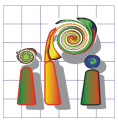
Master Thesis
Transformer-Based Classification of Non-Manifold Textured 3D Meshes

Mohammadreza Heidarianabaei

First Examiner: Prof. Dr. Franz Rottensteiner

Second Examiner: Dr. Max Mehltretter



Institute of Photogrammetry and Geoinformation

November 2024

Abstract

3D mesh segmentation is essential in diverse domains, including autonomous systems, medical imaging, and cultural heritage preservation, where the goal is to accurately classify individual components of a mesh. Traditional deep learning methods struggle with unstructured 3D data, particularly when dealing with non-manifold structures. Many existing techniques operate under the assumption of manifold mesh configurations, which restricts their effectiveness in real-world applications characterized by complex geometries.

In response to this challenge, we introduce NoMeFormer, a transformer-based framework designed to process any type of mesh without imposing structural constraints. This flexibility makes it particularly well-suited for non-manifold mesh segmentation. Our approach represents each mesh face as a token, allowing the model to utilize the order-invariant nature of transformers and learn meaningful representations without being limited by manifold prerequisites.

A distinctive feature of NoMeFormer is the incorporation of Local-Global (L-G) transformer blocks, which effectively address the quadratic complexity typically associated with transformer architectures. The model begins by aggregating features within spatial clusters of mesh faces, which are formed using k-means clustering. This is followed by a phase where long-range dependencies between faces are captured through global attention mechanisms. Such an architecture empowers the model to harness both low-frequency and high-frequency contextual information, enhancing its overall performance.

Our extensive experiments and ablation studies yield compelling results, NoMeFormer based on geometrical features achieves a mean F1 score of 58.9% on the Hessigheim 3D benchmark dataset. This demonstrates the framework's capacity to surpass the limitations imposed by manifold-based methodologies. As a result, NoMeFormer presents a robust solution for semantic segmentation and classification tasks involving non-manifold 3D meshes, showcasing its potential for broader applications in various fields requiring advanced mesh analysis.

Contents

List of Figures

List of Tables

List of Acronyms

DL Deep Learning

ANN Artificial Neural Network

NN Neural Network

MLP multi layer perceptrons

3D Three-Dimensional

ReLU Rectified Linear Unit

CNNs Convolutional Neural Networks

RNN Recurrent Neural Network

MHSA Multi-Head Self-Attention

NLP Natural Language Processing

GNNs Graph Neural Networks

ViT Vision Transformer

RF Random Forest

SOTA State of the Art

OA Overall Accuracy

mF1 Mean F1-Score

SGD Stochastic Gradient Descent

Chapter 1

Introduction

1.1 Introduction

Semantic segmentation of 3D meshes has become an essential task across various fields, driven by the growing dependence on 3D data for advanced modeling, simulation, and analytical purposes. Compared to 2D images and other 3D representations, 3D mesh segmentation offers distinct advantages, including geometric connectivity, clear surface information, computational efficiency, and in case of texture mesh detailed texture representation. In architecture, mesh segmentation facilitates detailed analysis of structural components, significantly benefiting building information modeling (BIM) and retrofitting processes (Gimenez et al., 2015). For autonomous vehicles, it enhances environmental perception, crucial for navigation and obstacle avoidance (Herb et al., 2021; Fawole and Rawat, 2024; El-Hakim et al., 2003). In the medical field, segmentation is vital for distinguishing anatomical structures in diagnostic imaging and surgical planning (Batchelor et al., 2012). Furthermore, In cultural heritage, semantic segmentation is instrumental in damage detection and preservation. By accurately identifying and classifying different regions of an artifact or historical structure (Agosto and Bornaz, 2017) (Balletti and Ballarin, 2019), while the gaming and entertainment industries utilize it to optimize rendering and animation pipelines. Overall, semantic segmentation is indispensable for improving workflow efficiency and accuracy across 3D-centric applications.

Deep learning has established itself as a dominant paradigm across various domains, excelling in tasks such as segmentation, classification, and generative modeling (LeCun et al., 2015; Ronneberger et al., 2015). Its superior performance has positioned it as the leading framework for these applications.

While the majority of deep learning frameworks are tailored for 1D or 2D inputs, extending these models to effectively learn feature representations from

3D data has become an active area of research within both computer vision and computer graphics (Ioannidou et al., 2017; Ahmed et al., 2018; Mildenhall et al., 2021). Recent efforts have aimed at adapting deep learning frameworks for 3D applications. However, applying traditional deep learning methods to 3D data is not straightforward due to the varying representations of 3D structures including volumetric grids, point clouds, and meshes.

Structured representations, such as volumetric and RGB-D data, offer relatively straightforward model design pathways due to their inherent similarity to images, allowing for direct adaptation of images architectures. However, these formats present significant limitations, particularly in memory consumption and computational complexity. The reliance on fixed grid resolutions constrains their ability to represent fine geometric details, making them suboptimal for surface-focused applications where precision is critical. Moreover, volumetric approaches tend to oversample regions of low interest while undersampling high-detail areas, leading to inefficient resource use and loss of critical surface information. As a result, their applicability diminishes in scenarios where capturing intricate surface geometry or fine-grained structural details is essential.

On the other hand, point clouds and meshes provide more efficient storage and offer higher fidelity in surface detail representation. Point clouds consist of discrete data points distributed in 3D space, capturing the geometric essence of an object's surface. Meshes take this further by connecting these points with edges and faces, forming a continuous surface that enhances detail representation, making them preferable for 3D tasks focused on precision and efficiency.

Despite the relative ease of generating point cloud data, the inherent lack of connectivity introduces ambiguity in representing complex structures. Consequently, meshes are often preferred as they offer smoother, more detailed visualizations of objects, thereby facilitating a more accurate interpretation of shapes and features. Beyond providing connectivity information, meshes also enable the representation of surface textures through a process known as texture mapping. In this process, textures are wrapped around the 3D mesh, imparting visual details that the mesh alone cannot convey—such as color variations, surface intricacies (e.g., wood grain or skin), and even simulated depth (e.g., bumps or grooves). Thus, meshes serve as more informative representation of 3D objects compared to other forms of 3D data representation. This enhanced representation is particularly beneficial for tasks such as classification, where a comprehensive understanding of real-world structures is critical for higher-level applications.

Although meshes as a representation for 3D data offer enhanced clarity regarding surface details and the geometry of three-dimensional objects, they pose significant challenges in designing neural networks adept at processing such complex data structures. To facilitate the integration of unstructured meshes into

neural network architectures and to enhance adaptability, manifold meshes are often utilized as input data. A manifold mesh is characterized by the property that all faces sharing a vertex form either a disc or a half-disc (Edelsbrunner and Harer, 2010). In simpler terms, each edge in a manifold mesh is shared by precisely two faces, and each vertex is linked to a singular connected component of faces. However, in many real-world applications, 3D meshes frequently do not conform to these manifold criteria, presenting additional hurdles for effective representation and processing.

In recent years, considerable research has been conducted in the field of 3D computer vision (Hanocka et al., 2019; Feng et al., 2019; Milano et al., 2020; Liang et al., 2022); however, only a limited number of studies have successfully tackled the intricacies of processing 3D non-manifold textured meshes for tasks such as semantic segmentation, classification, and generative modeling. To address this pressing issue, the primary objective of this thesis is to develop a robust transformer based deep learning framework that can directly operate on non-manifold textured meshes , leveraging the flexibility of transformer. This innovative framework aims to serve as a foundational model for various applications, including semantic segmentation, classification, self-supervised learning, and generative modeling, thereby paving the way for advancements in handling complex 3D data in practical scenarios.

Transformers initially emerged as a groundbreaking approach for NLP tasks (Vaswani, 2017), introducing a novel paradigm that diverges from traditional models such as RNNs and CNNs. Unlike these earlier architectures, Transformers are fundamentally rooted in the concept of attention, which allows the network to assign varying weights to different inputs, irrespective of their order or topological relationships. This property, known as order invariance, makes Transformers a compelling choice for processing unstructured data.

Following the demonstrated superiority of the Transformer architecture in NLP, substantial efforts have been devoted to adapting this framework for a wider array of data types and tasks. This trend has increasingly extended into the realm of 3D mesh representation learning (Liang et al., 2022). However, despite the emergence of Transformer-based methods for 3D meshes, these approaches have not fully capitalized on the order-invariant advantages of Transformers when handling non-manifold meshes. like traditional neural networks, which necessitate structured manifold meshes for representation learning, current Transformer-based methods often still rely on manifold meshes. Furthermore, they frequently overlook the incorporation of texture information, which is vital for achieving a comprehensive understanding of 3D data.

To address the challenges and limitations of existing methods, we propose the Non-Manifold Mesh Transformer. In our framework, each face of the mesh is

treated as an independent token, allowing for flexible feature aggregation without being constrained by the topological relationships between neighboring faces. This representation enables the model to capture both local and global patterns in the data, which is crucial for processing non-manifold meshes.

However, applying full attention across all faces in a mesh directly, as typical transformers do, leads to quadratic complexity in terms of computational cost, making such an approach infeasible for large-scale 3D data. To overcome this limitation while effectively learning both low- and high-frequency patterns within the data, inspired by (Chu et al., 2021) we introduce a two-step process, termed Local-Global (L-G) Transformer Blocks. This design reduces the computational burden by first focusing on local feature extraction within clusters of faces, followed by a global attention mechanism that captures broader context across the entire mesh. Through this hierarchical attention mechanism, our method preserves the rich details inherent in non-manifold meshes while maintaining computational efficiency.

In the first step, termed the Local Transformer Block, features are aggregated within patches using a self-attention mechanism applied to faces within each patch. These patches are generated by clustering adjacent faces in the 3D mesh through k-means clustering, ensuring the preservation of the local geometric structure. By focusing on localized regions of the mesh, this approach enables the model to effectively capture fine-grained variations and detailed local patterns—essential for addressing intricate geometric and textural differences in the data. Moreover, the Local Block summarizes the features within each patch into a single learnable token, referred to as the cluster token, which encapsulates the local representation for subsequent stages of the model.

In the second step, known as the Global Transformer Block, the cluster tokens generated by the local blocks are processed through an attention mechanism that allows each token to attend to the aggregated representations of other patches. This step enables the model to capture global contextual information across the entire mesh, facilitating the integration of long-range dependencies between spatially distant regions. The combined local and global attention mechanisms provide the model with a comprehensive understanding of both fine details and broader structural patterns.

Ultimately, the model performs semantic segmentation by predicting a label for each face of the mesh. The network is optimized through backpropagation by minimizing a suitable loss function, ensuring that it learns to associate each face with its corresponding semantic class. This two-step process efficiently balances local detail extraction with global context integration, making it highly effective for tasks requiring precise semantic understanding of non-manifold meshes.

In summary, our contributions can be outlined as follows:

- We propose a framework utilizing a transformer network capable of processing arbitrary 3D meshes without the necessity of imposing constraints such as manifold requirements. This can serve as a backbone for various tasks, such as segmentation and classification.
- To address the quadratic complexity of transformers, we present a novel approach that first clusters faces based on their spatial proximity to patches and then learns feature aggregation within local patches. Subsequently, a global block is implemented to capture global context and facilitate long-range interactions.
- We design a distinct branch to provide per-face feature representations to the model. This new branch integrates both geometrical and textural information, addressing the dimensional disparity between the two. By ensuring equal representation of both information types, this branch enhances the model's overall feature representation capability.
- We train the model under various configurations and conduct a comprehensive ablation study to evaluate the effectiveness of the proposed model components and input features.

Chapter 2

Literature Review

2.1 Related Work

In this section, we first present a brief review of deep learning-based 3D data processing techniques that utilize representations other than meshes, such as point clouds and voxels. Afterwards, we provide a detailed survey of research focused on deep learning-based 3D mesh analysis and processing, with particular emphasis on semantic segmentation.

2.1.1 Deep Learning on 3D data

Many works have addressed the design and development of deep learning frameworks tailored for 3D data processing. These approaches can be broadly categorized based on the type of 3D data representation used, such as voxel grids, point clouds, meshes, and textured meshes.

Early approaches to processing 3D data involved representing it as a voxel grid. This representation, analogous to 2D image structures, enabled the straightforward application of CNNs for 3D data analysis (Maturana and Scherer, 2015; Wu et al., 2015). Voxel-based approaches, however, are hindered by high memory consumption and computational inefficiency, especially when dealing with high-resolution data.While some efforts, such as OctNet (Riegler et al., 2017), have been made to address these challenges by hierarchically partitioning 3D space into octrees—where each octree splits the space based on data density and concentrates computations on relevant regions—more recent work has shifted toward processing point clouds. which offers a more memory-efficient and flexible representation that captures the raw geometry of 3D objects (Qi et al., 2017a,b; Qian et al., 2022; Hu et al., 2020). With the demonstrated success of transformers across various domains and their inherent order-invariant property, which obviates the need to

define the order of point cloud data, this has led researchers to explore their application in point cloud processing (Guo et al., 2021; Yu et al., 2022). However, the aforementioned methods are not capable of processing 3D textured meshes. In contrast, textured meshes combine geometric and appearance data, capturing both the object's shape and detailed surface information, such as color and texture. This richer representation significantly enhances feature learning for tasks like damage detection, surface condition analysis, and object recognition in complex environments. Furthermore, the vertex connectivity in textured meshes provides a coherent structure for feature aggregation, enabling more accurate spatial relationship modeling. This facilitates superior context-aware analysis, reducing ambiguities in learning geometric and visual features, and outperforming point clouds or voxel grids in tasks that demand high-fidelity surface information.

2.1.2 3D meshes

Due to the irregular structure of mesh data, many traditional deep learning models, originally designed for regular grid-based inputs, cannot be directly applied. This limitation has driven significant interest in adapting these models, as well as advancing geometric deep learning to address such data modalities (Bronstein et al., 2021). Researchers in this field focus on developing novel methods tailored to handle non-Euclidean structures, such as meshes. A primary objective of this section is to explore mesh processing techniques and representation learning for tasks like classification and semantic segmentation, which have been tackled using diverse methodologies and perspectives.

