Deep Learning-based surrogate models for parametrized PDEs: including geometrical features through graph neural networks

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica

Filippo Tombari, 10569815

Advisor:
Prof. Andrea Manzoni

Co-advisors:
Dr. Nicola Rares Franco
Dr. Stefania Fresca

Academic year:
2021-2022

Abstract: Mesh-based simulations play a key role when modeling complex physical systems that, in many disciplines across Science and Engineering, require the solution of time-dependent partial differential equations (PDEs). In this context, Full Order Models (FOMs), such as those relying on, e.g., the finite element method, can reach high levels of accuracy, however often yielding intensive simulations to run. For this reason, surrogate models are developed in order to replace computationally expensive solvers with more efficient ones, which can strike favorable trade-offs between accuracy and efficiency. This work explores the application of graph neural networks (GNNs) for the simulation of time-dependent PDEs depending on either physical or geometrical parameters. GNNs have recently shown great promise in solving complex problems in domains such as computer vision and natural language processing: this Thesis aims at investigating their potential in view of the efficient approximation of PDEs. The advantage of using GNNs in these problems relies on their ability to generalize to different geometries by introducing a suitable graph representation for the mesh.
The work starts by introducing the theoretical background of time-dependent PDEs and classical numerical methods for their solution. It then introduces the concept of GNNs, their architectures, and a possible way to apply them to graph-based problems. The Thesis proposes a novel method for using GNNs to solve PDEs by (i) converting the PDE into a graph-based problem and (ii) training a GNN on the resulting graph.
The effectiveness of the proposed approach is assessed through a series of experiments showing that GNNs are capable of outperforming traditional numerical methods in terms of computational efficiency and generalization to new scenarios.

Key-words: graph neural networks, partial differential equations, surrogate models

1. Introduction

In many areas of Science and Engineering, models governed by Partial Differential Equations (PDEs) which may depend on one or more parameters are ubiquitous. These parameters may be related to either the physical properties or the geometry of the domain which we consider. Modeling this kind of problem typically requires the introduction of a suitable mesh as first basic ingredient, to discretize the computational domain and to define the set of approximating functions required to represent the problem solution, ultimately allowing the search for highly accurate approximations. In this framework, Full Order Models (FOMs), which rely on numerical schemes such as the Finite Element Method (FEM)[4], Isogeometric Analysis [2], or the Spectral Element Method (SEM) [16] usually offer high levels of accuracy, however featuring very high computational costs, that become infeasible in many applications where the solver has to be called multiple times. For this reason, there is a growing literature aimed at replacing these expensive models with suitable cheaper models, also known as surrogate or Reduced Order Models (ROMs), which usually offer a very good trade-off between the computational cost and the accuracy of the simulation.

In this respect, recently, deep learning-based reduced order models have been proposed [6, 8, 9] to better tackle the nonlinearity often entailed by time-dependent parametrized PDEs, thus overcoming the limitations of classical ROMs built exploiting linear techniques, such as proper orthogonal decomposition (POD) [14]. As of now, DL-ROMs relying on deep neural networks, like Feed Forward Neural Networks (FFNNs) and Convolutional Neural Networks (CNNs), can be used for building efficient ROMs to solve mesh-based problems and they have been shown to be able to provide very accurate but cheap approximation bias when modeling time-dependent parametrized PDEs [5]. On the other hand, these architectures cannot infer anything about the geometry of the problem because their structure is strictly dependent on the employed mesh. Moreover, they do not leverage the features that characterize the mesh and its structure. For instance, when dealing with complex domains such as 3D domains and unstructured meshes, exploiting the mesh connectivity can provide a better understanding of the geometrical features of the problem.

For this reason, we introduce a class of surrogate models based on Graph Neural Networks (GNNs), which are particular neural nets that do not require information about the number of nodes or the number of edges of the mesh, potentially leading to a much more flexible approach that can accommodate different geometries at the same time. So far, inductive learning on graphs has been mainly used for node-classification problems [1, 3, 12], with only limited attempts to use GNNs in order to solve graph-based problems in various domains [23]. In the context of PDE simulation, a method for using GNNs to solve PDEs by representing the PDE as a graph and training a GNN on the resulting graph has been formerly proposed [10]. However, this approach requires a careful design of the graph structure, and the scalability of the method to high-dimensional problems still remains an open challenge. In the last few years, this approach was adopted for learning mesh-based simulations. In particular, Pfaff et Al. [20] show how Graph Neural Networks can be useful for learning adaptive mesh representations to handle problems with strongly nonlinear displacement, like the one of a flag waving in the wind. Moreover, Hernández et Al. [13] have recently used GNNs to build a physics-informed deep learning model which can also take into account the intrinsic mathematical structure of physical problems. More recently, Pegolotti et al. [19] employed GNNs to construct a reduced-order model for cardiovascular simulations that simulates blood flow dynamics on three-dimensional hemodynamic simulation data. In addition, Gladstone et al. [11] implemented two different GNNs to solve time-independent solid mechanics problems and demonstrated the effectiveness of this approach for generalization to unseen domains. In particular, the latter work presents a promising approach for reducing the depth of the network in situations where there is long-range exchange of information. Our work, on the other hand, highlights how deep GNNs are often necessary to accurately capture the dynamics of complex time-dependent problems.

On the other hand, in recent years, there has been increasing interest in applying deep learning techniques to the simulation of PDEs. For example, the work by Raissi et al. [22] proposed a physics-informed neural network (PINN) for solving PDEs. PINNs are a type of neural network that can enforce the physical constraints of a PDE while simultaneously approximating its solution. Another approach, proposed by Sirignano and Spiliopoulos [24], is to use convolutional neural networks (CNNs) to solve PDEs by discretizing the PDE onto a grid and treating it as an image. However, these methods show some limitations, such as the need for a large amount of data, difficulty in handling complex geometries, and lack of interpretability. In contrast, GNNs are a promising approach for solving PDEs due to their ability to handle graph-based problems and their interpretability since they can naturally incorporate geometric information and structural relationships between mesh nodes, which can be challenging to represent using traditional methods.

This work explores the advantages of using GNN-like architecture to model physical problems involving time-dependent PDEs that also depend on physical and geometrical parameters. The model presented exploits the graph representation of the mesh and the GNN capability of inferring its geometrical structure. We will also see how this approach requires a significantly small number of parameters compared to models based on FFNNs and CNNs. The structure of the work is as follows. In Section 2 we introduce the theoretical background of time-dependent parametrized PDEs; in Section 3 we describe our design choices for the network architecture, explaining the concepts upon which GNNs are built; in Section 4 we present our actual application to surrogate modeling and finally, in Section 5 we report some numerical results concerning a benchmark advection-diffusion problem either with a time-varying advection term or in a Stokes flow, exploring both 2D and 3D applications.

2. Modeling time-dependent PDEs

For the sake of generality, despite the numerical experiments will only refer to linear time-dependent parametrized PDEs, we consider a generic nonlinear, time-dependent PDE, since the results can also be extended to more general problems. In particular, we consider a PDE depending on a set of input parameters $\boldsymbol{\mu} \in \mathcal{P}$, where the parameter space $\mathcal{P} \subset \mathbb{R}^{n_\mu}$ is given by a bounded and closed set; in our analysis, input parameters represent physical and geometrical properties of the system, like, e.g., material properties, boundary conditions, or the shape of the domain.
For the sake of space, We adopt a fully algebraic perspective and assume to start from the high-fidelity (spatial) approximation of the PDE. Regardless of the spatial discretization adopted – such as, e.g., the finite element method, Isogeometric Analysis, or the spectral element method – the FOM can be expressed as a nonlinear parametrized dynamical system. Hence, given $\boldsymbol{\mu} \in \mathcal{P}$, we aim at solving the initial value problem:

$$\begin{cases}\mathbf{M}(\boldsymbol{\mu})\dot{\mathbf{u}}_h(t;\boldsymbol{\mu}) = \mathbf{f}(t, \boldsymbol{u}_h(t;\boldsymbol{\mu});\boldsymbol{\mu}), \quad t \in (0,T),\\ \mathbf{u}_h(0;\boldsymbol{\mu}) = \mathbf{u}_0(\boldsymbol{\mu})\end{cases} \tag{1}$$

where:

- $\mathbf{u}_h(\cdot,\boldsymbol{\mu}) : [0,T) \to \mathbb{R}^{N_h(\boldsymbol{\mu})}$ is the parametric solution of (1);
- $\mathbf{u}_0(\boldsymbol{\mu}) \in \mathbb{R}^{N_h(\boldsymbol{\mu})}$ is the initial condition;
- $\mathbf{f}(\cdot,\cdot,\boldsymbol{\mu}) : (0,T) \times \mathbb{R}^{N_h(\boldsymbol{\mu})} \to \mathbb{R}^{N_h(\boldsymbol{\mu})}$ is a (possibly nonlinear) function, encoding the system dynamics;
- $\mathbf{M}(\boldsymbol{\mu}) \in \mathbb{R}^{N_h(\boldsymbol{\mu}) \times N_h(\boldsymbol{\mu})}$ is the mass matrix of this parametric high-fidelity model; without any loss of generality, $\mathbf{M}(\boldsymbol{\mu})$ is assumed here to be a symmetric positive definite matrix.

The dimension $N_h(\boldsymbol{\mu})$ is related to the finite dimensional subspaces introduced for the sake of space discretization – here $h > 0$ denotes a discretization parameter, such as the maximum diameter of the elements in a computational mesh; consequently, $N_h(\boldsymbol{\mu})$ can be extremely large if the PDE problem describes complex physical behaviors and/or high degrees of accuracy are required for its solution. Furthermore, the number of nodes N_h is dependent on the geometrical parameters $\boldsymbol{\mu}$, as modifying them can alter the number of nodes present in the computational mesh. Consequently, we strive to discover solutions that can generalize across meshes with varying node counts.
We thus aim at approximating the set

$$\mathcal{S}_h = \{\mathbf{u}_h(t;\boldsymbol{\mu}) |\ t \in [0,T),\ \boldsymbol{\mu} \in \mathcal{P} \subset \mathbb{R}^{n_\mu}\} \subset \bigcup_{\boldsymbol{\mu}\in\mathcal{P}} \mathbb{R}^{N_h(\boldsymbol{\mu})} \tag{2}$$

of the solutions to Problem (1) when $(t;\boldsymbol{\mu})$ varies in $[0,T) \times \mathcal{P}$, also referred to as *solution manifold.* In order to solve Problem (1), suitable numerical schemes are used, such as the Backward Euler Method. Thus, denoting by $\mathbf{u}_h^n$, the solution $\mathbf{u}_h$ at time $t^n = n\Delta t$, $n \geq 0$, we need to solve:

$$\begin{cases}\mathbf{M}(\boldsymbol{\mu})\left(\dfrac{\mathbf{u}_h^{n+1}(\boldsymbol{\mu}) - \mathbf{u}_h^n(\boldsymbol{\mu})}{\Delta t}\right) = \mathbf{f}(t^{n+1}, \mathbf{u}_h^{n+1}(\boldsymbol{\mu});\boldsymbol{\mu}), \quad n = 0,\dots,N-1,\\ \mathbf{u}_h^0(\boldsymbol{\mu}) = \mathbf{u}_0(\boldsymbol{\mu}),\end{cases} \tag{3}$$

where $N = \frac{T}{\Delta t}$ is the total number of time steps. Equation (3) implies the solution, at each time step, of a nonlinear system which may increase the overall computational time. For this reason, we want to approximate the manifold in (2) in a way that reduces the time complexity of the simulation without losing much accuracy. In this work, we use a deep learning model based on graph neural networks. These deep neural networks allow us to find an approximate solution $\tilde{\mathbf{u}}(t,\boldsymbol{\mu}) \approx \mathbf{u}_h(t,\boldsymbol{\mu})$. Note that $\tilde{\mathbf{u}}$ is not strictly dependent on the space discretization parameter h of the computational mesh so our model is more flexible in accommodating different geometries at the same time. More precisely, we aim at introducing a suitable deep neural network that can learn the map $\boldsymbol{\Phi} : \bigcup_{\boldsymbol{\mu}\in\mathcal{P}} \mathbb{R}^{N_h(\boldsymbol{\mu})} \times \mathcal{P} \to \tilde{\mathcal{S}}_h$ such that:

$$\begin{cases}\tilde{\mathbf{u}}^{n+1}(\boldsymbol{\mu}) = \boldsymbol{\Phi}(\tilde{\mathbf{u}}^n(\boldsymbol{\mu});\boldsymbol{\mu}), \quad n = 0,\dots,N-1,\\ \tilde{\mathbf{u}}^0(\boldsymbol{\mu}) = \mathbf{u}_h^0(\boldsymbol{\mu}).\end{cases} \tag{4}$$

In this way, the model can be then cheaply evaluated at testing time, completely avoiding:

1. reassembling the operators any time $\boldsymbol{\mu}$ changes;
2. having to solve a nonlinear system at each time step.

Thus far, either ROMs based on reduced basis methods, such as POD-Galerkin, or DL-ROMs, have assumed that all the data is defined on the same domain, requiring the training data to have the same dimension N_h. Our approach addresses a more general problem that cannot be tackled by classical Reduced Order Models.

3. Graph Neural Networks

GNNs were initially conceived as an extension of Convolutional Neural Networks (CNNs) to operate on graph-structured data and overcome their limitations in this domain. Indeed, the convolution operator window is usually slid across a two-dimensional image, and some function is computed as it is passed through many layers. Now, the key property of such operation is that convolution takes a small rectangular section of the image, applies a function to it, and creates a new portion (a new pixel): the node at the center of that pixel combines information from its neighbors, as well as from itself, to generate a new value.
Moreover, CNNs suffer from the problem of node ordering. If we originally named the nodes A, B, C, D, and E, and subsequently labeled them B, D, A, E, and C, the output of a CNN would change. Since graphs are invariant to node ordering, we expect to receive the same result no matter how we order the nodes.
However, the idea does not easily transfer to graph-like structures, and it was initially unclear how to generalize convolutions over grids to convolutions over general graphs, where the neighborhood structure differs from node to node [18, 25].

GNNs aim to extend the idea of convolution to more complex spatial structures, constructing a learnable transformation that preserves the symmetries of the graph (i.e., permutation invariances) across all of its attributes, including nodes and edges. GNNs adopt a *graph-in, graph-out* architecture meaning that these model types accept a graph as input, with information loaded into its nodes and edges, and progressively transform these embeddings, without changing the connectivity of the input graph.

The fundamental ingredients of these architectures are the so-called *message passing* operations, which enable the aggregation of node information while leveraging the depth of the graph. This message-passing propagation can be seen as an information retrieval task from different levels of depth of the graph. In Figure 1 a simple visualization of the message propagation is shown. For each node, the information comes from the neighbors. In this way, adding message-passing steps can be seen as connecting nodes that can be also far from each other.

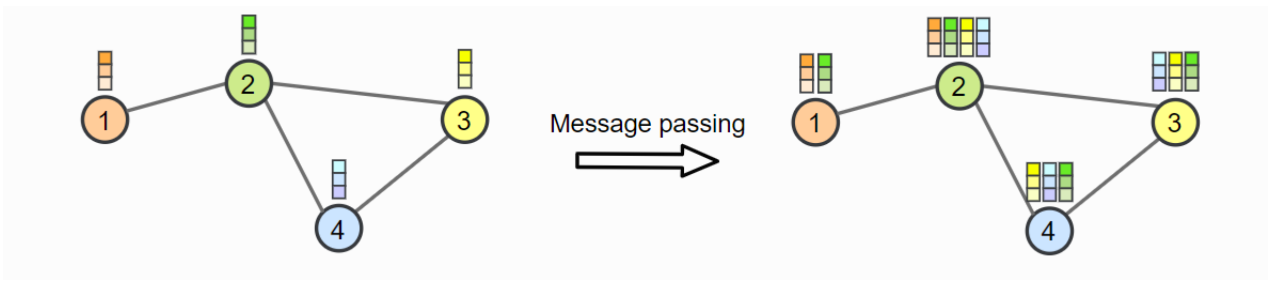

Figure 1: Message propagation and aggregation. The information is broadcast from different levels of depth of the graph. For each node, at each message passing step we collect information from the neighbors. In this way, adding message-passing steps can be seen as connecting nodes which can also be far from each other.

In order to do this, it is necessary to find a suitable representation of our graph. Graphs have different types of information that we shall potentially use to make predictions, such as nodes, edges, and connectivity. The first two features are relatively straightforward: for example, by means of nodes, we can form a node feature matrix $\mathbf{N}$ by assigning each node an index i and storing the feature for node i in $\mathbf{N}$. While these matrices can show a quite different nature, they can be processed without any special techniques.
However, representing graph connectivity is more complicated. Perhaps the most obvious choice would be to use an adjacency matrix since this is easily tensorisable. However, this representation would show some drawbacks: often the high number of nodes leads to very sparse adjacency matrices, which is space inefficient ($\mathcal{O}(n_{nodes}^2)$); moreover, many adjacency matrices can encode the same connectivity, as they are not permutation invariant with respect to the node labels. One elegant and memory-efficient way of representing sparse matrices is the edge connectivity matrix. It is a $n_{edges} \times 2$ matrix where each row k contains the indices of the source and destination nodes of the edge k. Using the edge connectivity matrix the space complexity reduces to $\mathcal{O}(n_{edges})$.

