target do:
    a. edge_id = Queue.pop()
    b. If edge_id is not removed continue to (c), else return to (a)
    c. Run clear_doublets(mesh) and clear_singlets(mesh)
    d. If edge_id is on boundary or edge configuration non-valid return to (a), else
       continue to (e).
    e. Perform edge_collapse(edge_id) operator.
    f. Run clear_doublets(mesh)  and clear_singlets(mesh)
```

While:

- **pooling_count** is an integer number which updates during the process each time an edge is removed.
- **clear_doublets** and **clear_singlets** are cleaning operations. Since some degenerate configurations can be created in the edge-collapse process, the pooling algorithm begins and ends with cleaning operations.
- **Non-boundary edge** is an edge with 6 valid edge neighbors.
- **Non-valid edge** configuration is one of the following:
    - More than 2 shared items for its pair edge neighbors.
    - More than 4 vertices are shared from each side of the edge.

Edge-collapse algorithm is implemented as follows:

```
Algorithm 2 – quad edge-collapse

1. Extract edge_id neighborhood information.
2. Let (u, v) be the vertices of the edge_id. Select vertex u for step (3).
3. Perform vertex_rotation around vertex u.
4. Perform diagonal_collapse from vertex v.
```

Pooling features:

While running pooling algorithm at least 2 edges will be removed. We do not remove completely the features of the removed edge, but merge it with other edge features according to the operator and configuration:

1. Doublet removal – let e_1, e_2 be the two edge features of the doublet edges. It will be merged with the features of the edges (a, b, c, d) of the new quad according to the illustration below (figure [16]). "Merge" means an average of the features:

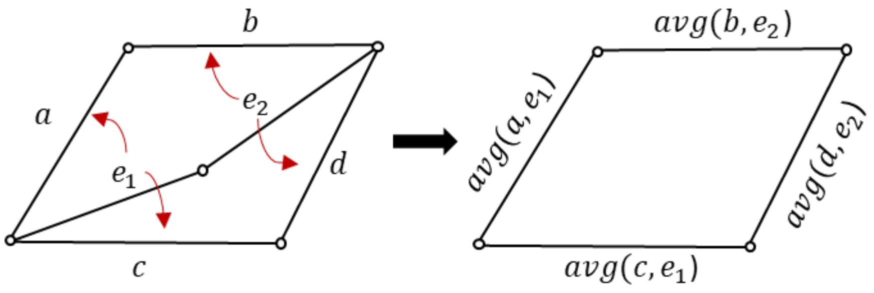

Figure 16 *– Feature aggregations during doublet removal operation.*

2. Diagonal collapse – let v_1, v_2 be vertices which will be merged to vertex v_{12}. The two edges a, b from v_1, which are in the same neighborhood of v_2, will be merged with the two edges c, d from v_2 (which are in the same neighborhood of v_1). The merged position vertex v_{12} is an average of the two vertices v_1, v_2 (figure [17]).

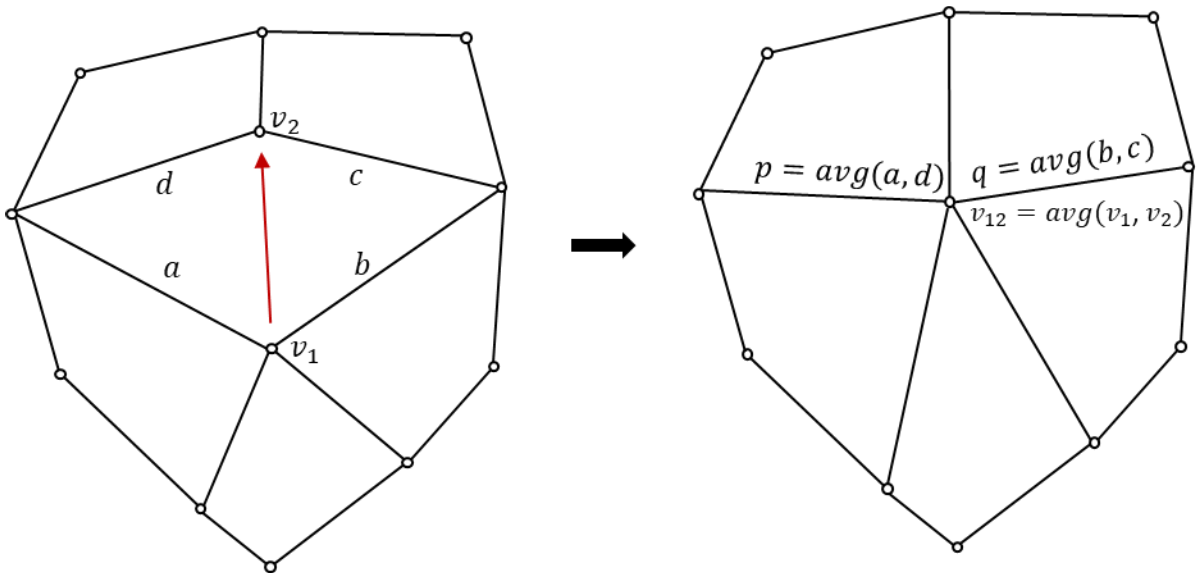

Figure 17 *- Features aggregation during diagonal collapse operation.*

3.2.3. Mesh Unpooling

Unpooling layer is used exactly the same as in the original MeshCNN for the triangle case. For explanation of this layer, we refer the reader to chapter 2 section 2.2.4.