Whereas Laupheimer (2022) proposes to transfer the problem to point cloud or image classification by specific transfer functions, significant efforts have been made to extend concepts of CNNs and pooling layers to mesh processing. The pioneering work (Masci et al., 2015) first introduced a notion of convolution for non-Euclidean domains. The authors extended traditional CNNs to curved surfaces, represented as Riemannian manifolds, by employing geodesic polar coordinates for local patches instead of the standard grid structure used in Euclidean space. This approach involves defining convolution operations based on geodesic distances, thereby respecting the intrinsic geometry of the surface. MeshNet (Feng et al., 2019) was proposed as a deep learning network designed specifically to operate on 3D mesh faces. It incorporates two key descriptors: a spatial descriptor, which captures positional information via the center of gravity (COG) of each face, and a structural descriptor, which extracts geometric features. The structural descriptor relies on (1) face-rotate convolution to encode internal face geometry and (2) face-kernel correlation to capture relationships between neighboring faces. MeshNet also enhances spatial feature aggregation through mesh convolution layers that expand the receptive field by leveraging neighboring face indices.

Hu et al. (2022) continue the effort to adapt CNNs for mesh processing, with a particular focus on pooling layers to expand the receptive field. Their approach utilizes subdivision surfaces, specifically Loop subdivision, to construct a fine-to-coarse hierarchy, analogous to pooling operations in CNNs.

MeshCNN (Hanocka et al., 2019) presents a different paradigm in mesh-based CNNs by focusing on edges as the primary entities for classification. In this approach, convolution operations are defined as learnable parameters are applied to the four edges incident to a given edge. To expand the receptive field MeshCNN employs an edge collapse pooling mechanism, where the network learns which edges to collapse, thereby dynamically adjusting the topology to improve the receptive field and hierarchical feature learning.

PD-Mesh (Milano et al., 2020) considers the mesh as a graph structure and extends pointwise convolution to mesh processing. The authors construct two mesh graphs: one where nodes represent faces and another where nodes represent edges. Features from adjacent nodes in both graphs are aggregated using (GAT). To mimic pooling operations, they apply a mesh simplification technique. MeshWalker (Lahav and Tal, 2020) defines a random walk on the vertices of the mesh and leverages the sequential nature of this process by using recurrent neural networks (RNNs) to learn mesh representations.

Previous methods impose constraints on input meshes, such as requiring them to be manifold. which restricts the network's ability to process arbitrary 3D meshes, particularly non-manifold geometries. Few works address the challenging constraints imposed by previous methods that assume a manifold mesh. Laplacian2Mesh (Dong et al., 2023) attempts to alleviate this by transforming the mesh into the spectral domain using the Laplacian matrix of the mesh, representing the mesh with the k eigenvectors of this matrix. The network is then trained in this spectral domain, followed by a transformation back to the spatial domain. Although this method can handle non-manifold meshes, it is primarily designed for vertex segmentation rather than face segmentation. Additionally, the transformation to the spectral domain imposes limitations on data representation. DiffusionNet (Sharp et al., 2022) is designed around three key components: Pointwise Perceptrons, Learned Diffusion, and Spatial Gradient Features. The core concept involves optimizing the steps in the diffusion equation to effectively learn the diffusion process. Additionally, a Multi-Layer Perceptron (MLP) is employed to process the raw features of each vertex alongside spatial gradient features, enhancing directional filtering. While this network can process various types of meshes, feature aggregation is limited to continuous and local fields of view due to its diffusion equation-based approach.

Inspired by the transformative impact of scaling Transformer models in NLP, other disciplines have also begun shifting towards Transformer architectures, aim-

ing to replace traditional CNNs for enhanced performance across various tasks. This shift has spurred the development of masked autoencoder networks (He et al., 2022) for self-supervised training of mesh data (Liang et al., 2022). These networks can be effectively utilized for downstream tasks, such as semantic segmentation. To address the quadratic complexity associated with attention mechanisms, this work leverages patches of data similar to those used in ViT (Dosovitskiy et al., 2020). However, Due to the irregular structure of meshes, traditional patching methods based on pixel grids cannot be directly applied. To address this, the authors leverage the MAPS algorithm (Lee et al., 1998), which aims to merge mesh faces to create a coarser representation. This approach treats the merged faces as single patches, concatenating their features and using patch embeddings to create tokens. However, this mesh simplification method introduces constraints on the input mesh and faces challenges when processing non-manifold meshes.

To our knowledge, no prior work has effectively adapted deep neural networks for processing non-manifold meshes. While considerable research has been conducted on the semantic segmentation and classification of 3D meshes with promising ac- curacy, nearly all methods impose manifoldness as a constraint on input meshes. Methods that do bypass this requirement often suffer from limitations, including spectral domain information loss, constrained local feature aggregation, or discontinuities in feature representation. The most closely related work that integrates transformers is (Liang et al., 2022), yet it, too, necessitates manifoldness for generating patches. To address these limitations, we leverage the unique property of transformers that permits order-invariant processing, eliminating the need for structural constraints on mesh input. Further, to make the trans- former applicable for large meshes, we draw inspiration from the two-step processing strategy used in (Chu et al., 2021) for efficient image processing with transformers.

Chapter 3

Background

In this section, we provide the necessary context and foundational knowledge required to understand the research presented in this thesis. This section is divided into two parts. The first part provides a review of the fundamentals of neural networks, followed by a detailed discussion on transformer networks. The second part focuses on 3D texture mesh representation and its components.

3.1 Artificial Neural Networks

The concept of ANNs is inspired by the structure of the human nervous system. The nervous system comprises neurons and synapses, which are interconnected to facilitate communication. Neurons transmit information through synapses, which act as junctions between them. The inception of ANNs can be traced back to Frank Rosenblatt's work on the Perceptron, an early neural network model designed for classifying data. The Perceptron operates by computing a weighted sum of inputs and applying a step activation function, making it suitable for linearly separable problems. During training, the model adjusts its weights to minimize errors.

A major limitation of Rosenblatt's Perceptron arises from its single-layer architecture. Initially, it was believed that this limitation would persist even with the addition of more layers, a viewpoint shared by some early pioneers in the field. However, this assumption was later proven incorrect, as demonstrated by the effectiveness of MLP.

In mathematical terms, ANNs can be conceptualized as functions that transform an input vector into an output vector. Specifically, this can be expressed as $y = f(x; \theta)$, where θ denotes the network's parameters. The network's objective is to learn the optimal values for θ to approximate the desired function as closely as possible. These functions encompass various components and methodologies which will be elaborated upon in the subsequent sections.

3.1.1 Structure of ANNs

The fundamental architecture of an ANN comprises three main types of layers:

- **Input Layer**: This is the initial layer of the network, responsible for receiving the raw data or input features. Each node in the input layer corresponds to a feature of the input data.
- **Hidden Layers**: Positioned between the input and output layers, these layers may consist of one or more stages. Each hidden layer comprises neurons that apply nonlinear transformations to the data. Neurons in these layers use activation functions to process the weighted sum of their inputs.
- **Output Layer**: The final layer of the network, which produces the model's output. The configuration of the output layer is determined by the specific task (e.g., classification, regression). For classification tasks, it commonly employs a softmax function to generate probabilities for each class.

3.1.2 Neurons and Activation Functions

- **Neurons**: Each neuron in an ANN is a computational unit that performs a weighted sum of its inputs, adds a bias term, and then applies an activation function to the result. Mathematically, the output of a neuron can be expressed as:

$$y = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{3.1}$$

  where w_i represents the weights, x_i are the input values, b is the bias, and ϕ is the activation function.

- **Activation Functions**: Activation functions introduce non-linearity into the network, allowing it to learn complex patterns. Common activation functions include:
  - **Sigmoid**: $\phi(x) = \frac{1}{1+e^{-x}}$ — Maps the output to a range between 0 and 1.
  - **ReLU**: $\phi(x) = \max(0, x)$ — Outputs the input directly if it is positive; otherwise, it outputs zero.
  - **Tanh**: $\phi(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ — Maps the output to a range between -1 and 1.

3.1.3 Learning Process

In any ANN, a set of learnable parameters is at its core. The training process refers to the procedure of adjusting these parameters to find their optimal values for a given task. In the context of supervised learning, "optimal" specifically refers to minimizing the discrepancy between the network's predictions and the corresponding labels for the input data. This is typically achieved through the following steps:

- **Forward Propagation**: During forward propagation, the input data $\mathbf{x} \in \mathbb{R}^n$ is passed through each layer of the network to generate an output. For a given layer l, the output $\mathbf{z}^{(l)}$ is computed as:

  $$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$$

  where $\mathbf{W}^{(l)}$ is the weight matrix, $\mathbf{a}^{(l-1)}$ is the activation from the previous layer, and $\mathbf{b}^{(l)}$ is the bias vector. The weighted sum $\mathbf{z}^{(l)}$ is then passed through an activation function σ, yielding the output of the layer:

  $$\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)})$$

  This process is repeated layer by layer until the final output $\hat{\mathbf{y}}$ is produced at the output layer.

- **Loss Function**: In machine learning, the loss function, denoted as $L(y, \hat{y})$, quantifies the discrepancy between the predicted output $\hat{y}$ and the true target values y. The loss function used by a machine learning algorithm often decomposes as a sum over training examples The specific formulation of L is contingent upon the problem's characteristics, the structure of the dataset, and the machine learning model employed. Consequently, the design and selection of appropriate loss functions L play a critical role in the optimization process and remain a pivotal focus of research in machine learning.

  In classification and semantic segmentation tasks, cross-entropy loss is typically adopted as the objective function due to its capability to quantify the divergence between the predicted probability distribution and the ground-truth distribution. Cross-entropy effectively penalizes erroneous predictions based on their confidence levels, thereby aligning the model's predictions with the correct class probabilities. By minimizing cross-entropy, the model is encouraged to maximize the log-likelihood of the true class labels, yielding sharper probability outputs. This formulation is particularly advantageous in segmentation tasks, where per-pixel classification accuracy is paramount.

Furthermore, cross-entropy loss is differentiable, facilitating efficient optimization via gradient descent methods. Its robustness and interpretability make it a preferred choice across a broad range of segmentation architectures. Mathematically, for a given set of predictions $\hat{y}_i$ and true labels y_i, the cross-entropy loss function L is defined as:

$$L = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

-

where N is the number of samples, y_i is the true label, and $\hat{y}_i$ is the predicted probability for the i-th sample. Cross-entropy is effective in multi-class classification problems, where it penalizes the incorrect predictions by taking into account the log probability of the correct class.

For semantic segmentation of mesh, where face-wise classification is involved, cross-entropy is applied across all faces in the mesh Given a face f, the face-wise cross-entropy loss can be formulated as:

$$L_{\text{face}} = -\sum_{c=1}^{C} y_f^c \log(\hat{y}_f^c)$$

where C is the total number of classes, y_p^c is the ground truth label for class c at face f, and $\hat{y}_f^c$ is the predicted probability for class c at face f. The overall loss for the entire image is the sum of the face-wise losses.

$$L = -\sum_{i=1}^{N} \sum_{c=1}^{C} y_i^c \log(\hat{y}_i^c)$$

- **Backpropagation**: Back-propagation is often misunderstood as the entire learning algorithm for neural networks, but it specifically refers to the method of computing gradients for the parameters. It efficiently calculates the partial derivatives of the loss function with respect to each parameter by applying the chain rule in reverse order from the output to the input.

  Consider a multi-layer neural network with L layers. Let $\mathbf{a}^{(l)}$ denote the activations at layer l, and $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ represent the weights and biases for layer l. As the forward pass computes $\mathbf{z}^{(l)}$ and $\mathbf{a}^{(l)}$. In the back-propagation step, we compute the gradients with respect to the parameters by propagating the error backwards. Starting with the output layer, the error term $\delta^{(L)}$ for the final layer L is:

$$\delta^{(L)} = \frac{\partial J}{\partial \mathbf{a}^{(L)}} \odot \sigma'(\mathbf{z}^{(L)})$$

where J is the cost function, $\odot$ denotes element-wise multiplication, and σ' is the derivative of the activation function.

For earlier layers $l = L-1, L-2, \ldots, 1$, the error term is propagated as:

$$\delta^{(l)} = (\mathbf{W}^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(\mathbf{z}^{(l)})$$

The gradients of the cost function with respect to the weights and biases for each layer are then given by:

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^T$$

$$\frac{\partial J}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}$$

Back-propagation efficiently computes these gradients for each layer, allowing the learning algorithm to update the parameters during training.

- **optimization**: Deep learning algorithms typically rely on some form of optimization. In essence, optimization involves adjusting the variable x to either minimize or maximize a specific function $f(x)$. Most often, these optimization tasks are framed as minimizing $f(x)$, since minimization is a more common and convenient formulation. When the objective is to maximize $f(x)$, this can still be approached using minimization techniques by instead minimizing $-f(x)$, effectively transforming the maximization problem into a minimization one. The process of optimization is crucial because it governs the learning dynamics in neural networks, influencing how models converge to optimal solutions during training.

  Optimization algorithms that rely on gradients are referred to as first-order optimization algorithms. In contrast, algorithms that also utilize the Hessian matrix, such as Newton's method, are classified as second-order optimization algorithms . Among these, gradient descent is the dominant algorithm for deep learning models, as it is a first-order iterative algorithm.

  We consider all the learnable parameters of the model as a vector $\mathbf{W}$. The idea behind gradient descent is to treat the loss function as a surface governed by the model. By iteratively taking small steps in the negative direction of

the gradient, we can converge to the minimum of the surface, which corresponds to the minimum of the loss function. Mathematically, this can be expressed as:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \nabla L(\mathbf{W}_t)$$

where $\mathbf{W}_t$ represents the parameters at iteration t, η is the learning rate, and $\nabla L(\mathbf{W}_t)$ is the gradient of the loss function with respect to the parameters at iteration t.