Now that we have all the ingredients to define a GNN, we can introduce a suitable graph representation of the mesh. We consider an undirected graph $G = (V, E)$, where $V = (v_i)_{i=1}^N$ is the set of vertices of the mesh, $E = (e_{ij})_{i,j=1}^N$ is the set of edges and $\mathbf{E}$ denotes the corresponding graph connectivity matrix. Unlike other types of graph data, such as social networks or citation graphs, computational meshes have a Euclidean structure, which means that every node i can be associated to its space coordinates $\mathbf{x}_i$. Often, every node is also associated with some features $\mathbf{v}_i$, which are also referred to as state variables. Among these features, we may include the solution of the system we are modeling together with some relevant information about the problem.

Since the solutions to PDEs, once an appropriate discretization has been introduced, can be seen as data defined on graphs (the computational mesh), we believe that these architectures can be useful in constructing surrogate models, as mentioned in Section 2. However, before delving into details, which will be presented extensively in Section 4, we take the opportunity to introduce a particular graph network model known in the literature for its expressive/approximation capabilities. This model is based on an architecture divided into three main blocks, called the Encoder, Processor, and Decoder, respectively. In the next section, the Encode-Process-Decode model is introduced and the *Message passing* step implementation is presented in detail.

3.1. The Encode-Process-Decode model

The architecture we will rely on to define a class of surrogate models for parametrized PDEs is the so-called Encode-Process-Decode architecture shown in Figure 2. It consists of an input encoding, then it performs M message passing steps through the Processor and ultimately combines a call to the Decoder, which outputs the model prediction. Before diving in every part of the network, it is convenient to introduce some notation; in particular:

- $\mathbf{v}_i \in \mathbb{R}^{d_{in}}$ are the features associated to each node i;
- $\mathbf{e}_{ij} \in \mathbb{R}^{e_{in}}$ are the features associated to each edge (i, j);
- $E_v : \mathbb{R}^{d_{in}} \to \mathbb{R}^l$, where $l > d_{in}$ is the latent dimension of the network, is the nodes features encoder;
- $E_e : \mathbb{R}^{e_{in}} \to \mathbb{R}^l$, where $l > e_{in}$, is the edges features encoder;
- $\mathcal{E} : \mathbb{R}^{3l} \to \mathbb{R}^l$ is the edges features processor;
- $\mathcal{N} : \mathbb{R}^{2l} \to \mathbb{R}^l$ is the nodes features processor;
- $\mathcal{D} : \mathbb{R}^l \to \mathbb{R}^{d_{out}}$, where d_{out} is the dimension of the output, is the decoder map.

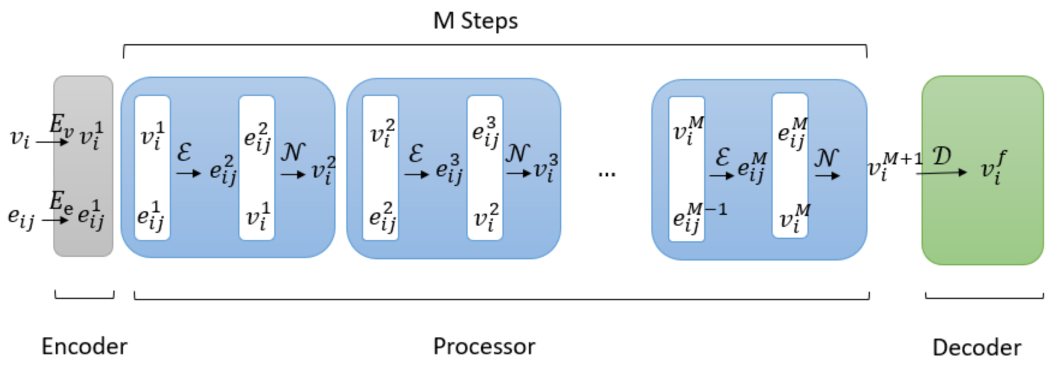
M Steps
$v_i \xrightarrow{E_v} v_i^1$
$e_{ij} \xrightarrow{E_e} e_{ij}^1$
v_i^1 $\mathcal{E}$ e_{ij}^2 $\mathcal{N}$ v_i^2
e_{ij}^1 e_{ij}^2 v_i^1
v_i^2 $\mathcal{E}$ e_{ij}^3 $\mathcal{N}$ v_i^3
e_{ij}^2 e_{ij}^3 v_i^2
...
v_i^M $\mathcal{E}$ e_{ij}^M $\mathcal{N}$
e_{ij}^{M-1} e_{ij}^M v_i^M
$v_i^{M+1} \xrightarrow{\mathcal{D}} v_i^f$
Encoder
Processor
Decoder

Figure 2: The Encode-Process-Decode model. The model is composed of 3 architectures: the Encoder embeds nodes and edges input features to higher dimensional spaces. The processor performs M message passing steps to allow connections between nodes far from each other, and the Decoder outputs the model prediction.

The functions $E_v, E_e, \mathcal{E}, \mathcal{N}, \mathcal{D}$ are modelled with Multi-Layer Perceptron (MLP) architectures [21], see Figure 3. An MLP architecture is a fully connected neural network in which at every layer a nonlinear function, called *activation function*, is applied to a linear transformation of the input vector, that is:

$$\mathbf{x}_{out} = f(\mathbf{W}\mathbf{x}_{in} + \mathbf{b}) \tag{5}$$

where $\mathbf{W}$ and $\mathbf{b}$ are respectively a learnable weights matrix and a learnable bias vector. However, the choice of the activation function f is relevant for the goodness of training and prediction; this topic is addressed in Section 5, where a series of numerical experiments is presented. As we can see, the degrees of freedom of the model only depend on the features' dimensions, so they are independent of the degrees of freedom of the mesh.

3.1.1. Encoder

The Encoder embeds mesh features into nodes features $\mathbf{v}_i$ and edge features $\mathbf{e}_{ij}$. In general, node features contain all the relevant dynamical quantities which are useful to understand the problem dynamics together

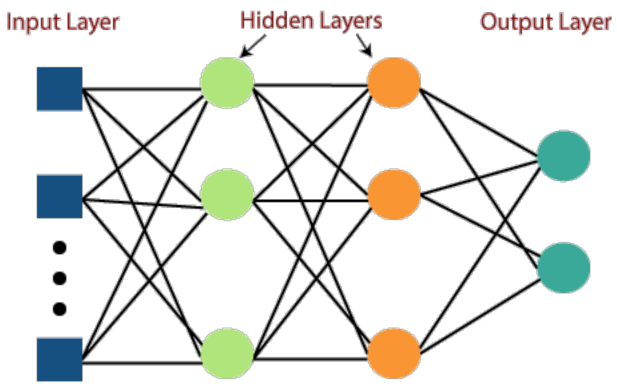

Figure 3: An example of a 2-hidden layers MLP

with a one-hot encoded vector indicating the boundary nodes. On the other hand, edge features contain the relevant geometrical information regarding the mesh such as the length of the edges $||\mathbf{x}_i - \mathbf{x}_j||_2$ and the L^2 norm of the coordinates $||\mathbf{x}_i||_2, ||\mathbf{x}_j||_2$. These edge features play an important role in capturing the local geometric characteristics of the mesh, which can provide useful information for solving partial differential equations on unstructured grids. In addition to the length of the edges, other relevant edge features may include the orientation of the edges and the angles between them. These features can be used to capture more complex geometric structures in the mesh, such as sharp corners or curved boundaries, which can significantly impact the behavior of the PDE solution. Furthermore, the edge features can be combined with the node features to provide a more comprehensive representation of the mesh, which can improve the accuracy of the PDE solution. The encoding is determined by the MLPs E_v and E_e, which expand input dimensions d_{in} and e_{in} to the latent dimension l, in order to extract as much information as possible from the input features.

3.1.2. Processor

The processor is the architecture that allows propagating the information along the edges, so it is the most important component of the learning process. The message propagation is repeated M times, where M represents the proximity, that is the number of hops, of the neighborhood which we are considering. This is a hyperparameter that has to be tuned according to the problem we are facing.

As we can see from Figure 2, the encoded node and edge features are first concatenated and passed to the model $\mathcal{E}$ in order to create new edge features which take into account both node input and edge input features. Note that the number of mesh edges is greater than the number of mesh nodes, so when we concatenate node and edge features we need to bring them to the same dimension. This is done by taking for each edge the features corresponding to the source node and to the destination node and concatenating them with the relative edge features. In this way, for each edge, we will have $3l$ features, where l is the latent dimension of the network. The MLP $\mathcal{E}$ then maps again them into dimension l. The new edge features are then combined with the input features to create a new node features tensor, which will be passed to the MLP $\mathcal{N}$. In this way, the new output node features are provided by a nonlinear function of both geometrical and dynamical information.

In the forward pass, it is necessary to reduce the edge features in order to match the number of mesh nodes. This reduction is performed with the `scatter_add` operation, see Figure 4. Given an input and an index tensor, this operation sums all the values of the input tensor which correspond to the same index value. The resulting output tensor has size equal to the number of indices in the index tensor. This operation can be also extended to multidimensional tensors by providing the axis in which the operation must be done. In the model, the operation is performed in the dimension corresponding to the number of the mesh edges, with the index tensor given by the destination nodes extracted from the edge connectivity matrix $\mathbf{E}$, which represents the connectivity of my mesh. Hence, we are summing all the edge features corresponding to edges having the same destination node.

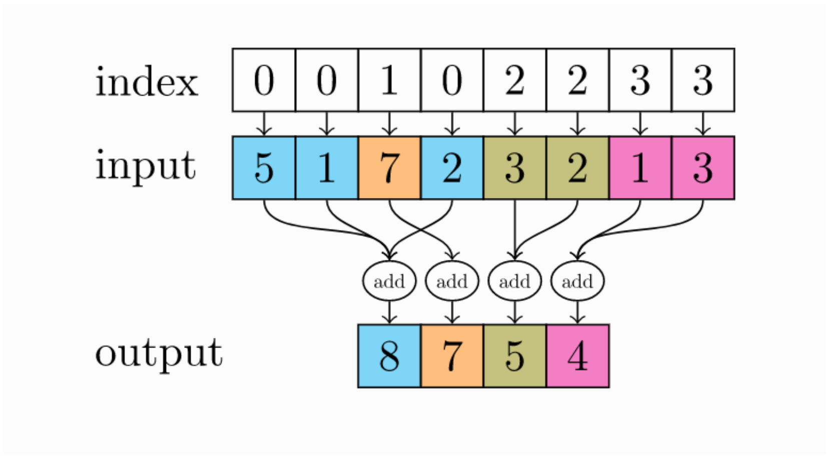

Figure 4: `scatter_add` operation. Given an input and an index tensor, it sums all the values of the input tensor which correspond to the same index value. The resulting output tensor has size equal to the number of indices in the index tensor. This operation can be also extended to multidimensional tensors by providing the axis in which the operation must be done.

3.1.3. Decoder

After performing M message passing steps, the node features tensor is passed to the final MLP, which decodes the relevant quantities we aim to predict based on the specific problem at hand. In particular, the values of these quantities for a given node i are dependent on the features of its neighbors and the coordinates of the relative nodes. In other words, these quantities can be expressed as a function of $\{\mathbf{v}_j\}_{j\in\mathcal{N}_i}$, $\{\mathbf{x}_j\}_{j\in\mathcal{N}_i}$, where $\mathcal{N}_i$ is the set of nodes in the neighborhood of node i, and a set of parameters $\boldsymbol{\mu}$, i.e., $F(\{\mathbf{v}_j\}_{j\in\mathcal{N}_i}, \{\mathbf{x}_j\}_{j\in\mathcal{N}_i}; \boldsymbol{\mu})$.
This approach enables us to capture the underlying dynamics of a physical system described by a time-dependent parametrized PDE. By encoding both the nodal features at a given time step t^n and the geometric properties of the mesh, we can then decode the system solution at time t^{n+1} as a function $F(\{\mathbf{v}_j^n\}_{j\in\mathcal{N}_i}, \{\mathbf{x}_j\}_{j\in\mathcal{N}_i}; \boldsymbol{\mu})$. As we will demonstrate in Section 5, this approach leads to accurate and robust predictions.

4. Application to surrogate modeling of parametrized PDEs

In our case, the goal is to predict an approximate solution $\tilde{\mathbf{u}}$ of the system at time t^{n+1} given the state of the system at time t^n for each node i, that is modeling a function Φ such that,

$$\begin{cases} \tilde{u}_i^{n+1}(\boldsymbol{\mu}) = \Phi(\{\mathbf{v}_j^n\}_{j\in\mathcal{N}_i}, \{\mathbf{x}_j\}_{j\in\mathcal{N}_i}; \boldsymbol{\mu}) \quad n = 0, \dots, N-1, \ i = 1, \dots, N_h \\ \tilde{u}_i^0(\boldsymbol{\mu}) = u_{h,i}^0(\boldsymbol{\mu}), \quad i = 1, \dots, N_h. \end{cases} \tag{6}$$

where $\mathcal{N}_i$ is the set of the nodes in the neighborhood of node i. Equation (6) is the nodal version of Equation (4), where the solution of the system at time t^n, i.e. $\tilde{u}_i^n$, is contained in the node features tensor $\mathbf{v}_i^n$ together with other relevant node features, and the dependence on the space coordinates is included to take into account the euclidean structure of the mesh.

Our approach aims to model the function Φ using an Encode-Process-Decode architecture that incorporates both nodal features at a specific time step t^n and the geometrical characteristics of the mesh. The architecture aggregates information from neighboring nodes, processes it, and decodes the system solution at time t^{n+1}. By doing so, we can evaluate Equation (6) independently of the number of nodes of the mesh, while simultaneously accounting for the graph structure of the mesh. This approach enables us to train the model using a variety of different geometries and subsequently predict solutions for new meshes that were not included in the training data.
This *inductive capability* is possible because GNNs first model the passing of the messages between neighboring nodes, and then the aggregation of the information at each node, in order to allow connections between nodes that are far from each other. Classical Feed Forward Neural Networks (FFNNs) are often prone to overfitting because they are fully connected, which means they may try to fit too closely to the training data and fail to generalize to new examples. GNNs, on the other hand, can help to mitigate overfitting by incorporating information from neighboring nodes and edges, which can improve the robustness of the model. However, CNNs can also be used to limit overfitting, but as we previously discussed, the convolution operator may struggle with

graph-based problems. Furthermore, both FFNNs and CNNs are still dependent on the number of nodes in the mesh, which may limit their generalization to different domains. Recently, a new class of architectures called Mesh-Informed Neural Networks (MINNs) has been introduced [7]. These architectures are specifically designed to handle mesh-based functional data and are capable of handling functional data defined on general domains of any shape better than classical FFNNs. Moreover, they can achieve reduced training times, lower computational costs, and better generalization capabilities. However, they require the number of nodes of the mesh N_h to be fixed, limiting their inductive capability. For this reason, to increase flexibility in generalizing to different domains, GNNs can be advantageous.