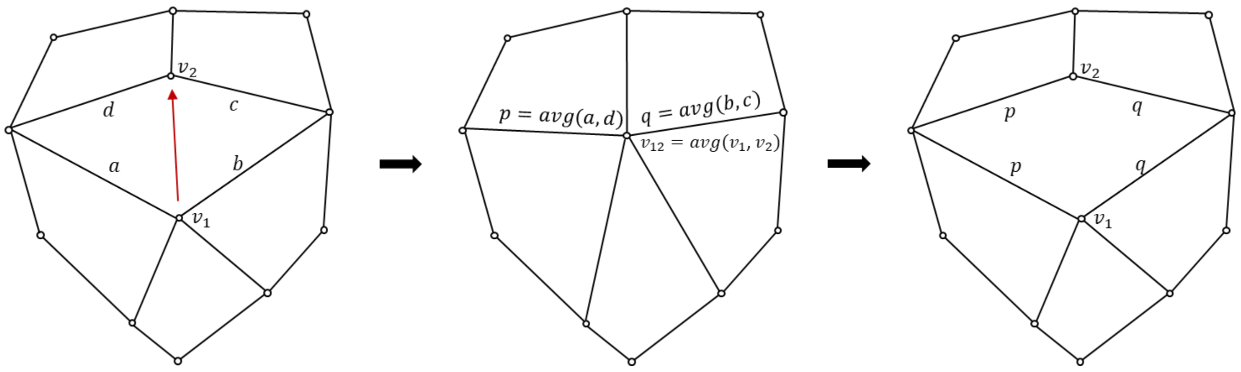

Figure 18 *- Feature aggregation during mesh pooling and unpooling.*

3.3. Quadratic mesh augmentations

Data augmentation is a technique used to increase the amount of training samples for the network. This technique reduces model overfitting to the training data since it acts as a regularizer.

Mesh augmentation is performed on the mesh structure, i.e., on the mesh vertices and edges, and includes the following operations: vertex position scaling, vertex translation, total mesh rotation, edge rotations and more.

As in MeshCNN for the triangle case, our input features are similarly invariant and therefore, applying rotation, translation and isotropic scaling will not generate new input features. In order to create new input features using augmentations we have used the following augmentations:

1. **Anisotropic scaling on vertex location**
   Let $S_x \sim N(\mu_x, \sigma_x^2)$, $S_y \sim N(\mu_y, \sigma_y^2)$, $S_z \sim N(\mu_z, \sigma_z^2)$ be one-dimension random variables, $v_p = (x, y, z)$ position of vertex v. One can apply an anisotropic scaling on v_p as follows:
   $$\widetilde{v_p} = (S_x x, S_y y, S_z z) \tag{10}$$
   This kind of vertex location scaling will change the input features since it changes the geometry of the quads in an anisotropic manner.

In our settings we used $\mu_x = \mu_y = \mu_z = 1$ and $\sigma_x^2 = \sigma_y^2 = \sigma_z^2 = 0.1$.

2. **Shift vertex location on mesh surface** – let $p \sim U[a,b]$ be a one-dimension random variable, e an edge with vertices positions $v_{p1} = (x_1, y_1, z_1)$, $v_{p2} = (x_2, y_2, z_2)$. One can apply vertex location shift on the mesh surface as follows:
$$\widetilde{v_p} = v_{p1} + p\left(v_{p2} - v_{p1}\right) \tag{11}$$
This kind of vertex shift can also be considered as vertex "slide" on the mesh surface and will change the input features.
In our setting we used $p \sim U[0.2,0.5]$.

3. **Edge-rotation** – One can perform edge-rotation (i.e., rotations around a vertex) which will change the mesh tessellation and therefore, will change the input features of the mesh.

The augmentations listed above were done on a small fraction of each mesh structure in order to create slightly different input features and train the network without overfitting.

Chapter 4: Dataset Generation

In this chapter, we will present our new quadratic mesh dataset for a classification task. As the best of our knowledge, pure-quadratic mesh classification datasets do not publicly exist, therefore, in order to demonstrate our QuadMeshCNN method for a classification task, we had to create a new dataset. We will describe how did we generated it using a commercially known algorithm and present a few examples from it.