In this formula, the gradient reveals the direction, while the learning rate sets the step size in that direction. In the SGD learning algorithm, it is necessary to specify a value for the learning rate parameter η. If η is very small, the learning process will proceed slowly. Conversely, if η is set too high, it can lead to instability in the training process. In practice, the best results are obtained by using a larger value for η at the start of training and then reducing the learning rate over time Bishop and Bishop (2023).

In gradient descent, we compute the gradient of the loss function with respect to the entire dataset. However, for large datasets, this process can be computationally expensive. Stochastic gradient descent mitigates this issue by approximating the gradient using only a small batch or a single training example at each iteration. The update rule for SGD can be written as:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \nabla L(\mathbf{W}_t; \mathbf{x}^{(i)})$$

Here, $\mathbf{x}^{(i)}$ represents a single training example (or a mini-batch) from the dataset. This also introduces some noise into the updates, allowing for more frequent updates but potentially more erratic convergence behavior compared to full gradient descent.

3.1.4 Multi-Layer Perceptron (MLP)

The MLP represents one of the foundational architectures in artificial neural networks. Comprising an input layer, one or more hidden layers, and an output layer as shown in figure 3.1, the MLP employs a fully connected topology, where each neuron in a given layer is connected to every neuron in the subsequent layer. This architecture enables the MLP to learn complex, non-linear representations by iteratively adjusting the weights of these connections through backpropagation.

Each neuron within the network applies a weighted sum of its inputs, followed by a non-linear activation function, such as the rectified linear unit (ReLU) or

sigmoid function. The use of these activation functions imparts the MLP with its non-linear capabilities, essential for capturing complex relationships in the data.

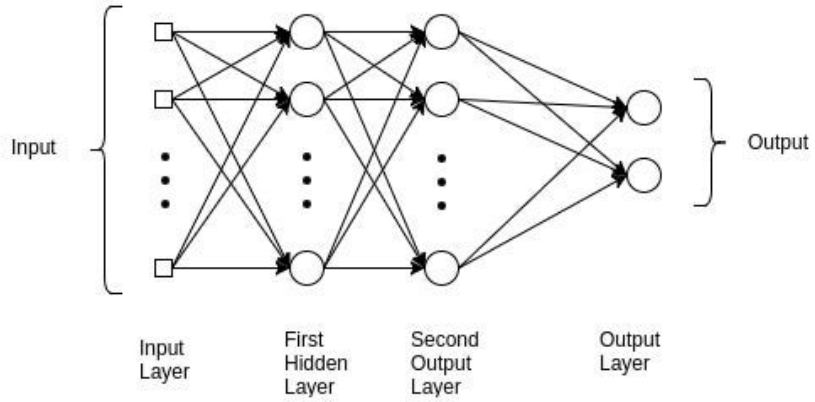


Figure 3.1: A MLP architecture showcasing multiple interconnected layers of neurons, where each layer transforms the input data through weighted connections and activation functions. Image source: (Vidhya, 2020)

Mathematically, the operation of an MLP can be formalized as follows. Given an input vector $\mathbf{x} \in \mathbb{R}^n$, the output of a hidden layer is computed as:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{3.2}$$

where $\mathbf{W} \in \mathbb{R}^{m \times n}$ represents the weight matrix, $\mathbf{b} \in \mathbb{R}^m$ denotes the bias term, and $\sigma(\cdot)$ is the non-linear activation function. The final output is derived by propagating the hidden layer activations through subsequent layers in a similar fashion. The adjustment of weights and biases is achieved via gradient descent methods, with the gradients computed using backpropagation.

MLPs are often used in conjunction with other architectures, such as convolutional neural networks (CNNs) or transformers.

3.1.5 Attention mechanism

Attention has been a pivotal development in deep learning. Since its introduction, the effectiveness of attention mechanisms has led to substantial progress across various domains in artificial intelligence, with many advancements building upon this module. The origin of attention traces back to recurrent neural networks (RNNs) and their difficulty in retaining information over long sequences, particularly in encoder-decoder architectures for machine translation. In these models,

the encoder would only pass the final hidden state to the decoder, which resulted in the loss of important information as sequence length increased.

To address this issue, the concept of attention mechanisms was first introduced by Bahdanau et al. Bahdanau (2014) and later Luong (2015). This attention framework allowed the decoder to focus on different parts of the input sequence, rather than relying solely on the final hidden state. Initially, these attention models were built on top of the RNN hidden states and were not independent frameworks. It wasn't until the introduction of self-attention, specifically scaled dot-product attention, in the Transformer model Vaswani (2017) that attention became a standalone mechanism, revolutionizing deep learning architectures.

Self-attention is a mechanism designed to enable models to focus on different parts of an input sequence during the processes of encoding or generating outputs. Unlike conventional conventional representation learning methods such as MLPs, CNNs. In traditional methods, learnable parameters directly function as weights for feature aggregation. In contrast, self-attention dynamically computes these weights based on the pairwise similarities between input elements, which, from this point onward, we refer to as tokens. The input tokens are projected into a different space through learnable parameters, and their pairwise similarities are computed using dot products. These similarities then determine the attention weights for aggregating features.

Input Representation

Given an input sequence $X = [x_1, x_2, \ldots, x_n]$, where each $x_i \in \mathbb{R}^d$ is a d-dimensional representation of the i-the element in the sequence, self-attention aims to produce an output sequence of the same length, where each output o_i is computed as a weighted sum of the inputs.

Key, Query, and Value Vectors

Each input element x_i is projected into three different vectors: a *query* vector q_i, a *key* vector k_i, and a *value* vector v_i. These projections are linear transformations of the input:

$$q_i = W_q x_i, \quad k_i = W_k x_i, \quad v_i = W_v x_i$$

where $W_q, W_k, W_v \in \mathbb{R}^{d_k \times d}$ are learned weight matrices that map the input dimension d to the attention dimension d_k. These vectors are collectively referred to as the query matrix Q, key matrix K, and value matrix V, where:

$$Q = [q_1, q_2, \ldots, q_n]^\top, \quad K = [k_1, k_2, \ldots, k_n]^\top, \quad V = [v_1, v_2, \ldots, v_n]^\top$$

Scaled Dot-Product Attention

The next step is to compute the attention weights that indicate how much focus should be placed on other elements in the sequence when computing the representation of each element. For each query q_i, the attention weights are computed by taking the dot product with each key k_j in the sequence, followed by a scaling factor and a softmax operation:

$$\text{Attention}(q_i, K) = \text{softmax}\left(\frac{q_i K^\top}{\sqrt{d_k}}\right)$$

Here, $q_i K^\top$ represents the dot product between the query vector q_i and each key vector k_j. The result is scaled by $\sqrt{d_k}$ to prevent the dot products from becoming too large, which could make the softmax function produce extremely small gradients during training.

The softmax function normalizes the attention scores so that they sum to 1:

$$\alpha_{ij} = \frac{\exp\left(\frac{q_i k_j^\top}{\sqrt{d_k}}\right)}{\sum_{j=1}^{n} \exp\left(\frac{q_i k_j^\top}{\sqrt{d_k}}\right)}$$

Here, α_{ij} represents the attention weight that element i places on element j. This weighting is applied to the value vectors.

Output Calculation

Once the attention weights are obtained, the output for each element is computed as the weighted sum of the value vectors v_j:

$$o_i = \sum_{j=1}^{n} \alpha_{ij} v_j$$

Thus, the output sequence $O = [o_1, o_2, \ldots, o_n]^\top$ is computed as:

$$O = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

Cross-Attention Mechanism

The cross-attention mechanism operates similarly to self-attention, but instead of deriving the queries, keys, and values from the same source, the queries come from one set of inputs (such as a later stage in a model), while the keys and values are derived from a different set of inputs (such as an earlier stage or another model component).

Mathematically, cross-attention is computed as:

$$\text{CrossAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

In cross-attention:

- $Q \in \mathbb{R}^{m \times d_k}$ is the query matrix derived from one set of inputs (where m is the length of the target sequence).
- $K \in \mathbb{R}^{n \times d_k}$ and $V \in \mathbb{R}^{n \times d_v}$ are the key and value matrices, respectively, derived from the different set of inputs (where n is the length of the source sequence).
- d_k and d_v are the dimensions of the key and value representations, respectively, and the softmax function is applied to the scaled dot-product between queries and keys.

3.1.6 Transformer Networks

The Transformer network, first introduced in Vaswani (2017), is fundamentally built upon the attention mechanism. While attention is the core of the network, the term 'Transformer' does not solely imply attention. In fact, the model comprises several distinct components, which will be reviewed in this section.

A Transformer is composed of an encoder-decoder structure, where both the encoder and decoder consist of multiple identical layers. Each encoder layer has two sub-layers: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. The decoder is similar to the encoder but contains an additional third sub-layer, which performs multi-head attention over the encoder's output. Although the original transformer model consists of both an encoder and a decoder, later developments have led to the use of either the encoder or the decoder independently in various learning models. In the context of this thesis, the model is built primarily using the transformer encoder as shown in figure 3.2.

The input sequence to the transformer can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d_{\text{model}}}$, where n is the sequence length and d_{model} is the dimensionality of each token's representation. For example, in NLP, n would correspond to the number of words in a sentence and d_{model} is the dimensionality of word embeddings.

Input Embedding

Given a sequence of input tokens $[x_1, x_2, \ldots, x_n]$, where each x_i represents a discrete token (such as a word or symbol in the case of text), the first step is to map each

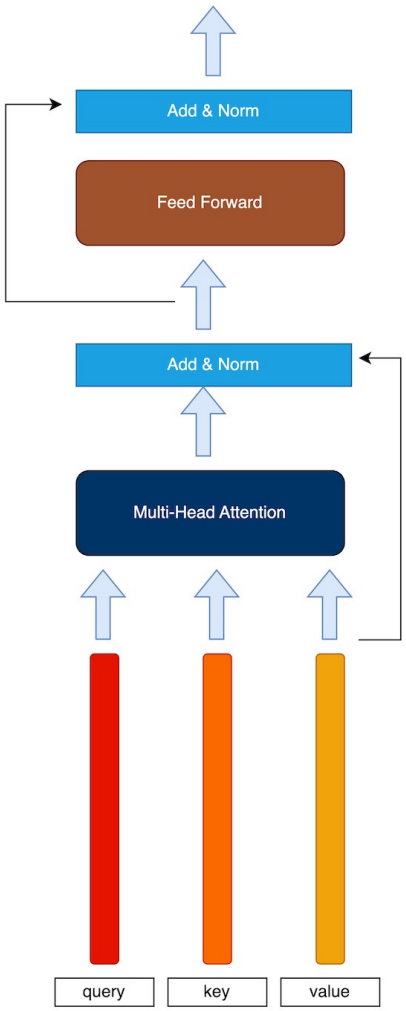


Figure 3.2: Transformer Encoder

token to a dense vector representation, often referred to as an *embedding*. Each token x_i is mapped to an embedding $e_i \in \mathbb{R}^d$, where d is the dimension of the embedding space.

$$e_i = \text{Embedding}(x_i)$$

Thus, the input sequence X becomes a sequence of embeddings $E = [e_1, e_2, \ldots, e_n]$, where $E \in \mathbb{R}^{n \times d}$.

Multi-Head Attention

To capture multiple types of relationships between tokens, the Transformer uses multi-head attention. Instead of applying self-attention once, it is applied h times in parallel, where each head learns different attention distributions. Each head has its own set of learned parameters for Q, K, and V, and the results are concatenated and linearly transformed.

The multi-head attention is given by:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O$$

$$\text{head}_i = \text{Attention}(Q\mathbf{W}_i^Q, K\mathbf{W}_i^K, V\mathbf{W}_i^V)$$

where $\mathbf{W}_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $\mathbf{W}_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ are learnable projection matrices.

By using multiple attention heads, the model can attend to different parts of the sequence and capture a richer set of relationships between tokens.

Positional Encoding

Since the Transformer does not inherently model the order of tokens, positional encodings are added to the input embeddings to provide information about the relative or absolute positions of tokens in the sequence. The positional encodings are computed using sinusoidal functions:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

where pos is the position of the token in the sequence, and i is the dimension of the positional encoding.

The positional encoding $\text{PE} \in \mathbb{R}^{n \times d_{\text{model}}}$ is added to the input embeddings to form the final input to the Transformer. It is important to note that, since the 3D position of each vertex is already part of the input features, our model does not include positional encoding.

Feed-Forward Network

Each layer of the Transformer includes a position-wise fully connected feed-forward network applied independently to each token. The feed-forward network consists of two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = \max(0, x\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ are learned weight matrices, and b_1 and b_2 are learned biases. Typically, d_{ff} is larger than d_{model}, such as $d_{\text{ff}} = 2048$ and $d_{\text{model}} = 512$.

Residual Connections and Layer Normalization

To facilitate training and improve gradient flow through the network, the Transformer uses residual connections around each sub-layer (self-attention and feed-forward network). A residual connection adds the input to the output of the sub-layer, followed by layer normalization:

$$\mathbf{z}_{\text{output}} = \text{LayerNorm}(\mathbf{z}_{\text{input}} + \text{sub-layer}(\mathbf{z}_{\text{input}}))$$

where LayerNorm is a layer normalization function that stabilizes the training process by normalizing the output across the feature dimensions.

Complexity Analysis

The time complexity of the attention mechanism is $\mathcal{O}(n^2 d_{\text{model}})$ due to the computation of attention scores between all pairs of tokens, making the vanilla Transformer inefficient for long sequences. Various efficient Transformer variants have been proposed to mitigate this issue by reducing the quadratic complexity.

This formal description of the Transformer network highlights its core components and mathematical underpinnings. The transformer architecture has paved the way for highly effective models in many domains, and its design continues to inspire new research into more efficient and scalable deep learning architectures.

3.2 3D Mesh Representation

A 3D mesh is a geometric data structure that represents a three-dimensional object using vertices, edges, and faces. Formally, a mesh M can be defined as a tuple:

$$M = (V, E, F)$$

where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is the set of **vertices** in $\mathbb{R}^3$, representing the spatial coordinates of the mesh points such that each $v_i = (x_i, y_i, z_i)$.
- $E = \{e_1, e_2, \ldots, e_m\}$ is the set of **edges**, which connect pairs of vertices. An edge e_j is defined as $e_j = (v_i, v_k)$, where $v_i, v_k \in V$.

