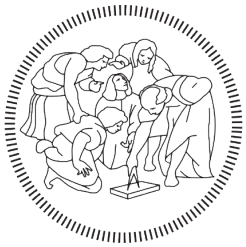


POLITECNICO
MILANO 1863

POLITECNICO DI MILANO

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING

Mathematical Engineering

Graph Neural Networks for solving graph problems

Authors: **Francesca Behrens** (10617071)
Paolo Botta (10612869)

Advisor: Prof. Paolo Zunino

Co-advisor: Prof. Andrea Manzoni
Dr. Nicola Rares Franco
Dr. Piermario Vitullo

Advanced Programming for Scientific Computing

A.Y. 2022/2023

Abstract

This report illustrates the work done for the course Advanced Programming for Scientific Computing (Prof. Luca Formaggia) held at Politecnico di Milano during the academic year 2022/2023.

The main purpose of this report is to investigate the relations between Graph Neural Networks (GNNs), a special type of Artificial Neural Network, and graph-defined problems. In particular, we have shown that GNNs are well-suited to solving such problems. The workflow we followed was: first, we investigated the theory behind the problems on graph and Graph Neural Networks, and then we prepared some numerical tests to show our claims. Our experiments were done on artificial data that was created by us. All the code we implemented is available on the following GitHub repository:

https://github.com/ploki99/GP-GNN

The most important result is the remarkable reduction in inference time we achieved using the Graph Neural Network to solve classical graph problems. Because of this very reason, we think that our work can be easily extended to real-life applications with very good results.

Keywords: Graph Neural Networks, graphs, graph Laplacian, Partial Differential Equations

Contents

Chapter 1

Introduction

Graphs provide a versatile framework for representing interconnected data, and their utility extends to all disciplines, from the social sciences to biology. This report explores some mathematical applications of graph neural networks (GNNs), a class of neural networks tailored to handle graph-structured data. For a rigorous introduction to network science, we refer to the book of Barabási [2].

Real-world networks have a unique structure that often differs from random mathematical networks. For example, we can mention the following:

- **Social networks**: they provide a rich source of graph drawing problems because they appear in an incredibly wide variety of forms and contexts.
- **Biological networks**, such as the brain neural network and the cardiovascular system.

Social network theory is based on the belief that individuals and organizations, which may appear independent, are connected within social relationships and interactions. The term "social network" was created to differentiate the relational perspective from other research approaches focusing on social groups and categories. In general, a social network comprises actors, such as individuals or organizations, together with some kind of relationship that connects them, often of a social nature. The structure of the network is typically represented as a graph, where actors are represented as vertices, and relationships are represented as edges, indicating the presence of a connection between two actors. For more details, see [3]. In `Figure` 1.1, you can see an example of a social network, where the nodes represent the people involved and the edges the relations between them.

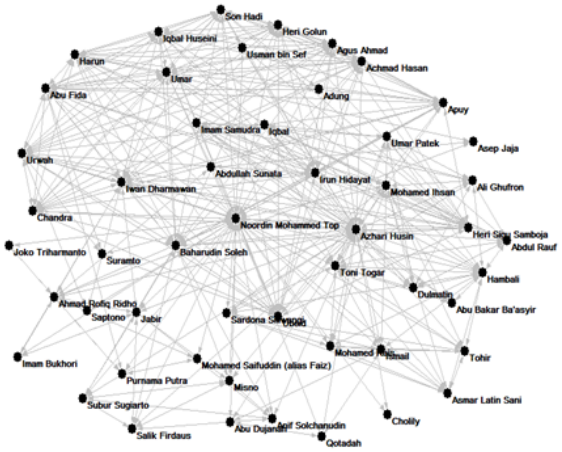


Figure 1.1: Example of social network, taken from [9]

For what concerns biological networks, the brain's neural networks are intricate systems of interconnected neurons that facilitate the transmission and processing of information. The brain's neural networks play a crucial role in various cognitive functions such as perception, learning, memory, and decision making. These networks are formed by the intricate wiring of billions of

neurons, which transmit electrochemical signals across synapses. In graph terms, we can think of neurons as the nodes of a graph and synapses as its edges. Moreover, such networks are very important because artificial neural networks (ANNs) are computational models inspired by the structure and functioning of the brain's neural networks. For more content on brain networks and brain organization, we suggest reading the work of Luiz Pessoa [11], where here we report one of the illustrations in `Figure` 1.2.

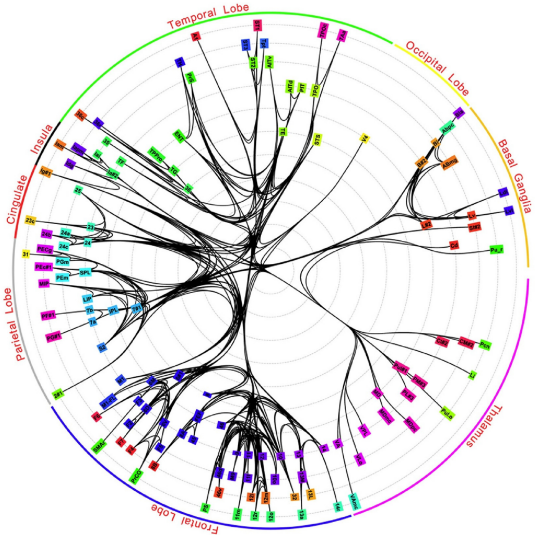


Figure 1.2: Simplified whole-brain network connectivity structure

The cardiovascular system is a vital network of organs, vessels, and tissues that work together to transport oxygen, nutrients, hormones, and waste products throughout the body. It consists of the heart, blood vessels (arteries, veins, and capillaries), and blood. Graphs can effectively represent and illustrate the functioning of the cardiovascular system, showing key parameters such as heart rate, blood pressure, and blood flow. These graphs provide valuable information on the interactions and dynamics of this complex system, helping healthcare professionals and researchers better understand and diagnose cardiovascular diseases and develop effective treatments. A very nice example can be found in the work of Pozrikidis [12], where the microcirculation in capillary networks is treated as a tree (which is a particular type of graph). An illustration of such a tree is in `Figure` 1.3.

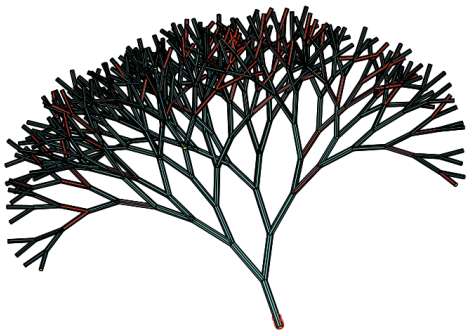


Figure 1.3: A tree-like capillary network model

Graph neural networks (GNNs) are a class of machine learning models that have gained considerable attention in recent years. Unlike traditional neural networks, which operate on grid-like data structures such as images or sequences, GNNs are designed to work with graph-structured data such as social networks, citation networks, or molecular structures. The key idea of GNNs is to exploit the connections and relationships between the nodes of a graph to improve learning and prediction tasks. With their growing popularity and promising results, GNNs have become a key

tool for analyzing, understanding, and making predictions on various real-world graph data. An incredible recent result by Remi Lam et al. [7] uses GNNs to learn effective medium-range global weather forecasting.

In our work, we used Graph Neural Networks to solve problems defined on graphs: in particular we investigated the resolution of the Laplacian system and elliptic partial differential equations. We want to remark that we utilized only randomly generated graphs that are different from the various kinds of network we have introduced previously. What we have set out in this introduction are the possible real-life applications of our approach to GNNs. In the following section, you can find the organization we used to write this report.

1.1 Organization of the report

In the first three chapters after the introduction, we state the basic theory behind our work; in particular:

- `Chapter` 2 is devoted to graph theory;
- `Chapter` 3 introduces our problem of interest, i.e. graph Laplacian system and elliptic PDEs;
- `Chapter` 4 contains the details of the structure of the Graph Neural Network.

Then, in `Chapter` 5, all the numerical experiments we have carried out are presented in detail. In `Chapter` 6 there is the technicality of the Python implementation, where you can find more about the organization and explanation of the code. Finally, the last chapter is dedicated to the conclusions.

Chapter 2

Elements of graph theory

2.1 Basic definitions and notation

We give in what follows the basic concepts of graph theory that we will use in the following. Refer to [4] for a more comprehensive introduction to graph theory.

Definition 2.1.1. A **graph** is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$. The elements of V are called **vertices** (or nodes) of the graph G, the elements of E are its **edges**.

The usual way to picture a graph is by drawing a dot for each vertex and joining two of these dots with a line if the corresponding two vertices form an edge. An example is shown in `Figure` 2.1

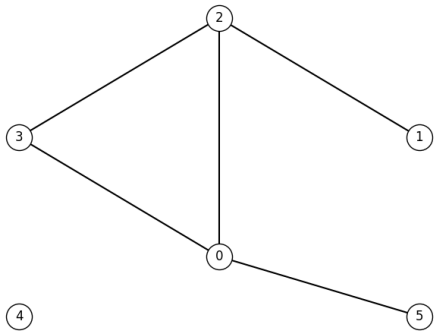


Figure 2.1: Generic graph with 6 vertices and 5 edges

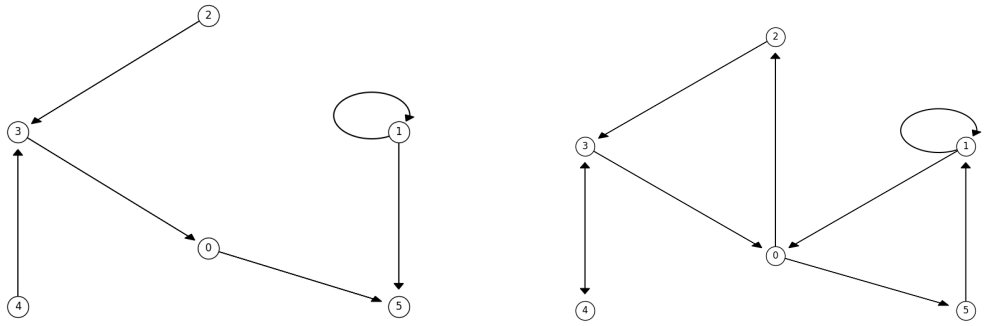
Remark. Given a graph $G = (V, E)$, we can write the sets V and E as $V = \{v_1, ..., v_N\}$ and $E = \{e_j = (v_i, v_k)\}_{j=1,...,M}$. With this notation, the graph has N vertices and M edges.

Definition 2.1.2. $G = (V, E)$ is **undirected** if no orientation is assigned to the edges, in this case, we do not distinguish between (v_i, v_j) and (v_j, v_i). Otherwise, the graph is **directed**. A directed graph admits a **self-loop** if $(v_i, v_i) \in E$

Now let us look at a graph's connectivity, distinguishing between direct and undirected graphs ([6]).

Definition 2.1.3. An undirected graph $G = (V, E)$ is **connected** if for each pair of vertices $v_1, v_2 \in V$ there exists a path from v_1 to v_2.
A directed graph $G = (V, E)$ is **strongly connected** if there is a directed path from every vertex to every other vertex. A directed graph is called **weakly connected** if its underlying undirected graph is connected.

In `Figure` 2.2 you can see the difference between strong and weak connections in a directed graph.



(a) Weakly connected (b) Strongly connected

Figure 2.2: Connection in directed graph

Let's now introduce the notion of cycles in a graph.

Definition 2.1.4. A **path** is a non-empty graph $P = (V, E)$ of the form

$$V = \{v_0, v_1, ..., v_k\} \qquad E = \{(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)\}$$

where the v_i are all distinct. We can refer to a path by the sequence of its vertices, writing $P = v_0 v_1 ... v_k$.

Definition 2.1.5. If $P = v_0 ... v_k$ is a path and $k \geq 2$ then the graph $C = P + v_k v_0$ is called a **cycle**. A graph without cycles is called an **acyclic graph**.

Remark. For a directed graph, self-loops are considered cycles and graphs like $C = v_0 v_1 v_0$ are cyclic graphs.

Definition 2.1.6. A graph $G = (V, E)$ is **sparse** if $M = O(N)$, that is, if the number of edges has the same order of magnitude as the number of vertices.

2.2 Graph Laplacian

From now on, we consider undirected and sparse graphs. We introduce the notion of graph Laplacian. For a complete overview of the problem, look at [10].

Definition 2.2.1. The **adjacency matrix** A of a graph is a $N \times N$ matrix defined as

$$A_{ij} = \begin{cases} 1, & (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

Definition 2.2.2. The **degree** of a vertex v is the number of vertices that are adjacent to it. Denote this by $d(v)$. The **degree matrix** D is a diagonal matrix of dimension $N \times N$, defined by

$$D_{ij} = \begin{cases} d(i), & i = j \\ 0, & \text{otherwise} \end{cases}$$

Remark. From a graphical point of view, the degree of a vertex is the number of connections that exit the vertex. For example, in `Figure` 2.1 we have $d(0) = 3$.

Definition 2.2.3. The **graph Laplacian** (or Laplacian matrix) of a graph G is defined as

$$L = D - A$$

Remark. From the previous definitions, we can also define the graph Laplacian as

$$L_{ij} = \begin{cases} d(i), & i = j \\ -1, & (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

Example 2.2.1. The Laplacian matrix of the graph in `Figure` 2.1 is

$$L = \begin{bmatrix} 3 & 0 & -1 & -1 & 0 & -1 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 & 0 \\ -1 & 0 & -1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Definition 2.2.4. The **discrete Laplacian** of $f : V \to \mathbb{R}$ on $G = (V, E)$ is defined as

$$(\Delta_G f)(v) = \sum_{w \in A_v} f(v) - f(w)$$

where A_v is the set of adjacent vertices to v.

Proposition 2.2.1. The linear operator Δ_G is represented by the Laplacian matrix L.

Definition 2.2.5. The **incidence matrix** B of a directed graph $G = (V, E)$ is a $N \times M$ matrix defined as

$$B_{ij} = \begin{cases} -1, & \text{if edge } e_j \text{ leaves vertex } v_i \\ 1, & \text{if edge } e_j \text{ enters vertex } v_i \\ 0, & \text{otherwise} \end{cases}$$

For an undirected graph, we can arbitrarily choose the edge directions to build the incidence matrix.

