POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Deep Learning-based Methods for Polygonal Grids Agglomeration

Numerical Analysis of Partial Differential Equations
Mathematical Engineering

Nicola Farenga, Gabriele Martinelli, Luca Saverio

Professors:
Prof. Alfio Quarteroni
Dr. Francesco Regazzoni

Supervisors:
Prof. Paola F. Antonietti
Dr. Enrico Manuzzi

Academic year:
2021-2022

Abstract: We present a Geometrical Deep Learning-based approach to tackle the problem of polygonal grids agglomeration via a graph partitioning algorithm, which makes use of a bisection model built on Graph Neural Networks (GNNs). The proposed algorithm exploits the geometrical information of the underlying graph representation of the input polygonal grid and is based on an unsupervised training procedure, that minimize a loss function defined as the expected value of the normalized cut. We explore the performance of two models architectures based on the GraphSAGE framework, by comparing them with respect to the state of the art methods, such as k-means and METIS. Our results highlight the capabilities of GNN-based methods to generate agglomerated grids of comparable quality with respect to k-means and METIS based methods, and their computational efficiency in terms of runtime. Moreover, we employ our method alongside Multigrid solvers to asses its performance, and further test its agglomeration capabilities to complex domains, showing a high degree of generalization both in terms of shapes and dimensions.

Key-words: Mesh Agglomeration, Graph Neural Networks, Multigrid Methods

1. Introduction

The numerical solution of differential problems is, nowadays, central to all fields of engineering, and it is expanding its boundaries to other fields which can benefit from numerical simulations. Despite the many innovations of recent years, as long as Finite Element-based Methods (FEM) are applied to discretize differential problems, the complexity of the physical domain is always going to represent a bottleneck of the simulation. For this reason, due to the high complexity of the physical domain arising from many engineering applications, novel approaches have been developed. An example is the introduction of multilevel methods, such as Multigrid methods (MG), which rely on a hierarchy of discretization characterized by different refinement levels. The adoption of this methods not only raise the issue of mesh generation, but also the one of mesh agglomeration or coarsening. Currently employed agglomeration methods rely on iterative algorithms, making them not particularly feasible for highly complex domain. Moreover, during the agglomeration procedure, it is fundamental to preserve the quality of the underlying mesh, because it might affect the overall performance of the method in terms of stability and accuracy. Indeed, by using a suitably agglomerated mesh, it is possible to achieve the same accuracy but with a much smaller number of degrees of freedom, when solving the numerical problem, leading to memory savings and lower computational loads.
In this project we present a Geometric Deep Learning-based approach to the problem of mesh agglomeration,

which relies on the recently rediscovered concept of Graph Neural Networks. We adopt a similar approach to the one presented by Gatti et al. [2021a], based on *Generalized Graph Partitioning*, but we exploit the underlying graph structure of the input grid and the geometrical features of the elements, instead of directly feeding the grid to the network. Being able to extract elements' geometrical features, allow us to avoid the need for spectral based modules for generating features embeddings.
The outline of the report is as follows. In Section 2 we introduce the problem of graph partitioning, since it is central to our proposed solution and to state-of-the-art agglomeration methods, whose overview is presented in Section 3.
The concept of Graph Neural Networks, Graph Convolutions and the GraphSAGE architecture is introduced in Section 4, since our models deeply rely on the latter one. The models' architecture, their training procedure and agglomeration algorithm are covered in Section 5.
In Section 6 we cover the implementation of training and inference pipelines, and in Section 7 we comment the quality of the obtained agglomerations, the models' shape-generalization capabilities and their runtime performances. Finally, in Section 8, we present two applications, one headed to test the employability of GNN-based agglomeration methods together with Multigrid solvers, and a second one to further test the extrapolation capabilities of our models on a complex domain.
The project's code repository is available at `github.com/farenga/meshGNN`.

2. Overview of Graph Partitioning

Most refined grid agglomeration algorithms, tackle this problem by reframing it into a *graph partitioning* problem. Graph partitioning is a NP-hard problem that consists of finding M disjoint sets of nodes $S_1, ..., S_M$ of the graph $G = (V, E)$, where $V = \{v_i\}_{i=1}^N$ and $E = \{e(v_i, v_j) : v_i, v_j \in V\}$ are the sets of nodes and the set of edges, respectively, such that $\cup_{i=1}^M S_i = V$ and $\cap_{i=1}^M S_i = \emptyset$.

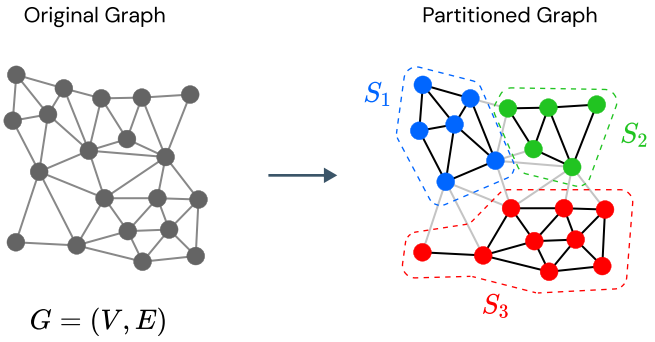


Figure 1: General framework employed for graph agglomeration via GNN-based model.

We assume of dealing always with undirected graphs, meaning that $e(v_i, v_j) \in E \iff e(v_j, v_i) \in E$. We will use the following graphs-related notation:

- $|V| = N$, nodes' set cardinality;
- $A \in \mathbb{R}^{N \times N}$ is the adjacency matrix;
- $X \in \mathbb{R}^{N \times F}$ is the nodes' features matrix, where F is the number of nodes' features;
- $\mathcal{N}(v_i) = \{v_j \in V : e(v_i, v_j) \in E\}$ denotes the neighborhood of the i-th node, containing the nodes v_j directly adjacent to v_i.

Moreover, we can define the so called *cut*, representing the number of edges connecting the disjoint sets of nodes resulting from the paritioned graph. In the case of two partitions, it can be defined as:

$$\text{cut}(S_1, S_2) = |\{e(v_i, v_j) \in E : v_i \in S_1, v_j \in S_2\}|, \tag{1}$$

and can be easily generalized to the case of M partitions, as

$$\text{cut}(S_1, ..., S_M) = \frac{1}{2} \sum_{k=1}^{M} \text{cut}(S_k, S_k^C) \qquad \text{with} \quad S_k^C = \bigcup_{\substack{j=1 \\ j \neq k}}^{M} S_j \setminus S_k. \tag{2}$$

Tipically, one of the main objectives of graph partitioning is to minimize the cut. Indeed we will adopt a framework based on the notion of *cut* to perform the train of our models. Graph partitioning is a recurrent problem in multiple fields, from social sciences to engineering. More specifically, in the field of scientific computing, it is a technique employed for parallelizing problems, distributing data over multiple processors, and computing the

solutions of discretized PDEs via multigrid or domain-decomposition methods.
The problem of mesh agglomeration is related to the one of graph partitioning since a mesh can be easily reframed as a graph, as explained in Algorithm 4. The graph representation of the mesh consists of placing nodes at the center of each of the mesh's elements, and by connecting them via an edge, if the relative elements in the mesh are adjacent. Then, a vector of features can be assigned to each node, storing geometrical information, such as nodes coordinates and elements' areas, as in our case. In this way a partitioning algorithm can be applied to the extracted graph, then by returning to the mesh representation, we obtain an agglomerated version of the original mesh.

3. State of the Art Methods for Mesh Agglomeration

The most common and widely used mesh agglomeration methods are based on clustering algorithms such as, k-means or METIS. Their popularity is due to multiple factors, but most of all due to their parallelizable nature, which makes them particularly well suited for in scientific computing applications. In the following we present an overview of those methods, since they will be later adopted to perform comparisons with the deep learning-based methods introduced by this project.

3.1. k-means

The k-means algorithm, introduced by Macqueen [1967], is a popular iterative procedure for partitioning an N-dimensional set into k sets. The key idea is to define k centroids in the space $\mathbb{R}^n$, and to iteratively update points' clusters labels in an unsupervised manner, by minimizing eucliden distances between the points x_i an centroids μ_j. Algorithm 1 shows the k-means algorithm.

Algorithm 1 k-means algorithm

Input Set of points $\{x_i\}_{i=1}^N \subset \mathbb{R}^n$, convergence criteria
Output Set of clusters labels $\{c_i\}_{i=1}^N$

Random initialization of centroids $\{\mu_1, ..., \mu_k\} \subset \mathbb{R}^n$
while !converged **do**
  $c_i \leftarrow \arg\min_j \|x_i - \mu_j\|^2, \ \forall i = 1, ..., N$
  $\mu^j \leftarrow \frac{\sum_{i=1}^M \chi_{\{c_i=j\}} x_i}{\sum_{i=1}^M \chi_{\{c_i=j\}}}, \ \forall j = 1, ..., K$
end while

In the k-means algorithm the stopping criteria can be based on the non-improving position of the centroids, no changes in the set of clusters, or maximum number of iterations reached. This clustering algorithm can be extended to the mesh agglomeration problem by taking into account the graph representation of the mesh itself. In our case, the k-means algorithm will be employed both in the agglomeration procedure and in the benchmarking-phase. The implementations employed are the built-in MATLAB function 'kmeans', and `sklearn.cluster.kmeans` from scikit-learn package (Pedregosa et al. [2011]) in the Python case.

3.2. METIS

METIS, developed by Karypis and Kumar [1999], is a serial software package for partitioning large irregular graphs, partitioning large meshes, and computing fill-reducing orderings of sparse matrices. As depicted in the scheme in Figure 2, METIS' graph partitioning algorithm is based on a multi-level graph bisection procedure, consisting of three main steps:

1. *Graph coarsening phase*: a series of successively smaller graphs is derived from the input graph. Each successive graph is constructed from the previous graph by collapsing together a maximal size set of adjacent pairs of vertices. This process continues until the size of the graph has been reduced to just a few hundred vertices.
2. *Initial partitioning phase*: a partition of the coarsest, and hence smallest, graph is computed.
3. *Uncoarsening phase*: the partitioning of the smallest graph is projected to the successively larger graphs by assigning the pairs of vertices that were collapsed together to the same partition as that of their corresponding collapsed vertex.

Most scientific computing-related applications actually rely on ParMETIS, which is an MPI-based parallel-version of the METIS library. The parallel library is well suited for large scale numerical applications, and it is based on parallel multi-level k-way graph partitioning algorithms.
In this project METIS will be employed for comparison pourposes, by relying on the MATLAB `METISMex` package, and on `NetworkX-METIS`, a Cython-based METIS add-on package for the `NetworkX` Python library.

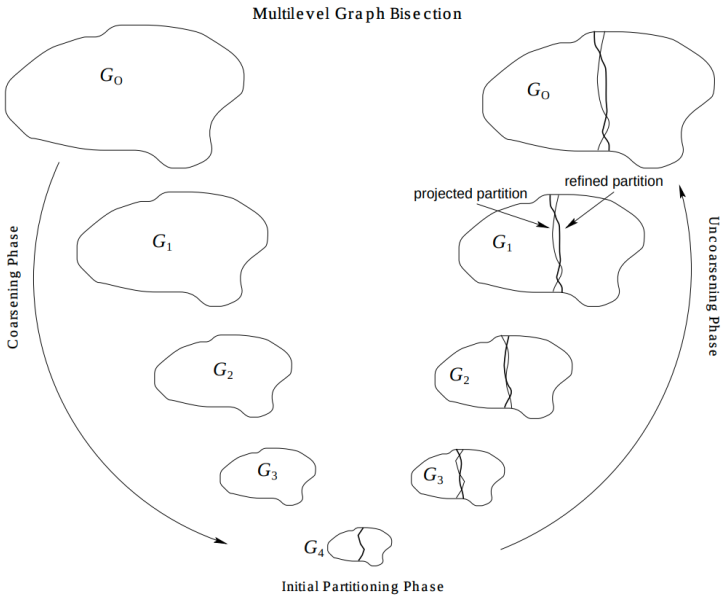


Figure 2: METIS graph bisection algorithm scheme, from Karypis and Kumar [1999].

