Appunti tesi

1. Deep Learning-based surrogate models for parametrized PDEs: handling geometric variability through graph neural networks

The use of GNNs as an alternative to surrogate models, with the aim of replacing computationally expensive solvers with more efficient ones.

In particular the paper deals with the problem of geometric variability.

Surrogate models are emulators that are able to reproduce full order models (FOM) outputs at a cheaper cost. This is known as Reduced order Modeling (ROM).

All existing approaches rely on the fact that the FOM to which they refer needs to be identified and fixed with its dofs and geometry, this poses a problem with parametrized domain PDEs.

GNNs are computational units that receive as input a set of node features and return a corresponding set as output.

GNNs adopt a local perspective, they are in fact able to process information at the nodal level, exploiting the fact that close nodes "communicates" together.

This is called (relational) inductive bias, the ability to induce a solution over an other. The implicit assumption is that the local effects are stronger than the global ones thanks to the influence over a give neuron of its closed ones.

PDEs are time dependent and parametric (parameters are both geometrical and physical). The FOM is expressed as a nonlinear high-dimensional parametrized dynamical system.

The set of solutions depending on parameters is also called solution manifold, and it's the soal of the paper to approximate it. GNNs are used to produce extremely flexible models capable to generalize to different and also unseen geometries, thanks to the inductive bias.\

GNNs adopt a graph-in graph-out architecture. Fundamental is the feature of message passing, each node generates a message and collects the ones of the neighboring nodes. Graph-based algorithms, such as, e.g, graph convolutional networks, GraphSage , graph attention networks, graph transformer operator and interaction networks, differ in the way the message is computed and the aggregation is performed.

Also edge features can be used.

To perform all these operations a structure to encode the graph need to be employed. The edge connectivity matrix consists in 2 columns where each row gives the starting and ending nodes of each edge.

About message-passing then architecture of Encoder-Processor-Decoder model is presented.

A graph forward pass is a computational unit that transforms the vertex features associated to the nodes. A message-passing block F is comprised of two Multi-Layer Perceptron (MLP) units.

Useful notation:

- Vertex features v
- Maps that associate respectively starting and ending vertex features of an edge v_in and v_out
- Edge features e
- Maps that associate to a node the sum of the features of the edges with the node as destination e_bar
- Concatenation operator that append in a single vector to maps with same input

The message block F is defined as:

$$F(\boldsymbol{v}, \boldsymbol{e}, G) = \psi_v \circ \left(\boldsymbol{v} \oplus \overline{\psi_e \circ (\boldsymbol{e} \oplus \boldsymbol{v}_{\text{in}} \oplus \boldsymbol{v}_{\text{out}})} \right)$$

The Encoder-Processor-Decoder model takes as input a graph defined over a mesh, a signal defined on the mesh vertices, and a global feature vector non spatial e.g. time. It is divided into 3 sub-modules.

Encoder module takes the inputs above and returns features in the nodes and the edges.

Respectively as:

$$\mathscr{E}_v(\boldsymbol{u}, \xi, G) := \Psi_{\mathscr{E}}^v \circ (\boldsymbol{u} \oplus \xi \oplus \boldsymbol{b}_G), \quad \mathscr{E}_e(\boldsymbol{u}, \xi, G) := \Psi_{\mathscr{E}}^e \circ \boldsymbol{e}_G,$$

With,

$$e_G(\mathbf{x}_1, \mathbf{x}_2) := \left[\frac{x_1^{(1)} + x_2^1}{2}, \dots, \frac{x_1^{(d)} + x_2^d}{2}, |\mathbf{x}_1 - \mathbf{x}_2| \right] .$$

The psi MLPs are learnable during training.

The Processor module takes as input the node and edge features generated by the encoding and applies the message-passing block F above m times. It only acts on the nodes features keeping the edge ones unchanged, and allows the communication between as many steps as the number m of units that the processor employs.

The decoder module consists in an MLP unit that from the node features output generates a vector of phi output signals.