Example 2.2.2. Consider the undirected graph in `Figure` 2.1. If we set the directions of the edges in increasing order (that is, if $(0, 5) \in E$ then the edge leaves 0 and enters 5) we obtain the following incidence matrix

$$B = \begin{bmatrix} -1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Proposition 2.2.2. The Laplacian matrix of an undirected graph can also be written as

$$L = BB^T$$

where B is the incidence matrix.

The Laplacian matrix has several properties. We list some of them in the following Proposition.

Proposition 2.2.3. The Laplacian matrix L of an undirected graph G has the following properties:

- L is a symmetric and positive semi-definite matrix.
- There always exists an eigenvalue of the Laplacian matrix equal to 0.
- The number of zero eigenvalues of L is the same as the number of connected graph components.

2.3 Metric and quantum graph

Definition 2.3.1. A connected graph G is said to be a **metric graph** if

1. to each edge e is assigned a length ℓ_e such that $0 < \ell_e < \infty$,
2. each edge is assigned a coordinate $x^e \in [0, \ell_e]$, which increases in a specified (but otherwise arbitrary) direction along the edge.

In general, the edges can be curved differentiable paths (without loops), but for simplicity, we assume that each edge is a straight line connecting the two vertices that define it.

Definition 2.3.2. The **volume** of a metric graph G is defined as

$$\mathbf{vol}(G) = \sum_{e \in E} \ell_e$$

We will consider only finite graphs, with all edge lengths finite, hence with $\mathbf{vol}(G) < \infty$.

Remark. With the aforementioned structure, the metric graph G transforms into a one-dimensional domain, where each edge e is assigned a variable x_e that locally represents the global coordinate variable x.

Definition 2.3.3. We define the **distance** $d(v_i, v_j)$ as the length of the shortest path in G between the vertices v_i and v_j. Endowed with this distance, G can be seen as a metric space.

Definition 2.3.4. The space $L^2(G) = \bigoplus_e L^2(e)$ is the space of all square-integrable measurable functions u on the edges of G; i.e.,

$$\|u\|^2_{L^2(G)} = \sum_{e \in E} \| u|_e \|^2_{L^2(e)} < \infty.$$

The Sobolev space $H^1(G) = \bigoplus_e H^1(e) \cap C^0(G)$ is the space of all continuous functions u on G, $u \in C^0(G)$, such that $u|_e$ belongs to $H^1(e)$ for each edge e, i.e.,

$$\|u\|^2_{H^1(G)} = \sum_{e \in E} \| u|_e \|^2_{H^1(e)} < \infty$$

The operators we discuss here are elementary but are capable of representing fascinating dynamics on metric graphs. In addition to the continuity requirement of the function in G, we will apply the **Neumann-Kirchhoff** conditions on the vertices. The condition we impose concerns the subset E_v of E, which consists of the edges incident to the vertex v and is as follows.

$$\sum_{e \in E_v} \frac{\mathrm{d}u}{\mathrm{d}x^e}(v) = 0 \quad \forall v \in V \tag{2.1}$$

Remark. The condition 2.1 corresponds to assuming Neumann conditions at all vertices. This condition also expresses the conservation of currents if the metric graph G is viewed as an electrical network, hence the name.

Definition 2.3.5. We define the one-dimensional **Schrödinger-type** operator as

$$\mathcal{H} : u(x) \longmapsto -\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + w(x)u(x) \tag{2.2}$$

The function $w(x)$ is a **potential** and we assume that $w \in L^\infty(G)$.

Definition 2.3.6. A **quantum graph** is a metric graph equipped with $\mathcal{H}$ defined by the operator 2.2 subject to the conditions 2.1 at the vertices.

Chapter 3

Problems defined on graphs

3.1 Graph Laplacian system

Laplacian graph matrices are symmetric, have zero sum of the rows, and have non-positive off-diagonal entries. Any matrix that satisfies these criteria is called a Laplacian matrix since it will always correspond to the Laplacian of a certain graph. The Laplacian matrix occurs in many applications where it is required for the solution of linear equations of the form

$$L\mathbf{x} = \mathbf{b} \tag{3.1}$$

where L is the Laplacian matrix of a connected undirected graph.

Remark. If we denote by $\mathbf{1}$ the constant vector with unit entries, we can write $L\mathbf{1} = L^T\mathbf{1} = \mathbf{0}$ since the sum of each row and each column of L is 0. Moreover, in the system defined by `Equation` 3.1, we have $\mathbf{1}^T\mathbf{b} = 0$, which means that the components of the vector $\mathbf{b}$ must sum up to zero.

We now recall a fundamental result of basic linear algebra.

Proposition 3.1.1. Consider the linear system $A\mathbf{x} = \mathbf{b}$ where A is a singular matrix. Then the system has either no solution or infinitely many solutions. Moreover, since A is singular, there exists a nonzero vector $\mathbf{y}$ for which $A\mathbf{y} = \mathbf{0}$. If $A\mathbf{x} = \mathbf{b}$ has a solution $\mathbf{x}$, then $\mathbf{x} + \alpha\mathbf{y}$ is also a solution for any $\alpha \in \mathbb{R}$.

As a consequence of the previous remark and proposition, we can state the following result.

Proposition 3.1.2. Consider the system defined by `Equation` 3.1. If $\mathbf{x}$ is a solution, then $\mathbf{x} + \alpha\mathbf{1}$ is also a solution for any $\alpha \in \mathbb{R}$.

Computing a solution for the graph Laplacian system may not be so easy, for what we have just presented. For a complete illustration of how to solve linear equations in Laplacian matrices, we refer to [14]. In general, computing such a solution may be very time consuming, especially with very big graphs that, however, appear in real applications. For this reason, we propose an alternative using Graph Neural Network which greatly reduces the time needed to compute the solution, as we will see in `Experiment` 5.4.

3.2 Elliptic PDEs

We consider a quantum graph with elliptic Hamiltonians, as we exposed in `Section` 2.3. We recall that the Hamiltonian acts on functions defined on a metric graph $G = (V, E)$, i.e.

$$\mathcal{H}u(x) = -\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + w(x)u(x)$$

where $w \in L^\infty(G)$ and $f : G \to \mathbb{R}$ are given functions.
With this in mind, we can define the following differential problem

$$\begin{cases} \mathcal{H}u = f & \text{in } G \\ \sum_{e \in E_v} \frac{\mathrm{d}u}{\mathrm{d}x^e}(v) = 0 & \forall v \in V \end{cases} \tag{3.2}$$

that is an elliptic partial differential equation with Neumann condition at every vertex of the graph. The solution to such problems is generally not analytical. The standard way to solve this is to use a numerical method, such as the Finite Element Method. In this work, we implemented FEM for PDE defined on graphs, but here we do not enter the theory details. For a complete exposition of finite elements for quantum graphs, we suggest reading the work of Arioli and Benzi [1].

3.2.1 Weak formulation

To solve a PDE with FEM is necessary to compute the weak formulation of the `Problem` 3.2, which requires $u \in H^1(G)$. We multiply the differential equation by a test function $v \in H^1(G)$ and we proceed by adding the integration by part into the graph G. If we consider the bilinear form

$$\mathfrak{h} : H^1(G) \times H^1(G) \to \mathbb{R}$$

then, we can write the weak formulation

$$\mathfrak{h}(u, v) = \int_G f(x)v(x)\mathrm{d}x \quad \forall v \in H^1(G)$$

where

$$\mathfrak{h}(u, v) = \sum_{e \in E} \left\{ \int_e \frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}x}\mathrm{d}x + \int_e u(x)v(x)w(x)\mathrm{d}x \right\}$$

Given these definitions, one can prove the following result that follows from the Lax-Milgram theorem.

Proposition 3.2.1. Let $f \in L^2(G)$ and suppose that there exists $w_0 \in \mathbb{R}$ such that $w(x) > w_0$ $\forall x \in G$. Then, `Problem` 3.2 has a unique weak solution $u \in H^1(G)$. Moreover, the following stability estimate holds:

$$||u||_{H^1(G)} \leq \frac{1}{\min(1, w_0)}||f||_{L^2(G)}$$

3.2.2 Dirichlet condition

So far, we have defined the elliptic problem only for Neumann conditions at the vertices. In practice, Dirichlet conditions are also very common, so we decided to take them into account. For this reason, we can rewrite `Problem` 3.2 in a more generic way

$$\begin{cases} -\Delta u + wu = f & \text{in } G \\ u(v) = g & \forall v \in V_D \\ \sum_{e \in E_v} \frac{\mathrm{d}u}{\mathrm{d}x^e}(v) = 0 & \forall v \in V_N \end{cases} \tag{3.3}$$

where V_D are the vertices with Dirichlet conditions and V_N are the ones with Neumann conditions. Of course, we have $V = V_N \cup V_D$.

Chapter 4

Graph Neural Networks

GNNs, or **Graph Neural Networks**, are a specific kind of neural network architecture. In simple terms, a GNN is a computational unit that can take as input a set of node features, represented as $\mathbf{u} : V \to \mathbb{R}^q$, where V denotes the vertices of a directed graph $G = (V, E)$ and E represents the edges. The GNN processes this input and produces a corresponding set of output node features $\mathbf{u'} : V \to \mathbb{R}^{q'}$. Importantly, the same GNN unit can be used to process data from different graphs. However, there are two restrictions:

1. the size of the input and output features, q and q', respectively, must be defined.
2. Each input-output pair must be defined over the same graph.

In the following, we will describe the same architecture used by Franco et al.[5] adapted to solve our problems of interest.

Remark. In the literature, the graph $G = (V, E)$ that defines the neural network usually represents a mesh of a physical domain. Here, we use GNNs to exploit problems directly defined on a physical graph. In fact, in our study, G is treated both as a physical graph (`Experiments` 5.3 and 5.4) and as a mesh of a physical graph (`Experiment` 5.5).

4.1 Message passing block

Definition 4.1.1. Given a graph $G = (V, E)$ and $l \in \mathbb{N}$ hidden features, we define

- the vertex features as $\mathbf{v} : V \to \mathbb{R}^l$
- the edge features as $\mathbf{e} : E \to \mathbb{R}^l$

Then, the **message passing layer** $F = F(\mathbf{v}, \mathbf{e}, G)$ takes as input $\mathbf{v}$, $\mathbf{e}$ and G and outputs a new collection of vertex features $\mathbf{v'} : V \to \mathbb{R}^l$.

In GNN architectures, F is considered a transformer that modifies the characteristics of the vertices connected to the nodes of the graph. A message-passing block, within these architectures, refers to a specific type of graph-forwarding routine, which exploits the local structure of the input graph G. This routine allows communication exclusively between neighboring nodes. Before entering the details of the architecture, we give some definitions.

Definition 4.1.2. Given a graph $G = (V, E)$ and a collection of vertex features $\mathbf{v} : V \to \mathbb{R}^l$, we define the maps

$$\mathbf{v}_{in} : E \to \mathbb{R}^l \qquad \mathbf{v}_{out} : E \to \mathbb{R}^l$$

given by

$$\mathbf{v}_{in}(v_i, v_j) = \mathbf{v}(v_i) \qquad \mathbf{v}_{out}(v_i, v_j) = \mathbf{v}(v_j)$$

Hence, given a directed edge $(v_i, v_j) \in E$, $\mathbf{v}_{in}$ computes the vertex features of the corresponding source node, while $\mathbf{v}_{out}$ computes the vertex features of the corresponding destination node.

Definition 4.1.3. Given a graph $G = (V, E)$ and a collection of edge features $\mathbf{e} : E \to \mathbb{R}^l$, we define the map $\overline{\mathbf{e}} : V \to \mathbb{R}^l$ as

$$\overline{\mathbf{e}}(v) = \sum_{v_i \in V : (v_i, v) \in E} \mathbf{e}(v_i, v)$$

That is, given a vertex $v \in V$ we sum all edge features of the edges with the destination node v.

Definition 4.1.4. Given two function with the same domain $\mathbf{f} : X \to \mathbb{R}^a$ and $\mathbf{g} : X \to \mathbb{R}^b$, the concatenation operator $\oplus$ is defined as

$$(\mathbf{f} \oplus \mathbf{g})(x) = [f_1(x), ..., f_a(x), g_1(x), ..., g_b(x)]$$

where $x \in X$ is a generic input.

Lastly, consider two Multi-Layer Perceptron (MLP) units

$$\Psi_v : \mathbb{R}^{2l} \to \mathbb{R}^l \qquad \Psi_e : \mathbb{R}^{3l} \to \mathbb{R}^l$$

then, the action of a message passing layer F can be defined as

$$F(\mathbf{v}, \mathbf{e}, G) = \Psi_v\Big(\mathbf{v} \oplus \overline{\Psi_e(\mathbf{e} \oplus \mathbf{v}_{in} \oplus \mathbf{v}_{out})}\Big) \tag{4.1}$$

In simpler terms, `Equation` 4.1 explains how the features of the vertices in the output graph, $F(\mathbf{v}, \mathbf{e}, G)$, are obtained. First, the information from the graph vertices is transferred to the edges and combined with the existing features, $\mathbf{e} \oplus \mathbf{v}_{in} \oplus \mathbf{v}_{out}$. Then, an MLP, Ψ_e, is used to extract meaningful information from the extended features. This extracted information is then transferred back to the node vertices, resulting in $\overline{\Psi_e(\mathbf{e} \oplus \mathbf{v}_{in} \oplus \mathbf{v}_{out})}$. These hidden features, which now exist within the graph vertices, are added to the original features. Finally, the combined features are passed through a terminal MLP block, denoted as Ψ_v.

Remark. In general the action of $F(\mathbf{v}, \mathbf{e}, G)$ is nonlinear, due to the presence of the MLPs Ψ_v and Ψ_e. We recall that the non-linearity of a multilayer perceptron is caused by the nonlinear activation function used.

For a graphical representation of the message passing step, look at the `Figure` 4.1.