4. Overview of Graph Neural Networks

Graph Neural Networks (GNNs) are deep learning architectures specifically meant to work with graph-structured data and developed in the broader field of Geometrical Deep Learning, which concerns the application of neural networks to non-Euclidean data structures.
The first GNNs examples are strictly rooted to the introduction of Recursive Neural Networks, due to their directed acyclic graph architecture, however, the recent rediscovery of GNNs was mainly caused by the central role played by Convolutional Neural Networks (CNNs) architectures, indeed GNNs can be seen as a generalization of CNNs to graph structured data. CNNs were introduced in the late 90s (LeCun et al., 1998) and employed to images-related tasks, and later extended to data of multiple nature, such as text data. The fact that CNNs are limited to work on Euclidean domains (two-dimensional grids or one-dimensional sequences, for images and text respectively) paved the way towards the extension of those architectures to more general data-structures.
The concepts underlying GNNs architectures are manifold, for this reason we will emphasize the ones related to our models' implementation, with a focus on the task related to this project: node classification.

4.1. GNNs Framework

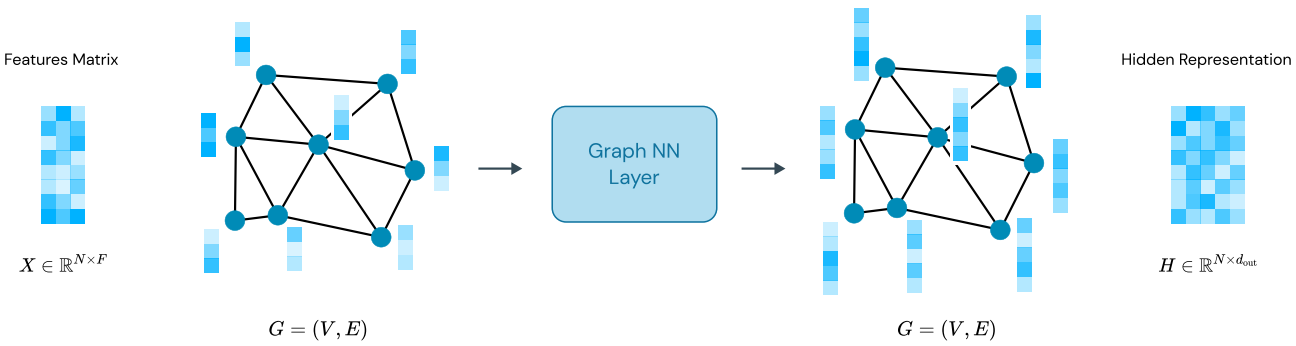


Figure 3: General GNN framework, consisting of an input graph G together with its features matrix X, and an output hidden representation H related to the same graph structure (since no pooling operations are involveld).

The general framework of Graph Neural Networks consists of a graph-in/graph-out architecture, meaning that this architectures accept a graph $G = (V, E)$ as input, consisting of an adjacency matrix A together with the information loaded into nodes (e.g. nodes' features matrix X), edges and global-context, and returns a graph with the same connectivity structure of the input together with a progressively transformed embedding of the carried information.
From this perspective a GNN can be defined as an optimizable transformation applied to one or multiple attributes of the graph (nodes, edges, global-context) preserving graph structure, so it is a permutation invariant transformation.
Since GNNs' information can be locally stored into nodes or edges, or globally related to the whole graph, different types of GNN-related tasks arise:

- *node-level task*, concerning nodes' properties prediction.
- *edge-level task*, concerning the prediction of their properties or their presence.
- *graph-level task*, concerning the prediction of global properties of the graph.

Our focus will be on the node-level tasks, since it is the class node classification belongs to. As introduced, GNNs iteratively update the node representations by combining information comining from the neighborhood of the node and the one stored in the node itself. We will denote X as the nodes features matrix (initial representations) related of the input graph G, and H^k, the hidden representation of those features after the k-th graph convolutional layer. Each layer is described by two functions:

- *aggregation function* Φ, defines how the information coming from the neighborhood $N(v_i)$ of the node v_i is aggregated, producing an aggregated state a_i^k cf. the definition (3) below.
- *combination function* Ψ, defines how the aggregated information a_i^k coming from the neighborhood $N(v_i)$ is combined with the one stored in the current node v_i cf. the definition (4) below.

So, formally, at step k, the hidden representation H_i^k for the node v_i is computed as:

$$a_i^k = \Phi^k(\{H_j^{k-1} : v_j \in \mathcal{N}(v_i)\}), \tag{3}$$

$$H_i^k = \Psi^k(H_i^{k-1}, a_i^k), \tag{4}$$

with X, the nodes' features matrix, taken as initial hidden state $H^0 = X$.
Different definitions of aggregation and combination functions leads to different GNNs architectures, such as Graph Convolutional Networks and GraphSAGE. We will cover both of them, with a focus on the latter, since it is the one employed in our model implementation.

4.2. Graph Convolutional Networks

Graph Convolutional Networks' (GCNs) architecture is characterized by the following propagation rule:

$$H^{k+1} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^k W^k), \tag{5}$$

which defines how the node representations are updated in each layer. In 5:

- $\tilde{A} = A + I_N$, is the adjacency matrix with added self-connections, with A the adjacency matrix of G and I_N the identity matrix of order N. This allows to incorporate the information stored in the node itself when updating its representation.
- $\tilde{D}$ is the diagonal matrix of degrees of $\tilde{A}$, where $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.
- $\sigma(\cdot)$ is the activation function, tipically ReLU or Tanh.
- $W^k \in \mathbb{R}^{F_k \times F_{k+1}}$ is the weights matrix associated to the k-th layer, representing a trainable linear transformation, where F_k and F_{k+1} are the features dimensions for the current and next layers, respectively.

The update equation (5) for the node v_i can be reformulated to highlight the aggregation and combination terms as follows

$$H_i^k = \sigma\left(\sum_{j \in \mathcal{N}(v_i)} \frac{\tilde{A}_{ij}}{\sqrt{\tilde{D}_{ii}\tilde{D}_{jj}}} H_j^{k-1} W^k + \frac{1}{\tilde{D}_{ii}} H_i^{k-1} W^k\right). \tag{6}$$

Therefore, via this reformulation, it can be easily seen that the role played by a node $v_j \in \mathcal{N}(v_i)$ in the update of the v_i's hidden representation, is measured by the weight of their edge $\tilde{A}_{ij}$ normalized by their node degrees.

4.3. GraphSAGE

The GraphSAGE framework, introduced by Hamilton et al. [2017], is based on the idea of letting the aggregation function to be learnable. The embedding generation allows to directly leverage nodes features, in order to learn an embedding function that generalizes to unseen nodes. Indeed this approach allows to have a focus both on

the topological structure of the graph, and, on the features distribution of the nodes' neighborhood. This is achieved by a SAmpling-and-aggreGatE behaviour of the embedding algorithm, reported in Algorithm 2

Algorithm 2 GraphSAGE forward propagation algorithm

Input Graph $G = (V, E)$, input features $X \in \mathbb{R}^{N \times F}$, number of layers K, activation function σ, weight matrices W^k, aggregation functions Φ^k, $\forall k = 1, ..., K$, concatenation function Ψ

Output Features matrix representation Z

1: $H^0 = X$
2: **for** $k = 1, ..., K$ **do**
3: **for** $v_i \in V$ **do**
4: $a_i^k \leftarrow \Phi^k(\{H_j^{k-1} : v_j \in \mathring{\mathcal{N}}(v_i)\})$
5: $H_i^k \leftarrow \sigma(W^k \cdot \Psi(H_i^{k-1}, a_i^k))$
6: **end for**
7: $H_i^k \leftarrow H_i^k / \|H_i^k\|_2$
8: **end for**
9: $Z = H^K$

In Algorithm 2 we have denoted with H_i^k the vector of hidden representations, extracted from the matrix H^k, related to the node v_i. The nature of the aggregation function Φ is central to the SAGE architecture, for which there are multiple candidates depending on the task to be performed.
Our focus will be on the *mean aggregator*, defined as the elementwise mean of the vectors in $\{H_i^{k-1}, \forall v_i \in \mathcal{N}(v_i)\}$, so the update of the lines (4,5) in Algorithm 2, can be rewritten as

$$H_i^k \leftarrow \sigma(W^k \cdot \text{mean}(\{H_i^{k-1}\} \cup \{H_j^{k-1}, \forall v_j \in \mathcal{N}(v_i)\})). \tag{7}$$

The key difference of the SAGE framework from standard GNNs architectures is in the sampling behaviour of the aggregation function. Indeed, instead of using the full set of neighbors $\mathcal{N}(v_i)$, a fixed-size subset $\mathring{\mathcal{N}}(v_i)$ is uniformly sampled at each iteration. This allows to achieve better generalization capabilities in terms of hidden representations generation, while the fixed-sized sample is adopted to keep the computational cost under control at each iteration.

5. GNNs-based Agglomeration

The general adopted framework for performing mesh agglomeration consists of a graph bisection GNN-based model which is recursively applied to graph extracted from the mesh. We would like to highlight the fact that the input grid can obviously be already seen as a graph, but it would not have any features attached to its nodes, since they are just the vertices of the mesh. There are some examples of GNNs-based models (Gatti et al. [2021a,b]) directly applied to the original grid, which employ additional GNN-based spectral modules to perform a preliminary embedding step, in order to extract features that can later be fed to a GNN-based partitioning module. In our case, the graph extraction step provides us with geometrical features that can be leveraged to perform a classification task and avoiding the need for an additional spectral embedding module.

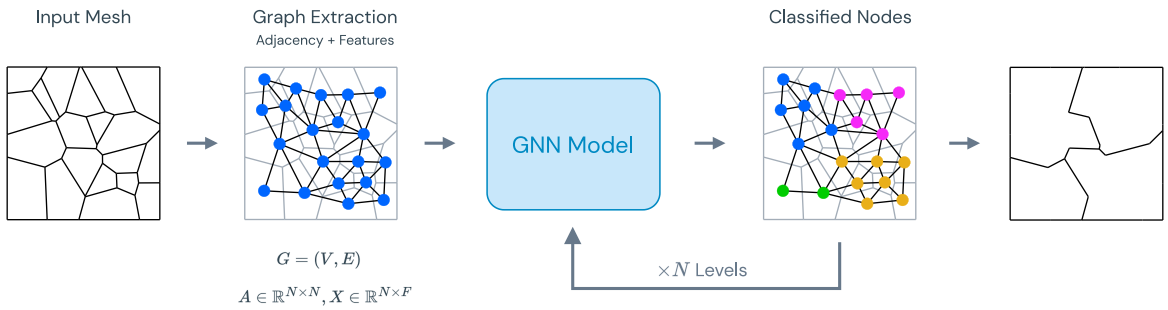


Figure 4: General framework employed for graph agglomeration via GNN-based model.

The graph-bisection model performs a classification task, by taking as input the graph $G = (V, E)$ and the features related to its nodes $X \in \mathbb{R}^{N \times F}$, and outputting a probablity tensor $Y \in \mathbb{R}^{N \times 2}$. Where Y_{ij} represents the probability that the node $v_i \in V$ belongs to the partition S_j, with $j = 1, 2$. This approach can be obviously

generalized to an arbitrary number of partitions, but it would require a specific model for each fixed number of classes, while the 2-classes case can be easily extended to a multi-class case (>2) by recursively calling the bisection model.
We will explore two models architectures, a base one, and more advanced one featuring skip connections and variable number of hidden representations per layer, both based on the GraphSAGE framework.