Applying this model to parametrized PDEs we want to predict the solution at time t^n+1, knowing it at time t^n, the method is based on th Runge-Kutta idea:

$$\Phi(\mathbf{u}_{\boldsymbol{\mu}}^n, t^n, \boldsymbol{\mu}) \approx \mathbf{u}_{\boldsymbol{\mu}}^{n+1} , \quad \Phi(\mathbf{v}, t^n, \boldsymbol{\mu}) := \mathbf{v} + \Delta t \tilde{\Phi}(\mathbf{v}, t^n, \mathscr{M}_{\boldsymbol{\mu}}^h) .$$

Phi ~~ is the GNN architecture analyzed above.

We identify the mesh with its underlying graph, and the dofs vector u with the vertex feature map v. In each node(or dof), the map takes the node and gives out the velocity or the solution of the pde.

It's noteworthy that the time t^n is treated as a global parameter.

Again it's underlined how the strength of this model lies in the possibility of evaluating the equation independently on the number of dofs, training the model using different meshes and predicting solutions for new geometries.

To train the GNN architecture N_train FOM time-sequences of solutions are generated, all with differente parameter vector mu.

The loss function to minimize presents both the Runge-Kutta type of equation that computes the time-step in the model and the equivalence between the GNN output and a suitable fi nite-difference approximation of the ground truth time-derivative.

Random noise and mini-batches are introduced to avoid error propagation when advancing much in time. Optimization I performed through back-propagation and ADAM with a decreasing learning rate over the epochs.

After training the performance of the model is assessed though the RMSE evaluation with respect to a new FOM simulation with the entire rollout.

The most important hyper-parameter to be tuned is the message-passing number of steps, a small one 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.

NUMERICAL RESULTS TO BE SUMMARIZED

2. A Comprehensive Deep Learning-Based Approach to Reduced Order Modeling of Nonlinear Time-Dependent Parametrized PDEs

About DL techniques it's stressed the value in terms of computational costs of using stochastic gradient descent with mini-batch samples.

Convolutional NNs are used to increase efficiency both in terms of memory and computational effort, in the case of high-dimensional data.

In the context of reduced order modeling (ROM) techniques, conventional methods (e.g. reduced basis RB with proper orthogonal decomposition POD) result inefficient when dealing with high accuracy levels.

A non-linear approach based on DL algorithms is proposed (DL-ROM), both in the nonlinear trial manifold and in the nonlinear reduced dynamic, during the resolution of parametric PDEs.

The assumption of reduced order modeling is that the solution of a high dimensional FOM lies on a low-dimensional manifold embedded in the space, and its approximation as a function of the parameters, through the trial manifold, is the goal of ROM.

About constructing the reduced trial manifold, the decoder function of a convolutional autoencoder is employed while a feedforward neural network and the encoder function of a convolutional autoencoder are used to model the reduced dynamics on it.

The approach is purely data-driven and relies on the computation of a set of FOM snapshots.

The task of linear projection-based ROMs is generalized to nonlinear ROMs case.

The solution manifold that needs to be approximated has the form:

$$\mathcal{S}_h = \{\mathbf{u}_h(t; \boldsymbol{\mu}) \mid t \in [0, T) \text{ and } \boldsymbol{\mu} \in \mathcal{P} \subset \mathbb{R}^{n_\mu}\} \subset \mathbb{R}^{N_h}.$$

Time variable plays the role of an additional coordinate so that the dimension of the manifold results at most to be the number of parameters n_mu plus one.

Each point u_h belonging to the solution manifold has a tangent space spanned by n_mu + 1 basis functions.

A reduced linear trial manifold is introduced to approximate the solution manifold, the former is a subspace of the latter, and is panned by the n columns of a matrix V.

In particular V represents the linear map from the intrinsic coordinates u_n (low-dimension n) to the approximation of u_h (high-dimension N_h).

POD is one of the most common tools for generating the linear trial manifold.