4.1. Dataset generation

4.1.1. Generating quadratic mesh

There are several methods which can take a triangle mesh input and convert it into a quadratic mesh structure. Instant Field-Aligned Meshes by [Jakob et al. 2015] presented their method for remeshing a surface into an isotropic quad-dominant one by optimizing both edge orientations and vertex positions in the output mesh. Quadriflow [Huang et al. 2018] modified Instant Meshes algorithm such that it produces meshes with many fewer singularities. They achieved these results by combining Instant Meshes objective with a system of linear and quadratic constraints. These algorithms, however, have a big drawback when a coarse mesh is required, as geometric details are lost during the process.

Quad-Remesher [by exoside] is a commercial automatic quad-remeshing algorithm which takes as an input a mesh containing triangles or polygons and returns a mesh mainly consisting of quadrangles following natural directions. In this work we have used Quad-Remesher add-on for Blender [Blender Online Community]. In order to create a pure-quadratic mesh, we fix the output mesh manually using Blender, by combining triangles and changing mesh connectivity, in the cost of losing some fine-details of the mesh structure.

4.1.2. Generating object instances

Some of our classes in the dataset were generated from a single high-resolution triangle mesh object, therefore, in order to create different instances of the same object we had to deform the initial and intermediate mesh structures. Using Blender, some mesh sculpting operations were done on the mesh structures in order to create variations of the object. These operations includes smoothing, flatting, creasing, thickening, and rotating operations on different parts of the object. Additionally, a global structure rotation and an anisotropic scaling applied to increase the variability of the dataset.

Then, Quad-Remesher algorithm were used to create a quad-dominant mesh, but for each new object instance we have used different input settings for the algorithm in-order to create a different mesh structure. Last, as already said, we fixed the non-quad faces creating a pure-quad mesh structure.

Following the described process above we have created the **QuadZoo5** dataset which will be used to demonstrate our QuadMeshCNN method on a classification task.

4.2. Dataset details

QuadZoo5 dataset consist of 5 different animal classes: cow, horse, lion, dilophosaurus ("dilo") and spider. The first three classes were created from an initial high-resolution triangle mesh models from [Jakob et al. 2015] and the last two classes were created from SHREC11 [Lian et al. 2011] dataset. We used only 5 types of classes since this process of dataset generation is time consuming and we only want to have a proof-of-concept.