4.1. Training and testing algorithms

From now on when writing the model, for sake of simplicity, we consider only the dependency on the solution at the previous step $\tilde{\mathbf{u}}^n$, where the dependency on node i is implicit. To train the model, we need to create a dataset of simulations using a Full Order Model (FOM) such as the finite element method, where we vary the parameters $\boldsymbol{\mu}$. By doing so, we can obtain a set of accurate solutions, known as ground truth solutions, represented by $\mathbf{u}_h(t;\boldsymbol{\mu})$. These ground truth solutions will be also used to evaluate the performance of the model during the training phase.
Model training is performed by one-step prediction, that is at each time step t^n we compute the output $F(\mathbf{u}_h^n(\boldsymbol{\mu}))$ by passing to the network as input the ground truth solution $\mathbf{u}_h^n(\boldsymbol{\mu}) = \mathbf{u}_h(t^n;\boldsymbol{\mu})$. This preserves memory usage from recursive training and it also allows parallelization through batches, exploiting the power of GPUs with tensor calculus.
Training consists of the solution of an optimization problem for each batch of training data. Denoting with $\boldsymbol{\theta}$ the vector of parameters of the GNN, collecting all the corresponding weights and biases of each component of the GNN, the optimization problem consists of the minimization of the following weighted loss with respect to $\boldsymbol{\theta}$:

$$\begin{aligned}\mathcal{L}_{batch}(\boldsymbol{\mu},\boldsymbol{\theta}) = w_1 \frac{1}{N_{batch}-1} \sum_{n=1}^{N_{batch}-1} ||\dot{\mathbf{u}}_h^n(\boldsymbol{\mu}) - \dot{\tilde{\mathbf{u}}}^n(\boldsymbol{\mu},\boldsymbol{\theta})||_2^2 + \\ w_2 \frac{1}{N_{batch}-1} \sum_{n=1}^{N_{batch}-1} ||\mathbf{u}_h^{n+1}(\boldsymbol{\mu}) - \tilde{\mathbf{u}}^{n+1}(\boldsymbol{\mu},\boldsymbol{\theta})||_2^2\end{aligned} \tag{7}$$

where $\mathbf{u}_h^n(\boldsymbol{\mu}) = \mathbf{u}_h(t^n;\boldsymbol{\mu})$ with $t^1,\dots,t^{N_{batch}}$ as the time instants relative to the batch, $\dot{\mathbf{u}}_h^n(\boldsymbol{\mu})$ is the ground truth derivative estimated via finite differences, that is:

$$\dot{\mathbf{u}}_h^n(\boldsymbol{\mu}) \approx \frac{\mathbf{u}_h^{n+1}(\boldsymbol{\mu}) - \mathbf{u}_h^n(\boldsymbol{\mu})}{\Delta t}. \tag{8}$$

Moreover, $\dot{\tilde{\mathbf{u}}}^n(\boldsymbol{\mu},\boldsymbol{\theta}) = F(\mathbf{u}_h^n)$, that is the outcome of the network obtained by passing as input $\mathbf{u}_h^n(\boldsymbol{\mu})$ should approximate the temporal derivative of the solution at time t^n. For this reason, $\tilde{\mathbf{u}}^{n+1}(\boldsymbol{\mu},\boldsymbol{\theta})$ is the prediction calculated with the Forward Euler method

$$\tilde{\mathbf{u}}^{n+1}(\boldsymbol{\mu},\boldsymbol{\theta}) = \mathbf{u}_h^n(\boldsymbol{\mu}) + \Delta t \dot{\tilde{\mathbf{u}}}^n(\boldsymbol{\mu},\boldsymbol{\theta}). \tag{9}$$

In order to make our model robust to numerous rollout steps and avoid overfitting, we need to ensure that the training takes count of propagation errors, so the following strategies are performed:

- Random shuffling of the training data after each epoch;
- Gaussian noise added on data at each training step.

During the training phase (see Algorithm 1), the optimal parameters of the GNN are found by optimizing loss 7 through the back-propagation and ADAM algorithms. Moreover, after a specific number of epochs, or iterations, defined by the milestones hyperparameter, the learning rate is decreased by a factor $\gamma = 0.1$. The optimization algorithm updates the values of the model parameters in an iterative manner until a stopping criterion is met. In our case, we stop the training when the maximum number of epochs chosen for the algorithm is reached; this is a hyperparameter that needs to be tuned to achieve the best performance of the model.

After the training, we collect the optimal vector of parameters $\boldsymbol{\theta}^\star$, so that during the testing phase, the GNN is used to make predictions through a scheme that recalls the Forward Euler method:

$$\begin{aligned}&\tilde{\mathbf{u}}^{n+1}(\boldsymbol{\mu};\boldsymbol{\theta}^\star) = \tilde{\mathbf{u}}^n(\boldsymbol{\mu};\boldsymbol{\theta}^\star) + \Delta t F(\tilde{\mathbf{u}}^n;\boldsymbol{\mu},\boldsymbol{\theta}^\star) \quad n = 0,\dots,N-1 \\ &\tilde{\mathbf{u}}^0(\boldsymbol{\mu}) = \mathbf{u}_h^0(\boldsymbol{\mu})\end{aligned} \tag{10}$$

We remark that, during testing, only the initial ground truth solution is provided to the network, as, thanks to (10), the surrogate model is then capable of simulating a full rollout. At each time step, the known boundary conditions are set in order to guarantee a stable prediction of the phenomenon. The prediction error is computed as the relative MSE (RMSE) error between the network prediction and the ground truth solution:

$$RMSE(\tilde{\mathbf{u}}, \mathbf{u}_h) = \frac{1}{N} \sum_{n=0}^{N} \frac{\sum_{i=0}^{N_h-1} (\tilde{u}_i^n - u_{h,i}^n)_2^2}{\sum_{j=0}^{N_h-1} (u_{h,i}^n)_2^2} \tag{11}$$

Algorithm 1 Training Algorithm

Input: Network $\mathbf{F}$. Dictionary D containing N_{train} training snapshots list $\mathbf{U}$, edge connectivity matrices list $\mathbf{E}$, edge features matrices list $\mathbf{W}$ and inner nodes list $\mathbf{I}$. Starting learning rate ν. Milestones $\mathbf{m}$ for the learning rate decay. Number of training epochs max_epoch. Batch size N_{batch}. Noise variance σ^2. Timestep Δt.

Output: Optimal model parameters $\boldsymbol{\theta}$

```
 1: $epoch = 0$.
 2: Randomly initialize $\boldsymbol{\theta}^0$
 3: **while** $epoch < max\_epoch$ **do**
 4:   Create the list $indices = [1, \dots, N_{train}]$ and shuffle it randomly
 5:   **for** $sim$ in $indices$ **do**
 6:     $\mathbf{U}_{sim} = \mathbf{U}[sim]$, $\mathbf{U}_{sim} \in \mathbb{R}^{N_t \times N_h \times N_f}$ where $N_t$ is the total number of timesteps, $N_h = N_h(\boldsymbol{\mu})$ is the number of nodes of the mesh and $N_f$ is the number of node features.
 7:     $\mathbf{E}_{sim} = \mathbf{E}[sim]$, $\mathbf{E}_{sim} \in \mathbb{R}^{N_{edges} \times 2}$.
 8:     $\mathbf{W}_{sim} = \mathbf{W}[sim]$, $\mathbf{W}_{sim} \in \mathbb{R}^{N_{edges} \times N_e}$ where $N_e$ is the number of edge features.
 9:     $\mathbf{I}_{sim} = \mathbf{I}[sim]$, $\mathbf{I}_{sim} \in \mathbb{R}_h^N$ with $\mathbf{I}_{sim}[i] = 1$ if node $i$ is an inner node, 0 otherwise.
10:     $b = 0$
11:     **while** $b < N_t$ **do**
12:       $\mathbf{U}_{batch} = U_{sim}[b : b + N_{batch}]$
13:       Create noise tensor $\boldsymbol{\Sigma} = \sigma \mathbf{Z}$ where $\mathbf{Z} \in \mathbb{R}^{N_{batch} \times N_i \times N_f}$ is a random tensor with $N_i$ inner nodes.
14:       Initialize $\mathbf{U}_{noise} = \mathbf{U}_{batch}$
15:       $\mathbf{U}_{noise}[:, I_{sim}] += \Sigma$
16:       Calculate target derivative $\mathbf{U}_{dot} = (\mathbf{U}_{batch}[1 :] - \mathbf{U}_{noise}[: -1])/\Delta t$
17:       Make a forward pass through the network $\mathbf{F}_{net} = \mathbf{F}(\mathbf{U}_{noise}, \mathbf{E}_{sim}, \mathbf{W}_{sim})$.
18:       Calculate network solution $U_{net} = \mathbf{U}_{noise}[: -1] + \Delta t \mathbf{F}_{net}$
19:       Calculate training loss $\mathcal{L}_{batch}$
20:       Back-propagation through the net and parameters update: $\boldsymbol{\theta}^1 = ADAM(\nu, \boldsymbol{\theta}^0)$.
21:       $\boldsymbol{\theta}^0 = \boldsymbol{\theta}^1$
22:       $b \leftarrow b + N_{batch}$
23:     **end while**
24:   **end for**
25:   **if** $epoch \in \mathbf{m}$ **then**
26:     Decay the learning rate by a factor $\gamma = 0.1$
27:   **end if**
28:   $epoch \leftarrow epoch + 1$
29: **end while**
    Pick the last weights updated $\boldsymbol{\theta}^1$
```

5. Numerical experiments

In this section, the model introduced in the previous sections is applied to an advection-diffusion problem in 3 different cases: (i) in a 2D square with a circular obstacle with a time-varying advection term; (ii) in a 2D Stokes flow in proximity of bump; (iii) in a 3D Stokes flow around a cylinder.

All the examples are studied in domains in which we have let the obstacle vary its position, then the first two examples are also studied in domains where the obstacle varies both its position and dimension. In this way, we have exploited the model capacity to infer the geometrical features of the mesh.

5.1. Advection-Diffusion problem in a square domain with a circular obstacle

In this section, the model is applied to the advection-diffusion problem:

$$\begin{cases} \frac{\partial u}{\partial t} - D\Delta u + \mathbf{b} \cdot \nabla u = 0 & \text{in } \Omega \times (0,T] \\ u(x,y) = (x-1)^2 + (y-1)^2 & \text{on } \partial\Omega \times (0,T] \\ u_0(x,y) = (x-1)^2 + (y-1)^2 & \text{in } \Omega \end{cases} \tag{12}$$

where $\Omega = (0,1)^2 \setminus C$, with $C = \{(x,y): \ (x-c_x)^2 + (y-c_y)^2 \leq (0.15)^2\}$. In our simulations, we parametrize the center of the circle as $\boldsymbol{\mu} = (c_x, c_y) \subset \mathcal{P}$ where $\mathcal{P} = \{(x,y): \ 0 < x < 1, \ y \geq 0.5\}$, and we let that vary for the generation of the training data. We also set $T = 2$, $D = 0.1$ and $\mathbf{b} = [1-t, 1-t]$ for $t \in [0,2]$. The time-varying advection parameter $\mathbf{b}$, allows us to observe a more complex phenomenon during the time horizon considered with respect to the case in which $\mathbf{b}$ is a steady vector field.
The ground truth FOM simulations of Problem (12) are obtained by first discretizing in space (following Equation (1)) using P1 Continuous Galerkin Finite Element Space and imposing Dirichlet boundary conditions. Then, we discretize in time using the Backward Euler Method following Equation (3). Two examples of FOM solutions are reported in Figure 5.

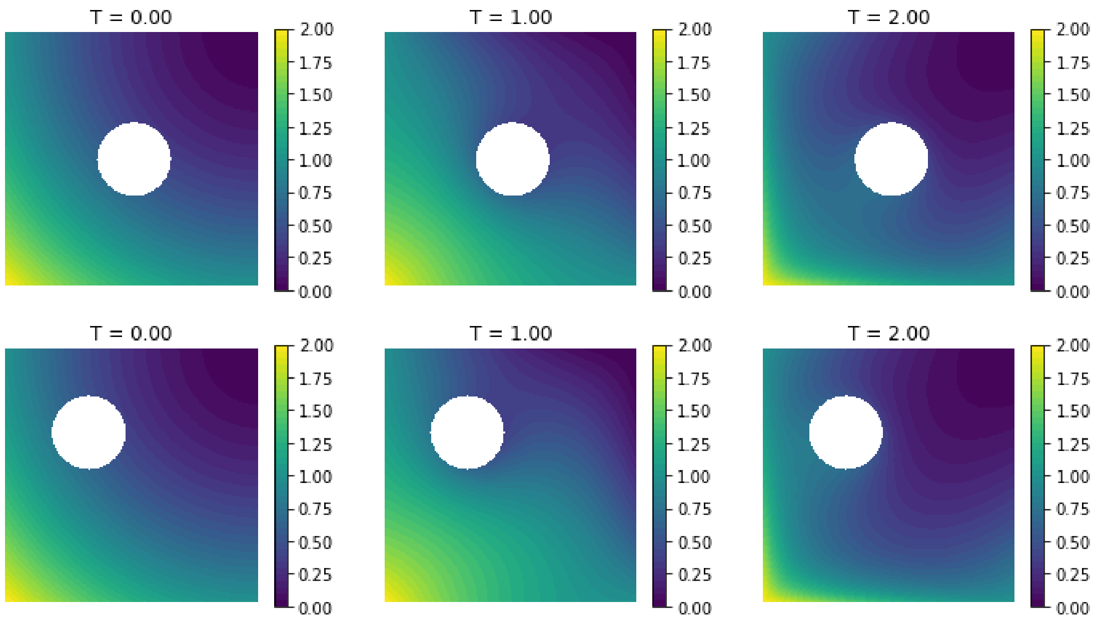

Figure 5: Test case 1, Advection-Diffusion problem, FOM solutions. Both rows represent 3 time steps of simulation. In the first row, the center of the obstacle is at $\boldsymbol{\mu} = (0.5, 0.5)$, while in the second row is at $\boldsymbol{\mu} = (0.32, 0.68)$.

5.1.1. Problem data

We collected a dataset composed of 100 simulations, each obtained for a different position of the center of the obstacle, with a number of mesh nodes varying from 770 to 790. The time step chosen is $\Delta t = 0.02$, resulting in 101 time snapshots for each simulation.

Before training the model, we need to define the input features, that is the information to pass to the network at time t^n, and the output features, that is the quantity we want to predict with the network at time t^n. In this example, the node input is a $N_{batch} - 1 \times N_h \times 3$ tensor, where N_{batch} is a hyperparameter, N_h varies across the simulations, and for each node $i = 1, \dots, N_h$ we have 3 features: the ground truth solution u_i^n at node i; the time instant t^n and a binary variable with value 1 if the corresponding node is on the boundary and 0 otherwise. Another input information to provide is the edge attributes tensor: this is a $N_{batch} - 1 \times N_{edges} \times 3$ tensor containing for each edge, the coordinates of its nodes and its length calculated as the Euclidean norm of the distance of the coordinates.
The output, instead, is a $N_{batch} - 1 \times N_h \times 1$ tensor which represents the derivative $\dot{\tilde{u}}(t)$ evaluated at time t^n. Here the N_{batch} hyperparameter allows calculating in parallel different outputs corresponding to consecutive time instants, that is for a given batch of time instants $b = t^1, \dots, t^{N_{batch}}$, we pass simultaneously the network the input tensors from t^1 to time $t^{N_{batch}-1}$ and the net gives as output the derivative of the solution at the same time instants. In this problem, we use $N_{batch} = 25$, which is a good compromise between the speed and effectiveness of the training.
There are also other hyperparameters to provide, which we summarize below:

- $l = 32$, where l is the latent dimension of the network;
- noise variance $\sigma^2 = 10^{-6}$;
- total number of epochs $max\_epoch = 1500$;
- learning rate $\nu = 10^{-3}$ with decay $\gamma = 0.1$ every 500 epochs;
- SiLU ($\text{silu}(x) = \frac{x}{1+e^{-x}}$) is chosen as activation function for each MLP layer because we have to guarantee high-order differentiability to model derivatives;
- number of MLP layers $mlp\_layers = 2$;
- message Passing steps $mp\_steps = 12$.

The training set is composed of 80 simulations, while the test set has 20 simulations, both chosen randomly among the 100 FOM simulations run.
For each batch, the loss is computed using only the MSE between the derivative computed by the network and the ground truth derivative calculated with the finite differences scheme presented in Section 4. In this case, we set the loss weights as $w_1 = 1$ and $w_2 = 0$, so we do not have the term that involves Forward Euler prediction in the loss function.

5.1.2. Numerical results

The results of the rollout predictions of the test simulations are summarized in Table 1. As we can see, all the predictions RMSEs are of order 10^{-3} or 10^{-4}. Moreover, our model outperforms significantly the ground truth solver in the simulation time at testing stage. The best prediction, together with its ground truth solution, is shown in Figure 6.

The dynamic of the problem is well predicted and no propagation errors are spotted. Hence, our model and the training strategy adopted seem to be in principle good at solving problems concerning evolutionary PDEs in which multiple rollout time steps need to be predicted.

Test case 1. Advection-Diffusion problem

	RMSE (mean)	**RMSE (max)**	**RMSE (min)**	$\mathbf{t_{gt}(s)}$	$\mathbf{t_{pred}(s)}$
AD 1	1.2×10^{-3}	6.1×10^{-3}	4×10^{-4}	≈ 159.8	≈ 9.83

Table 1: Test case 1. Advection-Diffusion problem. Results of the test set predictions.

However, it is worth showing that the model predicts well also simulations in which the position of the obstacle changes, leading to different dynamics in the trajectories.

In Figure 7, the predicted solution in the case where the obstacle is closer to the source is reported. Despite the trajectories being different from the others seen before, the directions of propagations are always accurately captured. The RMSE plot shows that, although the errors are larger on the boundary in the first rollout phase ($T < 1.00$), in the end they are higher around the obstacle, where the change of direction of the advection vector $\mathbf{b}$ makes the prediction of the solution more difficult.