(READ AGAIN POD AND TAKE SOME NOTES)

A nonlinear ROM to approximate u_h is built. The reduced nonlinear trial manifold becomes:

$$\tilde{\mathcal{S}}_n = \{\boldsymbol{\Psi}_h(\mathbf{u}_n(t; \boldsymbol{\mu})) \mid \mathbf{u}_n(t; \boldsymbol{\mu}) \in \mathbb{R}^n,\ t \in [0, T) \text{ and } \boldsymbol{\mu} \in \mathcal{P} \subset \mathbb{R}^{n_\mu}\} \subset \mathbb{R}^{N_h}.$$

In the DL-ROM proposed neural networks are used to replace the nonlinear function (psi) that maps the n dofs low-order solution in the trial manifold to the approximation u~h of the solution, and the one that maps a couple of parameters and time into the solution in the trial manifold (phi).

The DL-ROM technique is composed of two parts, respectively, for the reduced dynamics learning (phi) and the reduced trial manifold learning (psi).

“Hereon, we denote by Ntrain, Ntest and Nt the number of training-parameter, testing-parameter, and time instances, respectively, and set Ns = Ntrain Nt . The dimension of both the FOM solution and the ROM approximation is Nh, while n << Nh denotes the number of intrinsic coordinates.”

A deep feedforward neural network (DFNN) with L layers is employed to approximate phi while the decoder function of a convolutional autoencoder (AE) plays the role of psi. The parameters to be optimized are thus the ones of the decoder function theta_D and the ones for the DFNN theta_DF.

The optimization is performed minimizing a loss function with respect to both vectors of parameters with ADAM algorithm (a stochastic gradient descent method).

A learning rate, a batch size and a number of epochs must be fixed, and cross-validation is performed in the splitting pf training and test data.

The encoder part of the convolutional autoencoder is also exploited to feed the FOM solutions into the network during learning. This action adds new parameters theta_E and changes the loss function used during training as follows:

$$\mathcal{L}(t^k, \boldsymbol{\mu}_i; \boldsymbol{\theta}) = \frac{\omega_h}{2} \|\mathbf{u}_h(t^k; \boldsymbol{\mu}_i) - \tilde{\mathbf{u}}_h(t^k; \boldsymbol{\mu}_i, \boldsymbol{\theta}_{DF}, \boldsymbol{\theta}_D)\|^2 + \frac{1-\omega_h}{2} \|\tilde{\mathbf{u}}_n(t^k; \boldsymbol{\mu}_i, \boldsymbol{\theta}_E) - \mathbf{u}_n(t^k; \boldsymbol{\mu}_i, \boldsymbol{\theta}_{DF})\|^2 .$$

Inputs and outputs of the DL-ROM are rescaled in the range [0,1]

NUMERICAL RESULTS

3. Multiscale Graph Neural Network Autoencoders for Interpretable Scientific Machine Learning

The aim is to solve two main limitations of autoencoders (AE) models though a novel architecture based on GNNs. The two issues are latent space interpretability and compatibility with unstructured meshes.

Reduction of the number of nodes is performed in the encoding phase producing interpretable latent graph representations, namely masked fields. This will address the first issue while about the second, multi-scale message passing (MMP) is employed.

In the context of model development, the autoencoding goal is to replace linear projection used in methods like POD with non-linear projection that improves the predictive accuracy of unsteady processes.

Autoencoding consists of 3 parts, encoder, integrator, decoder. The encoder projects the flow-field at the initial timestep onto a lower-dimensional space (latent space), the integrator solves the problem advancing in time in this simpler space either with physics-based models such as ODEs or with data-driven tools (neural ordinary differential equations, recurrent neural networks). The decoder maps the output of the integrator, low-dimensional solution at advanced timestep to the starting space of my model.

Interpretability, physical significance, and generalizability of variables in the latent space are the crucial points.

Most popular approaches in this framework are multi-layer perceptrons (MLPs), orthogonal basis projections via POD, or convolutional neural networks.