- $F = \{f_1, f_2, \ldots, f_p\}$ is the set of **faces**, typically represented as triangles. Each face f_l is defined by an ordered triplet of vertices (v_i, v_j, v_k), which form a plane in 3D space.

There are two primary types of 3D meshes: triangular meshes and quadrilateral meshes. Triangular meshes are the most common due to their simplicity and computational efficiency.

3.2.1 Geometry of Vertices and Faces

Each vertex v_i is defined by its position in 3D space:

$$v_i = (x_i, y_i, z_i) \quad \text{where} \quad x_i, y_i, z_i \in \mathbb{R}$$

The surface normal $\mathbf{n}$ for a triangular face f composed of vertices (v_i, v_j, v_k) can be calculated using the cross product of the two edge vectors $\mathbf{v_{ij}}$ and $\mathbf{v_{ik}}$, where:

$$\mathbf{v_{ij}} = v_j - v_i, \quad \mathbf{v_{ik}} = v_k - v_i$$

The normal vector $\mathbf{n}$ is then:

$$\mathbf{n} = \frac{\mathbf{v_{ij}} \times \mathbf{v_{ik}}}{\|\mathbf{v_{ij}} \times \mathbf{v_{ik}}\|}$$

This normal vector $\mathbf{n}$ provides important information about the orientation of the face relative to the 3D space and is essential for surface rendering, lighting computations, and mesh transformations.

3.2.2 Connectivity and Topology

The **connectivity** of the mesh refers to how the vertices and edges are connected to form faces. The mesh's topology is typically described by the adjacency relations between vertices, edges, and faces. The Euler characteristic χ is a fundamental topological property, given by:

$$\chi = V - E + F$$

This invariant holds for most types of surfaces and can be used to verify the consistency of a 3D mesh. For example, a simple convex polyhedron will have $\chi = 2$.

3.2.3 Surface Area and Volume Calculation

The total surface area A of the mesh can be computed by summing the areas of individual triangular faces. The area A_f of a triangular face with vertices v_i, v_j, and v_k is given by:

$$A_f = \frac{1}{2}\|\mathbf{v_{ij}} \times \mathbf{v_{ik}}\|$$

Thus, the total surface area A of the mesh is:

$$A = \sum_{f \in F} A_f$$

For a closed 3D mesh, the volume V enclosed by the surface can be approximated using the divergence theorem or by summing the signed volumes of the tetrahedra formed between each triangular face and a common reference point v_0:

$$V = \frac{1}{6} \sum_{f \in F} (\mathbf{v_i} \times \mathbf{v_j}) \cdot \mathbf{v_k}$$

where $\mathbf{v_i}, \mathbf{v_j}, \mathbf{v_k}$ are the vertices of the face.

3.2.4 Non-Manifold Mesh

A non-manifold mesh is a geometric structure that does not follow the regular properties of a 2-manifold surface. In a 2-manifold mesh, each edge is shared by exactly two faces, and each vertex has a single neighborhood that is homeomorphic to a disk. In contrast, non-manifold geometries may exhibit features such as:

- **Non-manifold vertices:** A vertex is shared by multiple disconnected regions or surfaces, causing ambiguity in defining a single normal direction or local neighborhood.
- **Non-manifold edges:** An edge is shared by less or more than two faces, violating the 2-manifold assumption.
- **Mixed-dimensional elements:** Meshes that contain elements of varying dimensions, such as combining surfaces and line segments, are considered non-manifold.

In practice, non-manifold meshes arise in various scenarios, such as outdoor 3D data , complex CAD models, medical imaging data, or topologically intricate objects that cannot be represented with a simple manifold structure.

Formal Definition and Properties

Formally, a non-manifold mesh $M = (V, E, F)$ is characterized by the failure of certain conditions that are satisfied by 2-manifolds:

- Non-manifold vertex: A vertex $v \in V$ is non-manifold if it is shared by more than one set of connected neighborhoods, meaning that the local neighborhood of v is not homeomorphic to a 2D disk. For a vertex v_i, the set of incident edges E_i and faces F_i forms disjoint sets of connected components.

- Non-manifold edge: An edge $e_j = (v_i, v_k) \in E$ is non-manifold if it is shared by more or less than two faces. For example, if the edge e_j belongs to faces f_1, f_2, f_3, then:

  $$|\text{Incident faces on } e_j| > 2$$

  This condition causes issues in algorithms that rely on the assumption that each edge belongs to exactly two faces, such as certain mesh refinement or simplification algorithms.

Topology of Non-Manifold Meshes

The topology of a non-manifold mesh can no longer be described using the basic Euler characteristic formula $\chi = V - E + F$, since this formula assumes a 2-manifold structure. For non-manifold meshes, we can use a modified Euler characteristic that incorporates additional terms for non-manifold vertices, edges, and higher-order elements.

Let χ_{nm} be the modified Euler characteristic for a non-manifold mesh. This can be expressed as:

$$\chi_{nm} = \chi + N_{nmv} + N_{nme} + N_{nmf}$$

where:

- χ is the standard Euler characteristic for a 2-manifold mesh.
- N_{nmv} is the number of non-manifold vertices.
- N_{nme} is the number of non-manifold edges.
- N_{nmf} is the number of non-manifold faces, i.e., faces that cannot be defined by two-manifold geometry.

Thus, for a non-manifold mesh, the Euler characteristic is adjusted to account for topological irregularities.

Geometric Properties of Non-Manifold Meshes

Non-manifold meshes introduce challenges in calculating geometric properties such as normals, curvature, surface area, and volume:

- Normals: In non-manifold regions, particularly at non-manifold vertices and edges, there may be multiple or undefined surface normals. For example, if an edge is shared by three faces, each face may have a different normal, and it is unclear how to define the normal of the edge itself.

- Curvature: The curvature at non-manifold vertices and edges becomes complex and may be undefined. The Gaussian curvature K at a manifold vertex v_i is typically given by the angular deficit:

$$K(v_i) = 2\pi - \sum \theta_f$$

  where $\sum \theta_f$ is the sum of the angles at v_i for each incident face. However, in non-manifold vertices, the angular deficit becomes ill-defined, as the neighborhood is not homeomorphic to a disk.

- Surface Area and Volume: Surface area calculations on non-manifold meshes require special care. The total surface area A of the mesh can still be computed by summing the areas of individual faces, but care must be taken at non-manifold edges where multiple faces meet. Similarly, volume computations using the divergence theorem or tetrahedral decomposition may fail for non-manifold meshes due to disconnected regions or undefined face normals.

The area A_f of a triangular face $f = (v_i, v_j, v_k)$ in a non-manifold region remains:

$$A_f = \frac{1}{2}\|\mathbf{v_{ij}} \times \mathbf{v_{ik}}\|$$

However, due to the non-manifold topology, it is necessary to adapt algorithms for integration over surfaces to handle irregular connectivity at non-manifold edges and vertices.

Non-Manifold Mesh Processing

Handling non-manifold meshes requires specialized algorithms. Standard manifold mesh operations, such as simplification, subdivision, or smoothing, may fail or produce incorrect results on non-manifold geometry. Approaches to processing non-manifold meshes include:

- Re-meshing: Convert the non-manifold mesh to a manifold representation by duplicating vertices or edges to separate disconnected regions.
- Topology-aware algorithms: Algorithms that explicitly handle non-manifold topology by accounting for multiple incident faces on edges or disconnected vertex neighborhoods.

In conclusion, non-manifold meshes pose significant challenges due to their complex topology and geometry. Handling such structures requires careful consideration of connectivity and modified algorithms for geometric processing.

3.2.5 Textured Meshes

A textured mesh is a 3D mesh that incorporates surface detail by mapping a 2D texture onto its geometry. The 2D texture is typically an image, and each face or vertex of the 3D mesh is associated with texture coordinates that define how the texture is applied to the surface. Formally, a textured mesh T can be defined as:

$$T = (M, \mathbf{C}, \mathbf{T}, \mathbf{UV})$$

where:

- $M = (V, E, F)$ is the underlying 3D mesh, composed of vertices V, edges E, and faces F.
- $\mathbf{T} = \{t_1, t_2, \ldots, t_k\}$ is the set of textures, where each t_i is represented as a tensor of size $\mathbb{R}^{H \times W \times C}$:
$$t_i \in \mathbb{R}^{H \times W \times C}$$
where H is the height, W is the width, and C is the number of color channels (e.g., $C = 3$ for RGB or $C = 4$ for RGBA).
- $\mathbf{UV}$ is the set of texture coordinates defined as:
$$\mathbf{UV} = \{(u_j, v_j) \mid j = 1, 2, \ldots, m\}$$
where $(u_j, v_j) \in [0, 1] \times [0, 1]$ maps each vertex or face of the 3D mesh to corresponding points in the 2D texture space.

Texture Mapping and UV Coordinates

In a textured mesh, each vertex of the 3D mesh is assigned a corresponding point in the 2D texture space, referred to as texture coordinates or UV coordinates. This process, known as UV mapping, defines how the 2D texture is applied to the

3D surface by mapping the geometry of the mesh onto the texture image. Each vertex $v_i \in V$ is assigned a 2D coordinate (u_i, v_i), where $u_i, v_i \in [0, 1]$ represent normalized positions within the texture image.

For a triangular face $f = (v_i, v_j, v_k)$, the texture coordinates are represented as:

$$\mathbf{UV}_f = ((u_i, v_i), (u_j, v_j), (u_k, v_k))$$

These UV coordinates dictate how the texture wraps over the surface of the 3D object, and they are used to sample color information from the texture image during rendering. To apply the texture accurately, the coordinates for any point p on a triangular face f need to be calculated based on the texture coordinates of the face's vertices.

For any point p on the surface of the triangle, its texture coordinates (u_p, v_p) are found by interpolating between the texture coordinates of the triangle's vertices using barycentric coordinates. If the barycentric coordinates of p relative to the vertices v_i, v_j, v_k are $\lambda_1, \lambda_2, \lambda_3$, the texture coordinates for p can be computed as:

$$(u_p, v_p) = \lambda_1(u_i, v_i) + \lambda_2(u_j, v_j) + \lambda_3(u_k, v_k)$$

In practice, the color value at (u_p, v_p) is typically obtained from the texture image using bilinear interpolation. This technique computes the color at (u_p, v_p) by taking a weighted average of the four nearest pixels in the texture image, resulting in smooth transitions between texture pixels.

The process of UV unwrapping is essential for defining an appropriate mapping from the 3D surface to the 2D texture space. UV unwrapping involves flattening the 3D geometry into a 2D representation while minimizing distortions such as stretching or compression, especially in regions with high curvature. The texture mapping function $\mathcal{T}$ should aim to minimize distortion by optimizing for area and angular preservation in the UV space.

Chapter 4

Methodology

In this section, we present our proposed methodology for addressing the semantic segmentation of non-manifold meshes. Section 4.1 provides an overview of the general structure of the architecture of the proposed model. In Section 4.2,provides a detailed account of the clustering mechanism and the generation of local patches. Section 4.3 we describe the process of feature extraction and the propagation of input features through the model. Section 4.4 offers an in-depth explanation of the model's local and global components. Section 4.5 presents the classification head. Lastly, Section 4.6 outlines the training procedure employed.

4.1 Overview

The input of our method consists of a textured 3D triangulated mesh $\mathcal{M} = (\mathcal{V}, \mathcal{F}, \mathcal{T}, \mathcal{I})$, where $V = \{v_i \mid v_i \in \mathbb{R}^3\}$ is the set of vertices, each corresponding to a point in 3D space, and $F = \{f_i \mid f_i = (v_{a,i}, v_{b,i}, v_{c,i}),\ v_{a,i}, v_{b,i}, v_{c,i} \in V\}$ denotes the set of triangular faces, where each face f_i is defined by three vertices $v_{a,i}, v_{b,i}, v_{c,i}$. The texture is contained in a texture image $\mathcal{I} \in \mathbb{R}^{H \times W \times 3}$, and the faces are linked to their texture by texture coordinates, defined as a set of triplets $\mathcal{T} = \{[(x_{a,i}, y_{a,i}), (x_{b,i}, y_{b,i}), (x_{c,i}, y_{c,i})] \, \forall f_i \in \mathcal{F}\}$; each triplet contains the coordinates of the vertices of the corresponding face in the texture image $\mathcal{I}$, and the pixels inside the triangle given by these coordinates represent the texture of the face.

The primary objective of our model is to perform semantic segmentation, i.e. to assign each face of the mesh to one of a set of predefined classes. The model output is represented as $\hat{\mathbf{Y}} \in \mathbb{R}^{|F| \times N_C}$, where $|F|$ is the number of faces and N_C is the number of predefined classes; $\hat{\mathbf{Y}}$ thus contains a vector of class scores for every face of the mesh, and the class label is defined as the class having the maximum score.