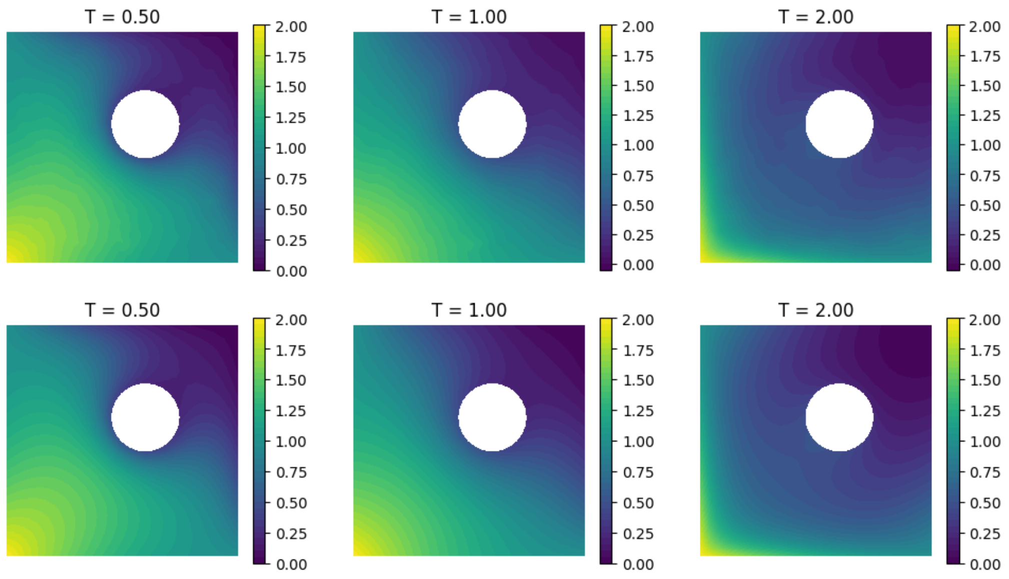

Figure 6: Test case 1, Advection-Diffusion problem, best model prediction ($\boldsymbol{\mu} = (0.65, 0.63)$). First row: rollout prediction. Second row: ground truth solution .

We highlight that a GNN-based approach follows a *local-to-global* generalization approach, first processing information locally through the Encoder, and then aggregating what has been processed from the neighborhood to connect the mesh nodes. However, this approach may result difficult for some trajectories and yield worse predictions at critical points of the mesh, such as along the boundary. The lack of smoothness in PDE solutions is a common challenge faced by deep learning-based surrogate models, as neural networks are known to struggle with capturing such properties.

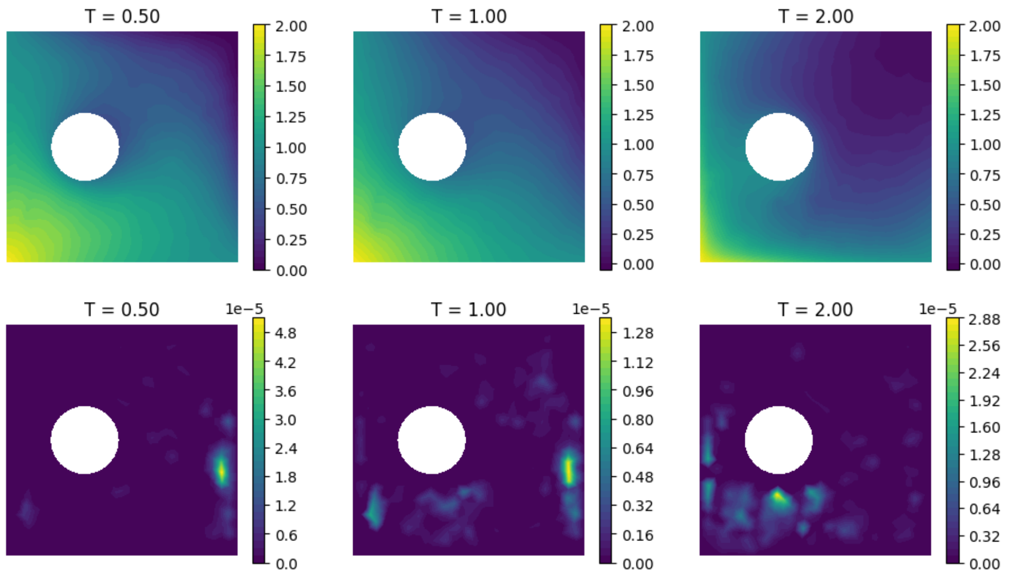

Figure 7: Test case 1, Advection-Diffusion problem. Prediction obtained for $\boldsymbol{\mu} = (0.29, 0.5)$ with the obstacle close to the source. First row: rollout prediction. Second row: RMSE related to each time step between the prediction and the corresponding ground truth solution.

The prediction which leads to the worst RMSE over all the tests that have been carried out according to

Table 1 is shown in Figure 8. The prediction in any case is good near the obstacle and the source; however, after 100 rollout time steps, the prediction gets worse along the boundary of the domain as we can see also from the plots of the RMSE for different time steps.
Hence, even if the overall predicted dynamics is good, the change of the solution after $T = 1.00$ due to the change of the direction of the advection vector $\mathbf{b}$ causes an error propagation which results in a bad node prediction near the lower edge of the boundary in the ending trajectories.
The plots of the errors show this trend, even if we can keep the order of magnitude of the RMSE in each snapshot still near 10^{-5}.

In this situation, the position of the obstacle plays a crucial role in the accuracy of the prediction because the phenomenon is inherently more challenging to model. Additionally, the problem can be compounded by an initial error that propagates throughout the entire simulation time, which can make it even harder for the model to correctly predict the outcome. Although the model is trained to be robust to small perturbations in the input, unexpected perturbations can still cause it to fail.

To address these challenges, reducing the batch size during training can be a potential solution, as it allows for a slower learning process and reduces the likelihood of overfitting on noisy data. However, this can also lead to reduced model performance, and the majority of good predictions may suffer as a result. Therefore, a balance must be struck between reducing overfitting and maintaining the model's overall predictive power. Future research could explore other ways to mitigate the impact of initial errors and unexpected perturbations on GNN-based simulation

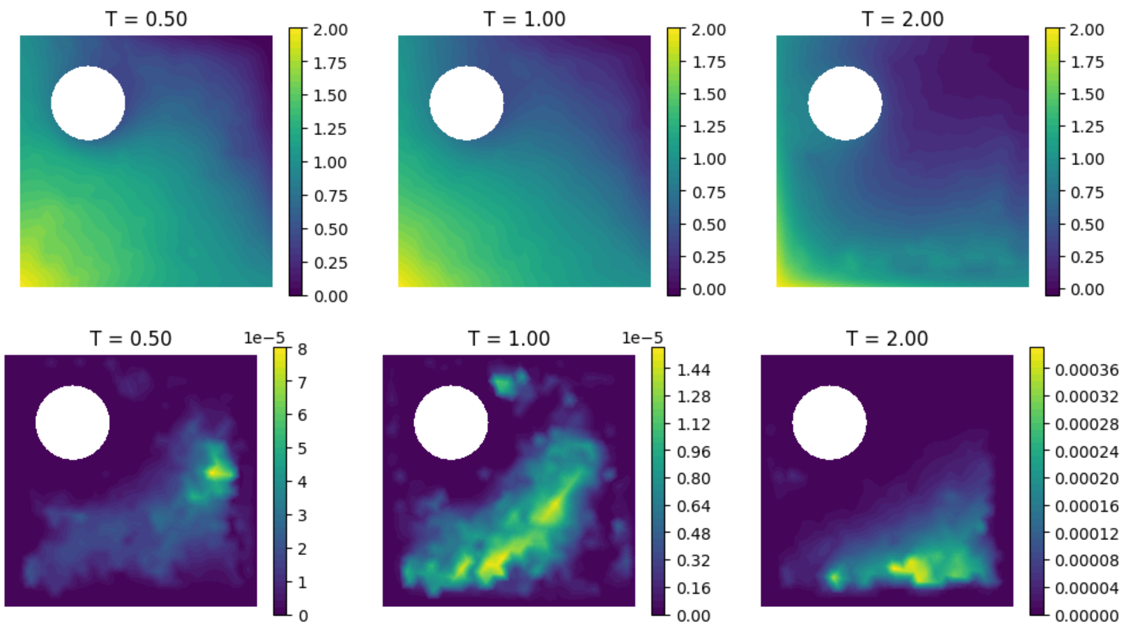

Figure 8: Test case 1, Advection-Diffusion problem. Prediction obtained for $\boldsymbol{\mu} = (0.25, 0.75)$ with the obstacle on the top left corner. First row: rollout prediction. Second row: RMSE related to each time step between the prediction and the corresponding ground truth solution.

Another key aspect to analyze is the trend of the L^2 relative error as the simulation time varies. This error represents, at each time step, the L^2 error between the prediction and the FOM solution divided by the L^2 norm of the FOM solution. In Figure 9, on the left, we can see 3 curves that represent the evolution of the L^2 error between our solution and the ground truth over the time interval $[0, T]$. The curves show respectively how the first, second, and third quantiles of the error vary in time. The trend is coherent with the results previously shown, indeed we can notice a substantial increase in the error as the simulation time increases, and in particular, the final instants strongly influence the RMSE of the predictions. The first quantile and the median are similar, while the third shows more oscillations in the error. However, for most of the simulation time, we have good bounds for the error, which is kept around 10^{-3}.

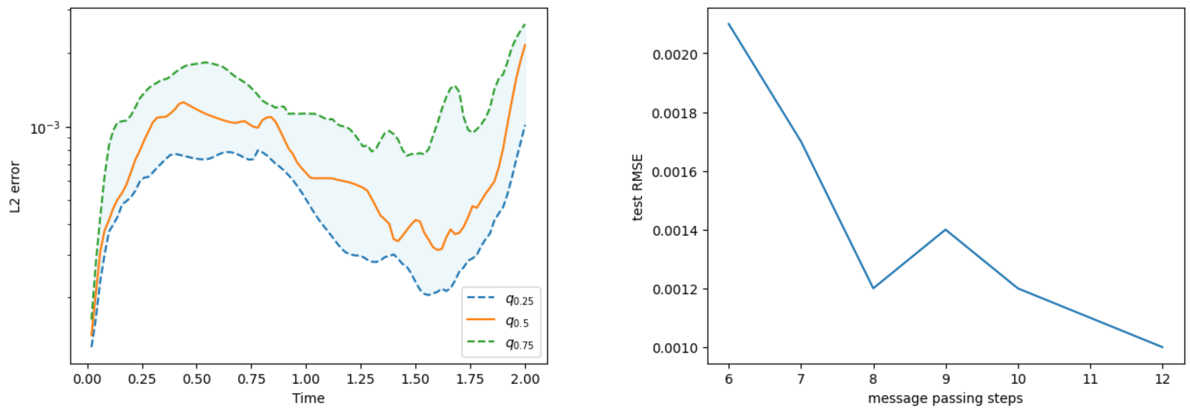

Figure 9: Test case 1, Advection-Diffusion problem. Left: L^2 relative error vs Time plot. The dashed lines represent the first and the third quantiles of the L^2 errors among all the test predictions, while the orange line is the median. The shaded area can be considered a confidence region for the simulation error. Right: Test RMSE vs message passing steps.

Despite the anomaly shown in Figure 8, it is worth noticing that the model is completely mesh-independent, and for the majority of trajectories provides a good prediction for long rollouts, keeping the number of parameters to a surprisingly small number. Indeed for this example, the model has 91009 parameters, which is easy to reach if you try to solve a similar problem with a fully connected network.

5.1.3. The *message passing steps* hyperparameter

Among all the hyperparameters, the one most influencing the goodness of the model is the number of message-passing steps. This number represents how much in-depth we look at the neighborhood when we propagate the message. A small number of message-passing steps may result in underfitted areas of the mesh, while a big one will slow down the training, increasing too much the number of parameters, possibly yielding overfitting.

This parameter has been tuned by hand after having tried multiple values and the best one resulted to be 12. A plot of the corresponding results found can be seen in Figure 9, on the right. The test RMSE reaches a local minimum for 8 message-passing steps. This is the best choice if we want to keep control of the number of total parameters of the network, which are only 61825 in this case.
However, in the experiments made with higher message-passing steps, we have a better approximation of the trajectories for almost all the simulations and the number of parameters does not grow too much.

5.1.4. Generalization to obstacles with different dimensions

This advection-diffusion example can also be extended to domains in which both the position and the dimension of the obstacle change. We slightly modify our training dataset by adding new simulations in which the obstacle has both a smaller and a higher radius than before and this will improve the generalization of the model. Hence, the geometrical parameter now is $\boldsymbol{\mu} = (c_x, c_y, r) \in \mathcal{P} = \{(x, y) : \ 0 < x < 1, \ y \geq 0.5\} \times \{0.1, 0.15, 0.2\}$.
Again, we test the model on new simulations which have varying obstacle positions and dimensions. In Figures 10 and 11 the predictions for two new test simulations can be seen. The model can generalize well on this problem also if the geometries differ a lot from each other. Moreover, there is no need to increase the number of message-passing steps so the number of parameters can be kept under control.

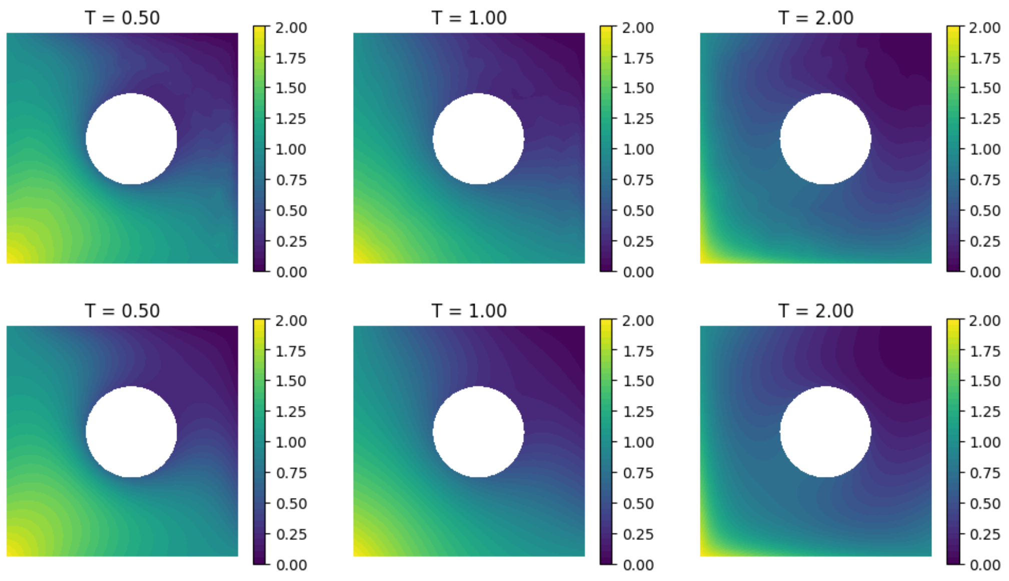

Figure 10: Test case 1, Advection-Diffusion problem. Prediction obtained for $\boldsymbol{\mu} = (0.55, 0.53, 0.2)$. First row: rollout prediction. Second row: ground truth solution.

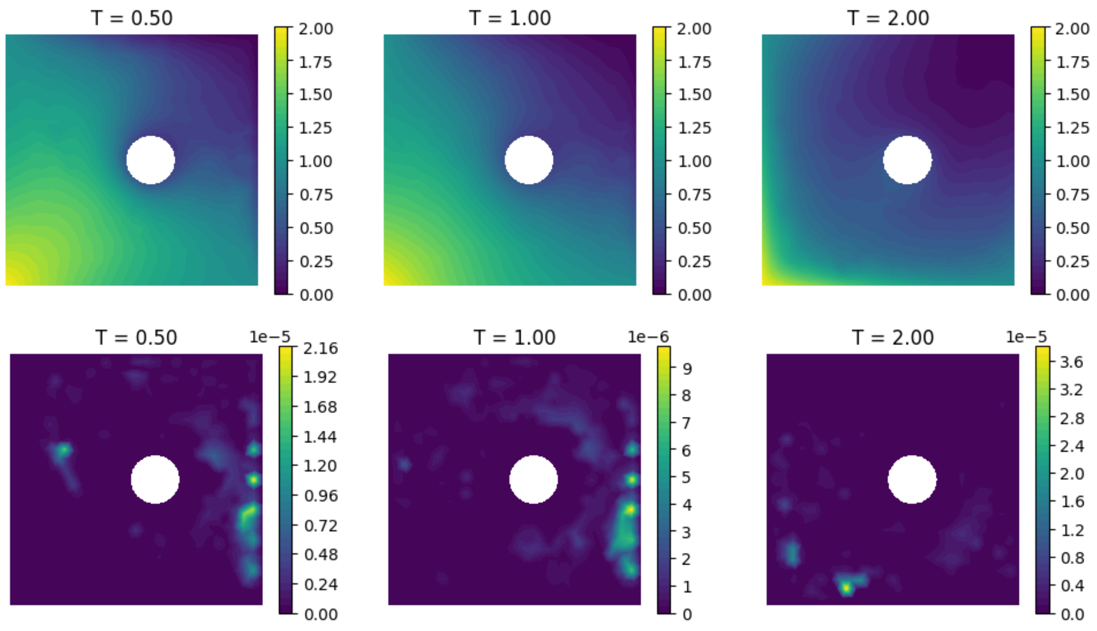

Figure 11: Test case 1, Advection-Diffusion problem. Prediction obtained for $\boldsymbol{\mu} = (0.6, 0.52, 0.1)$. First row: rollout prediction. Second row: RMSE related to each timestep between the prediction and the corresponding ground truth solution.