5.1. Graph Bisection Models

Both the models employed for graph bisection consist of a modular structure, featuring a *normalization module* and a *partitioning module*.
The *normalization module* acts on the features matrix X, in our case a $N \times 3$ matrix, since we consider as initial nodes' embedding geometrical fetures, such as 2D elements' center coordinates and elements' area. The normalization is performed both on nodes' coordinates and nodes' areas, the former are translated to the origin and rescaled in the domain $(-1, 1)^2$, while the latter are scaled within the interval $(0, 1)$. We set

$$X = \left[\mathbf{x} \mid \mathbf{y} \mid \mathbf{a} \right], \tag{8}$$

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \text{mean}(\mathbf{x})}{\max(\mathbf{x})}, \quad \tilde{\mathbf{y}} = \frac{\mathbf{y} - \text{mean}(\mathbf{y})}{\max(\mathbf{y})}, \quad \tilde{\mathbf{a}} = \frac{\mathbf{a}}{\max(\mathbf{a})}, \tag{9}$$

where $\mathbf{a}$ is the vector containing the elements' areas. Finally a rotation is applied to the coordinates in order to make the normalized graphs always have the same orientation. After the normalization step, the *partitioning module* ingests both the graph $G = (V, E)$ and the normalized features $\tilde{X}$. We will distinguish between two partitioning modules, that allow us to define two different models: SAGE-Base, SAGE-Res. Both modules are implemented by relying on Pytorch Geometric (Fey and Lenssen [2019]), a Pytorch-based geometric deep learning library.

5.1.1. SAGE-Base

The base model consists of a block of GNN-layers for the first step of features embedding, then the processed features are fed into a dense neural network acting as a classifier. More specifically the embedding step involves 4 SAGE convolutional layers (SAGEConv), with a dimensionality of the processed features which is kept constant throughout the model ($d_{out} = 64$). The SAGEConv layers are followed by tanh nonlinear activation function. So the transformation of the l-th layer can be written as:

$$H_i^{l+1} = \tanh(H_i^l W_1^l + (\text{mean}_{j \in \mathcal{N}(v_i)} H_j^l) W_2^l) \qquad \forall i = 1, ..., N \quad \forall l = 0, 1, 2, 3, \tag{10}$$

where H_i^l is the features vector related to the node v_i at the k-th step, while W_1^l and W_2^l are l-th layer's weights matrices ($\in \mathbb{R}^{d_{in} \times d_{out}}$), whose entries are updated via backpropagation during the training process.

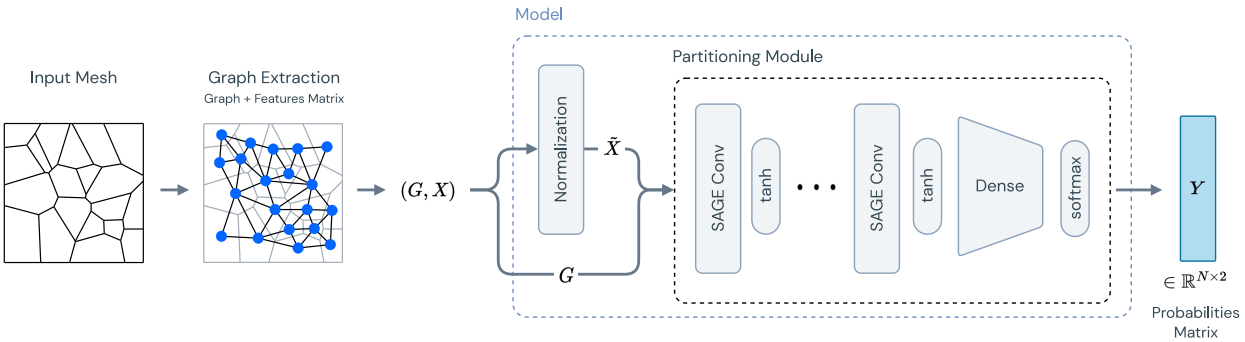


Figure 5: SAGE-Base model's structure, consisting of a normalization module, responsible of rescaling the features extracted from the input mesh, and of a partitioning module made of a stack of graph convolutions and a dense classifier. The ouput is a matrix of probabilities $Y \in \mathbb{R}^{N \times 2}$, containing the probablities of belonging either to one of the two classes for each node.

The choice of the activation function as tanh is supported by the fact of wanting to keep the features in the $[-1, 1]$ interval, so that the geometrical information, at least the one regarding the rescaled coordinates, will be kept in the same domain as information flows through the layers.

For the classification stage, three densely connected linear layers have been employed with a progressively decreasing dimensions of the hidden state, alternated by tanh activation functions, besides the last one, where a softmax activation function has been employed to get classes probabilities $Y_{ij} = p(v_i \in S_j|(G, X)) \in [0, 1]$. The softmax non-linearity, applied to a vector $\mathbf{z} \in \mathbb{R}^n$, is defined as

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{N}}, \qquad i = 1, ..., n.$$

The resulting model consists of 28K parameters, where 25K comes from the GNN-block, while the remaining 3K are within the dense classifier at the end of the network.

5.1.2. SAGE-Res

The SAGE-Res model is a model which features a structure similar to the one of the U-Net (Ronneberger et al. [2015]), with the usage of *residual connections* and with a progressive increase in the depth of the hidden representations in the first block of convolutions, and with a progressive contraction of the dimensions in the second block. The main difference with respect to a U-Net is the lack of pooling and upooling layers. This choice is due to the fact that the pooling and unpooling tasks involving graphs are more expensive from a computational point of view than those implemented in a standard CNNs framework.

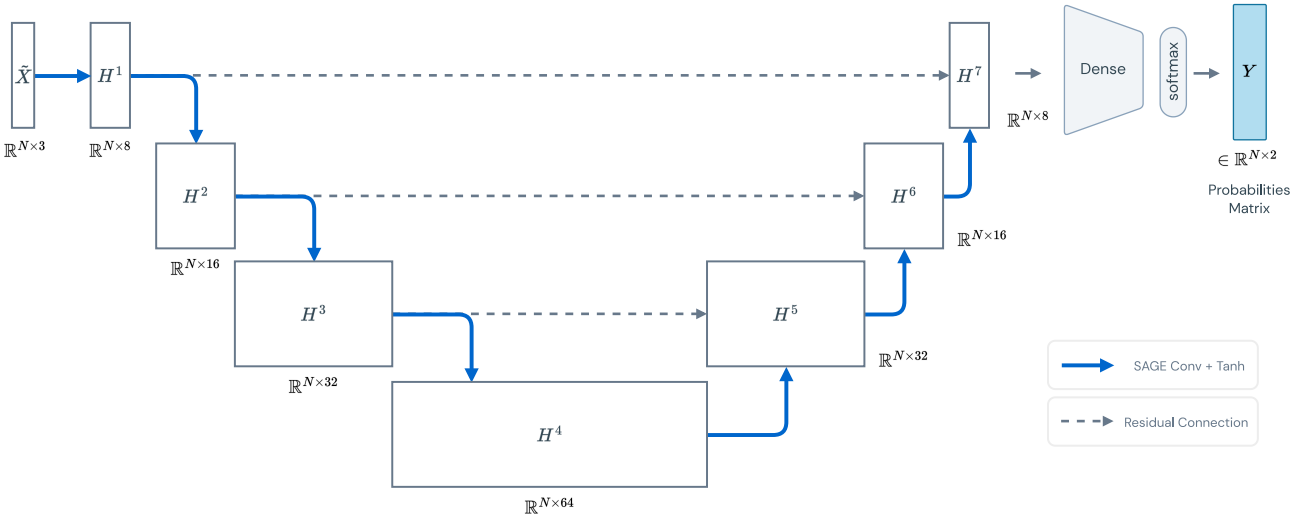


Figure 6: SAGE-Res partitioning module's structure, employing 7 layers of SAGEConv (blue arrows), and residual connections (dashed arrows). The classification stage relies on a dense classifier with a similar structure of the one employed in the base model.

In a general framework, a residual connection consists of summing the input of a learnable residual block $f_\theta(\cdot)$ to the output of the block itself, as defined by He et al. [2015]

$$h^{l+1} = f_\theta(h^l) + h^l. \tag{11}$$

In our U-Net-inspired architecture, the residual connections are not defined between two consecutive layers $l, l+1$, but they are established between the layers that share the same hidden state dimensions, as can be observed in the architecture scheme depicted in Figure 6. Therefore, given that l_{prev} and l_{next} match their input/output dimensions, the transformation can be written as

$$H^{l_{\text{next}}} = \tanh(\text{SAGEConv}(G, H^{l_{\text{prev}}})) + H^{l_{\text{prev}}}. \tag{12}$$

A key aspect that differentiate the SAGE-Res partitioning module from the base one is the fact that the dimensionality of the hidden features in non-constant. Indeed the normalized input features $\tilde{X} \in \mathbb{R}^{N\times 3}$ goes through the first stack of SAGEConvs, where the dimensionality of hidden features is progressively increased and setted as [8,16,32,64], then the extended hidden representation goes through a second stack of SAGEConvs which successively compresses the dimensions to [32,16,8]. Besides having more layers than the base model's architecture, the variable dimensionality of SAGE-Res' hidden representations allows to have a deeper model with a lower number of parameters (11K).

5.2. Training

The training of the models has been performed by employing an unsupervised approach. The framework adopted is based on a novel formulation of the loss function, introduced by Gatti et al. [2021a], that allows to take into account the expected value of the normalized cut. Following the graph partitioning introduction given in Section 2, we consider the generalized notion of *normalized cut*

$$\text{Ncut}(S_1, ..., S_M) = \sum_{k=1}^{M} \frac{\text{cut}(S_k, S_k^C)}{\text{vol}(S_k, V)}, \tag{13}$$

where the *volume* of the partition S_k is defined as

$$\text{vol}(S_k, V) = |\{e(v_i, v_j) \in E : v_i \in S_k, v_j \in V\}|, \tag{14}$$

and represents the total degree of all nodes of the k-th partition. The *normalized cut* differs from the standard cut, since it allow us to take into account balanced partitions. Now we would like to reframe the quantity introduced in order to perform the training of the model, by encoding the normalized cut into the loss function. To do so, we recall that the model provides us with Y_{ij}, the probability that the node v_i belongs to the class $j = 1, 2$. For this reason we can define the *expected cut*, given two partitions, S_k and the complementary one S_k^C

$$\mathbb{E}[\text{cut}(S_k, S_k^C)] = \sum_{i=1}^{N} \sum_{v_j \in \mathcal{N}(v_i)} Y_{ik}(1 - Y_{jk}) = \sum_{i=1}^{N} \sum_{j=1}^{N} Y_{ik}(1 - Y_{kj^T})A_{ij}. \tag{15}$$

Let D be the column vector of degrees, such that $D_i = \text{degree}(v_i)$. Then

$$\mathbb{E}[\text{vol}(S_k, V)] = (Y^T D)_k = \Gamma_k. \tag{16}$$

Therefore, the *expected normalized cut* can be defined as

$$\mathbb{E}[\text{Ncut}(S_1, ..., S_M)] = \sum (Y \oslash \Gamma)(1 - Y)^T \odot A, \tag{17}$$

where $\oslash$ and $\odot$ denote the element-wise division and multiplication, respectively, and the summation, is a sum over all the elements of the matrix resulting from $(Y \oslash \Gamma)(1 - Y)^T \odot A$. So we consider the following *loss* function

$$\mathcal{L}(W; Y, G) = \sum_{k=1}^{2} \sum_{i,j=1}^{N} \frac{Y_{ik}(1 - Y_{kj}^T)A_{ij}}{\Gamma_k}, \tag{18}$$

that is going to be minimized during the training process, where W represents the weights of the model. The loss function has been implemented in the following algorithm, by relying on the Pytorch Geometric graph structure:

```
from torch_geometric.utils import degree

def loss_normalized_cut (y , graph):
  d = degree(graph.edge_index[0], num_nodes=y.size(0))
  gamma = torch.t(y) @ d
  c = torch.sum(y[graph.edge_index[0], 0]*y[graph.edge_index[1], 1])
  return torch.sum(torch.div(c,gamma)).to(device)
```

Listing 1: Pytorch implementation of the normalized-cut loss function.