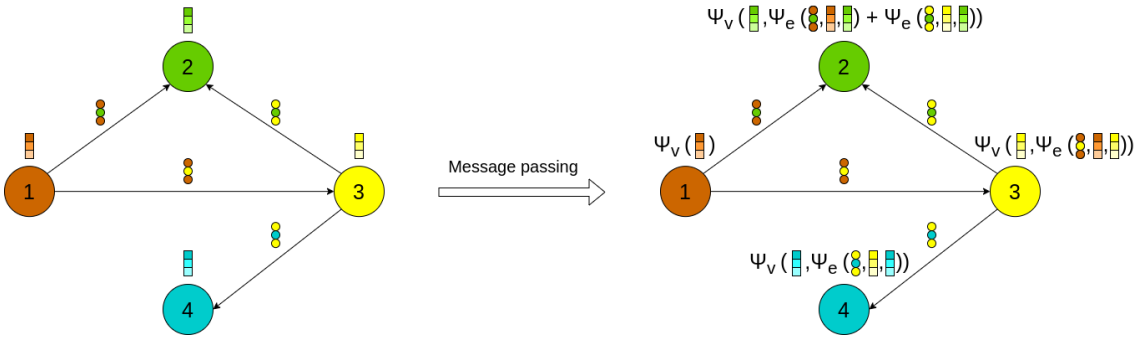


Figure 4.1: Message propagation and aggregation. For each node, at each stage of the message passage, information is collected from neighboring nodes and combined using the two MLPs. Notice that the edge features are denoted by the circles and we can see that they remain unchanged after the message passing step. We used the comma to indicate the concatenation operator.

Remark. The aforementioned architecture can be adopted in case the edge features are not relevant to the problem of interest, as we will show in `Experiments` 5.3 and 5.4. In these cases the `Equation` 4.1 can be rewritten as

$$F(\mathbf{v}, G) = \Psi_v\Big(\mathbf{v} \oplus \overline{\Psi_{e^*}(\mathbf{v}_{in} \oplus \mathbf{v}_{out})}\Big) \tag{4.2}$$

with the inner MLP that changes domain, that is $\Psi_{e^*} : \mathbb{R}^{2l} \to \mathbb{R}^l$.

4.2 Encoder - Processor - Decoder

As the name suggests, the Encoder-Processor-Decoder model consists of three modules, which we explain in detail below. All of these modules have a common hidden dimension, l, which we assume to be fixed throughout the discussion.

4.2.1 Encoder module

In general, the following features are defined on a generic graph $G = (V, E)$:

- input signal defined over the vertices $\mathbf{u} : V \to \mathbb{R}^q$ with $q \neq l$,
- input signal defined over the edges $\mathbf{e}^* : E \to \mathbb{R}^s$ with $s \neq l$,
- output signal defined over the vertices $\mathbf{u}' : V \to \mathbb{R}^{q'}$ with $q' \neq l$.

Due to the dimension of mismatch, it is important to transform the input signals to match the dimension of hidden features l. The simplest way to do that is to use multilayer perceptron networks

$$\mathscr{E}_v(\mathbf{u}) = \Psi^v_{\mathscr{E}}(\mathbf{u}) \qquad \mathscr{E}_e(\mathbf{e}^*) = \Psi^e_{\mathscr{E}}(\mathbf{e}^*)$$

While the input signal $\mathbf{u}$ is very problem dependent, usually the edge features $\mathbf{e}^*$ are chosen in the following way. Suppose that $V \subset \mathbb{R}^d$, then $\mathbf{e}^* : E \to \mathbb{R}^{d+1}$ (i.e. $s = d + 1$) and they are computed as

$$\mathbf{e}^*((i,j)) = \Big[\frac{x_i^{(1)} + x_j^{(1)}}{2}, ..., \frac{x_i^{(d)} + x_j^{(d)}}{2}, |\mathbf{x_i} - \mathbf{x_j}|\Big] \tag{4.3}$$

where $(i, j) \in E$. Vectors $\mathbf{x_i} \in V$ and $\mathbf{x_j} \in V$ are the corresponding space coordinates. In plain words, these edge features are made up of the midpoints of each space coordinate and the edge length.

4.2.2 Processor module

The processor module is composed of m message passing steps $F_1, ..., F_m$ each defined as `Equation` 4.1. Here, the encoded features are used, that is, $\mathbf{v} = \mathscr{E}_v(\mathbf{u})$ and $\mathbf{e} = \mathscr{E}_e(\mathbf{e}^*)$. The output of the processor module is given by

$$\mathscr{P}(\mathbf{v}, \mathbf{e}, G) = F_m(\mathbf{h_m}, \mathbf{e}, G)$$

where

$$\begin{cases} \mathbf{h_1} = \mathbf{v} \\ \mathbf{h_{j+1}} = F_j(\mathbf{h_j}, \mathbf{e}, G) \quad j = 1, ..., m \end{cases}$$

hence the final result is obtained by iteratively applying the m message passing layer. We highlight that m is a hyper-parameter of the network and the choice is very important to obtain a good result. In particular, a processor module with m units enables communication between nodes that are m edges away. Thus, increasing m we move from local to nonlocal transformations.

Remark. The processor module updates the node features at each message passing step, while the edge features are kept constant throughout the process.

4.2.3 Decoder module

The processor module gives as output a node features $\mathbf{v}' : V \to \mathbb{R}^l$ with $\mathbf{v}' = \mathscr{P}(\mathbf{v}, \mathbf{e}, G)$. Since we have to match the real output signal $\mathbf{u}' : V \to \mathbb{R}^{q'}$ we can define and use an MLP in the following way

$$\mathscr{D}(\mathbf{v}') = \Psi_{\mathscr{D}}(\mathbf{v}')$$

and, of course, the final outcome will be $\mathbf{u}' = \mathscr{D}(\mathbf{v}')$.

Remark. For a simple problem, one can choose to avoid the use of the encoder and the decoder, in this case, we would have $q = l = q'$. The network will work the same as we will show in `Experiment` 5.3.

4.2.4 Complete architecture

Overall, the complete network can be expressed as

$$\Phi(\mathbf{u}, \mathbf{e}^*, G) = \mathscr{D}(\mathscr{P}(\mathscr{E}_v(\mathbf{u}), \mathscr{E}_e(\mathbf{e}^*), G))$$

moreover, for a graphical representation, one can refer to the `Figure` 4.2.

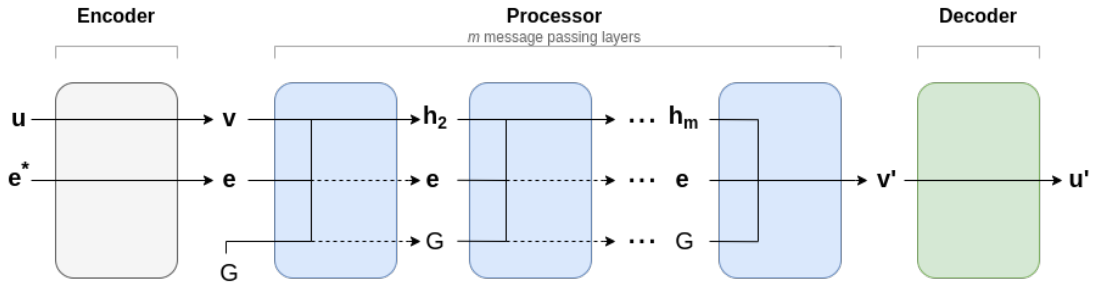


Figure 4.2: Visual representation of the Encoder-Processor-Decoder model. Rigid arrows represent network computations, while dashed arrows act like pointers (no computation implied).

Remark. In the field of surrogate and reduced-order modeling, the terms "Encoder" and "Decoder" are commonly used in the context of dimensionality reduction. Typically, the Encoder is responsible for compressing the data, while the Decoder deals with its reconstruction. However, in this particular case, the meaning is completely different. In this specific context, the Encoder module functions as a feature extractor and has the potential to increase the dimensionality of the input. On the other hand, the Decoder module is used to map the features extracted from each node onto the nodal output space. This mapping typically involves reducing the dimensionality at each node, resulting in the desired quantities being recovered. Although this notation may confuse some readers, we have chosen to adhere to the terminology adopted by the Graph Neural Network community.

4.3 Training procedure

4.3.1 Dataset

The data set for a GNN typically consists of a set of graphs, where each graph is associated with a set of labeled nodes. In our case, the graphs are generated randomly using the procedure we will present in `Experiment` 5.1. The labels we used are the input and output signals; we will explain how we compute these values in detail in the numerical experiments. The dataset can be further partitioned into training, validation, and test sets, which are used to train, modify, and evaluate the model, respectively.

4.3.2 Loss function

From now on, we assume that the output signal $\mathbf{u}'$ of the whole network is a scalar function, that is, $u' : V \to \mathbb{R}$ ($q' = 1$). With this assumption, we denote by the letter $\mathbf{y}$ the output signal vector and by $\hat{\mathbf{y}}$ the network output vector. The choice of the loss function is one of the key aspects that define a machine learning algorithm; in this case, we decided to use the classic mean square error defined as

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{n_i} \sum_{j=1}^{n_i} \left(y_i^{(j)} - \hat{y}_i^{(j)}(\theta)\right)^2 \tag{4.4}$$

where N is the number of graphs in the dataset, $\mathbf{y_i} \in \mathbb{R}^{n_i}$ is the real output signal of the n_i nodes of the i-th graph and $\hat{\mathbf{y}}_\mathbf{i} \in \mathbb{R}^{n_i}$ is the network output of the i-th graph. We denote all the learnable parameters of the network with θ.

Remark. Notice that the double summation in `Formula` 4.4 is necessary because, in principle, the number of nodes can vary from one graph to another. If $n_i = n \ \ \forall i \in N$ then one can also write

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left|\left|\mathbf{y_i} - \hat{\mathbf{y}}_\mathbf{i}(\theta)\right|\right|_2^2$$

4.3.3 Training algorithm

We implemented a standard procedure to train our network. In particular, we used Adam optimizer with an initial learning rate of 0.001. The learning rate is automatically reduced on the plateau by a factor of 10. This means that after 20 epochs where the validation loss does not decrease, we reduce the learning rate. Finally, we used early stopping to avoid overfitting: if the validation loss did not decrease for a given number of epochs (patience), we stopped the training procedure. After the train loop, we keep the model with the lowest validation loss. A simplified pseudocode of what we have implemented can be found in `Algorithm` 1.

Algorithm 1 Training procedure

Require: model ▷ GNN model to train
Require: training ▷ Training set
Require: validation ▷ Validation set
Require: loss_function ▷ Loss function to use
Require: optimizer ▷ Optimizer used to minimize the loss
Require: epochs ≥ 1 ▷ Number of epochs

```
function TRAIN(model, training, validation, loss_function, optimizer, epochs)
    best_val_loss ← ∞                                   ▷ Best validation loss
    while epoch ≤ epochs do
        ▷ Train one epoch
        for data in training do
            outputs ← model(data)                       ▷ Evaluate model
            loss ← loss_function(outputs, data)         ▷ Compute loss function
            loss.backward()                             ▷ Compute loss gradient
            optimizer.step()                            ▷ Adjust learning weights
        end for
        ▷ Evaluate on validation set
        val_loss ← 0
        for data in validation do
            outputs ← model(data)                       ▷ Evaluate model
            loss ← loss_function(outputs, data)         ▷ Compute loss function
            val_loss ← val_loss + loss
        end for
        ▷ Track best performance
        if val_loss < best_val_loss then
            best_val_loss ← val_loss
            model.save_best()                           ▷ Save the model state
        end if
        ▷ Reduce learning rate
        if learning_rate(val_loss) then
            optimizer.reduce_learning_rate()
        end if
        ▷ Check early stopping
        if early_stopping(val_loss) then
            break
        end if
        epoch ← epoch + 1
    end while
end function
```

Chapter 5

Numerical experiments

Before starting the illustration of the results, we point out that all the simulations were done on a portable computer with the following specification

- Processor: `Intel® Core™ i5-8250U CPU @ 1.60GHz × 8`
- Ram: `8GB DDR4`
- Operating system: `Ubuntu 22.04.3 LTS`

Therefore, all times and performances presented must be considered as an indication of the goodness of the implementation, but not as a "truth time". In fact, the time may vary for different code executions, as the operating system performs several tasks together.

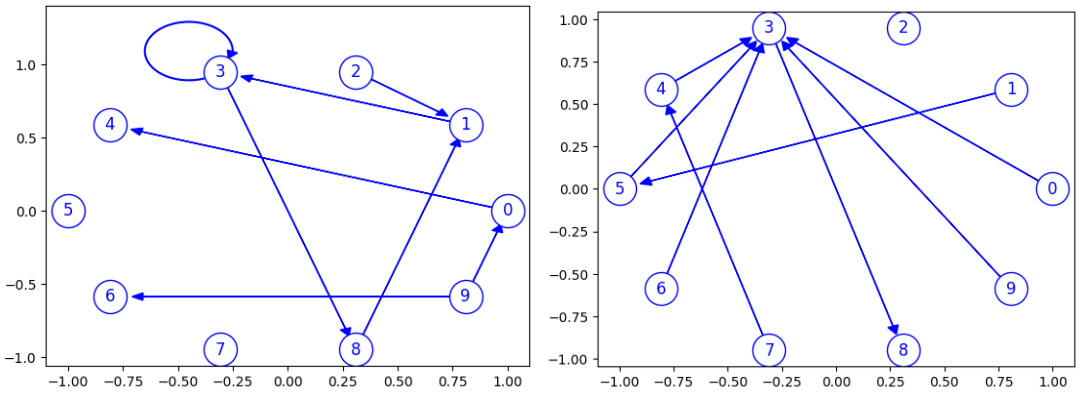
5.1 Graphs generation

Since Neural Networks are very "data hungry", it is important to have different graphs to train correctly our networks. For this reason, we implemented a code that can randomly generate a graph given certain properties. In this section, we present all the possible graphs that can be generated, referring to the definitions we introduced in `Paragraph` 2.1. In particular, the following properties can be specified.