An important question that arises is whether our model can predict solutions where the obstacle has a different shape, without requiring retraining of the network. To investigate this, we present an example in Figure 12 of a prediction obtained with a square obstacle located in the top right of the domain, that is corresponding to the parameters $\boldsymbol{\mu} = (r_x, r_y, L) = (0.7, 0.7, 0.3)$, where r_x and r_y are the coordinates of the center of the square, and L is the length of its edge. Surprisingly, the errors, in this case, are of the same order of magnitude as those discussed earlier, and the prediction of the overall dynamics is remarkably accurate. This result is attributed to the ability of the model to understand different geometries by means of its inductive structure. GNNs, in particular, can automatically incorporate the geometrical structure of the domain by utilizing both the edge connectivity matrix and the edge features. However, some difficulty is observed in handling the nodes surrounding the obstacle, especially at the corners, but this does not appear to affect the overall accuracy of the prediction. These findings suggest that our model has the potential to generalize well to other geometries, without the need for extensive retraining, thus enhancing its practical applicability in real-world scenarios.

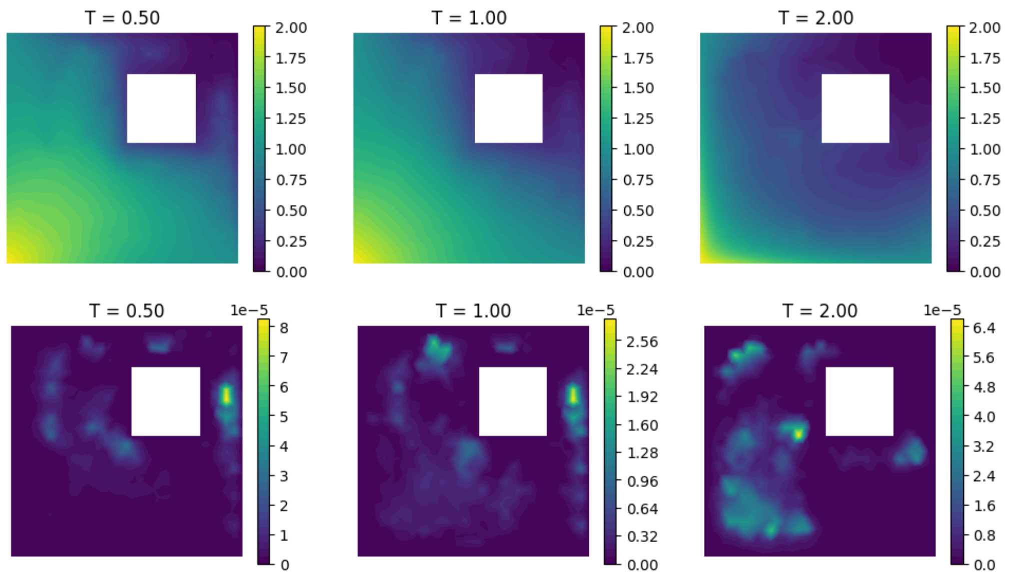

Figure 12: Test case 1, Advection-Diffusion problem. Prediction obtained for $\boldsymbol{\mu} = (0.7, 0.7, 0.3)$ with a square obstacle. First row: rollout prediction. Second row: RMSE related to each timestep between the prediction and the corresponding ground truth solution.

5.2. Advection-Diffusion problem in a 2D Stokes flow in proximity of a bump

We want to solve an advection-diffusion problem like (12), but now the advection coefficient $\mathbf{b}$ is obtained by solving the Stokes problem:

$$\begin{cases} -\nu\Delta\mathbf{b} + \nabla p & = 0 \qquad \text{in } \Omega \\ \nabla \cdot \mathbf{b} & = 0 \qquad \text{in } \Omega \end{cases} \tag{13}$$

where p is a pressure field and the boundary conditions are given by:

$$\mathbf{b} = 0 \qquad \text{on } \Gamma_D \tag{14}$$

$$\mathbf{b} = \mathbf{b_{in}} \qquad \text{on } \Gamma_{in} \tag{15}$$

$$\nu\frac{\partial\mathbf{b}}{\partial\mathbf{n}} - p\mathbf{n} = 0 \qquad \text{on } \Gamma_N \tag{16}$$

where

$$\mathbf{b_{in}} = (\frac{40Uy(0.5-y)}{0.5^2}, 0), \quad U = 0.3, \quad \nu = 10^{-3} \tag{17}$$

$\mathbf{b_{in}}$ represents the value of $\mathbf{b}$ at the inflow Γ_{in} and Γ_D and Γ_N are respectively the Dirichlet side and Neumann outflow boundaries.
The domain Ω is the rectangle $(0,1)\times(0,0.5)$ with a bump in the upper edge. Here $\Gamma_{in} = \{x = 0\}$, $\Gamma_D = \{y = 0\} \cup \{y = 0.5\}$ and $\Gamma_N = \{x = 1\}$. During our simulations, we shift the position of the bump in a way that its center c_x varies from 0.35 to 0.65. Hence, we have $\mu \in \mathcal{P} = [0.35, 0.65]$. Regarding the Advection-Diffusion problem, at the inflow Γ_{in}, we impose the Dirichlet boundary condition $u_{in}(x,y) = \frac{4y(0.5-y)}{0.5^2}$, which is also the initial condition, on Γ_D we impose no-slip boundary condition and on Γ_N we set $\frac{\partial u}{\partial n} = 0$. The final simulation time is $T = 0.5$ and $D = 0.01$. Our results focus only on the approximation of the solution u of the Advection-Diffusion problem and an example of a FOM solution is reported in Figure 13.

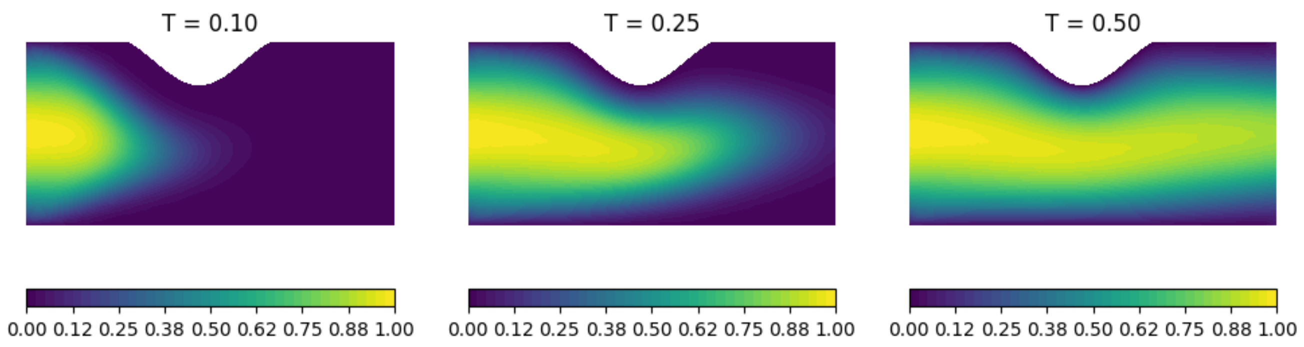

Figure 13: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. 3 timesteps of a FOM simulation. The center of the bump here is at $\mu = 0.4696$.

5.2.1. Problem data

Our dataset is composed of 125 simulations, each obtained for a different position of the bump so the number of mesh nodes varies from 937 to 1042. The time step chosen is $\Delta t = 0.01$, resulting in 51 time snapshots for each simulation. We consider the same node and edge input features of the previous example. For this example, the loss function is slightly different from the previous test case, indeed now the two terms defined in Section 4 are equally weighted ($w_1 = 0.5, w_2 = 0.5$).
Most of the hyperparameters are the same as the ones chosen before, that is:

- $l = 32$, where l is the latent dimension of the network;
- noise variance $\sigma^2 = 10^{-6}$;
- total number of epochs $max\_epoch = 3000$;
- learning rate $\nu = 10^{-3}$ with decay $= 0.1$ after 500 and 1000 epochs;
- SiLU is chosen as activation function for each MLP layer;
- number of MLP layers $mlp\_layers = 2$;
- message passing steps $mp\_steps = 15$.

The training set is composed of 100 simulations while the test set includes 25 simulations, both chosen randomly among the 125 FOM simulations.

5.2.2. Numerical results

The results of the rollout predictions of the test simulations are summarized in Table 2. In this more complex problem, the RMSEs are higher than the ones in the previous example, but we still outperform the ground truth solver in terms of time efficiency.

Advection-Diffusion problem in a 2D Stokes flow

	RMSE (mean)	**RMSE (max)**	**RMSE (min)**	$\mathbf{t_{gt}}$**(s)**	$\mathbf{t_{pred}}$**(s)**
AD 2	1.64×10^{-2}	7.35×10^{-2}	1.2×10^{-3}	≈ 115.65	≈ 7.51

Table 2: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. Results of the test set predictions.

The best prediction, together with its ground truth solution, is shown in Figure 14. The dynamic predicted is very accurate since we do not spot any propagation error. The prediction seems to get worse in some nodes which are either close to the bump or to the upper edge, in which we have imposed no-slip boundary conditions.

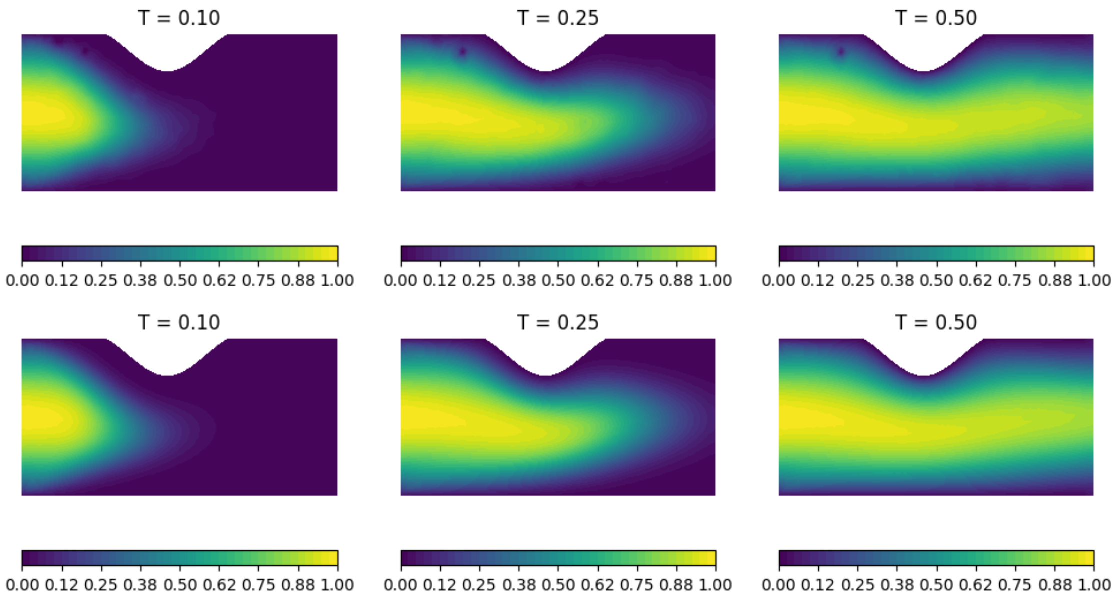

Figure 14: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. Best model prediction ($\mu = 0.42$). First row: rollout prediction. Second row: ground truth solution.

The model performs well in predicting simulations in which the bump has different positions, as shown in Figure 15. However, some numerical errors can be observed in nodes close to the inflow, which are higher compared to the other nodes, but they are kept under control and remain of the order 10^{-5} as shown in the plot. It is worth recalling that these initial errors tend to fade out, and the accuracy improves as the simulation evolves. This behavior is particularly evident when the bump is not located too close to the inflow, indeed higher errors are usually caused by particular positions of the bump.

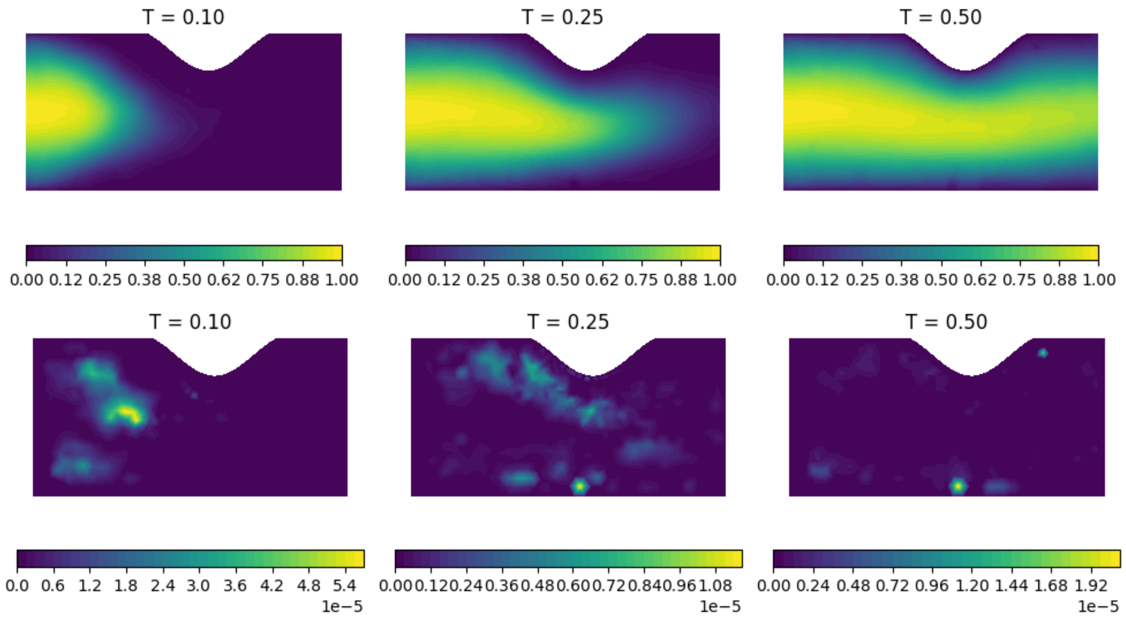

Figure 15: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. Prediction obtained for $\mu = 0.58$ with the bump on the right part of the upper edge. First row: rollout prediction. Second row: RMSE related to each timestep between the prediction and the corresponding ground truth solution

If the position of the bump is too close to the inflow, it heavily influences the prediction in the initial time steps. As a result, fixing the error in some nodes becomes increasingly difficult as time evolves. This can be seen quite well in Figure 16, which represents a worst-case scenario for our predictions. The behavior of the nodes around the inflow has a significant impact on the accuracy of the simulation, as any error arising in this region can propagate throughout the domain. The proximity of the bump to the inflow also plays a crucial role in the self-adjustment of the simulation. Despite these challenges, the model overall performs very well, as demonstrated by the accuracy of its predictions.

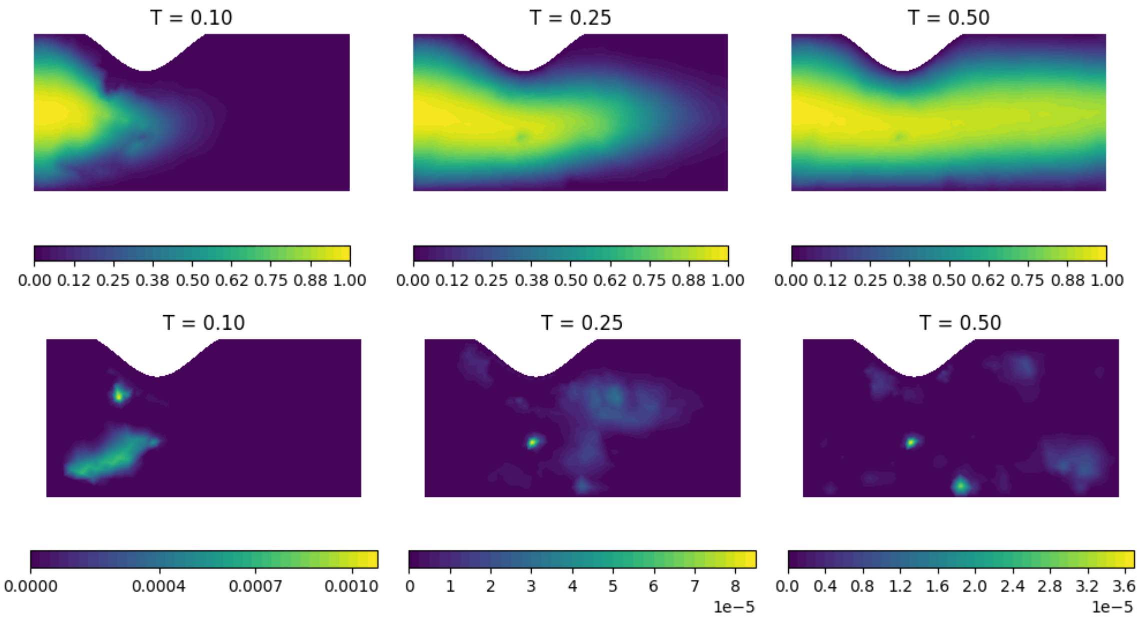

Figure 16: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. Worst case scenario: bump close to the inflow ($\mu = 0.355$). First row: rollout prediction. Second row: RMSE related to each timestep between the prediction and the corresponding ground truth solution

Our qualitative considerations are also supported by the plot in Figure 17, which reports the behavior in time of the L^2 relative error between the prediction and the FOM solution. The prediction is more difficult in the first time instants when the inflow and the bump position determine the system dynamics. The third quantiles are influenced by the worst-case scenario. Finally, the model is able to self-adjust, that is the errors tend to decrease as the time of the simulation evolves. This is an important quality since it means that it is robust to the presence of noise during the simulation.