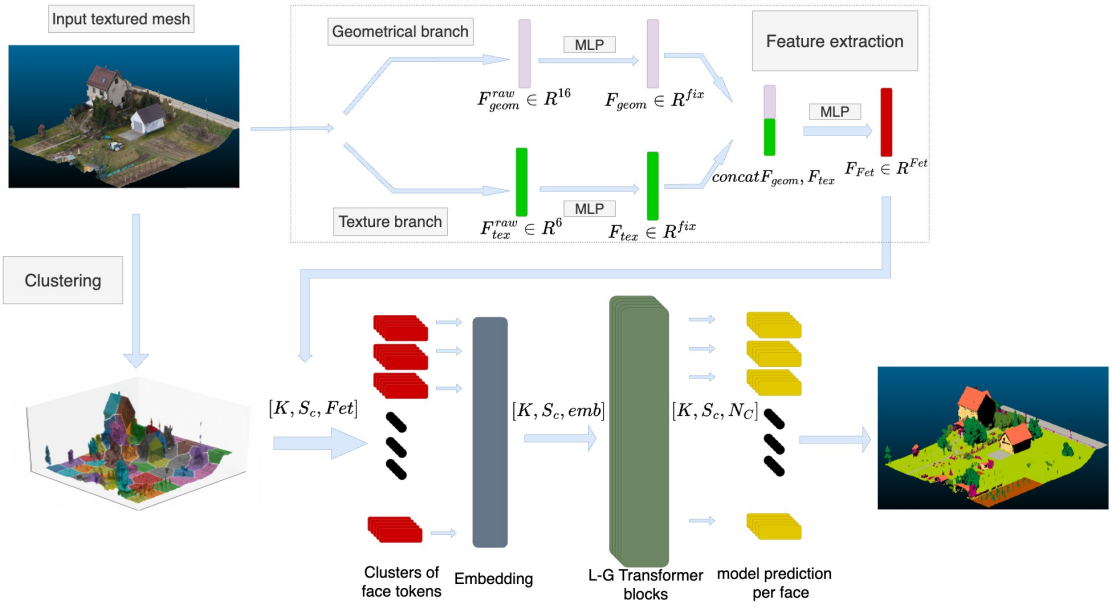


Figure 4.1: General architecture of our model. The feature extraction branch extracts a feature vector for every face, considering textural and geometrical information. K-means is used to generate clusters of faces. The face feature vectors are structured according to the clusters, and then they are passed on to a series of L-G transformer blocks. The output of the final block is processed by a classification head to yield class predictions. Numbers in brackets indicate the dimensionality of the tensors passed on to the subsequent blocks.

For 3D mesh data, each entity—such as a face or vertex—lacks any inherent, pre-defined features. This limitation necessitates the precomputation of specific features for each entity of interest. generally, any arbitrary 3D mesh is limited to geometric attributes, as detailed in Section 4.3.1. For instance, geometric attributes—such as face normals, curvatures, and angles—can be computed for individual faces. However, this preprocessing step can be computationally expensive. Therefore, to streamline efficiency, we prioritize a minimal yet effective feature representation per face.

Beyond geometric attributes, textured meshes offer an additional source of information: textural features. These can be particularly advantageous in cases where geometric data alone may be insufficient to distinguish between classes, such as low vegetation and bare soil, where textural cues are critical for accurate classification. We represent textural information as a set of pixel-based features, as outlined in 4.3, which provides complementary data to geometric features.

When combining geometric and textural features, their simple concatenation can lead to imbalanced feature representations, where one feature type dominates the other. To address this, we propose a dedicated feature extraction branch 4.3 that integrates both geometric and textural features in a balanced manner, ensuring each contributes effectively to the final representation of each face. This approach enables a more nuanced and robust representation, accommodating both sources of information and enhancing the overall expressiveness of the mesh data for downstream tasks.

As previously mentioned, we rely on Transformer networks to process 3D mesh faces due to their inherent order-invariant nature, allowing us to bypass concerns about the topological relationships that constrain traditional manifold-based mesh structures. Unlike other architectures that are sensitive to the ordering and connectivity of data points, Transformers offer the flexibility to operate on unstructured data without such restrictions. However, the quadratic complexity $O(N^2)$ of the standard Transformer architecture, driven by the pairwise attention between all faces, poses significant computational challenges when applied to large 3D meshes, making full attention impractical.

To mitigate this issue, we propose a hierarchical approach that first learns features within *local patches* of the mesh. Given the irregular, non-grid-based structure of mesh data, generating these local patches is not straightforward. Instead of adhering to a fixed patch size or structure, we leverage the Transformer's ability to process sequences of varying lengths. This flexibility allows us to dynamically create patches of varying sizes using a clustering algorithm—specifically, *k-means clustering* in our case as describe in detail in 4.2.

Once the local patches are formed, the Transformer is applied to each patch independently terms as local block, learning representations within these localized regions. This process is conceptually similar to the convolution operation in CNNs, which captures local context, but with the added advantage that Transformers, through shared weights, apply the self-attention mechanism across patches without being constrained by the local neighborhood structure. This enables the model to effectively capture low-frequency features and local context within each patch, while maintaining the flexibility and capability of Transformer-based representation learning.

Learning local context is crucial, especially for semantic segmentation tasks on mesh faces, where detailed understanding of short-range interactions between neighboring faces is essential. However, grasping global context is equally important to capture broader structural relationships. To achieve this, we introduce a learnable global token that is concatenated with the tokens of each cluster, effectively summarizing the features of an entire cluster. This token is then utilized within a global transformer block to facilitate cross-cluster interactions.

The combination of these two components—the local transformer block for intra-cluster interactions and the global transformer block for inter-cluster attention—is termed the L-G block, as detailed in Section 4.4. This sequential integration of local and global attention enables a comprehensive understanding of both fine-grained and global patterns, enhancing the segmentation capabilities of the model.

4.2 Clustering

In transformer networks, the input is required to be in the form of a sequence of tokens. When the data consists of a 3D mesh and the objective is to assign labels to each face, treating all faces as input tokens becomes impractical. This limitation arises due to the quadratic complexity of the attention mechanism in transformers, which scales as $O(n^2)$, where n represents the number of tokens (or faces, in this case). As the number of faces increases, the computational cost grows prohibitively, making such an approach inefficient for large meshes. Therefore, strategies to reduce token complexity or sparsify the attention mechanism are essential to ensure tractable computation.

To adapt transformers for data modalities other than language, various approaches have been proposed. A well-known method to address the complexity of attention is the Vision Transformer (ViT) (Dosovitskiy et al., 2020), which first splits an image into small patches, where each patch serves as a token. However, this approach is not directly applicable to the domain of 3D meshes due to two main reasons.

First, the data structure of a 3D mesh is irregular and not grid-based, making it impossible to create regular and fixed patches of faces. Second, local features are crucial for representation learning, and simply dividing the mesh into patches of arbitrary sizes can overlook important local feature aggregation.

To address these two challenges, as illustrated in Figure 4.2, local patches of faces are first generated using k-means clustering. Let V represent the set of vertices of the input mesh $G = (V, E, F)$, where E denotes the set of edges and F the set of faces. The vertices V are partitioned into k clusters, $C_1, C_2, \ldots, C_k$, using the k-means algorithm based on their spatial proximity. Each cluster corresponds to a local region of the mesh. Mathematically, the objective of k-means is to minimize the sum of squared distances between each vertex $v_i \in V$ and the centroid of the cluster it belongs to, given by:

$$\min \sum_{i=1}^{k} \sum_{v \in C_i} \|v - \mu_i\|^2$$

where μ_i is the centroid of cluster C_i. These clusters define the local patches, which preserve the local geometry and features of the mesh.

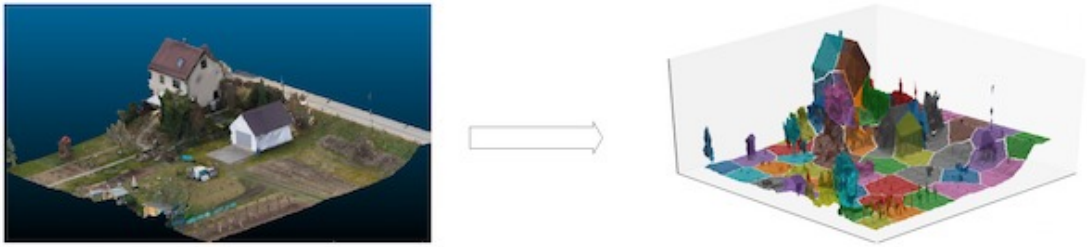


Figure 4.2: Clustering results of the 3D mesh faces. Each color represents a distinct cluster, which serves as a local patch for input into the model.

Once clusters are formed based on the 3D positions of the vertices, we proceed to group the faces into corresponding clusters. Specifically, a face is assigned to the cluster to which the majority of its vertices belong. For a face f with vertices v_1, v_2, v_3, if all vertices share the same cluster C_i, the face is trivially assigned to C_i.

In the case of border faces, where the vertices belong to different clusters, we apply a majority voting scheme. The face is assigned to the cluster containing the majority of its vertices. Formally, if a face has two vertices $v_1, v_2 \in C_i$ and the third vertex $v_3 \in C_j$ where $i \neq j$, the face is assigned to C_i due to the majority of vertices being in C_i.

However, in the edge case where each vertex of the face belongs to a different cluster (i.e., $v_1 \in C_i, v_2 \in C_j, v_3 \in C_k$ where $i \neq j \neq k$), the face is randomly assigned to one of the three clusters.

This process ensures that each face is consistently assigned to a cluster based on its vertex composition, resolving ambiguities in cases of vertex-sharing between clusters.

Once all the faces are assigned to clusters, the input mesh G can be represented as $[C, S, F_{\text{feat}}]$, where C denotes the number of clusters (a hyperparameter), S represents the number of faces in each cluster, which may vary across clusters, and F_{feat} is the feature vector associated with each face. This representation serves as the input to the model.

After that, each face is assigned to the cluster The resulting representation of the mesh faces is given by

$$F = \bigcup_{k=1}^{K} C_k,$$

where $C_k = \{f_{k1}, f_{k2}, \ldots, f_{kN_k}\}$, f_{kj} is the j^{th} face in cluster C_k and N_k is the number of faces in that cluster.

4.3 Feature extraction

Textured meshes contain two types of information: (1) textural information and (2) geometrical information. Thus, the feature extraction branch comprises two distinct sub-branches: the first sub-branch is dedicated to extracting geometrical features F_{geom}, which are derived from hand-crafted features designed to capture the intrinsic structure of the mesh geometry. The second sub-branch focuses on extracting textural features F_{tex}, which capture detailed surface characteristics.

The outputs of these sub-branches are concatenated along the feature dimension to form a combined feature vector, represented as $[F_{\text{tex}}, F_{\text{geom}}]$. This concatenated vector is then passed through a multi-layer perceptron (MLP), refining and transforming it into the final feature representation $F_{\text{Fet}} \in \mathbb{R}^{Fet}$ for each face in the mesh. The geometric and texture branches are discussed in Sections 4.3.1 and 4.3.2, respectively.

4.3.1 geometric branch

This branch is specifically designed to capture the geometrical properties of each face in the mesh. We begin by defining a set of seven handcrafted features for each face, which include the face's area, its normal vector, and the angles formed between the face and its neighboring faces. These features constitute the vector $F^{\text{hc}}_{\text{geom}} \in \mathbb{R}^7$.

In addition, the raw 3D coordinates of the vertices, normalized relative to the center of each cluster and represented as a nine-dimensional vector, are concatenated with the handcrafted geometric features. This results in a 16-dimensional feature vector, $F^{\text{raw}}_{\text{geom}} \in \mathbb{R}^{16}$. The inclusion of these vertex coordinates is crucial, as it introduces positional information that serves as an inductive bias in the attention mechanism, enhancing the network's ability to capture spatial context. Furthermore, these coordinates may assist in extracting high-level geometric features that go beyond the initial handcrafted descriptors.

The combined feature vector is then processed by a MLP, yielding the final geometric feature vector $F_{\text{geom}} \in \mathbb{R}^{\text{fix}}$ with a fixed dimensionality fix.

4.3.2 Textural branch

The textural information in our dataset consists of RGB values derived from corresponding images, as discussed in detail in Section 3.2.5. For each face, three

UV coordinates indicate the image coordinates of the face's vertices. While these coordinates provide direct RGB values for three pixels, they do not fully capture the color representation of the entire face, as there may be additional pixel regions between the three vertex pixels. To extract all RGB values associated with each face, we first generate a grid of candidate pixels based on the initial vertex pixels. We then calculate the barycentric coordinates of each pixel to determine whether it lies inside or outside the triangle. The pixel values corresponding to each face

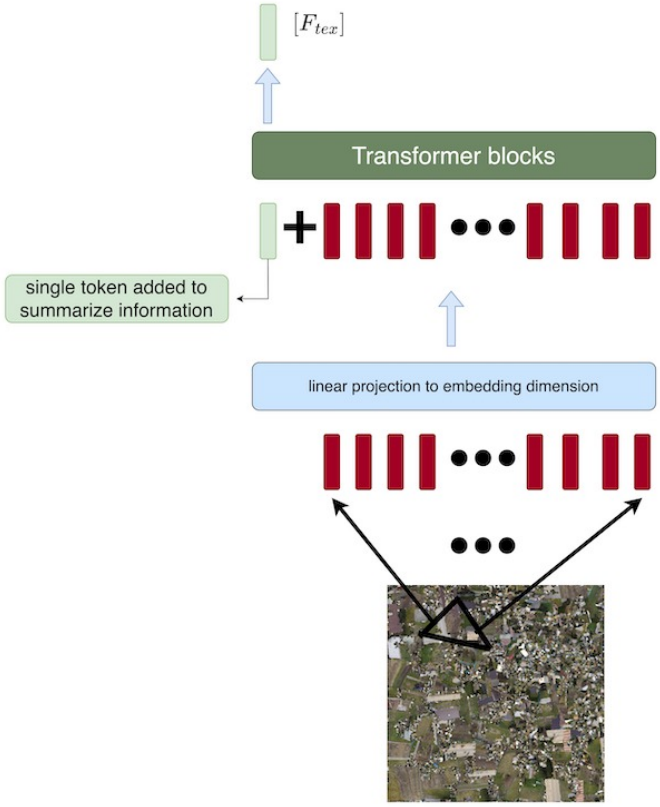


Figure 4.3: Architecture for Textural Feature Extraction. Each triangular face of the mesh is represented by a corresponding set of pixels shown in red that undergo processing through a transformer block. This operation distills the pixel information into a single token, which is subsequently integrated into the main model.

are represented as $[P, C]$, where P is the number of pixels and C is the number of channels. Since P can vary for different faces, it is necessary to standardize the dimensionality of textural information across all faces. Two approaches are considered for this: the first is a straightforward method that relies on the statistical properties of the pixel values, specifically the mean and standard deviation of

each channel. The second approach show in figure 4.3 involves using a transformer network to summarize the pixel information into a single token. However, due to hardware limitations, training the transformer network was not feasible, and we rely on the statistical approach.