Following SHREC11 dataset used in MeshCNN, which consist of 16 train objects and 4 test objects for each class, we created 11 train objects and 4 test objects for each class, producing 75 pure-quadratic mesh objects in total. The number of edges in a mesh model varies from 1396 edges to 1954 edges. See Table [1] below for comparison between SHREC11 and QuadZoo5.

	# Classes	**# Train examples**	**# Test examples**	**# Edges input**	**Mesh type**
SHREC11	30	16	4	750	Triangles
QuadZoo5	5	11	4	[1396, 1954]	Quads

Table 1 *- Dataset comparison: SHREC11 vs. our new QuadZoo5 dataset*

Figure [19] in the next page presents some examples from QuadZoo5 dataset.

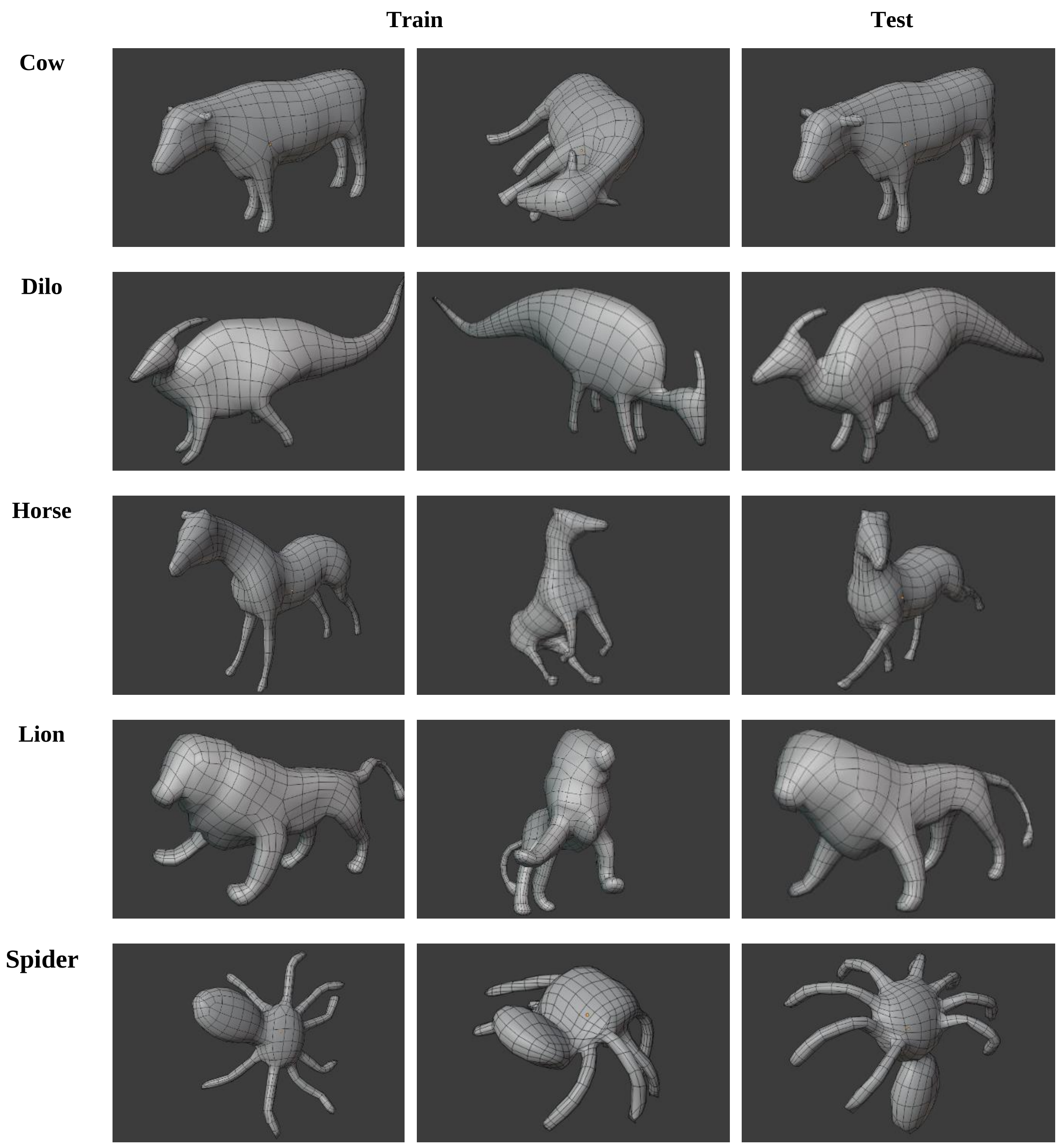

Figure 19 *- QuadZoo5 samples from train and test sets*

Chapter 5: Experiments and Results

5.1. Classification model

For classification task we used the classification model suggested in MeshCNN. Figure [20] below describes schematically the network architecture, while in Table [2] below we detail the network configuration.

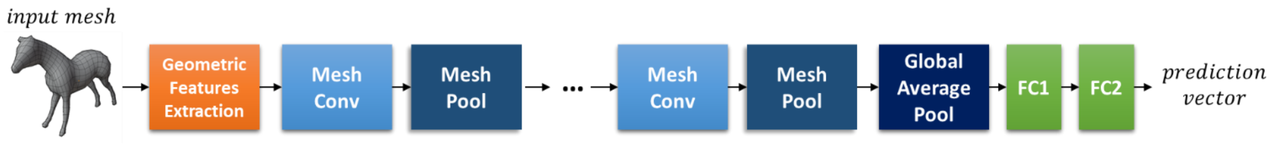

Figure 20 *- Classification model building blocks*

Layer	**Output**
Input mesh	-
Geometric features extraction	$n_e x7$ $n_e \in [1396, 1954]$
MeshConv_1 + ReLU	$n_e x64$
MeshPool_1 (out res. 1100)	$1100x64$
MeshConv_2 + ReLU	$1100x128$
MeshPool_2 (out res. 900)	$900x128$
MeshConv_3 + ReLU	$900x256$
MeshPool_3 (out res. 600)	$600x256$
MeshConv_4 + ReLU	$600x256$
MeshPool_4 (out res. 400)	$400x256$
Global Average Pool	$1x256$
FC_1 + ReLU	$1x100$
FC_2	$1x5$
Prediction vector	$1x5$
Classification (arg max)	Class number

Table 2 *– classification network configuration for QuadZoo5.*

5.2. Training

We have implemented QuadMeshCNN on top of MeshCNN project, i.e., using PyTorch framework. We used Adam optimizer with initial learning rate of $1e^{-3}$, batch size of 8, and used a scheduler with lambda rule according to:

$$lr\_l = 1 - \frac{max(0, epoch + 1 - niter)}{niter\_decay + 1} \quad (12)$$

with $niter = 350$ and $niter\_decay = 700$.

For the training data we have used 30 variants of data augmentations of anisotropic vertex scaling (10% of vertices in a mesh), shift for vertices on surface (10% of vertices in a mesh), and edge rotations (10% of edges in a mesh).

The training loss for the classification task is the basic loss function – cross entropy loss.

We have trained the model for 1050 epochs until convergence. Figure [21] below shows the training loss convergence.