Furthermore, the decrease of the L^2 relative error in time indicates that the model is able to capture the system behavior more accurately at later time instants. This is expected since the initial condition and boundary conditions are better defined and the system behavior becomes more predictable as time evolves. Moreover, the third quantile in the plot shows that the model worst-case error is still relatively small, hence the model is robust to unexpected perturbations during the simulation. This is a desirable property since real-world problems often have some degree of uncertainty or noise, and a model that can handle such situations is more likely to be useful in practice.

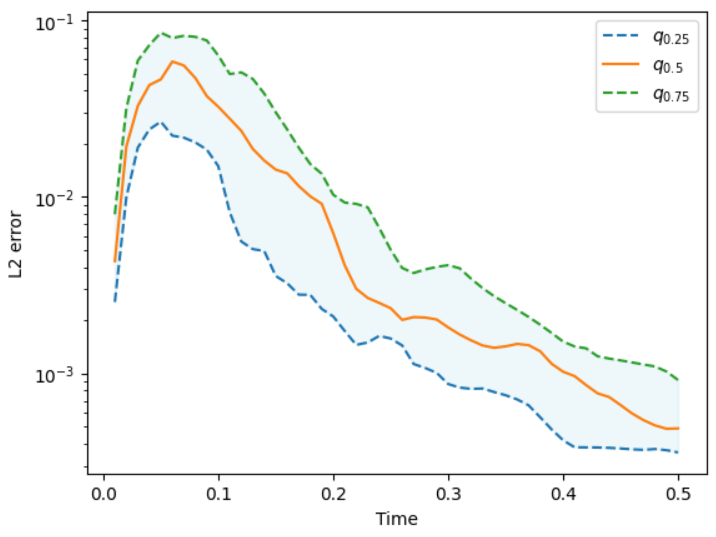

Figure 17: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. L^2 relative error vs Time plot: The dashed lines represent the first and the third quantiles of the L^2 errors among all the test predictions, while the orange line is the median. The shaded area can be seen as a confidence region for the simulation error.

5.2.3. Generalization to bumps with different positions and dimensions

This example can be generalized by letting the bump also vary also along the lower edge, and its dimension change. Hence, we consider a new dataset composed of 185 simulations in which the height of the bump might be in the set $h = \{0.08, 0.12, 0.175\}$ and its center can vary along both the upper and lower edge in the interval $[0.4, 0.6]$. In this way, we keep the bump sufficiently far from both the inflow and the outflow since we have previously seen that this may imply numerical errors in the GNN prediction. Hence, the geometrical parameter now becomes $\boldsymbol{\mu} = (c_x, c_y, h) \in \mathcal{P} = [0.4, 0.6] \times \{0., 0.5\} \times \{0.08, 0.12, 0.175\}$.

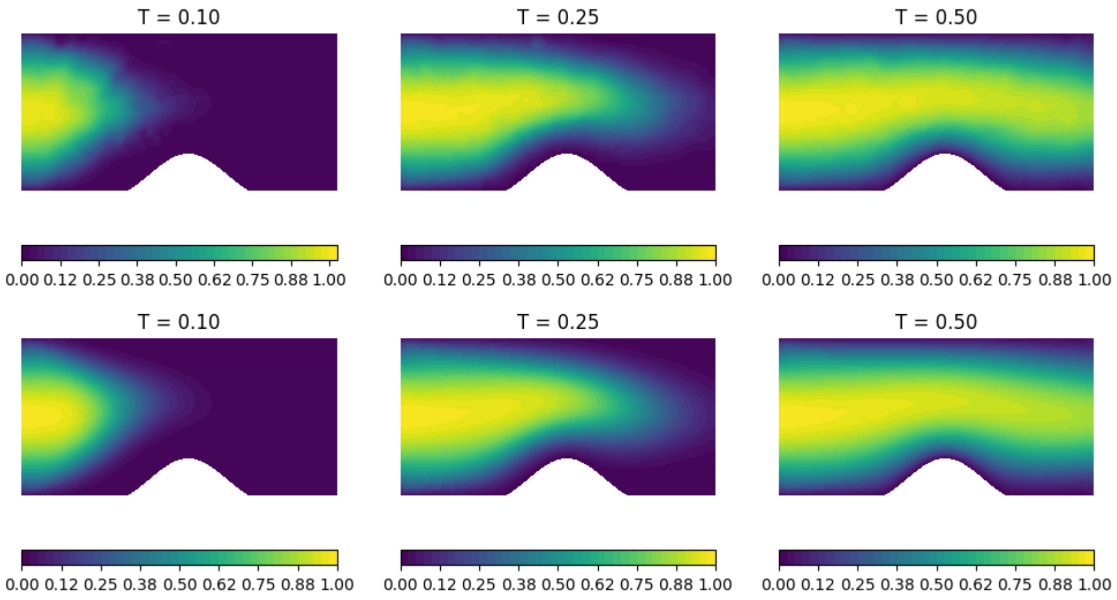

Figure 18: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. Prediction obtained for $\boldsymbol{\mu} = (0.528, 0, 0.12)$ with the bump in the lower edge with center at $x = 0.528$ and height $h = 0.12$. First row: rollout prediction. Second row: ground truth solution.

The results show that the implemented GNN-based can learn correctly the geometry of the problem even if we let the domain vary a lot in our dataset. This is important since we can overcome overfitting, which is common when using deep neural networks. In the following lines, some predictions of rollout simulations not seen during training are presented. In Figure 18, the prediction computed by the GNN when the bump is on the lower edge with height $h = 0.12$ and center in $x = 0.528$, that is $\boldsymbol{\mu} = (0.528, 0, 0.12)$, is reported. As in the previous example, the network shows some difficulty to predict the nodes close to the inflow but improves as the simulation time varies leading to a good approximation of the flow in the final steps. As we have previously seen, GNN-based models are often not good at predicting the regularity of the solution pattern, as we can see from the prediction at $T = 0.1$ in Figure 18. Here, we can also notice that the smoothness of the solution gets better as the simulation time increases.

In Figure 19 the height of the bump influences a lot the dynamic of the system, however, the network correctly infers the behavior of the flow around the obstacle. Here the bump has height $h = 0.175$ and is located in the lower edge with center at $x = 0.453$, that is $\boldsymbol{\mu} = (0.453, 0, 0.175)$. The height of the bump has a significant impact on the accuracy of model prediction, particularly near the upper edge of the domain. Errors that arise in this region can propagate throughout the domain, affecting the accuracy of predictions at other locations as well. However, the self-adjustment mechanism of the model is effective in mitigating these errors as they propagate toward the outflow, resulting in improved accuracy in this region. Overall, the model ability to account for the influence of the bump height on the flow dynamics contributes to its strong predictive performance.

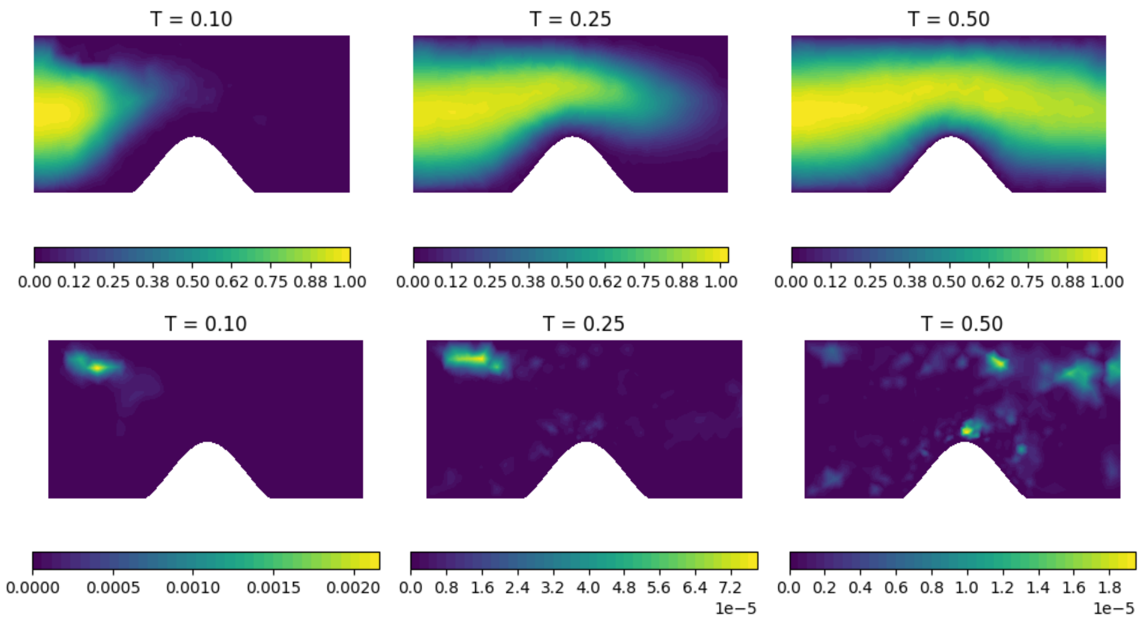

Figure 19: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. Prediction obtained for $\boldsymbol{\mu} = (0.453, 0, 0.175)$ with the bump in the lower edge with center at $x = 0.453$ and height $h = 0.175$. First row: rollout prediction. Second row: RMSE related to each timestep between the prediction and the corresponding ground truth solution.

In Figure 20 the bump has height $h = 0.08$ and is in the upper edge with center at $x = 0.467$, that is $\boldsymbol{\mu} = (0.467, 0.5, 0.08)$. The accuracy of the model's predictions decreases when the size of the bump is smaller. This is primarily due to the fact that as the size of the domain increases, so does the number of nodes, making the inference process more challenging. In this problem, the number of nodes varies from 936 to 1054, which is a wide range for unstructured meshes and geometries that differ significantly from each other. As a result, error propagation is more significant in this case compared to the other examples. This is evidenced by the persistence of high errors at $T = 0.25$. Nevertheless, despite the irregularity of the prediction, the overall dynamics are well-predicted by the model. This suggests that the model can effectively capture the underlying physics of the system, even in cases where the inference is more challenging due to the higher number of nodes.

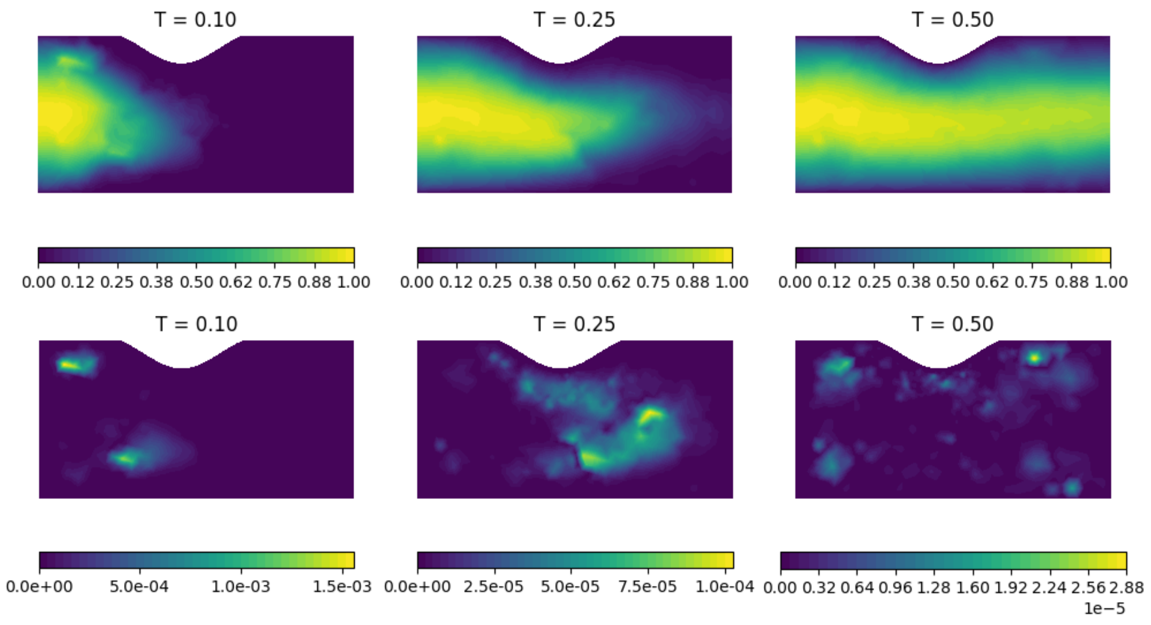

Figure 20: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. Prediction with bump in the lower edge with center at $x = 0.453$ and height $h = 0.08$ ($\boldsymbol{\mu} = (0.467, 0.5, 0.08)$). First row: rollout prediction. Second row: RMSE related to each timestep between the prediction and the corresponding ground truth solution.

Upon observing the L^2 relative error plot on the test set in Figure 21, we can draw some quantitative conclusions regarding the previously discussed results. The plot indicates that the test error has an appropriate upper bound and that it initially increases significantly during the first few time steps, which is consistent with the observed prediction behavior. We can see that the L^2 error trend is similar to that of Figure 17, but it decays more rapidly. This suggests that the self-adjustment property of the model is maintained even when the problem geometry exhibits greater variability.

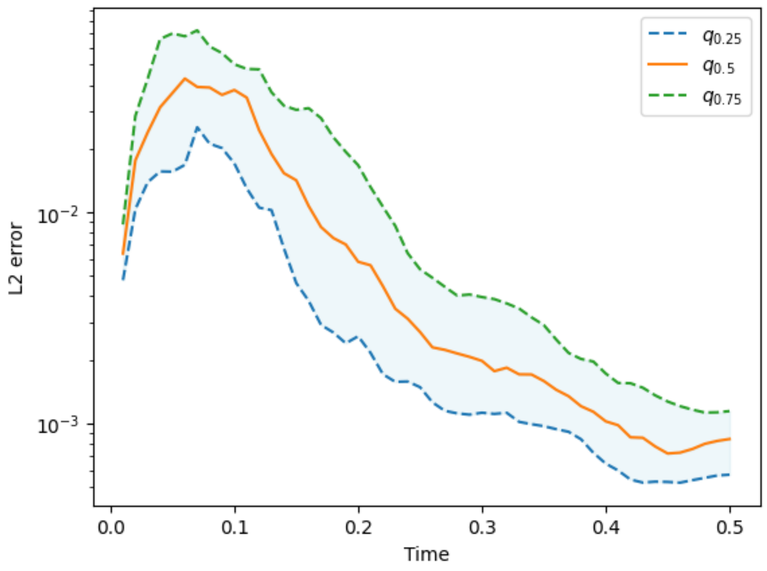

Figure 21: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. L^2 relative error vs Time plot.

We can further test the robustness of our model by evaluating its ability to predict simulations with varying shapes of the bump, without requiring any retraining. Figure 22 displays the prediction results of a simulation with a triangular bump located on the upper edge. In addition to the fact that the errors are of the same order of magnitude as previously discussed, the overall dynamics are accurately predicted. However, the regularity of the solution poses some difficulty for the model. Nonetheless, this does not appear to significantly impact the accuracy of the prediction.

This example highlights the flexibility of graph neural networks in handling different geometries. In scenarios where there is limited data available for training, the inductive capabilities of our model enable us to extend simulations by slightly modifying the domain, while still producing reliable results.