The complete training routine is available in the Appendix A.
To perform the training a train and a validation datasets were generated (both available in the project repository). The training dataset consists of 800 meshes, with 200 meshes per type (squares, triangles, random triangles, random Voronoi), in order to have a balanced dataset. The validation dataset consists of 200 meshes, 50 per type. The dimensionality of the datasets has been choosen to keep a 80/20 split between train and test respectively. Due to the low dimensionality of the datasets, two main strategies have been adopted to prevent overfitting behaviours:

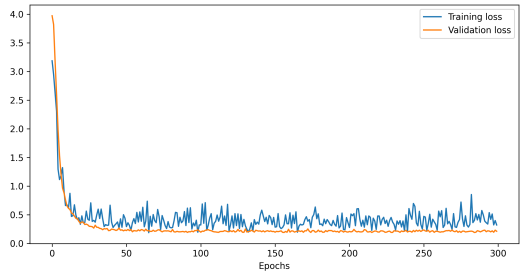
- *Regularization*: an L2-regularization term as been summed to the normalized cut loss, in order to prevent overfitting behaviours, leading to the following loss definition

$$\hat{\mathcal{L}}(W; Y, G) = \mathcal{L}(W; Y, G) + \lambda \|W\|_2^2, \tag{19}$$

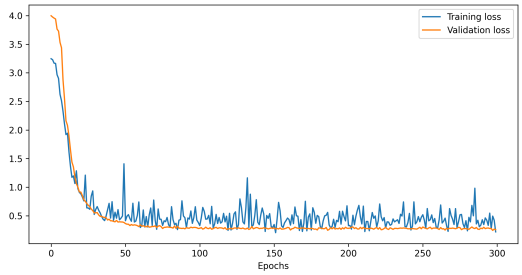
  this has been achieved by adding a weight-decay parameter in the definition of the torch optimizer.

- *Data augmentation*: each sample in the training set has been rotated by a random angle at each training iteration, to prevent overfitting and to make the model more robust with respect to different configurations of the input mesh.

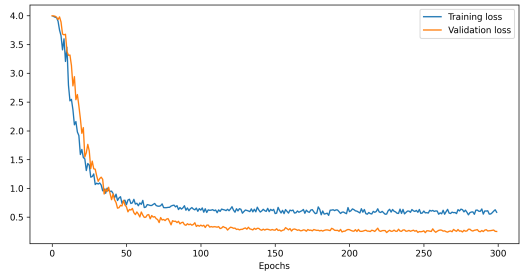
The training has been performed by adopting the Adam optimizer with a *learning rate* γ = 1e-5 and a *weight decay* parameter λ = 1e-5, with 300 epochs and batch size 4.
The random rotation and the order shuffling of the training dataset introduced some oscillations in the loss, expecially in the training one, for this reason a *learning rate reduction schedule* has been employed to reduce the phenomenon, by halving the learning rate after 20 iterations of no improvements in the training loss.



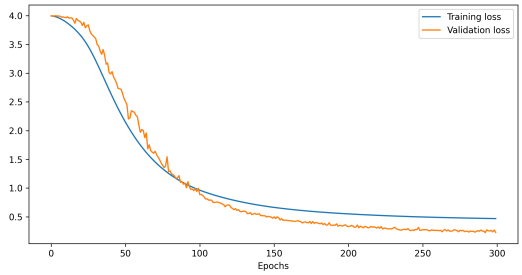
(a) SAGE-Base



(b) SAGE-Res



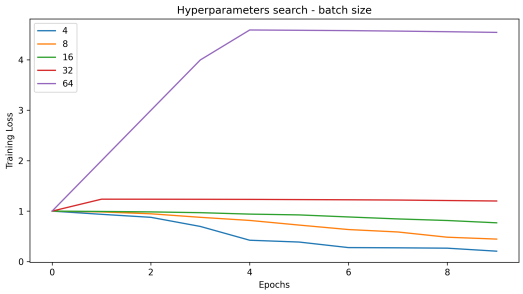
(c) SAGE-Base Voronoi



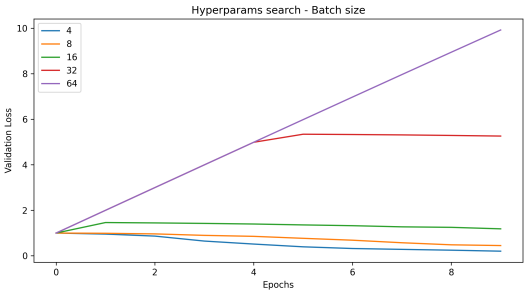
(d) SAGE-Res Voronoi

Figure 7: Training and validation losses for SAGE-Base and SAGE-Res models.

Dealing with the deeper SAGE-Res architecture obviously comes at a cost in terms of training, indeed the convergence of the loss function is a bit slower in the SAGE-Res case. This is reflected in terms of training time, with $\sim$ 35 min for SAGE-Base case, while $\sim$ 40 min are required for the training of SAGE-Res.
In order to understand the models' generalization capabilities the models have been trained on a dataset containing only Voronoi meshes, an tested on all the other shapes. As can be seen from the losses reported in Figure 7(c) and in Figure 7(d), training the models only on a specific mesh type makes the training procedure more stable, with the loss having reasonably lower magnitude of oscillations and a much smoother behaviour. Models' generalization capabilites in terms of shape extrapolation are further investigated in Section 7.3.



(a) Training loss



(b) Validation loss

Figure 8: Training and validation losses trends of the first 10 epochs for different values of batch size.

The low value adopted for the batch size is derived from some training test-runs, which where done to assess the stability of the training process. As can be seen in Figure 8, as the batch size increases, [4, 8, 16, 32, 64], the behaviour of the loss worsens, up to the value 64, where we have instabilities since the very first steps.

5.3. Agglomeration Algorithm

As already mentioned, the developed models tackle the problem of graph bisection. Therefore, in order to perform a general partitioning of the graph into multiple clusters, the bisection model has to be included into a partitioning procedure. Since our focus is on mesh agglomeration, rather than graph partitioning, we will consider as a target measure, to end the partitioning routine, the target mesh element size h^*, instead of considering the number of clusters, as we would do in a clustering-based framework.
In Algorithm 3 we summarize the proposed algorithm.

Algorithm 3 Agglomeration algorithm

Input Input mesh $\mathcal{T}_h$, target mesh size h^*, bisection model $\mathcal{M}$
Output Agglomerated mesh $\mathcal{T}_{h^*}$

Function AGGLOMERATE ($\mathcal{T}_h$, h^*)

elementList, elementList1, elementList2 = [ ]
for $K \in \mathcal{T}_h$ **do**
  if $h_K \leq h^*$ **then**
    elementList.insert(K)
  end if
end for
if elementList== $\emptyset$ **then**
  return $\mathcal{T}_h$
end if
$G, X \leftarrow$ extractGraph($\mathcal{T}_h$)
$Y \leftarrow \mathcal{M}(G, X)$
$\mathcal{T}_h^{(1)}, \mathcal{T}_h^{(2)} \leftarrow$ getPartitions($\mathcal{T}_h, Y$)
elementList1 $\leftarrow$ AGGLOMERATE($\mathcal{T}_h^{(1)}, h^*$)
elementList2 $\leftarrow$ AGGLOMERATE($\mathcal{T}_h^{(2)}, h^*$)

6. Pipeline Implementation

Due to the lack of a MATLAB geometric deep learning package we opted to proceed with the development of the models by relying on Python, more specifically by adopting Pytorch Geometric (Fey and Lenssen [2019]), a Pytorch-based library. Concerning the whole project pipeline, this choice led to the development of a hybrid MATLAB-Python solution. We can distinguish between two main pipelines:

- *Training pipeline*, concerning dataset generation, models development and training.
- *Inference pipeline*, concerning the usage of the model in evaluation-mode.

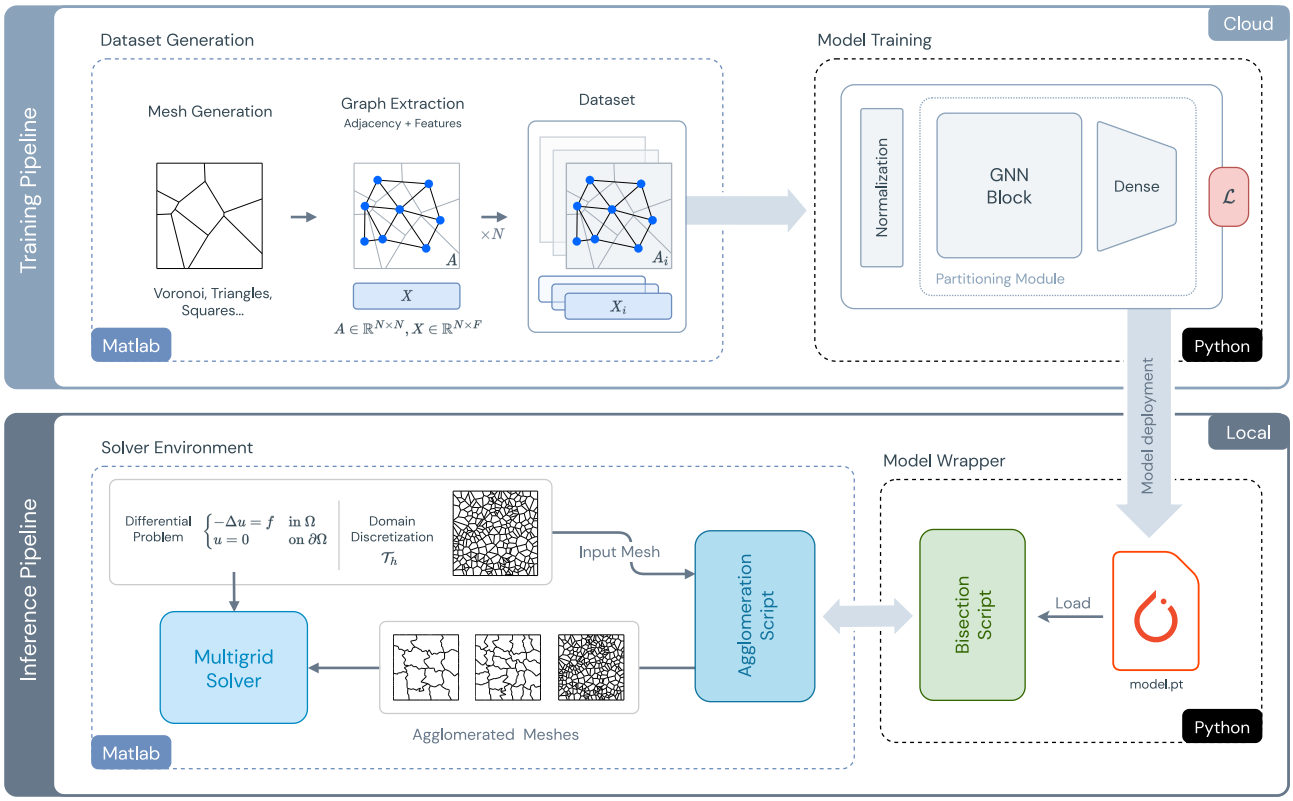


Figure 9: General framework employed for graph agglomeration via GNN-based model.

6.1. Training Pipeline

The training pipeline relies on both MATLAB and Python and completely runs on cloud. MATLAB is employed for dataset generation, which essentially consists of 3 main steps, *mesh generation*, *graph extraction*, *bundling* graph together to create the dataset. For the meshes generation and graphs extraction steps we relied on the MATLAB package provided to us as a starting point. In particular, for the graph extraction step we adopted the procedure highlighted in Algorithm 4. The procedure consists of placing graph's nodes at the center of each one of the mesh's elements, and connect them via an edge if the relative elements in the original mesh are adjacent. Finally, it is possible to assign to each node a vector of features storing geometrical information, such as its center's coordinates and its area.
After the graph extraction phase, the graphs, meaning adjacency matrices alongside their features matrices, are bundled to create a dataset. The dataset is then loaded into a Python environment, more specifically we relied on Google's Colab cloud platform. Here the development of the models took place, by relying on Pytorch Geometric, together with their training and a first graph bisection test. The machines used for the training featured a 2.20GHz Intel Xeon processor, with 12GB of RAM memory and NVIDIA Tesla T4 GPU with 16GB of GDDR6 memory.