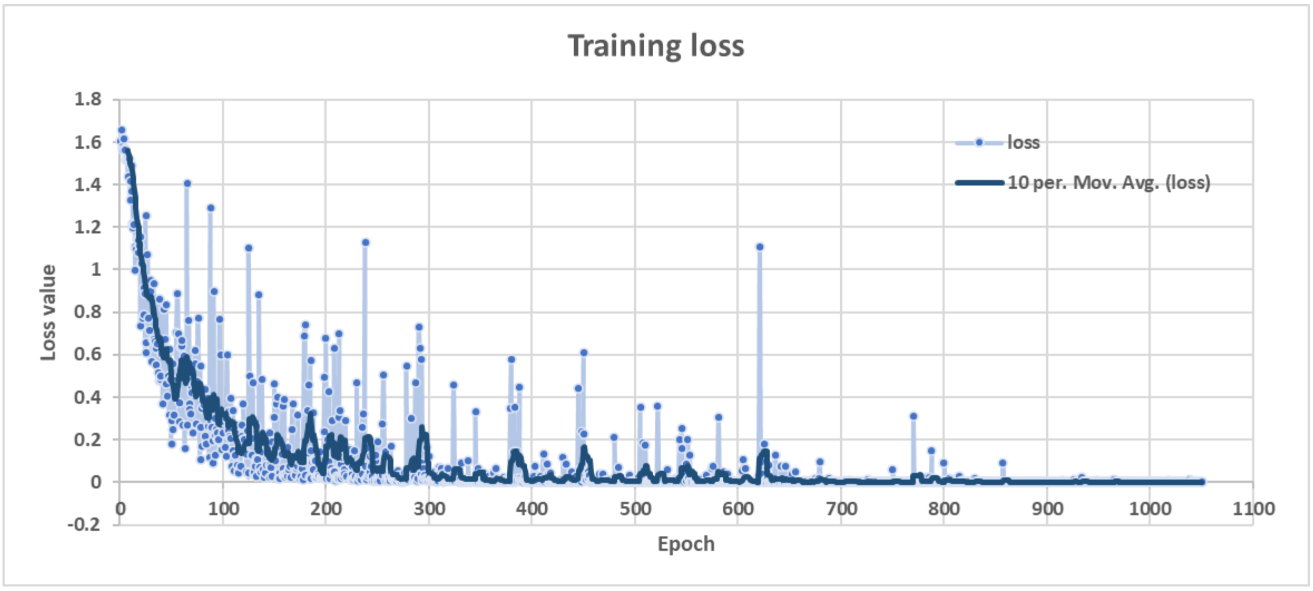

Figure 21 *– Training loss convergence of the suggested classification model on QuadZoo5.*

5.3. <u>Evaluation metrics</u>

To evaluate our model on the classification task we have used the basic metric of **average accuracy** measure. In the evaluation step we also used data augmentations as we have used in the training phase in-order to make sure our model is robust to some variants of mesh structure.

The **confusion matrix** is a summary of prediction results of a classification model on the test data. It summarizes the percentage of correct predictions for each class and incorrect predictions for each class.

5.4. Results

5.4.1. Quantitative results

Our classification model **average accuracy** on the QuadZoo5 dataset reached to **95%**. Table [3] details the averaged **confusion matrix** and figure [22] presents the average accuracy of the test set during training.

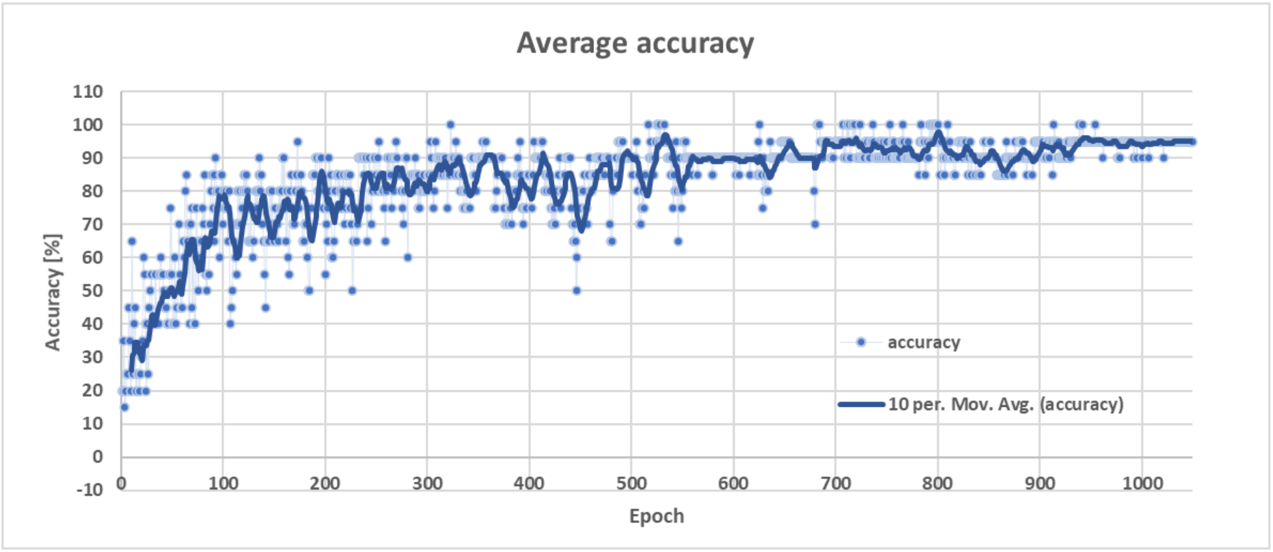