The resulting feature vector for each face is a six-dimensional vector $F_{\text{tex}}^{\text{raw}} \in \mathbb{R}^6$. This vector is subsequently passed through a multi-layer perceptron (MLP), producing a final texture feature vector $F_{\text{tex}} \in \mathbb{R}^{\text{fix}}$ with the same dimensionality as the output from the geometrical feature branch.

While these handcrafted features provide a basis for representing texture, they may be limited in capturing the full complexity and richness of the texture information inherent in each face. In future work, we plan to employ transformers to extract texture features directly, leveraging their capacity for detailed and high-dimensional feature extraction.

4.4 L-G transformer branch

The input to the L-G transformer branch is represented as

$$X = \left\{ C_k^F = \{F_{\text{Fet},k1}, F_{\text{Fet},k2}, \ldots, F_{\text{Fet},kN_k}\} \right\}_{k=1}^{K},$$

where C_k^F is a tensor containing the feature vectors of all faces in cluster C_k within the K clusters, and $F_{\text{Fet},ki}$ denotes the feature vector associated with face f_{ki} in that cluster.

Although transformers are theoretically capable of handling sequences of arbitrary lengths, practical implementation requires uniform dimensions across a batch. To address this, we pad each cluster's input to match the length of the longest sequence of faces. A binary attention mask is employed to prevent interactions between padded tokens and face tokens, assigning a value of 1 to face tokens and 0 to padded tokens.

An attention mask is a binary matrix that indicates which tokens should participate in the attention mechanism. In this case, it prevents the model from attending to padded tokens by assigning a mask value of zero to them and one to the valid face tokens. This ensures that attention is computed only between meaningful face tokens, ignoring the padded elements, allowing the transformer to focus on relevant data.

Consequently, the input can be represented as a tensor $X \in \mathbb{R}^{K \times S_c \times \text{Fet}}$, where S_c is the maximum number of faces in any cluster.

This input X is then passed through an embedding block, which uses a MLP to transform each token from the dimension Fet to an embedding dimension emb, resulting in a tensor $E \in \mathbb{R}^{K \times S_c \times \text{emb}}$.

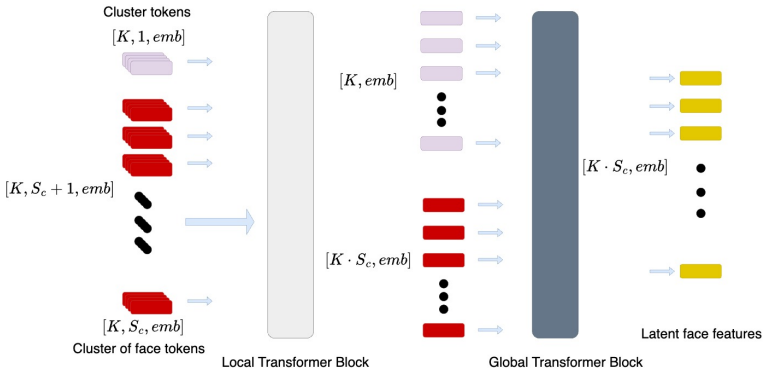


Figure 4.4: The L-G transformer block is comprised of two distinct components. The local transformer block focuses on learning fine-grained details within face clusters, generating two sets of sequences: cluster tokens (depicted in pink) and face tokens (depicted in red). Subsequently, the global transformer block processes these sequences through cross-attention mechanisms to effectively capture the global context.

The tensor E is subsequently processed through a sequence of L-G transformer blocks; the total number of blocks, which is a hyperparameter, is set to six in our experiments. Each L-G block is specifically designed to extract both local and global contextual information from the input mesh. This is achieved through a dual structure comprising a local transformer sub-block and a global transformer sub-block, each optimized for its respective function, as detailed in Sections 4.4.1 and 4.4.2.

An L-G block is defined as a pairing of one local and one global sub-block. The architecture of an L-G block is illustrated in Figure 4.4.

4.4.1 Local block

The local transformer block enables the model to capture fine-grained features within each cluster. By leveraging MHSA, the block focuses on key interactions between vertices or faces within the cluster, which is especially critical for non-manifold meshes, where accurate segmentation relies heavily on local feature aggregation. Additionally, the local block serves a secondary role by effectively summarizing local geometric and textural features into a cluster token. This token is later used by global transformer blocks.

Inspired by (Devlin et al., 2019), a single learnable *cluster token* is concatenated with the face tokens in each cluster. This modifies the input tensor from $E \in \mathbb{R}^{K \times S_c \times \text{emb}}$ to $E_{\text{cont}} \in \mathbb{R}^{K \times (S_c+1) \times \text{emb}}$. The local transformer block thus learns

high-level features within the faces of a cluster while summarizing them into this dedicated *cluster token*, which will subsequently be utilized in the global block for long-range feature aggregation.

Each local transformer block follows the standard transformer architecture described in Section 3.1.6, utilizing shared learnable parameters to project the input features into key, query, and value representations for each cluster. Multi-head self-attention (MHSA) is then applied, where the attention mechanism operates within the same cluster, using the projected keys, queries, and values to compute attention weights. This captures intra-cluster dependencies, allowing the model to focus on local features. The output is subsequently passed through layer normalization and a feed-forward neural network with residual connections, ensuring stability and enhanced gradient flow. We can express the output of the local block as having the same dimensions as its input, $\mathbf{X} \in \mathbb{R}^{B \times C \times (S_c+1) \times \text{emb}}$. The output of this block represents higher-level features learned through local feature aggregation within clusters.

The output from this block undergoes layer normalization and is passed through a MLP composed of two fully connected layers with ReLU activations, integrated with residual connections.

The resulting output of the local block retains the input shape as $Z_{\text{local}} \in \mathbb{R}^{K \times (S_c+1) \times \text{emb}}$, representing enriched latent local features.

4.4.2 Global block

In contrast to the local block, the global block serves a complementary function within the L-G architecture. While the local block is dedicated to aggregating and learning fine-grained features from localized clusters, capturing the global context of the input mesh is important. The global block enables the model to account for long-range dependencies and interactions across the entire mesh, potentially enriching the representations learned from local features. By incorporating global context, the model can identify patterns that span broader spatial regions, which is crucial for tasks requiring an understanding of both local details and global structures.

To begin, the output Z_{local} from the Local Block is split into two components: a set of cluster tokens $Z_{\text{clusters}} \in \mathbb{R}^{K \times \text{emb}}$ and a set of face tokens $Z_{\text{faces}} \in \mathbb{R}^{(K \cdot S_c) \times \text{emb}}$ (see Fig. **??**). In the Global Block, cross-attention is applied, allowing the model to compute attention between cluster tokens and face tokens, thus capturing inter-cluster dependencies.

Within the cross-attention block, the sequence Z_{clusters} of cluster tokens is used to generate the key and value matrices, while the sequence Z_{faces} of face tokens provides the queries. Similar to the Local Block, the Global Block follows the

original transformer architecture (Vaswani, 2017), but with cross-attention rather than self-attention. This results in a tensor $Z^o_{\text{global}} \in \mathbb{R}^{(K \cdot S_c) \times \text{emb}}$.

The tensor Z^o_{global} is then reshaped back to the original cluster-based structure, which is possible since the association of faces with clusters is known. This yields an output tensor $\mathbf{Z}_{\text{global}} \in \mathbb{R}^{K \times S_c \times \text{emb}}$, which serves as the input to the subsequent block.

Each global transformer block is structured similarly to the local block regarding its components. However, instead of self-attention, it utilizes multi-head cross-attention, enabling the attention mechanism to function between the cluster tokens and the mesh faces. Following this, the output is processed through layer normalization and a feed-forward neural network with residual connections.

The output of the global block will match the size of the queries, as it is calculated based on the cross-attention mechanism. Consequently, the output from the global block will be $\mathbf{X} \in \mathbb{R}^{B \times \sum_{i=1}^{C} S_i \times \text{emb}}$.

To analyze how the global block reduces complexity while learning the global context, let us assume the total number of faces in the 3D mesh is F, calculating all interactions between faces using a traditional attention mechanism would have a complexity of $O(F^2)$. However, by utilizing cluster tokens and faces, this complexity is reduced to $O(F \times C)$, where C is the number of clusters—significantly smaller than the number of faces. This reduction makes it feasible to capture interactions between clusters and learn the global context of the mesh efficiently.

4.5 Classification

The output Z_{global} from the final L-G transformer block undergoes further processing in a dedicated classification head to generate class predictions for each face in the mesh. This classification head is composed of a MLP that projects each face's feature vector into a space of raw class scores of dimension N_C, where N_C denotes the number of possible classes.

These raw class scores are subsequently normalized using the softmax function to produce probabilistic class scores for each face. This normalization results in a tensor of shape $K \times S_c \times N_C$, where each entry represents the probability distribution over classes for a face within a cluster. This tensor encodes class probabilities across clusters, enabling direct interpretability of each face's likelihood to belong to specific classes.

Following this step, padding tokens, introduced to ensure uniform sequence length across clusters, are removed based on the attention mask. This removal restores the tensor to its original, cluster-based structure, ensuring that only valid face tokens are retained. The final, fully processed output is represented as $\hat{\mathbf{Y}}$, which aligns with the initial input mesh structure as outlined in Section 4.1.

4.6 Network Training

Training is based on the categorical cross-entropy loss L_{ce} to measure the divergence between the predicted probabilities for each class associated with individual faces and their corresponding true labels. It aggregates the loss across all faces in the mesh and all classes:

$$L_{ce} = \sum_{i=1}^{|F|} \sum_{c=1}^{N_C} y_{ic} \cdot \log(\hat{y}_{ic}), \tag{4.1}$$

where $|F|$ represents the total number of faces in the mesh, N_C denotes the total number of classes, y_{ic} is the ground truth binary indicator face i to belong to class c ($y_{ic} = 1$) or not ($y_{ic} = 0$), and $\hat{y}_{ic}$ is the predicted probability for the face i to belong to class c.

Chapter 5

Experiment

5.1 Dataset

To evaluate and validate our proposed model, we utilized the Hessigheim 3D (H3D) dataset Kölle et al. (2021), which includes both high-resolution point clouds and non-manifold textured 3D meshes. Our study focuses solely on the mesh modality (H3D-Mesh). The dataset consists of three subsets: training, validation, and test. However, since the test set is not yet publicly available or published, we rely on the validation set for model validation and examination. Therefore, the training and validation sets, comprising 9,236,637 mesh faces for training and 2,577,554 faces for validation, covering surface areas of 36,445 m^2 and 8,050 m^2 respectively, are used in this thesis. The dataset includes 11 finely labeled semantic classes: Low Vegetation, Impervious Surface, Vehicle, Urban Furniture, Roof, Façade, Shrub, Tree, Soil/Gravel, Vertical Surface, and Chimney. These labels are transferred from the point cloud to the mesh using a geometry-driven approach. However, approximately 40% of the mesh faces remain unlabeled, primarily in regions where the mesh exceeds the annotated point cloud area. The high-resolution mesh, with its detailed surface representation and texture information, is particularly suited for fine-grained semantic segmentation, making it ideal for benchmarking and validating.

5.2 Evaluation metrics

In the context of classification, the performance of a model is often evaluated using four fundamental components: *True Positives (TP)*, *False Positives (FP)*, *False Negatives (FN)*, and *True Negatives (TN)*. These components are derived from comparing the predicted class labels to the actual ground truth.

- **True Positives (TP)**: These are instances where the model correctly predicts the positive class. In other words, the model classifies an instance as positive, and it is indeed positive according to the ground truth.
- **False Positives (FP)**: These are instances where the model incorrectly predicts the positive class. This happens when the model classifies an instance as positive, but it is actually negative. False positives are also referred to as *Type I errors*.
- **False Negatives (FN)**: These are instances where the model incorrectly predicts the negative class. In this case, the model classifies an instance as negative, but it is actually positive according to the ground truth. False negatives are also known as *Type II errors*.
- **True Negatives (TN)**: These are instances where the model correctly predicts the negative class. The model classifies an instance as negative, and it is indeed negative according to the ground truth.

These four components form the foundation for various performance metrics that evaluate the quality of a classification model.

5.2.1 Confusion Matrix

A *Confusion Matrix* is a tabular representation used to summarize the performance of a classification algorithm. It provides insight into the distribution of predicted and actual class labels, helping to identify not only the correct classifications but also the types of errors made by the model. For a binary classification problem, the confusion matrix is structured as follows:

$$\begin{bmatrix} \text{TP} & \text{FP} \\ \text{FN} & \text{TN} \end{bmatrix}$$

The rows of the confusion matrix represent the actual class labels (positive or negative), while the columns represent the predicted class labels. This matrix is an essential tool for visualizing the performance of a classification model, allowing us to calculate performance metrics such as *Precision*, *Recall*, *Accuracy*, and the *F1 Score*.

For multi-class classification problems, the confusion matrix is extended to a $CLs \times CLs$ matrix, where CLs represents the number of classes. Each cell in this matrix represents the number of instances where the true class label is i, and the predicted class label is j, allowing us to compute evaluation metrics for each class.

5.2.2 Mean F1 Score

The mean F1 score is a key metric for evaluating the performance of a classifier, particularly in cases of imbalanced datasets where certain classes may dominate. It is the harmonic mean of *precision* and *recall* for each class. For a specific class i, the F1 score is computed as follows:

$$\text{F1}_i = 2 \times \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

where:

- Precision_i is the ratio of correctly predicted instances of class i to the total number of instances predicted as class i. This can be expressed as:

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}$$

  where TP_i is the number of true positives for class i, and FP_i is the number of false positives for class i.

- Recall_i is the ratio of correctly predicted instances of class i to the total number of actual instances of class i. This is given by:

$$\text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}$$

  where FN_i is the number of false negatives for class i.

Thus, the F1 score for class i balances precision and recall as a harmonic mean, ensuring that both false positives and false negatives are equally accounted for.