- Number of vertices of the graph.
- Dimension of the space (for dimensions 2 and 3, the graph can also be plotted).
- Limits of the bounding box that contains the coordinates of the vertices.
- Minimum number of edges of the graph (this number is ignored if it is incompatible with the other option set).
- Directed/undirected graph.
- Possibility of having self-loop (only for directed graph).
- Strong connection (only for directed graph).
- The weak connection, in cases of undirected graphs, coincides with the connection property.
- The acyclicity to force the graph to have no cycles.

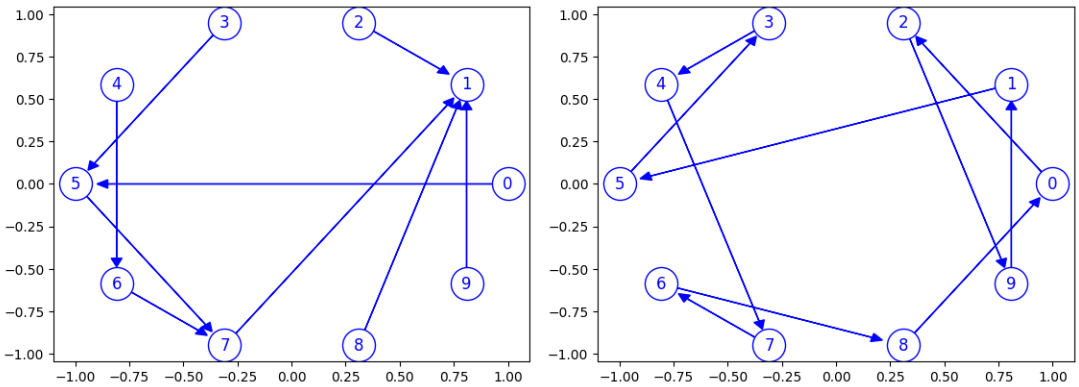
5.1.1 Directed graphs

Our code can randomly generate both the coordinates of the vertices and the edges. In the following, we fix the vertices' coordinates for a better visualization of the different graphs.
In `Figure` 5.1, we can see a generic directed graph where no particular properties were set and an acyclic directed graph. In the last case, the self-loops are automatically avoided.
In `Figure` 5.2, we can see connected graphs. For a weak connection, the graph must have at least $N-1$ edges, where N is the number of vertices. Moreover, we can notice that the strong connection implies that the graph cannot be acyclic.



(a) Generic directed graph (b) Acyclic directed graph

Figure 5.1: Examples of directed graphs - 1



(a) Weakly connected graph (b) Strongly connected graph

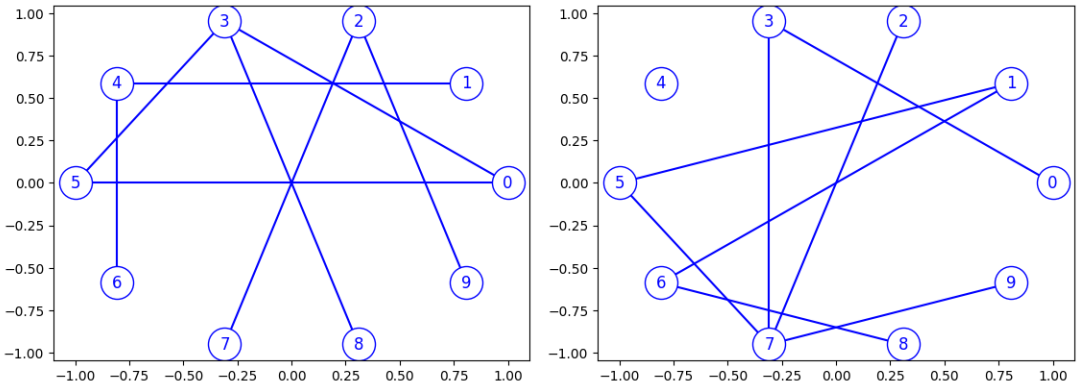
Figure 5.2: Examples of directed graphs - 2

5.1.2 Undirected graphs

From a mathematical point of view, an undirected graph is such that

$$(i, j) \in E \implies (j, i) \in E \tag{5.1}$$

However, in our implementation, we decided to save only one edge to reduce memory consumption and avoid duplicate procedures in some cases.



(a) Generic undirected graph (b) Acyclic undirected graph

Figure 5.3: Examples of undirected graphs

In `Figure` 5.3, there is the plot of a generic undirected graph and an acyclic undirected graph. From now on, we consider as default an **undirected, connected and acyclic** graph (an example in `Figure` 5.4). This means that all the GNNs we present in this work were trained on such graphs.

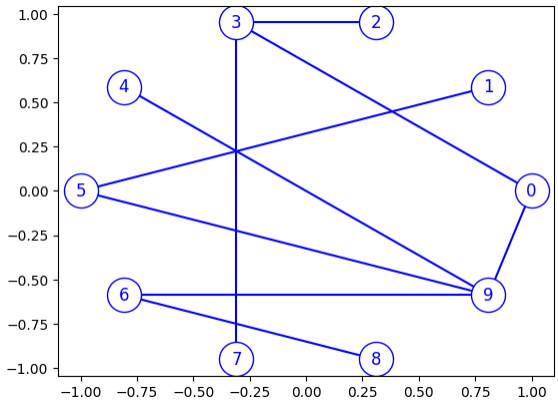
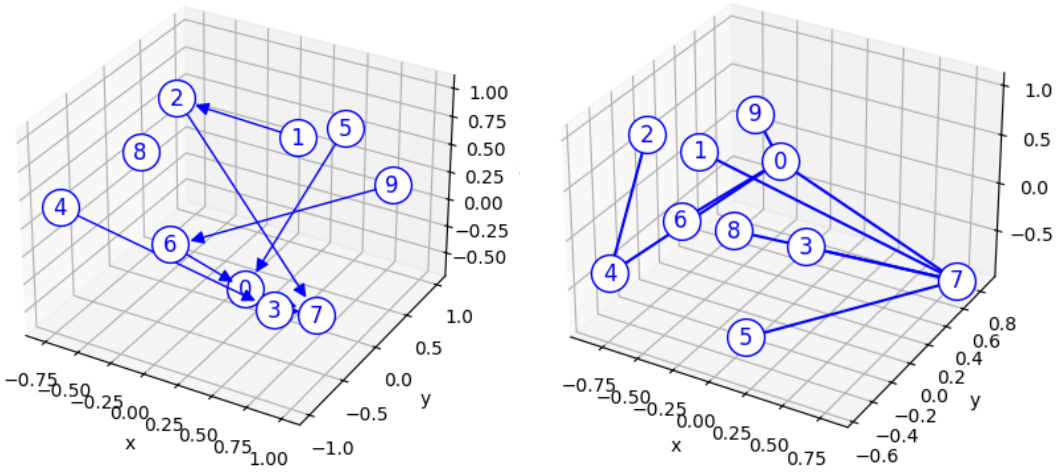


Figure 5.4: Default graph: undirected, connected, and acyclic

5.1.3 3D space

As we have mentioned before, we can generate a graph in N-dimensional space simply by changing the dimension of the coordinates of the vertices. For what concerns the generation of edges, nothing changes between different dimensions. In the case of 3D space, a plot method is also provided, and we can see an example in `Figure` 5.5 (notice that in these cases the coordinates of the vertices were also randomly generated).



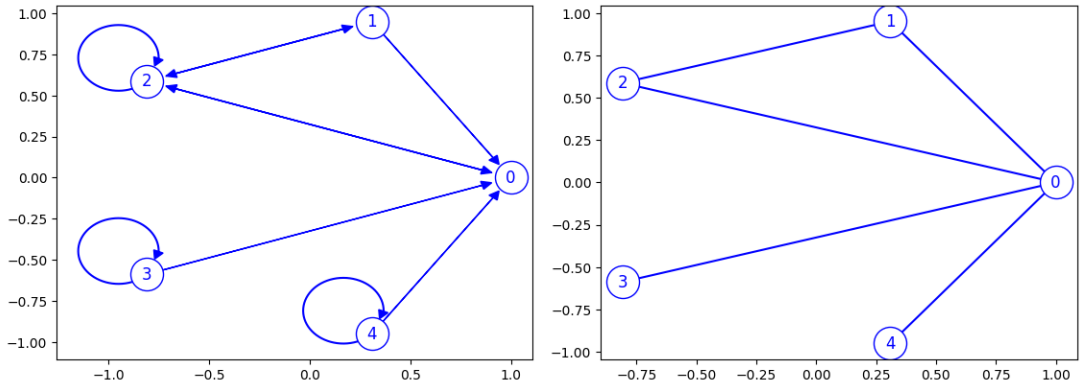
(a) DAG (directed acyclic graph) in 3D

(b) Default graph in 3D

Figure 5.5: Example of graphs in 3D space

5.1.4 Algorithms for computing the properties of graphs

Another interesting feature we have developed is the possibility of letting the code understand the type of graph, given the matrix of the vertex coordinates and the edge matrix. Here, we remark that the edge matrix used to infer the properties of the graph must satisfy `Property` 5.1 in the case of undirected graphs.



(a) Directed case

(b) Undirected case

Figure 5.6: Graphs used to infer properties

Consider the graphs in `Figure` 5.6, our algorithm has computed the following properties

	Graph (a)	**Graph (b)**
Number of vertices	5	5
Dimension of the space	2	2
Directed graph	Yes	No
Self loops	Yes	No
Strong connection	No	No
Weak connection	Yes	Yes
Acyclic graph	No	No
Number of cycles	6	1

5.1.5 Computation time

Finally, we focused our attention on the creation time needed for our implementation. We considered a generic directed graph, a DAG, and an undirected, connected, and acyclic graph (the default one). We tested the creation time for an increasing sequence of vertices: 10, 50, 100, 500, 1000, 5000, 10000, 20000, 50000, 100000, 200000, 500000, and 1000000.

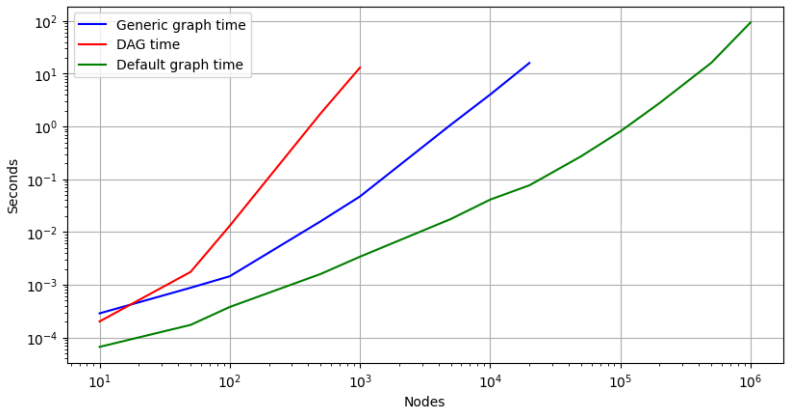


Figure 5.7: Comparison of creation time for different types of graph

In `Figure` 5.7, we can see the results, where we stopped generating the graph when the computation time became too high. We optimized the code to be able to quickly create a default graph, even with a very high number of vertices. For the other two cases, the bottlenecks were the acyclicity to be maintained for the DAG and the random generation of edges for the generic graph (in particular, checking whether an edge is already present in the graph). Overall, we are satisfied with this result because for what concerns the GNNs, we worked only with undirected, connected, and acyclic

graphs. In a future version of the code, a possible upgrade could be the generation time for all types of graphs.

5.2 FEM on graphs

After the simulation of graph generation, we proceed to test the accuracy of the FEM implementation. To do so, we consider three different elliptic problems, all defined in a star-shaped domain as in `Figure` 5.8.

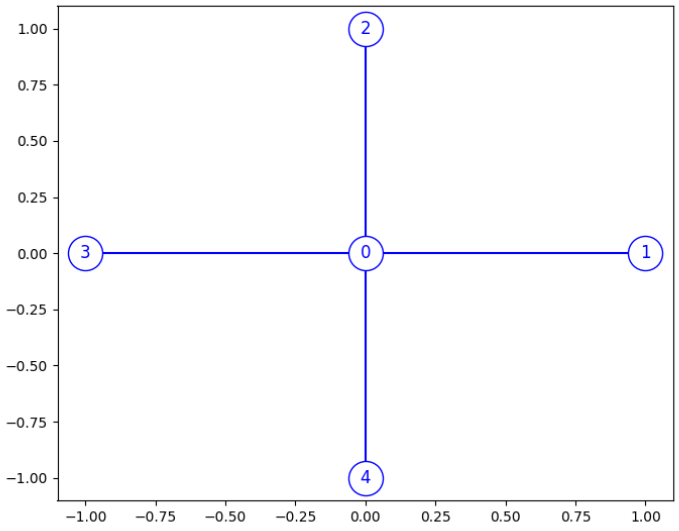


Figure 5.8: Graph domain with 5 vertices and 4 edges.