Figure 22 *- Average accuracy on the test set of the suggested classification model on QuadZoo5.*

	Predicted class					
	Class	**Cow**	**Dilo**	**Horse**	**Lion**	**Spider**
True class	**Cow**	75%	0%	0%	25%	0%
	Dilo	0%	100%	0%	0%	0%
	Horse	0%	0%	100%	0%	0%
	Lion	0%	0%	0%	100%	0%
	Spider	0%	0%	0%	0%	100%

Table 3 *- Confusion matrix of the classification model on the test set.*

5.4.2. Qualitative results – pooling results

Here we present a comparison of some input mesh examples and the last pooling outputs (figure [23]).

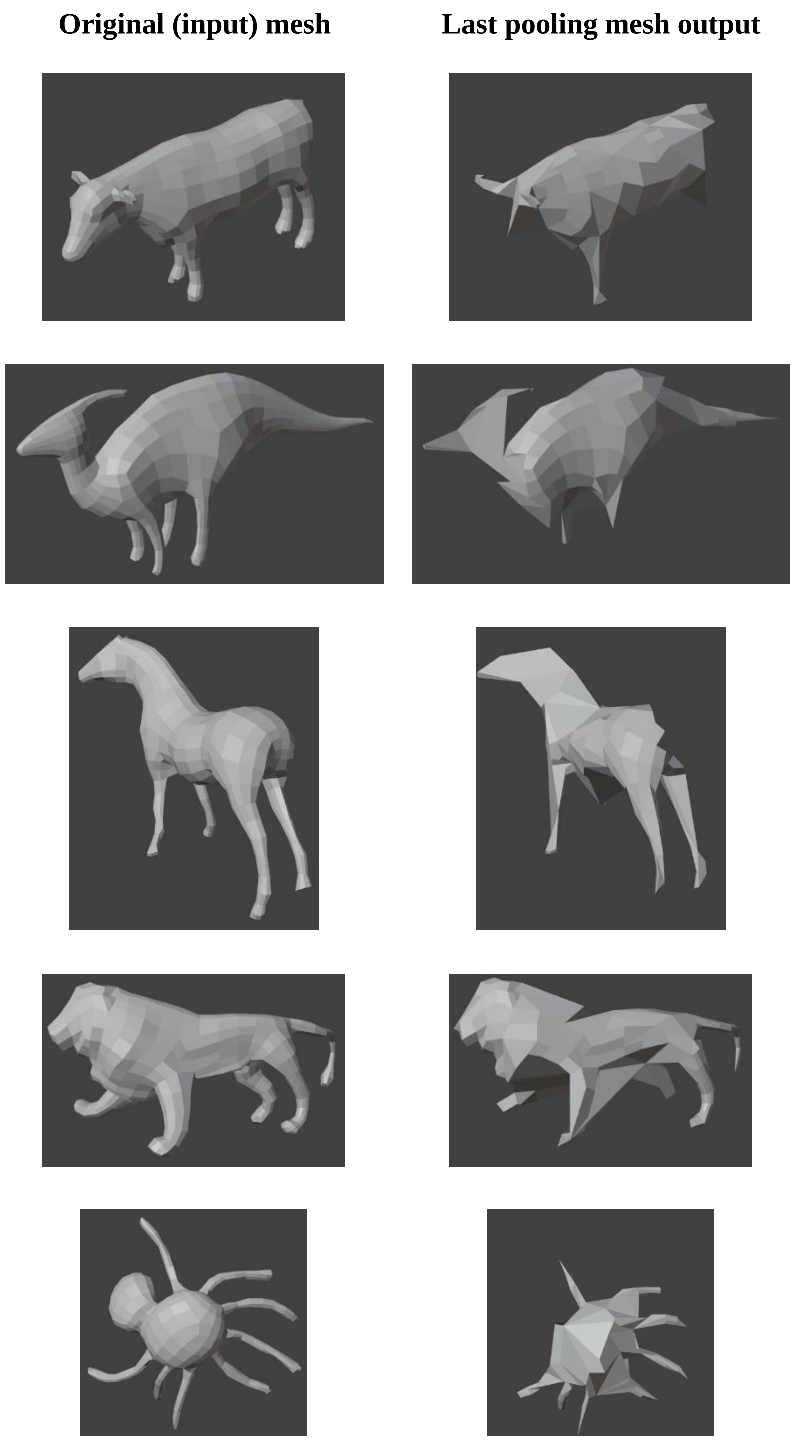

Figure 23 *- Input meshes vs. output meshes of the last pooling layer*

The network learns to preserve the most important edges of each shape in order to classify it correctly. One can observe that for the lion label it preserves the head structure while for the cow and horse labels it preserves the body. For the spider label it preserves the number of legs.