Limitations of these models lie in the incapability of being compatible with unstructured meshes corresponding to complex geometries while adjusting to variations of geometric configurations without an expensive additional training. Moreover data-based ROMs are essentially black-boxes that do not allow interpretability and all the benefits that derive from it.

GNNs are successful and useful thanks to generalizability and ambiguity in graph representations of data. In the framework of CFD modeling several results have shown how GNNs trained on one mesh can be easily adapted to different meshes. There is also an exploration of graph pooling techniques.

To summarize unsteady flow modeling is treated with a GNN-based autoencoder. The goal is latent space interpretability, and this is accomplished by employing Top-k pooling strategy. Moreover a series of multi-scale message passing (MMP) layers are included between pooling operations.

Graph based decoder and encoder are called E and D, input graph is G_0 and latent graph is G_L. The latent space, in this case corresponding to G_L, must be physically interpretable accordingly to the paper goal.

Main blocks of the network are MMP layers that model information exchange between neighboring nodes. Graph dimensionality reduction is performed by graph pooling layers, in particular Top-K pooling layers that through node sampling and sensor identification, produce interpretable latent graph representations.

The input graph is represented as a tuple (V_0, E_0, A_0) of nodes and edge features and the adjacency matrix. N0v, F0v, N0e, F0e are respectively the numbers of nodes, node features, edges and edge features.

From adjacency matrix A_0, neighboring nodes and node degree can be extracted.

Nodes are volumes centroids so that N0c is equal to the number of cells, in particular the node attributes correspond to the flow field snapshots. Edges are instantiated relying on shared cell faces, in addition edges that connect nodes within a user defined radius are introduced. The initial edge matrix feature presents the distance vectors connecting the sender and the receiver node of the edge.

The encoder has L levels, to each one corresponds a Graph G_l, with a number of nodes Nlv belonging to the range (N0v, Nlv). The ratio between N0v/Nlv is denoted as reduction factor RFG.

Before feeding nodes and edges features to the encoder, these features need to be encoded, this happens thanks to a multi-layer-perceptron (MLP) that embeds nodes and edges in a hidden feature space, with same dimension for both (F0v=F0e).

MMP layers do not modify graph connectivity A_l while they affect nodes and edges features. On the other hand, pooling operations modify connectivity.

Decoder unsampling consists of a series of graph unpooling layers that mirror the encoding operations, alternated with MMP layers with the same operations of the encoding stage but with different parameters.

In the end the final reconstructed flow field is recovered by a node-wise features decoder that reverses the embedding procedure used at the beginning of the encoder, mapping from the hidden channel to the original domain.

Every MMP is made of several message passing blocks, each on different coarse level with respect to the baseline mesh. Each block, in turn, presents multiple single-scale message passing (MP) layers.

In the first part of the MMP structure, after a message passing block, node and edge attributes are interpolated to a coarser graph level with a larger length scale, in the second attributes are interpolated back to the starting point.

The coarsening action is based on a voxel-clustering algorithm, a voxel grid is positioned over the domain, each voxel represents a cluster and its centroid will represent the new nodes of the coarse grid Nodes are clearly assigned using the distance between the centroids while edges are introduced in correspondence of an edge of the fine graph crossing the interface of two voxels. In particular, edges attributes are computed as the average of the ones belonging to this interface intersection.

About interpolation, a stencil composed by a new set of edges connecting children to parents nodes is used to map fine-to-coarse node attributes and vice versa.

Every MP layer composing the message passing block is built in 3 steps: 1. Edge Update, 2. Edge Aggregation, 3. Node Update.