Remark. The graph $G = (V, E)$ of the domain we considered for these three simulations has the following vertices and edges

$$V = \{0, 1, 2, 3, 4\}, \qquad E = \{(0, 1), (0, 2), (0, 3), (0, 4)\},$$

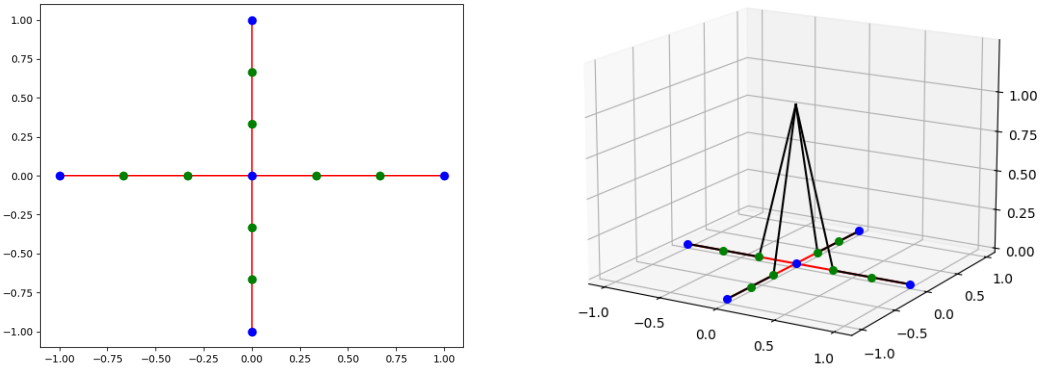
with vertices coordinates

$$\mathbf{x} = \begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \\ -1.0 \\ 0.0 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ -1.0 \end{bmatrix}.$$

To solve the problems with FEM, it's necessary to create a mesh from the physical graph. Starting from this graph, we added n_{ref} refinement nodes per each edge. Moreover, since the edges are straight lines, it is possible to use classical one-dimensional finite elements along the edges. Thus, standard hat basic functions are built for each discretization node: fixed an edge e, we construct n_{ref} hat basis function ψ_j^e with support $[x_{j-1}^e, x_{j+1}^e]$ defined as

$$\psi_j^e(x) = \begin{cases} 1 - \frac{|x_j^e - x|}{h_e} & \text{if } x \in [x_{j-1}^e, x_{j+1}^e] \\ 0 & \text{otherwise} \end{cases}$$

where $h_e = \frac{\ell_e}{n_{ref}+1}$ since we used equispaced refinements (remember that ℓ_e is the edge length). For the original vertices, a similar approach is used, keeping in mind that a vertex can have an arbitrary number of edges leaving or reaching it. A very useful example with the star-shaped domain and a refinement of two nodes per edge can be found in `Figure` 5.9.
Finally, as an index of goodness of FEM implementation, we decided to use the infinity norm error defined as $e_\infty = ||u - \hat{u}||_\infty$ where u is the exact solution and $\hat{u}$ is the FEM solution.



(a) (b)

Figure 5.9: **Plot (a)**: Mesh of the star-shaped domain with two refinements per edge. The colored dots represent the nodes of the mesh. Those in blue are the physical vertices of the graph, the green ones represent the refinement nodes.
Plot (b): Behaviour of the hat function for the node with coordinates $(0, 0)$.

5.2.1 Constant function

First, consider the following problem

$$\begin{cases} -\Delta u + (x^2 + y^2 + 1)u = 5(x^2 + y^2 + 1) & \text{in } G \\ \sum_{e \in E_v} \frac{\mathrm{d}u}{\mathrm{d}x^e}(v) = 0 & \forall v \in V \end{cases} \tag{5.2}$$

that has the constant solution $u(x, y) = 5$. We use a mesh with 10 nodes per edge, and we expected the fem solution to be exact since we are dealing with constant function.

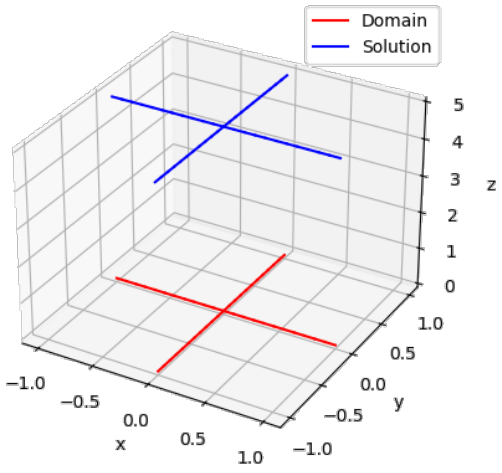


Figure 5.10: Computed solution of `Problem` 5.2

The plot of the solution can be found in `Figure` 5.10, and the infinity norm error is $e_\infty = 1.69 \times 10^{-14}$.

5.2.2 Linear function

The second problem we tested was the mixed Dirichlet-Neumann-Kirchhoff problem with the following formulation

$$\begin{cases} -\Delta u + (x^2+y^2+1)u = (x+y)(x^2+y^2+1) & \text{in } G \\ \sum_{e \in E_{v_0}} \frac{\mathrm{d}u}{\mathrm{d}x^e}(v_0) = 0 \\ u(v_1) = 1 \\ u(v_2) = 1 \\ u(v_3) = -1 \\ u(v_4) = -1 \end{cases} \tag{5.3}$$

that has the solution $u(x, y) = x + y$. For this problem, we also use a mesh with 10 nodes per edge. We anticipated the FEM solution to be precise as well since we have a linear solution.

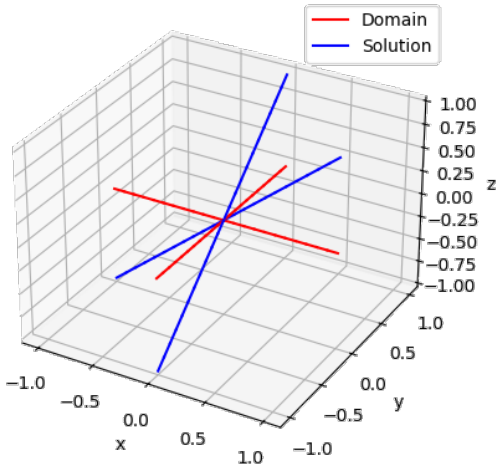


Figure 5.11: Computed solution of `Problem` 5.3

The plot of the solution can be found in `Figure` 5.11 and the infinity norm error is $e_\infty = 7.22 \times 10^{-16}$.

5.2.3 Function of a single variable

The last test we did concerns a solution that is a function only of x (i.e. $u(x, y) = u(x)$) and it is no longer linear. The definition of the problem is

$$\begin{cases} -\Delta u + u = (4\pi^2+1)sin(2\pi x) & \text{in } G \\ \sum_{e \in E_{v_0}} \frac{\mathrm{d}u}{\mathrm{d}x^e}(v_0) = 0 \\ u(v) = 0 & \forall v \neq v_0 \end{cases} \tag{5.4}$$

which has solution $u(x, y) = sin(2\pi x)$. Here, we use a mesh with 50 nodes per edge. We report the plot of the calculated solution in `Figure` 5.12. In this case, the infinity norm error is $e_\infty = 3.10 \times 10^{-5}$.

Remark. One may have noticed that the infinity norm error for the first two problems is essentially zero while the error of `Problem` 5.4 is higher. This is probably because the third pde has a non-linear solution, and we did not use so many internal points for the mesh. However, for the learning procedure we will do for the Graph Neural Network, we will consider the output of the FEM as the correct one. This means that the GNN error will be computed on the basis of the FEM solution.

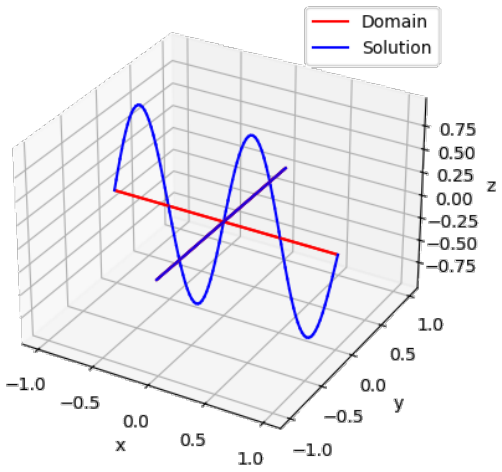


Figure 5.12: Computed solution of `Problem` 5.4

5.3 Discrete Laplacian with GNN

5.3.1 Problem

We started testing the implementation of our version of Graph Neural Network with a very easy task. Consider the graph Laplacian system

$$L\mathbf{x} = \mathbf{b} \tag{5.5}$$

where L is the Laplacian matrix of an **undirected, connected and acyclic** graph $G = (V, E)$. We aim to compute the vector $\mathbf{b}$ given the vector $\mathbf{x}$ and the graph G. We recall that one can avoid the matrix-vector computation using the definition of discrete Laplacian

$$(L\mathbf{x})^{(v)} = b^{(v)} = \sum_{w \in A_v} x^{(v)} - x^{(w)} \qquad \forall v \in V \tag{5.6}$$

where A_v denotes the set of vertices adjacent to v (i.e. directly connected to v through an edge).

5.3.2 Architecture of the network

Following our notation, we decided to use:

- A dataset of $N = 1000$ graphs randomly generated with number of vertices $n \sim \mathcal{U}(100, 200)$. The training-validation-testing division of the data set used was 70-20-10.
- The vector $\mathbf{x}$ as input signal, randomly generated with $\mathbf{x} \sim \mathcal{U}(0, 1)$. Each graph in the data set is associated with a different vector $\mathbf{x}$.
- The vector $\mathbf{b}$ as the output signal, computed using `Formula` 5.6.
- No edge features since there is no dependence on spatial information.
- No Encoder and no Decoder. Thus, the dimension of the hidden features is the same as that of the input and output signals.
- Only one message passing step ($m = 1$) because from `Formula` 5.6 we can notice that the result at certain nodes depends only on the directly connected vertices.

The network has the following summary output.

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
|-Encoder: 1-1                           [-1, 1]                   --
|    |-Sequential: 2-1                   [-1, 1]                   --
|    |-Sequential: 2-2                   [-1]                      --
|-ModuleList: 1                          []                        --
|    |-MessagePassingBlock: 2-3          [-1, 1]                   --
|    |    |-Sequential: 3-1              [-1, 1]                   3
|    |    |-SumAggregation: 3-2          [-1, 1]                   --
|    |    |-Sequential: 3-3              [-1, 1]                   3
|-Decoder: 1-2                           [-1, 1]                   --
|    |-Sequential: 2-4                   [-1, 1]                   --
==========================================================================
Total params: 6
Trainable params: 6
Non-trainable params: 0
```

We can notice that the only parameters of the network are the MLPs present in the message-passing layer. Even the two inner networks are very straightforward: we used only one layer for each network. Overall, we can say that this is the simplest network that can be built with our architecture.

5.3.3 Results

The network reaches train and validation values losses in the order of magnitude of 10^{-32} after less than 100 epochs. Also in the test set, we have the same values of the loss function. Even if the problem was very easy, we are satisfied with the results, because in practice the network can compute the discrete Laplacian with no error. To better understand this claim, let us introduce the relative error:

$$e_{rel}^{(i)} = \frac{|y^{(i)} - \hat{y}^{(i)}|}{y^{(i)}} \qquad \forall i \in V$$

where $y^{(i)}$ is the real output value at node i and $\hat{y}^{(i)}$ is the network output at node i. Then, we can define the **mean relative error** as

$$E_{rel} = \frac{1}{|V|} \sum_{i \in V} e_{rel}^{(i)} \tag{5.7}$$

Even the mean relative error just defined was less or equal to 10^{-15} for all the graphs in the testing set.
Afterward, we wanted to stress our network by progressively increasing the number of vertices in the graph and computing the discrete Laplacian with a random vector of the same dimension. We then compared the time needed by the three methods to compute the discrete Laplacian, i.e.:

1. using matrix-vector multiplication as in `Formula` 5.5,
2. using the definition of discrete Laplacian as in `Formula` 5.6,
3. using the Graph Neural Network.

We used the following numbers of nodes: 10, 50, 100, 500, 1000, 5000, 10000, 20000, 50000, 100000, 200000, 500000 and 1000000. The matrix-vector multiplication method has only been performed up to 20000 nodes because after that the calculation time and the size of the matrix L will become infeasible. We can see the needed time in `Figure` 5.13.
As might be expected, for a small number of nodes matrix-vector multiplication is the best method, but its time increases rapidly and soon becomes unmanageable. The most appealing result is that the GNN inference time is between 10 and 100 times lower than the definition time for very large graphs. This result is even more astonishing if we look at the mean relative error shown in `Figure` 5.14.

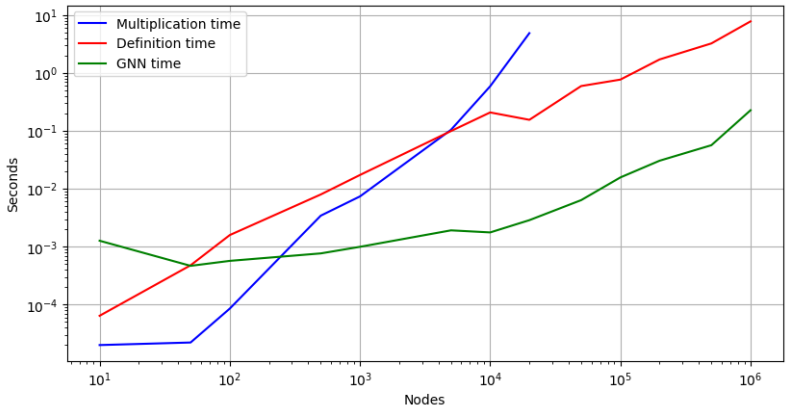


Figure 5.13: Comparison of time needed

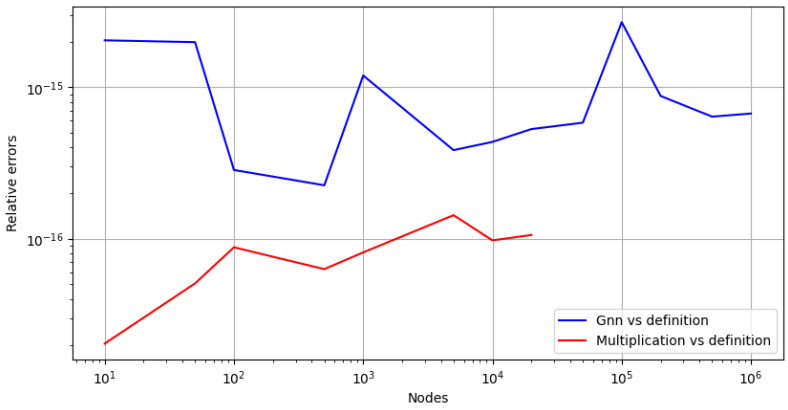


Figure 5.14: Comparison of errors

We can see that the mean relative errors of the GNN are not much bigger than the relative errors between the definition and matrix-vector multiplication (that are the same way from a mathematical point of view). Moreover, we notice that in practice the relative errors of the GNN do not increase in the number of nodes.

Remark. These last two results denote an outstanding generalization property of the GNN because we stress that the network was trained with graphs with 100 to 200 vertices.

5.4 Graph Laplacian system with GNN

5.4.1 Problem

We now consider the problem of solving the graph Laplacian system, that is

$$L\mathbf{x} = \mathbf{b}$$

where L is the Laplacian matrix of an **undirected, connected and acyclic** graph $G = (V, E)$ with n vertices. As we have already mentioned in `Section` 3.1, the L matrix is not invertible, making the solution of the system hard to compute. To compare the performance of the GNN solution, we use a standard optimization technique to solve the system which is

$$\mathbf{x} = \arg\min_{\hat{\mathbf{x}} \in \mathbb{R}^n} ||L\hat{\mathbf{x}} - \mathbf{b}||_2^2 \tag{5.8}$$

In cases of multiple solutions, $\mathbf{x}$ with minimum l^2-norm is considered.

5.4.2 Architecture of the network

For this problem, the network has the following specifications:

- A dataset of $N = 500$ graphs randomly generated with number of vertices $n \sim \mathcal{U}(100, 200)$. The training-validation-testing division of the dataset used was 70-20-10.
- The vector $\mathbf{x}$ as the output signal, randomly generated with $\mathbf{x} \sim \mathcal{U}(1, 2)$. Each graph in the dataset is associated with a different vector $\mathbf{x}$.
- The vector $\mathbf{b}$ as input signal, computed using `Formula` 5.6.
- No edge features since there is no dependence on spatial information.
- Encoder that increases the size of input features to match the size of hidden features. We chose to use 5 hidden features. At the end of the Processor, we used a Decoder to go back to a vector in one-dimensional space.
- Ten message passing steps ($m = 10$). This value was tuned by hand by us.

The network has the following (truncated) summary output.

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
|-Encoder: 1-1                           [-1, 5]                   --
|    |-Sequential: 2-1                   [-1, 5]                   --
|    |    |-Linear: 3-1                  [-1, 5]                   10
|    |-Sequential: 2-2                   [-1]                      --
|-ModuleList: 1                          []                        --
|    |-MessagePassingBlock: 2-3          [-1, 5]                   --
|    |    |-Sequential: 3-2              [-1, 5]                   55
|    |    |-SumAggregation: 3-3          [-1, 5]                   --
|    |    |-Sequential: 3-4              [-1, 5]                   405
.    .
.    .    ...
.    .
|    |-MessagePassingBlock: 2-12         [-1, 5]                   --
|    |    |-Sequential: 3-29             [-1, 5]                   55
|    |    |-SumAggregation: 3-30         [-1, 5]                   --
|    |    |-Sequential: 3-31             [-1, 5]                   405
|-Decoder: 1-2                           [-1, 1]                   --
|    |-Sequential: 2-13                  [-1, 1]                   --
|    |    |-Linear: 3-32                 [-1, 1]                   6
==========================================================================
Total params: 4,616
Trainable params: 4,616
Non-trainable params: 0
```