Chapter 6: Conclusions and Future work

6.1. Discussion and Conclusions

In this work we presented QuadMeshCNN, which is an extension of MeshCNN [Hanocka 2019] for pure-quad mesh inputs.

A key aspect of the QuadMeshCNN method is quad mesh pooling algorithm, which is based on Practical Quad Mesh Simplification [Tarini et al. 2010] operations. According to our experience, simplifying quad meshes is a much more complex process than simplifying triangle meshes, both in terms of computation and concept.

Moreover, we generated a pure-quad mesh classification dataset, QuadZoo5, to demonstrate our method, and we showed that a model based on our method achieved a 95% average accuracy rate.

6.2. Future work

Our first suggestion for future work refers to one limitation of our method - it can only be used for pure quad meshes. Pure quad meshes are difficult not only to create, but also to maintain once a simplification algorithm has been applied. We suggest combining both MeshCNN and QuadMeshCNN methods in such a way the combined method can handle triangles and quads at the same time.

Our second suggestion would be optimizing mesh pooling algorithm. Specifically, one can optimize the pooling layer by collapsing edges which do not affect each other in parallel. Another suggestion would be to implement the enhanced pooling layer suggested in [Barda et al. 2021] in order to improve model learning.

Another aspect of MeshCNN which was not included in our work is evaluating our method on mesh segmentation task. This will include generating new segmentation dataset our converting existing triangle mesh dataset into pure-quad mesh structure.

References

BARDA, A., EREL, Y., AND BERMANO, A.H. 2021. MeshCNN Fundamentals: Geometric Learning through a Reconstructable Representation. *1*, 1.

BLENDER ONLINE COMMUNITY. Blender - a 3D modelling and rendering package. http://www.blender.org.

BOMMES, D., LÉVY, B., PIETRONI, N., ET AL. 2013. Quad-mesh generation and processing: A survey. *Computer Graphics Forum 32*, 6, 51–76.

BORDEN, M.J. 2001. Sheet Insertion and Extraction for Mesh Generation. January 2017.

BOSCAINI, D., MASCI, J., RODOLÀ, E., AND BRONSTEIN, M. 2016. Learning shape correspondence with anisotropic convolutional neural networks. *Advances in Neural Information Processing Systems*, 3197–3205.

CHEN, L.C., PAPANDREOU, G., KOKKINOS, I., MURPHY, K., AND YUILLE, A.L. 2018. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence 40*, 4, 834–848.

DANIELS, J., SILVA, C.T., AND COHEN, E. 2009. Localized quadrilateral coarsening. *Computer Graphics Forum 28*, 5, 1437–1444.

DANIELS, J., SILVA, C.T., SHEPHERD, J., AND COHEN, E. 2008. Quadrilateral mesh simplification. *ACM Transactions on Graphics 27*, 5.

EXOSIDE. Quad Remesher. https://exoside.com/quadremesher/.

FENG, Y., FENG, Y., YOU, H., ZHAO, X., AND GAO, Y. 2019. MeshNet: Mesh neural network for 3D shape representation. *33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019 and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019*, 8279–8286.

GEZAWA, A.S., ZHANG, Y., WANG, Q., AND YUNQI, L. 2020. A review on deep learning approaches for 3d data representations in retrieval and classifications. *IEEE Access 8*, 57566–57593.

GONG, S., CHEN, L., BRONSTEIN, M., AND ZAFEIRIOU, S. 2019. SpiralNet++: A fast and highly efficient mesh convolution operator. *Proceedings - 2019 International Conference on Computer Vision Workshop, ICCVW 2019*, 4141–4148.

HANOCKA, R. 2019. MeshCNN: A network with an edge. *ACM Transactions on Graphics 38*, 4.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1997*, 189–198.

HUANG, J., ZHOU, Y., NIESSNER, M., SHEWCHUK, J.R., AND GUIBAS, L.J. 2018. QuadriFlow: A Scalable and Robust Method for Quadrangulation. *Eurographics Symposium on Geometry Processing 37*, 5.