Algorithm 4 Graph extraction

Input 2-D input mesh $\mathcal{T}_h = \{K_i\}_{i=1}^{N_k}$, elements faces $\mathcal{F}_h = \{F_i\}_{i=1}^{N_f}$
Output Extracted graph $G = (V, E)$

Initialize graph G: $V, E \leftarrow [\,]$
for $K_i \in \mathcal{T}_h$ **do**
  $v_i \leftarrow K_i$
  $V.\text{insert}(v_i)$
end for
$C \leftarrow \text{zeros}(\text{length}(\mathcal{F}_h),2)$
$k \leftarrow 1$
for $F_j \in \mathcal{F}_h$ **do**
  $L \leftarrow \{i : F_j \in \mathcal{F}_h \cap \partial K_i\}$
  if $\text{length}(L) == 2$ **then**
    $C(k,:) \leftarrow L$
    $k \leftarrow k + 1$
  end if
end for
for $l = 1, ..., k-1$ **do**
  $e(v_i, v_j) \leftarrow$ edge $C(l,:)$
  $E.\text{insert}(e(v_i, v_j))$
end for
$G \leftarrow (V, E)$
return G

6.2. Inference Pipeline

After the training procedure, the models are deployed locally, in a `state_dict` format, which consists of a Python dictionary object that maps each layer to its parameter tensor. In order to let MATLAB and Python communicate, a wrapper has been developed to enable the possibility to call the Python script that manages the models from a MATLAB environment. The wrapper (`runmodel.py`) is a python script that after loading the selected model in evaluation-mode, performs the graph bisection step, and is called by the broader agglomeration script (based on Algorithm 3) each time a 2-classes partitioning is needed. The agglomeration script allows to extract the underlying graph of the mesh, feed it to the Python bisection script, and then return the agglomerated mesh built from the clustered graph. Then the produced agglomerated meshes for different mesh sizes are stored locally, in order to let the multigrid solver, from the provided MATLAB package, compute the solution of a given differential problem.

7. Numerical Results

In this section we report some numerical computations.

7.1. Quality Metrics

To measure the capabilities of the model to perform mesh agglomeration we consider the following quality metrics. First, we define some useful quantities, such as the diameter of a general poligon K, as

$$\text{diam}(K) := \sup\{|x - y|,\ x, y \in K\}, \tag{20}$$

N_k is the number of elements of the mesh and the mesh size $h = \max_{i=1:N_k} \text{diam}(K_i)$, where $\mathcal{T}_h = \{K_i\}_{i=1}^{N_k}$. We define the following quality metrics, as in Antonietti and Manuzzi [2021].

1. Uniformity Factor (UF):

$$\text{UF}_i = \frac{\text{diam}(K_i)}{h}, \qquad i = 1, ..., N_k, \tag{21}$$

2. Circle Ratio (CR):

$$\mathrm{CR}_i = \frac{\max_{\{B(r)\subset K_i\}} r}{\min_{\{K_i\subset B(r)\}} r}, \qquad i = 1, ..., N_k, \tag{22}$$

where $B(r)$ is a ball of radius r, therefore we are measuring the ratio between the radius of the inscribed circle of K_i and the radius of the circumscribed circle of K_i.
Both metrics are scale independent and cannot assume a value higher than 1. Moreover, the more regular the polygons are, the closer to 1 those metrics should be.

7.2. Agglomeration Capabilities

We used a testing set composed of 200 meshes, 50 for each type. Then, after agglomerating all of them with four different methods: METIS, kmeans, SAGE-Base and SAGE-Res, we computed the average value of the considered quality metrics, as shown in Table 1.

Method	**Triangles**	**Random Triangles**	**Random Voronoi**	**Squares**
METIS	0.3521	0.3282	0.3681	0.5669
k-means	0.5036	0.3698	0.4181	0.7071
SAGE-Base	0.4336	0.3555	0.3720	0.6465
SAGE-Res	0.3512	0.3582	0.3743	0.6370

Table 1: Average value of the Circle Ratio (CR) for each of the set of grids. It is possible to observe that k-means algorithm presents the highest values for the whole set of grids. This was predictable, since k-means is based on this metric. However, the SAGE methods produce very similar results for non-regular meshes, while for regular ones they generate better results than METIS.

Method	**Triangles**	**Random Triangles**	**Random Voronoi**	**Squares**
METIS	0.8914	0.7832	0.8164	0.9168
k-means	0.9572	0.8139	0.8488	1.0
SAGE-Base	0.9490	0.8198	0.8563	0.9689
SAGE-Res	0.9394	0.8168	0.8496	0.9612

Table 2: Average value of the Uniformity Factor (UF) for each of the set of grids. The SAGE methods produce very similar results to the k-means algorithm, specifically the latter one is slightly better than the former for regular grids, while the situation changes for non-regular ones. METIS produces the worst values.

Finally, in order to visualize the distribution of these metrics, we decided to randomly select one mesh for each type from the testing dataset. In Figure 10 below we plot the original meshes together with the agglomerated ones.

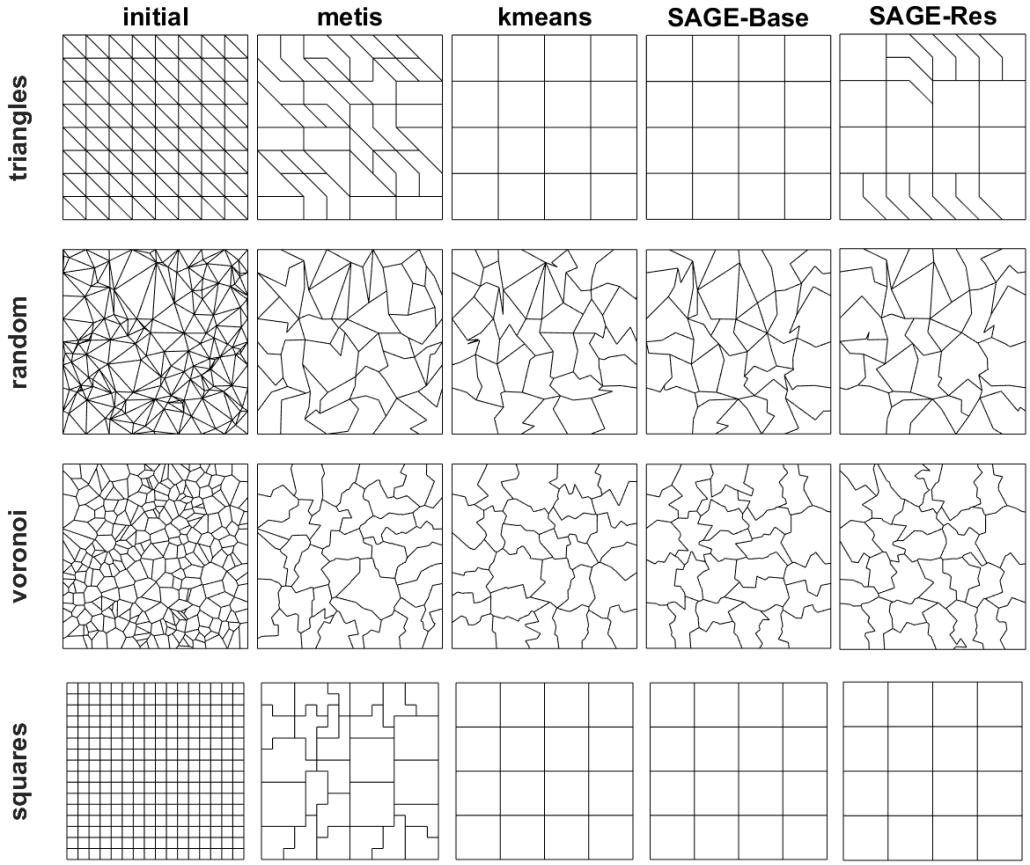


Figure 10: We can see that the SAGE-Base algorithm produces the same grids for the regular ones, trinagles and squares, while the SAGE-Res only for the latter type.

To visualize the distribution of the metrics for the considered grids in Figure 11 we plot different histograms for each type and metric.

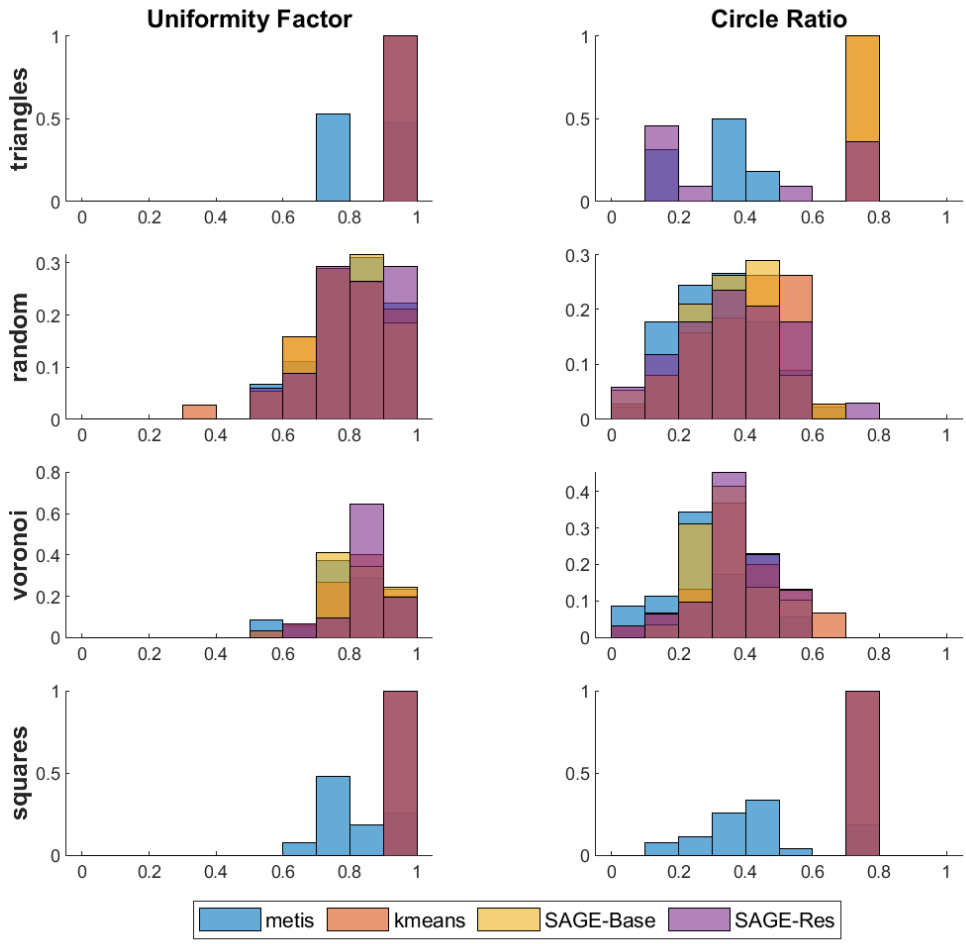


Figure 11: Histogram of the quality metrics for the samples selected.

The histogram confirms our expectations. In the case of regular meshes (triangles, squares) the k-means algorithm performs very well, however both the SAGE-Base and SAGE-Res methods achieve very good results, as can be seen by the CR/UF distributions. For the random triangles and random voronoi, which are non-regular, these three methods show a similar general distribution, with METIS having the worst distributions out of the four.

7.3. Shape Extrapolation

In order to test the shape generalization capabilities of GNN-based models we have restricted the training of the models only to one specific type of meshes (random Voronoi) and then we have performed a uniformity test on the four types of meshes, using the same sample as in Section 7.2, measuring the shape-extrapolation performances.