Remark. We want to point out that we train our network in such a way that it will produce a solution in the interval $[1, 2]$. This is an arbitrary interval that we choose as a baseline. For an actual application, one may need to pick this interval more carefully. However, recalling `Proposition` 3.1.2, the key aspect of the chosen interval is not the mean, but the amplitude. Indeed, with our choice, one may subtract the empirical mean of $\mathbf{x}$ to recover a zero mean solution (that is, also the one with minimum l^2-norm) in the interval $[-0.5, 0.5]$.

5.4.3 Results

This time, the training was much more intense and ended with losses on the order of magnitude of 10^{-3}. These values were also obtained for what concerns the test set. We recall the mean relative error defined in `Formula` 5.7. After evaluation in the test set, we can observe that the maximum

relative error found is 3.77%. Of course, these values are much higher than the ones obtained in `Experiment` 5.3, but they are still very promising.
Then, as we did before, our goal was to evaluate the capacity of our network by gradually increasing the number of vertices in the graph and calculating the solution of the Laplacian system of the graph. We compare the time needed by the network and by solving the optimization `Problem` 5.8. We used the following numbers of nodes: 10, 50, 100, 500, 1000, 5000, 10000, 20000, 50000, 100000, 200000, 500000 and 1000000. The solution of the optimization problem was done only up to 10,000 nodes due to the greatly increasing time needed. In `Figure` 5.15, we can see the required time.

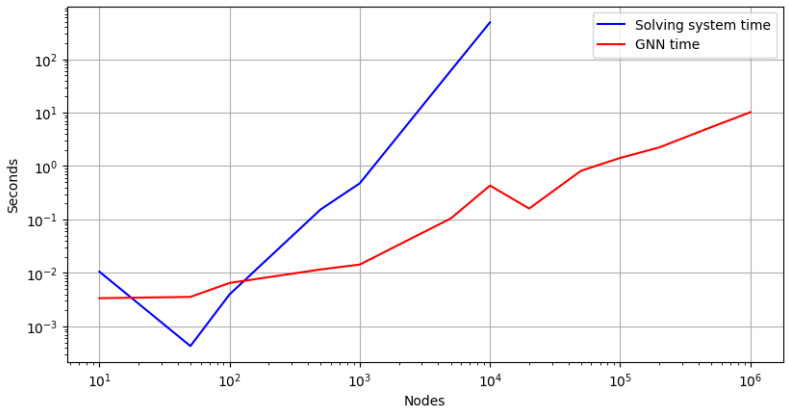


Figure 5.15: Comparison of time needed

As we have anticipated, the time needed by the optimization problem increases very quickly, while the inference time of the neural network is way more reasonable. In particular, we can see that even with a million nodes, the network took only 10 seconds to compute the solution.

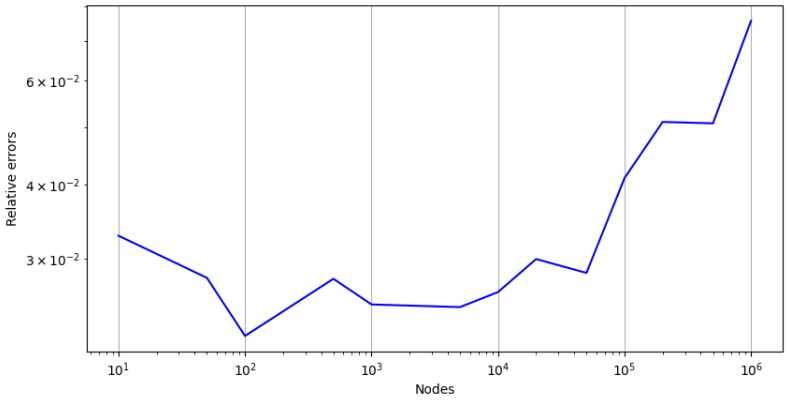


Figure 5.16: Mean relative errors of the GNN solution

Let's now look at the errors reported in `Figure` 5.16. This time, we can notice that the relative errors are slightly increasing, in particular for graphs with a very high number of nodes (hence graphs that are very distant from the training set composed of graphs with 100 to 200 nodes). However, for all systems solved by GNN, the mean relative errors are less than 10%.

Remark. From what we have obtained, it seems that the same relative error is kept by increasing the dimension of the graph by 2/3 order of magnitude. It is reasonable to think that if one needs a good approximation for graphs with 1 million nodes, he would need a training set of graphs with more or less 10000 vertices.

5.5 Elliptic PDE with GNN

After all the simulations we have performed, we put all the results together to implement the key experiment of this work, namely the implementation of a PDE solver with Graph Neural Networks.

5.5.1 Problem

Consider a generic elliptic PDE problem such as `Problem` 3.3, defined on an **undirected, connected and acyclic** graph $G = (V, E)$. We consider a two-dimensional space to be able to plot all the results with clarity. In particular, we want to solve the following problem

$$\begin{cases} -\Delta u + u = 0 & \text{in } G \\ u(v) = g & \forall v \in V_D \\ \sum_{e \in E_v} \frac{\mathrm{d}u}{\mathrm{d}x^e}(v) = 0 & \forall v \in V_N \end{cases} \tag{5.9}$$

where

$$g(x, y) = \begin{cases} 2 & \text{if } x \geq 0 \\ 1 & \text{otherwise} \end{cases}$$

and where we impose Dirichlet conditions on all the boundary vertices. By boundary vertices, we mean the vertices that have only one outgoing edge. For a visual explanation, see `Figure` 5.17.

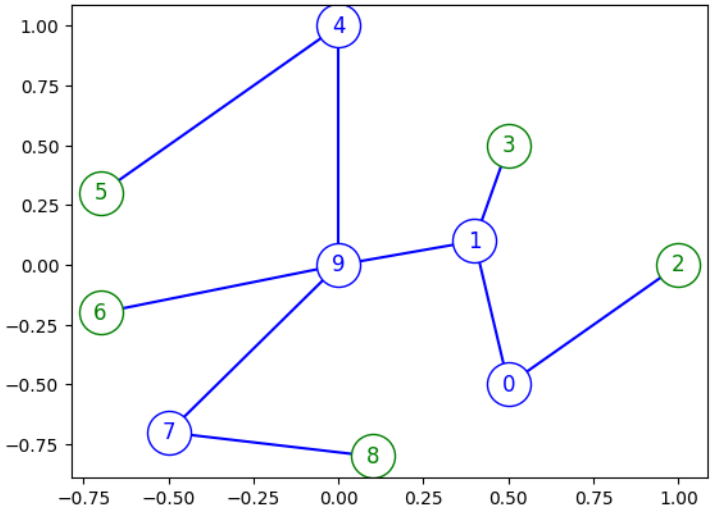


Figure 5.17: Undirected, connected, and acyclic graph. In green are the boundary vertices where Dirichlet conditions are applied. In blue are the vertices with Neumann conditions.

5.5.2 Architecture of the network

For this problem, the network has the following specifications:

- A dataset of $N = 500$ graphs randomly generated with number of vertices $n \sim \mathcal{U}(25, 50)$. The training-validation-testing division of the dataset used was 70-20-10.
- For each graph, a mesh with 4 nodes per edge is created, and the PDE is solved using our FEM implementation.
- A matrix of dimension $(n_{nodes}, 2)$ as an input signal. The first column has 1 in all entries, whereas the second column has 1 if the corresponding vertices are on the boundary and 0 otherwise. Notice that n_{nodes} is the number of vertices of the graph that represents the mesh.
- The vector of FEM solution as the output signal.
- 3 edge features, computed as we previously explained. See `Equation` 4.3.

- Encoder that increases the size of input features to match the size of hidden features. We chose to use 5 hidden features. At the end of the Processor, we used a Decoder to go back to a vector in one-dimensional space.
- Forty message passing steps ($m = 40$). This value was calculated from the fact that in `Experiment` 5.4 we had $m = 10$. Since here we use 4 nodes per edge, to reach the same distance in the physical graph, we have to multiply $10 \times 4 = 40$.

The network has the following (truncated) summary output.