JAKOB, W., TARINI, M., PANOZZO, D., AND SORKINE-HORNUNG, O. 2015. Instant Field-Aligned Meshes. *ACM Trans. Graph 34*, 6, 189–1.

KALOGERAKIS, E., AVERKIOU, M., MAJI, S., AND CHAUDHURI, S. 2017. 3D Shape segmentation with projective convolutional networks. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017 2017-Janua*, 6630–6639.

LIAN, Z., GODIL, A., BUSTOS, B., ET AL. 2011. Shape Retrieval on Non-rigit 3D Watertight Meshes. *Eurographics Workshop on 3D Object Retrieval (3DOR)*.

LIM, I., DIELEN, A., CAMPEN, M., AND KOBBELT, L. 2019. A simple approach to intrinsic correspondence learning on unstructured 3D meshes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 11131 LNCS*, 349–362.

MARIO BOTSCH, LEIF KOBBELT, MARK PAULY, P. Polygon Mesh Processing - Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, Bruno Levy - Google Books. https://books.google.com.au/books?hl=en&lr=&id=8zX-2VRqBAkC&oi=fnd&pg=PP1&dq=Polygon+Mesh+Processing&ots=Z_7R-jyW1t&sig=jZWDARomTRP4nylsoSUspT9GKvU#v=onepage&q=Polygon Mesh Processing&f=false.

MASCI, J., BOSCAINI, D., BRONSTEIN, M.M., AND VANDERGHEYNST, P. 2015. Geodesic Convolutional Neural Networks on Riemannian Manifolds. *Proceedings of the IEEE International Conference on Computer Vision 2015-Febru*, 832–840.

MURDOCH, P., BENZLEY, S., BLACKER, T., AND MITCHELL, S.A. 1997. The spatial twist continuum: A connectivity based method for representing all-hexahedral finite element meshes. *Finite Elements in Analysis and Design 28*, 2, 137–149.

POULENARD, A. AND OVSJANIKOV, M. 2018. Multi-directional geodesic neural networks via equivariant convolution. *SIGGRAPH Asia 2018 Technical Papers, SIGGRAPH Asia 2018 37*, 6.

QI, C.R., SU, H., MO, K., AND GUIBAS, L.J. 2017. PointNet: Deep learning on point sets for 3D classification and segmentation. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017 2017-Janua*, 77–85.

SCHULT, J., ENGELMANN, F., KONTOGIANNI, T., AND LEIBE, B. 2020. DualConvMesh-Net: Joint Geodesic and Euclidean Convolutions on 3D Meshes. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 8609–8619.

SERMANET, P., EIGEN, D., ZHANG, X., MATHIEU, M., FERGUS, R., AND LECUN, Y. 2014. Overfeat: Integrated recognition, localization and detection using convolutional networks. *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*.

SHEPHERD, J.F., DEWEY, M.W., WOODBURY, A.C., BENZLEY, S.E., STATEN, M.L., AND OWEN, S.J. 2010. Adaptive mesh coarsening for quadrilateral and hexahedral meshes. *Finite Elements in Analysis and Design 46*, 1–2, 17–32.

SIMONYAN, K. AND ZISSERMAN, A. 2015. Very deep convolutional networks for large-scale image recognition. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 1–14.

SU, H., MAJI, S., KALOGERAKIS, E., AND LEARNED-MILLER, E. 2015. Multi-view convolutional neural networks for 3D shape recognition. *Proceedings of the IEEE International Conference on Computer Vision 2015 Inter*, 945–953.

TARINI, M., PIETRONI, N., CIGNONI, P., PANOZZO, D., AND PUPPO, E. 2010. Practical quad mesh simplification. *Computer Graphics Forum 29*, 2, 407–418.

THIBEAULT, R., HONG, L., HOLLENBERG, C., AND CHRISTIAN, J. 2019. Triangles Vs . Quadrilaterals : Selecting the Right 3D Model Format for Space Science and Exploration. *2nd RPI Space Imaging Worop*, 1–11.

VERMA, N., BOYER, E., AND VERBEEK, J. 2018. FeaStNet: Feature-Steered Graph Convolutions for 3D Shape Analysis. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2598–2606.

WU, Z., SONG, S., KHOSLA, A., ET AL. 2015. 3D ShapeNets: A deep representation for volumetric shapes. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition 07-12-June*, 1912–1920.

ZHOU, Y., WU, C., LI, Z., ET AL. 2020. Fully convolutional mesh autoencoder using efficient spatially varying kernels. *Advances in Neural Information Processing Systems 2020-Decem*, NeurIPS.

Supplementary material

- Our code is published on github: *https://github.com/Omri-L/QuadMeshCNN*
- Our QuadZoo5 dataset can be downloaded from:

  https://drive.google.com/file/d/1A1h_3teFS51H7hTw-Omphe8IQfDHEZth/view?usp=sharing