Method	Triangles	Random Triangles	Random Voronoi	Squares
SAGE-Base	0.7071	0.3522	0.3520	0.7071
SAGE-Res	0.4147	0.3554	0.3624	0.7071
SAGE-Base Voronoi	0.2679	0.3354	0.3386	0.3227
SAGE-Res Voronoi	0.3146	0.3233	0.3998	0.3068

Table 3: Averages values of the Circle Ratio (CR) measure.

Method	Triangles	Random Triangles	Random Voronoi	Squares
SAGE-Base	1.0	0.7994	0.8422	1.0
SAGE-Res	1.0	0.8204	0.8487	1.0
SAGE-Base Voronoi	0.9116	0.7982	0.8137	0.7883
SAGE-Res Voronoi	0.8539	0.7480	0.7193	0.7514

Table 4: Averages values of the Uniformity Factor (UF) measure.

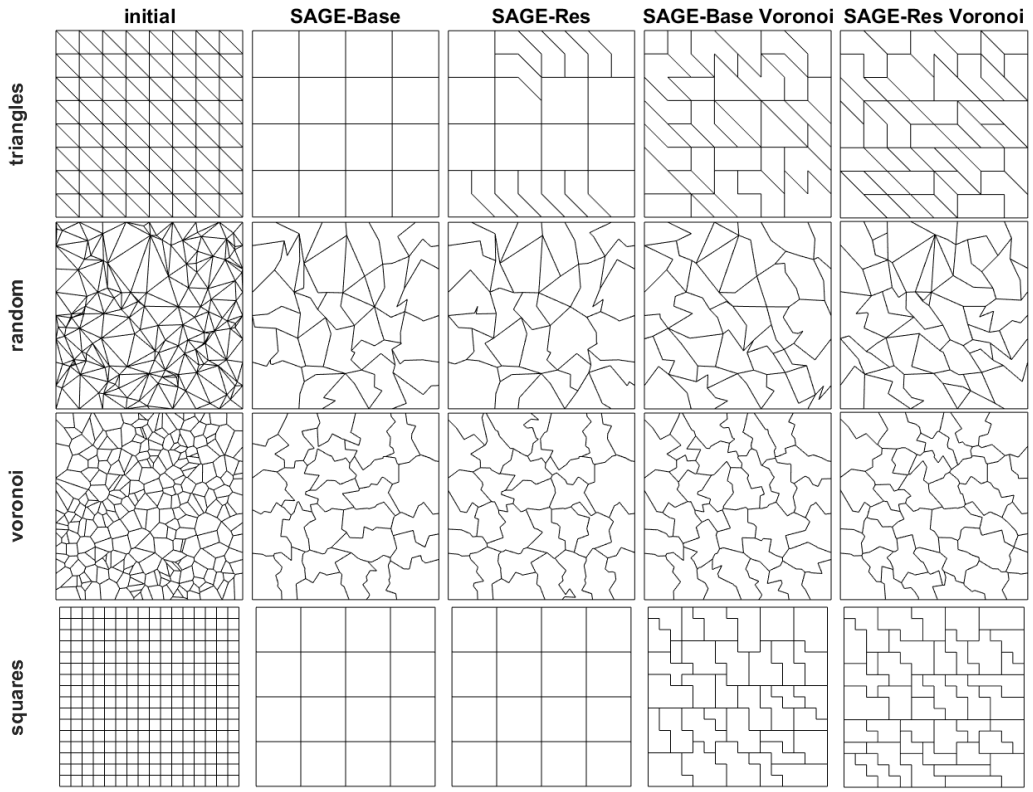


Figure 12: It is easy to observe that the biggest differences are in the agglomeration of regular meshes, especially the squares one, while for the non-regular grids the different algorithms produce similar results, this is in agreement with the metrics' values.

7.4. Runtime Performances

The main advantage of our deep learning-based solution is in its non-iterative nature, with respect to state of the art methods, such as k-means or METIS, being both iterative algorithms. The non-iterative structure of our models allow us to perform graph bisection in a single forward pass, leading to a lower asymptotic computational complexity. To make a fair comparison, we opted to test methods with respect to their graph bisection capabilites, since, testing on the full agglomeration case, would have meant involving multiple environments (MATLAB, Python), and obviously our models would be penalized due to MATLAB-Python communications. The time benchmarking test set contains 21 random Voronoi meshes of increasing size, from 25 to 5000 elements. Since runtimes are not deterministic, but rather random variables, and since this random behaviour is accentuated on cloud platforms with shared resources (such as Google Colab), we opted to perform a sampling of 20 runtimes, for each mesh in the test set.

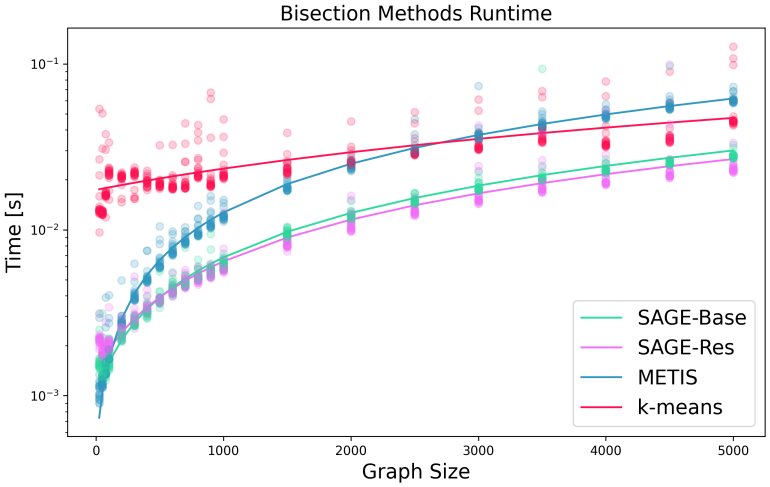


Figure 13: Bisection runtimes [s] vs mesh sizes [25, 50, 75, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000]. Time's y-axis is logarithmic.

As can be seen in Figure 13, both our GNNs-based models outperform METIS and k-means. An interesting observation is that SAGE-Res has better runtime performances than SAGE-Base, besides its a deeper model. Indeed, employing a U-Net style architecture has allowed us to increase the depth of the model, meaning with a larger number of layers, and then of hidden representations, by keeping at the same time the number of parameters in the graph convolutional block relatively low. This highlights the possibilities to further develop GNN-based architectures that not only achieve better agglomeration results, but that lead more computationally-efficient solutions.

8. Applications

8.1. Multigrid Methods

In this section we aim to measure the performances of our models in a real case scenario, in particular by feeeding the output agglomeration to a Multigrid solver.

8.1.1. Overview of Multigrid Methods

In the field of numerical analysis, a multigrid method (MG method) is an algorithm for solving differential equations using a hierarchy of discretizations of the physical domain. The core of these methods is to accelerate the convergence of a basic iterative method, a practice which is knows as relaxation, by a global correction of the fine grid solution approximation from time to time, accomplished by solving a coarse problem.
In order to accelerate the convergence of an iterative method we can use an auxiliary numerical problem obtained by the discretization of the same differential problem on a less refined mesh. Firstly, do some iterations on the fine grid, compute the associated residue and shrink it to a coarser grid. Then, solve the reduced linear system and extend the solution to the finer grid. Moreover, MG methods can be used as solvers as well as preconditioners. For instance, possible multigrid schemes are the two-level, the V-cycle and the W-cycle multigrid algorithms. However, in order to make use of them two key ingredients are needed: the inter-grid transfer operators and the smoothing scheme. The two-level, as the name suggests, uses a set of two grids, while the other two methods require more levels of agglomeration.
For these reasons, to have a computationally efficient algorithm, it is fundamental to have a powerful agglomeration method, both in terms of computational cost and in the quality of the generated grids. This justifies the need for agglomeration algorithms with a lower computational complexity, with respect to iterative methods.

8.1.2. DG Formulation

We define a jump and average operators across the edges $F \in \mathcal{F}_j$, $j = 1, ..., J$. Let $\boldsymbol{\tau}$ and v be sufficiently smooth functions. Let $\boldsymbol{\tau}^{\pm}$ and $v^{\pm}$ be the traces of $\boldsymbol{\tau}$ and v on F from $K^{\pm}$, respectively, and let $\mathbf{n}^{\pm}$ be the outward unit normal vector to $\partial K^{\pm}$.

$$
\begin{aligned}
[\![\boldsymbol{\tau}]\!] &= \boldsymbol{\tau}^+ \cdot \mathbf{n}^+ + \boldsymbol{\tau}^- \cdot \mathbf{n}^-, \qquad \{\{\boldsymbol{\tau}\}\} = \frac{\boldsymbol{\tau}^+ + \boldsymbol{\tau}^-}{2}, \qquad F \in \mathcal{F}_j^I \\
[\![\mathbf{v}]\!] &= \mathbf{v}^+ \mathbf{n}^+ + \mathbf{v}^- \mathbf{n}^-, \qquad \{\{\mathbf{v}\}\} = \frac{\mathbf{v}^+ + \mathbf{v}^-}{2}, \qquad F \in \mathcal{F}_j^I
\end{aligned} \tag{23}
$$

If $F \in \mathcal{F}_j^B$ is a boundary face, we set $\{\{\boldsymbol{\tau}\}\} = \boldsymbol{\tau}$, $[\![\mathbf{v}]\!] = v\mathbf{n}$.

Let $\mathcal{R}_j : [L^1(\mathcal{F}_j)]^2 \to [V_j]^2$ be the lifting operator defined as

$$
\int_\Omega \mathcal{R}_j(\mathbf{q}) \cdot \boldsymbol{\eta} = -\int_{\mathcal{F}_j} \mathbf{q} \cdot \{\{\boldsymbol{\eta}\}\} ds \qquad \forall \boldsymbol{\eta} \in [V_j]^2 \tag{24}
$$

The bilinear form $\mathcal{A}_j(\cdot,\cdot) : V_j \times V_j \to \mathbb{R}$ corresponding to the symmetric interior penalty DG method is defined by

$$
\mathcal{A}_j(u,v) = \sum_{k \in \mathcal{T}_j} \int_k [\nabla u \cdot \nabla v + \mathcal{R}_j([\![u]\!]) \cdot \nabla v + \mathcal{R}_j([\![v]\!]) \cdot \nabla v] dx + \sum_{F \in \mathcal{F}_j} \int_F \sigma_j [\![u]\!] \cdot [\![v]\!] ds \tag{25}
$$

where $\sigma_j \in L^\infty(\mathcal{F}_j)$ denotes the interior penalty stabilization function

$$
\sigma_j(x) = C_\sigma^j \frac{p_j^2}{\{h_K\}_H}, \qquad x \in F \in \mathcal{F}_j \tag{26}
$$

with $C_\sigma^j > 0$ independent of p_j, $|F|$ and |K|, and where $\{\cdot\}_H$ is

$$
\{h_K\}_H = \begin{cases} \frac{2h_{K^+} h_{K^-}}{h_{K^+} + h_{K^-}}, & F \in \mathcal{F}_j^I,\ \overline{F} \subset \partial\overline{K^+} \cap \partial\overline{K^-} \\ h_K, & F \in \mathcal{F}_j^B,\ \overline{F} \subset \partial\overline{K} \cap \partial\overline{\Omega} \end{cases} \tag{27}
$$

8.1.3. Multigrid Framework

The key ingredients for the multigrid solvers are the inter-grid transfer operators, i.e. the prolongation operator, which transfer vectors from the coarse grid to the fine grid, and the restriction operator, which allows to perform the inverse operation.
We denote I_j^{j-1} the prolongation operator from the coarser space V_j to the finer space V_{j-1}, that is the natural injection operator, since $V_j \subset V_{j-1}$ i.e. the spaces are nested, and the restriction operator I_{j-1}^j is the adjoint of I_j^{j-1}. We introduce also the operators $A_j : V_j \to V_j$, defined as

$$
(A_j \omega, v)_{L^2(\Omega)} = \mathcal{A}_j(\omega, v) \qquad \forall \omega, v \in V_j, \qquad j = 1, ..., J \tag{28}
$$

and we denote by $\Lambda_j \in \mathbb{R}$ the maximum eigenvalue of A_j, $j = 2, ..., J$. Moreover let Id_j be the identity operator on the level V_j. We choose as smoothing scheme the Richardson method with

$$
B_j = \Lambda_j \, \mathrm{Id}_j \qquad j = 2, ..., J \tag{29}
$$

In our specific case we use the V-cycle algorithm with 3 levels of mesh refinement. Given the polynomial degree for the DG method and the number of pre- and post-smoothing steps, the recursive procedure in outlined in Algorithm 5, where we observe that on the level $j = J$ the problem is solved exactly.