To calculate the mean F1 score across all classes, we average the F1 scores of each class. Given CLs classes, the mean F1 score is calculated as:

$$\text{Mean F1} = \frac{1}{CLs} \sum_{i=1}^{CLs} \text{F1}_i$$

This average provides an overall evaluation of the classifier's ability to balance precision and recall across multiple classes, which is particularly useful in datasets with uneven class distributions.

5.2.3 Overall Accuracy

overall accuracy is one of the simplest and most widely used performance metrics. It measures the proportion of correctly classified instances among all instances

in the dataset. Formally, if the total number of samples is N, and the number of correctly predicted samples is $\sum \text{True Positives}_i$, then the overall accuracy is defined as:

$$\text{Accuracy} = \frac{\sum_{i=1}^{CLs} \text{True Positives}_i}{N}$$

This metric provides a general measure of the classifier's performance across all classes. However, in the presence of imbalanced classes, overall accuracy can be misleading. In such cases, models may achieve high accuracy by predominantly predicting the majority class correctly while performing poorly on minority classes.

For a balanced understanding of model performance, both the mean F1 score and overall accuracy should be reported together. The mean F1 score provides insight into how well the model balances precision and recall across all classes, while overall accuracy gives a broad overview of the model's correctness.

5.3 Experimental Setup

The Experimental setup we utilized includes the general network hyperparameters, trinaing setup , augmentation strategies and dropout.

The network's hyperparameters are tuned based on the results from the validation set. The number of L-G blocks is set to 6, with an embedding dimension of 256, and 300 clusters are used for the k-means clustering. The deep learning model is implemented using the PyTorch library Paszke et al. (2017).

Network Training The training process follows the approach outlined in **??**. The model's initial parameters are randomly initialized based on the method in Glorot and Bengio (2010), and the optimization is carried out using the SGD optimization method. The Adam optimizer Kingma and Ba (2014) is chosen with an initial learning rate of 0.0001, and a scheduler is applied with a step size of 1000 and a gamma value of 0.9. Each experiment is run for 100 epochs, and all training is performed on an NVIDIA A100 80GB GPU.

Dropout and Augmentation Settings To mitigate overfitting, we implement both dropout and tailored data augmentation strategies throughout training.

A dropout (Srivastava et al., 2014) rate of 0.1 is applied, where 10% of neurons are randomly deactivated during each forward pass. This stochastic removal of units prevents over-reliance on specific neurons, driving the network to capture more generalized and robust feature representations.

For data augmentation, we employaugmentation optimized for 3D mesh data, consisting of:

- **Rotation Range (45 degrees):** Meshes are randomly rotated by up to $\pm 45^{\circ}$ across all three axis, simulating diverse viewing angles and enhancing the model's ability to generalize across various object orientations.
- **Scale Range (2):** Random scaling between 0.5 and 2 ensures robustness to size variations, allowing the model to handle meshes with differing dimensions.
- **Noise Standard Deviation (0.01):** Gaussian noise with a standard deviation of 0.01 is added to vertex positions, simulating real-world sensor noise and ensuring resilience against minor geometric perturbations.

Together, dropout and augmentation enhance the model's ability to generalize to unseen data by introducing regularization and diversity during training.

5.4 prior methodology

We conducted a comprehensive series of experiments to rigorously evaluate our model and explore the efficacy of various configurations. However, it is important to note that the dataset's training and validation sets are publicly accessible, which restricts our ability to compare our model with other frameworks due to the unavailability of the test set. To facilitate a comparative analysis of our model's performance against existing methodologies, we aimed to train a model that achieves the highest results on the announced leaderboard for the dataset and subsequently evaluated it on the validation set as a baseline for comparison with our results.

Based on the announced results Kölle et al. (2021), the Random Forest classifier achieved the highest performance in terms of the mean F1 score. Therefore, in our initial experiment, we trained the Random Forest classifier using a specific combination of geometrical and textural features. For the geometrical features, we employed face normal vectors, face surface areas, and anglesand 3D vertex coordinate. For the textural information, we utilized pixel value statistics per channel, resulting in a 22-dimensional feature vector, as shown in the table5.3. Although the features utilized differ from those employed in Kölle et al. (2021), which included a broader array of inputs, we selected these particular features as they are also integral to our proposed model. This decision was made to facilitate a fair comparison, emphasizing our objective of assessing the model's capabilities with the same input. It can be inferred that incorporating additional descriptive

data would likely improve the model's performance. We used a RF with 100 trees and a maximum depth of 20. Nodes receiving fewer than 10 samples were not split further in the training process, and the Gini index was used to select the optimal separating surface in each node in the training procedure.

5.5 Model variant

In our analysis, we focused on three critical aspects: architectural design, input feature selection, and model parameter tuning. By systematically investigating these configurations, we aimed to optimize our model's performance and enhance our understanding of its capabilities based on the available training and validation datasets.

Name	Model	Features
NMF-L_1	NMF-L	$F_{geom}^{hc} \in \mathbb{R}^7$
NMF-L_2	NMF-L	$F_{geom}^{raw} \in \mathbb{R}^{16}$
NMF-L_3	NMF-L	$[F_{geom}^{raw}, F_{tex}^{raw}] \in \mathbb{R}^{22}$
NMF-L_4	NMF-L	$FEX(F_{geom}^{raw}, F_{tex}^{raw})$
NMF-LG_1	NMF-L+G	$F_{geom}^{hc} \in \mathbb{R}^7$
NMF-LG_2	NMF-L+G	$F_{geom}^{raw} \in \mathbb{R}^{16}$
NMF-LG_3	NMF-L+G	$[F_{geom}^{raw}, F_{tex}^{raw}] \in \mathbb{R}^{22}$
NMF-LG_4	NMF-L+G	$FEX(F_{geom}^{raw}, F_{tex}^{raw})$
RF	RF	$[F_{geom}^{raw}, F_{tex}^{raw}] \in \mathbb{R}^{22}$

Table 5.1: Overview of the experiments conducted. Name: the name by which an experiment is referred to in the text. Model: the model used (NMF-L: NoMeFormer with local blocks only, NMF-L+G: NoMeFormer with local and global blocks; RF: Random Forest. Features: features used as input. $FEX(\cdot)$ indicates the use of the feature extraction branch to generate a feature vector for each face.

In our experimental setup, we designed two main groups of experiments to evaluate the performance of NoMeFormer under varying architectural configurations. The first group includes experiments where only the six local attention blocks are activated within the Local-Global (L-G) Transformer branch. These experiments, denoted as *NMF-L* in Table 5.1, restrict attention to local clusters, enabling us to examine the effectiveness of local feature aggregation in isolation. In the second group, both local and global attention blocks are employed, denoted as *NMF-LG* in Table 5.1. The inclusion of global attention allows information exchange across clusters, providing an assessment of the added value of cross-cluster communication and the influence of global context on model performance. Each

group consists of four experimental variations, distinguished by how the feature vectors for the mesh faces are defined.

For the first experiment in each group, labeled *NMF-L*$_1$ and *NMF-LG*$_1$, the model is provided only with a set of seven hand-crafted geometric features, $F_{geom}^{hc} \in \mathbb{R}^7$, representing fundamental shape information. These features are then passed through a MLP), which maps them to a higher-dimensional feature space F_{Fet} wih a dimensionality of $Fet = 64$, establishing a baseline for feature extraction based purely on geometric descriptors.

In the second variant, *NMF-L*$_2$ and *NMF-LG*$_2$, we extend the geometric representation by incorporating the nine vertex coordinates of each face into the feature set, enhancing spatial information. This leads to a raw feature vector, $F_{geom}^{raw} \in \mathbb{R}^{16}$, which is also processed by an MLP to achieve a consistent feature dimensionality of $Fet = 64$. This raw 3D data serves two key purposes: first, as positional encoding, ddressing the lack of inherent order awareness in the attention mechanism.

Without explicit positional information, the model cannot distinguish between features originating from different regions of the mesh. This experiment allows us to explore whether incorporating raw 3D values introduces an inductive bias in the model, enabling it to capture the importance of input data order and spatial structure. Moreover, raw 3D data plays a complementary role. While handcrafted geometrical features are used as part of the input, the inclusion of 3D coordinates allows the model to learn additional geometrical representations that are not explicitly captured by handcrafted features. This aligns with the core concept of representation learning in deep learning, where models typically rely on raw data without predefined feature engineering.

The third experimental variant, denoted as *NMF-L*$_3$ and *NMF-LG*$_3$, enriches the feature representation further by integrating textural attributes with the geometric descriptors. Here, we concatenate the raw hand-crafted texture features $F_{tex}^{raw} \in \mathbb{R}^6$, discussed in detail in Section 4.3.2, with the expanded geometric vector $F_{geom}^{raw} \in \mathbb{R}^{16}$ from the second variant. This combined vector leverages both shape and radiometric information to produce a more comprehensive feature representation. The concatenated vector is passed through an MLP, yielding the final feature vector F_{Fet} with a dimensionality of $Fet = 64$, allowing the model to exploit both structural and radiometric cues for enhanced context representation.

In the fourth and final variant for each group, denoted as *NMF-L*$_4$ and *NMF-LG*$_4$, we employ the comprehensive feature extraction branch outlined in Section 4.3. This branch utilizes all available input data, processing it through a structured pipeline to yield a refined feature vector with both fix and Fet set to 64. This experiment serves two purposes: first, to once again evaluate the effectiveness of textural information as an additional input; and second, through comparison with Experiment 3, to assess the impact of the feature extraction

branch, which aims to represent both sets of features equally.

Together, these experiments provide insight into the roles of local versus global attention mechanisms and the relative contributions of geometric, positional, and radiometric features in improving mesh segmentation performance.

Chapter 6

Results and Discussion

This section presents and evaluates the results obtained from our experiments. Each experiment was repeated three times, and we report the mean and standard deviation of the mean F1 score, and overall accuracy to assess the robustness of the results, irrespective of initialization. To ensure a fair comparison, the training setup for each experiment was held constant as outlined in 5.3, maintaining consistency across all evaluations.

6.1 Comparison with RF

Name	$mF1$ [%]	OA [%]
NMF-LG_2	**58.9**	**61.1**
RF	31.1	39.3

Table 6.1: Mean F1 score ($mF1$) and Overall Accuracy (OA) results were obtained for the RF and the best variant of NoMeFormer Name: name of the experiment according to Table 5.1

Table 6.1 shows the classification results of the mentioned two models, comparing the baseline RF model with The best NoMeFormer variant, NMF-LG_2, which outperforms the RF by 25.8% in mean F1 score and by 21.8% in OA.

This performance gain is consistent with expectations, as NoMeFormer is transformer based architecture, equipped with L-G trans- former blocks, enables effective feature aggregation across both local and global scales. In contrast, the RF, limited by its reliance on predefined handcrafted features, lacks the capacity for representation learning and robust feature aggregation, thereby constraining its ability to capture the intricate, high-level pat- terns present in the data. NoMe-

Former's representation learning framework effectively tackles this complexity by employing comprehensive feature aggregation.

However, as discussed, the models currently operate with a minimal feature set, which differs from that used in (Kölle et al., 2021). This inherently favors deep learning models, such as transformers, due to their robust representation learning capabilities. Given this setup, it is anticipated that the RF model will exhibit improved performance as feature complexity increases. Moreover, as increasingly sophisticated features are provided as input to both models, we expect the performance gap to narrow further.

6.2 Ablation study

Name	$mF1$ [%]	OA [%]
NMF-L_1	45.9 ±0.2	52.0 ±0.2
NMF-L_2	49.2 ±0.5	53.1 ±0.2
NMF-L_3	42.7 ±1.0	48.9 ±0.5
NMF-L_4	46.8 ±0.6	50.6 ±0.2
NMF-LG_1	50.3 ±0.3	53.9 ±0.2
NMF-LG_2	**58.9** ±0.5	**61.1** ±0.2
NMF-LG_3	49.5 ±0.9	53.7 ±0.4
NMF-LG_4	52.4 ±0.9	56.8 ±0.2

Table 6.2: Mean F1 score ($mF1$) and Overall Accuracy (OA) results obtained for the experiments involving different varinat of NoMeFormer models. Name: name of the experiment according to Table 5.1.

In comparing the experimental results across two configurations—one employing both local and global blocks in the L-G Transformer (NMF-LG) and another using only local blocks (NMF-L)—the benefit of integrating global blocks is apparent. Across all four variants with different feature vector definitions, the networks with combined local-global blocks consistently outperform those with local blocks alone by a significant margin, with improvements in $mF1$ ranging from 4.4% to 9.7%. This underscores the importance of incorporating long-range interactions via global blocks and cluster tokens. By encoding the representation of feature sets into a single token and propagating it across the remaining faces in the mesh, the model efficiently captures global context, which directly enhances performance in the classification task. These findings further confirm the suitability of the L-G Transformer blocks, as outlined in Section 4.4, for the semantic segmentation of 3D meshes.

Model Variant	Low Vegetation	Impervious Surface	Vehicle	Urban Furniture	Roof	Façade	Shrub	Tree	Soil	Vertical Surface	Chimney
NMF-L_2	75.66	70.22	46.32	56.61	80.70	68.24	31.31	84.86	17.36	53.16	57.95

Table 6.3: Class-wise F1 Scores for the Optimal Variant of the NoMeformer (NMF-L_2)

From the perspective of input feature impact, the baseline model, which uses only seven handcrafted features in the local-only configuration (*NMF-*L_1), achieved an $mF1$ of 45.9, this set of feature shows a marked improvement of 9.7% in $mF1$ when global blocks are introduced (*NMF-*LG_1), specifically for this feature set.

The evaluation of the influence of 3D vertex coordinates, specifically in the variants (*NMF-*L_2) and (*NMF-*LG_2), on face classification reveals notable enhancements in model performance across both configurations. Specifically, the integration of the local block yields a classification performance improvement of 3.3%, while the inclusion of the global block results in a substantial increase of 8.6%, outperforming all other model variants by a significant margin. These results underscore the efficacy of leveraging positional information to introduce an inductive bias, enabling the model to better discern the origins of input features. Further analysis of the class-wise F1 scores for this variant is depicted in Table 6.3. The classes with a higher number of instances, such as "tree" and "low vegetation," achieve the highest F1 scores, while some classes, like "soil" and "underbrush," exhibit lower scores. This is primarily due to their similarity in geometrical features with the "low vegetation" class.