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
|-Encoder: 1-1                           [-1, 5]                   --
|    |-Sequential: 2-1                   [-1, 5]                   --
|    |    |-Linear: 3-1                  [-1, 5]                   15
|    |-Sequential: 2-2                   [-1, 5]                   --
|    |    |-Linear: 3-2                  [-1, 5]                   20
|-ModuleList: 1                          []                        --
|    |-MessagePassingBlock: 2-3          [-1, 5]                   --
|    |    |-Sequential: 3-3              [-1, 5]                   215
|    |    |-SumAggregation: 3-4          [-1, 5]                   --
|    |    |-Sequential: 3-5              [-1, 5]                   325
.    .
.    .    ...
.    .
|    |-MessagePassingBlock: 2-42         [-1, 5]                   --
|    |    |-Sequential: 3-120            [-1, 5]                   215
|    |    |-SumAggregation: 3-121        [-1, 5]                   --
|    |    |-Sequential: 3-122            [-1, 5]                   325
|-Decoder: 1-2                           [-1, 1]                   --
|    |-Sequential: 2-43                  [-1, 1]                   --
|    |    |-Linear: 3-123                [-1, 1]                   6
==========================================================================
Total params: 21,641
Trainable params: 21,641
Non-trainable params: 0
```

Remark. We would like to point out the following observations:

- We used smaller graphs in the dataset because we have to consider the mesh refinement. Since we are dealing with undirected, connected, and acyclic graphs, we have

$$n_{nodes} = n + (n-1) \times 4 \tag{5.10}$$

  Hence, for example, a physical graph with 30 vertices has 146 mesh nodes.

- In this case, we have different dimensions between the input and output signal. Following our notation, we have $q = 2$ and $q' = 1$. Of course, in such cases the use of encoders and decoders is essential.
- We decided to use only 4 internal nodes because otherwise the number of message-passing steps to use to keep the same distance reached would be too high. For future implementation, we may consider using two different message-passing step procedures: one that acts on the physical vertices of the graph and the other for the mesh nodes.
- During the network training phase, we noticed that it is better to have a deep network (high m) with simple inner MLPs than a few message-passing steps with inner MLPs with many parameters. Since these two alternatives would have approximately the same number of total parameters, we again see how important the message-passing steps are to obtain a good result.

Example 5.5.1. If we did not consider the mesh refinement, the graph in `Figure` 5.17 will have the following input signals matrix.

$$U = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}^T$$

5.5.3 Results

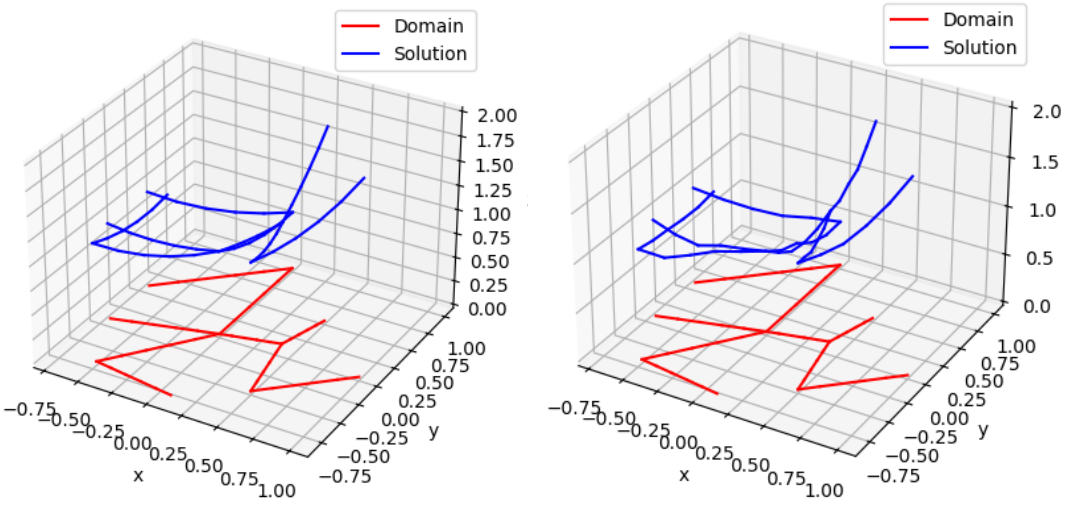
Since this time we have many more parameters, the training took more time. In the end, we had training and validation losses on the order of magnitude of 10^{-3}, similar to those obtained in `Experiment` 5.4. For what concerns the testing set (composed of 50 graphs), we have:

	Minimum value	**Mean value**	**Maximum value**
Loss value	4.78×10^{-4}	2.02×10^{-3}	5.60×10^{-3}
Mean relative error	3.05%	5.85%	15.1%

where the mean relative error was computed using `Formula` 5.7. Furthermore, we point out that only 3 solutions have a relative error greater than 10%.

Visual comparison

To better understand the network capability to calculate the solution of the `Problem` 3.3, we compare the graph of an FEM solution with the corresponding GNN solution. In particular, considering a domain such as the graph in `Figure` 5.17, the comparison plot is in `Figure` 5.18.



(a) FEM solution

(b) GNN solution

Figure 5.18: Visual comparison of solutions

It can be seen from the graph that the GNN solution is not so bad, but still not as smooth as the FEM solution. This result is better when one considers that the domain was not present in the training set and the validation set. Finally, we compute the relative error that is 5.06%.

Size increasing graphs

The last test was to compare the time required by FEM and GNN to compute the solution. We used the following increasing number of vertices: 5, 10, 20, 50, 100, 200, 500, 1000, 1500, 2000, 5000, 10000 and 20000. Always 4 nodes for each edge were used. Notice that the actual number of nodes of the mesh graph is higher and can be computed using `Formula` 5.10. The FEM solution was computed only up to 2000 physical vertices because of the rapidly increasing time needed. The comparison of the result can be found in `Figure` 5.19.
As we may have expected from reading the previous experiments, the inference time of the GNN is much lower than the FEM time, as we keep increasing the size of the graphs. Moreover, for large graphs, the GNN time is even lower than the time needed to build the mesh.

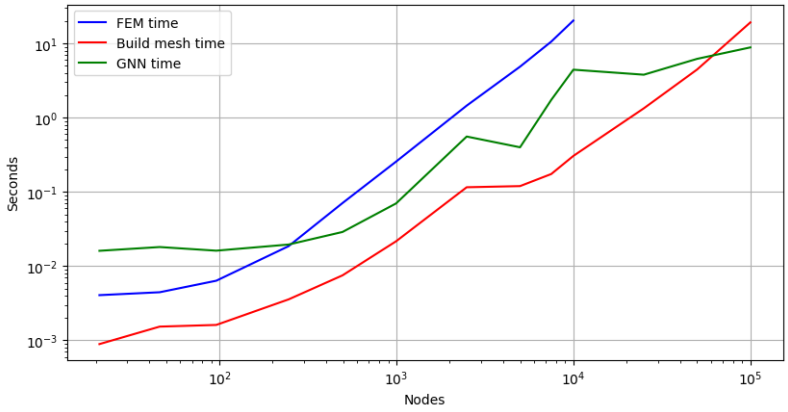


Figure 5.19: Comparison of time needed

To conclude, we can look at the relative errors, available unfortunately only when the FEM solution was calculated. The error plot is in `Figure` 5.20 and we can notice that the error is always below 10%. Overall, the results obtained are very encouraging, even considering that this is only a first test and there are still many details that can be improved.

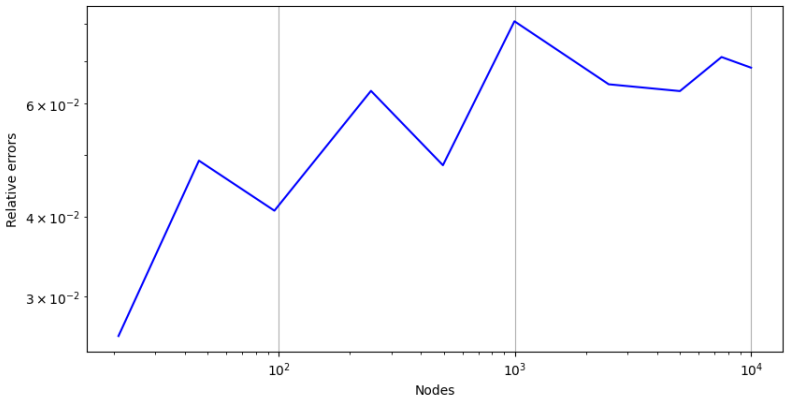


Figure 5.20: Mean relative errors of the GNN solution

Chapter 6

Python implementation

We decided to use Python as the programming language to write our code because it is one of the most widely used languages for deep learning development. Moreover, it has several frameworks that make creating Neural Networks very easy. All of our code, including that used for numerical experiments, is available in the following GitHub repository:

https://github.com/ploki99/GP-GNN

6.1 Python features

Unlike C++, Python is a much more 'permissive' language: variables don't have a type, there are no private methods, etc... . To write clean, versatile, reusable, and maintainable code, we decided to use the following tools:

1. **Decorators**, they allow you to extend the behavior of a function or class without changing its source code,
2. **Name mangling**, any identifier with two leading underscores and one trailing underscore is textually replaced with `_classname__identifier`, where `classname` is the name of the current class. With name mangling, we can have pseudo-private methods and attributes. For further details, see `Example` 6.1.1.

In particular, we used the decorators:

- **@classmethod**
  Class methods are methods that are bound to a particular class. These methods are not tied to any instances of the class. They are handy, for instance, in the following case

```
class TestClass:
  def __init__(self, a, b):
    self.first = a
    self.second = b

  @classmethod
  def get_object(cls):
    return cls(1, 2)

obj = TestClass.get_object()
```

  where we use a class method to return an instance of the object.

- **@staticmethod**
  Static methods are utility methods not bound to a class or an instance of the class. An example is the following code

```
class TestClass:
    @staticmethod
    def static_method():
        print("This is a static method")

TestClass.static_method()
```

- **@abstracmethod**
  Abstract methods are declared inside an abstract class without any method definition. This method is meant to be implemented by the base class, which implements the parent abstract class. The decorator can be used in the following way

```
from abc import ABC, abstractmethod

class Parent(ABC):
    @abstractmethod
    def abstract_method(self):
        pass


class Child(Parent):
    def abstract_method(self):
      print("Child abstract method")

child = Child()
child.abstract_method()
```

Example 6.1.1. Here, we show a usage example of name mangling. Let us suppose to have the following snippet of code

```
class Person:
    def __init__(self, name):
        self.__name = name

    def display_name(self):
        print(self.__name)

p = Person("Pippo")
```

Then, we can access `Person.__name` using `p.display_name()`. On the other hand, if you type `print(p.__name)`, you will get an error. However, accessing the pseudo-private attribute using `print(p._Person__name)` is still possible. Of course, all this works even for class methods.

6.2 Dependencies

The main dependencies we used to do our project are:

- **FEniCS**. It is a popular open-source computing platform for solving partial differential equations (PDEs) with the finite element method (FEM). In particular, we included the Dolfin module to create a mesh starting from a graph and then solve the associated elliptic PDE. We used The FEniCS book [8] as a reference.
- **PyTorch**. It's a Python package that provides tensor computation and deep neural networks built on a tape-based autograd system. We used it to create the various layers of the neural network. As a reference, we followed the instructions on the PyTorch website [15].
- **PyG** (PyTorch Geometric). It's a library built upon PyTorch to easily write and train Graph Neural Networks (GNNs) for a wide range of applications related to structured data. It was fundamental to create the Message Passing Blocks and manage the creation of the dataset composed of graphs, making the execution done in parallel. We used the tutorials on the official website [13].
- **Matplotlib**: a comprehensive library for creating static, animated, and interactive visualizations in Python. We used it to plot the graphs, the solutions of the PDEs, and other useful plots.

The aforementioned libraries have their dependencies (such as NumPy), so we suggest following the instructions on the GitHub repository to create a conda environment. In this way, all the needed modules are automatically installed without any conflicts and the execution of the project becomes much easier.

6.3 Code organization and explanation

The code in the GitHub repository is organized as follows

```
|-- examples
|    |-- discrete_laplacian/
|    |-- fem_usage/
|    |-- graph_laplacian/
|    |-- graphs_generation/
|    |-- pde_with_gnn/
|-- LICENSE
|-- main.py
|-- README.md
|-- src
     |-- gnn/
     |-- graph/
     |-- pde/
```

where the `examples` folder contains the codes used to run the numerical experiments. The files contained here are called from `main.py` which is the main entry of the project (see `Section` 6.4 for more details). Instead, the main work is contained in the `src` folder. Here we can see that the code is arranged to match the structure of this report, i.e.:

- The `graph` folder contains the code used to generate random graphs, following the theory illustrated in `Chapter` 2.
- The `pde` folder has the FEM implementation explained in `Chapter` 3.
- Lastly, in the `gnn` folder there is the code for our version of Graph Neural Network, as we presented in `Chapter` 4.

The full details of how we implemented these last things are in the following paragraphs.

6.3.1 Graph generations

In `Figure` 6.1 you can see the classes we have created, with the main public interface written in the diagram. With the rigid "Has" arrow, we denote that a class has, as an attribute, a certain

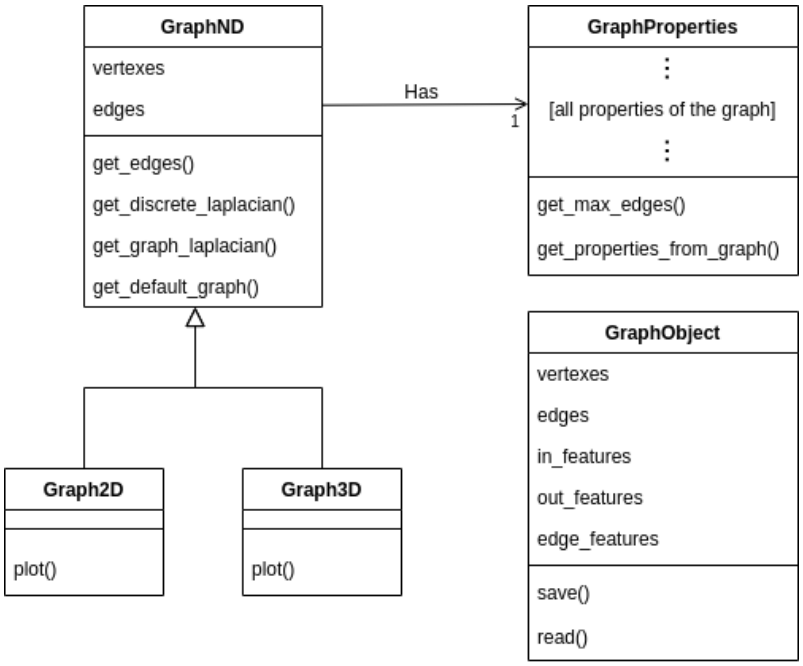


Figure 6.1: Class diagram for graph random generation

instance of another class. The nearby number is the multiplicity of the object (if it's greater than one, it indicates a "list" of objects). On the other hand, the vertical arrow denotes inheritance.

We can notice that the classes **Graph2D** and **Graph3D** extend the parent **GraphND** by adding the plot method. The "all properties of the graph" are the ones we have already stated. Finally, the **GraphObject** class is used to save on memory a graph with additional optional data: this is fundamental to creating the dataset used to train the networks.

Remark. The pseudo-private methods are not included in this diagram and the following ones. Moreover, not all the public interface is present, and the arguments of the methods are omitted. This is done to avoid unnecessary verbosity since the objective of these figures is to roughly understand the code structure.

Example 6.3.1. The easiest way to create a graph is to use our default options and type:

```
# Import the necessary class
from src.graph.graph2D import Graph2D

# Get a default graph with 10 vertices
graph = Graph2D.get_default_graph(n_vertexes=10)
# Plot the created graph
graph.plot(title="Random generated graph")
```

Then, we can save it on memory:

```
# We have to import also this
from src.graph.graphObject import GraphObject

# Create object
obj = GraphObject(vertexes=graph.vertexes, edges=graph.get_edges())
# Save to file
obj.save("random_graph")
```

6.3.2 FEM on graphs

To solve partial differential equations on graphs, we implemented the classes in `Figure` 6.2. In particular, we can notice that **PdeSolver** has the abstract method `solve()` because in general one could have different types of equations. In our case, we specialized the class only for elliptic PDEs, as the one in `Equation` 3.3. However, the code is easily extendable for other types of equations. The **GraphMesh** class is used as the interface to the **Mesh** class that belongs to the Dolfin library, and it's able to create a mesh starting from an instance of **GraphND**. Finally, we

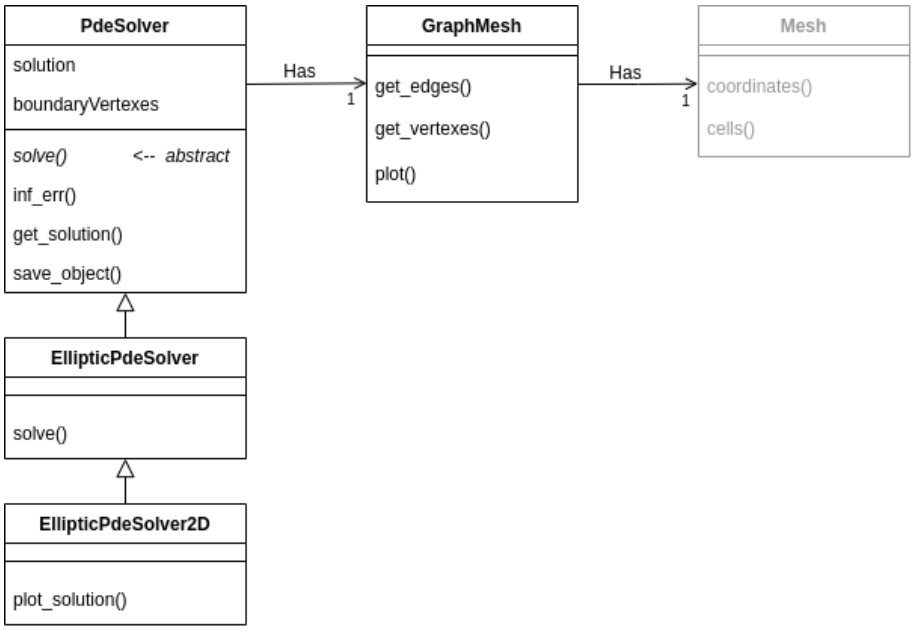


Figure 6.2: Class diagram for FEM on graphs

have the class **EllipticPdeSolver2D** which adds the plot of the solution in case of 2D problems.

Even if the plot method for 3D problems is not provided, the code can compute the solution also for 3D-domain PDEs.

Remark. In all the class diagrams, we used the gray color for the classes that we did not have implemented since they belong to other libraries. Moreover, we omitted the inheritance from **ABC**, necessary in case the class has abstract methods.

Example 6.3.2. Suppose we want to solve the following elliptic PDE

$$\begin{cases} -\Delta u + u = x + y & \text{in } G \\ \sum_{e \in E_v} \frac{\mathrm{d}u}{\mathrm{d}x^e}(v) = 0 & \forall v \in V \end{cases}$$

for a randomly generated graph $G = (V, E)$. Then, we can use the following code:

```
# Import Dolfin class
from dolfin import Constant, Expression
# Import the necessary class
from src.graph.graph2D import Graph2D
from src.pde.ellipticPdeSolver2D import EllipticPdeSolver2D

# Create graph
graph = Graph2D.get_default_graph(n_vertexes=10)
# Define problem functions
w = Constant(1)
f = Expression("x[0]+x[1]", degree=2)
# Solve pde
pde = EllipticPdeSolver2D(graph)
pde.solve(w, f)
# Plot computed solution
pde.plot_solution(title="Computed solution")
```

6.3.3 GNN

In `Figure` 6.3 there are the classes we have implemented for the creation of the Graph Neural Network. In particular, the main class is **GraphNeuralNetwork** which has the **Encoder** and **Decoder** objects and m **MessagePassingBlock** instances. We recall that m denotes the number of message-passing steps. We can note that 3 out of 4 aforementioned classes derive from **Module**,

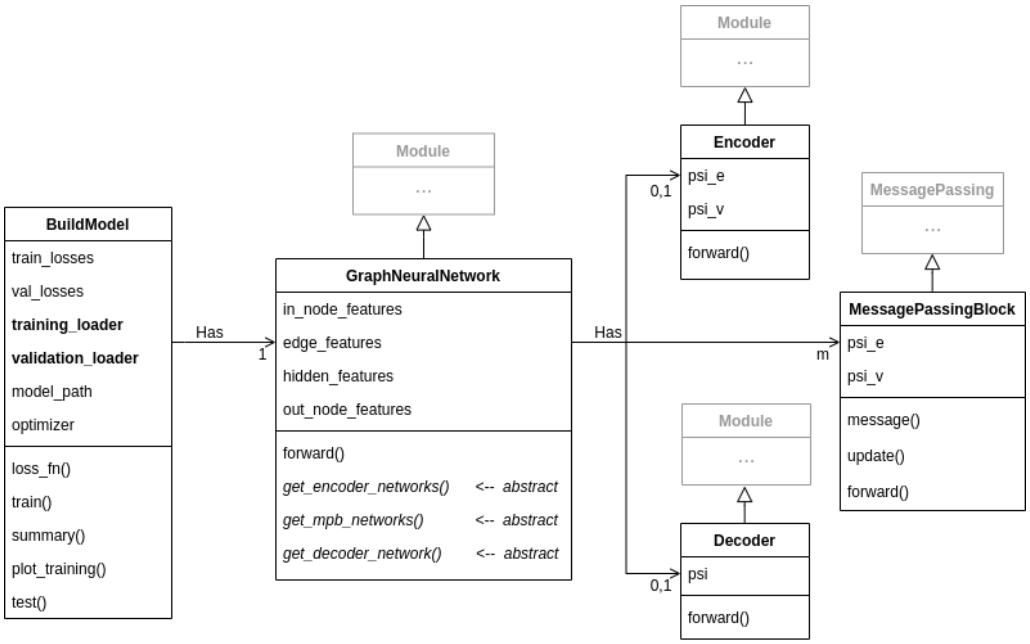


Figure 6.3: Class diagram for Graph Neural Networks

which is a PyTorch class used to create Neural Networks. On the other hand, **MessagePassing-Block** derives from **MessagePassing**, which is a class from PyTorch Geometric, and allows the

easy creation of message-passing layers. To keep the code as general as possible, **GraphNeuralNetwork** has three abstract methods used to define the multi-layer perceptron networks needed by the various components of the whole GNN. For this reason, it's necessary to derive this class to create a GNN, as we can see in the following example.

Example 6.3.3. As a usage example of the inheritance from **GraphNeuralNetwork** class, we report the code used to create the base class for the discrete Laplacian simulation:

```
from torch import nn
from src.gnn.graphNeuralNetwork import GraphNeuralNetwork

class DiscreteLaplacianGNN(GraphNeuralNetwork):
    def __init__(self):
        # Call the parent constructor specifying the number of features used
        super().__init__(in_node_features=1, edge_features=0, hidden_features=1,
                         mp_steps=1, out_node_features=1)

    def get_encoder_networks(self):
        # MLP for encoding edge features
        psi_e = nn.Sequential()
        # MLP for encoding node features
        psi_v = nn.Sequential()
        return psi_e, psi_v

    def get_mpb_networks(self):
        # MLP used for message step
        psi_e = nn.Sequential(
            nn.Linear(2 * self.hidden_features, self.hidden_features)
        )
        # MLP used for update step
        psi_v = nn.Sequential(
            nn.Linear(2 * self.hidden_features, self.hidden_features)
        )
        return psi_e, psi_v

    def get_decoder_network(self):
        # MLP for decoding node features
        psi = nn.Sequential()
        return psi
```

Lastly, the class **BuildModel** is used to build, train, and test a given GNN with a given dataset. For what concerns the creation of the dataset, we can dive into more details. The attributes **training_loader** and **validation_loader** are instances of **DataLoader** class, which is organized as shown in Figure 6.4. Here, we can see that **DataLoader** needs a derived class of **Dataset**,

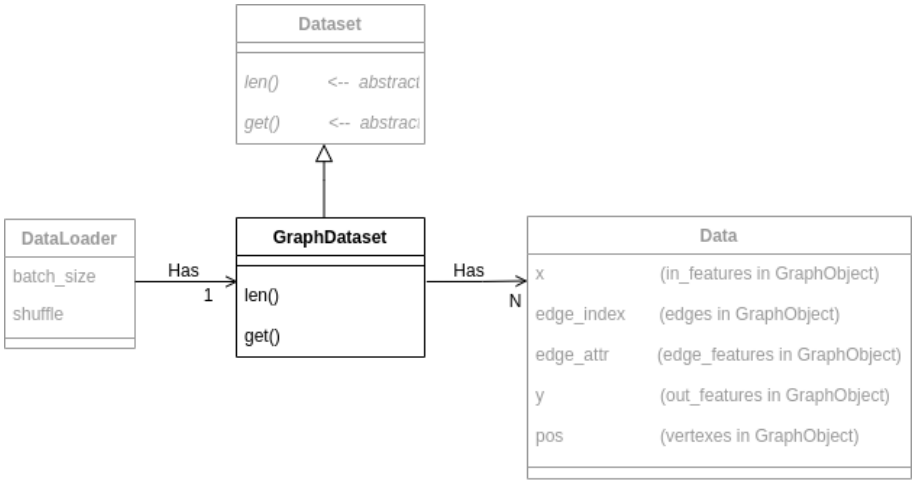


Figure 6.4: Class diagram for the dataset creation

which in our case is called **GraphDataset**. This last object has to manage the N graphs of the dataset. To do that, each graph (previously saved in memory as **GraphObject** object) has to be

converted to a **Data** instance. Once the graph is converted as **Data** object, the **DataLoader** can correctly load and use all the information that defines an instance of **GraphObject**.

Remark. This last conversion procedure may seem unnecessarily complex, but it is essential to ensure proper **parallelization of the code**. As a result, the training of our Graph Neural Network is executed in parallel (if the machine supports it), and running times are greatly reduced.

6.4 How to run the experiments

As we mentioned before, the file `main.py` is the main entry to run the simulations. There are three major ways to execute this file:

- If you want to see the help message, type

```
python main.py -h
```

- To run the experiment, use

```
python main.py -r {1, 2, 3, 4, 5}
```

  where you have to choose a number between 1 and 5. The numbers denote the following simulations

  1. Graph generation: generate random graphs given properties;
  2. FEM usage: test code for FEM on graph implementation;
  3. Discrete Laplacian: compute discrete Laplacian with GNN;
  4. Graph Laplacian: solve graph Laplacian system with GNN;
  5. PDE with GNN: solve PDE defined on a graph with GNN.

- For each experiment, you can specify certain options, adding `-o name=value`. Use the following command to see all the available options (with the default values)

```
python main.py -l
```

  You can also decide to not specify any options, in this case, the default ones are used.

Remark. Running some simulations may be very time-consuming, so we suggest running with the default options (i.e. do not specify any option). Even better, you can look at the **Python notebook** to see the results without losing computation time. Indeed, for each simulation, there is the Python notebook in the corresponding folder with the same outputs presented in this report.

Example 6.4.1. If you want to run the discrete Laplacian simulation, specify to create a new dataset, and train again the neural network, you can use the command:

```
python main.py -r 3 -o train=True -o data=True
```

Instead, if you want to run the graph generation simulation, changing the seed used to randomly create graphs, you can type:

```
python main.py -r 1 -o seed=33
```

Chapter 7

Conclusion and future works

Our work proved that GNNs are a valid alternative for solving graph-defined problems. Despite the good results we obtained, we would like to stress that this is just the beginning: the approach we used can be generalized to reach even better achievements. Indeed, we did not test the networks with real-life applications and we used only undirected, connected, and acyclic graphs. For these reasons, we have thought about possible future developments:

- Test the network with other problems defined on graphs (also using real data). In terms of implementation changes, this should be simple enough because the code is easily extendable.
- Use different types of graphs: investigate the behavior of the network with cycles and directed graphs.
- Realize a better way to perform message-passing steps in case of fine mesh refinements. For example, we could use one message-passing procedure for the physical vertices of the graph and another for the internal nodes along the edges.

For what concerns the "implementation" part, we state the following possible updates that can be done to the code:

- Since we will be using other types of graphs, improve the generation time for all of them, by thinking of smarter algorithms (eventually, using a parallel approach).
- Test the implementation on better machines, which can run multiple threads in parallel, to see if the parallelization part of the code works properly.
- Use pybind to enhance execution time: write some portions of the code in C++ and then link them to Python. For now, we did not use this approach because the heaviest computation part of our implementation is (of course) the training of the neural networks. This part is already optimized by the PyTorch library we used. Furthermore, we configured the data set to allow us to perform the training in parallel. In any case, it could be interesting to rewrite some procedures, such as graph generation, in C++.

Overall, we are satisfied with the decreasing inference time we achieved and the generalization capability of the GNNs, which we consider the most promising results we have found.

Bibliography

[1] Mario Arioli and Michele Benzi. "A finite element method for quantum graphs". In: *IMA Journal of Numerical Analysis* 38.3 (2018), pp. 1119–1163.

[2] Albert-László Barabási. *Network science*. Cambridge University Press, 2016.

[3] Ulrik Brandes, Linton C Freeman, and Dorothea Wagner. *Social networks*. 2013.

[4] Reinhard Diestel. *Graph Theory*. 3rd edition. Graduate Texts in Mathematics. Springer, 2006.

[5] Nicola Rares Franco et al. "Deep Learning-based surrogate models for parametrized PDEs: handling geometric variability through graph neural networks". In: *arXiv preprint arXiv:2308.01602* (2023).

[6] Frank Kammer and Hanjo Täubig. "Graph connectivity". In: (2004).

[7] Remi Lam et al. "Learning skillful medium-range global weather forecasting". In: *Science* 382.6677 (2023), pp. 1416–1421.

[8] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*. Vol. 84. Springer Science & Business Media, 2012.

[9] Sadia Majeed et al. "Social Network Analysis Visualization Tools: A Comparative Review". In: *2020 IEEE 23rd International Multitopic Conference (INMIC)*. IEEE. 2020, pp. 1–6.

[10] Bojan Mohar. "Graph Laplacians". In: *Topics in Algebraic Graph Theory* (2004), p. 113.

[11] Luiz Pessoa. "Understanding brain networks and brain organization". In: *Physics of life reviews* 11.3 (2014), pp. 400–435.

[12] C Pozrikidis. "Numerical simulation of blood flow through microvascular capillary networks". In: *Bulletin of mathematical biology* 71 (2009), pp. 1520–1541.

[13] PyG Team. *PyG Documentation — pytorch_geometric documentation*. URL: `https://pytorch-geometric.readthedocs.io/en/stable/`.

[14] Daniel A Spielman. "Algorithms, graph theory, and linear equations in Laplacian matrices". In: *Proceedings of the International Congress of Mathematicians 2010 (ICM 2010) (In 4 Volumes) Vol. I: Plenary Lectures and Ceremonies Vols. II–IV: Invited Lectures*. World Scientific. 2010, pp. 2698–2722.

[15] Suraj Subramanian, Seth Juarez, Cassie Breviu, Dmitry Soshnikov, and Ari Bornstein. *Learn the Basics — PyTorch Tutorials*. URL: `https://pytorch.org/tutorials/beginner/basics/intro.html`.