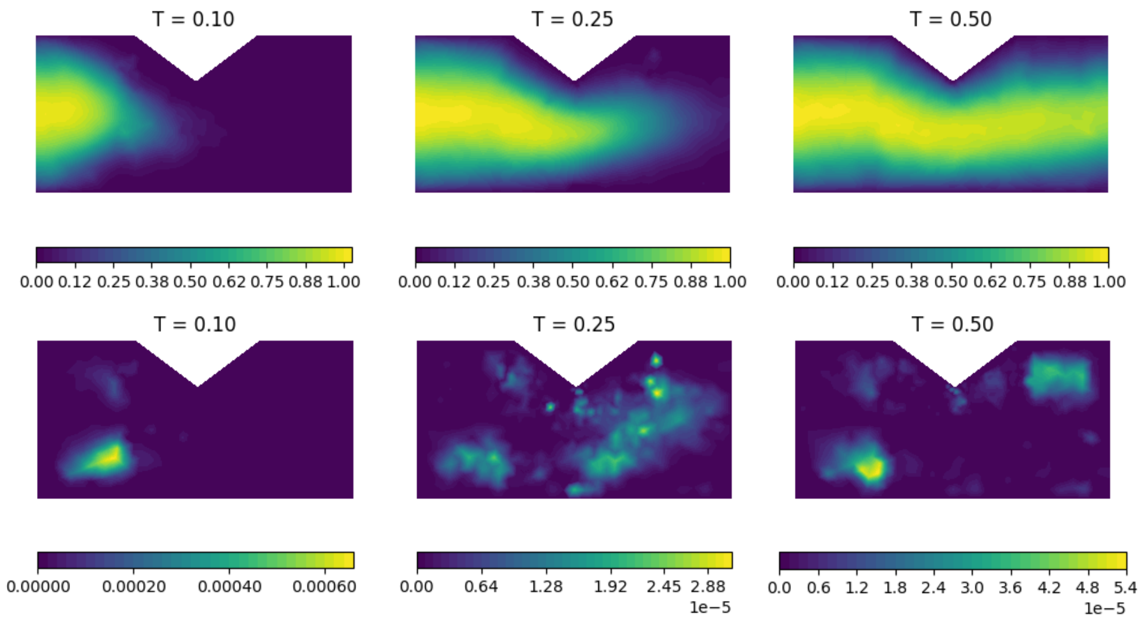

Figure 22: Test case 2, Advection-Diffusion problem in a 2D Stokes flow. Prediction with a triangular bump on the upper edge. First row: rollout prediction. Second row: RMSE related to each timestep between the prediction and the corresponding ground truth solution.

5.3. Advection-Diffusion problem in a 3D Stokes flow around a cylinder

We consider the same problem discussed in Section 5.2 but in a 3D domain obtained by an extrusion on the z-axis of rectangle $R = (0, 1) \times (0, 0.5)$ with a circular hole $C = \{(x, y) : (x - c_x)^2 + (y - c_y)^2 \leq (0.05)^2\}$. We have let the position of the obstacle vary randomly in the rectangle $R_c = [0.2, 0.4] \times [0.2, 0.3]$ resulting in 150 different simulations, that is $\boldsymbol{\mu} = (c_x, c_y) \in \mathcal{P} = [0.2, 0.4] \times [0.2, 0.3]$. An example of FOM solution can be seen in Figure 23.

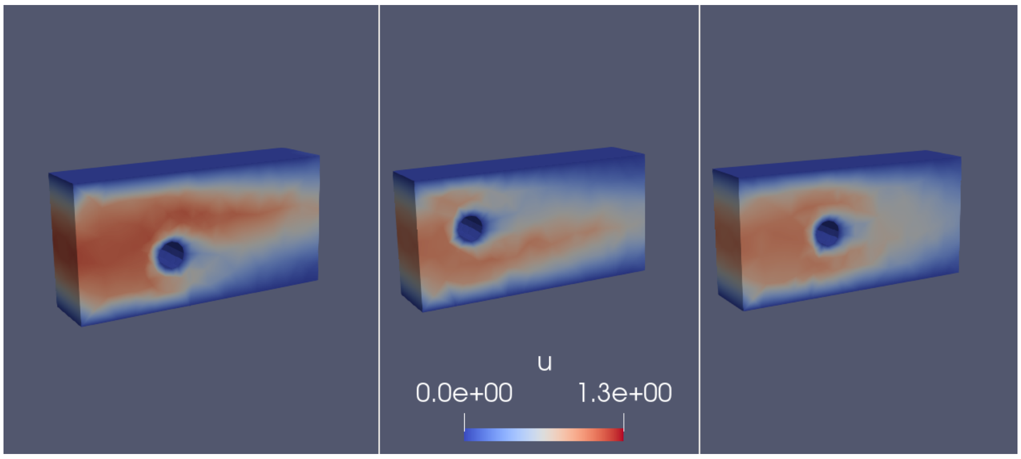

Figure 23: 3 different ground truth simulations.

5.3.1. Problem data

The mesh nodes of the simulations vary from 1353 to 1542, which increases the complexity of the problem with respect to the other examples we have discussed. We consider the same node features of the previous examples, but now the edge features are a $N_{batch} - 1 \times N_{edges} \times 4$ tensor, the 3 coordinates and the edge length. The batch size is still $N_{batch} = 25$ and these are the other hyperparameters chosen:

- $l = 32$, where l is the latent dimension of the network;
- noise variance $\sigma^2 = 10^{-5}$;
- total number of epochs $max\_epoch = 2000$;
- learning rate $\nu = 10^{-4}$ with decay $= 0.1$ after 500 and 1000 epochs;
- SiLU is chosen as the activation function for each MLP layer;
- number of MLP layers $mlp\_layers = 2$;
- message Passing steps $mp\_steps = 18$;
- The loss weights $w_1 = 0.5,\ w_2 = 0.5$.

The training set is composed of 125 simulations while the test set includes 25 simulations, both chosen randomly among the 125 FOM simulations. In this third example, we are face the challenge of addressing both dynamical system complexity and computational complexity. To tackle these challenges, we have employed a specific training strategy where the model is trained for a relatively shorter period of 2000 epochs, but with an increased number of message passing steps (18) to better capture the dynamics of the system. Additionally, we have also increased the noise variance from 10^{-6} to 10^{-5} to enable the model to learn to handle higher levels of error propagation. This is particularly important as error propagation is a common issue when working with simulation rollouts for this problem. By adopting this training strategy, we aim to strike a balance between model accuracy and computational efficiency while still being able to capture the complex dynamics of the system.

5.3.2. Numerical Results

The results of the rollout predictions of the test simulations are summarized in Table 3. It is evident that the higher error obtained in this example is due to the increased complexity of the problem. However, it is noteworthy that despite the higher error, there is a significant improvement in time complexity. Once trained, our model can simulate up to two orders of magnitude faster in this example compared to the finite element solver. This reduction in time complexity can lead to faster and more efficient simulations, which is particularly important for time-critical applications or when a large number of simulations are required. Therefore, despite the slightly higher error, using our model can still provide a significant advantage in terms of time and computational resources.

Advection-Diffusion problem in a 3D Stokes flow

	RMSE (mean)	**RMSE (max)**	**RMSE (min)**	$\mathbf{t_{gt}}$**(s)**	$\mathbf{t_{pred}}$**(s)**
AD 3	4.37×10^{-2}	6.24×10^{-2}	1.94×10^{-2}	≈ 729.42	≈ 10.4

Table 3: Test case 3, Advection-Diffusion problem in a 3D Stokes flow. Results of the test set predictions.

Upon examining the predictions in greater detail, as shown in Figure 24, a comparison can be made between the prediction of a simulation with the obstacle positioned centrally, and its corresponding ground truth solution. It is evident that the simulation deteriorates as it progresses toward the outflow. Unlike the 2D case, self-adjustment is not observed in this scenario, as the nodes located to the right of the obstacle are heavily influenced by its position. This may result in some values being underestimated in the prediction, particularly in the tail of the flow. Unfortunately, this is a known drawback of using message passing neural networks, like the one implemented in this study, as they tend to homogenize predictions if the network has too many nodes. Therefore, even if the dynamics are predicted accurately, node values may be more dispersed. Furthermore, this problem is exacerbated by an increase in the number of message passing steps, which, in this example, are necessary for an acceptable prediction.

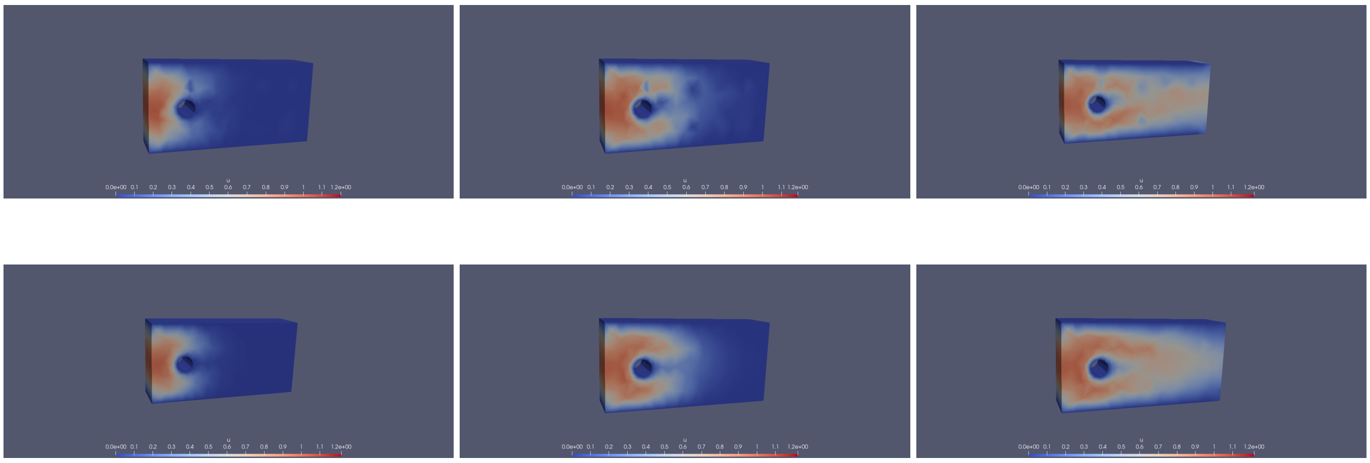

Figure 24: Test case 3, Advection-Diffusion problem in a 3D Stokes flow. Prediction obtained for $\boldsymbol{\mu} = (0.29, 0.25)$. 3 time steps of simulation. First row: Rollout prediction. Second row: ground truth solution.

Nevertheless, as illustrated in Figure 25, modifying the position of the obstacle does not significantly affect the overall accuracy of the solution. Despite the aforementioned issues, the flow pattern is captured correctly, and no propagation errors are observed. Of remarkable importance is the consistently accurate prediction in the proximity of the obstacle, which is always a critical aspect to be predicted. This observation underscores the model ability to learn the geometrical properties of the problem while preserving the graph structure of the mesh. Therefore, these results suggest that the model is sufficiently robust in predicting flow patterns in various configurations, and can generalize well to other geometries.

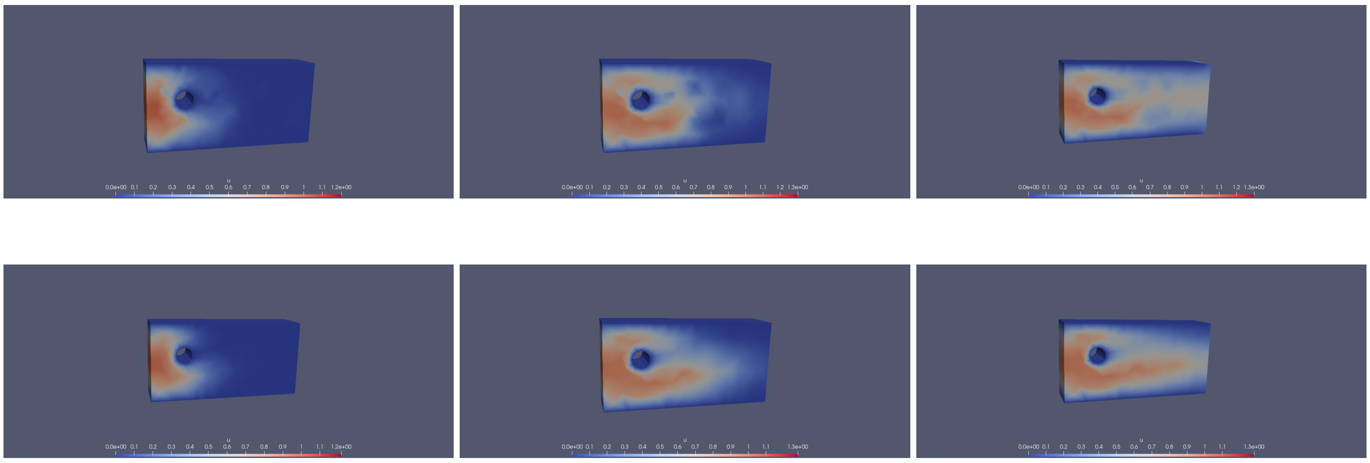

Figure 25: Test case 3, Advection-Diffusion problem in a 3D Stokes flow. Prediction obtained for $\boldsymbol{\mu} = (0.23, 0.3)$. 3 time steps of simulation. First row: Rollout prediction. Second row: ground truth solution.

Upon observing the L^2 relative error plot on the test set in Figure 26, we can draw quantitative conclusions regarding the previously discussed results. The plot indicates that the test error has an appropriate upper bound and that it initially increases significantly during the first few time steps, which is consistent with the observed prediction behavior. After the initial increase, the error gradually decays, showing that the model has learned the underlying dynamics of the system. However, towards the end of the simulation, we observe a slight increase in the error, which is coherent with what we have previously mentioned about the tendency of these architectures to dispersion. This behavior may be due to the accumulation of errors during the long-term prediction. Therefore, we can conclude that while the GNN-based model shows promising results, there is still room for improvement in terms of accuracy and robustness.

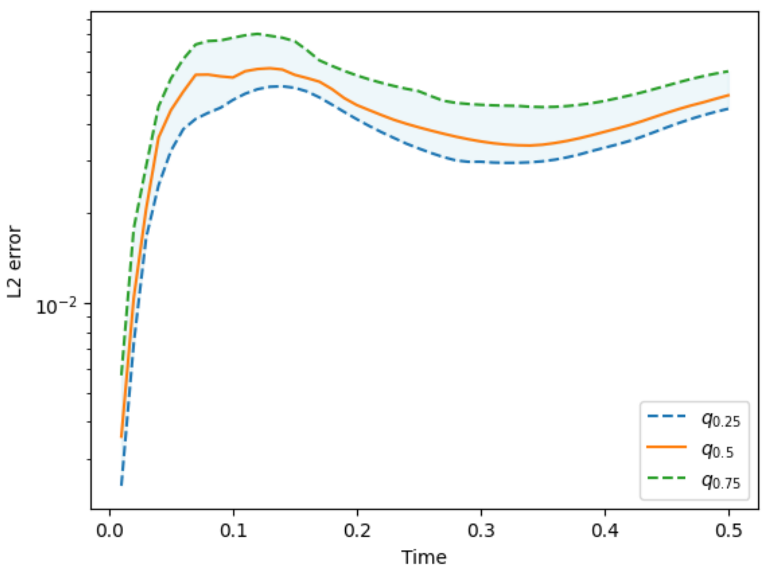

Figure 26: Test case 3, Advection-Diffusion problem in a 3D Stokes flow. L^2 relative error vs Time plot: The dashed lines represent the first and the third quantiles of the L^2 errors among all the test predictions, while the orange line is the median. The shaded area can be seen as a confidence region for the simulation error.

5.4. Comparison with Feed Forward Neural Networks

Feed Forward Neural Networks (FFNNs) are usually employed for building reduced-order models because they have the capability to capture strong nonlinearity through their fully connected structure [15, 17]. Hence, we compare the performance of a common FFNN to our GNN on the two examples discussed in Sections 5.1 and 5.2. In particular, to make things simpler for FFNN, we consider for each problem the following datasets:

- for the first Advection-Diffusion problem we let the obstacle vary only in its position, resulting in 100 simulations, among which we choose randomly 80 simulations for the training set and 20 for the test set;
- For the Advection-Diffusion problem in a 2D Stokes flow we let the bump vary in its position along the upper and lower edges but not in its height, resulting in 125 simulations, among which we choose randomly 100 simulations for the training set and 25 for the test set.

In order to train an FFNN, we need to bring all the simulations to the same degrees of freedom, so we interpolate the region of interest onto a modeling 128×128 vertices grid so that the order of the nodes in the mesh is preserved. Then we build an Encoder-Decoder fully connected network to approximate the solution of the system at time t^{n+1}, given the one at time t^n, as described in Equation (4). The training is performed by one-step prediction and minimizing the batch loss in Equation (7), as already presented in Section 4.1. Moreover, we pass as input to the network a $N_{batch} \times 128 \times 128 \times 3$ tensor containing the same features as the GNN model. Since FFNNs tend to overfit if trained for a long time, we trained the model for 500 epochs, using the same learning rate and loss weights as the ones described in Sections 5.1 and 5.2.
The prediction at testing is done by exploiting the rollout of the simulation as shown in Equation (10) in order to compare the robustness to propagation errors of the two models. In Figure 27 we can clearly see that the predictions done with GNN have less variance than the ones done with FFNN. Even if the range of the errors is similar, the GNN errors are overall better despite some anomalies.