The results in Table 6.2 show that integrating handcrafted textural features did not yield the expected results. While it was anticipated that textural features would improve classification—particularly in cases where geometrical features are insufficiently descriptive to distinguish between certain classes, such as *Soil* and *Grass*—the inclusion of textural data actually resulted in a reduction of the $mF1$ score. Specifically, concatenating the hand-crafted geometrical and textural features (*NMF-*LG_3) leads to a notable 9.4% drop in $mF1$ performance compared to the version trained without textural information, *NMF-*LG_2; the overall accuracy (OA) is also reduced by 6.4%.

While introducing the feature extraction branch (*NMF-*LG_4) helps mitigate performance degradation, boosting $mF1$ by 2.9% and OA by 3.1%. However, these improvements are still significantly lower than the performance achieved by the variant *NMF-*LG_2, which excludes textural information (6.5% in $mF1$, 4.3% in OA). This reduction can likely be attributed to the method of incorporating textural information into the model. Specifically, textural data were represented statistically through means and standard deviations per band, which inadequately captures the intricate and informative aspects of texture. Consequently, this approach likely introduces noisy features that poorly correlate with the output variable, hindering the model's ability to effectively learn the relationships between relevant texture features and the target variable. This limitation underscores the need for an improved textural feature extraction branch in future work.

In addition to evaluating performance metrics, we performed a thorough analysis of the most prevalent misclassification errors observed in the models. A notable

	LV	IS	VE	UF	RO	FA	SH	TR	SO	VS	CH
LV	188159	45395	232	4884	1320	2374	2855	1265	12983	29	0
IS	47181	159459	305	2928	7381	4839	804	868	2105	1092	0
VE	2001	5578	8822	4559	2408	1891	345	1089	1	273	37
UF	6951	5240	901	29842	6504	8954	5659	21339	198	414	24
RO	13159	7604	178	2023	188854	9974	1031	14905	119	132	495
FA	5977	2415	446	5903	6927	94463	1700	8523	12	1257	37
SH	4945	2180	2119	18047	3716	5823	11460	9161	219	69	1
TR	5619	274	759	7641	11187	9471	2856	267231	41	92	24
SO	27910	7850	1	196	0	250	92	59	2079	0	0
VS	193	1883	4	1102	663	8009	87	2086	0	10256	0
CH	110	0	0	143	1141	3	2	619	0	1	1793

Table 6.4: Confusion Matrix for Classes. Abbreviations: LV = Low Vegetation, IS = Impervious Surface, VE = Vehicle, UF = Urban Furniture, RO = Roof, FA = Façade, SH = Shrub, TR = Tree, SO = Soil, VS = Vertical Surface, CH = Chimney.

challenge arose in distinguishing between geometrically similar classes, such as low-vegetation soil and shrub areas interspersed with urban furniture. This difficulty is illustrated in the confusion matrix 6.4 of model predictions, which reveals that almost all instances of the soil class were misclassified as low vegetation, and a significant number of shrub class instances were incorrectly identified as urban furniture. These misclassifications likely stem from two factors: (1) the close similarity of geometrical features across these classes and (2) an imbalance in class representation within the training set. This imbalance biases the model toward classes with more abundant examples, especially for geometrically similar classes, leading to overlapping feature representations and, subsequently, higher rates of misclassification.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we present NoMeFormer, a transformer-based network specifically developed to process arbitrary 3D meshes without imposing manifold constraints. This innovative framework addresses a crucial limitation of many existing deep learning models that struggle with non-manifold structures. Our approach efficiently captures fine-grained details, high-frequency patterns, and global context, essential for accurately segmenting complex geometries. A key component of NoMeFormer is the introduction of the Local-Global (L-G) block, a transformer architecture that sequentially processes both local and global dependencies. By leveraging this architecture, we can maintain the intricate relationships within local patches while also capturing broader contextual information across the entire mesh.

Moreover, NoMeFormer leverages a minimal set of input features, creating a streamlined processing pipeline that boosts efficiency and reduces the need for sophisticated handcrafted features. This design leads to a Achieving an mF1 score of 58.9 on the Hessigheim benchmark dataset, where our model demonstrates capabilities in semantic segmentation tasks. The results underscore the potential of NoMeFormer to redefine how non-manifold meshes are processed in various applications, ranging from cultural heritage preservation to autonomous navigation systems. As the field continues to evolve, our framework paves the way for future research and advancements in mesh processing, highlighting the importance of adaptable architectures capable of addressing the complexities of real-world data.

7.2 Outlook

However, despite these advancements, several limitations remain that warrant further investigation. A primary area for future research should focus on the integration of textural information. While we anticipated that incorporating this information would be highly beneficial, our findings indicate that it inadvertently compromises the network's classification performance. In this work, we relied on a static representation of textural information, which is the simplest form of integration. To enhance this aspect, we proposed an additional transformer block capable of training end-to-end with the rest of the network. This block would learn to summarize the textural information dynamically, potentially improving classification outcomes.

Nonetheless, incorporating this additional component into the network significantly increases the computational intensity of training. With the current hardware resources, implementing this solution proved challenging. Addressing this computational burden is essential for making the integration of textural information feasible in practical applications. Future work should aim to optimize this integration strategy, perhaps through more efficient architectures or advanced training techniques that can handle the increased complexity. By resolving these issues, we can unlock the full potential of NoMeFormer, enabling it to leverage textural information effectively while maintaining performance across various tasks and datasets.

Furthermore, considering the data-intensive nature of transformer models, pretraining on large and diverse datasets is crucial for allowing the model to develop more generalizable features. This pretraining phase is essential for equipping the network with a robust understanding of various mesh structures and characteristics. By exposing the model to a wide range of examples, it can learn to recognize important patterns and features that might not be present in smaller, task-specific datasets. This broad exposure can significantly enhance the model's performance, especially in real-world applications where data variability is common.

Exploring self-supervised pretraining techniques, as proposed by (He et al., 2022), offers a promising avenue for improving the model's ability to adapt to various mesh configurations. These techniques allow the model to learn from the data itself without requiring extensive labeled datasets, thus making it feasible to train on large volumes of unannotated mesh data. By leveraging this approach, NoMeFormer can better identify significant patterns and relationships within the data, ultimately leading to improved performance in downstream applications. This is particularly important in semantic segmentation tasks, where the availability of task-specific labeled data may be limited. By adopting self-supervised pretraining strategies, we expect to enhance the model's generalization capabilities, making it more effective in diverse scenarios and more resilient to variations in input data.

Bibliography

Eros Agosto and Leandro Bornaz. 3d models in cultural heritage: approaches for their creation and use. *International Journal of Computational Methods in Heritage Science (IJCMHS)*, 1(1):1–9, 2017.

Eman Ahmed, Alexandre Saint, Abd El Rahman Shabayek, Kseniya Cherenkova, Rig Das, Gleb Gusev, Djamila Aouada, and Björn E. Ottersten. A survey on deep learning advances on different 3d data representations. *arXiv: Computer Vision and Pattern Recognition*, 2018.

Dzmitry Bahdanau. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Caterina Balletti and Martina Ballarin. An application of integrated 3d technologies for replicas in cultural heritage. *ISPRS International Journal of Geo-Information*, 8(6):285, 2019.

Philip G Batchelor, PJ "Eddie" Edwards, and Andrew P King. 3d medical imaging. *3D Imaging, Analysis and Applications*, pages 445–495, 2012.

Christopher M Bishop and Hugh Bishop. *Deep learning: Foundations and concepts*. Springer Nature, 2023.

Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

Xiangxiang Chu, Zhi Tian, Yuqing Wang, Bo Zhang, Haibing Ren, Xiaolin Wei, Huaxia Xia, and Chunhua Shen. Twins: Revisiting the design of spatial attention in vision transformers. *Advances in neural information processing systems*, 34: 9355–9366, 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.

Qiujie Dong, Zixiong Wang, Manyi Li, Junjie Gao, Shuangmin Chen, Zhenyu Shu, Shiqing Xin, Changhe Tu, and Wenping Wang. Laplacian2mesh: Laplacian-based mesh understanding. *IEEE Transactions on Visualization and Computer Graphics*, 2023.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Herbert Edelsbrunner and John Harer. *Computational Topology: An Introduction*. 01 2010. ISBN 978-0-8218-4925-5. doi: 10.1007/978-3-540-33259-6_7.

Sabry F El-Hakim, J-A Beraldin, Michel Picard, and Antonio Vettore. Effective 3d modeling of heritage sites. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, pages 302–309. IEEE, 2003.

Oluwajuwon A Fawole and Danda B Rawat. Recent advances in 3d object detection for self-driving vehicles: A survey. *AI*, 5(3):1255–1285, 2024.

Yutong Feng, Yifan Feng, Haoxuan You, Xibin Zhao, and Yue Gao. Meshnet: Mesh neural network for 3d shape representation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 8279–8286, 2019.

Lucile Gimenez, Jean-Laurent Hippolyte, Sylvain Robert, Frédéric Suard, and Khaldoun Zreik. Reconstruction of 3d building information models from 2d scanned plans. *Journal of Building Engineering*, 2:24–35, 2015.

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

Meng-Hao Guo, Jun-Xiong Cai, Zheng-Ning Liu, Tai-Jiang Mu, Ralph R Martin, and Shi-Min Hu. Pct: Point cloud transformer. *Computational Visual Media*, 7:187–199, 2021.

Rana Hanocka, Amir Hertz, Noa Fish, Raja Giryes, Shachar Fleishman, and Daniel Cohen-Or. Meshcnn: a network with an edge. *ACM Transactions on Graphics (ToG)*, 38(4):1–12, 2019.

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the*

IEEE/CVF conference on computer vision and pattern recognition, pages 16000–16009, 2022.

Markus Herb, Tobias Weiherer, Nassir Navab, and Federico Tombari. Lightweight semantic mesh mapping for autonomous vehicles. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6732–3738. IEEE, 2021.

Qingyong Hu, Bo Yang, Linhai Xie, Stefano Rosa, Yulan Guo, Zhihua Wang, Niki Trigoni, and Andrew Markham. Randla-net: Efficient semantic segmentation of large-scale point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11108–11117, 2020.

Shi-Min Hu, Zheng-Ning Liu, Meng-Hao Guo, Jun-Xiong Cai, Jiahui Huang, Tai-Jiang Mu, and Ralph R Martin. Subdivision-based mesh convolution networks. *ACM Transactions on Graphics (TOG)*, 41(3):1–16, 2022.

Anastasia Ioannidou, Elisavet Chatzilari, Spiros Nikolopoulos, and Ioannis Kompatsiaris. Deep learning advances in computer vision with 3d data: A survey. *ACM computing surveys (CSUR)*, 50(2):1–38, 2017.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Michael Kölle, Dominik Laupheimer, Stefan Schmohl, Norbert Haala, Franz Rottensteiner, Jan Dirk Wegner, and Hugo Ledoux. The hessigheim 3d (h3d) benchmark on semantic segmentation of high-resolution 3d point clouds and textured meshes from uav lidar and multi-view-stereo. *ISPRS Open Journal of Photogrammetry and Remote Sensing*, 1:11, 2021. ISSN 2667-3932. doi: https://doi.org/10.1016/j.ophoto.2021.100001.

Alon Lahav and Ayellet Tal. Meshwalker: Deep mesh understanding by random walks. *ACM Transactions on Graphics (TOG)*, 39(6):1–13, 2020.

Dominik Laupheimer. *On the Information Transfer Between Imagery, Point Clouds, and Meshes for Multi-Modal Semantics Utilizing Geospatial Data*. PhD thesis, Institute of Photogrammetry and Remote Sensing, University of Stuttgart, Germany, 2022.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521 (7553):436–444, 2015.

Aaron WF Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. Maps: Multiresolution adaptive parameterization of surfaces. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 95–104, 1998.

Yaqian Liang, Shanshan Zhao, Baosheng Yu, Jing Zhang, and Fazhi He. Meshmae: Masked autoencoders for 3d mesh data analysis. In *European Conference on Computer Vision*, pages 37–54. Springer, 2022.

Minh-Thang Luong. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Jonathan Masci, Davide Boscaini, Michael Bronstein, and Pierre Vandergheynst. Geodesic convolutional neural networks on riemannian manifolds. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 37–45, 2015.

Daniel Maturana and Sebastian Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 922–928. IEEE, 2015.

Francesco Milano, Antonio Loquercio, Antoni Rosinol, Davide Scaramuzza, and Luca Carlone. Primal-dual mesh convolutional neural networks. *Advances in Neural Information Processing Systems*, 33:952–963, 2020.

Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

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 *NIPS-W*, 2017.

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 conference on computer vision and pattern recognition*, pages 652–660, 2017a.

Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in neural information processing systems*, 30, 2017b.

Guocheng Qian, Yuchen Li, Houwen Peng, Jinjie Mai, Hasan Hammoud, Mohamed Elhoseiny, and Bernard Ghanem. Pointnext: Revisiting pointnet++ with improved training and scaling strategies. *Advances in neural information processing systems*, 35:23192–23204, 2022.

Gernot Riegler, Ali Osman Ulusoy, and Andreas Geiger. Octnet: Learning deep 3d representations at high resolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3577–3586, 2017.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

Nicholas Sharp, Souhaib Attaiki, Keenan Crane, and Maks Ovsjanikov. Diffusionnet: Discretization agnostic learning on surfaces. *ACM Transactions on Graphics (TOG)*, 41(3):1–16, 2022.

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.

A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Analytics Vidhya. Mlp: Multi-layer perceptron - simple overview, 2020. Retrieved from.

Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

Xumin Yu, Lulu Tang, Yongming Rao, Tiejun Huang, Jie Zhou, and Jiwen Lu. Point-bert: Pre-training 3d point cloud transformers with masked point modeling. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 19313–19322, 2022.