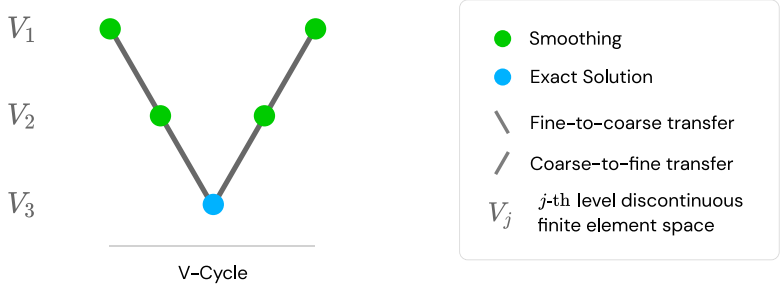


Figure 14: Multigrid V-cycle scheme.

Algorithm 5 One iteration of the Multigrid V-cycle scheme

if $j = J$ **then**
  $\mathrm{MG}_\nu(J, f, z_0, m) = A_J^{-1} f$
else
  Pre-smoothing:
  for $i = 1, ..., m$ **do**
    $z^{(i)} = z^{(i-1)} + B_j^{-1}(g - A_j z^{(i-1)})$
  end for
  Coarse grid correction:
  $r_{j+1} = I_{j-1}^{j}(f - A_j z^{(m)})$
  $e_{j+1} = \mathrm{MG}_\nu(j+1, r_{j+1}, 0, m)$
  $z^{(m+1)} z^{(m)} + I_j^{j-1} e_{j-1}$
  Post-smoothing:
  for $i = m+2, ..., 2m+1$ **do**
    $z^{(i)} = z^{(i-1)} + B_j^{-1}(g - A_j z^{(i-1)})$
  end for
  $\mathrm{MG}_\nu(J, f, z_0, m) = z^{2m+1}$
end if

8.1.4. Test Case: Poisson Problem

As in Antonietti and Pennesi [2019] we consider as test case the following Poisson problem with homogeneous Dirichlet boundary conditions:

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases} \tag{30}$$

and particularly, its weak formulation:

$$\text{Find } u \in V = \mathrm{H}^2(\Omega) \cap \mathrm{H}_0^1(\Omega) \quad \text{s.t.} \quad \mathcal{A}(u, v) = \int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx, \qquad \forall v \in V \tag{31}$$

with $\Omega \subset \mathbb{R}^2$, a polygonal domain with Lipshitz boundary and $f \in L^2(\Omega)$. Since we want to apply a multigrid method to solve the problem, we introduce a sequence of nested meshes on the domain Ω, each of which composed by disjoint open elements of diameter h_k, such that $\overline{\Omega} = \bigcup_{K \in \mathcal{T}_j} \overline{K}$, $j = 1, ..., J$, where J is the number of agglomeration levels being considered. We define the mesh size of $\mathcal{T}_j$ as $h_j = \max_{K \in \mathcal{T}_j} h_K$. To each $\mathcal{T}_j$ we associate the corresponding finite element space V_j, defined as

$$V_j = \{v \in L^2(\Omega) : v|_K \in \mathcal{P}_{p_j}(K),\ K \in \mathcal{T}_j\} \tag{32}$$

where $\mathcal{P}_{p_j}(K)$ is the space of polynomials of total degee at most $p_j \geq 1$ on $K \in \mathcal{T}_j$.
By applying a MG scheme to solve the problem (31) on the meshes agglomerated by our GNN-based models we aim to verify the quality of the generated grids with respect to the ones the state of the art methods produce. Clearly this type of comparison cannot be defined by a quality metric, however we can use as a kind of quality indicator the number of iterations the scheme needs to reach convergence.
In Table 5 we compare the performances of 4 different types of agglomeration algorithms, METIS, k-means, SAGE-Base and SAGE-Res, on four different mesh types. As can be seen, the number of iterations needed by the MG solver for converging with the GNN-based agglomeration (SAGE-Base, SAGE-Res) is comparable to the cases where state-of-the-art agglomeration methods were employed. The cases in which the GNN-methods are outperformed by the other ones, are characterized by a small difference in terms of number of iterations, meaning that even in the worst case, we remain in an acceptable range.

			Grids			
s	p	**Method**	**Triangles**	**Random triangles**	**Random Voronoi**	**Squares**
5	2	METIS	8	41	**16**	23
		k-means	8	**36**	19	23
		SAGE-Base	8	46	19	23
		SAGE-Res	8	37	17	23
	3	METIS	**11**	55	**22**	**31**
		k-means	12	**54**	25	32
		SAGE-Base	12	55	25	32
		SAGE-Res	13	55	25	33
6	2	METIS	6	33	**13**	17
		k-means	6	27	15	17
		SAGE-Base	6	35	15	17
		SAGE-Res	6	**27**	14	17
	3	METIS	**10**	**39**	**20**	**25**
		k-means	11	41	24	28
		SAGE-Base	11	41	**20**	28
		SAGE-Res	**10**	**39**	22	27
7	2	METIS	**5**	25	**11**	13
		k-means	**5**	23	12	13
		SAGE-Base	**5**	27	**11**	13
		SAGE-Res	6	**21**	**11**	13
	3	METIS	**7**	**32**	17	**20**
		k-means	8	**32**	18	21
		SAGE-Base	8	33	**15**	21
		SAGE-Res	9	33	18	22
8	2	METIS	**4**	21	**9**	11
		k-means	5	**18**	10	11
		SAGE-Base	5	22	**9**	11
		SAGE-Res	5	**18**	10	11
	3	METIS	**7**	27	**13**	**17**
		k-means	8	27	15	18
		SAGE-Base	8	26	**13**	18
		SAGE-Res	8	**25**	15	18
9	2	METIS	4	17	8	**9**
		k-means	4	15	8	**9**
		SAGE-Base	4	17	8	**9**
		SAGE-Res	4	**14**	8	10
	3	METIS	**6**	22	12	**14**
		k-means	7	**21**	12	**14**
		SAGE-Base	7	22	**11**	**14**
		SAGE-Res	7	**21**	12	15
10	2	METIS	4	15	7	8
		k-means	4	**13**	7	8
		SAGE-Base	4	15	7	8
		SAGE-Res	4	**13**	7	8
	3	METIS	**5**	19	10	12
		k-means	6	19	11	12
		SAGE-Base	6	19	**9**	12
		SAGE-Res	6	**18**	11	12

Table 5: The table shows the number of iterations required by the multigrid solver to converge, depending on the smoothness (s) and the polynomial degree (p). It is possible to observe that GNN-based method is comparable to the state of the art ones, moreover it outperforms the other methods in the case of random triangles grids, while SAGE-Base is slightly better than SAGE-Res in the random Voronoi case.

8.2. Human Brain

In order to further test the generalization capabilities of our model, we performed a test on a much more complex domain, with respect to the meshes the network was trained on, both in terms of shape, dimensions and number of elements. The employed grid consists of a coronal section cut off from a 3D mesh of a human brain and it is composed of exactly 14372 not regular triangular elements. Therefore, it does not have a squared domain and presents many irregularities, such as constrictions near the outer boundary and narrowed sections.

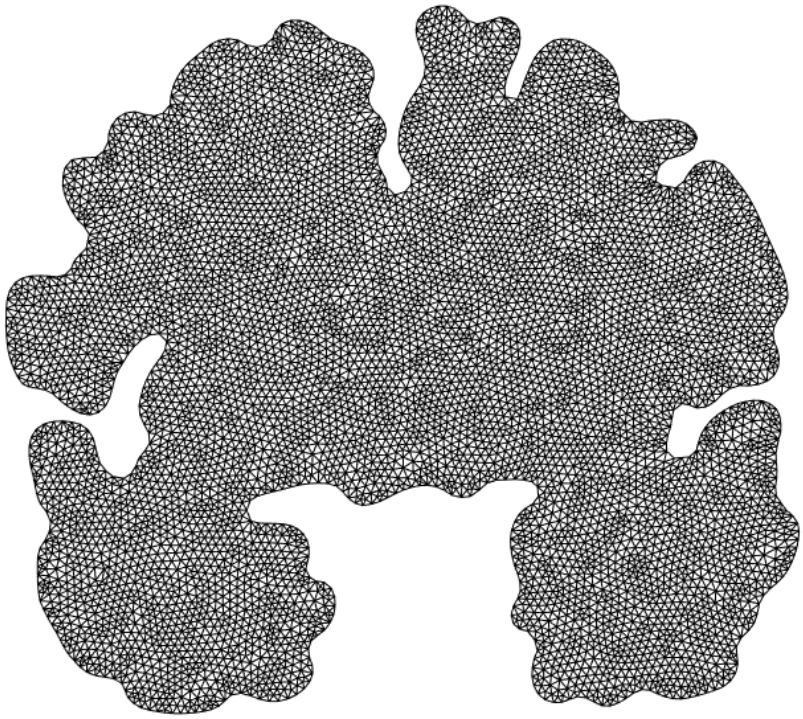
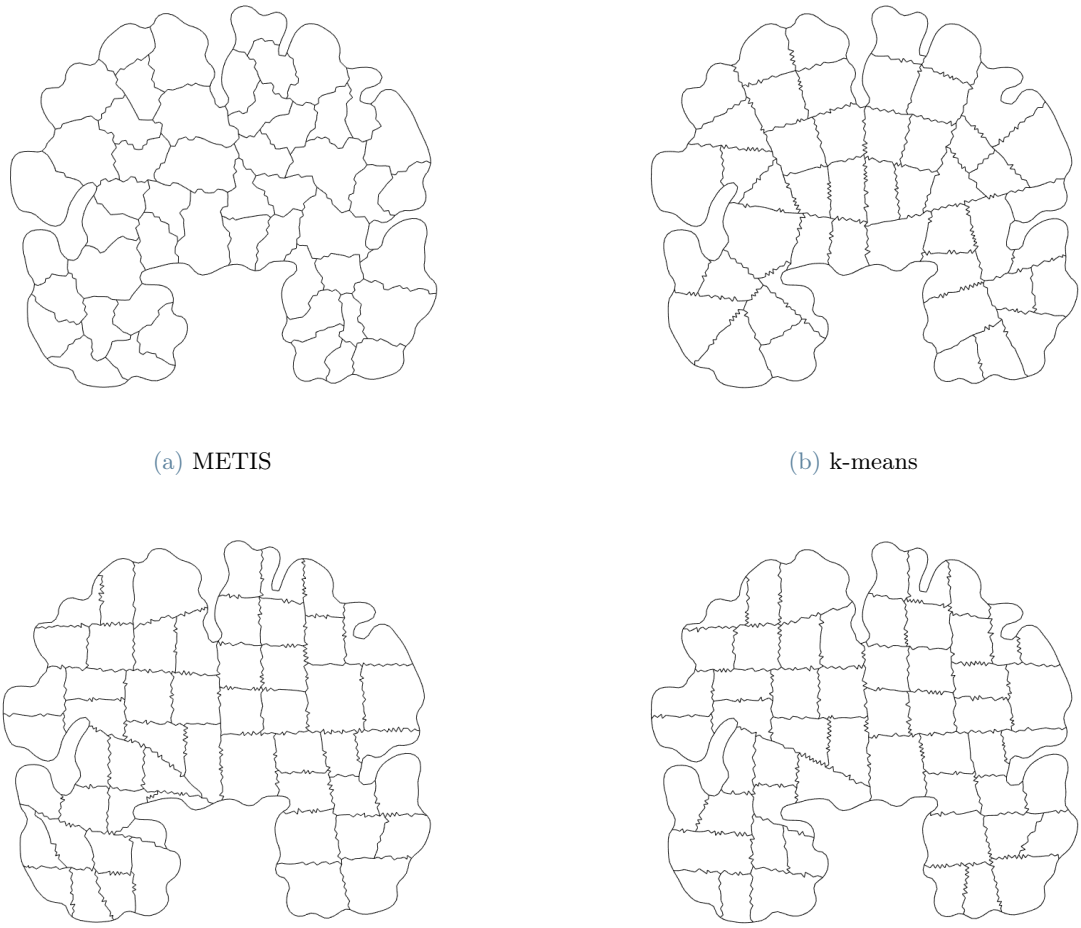


Figure 15: Initial discretization of the brain section.