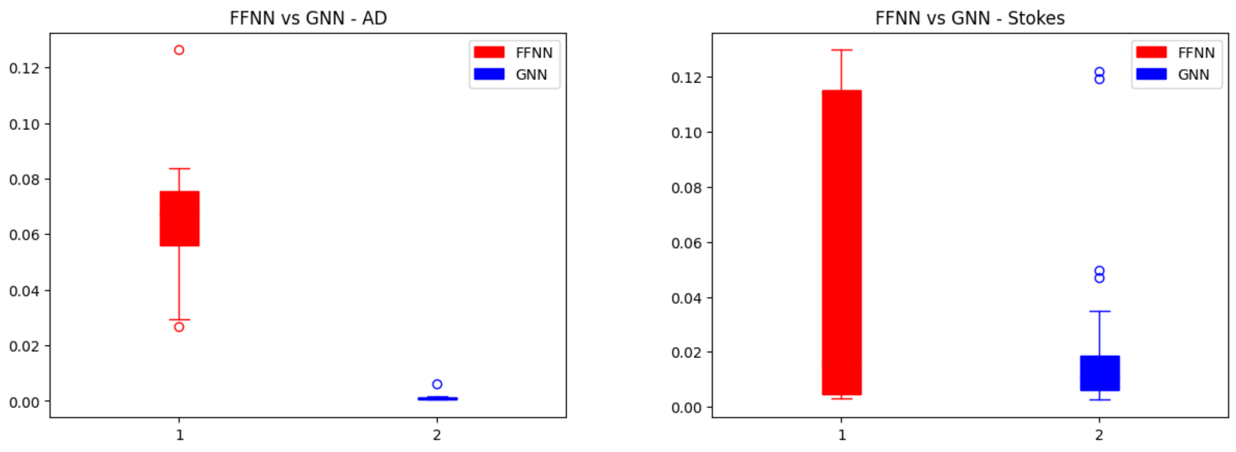

Figure 27: Boxplots of the RMSEs of the two predictions

Two examples of FFNN predictions on test set simulations, together with the respective GNN predictions are shown in Figures 28 and 29. The predictions are done using the same values of geometrical parameters in order to highlight the different performances in the generalization on unseen domains. It is indeed, evident that there is a significant difference between using an FFNN and a GNN for solving problems that require making inferences on the geometry. While FFNNs may capture the overall dynamic of the system quite well, they do not consider any geometric properties of the solution. This limitation becomes particularly evident in simple examples, which implies that the benefits of using a GNN are more pronounced when dealing with more complex problems.

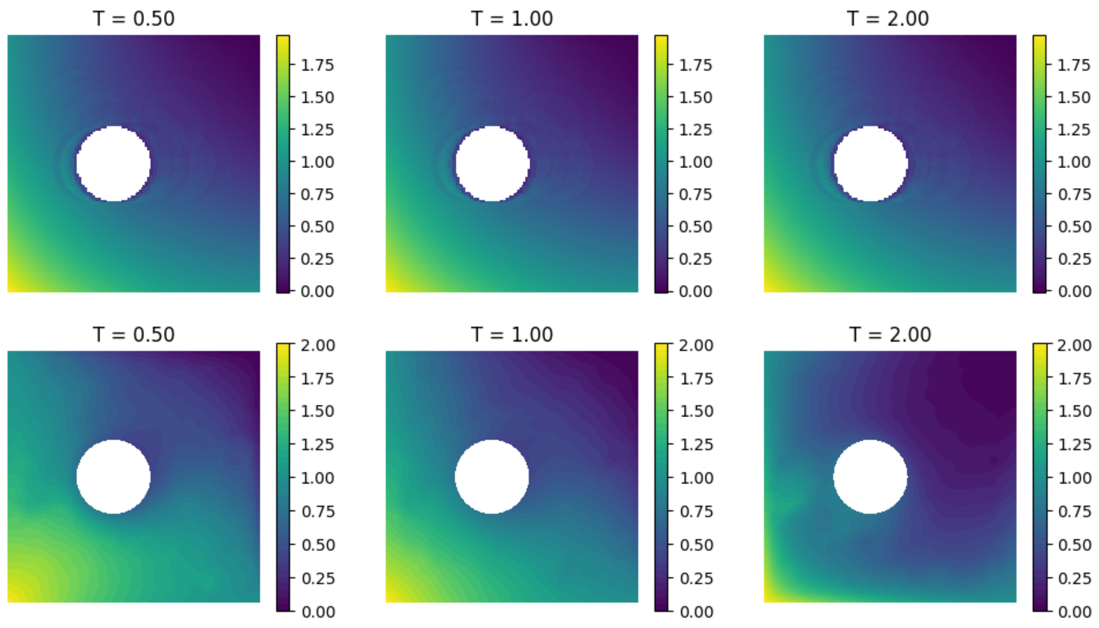

Figure 28: Test case 1, Advection - Diffusion problem. Comparison between FFNN and GNN prediction for $\boldsymbol{\mu} = (c_x, c_y) = (0.4, 0.5)$. First row: FFNN prediction. Second row: GNN prediction.

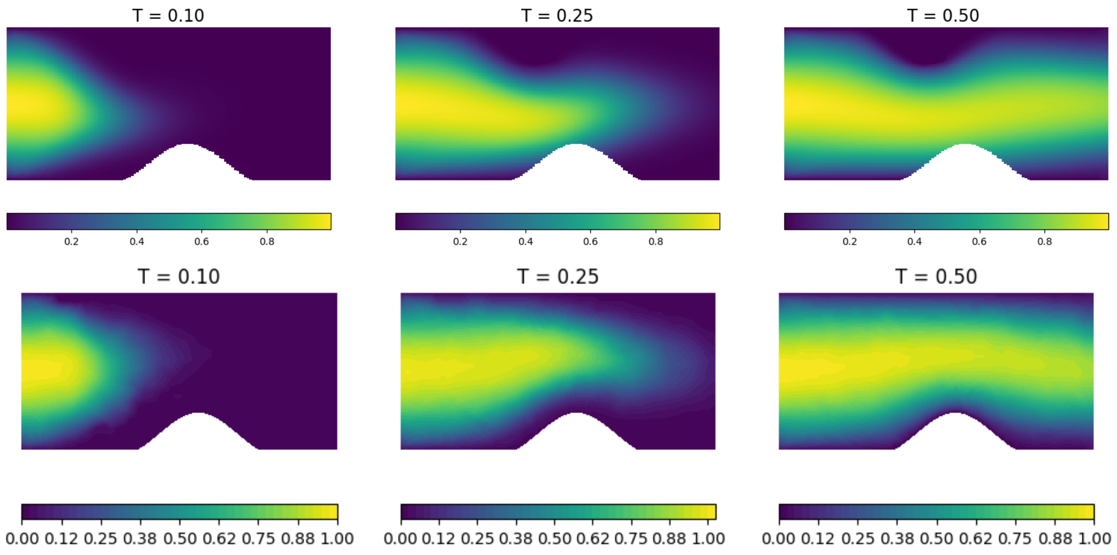

Figure 29: Test case 2, Advection - Diffusion problem in a 2D Stokes flow. Comparison between FFNN and GNN prediction for $\boldsymbol{\mu} = (c_x, c_y) = (0.6, 0)$. First row: FFNN prediction. Second row: GNN prediction.

Another key aspect to analyze is the number of parameters of the model. The parameters of an FFNN may increase fast due to its fully connected structure; this implies a higher tendency to overfit, as we can see from Figure 29, and moreover, it gets the model less scalable as the complexity of the problem increases.

One solution to address both of these issues is the use of grid-based models, such as Convolutional Neural Networks (CNNs). These models can reduce the number of parameters by sharing them, which helps to mitigate the overfitting issue. In Equation 5, the weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ are shared across simulations in the CNN architecture. However, this may lead to lower prediction accuracy due to excessive generalization. This is also a drawback of GNN models, which often struggle to propagate information globally across the mesh.

On the other hand, while CNNs can capture the dynamics of the system properly, they do not consider the geometric structure of the problem, making them unsuitable for complex geometries. Therefore, while CNNs are a viable alternative to FFNNs, they are not a perfect solution for problems that require the consideration of both geometry and dynamics, such as the ones we have discussed. In this context, Mesh-Informed Neural Networks (MINNs) are a good choice since they can incorporate geometric features into their definition. However, they are still limited in their ability to interpolate the solution in a more general domain, where the number of nodes in the mesh may vary.

6. Conclusions

Graph Neural Networks have been designed to perform *inductive* inference on the geometric structure of a given graph-structured problem, such as a mesh-based simulation of a problem governed by PDEs. In contrast to Feedforward Neural Networks (FFNNs) and Convolutional Neural Networks (CNNs), GNNs can naturally handle problems with varying geometrical parameters since they intrinsically incorporate mesh features such as the edge connectivity matrix. Moreover, since they are independent of the input degrees of freedom (dofs) of the mesh, they can generalize to different mesh structures.
As demonstrated through the result obtained in this work, this property, referred to as the *inductive capability* of GNNs, is a key advantage of this approach, and its potential is clearly illustrated. In principle, a GNN model can be trained to solve a problem and then generalizes the solution to different domains. This is particularly useful in time-dependent PDEs where different geometries entail different solution patterns. Therefore, the use of GNNs can provide a powerful and flexible tool to solve physical problems modeled by PDEs, which depend in particular on geometrical parameters, efficiently, and in a computationally tractable manner.

GNNs exhibit a lower tendency to overfit than FFNNs because the learned map Φ is the same for each mesh node by definition. Additionally, the robustness of this model to propagation errors during rollout prediction is another major advantage. In most of the cases presented in the examples, the GNN model successfully and accurately simulated long rollouts using only the initial solution as input. This highlights the better ability of GNNs to handle physical problems in a computationally efficient and robust manner.
These properties make GNNs an attractive tool for solving a wide range of physical problems, especially those that are difficult to model using conventional approaches.

However, it is important to note that some specific scenarios need to be considered, especially when dealing with complex domain topologies. In such cases, the quality of the prediction may be affected by the complexity of the geometry, which may require a finer mesh resolution and a more significant computational effort. Additionally, another limitation of using Graph Neural Networks is the computational complexity associated with simulating over fine meshes. This can lead to a higher number of message-passing steps, resulting in an increased computational cost and reduced accuracy due to inefficient propagation.
Despite these challenges, the use of GNNs remains an attractive option for modeling physical problems with complex geometries and dynamics, especially compared to FFNNs, which are limited in their ability to model complex spatial relationships, and CNNs, which have been shown to be effective in image analysis and processing, but they require fixed grid-like data structures, and may not be well-suited for problems with irregular geometries.

Hence, in future research, it may be advantageous to explore the potential of combining well-established deep learning-based reduced order models, such as autoencoders and U-Net-like architectures, with graph representations. This hybrid approach could potentially lead to improved accuracy in predicting simulations, even on more refined meshes, while simultaneously reducing computational complexity.
Furthermore, additional investigations could explore the use of attention mechanisms or other forms of neural network architectures that can selectively weight the contributions of different nodes in the graph, potentially improving the performance of the model on more complex geometries.

References

[1] Aleksandar Bojchevski and Stephan Günnemann. Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking. *arXiv preprint arXiv:1707.03815*, 2017.

[2] Annalisa Buffa and Giancarlo Sangalli. *Isogeometric analysis: a new paradigm in the numerical approximation of PDEs: Cetraro, Italy 2012*, volume 2161. Springer, 2016.

[3] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900, 2015.

[4] F. Casadei, E. Gabellini, G. Fotia, F. Maggio, and A. Quarteroni. A mortar spectral/finite element method for complex 2d and 3d elastodynamic problems. *Computer Methods in Applied Mechanics and Engineering*, 191(45):5119–5148, 2002.

[5] Federico Fatone, Stefania Fresca, and Andrea Manzoni. Long-time prediction of nonlinear parametrized dynamical systems by deep learning-based reduced order models. *arXiv preprint arXiv:2201.10215*, 2022.

[6] Nicola Franco, Andrea Manzoni, and Paolo Zunino. A deep learning approach to reduced order modelling of parameter dependent partial differential equations. *Mathematics of Computation*, 92(340):483–524, 2023.

[7] Nicola Rares Franco, Andrea Manzoni, and Paolo Zunino. Learning operators with mesh-informed neural networks. *arXiv preprint arXiv:2203.11648*, 2022.

[8] Stefania Fresca, Luca Dede', and Andrea Manzoni. A comprehensive deep learning-based approach to reduced order modeling of nonlinear time-dependent parametrized pdes. *Journal of Scientific Computing*, 87:1–36, 2021.

[9] Stefania Fresca and Andrea Manzoni. Pod-dl-rom: enhancing deep learning-based reduced order models for nonlinear parametrized pdes by proper orthogonal decomposition. *Computer Methods in Applied Mechanics and Engineering*, 388:114181, 2022.

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

[11] Rini Jasmine Gladstone, Helia Rahmani, Vishvas Suryakumar, Hadi Meidani, Marta D'Elia, and Ahmad Zareei. Gnn-based physics solver for time-independent pdes. *arXiv preprint arXiv:2303.15681*, 2023.

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

[13] Quercus Hernández, Alberto Badías, Francisco Chinesta, and Elías Cueto. Thermodynamics-informed graph neural networks. *arXiv preprint arXiv:2203.01874*, 2022.

[14] Saddam Hijazi, Giovanni Stabile, Andrea Mola, and Gianluigi Rozza. Data-driven pod-galerkin reduced order model for turbulent flows. *Journal of Computational Physics*, 416:109513, 2020.

[15] Sean T Kelly and Bogdan I Epureanu. Data-driven reduced-order model for turbomachinery blisks with friction nonlinearity. In *Nonlinear Structures & Systems, Volume 1: Proceedings of the 40th IMAC, A Conference and Exposition on Structural Dynamics 2022*, pages 97–100. Springer, 2022.

[16] Usik Lee. *Spectral element method in structural dynamics.* John Wiley & Sons, 2009.

[17] Nikolaj T Mücke, Sander M Bohté, and Cornelis W Oosterlee. Reduced order modeling for parameterized time-dependent pdes using spatially and memory aware deep learning. *Journal of Computational Science*, 53:101408, 2021.

[18] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2014–2023, New York, New York, USA, 20–22 Jun 2016. PMLR.

[19] Luca Pegolotti, Martin R Pfaller, Natalia L Rubio, Ke Ding, Rita Brugarolas Brufau, Eric Darve, and Alison L Marsden. Learning reduced-order models for cardiovascular simulations with graph neural networks. *arXiv preprint arXiv:2303.07310*, 2023.

[20] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

[21] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.

[22] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[23] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4470–4479. PMLR, 10–15 Jul 2018.

[24] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, dec 2018.

[25] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):1–23, 2019.

Abstract in lingua italiana

Le simulazioni numeriche svolgono un ruolo chiave nella modellizzazione di sistemi fisici complessi che, in molte discipline della Scienza e dell'Ingegneria, richiedono la soluzione di equazioni alle derivate parziali (EDP) dipendenti dal tempo. In questo contesto, i Full Order Models (FOM), come quelli che si basano, ad esempio, sul metodo degli elementi finiti, possono raggiungere alti livelli di precisione, richiedono tuttavia spesso simulazioni computazionalmente costose da eseguire. Per questo motivo, vengono sviluppati modelli surrogati al fine di sostituire i solutori computazionalmente costosi con altri più efficienti, che possano garantire trade-offs favorevoli tra accuratezza ed efficienza. Questo lavoro esplora l'applicazione di Graph Neural Networks (GNN) per la simulazione di EDP parametrizzate dipendenti dal tempo definite su geometrie di forma variabile, che possono essere ricondotte a grafi. Le GNN hanno recentemente dimostrato grande potenziale nella risoluzione di problemi complessi in domini come la computer vision e il natural language processing: questa Tesi si propone di investigarne il potenziale per la simulazione numerica di EDP. Il vantaggio nell'utilizzo di GNN in questi problemi risiede nella loro capacità di generalizzare a diverse geometrie mediante l'introduzione di una rappresentazione grafica adatta al dominio del problema. Dopo aver passato brevemente in rassegna la classe di problemi su cui si concentra il lavoro, viene introdotto il concetto di GNN, la sua architettura e una possibile applicazione ai problemi basati su grafi. La Tesi propone un nuovo metodo per utilizzare le GNN per risolvere le EDP attraverso (i) la conversione della EDP in un problema basato su grafo e (ii) il training di una GNN sul grafo risultante. L'efficacia del metodo proposto viene valutata attraverso una serie di esperimenti che dimostrano come la strategia investigata in questa Tesi sia superiore ai metodi numerici tradizionali in termini di efficienza computazionale e generalizzazione a nuovi scenari.

Parole chiave: graph neural networks, equazioni alle derivate parziali, modelli surrogati

Acknowledgements

I would like to express my deepest gratitude to my advisor, Prof. Andrea Manzoni, for his support, guidance, and encouragement throughout this research project. His expertise, insightful comments, and constructive criticism have been invaluable to me.

I would also like to thank my co-advisors, Dr. Nicola Rares Franco and Dr. Stefania Fresca, for their feedback and suggestions, which have helped me improve my work. I am grateful to them for taking the time to review my thesis and offer their expertise.

I am grateful to my family and friends who have supported me emotionally and encouraged me to persevere. Their love and encouragement have been a source of strength throughout this process.

Finally, I would like to acknowledge the support and resources provided by MOX laboratory at Politecnico di Milano. Their facilities and staff have been instrumental in facilitating my research.