This is the explicit structure pf the p-th MP layer (rk and sk are receiver and sender node indexes):

$$\begin{aligned} \text{Step 1 (Edge Update):} \quad & \mathbf{e}_k^p = f_e^p(\mathbf{e}_k^{p-1} | \mathbf{v}_{s_k}^{p-1} | \mathbf{v}_{r_k}^{p-1}), \quad k = 1, \ldots, n_e, \\ \text{Step 2 (Edge Aggregation):} \quad & \mathbf{a}_i^p = \frac{1}{|N(i)|} \sum_{\{k : r_k = i\}} \mathbf{e}_k^p, \quad i = 1, \ldots, n_v, \\ \text{Step 3 (Node Update):} \quad & \mathbf{v}_i^p = f_v^p(\mathbf{v}_i^{p-1} | \mathbf{a}_i^p), \quad i = 1, \ldots, n_v. \end{aligned}$$

fep and fvp are independent multi-layer perceptrons (MLPs).

The other building block of the GNN autoencoder is the Top-k pooling and unpooling layer.

The goal is to reduce the degrees of freedom of the system in terms of number of nodes. Adaptive node sampling is performed: node positions in the reduced graph coincide with the ones in the input graphs but the set of indexes is a function of the input node attributes (changing in time if the attributes are time evolving).

This allows for built-in interpretability of the latent graph G_L, that can be visualized oin the physical space thorough masked fields usage.

The points of strength of this approach are the compatibility with unstructured meshes and complex geometries and the possibility to adapt to any user defined regression problem.

Parameters needed for pooling are a learnable projection vector p_l and the number of nodes K for level l+1 (K=Nlv/RFl, RFl reduction factor at level l).

Firstly the feature node matrix is projected through the vector p_l to get a one-dimensional representation, then the resulting features are ranked in descending order and truncated after the K-th element. The K nodes associated to the selected features will be the retained sampled nodes of the level.

The set results $\mathcal{I}_{l+1} = \mathrm{rank}(\mathbf{y}_l, K)$, with $\mathbf{y}_l = \dfrac{\mathbf{V}_l \mathbf{p}_l}{\|\mathbf{p}_l\|} \in \mathbb{R}^{N_l^v \times 1}$.

Once obtained the set of nodes the reduced graph quantities V, E and A for the next level can be extracted.

While nodes of the input graph G_0 are physically fixed, nodes of the latent graph G_L can change in time since the projection is fixed but the input features are time dependent. Latent graphs can be visualized via masked fields that show the level index l of each cell in the domain.

In the end, unpooling is a data distribution operation:

$$\widetilde{\mathbf{V}}_l = \mathrm{distribute}(\mathbf{V}_{l+1}, \mathcal{I}_{l+1}, \mathbf{0}_l).$$

A zero-value matrix of original problem nodes dimension is filled in correspondence of the set of nodes I_l+1, an MMP layer after unppoling accounts for distributing values in the unpooled graph and filled zero-value nodes.

Code:

core_model	D defines several classes that are essential for constructing a Graph Neural Network (GNN) architecture. 1. **MLP (Multi-Layer Perceptron)** • This class implements a generic multi-layer perceptron, a type of feedforward neural network, that can be used for various transformation tasks within the GNN. It's characterized by its flexibility in terms of the number of layers and the size of each layer (input, hidden, and output channels).

	2. **EdgeModel** • The EdgeModel class is designed to update the features of the edges in the graph based on the current state of the nodes they connect and possibly their own features. This is crucial for tasks where edge attributes play a significant role in the graph's representation. 3. **NodeModel** • Similar to the EdgeModel but focused on the nodes, the NodeModel class updates the features of the nodes based on their own features, the features of their neighbors, and the features of the connecting edges. This class is key to learning node-level representations. 4. **MPLayer (Message Passing Layer)** • This class implements the message passing mechanism of GNNs, allowing nodes to send and receive information from their immediate neighbors. This process is central to the operation of GNNs, enabling the aggregation of local neighborhood information to learn representations. 5. **GNN (Graph Neural Network)** • The GNN class encapsulates the entire graph neural network model, likely utilizing the previously mentioned models (EdgeModel, NodeModel, MPLayer) to process the graph data. This class orchestrates how node and edge information is updated and aggregated to perform the desired task, whether it be node classification, graph classification, or another graph-based learning task.