Due to the mesh complexity, its agglomeration is a computationally costly process, resulting in a runtime of 15 min for our GNN-based methods. Testing both the k-means and METIS algorithms on the same mesh however has revealed the computational capabilities of the model, especially with respect to the latter one. Also we need to take into account that the model is still implemented in a hybrid form, therefore to really grasp its power we would need to compare the methods on the same ground, as we have performed in Section 7.4.
In the MATLAB agglomeration script we set a vector of different target sizes for the agglomerated grids (h^*) and produce a mesh for each value. In order to define the vector we take the maximum distance between two vertices of the mesh (D) and multiplying it by a number which belongs to the interval $[0, 1]$. We set:

$$h^* = D \cdot [0.9,\ 0.8,\ 0.7,\ 0.6,\ 0.5,\ 0.4,\ 0.3,\ 0.2]. \tag{33}$$

Below it is possible to observe the resulting meshes using the four usual methods: METIS, k-means, SAGE-Base and SAGE-Res.



(a) METIS

(b) k-means

(c) SAGE-Base

(d) SAGE-Res

Figure 16

In Table 6 we report the average metrics, Circle Ratio and Uniformity Factor, computed over the whole agglomeration

Method	**CR**	**UF**
METIS	0.4263	0.7822
k-means	0.5181	0.7935
SAGE-Base	0.4649	0.7864
SAGE-Res	0.4682	0.7773

Table 6: Circle ratio (CR) and uniformity factor (UF) for the different methods employed. As can be seen the GNN-based methods related metrics are comparable to those of the iterative methods (k-means, METIS) even if the models were trained only on meshes with significantly less elements (maximum 1000 elements).

We also plot the histogram of the quality metrics in order to see the complete distribution.

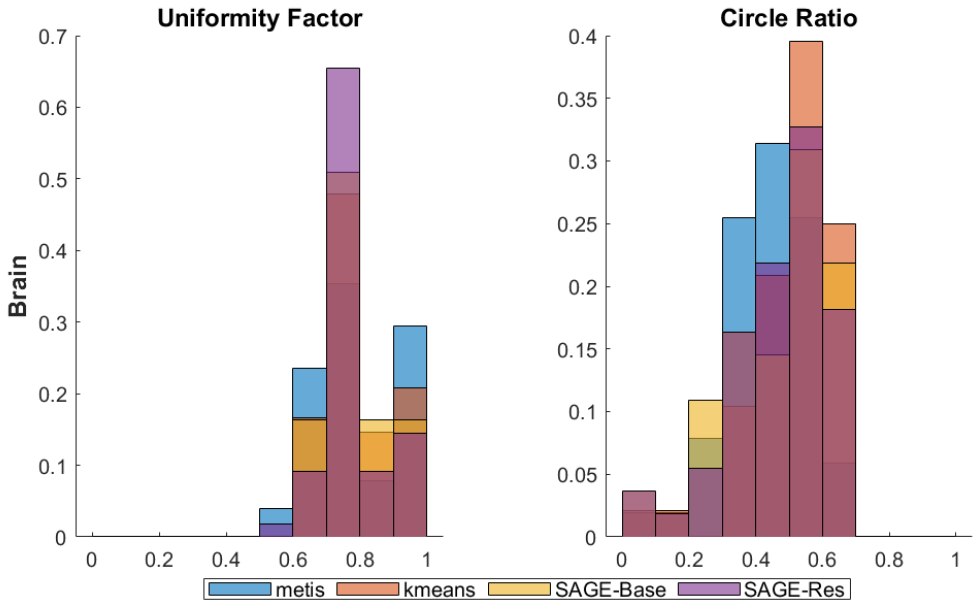


Figure 17: Distribution of the quality metrics (UF,CR) for the agglomerated brain mesh.

It is possible to see that for all the methods we used the values have a similar behavior, therefore, we can affirm that the SAGE-based algorithms are comparable to the state of the art ones, even if the network was trained on meshes much smaller and more regular in terms of domain than this one.

9. Conclusions

In this project we developed a Deep Learning-based approach to tackle the problem of polygonal grids agglomeration via a graph partitioning algorithm based on Graph Neural Networks. By generating a diverse dataset we were able to show the high degree of generalization achievable by GNN-based models, in terms of mesh size, elements' shape and domain complexity. Furthermore, by feeding the agglomerated grids to a multigrid solver, applied to a Poisson problem with different values of smoothness and polynomial degree, we observed that the number of iterations needed for convergence is comparable to the cases in which METIS and k-means agglomerations were employed. This results make us confident about the possibility to employ GNN-based methods to real case scenarios. Having carried out multiple tests, we can affirm that GNN-based models represent a promising approach to tackle graph-structured problems, likewise the one of mesh agglomeration. Moreover our approach can be extended to much broader graph partioning/clustering-related problems, with the possibility to tackle different problems with a similar framework. Further improvements can be made both to the DL framework, by scaling the network or by reframing the approach to a reinforcement learning one, as proposed by Gatti et al. [2021b], or to the global agglomeration algorithm, by parallelizing graph bisection routines.

Acknowledgements

We would like to thank Prof. Paola Francesca Antonietti for following us during the development of the project and for giving us the opportunity to further develop it by applying it to real cases. Moreover, we would like to also thank Dr. Enrico Manuzzi for all the help we received, for the provided MATLAB package, and for his willingness to do early morning meetings.

References

P. Antonietti and G. Pennesi. V-cycle multigrid algorithms for discontinuous galerkinmethods on non-nested polytopic meshes. *Journal of Scientific Computing*, 78:625–652, 2019.

P. F. Antonietti and E. Manuzzi. Refinement of polygonal grids using convolutional neural networks with applications to polygonal discontinuous galerkin and virtual element methods. *Journal of Computational Physics 452 (2022) 110900*, 2021. doi: 10.1016/j.jcp.2021.110900.

M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

A. Gatti, Z. Hu, T. Smidt, E. G. Ng, and P. Ghysels. Deep learning and spectral embedding for graph partitioning, 2021a.

A. Gatti, Z. Hu, T. Smidt, E. G. Ng, and P. Ghysels. Graph partitioning and sparse matrix ordering using reinforcement learning and graph neural networks, 2021b.

W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs, 2017.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

G. Karypis and V. Kumar. Kumar, v.: A fast and high quality multilevel scheme for partitioning irregular graphs. siam journal on scientific computing 20(1), 359-392. *Siam Journal on Scientific Computing*, 20, 01 1999. doi: 10.1137/S1064827595287997.

J. Macqueen. Some methods for classification and analysis of multivariate observations, 1967.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

A. Training Implementation

The training loop was based on the following algorithm

Algorithm 6 Training Procedure

Input Model $\mathcal{M}$ with learnable weights W, Training set $\mathcal{D}_{\text{train}} = \{(G_i, X_i)\}_{i=1}^{N_{\text{train}}}$
Validation set $\mathcal{D}_{\text{val}} = \{(G_i, X_i)\}_{i=1}^{N_{\text{val}}}$, learning rate lr, batch size b
max epochs number `maxEpochs`, Optimizer $\mathcal{O}$, learning rate scheduler $\mathcal{S}_{lr}$
Output Trained model $\mathcal{M}$

for $i = 1, ...,$ `maxEpochs` **do**
  shuffle $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}$
  for $G_{\text{batch}}, X_{\text{batch}} \in \mathcal{D}_{\text{train}}$ **do**
    random rotate $G_i \in G_{\text{batch}}$
    $Y_{\text{batch}} \leftarrow \mathcal{M}(G_{\text{batch}}, X_{\text{batch}})$
    $\mathcal{L}_{\text{batch}}^{\text{train}} \leftarrow \mathcal{L}(Y, G_{\text{batch}})$
    $W \leftarrow \mathcal{O}(\mathcal{L}_{\text{batch}}^{\text{train}}, lr)$
    $lr \leftarrow \mathcal{S}_{lr}(\mathcal{L}_{\text{batch}}^{\text{train}}, lr)$
  end for
  for $G_{\text{batch}}, X_{\text{batch}} \in \mathcal{D}_{\text{val}}$ **do**
    random rotate $G_i \in G_{\text{batch}}$
    $Y_{\text{batch}} \leftarrow \mathcal{M}(G_{\text{batch}}, X_{\text{batch}})$
    $\mathcal{L}_{\text{batch}}^{\text{val}} \leftarrow \mathcal{L}(Y, G_{\text{batch}})$
  end for
end for
return $\mathcal{M}$

and was implemented in the following manner, where the shuffling operation of the datasets was performed by calling the `sklearn` built-in function. All the developed functions to manage the datasets, get samples and perform transformations are available online within the repository of the project, together with the complete training python notebook.

```
from sklearn.utils import shuffle
import random
optimizer = torch.optim.Adam(model.parameters(), lr=1e-5, weight_decay=1e-5)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',factor
    =0.5, patience=20, threshold=0.0001, threshold_mode='abs')

epochs=300
batch_size=4

loss_array = []
loss_val_array = []
epoch_loss_array = []
epoch_loss_val_array = []
lr_array = []

start = time.time()

for epoch in range(1, epochs+1):

    adjacencies_s, coords_s, areas_s = shuffle(adjacencies, coords, areas)
    adjacencies_test_s, coords_test_s, areas_test_s = shuffle(adjacencies_test,
    coords_test, areas_test)

    model.train()
    loss=torch.tensor([0.]).to(device)

    # Training
    for i in range(dataset_size):
      g = (get_sample(adjacencies_s, coords_s, areas_s,i,
```

```
                        randomRotate=bool(random.getrandbits(1)),
                        selfloop=True)).to(device)
      out = model(g.x,g.edge_index)
      loss += loss_normalized_cut(out, g)

      if i%batch_size==0 or i==dataset_size-1:
        optimizer.zero_grad()
        loss_array.append(loss.item())
        loss.backward()
        optimizer.step()
        loss=torch.tensor([0.]).to(device)

    epoch_loss_array.append(sum(loss_array[-batch_size:])/batch_size)
    scheduler.step(loss_array[-1])
    lr_array.append(optimizer.param_groups[0]['lr'])

    # Validation
    model.eval()
    loss_val = torch.tensor([0.]).to(device)

    for i in range(dataset_size_test):
      g_val = (get_sample(adjacencies_test, coords_test, areas_test,i,
                          randomRotate=bool(random.getrandbits(1)),
                          selfloop=True)).to(device)
      out_val = model(g_val.x,g_val.edge_index)
      loss_val += loss_normalized_cut(out_val, g_val)
      if i%batch_size==0 or i==dataset_size-1:
        loss_val_array.append(loss_val.item())
        loss_val = torch.tensor([0.]).to(device)
    epoch_loss_val_array.append(sum(loss_val_array[-batch_size:])/batch_size)

    print('epoch:', epoch,
          '\t\tloss:', round(epoch_loss_array[-1],5),
          '\t\tvalidation loss:', round(loss_val_array[-1],5),
          '\t\tlr:', optimizer.param_groups[0]['lr'])
training_time = time.time()-start
```

Listing 2: Python training/validation loop.