EuroHPC
Joint Undertaking

Data driven Computational Mechanics at EXascale

DCoMEX

Data driven Computational Mechanics at EXascale

Work program topic: EuroHPC-01-2019
Type of action: Research and Innovation Action (RIA)

Report on DMAP and CAE-based surrogate models

DELIVERABLE D2.2

Version No 1

http://www.dcomex.eu/
This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking ('the JU'), under Grant Agreement No 956201

DOCUMENT SUMMARY INFORMATION

Project Title	**Data driven Computational Mechanics at EXascale**
Project Acronym	DCoMEX
Project No:	956201
Call Identifier:	EuroHPC-01-2019
Project Start Date	01/04/2021
Related work package	WP 2
Related task(s)	Task 2.2
Lead Organisation	NTUA
Submission date	08/05/2023
Re-submission date	
Dissemination Level	PU

Quality Control:

	Who	**Affiliation**	**Date**
Checked by internal reviewer	George Stavroulakis	NTUA	08/05/2023
Checked by WP Leader	Vissarion Papadopoulos	NTUA	08/05/2023
Checked by Project Coordinator	Vissarion Papadopoulos	NTUA	08/05/2023

Document Change History:

Version	**Date**	**Author (s)**	**Affiliation**	**Comment**
1.0	06.05.2023	Ioannis Kalogeris	NTUA	

Table of Contents

1 Description

Deliverable 2.2 is associated to **WP2 “Surrogate modelling”** of the DCoMEX project, and it provides a comprehensive report on the development of a novel surrogate modeling scheme based on the diffusion maps algorithm (DMAP). In the amended version of the proposal, an equivalent methodology has been proposed to build surrogate models of complex systems which relies on convolutional autoencoders (CAEs) instead of the DMAP algorithm. Both algorithms perform the same operation, each with its own merits and shortcomings, therefore two distinct surrogate modeling strategies were developed within DCoMEX and presented herein. The outline of the deliverable is the following: Section 2 illustrates the theoretical background behind the DMAP algorithm, its algorithmic implementation in the MSolve software and a surrogate modeling strategy based on DMAP. In a similar fashion, Section 3 presents the basic idea of CAEs, their algorithmic implementation in MSolve and the dedicated surrogate modeling strategy for complex engineering problems.

2 The diffusion maps algorithm

2.1 Theoretical background

Let $\boldsymbol{U} = [\boldsymbol{u_1}, \cdots, \boldsymbol{u_N}]$ be a data set consisting of vectors $\boldsymbol{u_i} \in R^d$, which can be seen as N distinct realizations of an R^d-valued random variable and sampled independently with density $q(\boldsymbol{u})$. Next, assume a connectivity measure K between data pairs $\boldsymbol{u}_i, \boldsymbol{u}_j$ such as the Gaussian kernel

$$K_\epsilon(\boldsymbol{u_i}, \boldsymbol{u_j}) = exp\left(\frac{-\left(\left\|\boldsymbol{u_i} - \boldsymbol{u_j}\right\|^2\right)}{4\varepsilon}\right)$$

Next, a discrete approximation to the Laplacian L_ε is constructed as follows:

- Estimate the densities q_ε at the sample points $\boldsymbol{u}_i$ as

$$q_\varepsilon(\boldsymbol{u}_i) = \frac{1}{N}\sum_{j=1}^{N} K_\varepsilon(\boldsymbol{u}_i, \boldsymbol{u}_j)$$

- Normalize the previously defined kernel K_ε as

$$\widehat{K_\varepsilon}(\boldsymbol{u}_i, \boldsymbol{u}_j) = \frac{K_\varepsilon(\boldsymbol{u}_i, \boldsymbol{u}_j)}{q_\varepsilon(\boldsymbol{u}_i)^\alpha q_\varepsilon(\boldsymbol{u}_j)^\alpha}$$

  Where for $\alpha = 1$ the discrete Laplacian approximates the Laplace-Beltrami operator, while $\alpha = 1/2$ approximates a diffusion operator.

- Estimate the new densities

$$\widehat{q_\varepsilon}(\boldsymbol{u}_i) = \frac{1}{N}\sum_{j=1}^{N} \widehat{K_\varepsilon}(\boldsymbol{u}_i, \boldsymbol{u}_j)$$

- If we define the matrix $\boldsymbol{K} = [K_{ij}] = \widehat{K_\varepsilon}(\boldsymbol{u}_i, \boldsymbol{u}_j)$ and the diagonal matrix $\boldsymbol{D} = [D_{ii}] = q_\varepsilon(\boldsymbol{u}_i)$, then the discrete approximation of the weighted Laplacian is given by the expression:

$$\boldsymbol{L}_{\varepsilon} = \frac{\boldsymbol{D}^{-1}\boldsymbol{K} - \boldsymbol{I}_{\boldsymbol{N}}}{\varepsilon}$$

The solution to the eigenvalue problem $\boldsymbol{L}_{\varepsilon}\boldsymbol{\psi} = \lambda\boldsymbol{\psi}$ will produce the sequence of eigenvalues $0 = \lambda_0 \geq \lambda_1 \geq \lambda_2 \geq \cdots$ and right eigenvectors $\boldsymbol{\psi}_j$ for the operator. In practice, only the first n non-trivial eigenvectors are kept with n obtained from the expression

$$n = argmin_{n,n\geq 2}\left(\frac{\lambda_1}{\lambda_n} < tol\right)$$

Then, the diffusion map operator $\Psi_{\varepsilon}: u \to R^n$can be defined as

$$\Psi_{\varepsilon}(\boldsymbol{u}) = \left[e^{\lambda_1\varepsilon}\psi_1(\boldsymbol{u}), e^{\lambda_2\varepsilon}\psi_2(\boldsymbol{u}), \ldots, e^{\lambda_n\varepsilon}\psi_n(\boldsymbol{u})\right]$$

2.2 Diffusion maps with variable-bandwidth kernels

In several data-driven applications, the samples follow some distribution which is unknown a priori. It is expected that the samples belonging to the tails of the distribution will be fewer and, thus, there will be regions on the manifold that will be more sparsely delineated. To address this issue in classical kernel methods the idea of the variable-bandwidth (or self-tuning) kernels has been proposed and illustrated herein. The main differentiation with respect to the classical DMAP algorithm lies in the form of the kernel used, which in this setting becomes:

$$K_{\varepsilon}^{VB}(\boldsymbol{u_i}, \boldsymbol{u_j}) = exp\left(\frac{-\left(\left\|\boldsymbol{u_i} - \boldsymbol{u_j}\right\|^2\right)}{\boldsymbol{4}\varepsilon\rho(\boldsymbol{u_i})\rho(\boldsymbol{u_j})}\right)$$

Following the construction for the graph Laplacian of the previous sections, in this case the sample densities are

$$q_{\varepsilon}^{VB}(\boldsymbol{u}_i) = \sum_{j=1}^{N}\frac{K_{\varepsilon}(\boldsymbol{u}_i, \boldsymbol{u}_j)}{\rho(\boldsymbol{u}_i)^m}$$

which are used to construct the kernel

$$K_{\varepsilon,\alpha}^{VB}(\boldsymbol{u}_i, \boldsymbol{u}_j) = \frac{K_{\varepsilon}^{VB}(\boldsymbol{u}_i, \boldsymbol{u}_j)}{q_{\varepsilon}^{VB}(\boldsymbol{u}_i)^{\alpha} q_{\varepsilon}^{VB}(\boldsymbol{u}_j)^{\alpha}}$$

Setting $q_{\varepsilon,\alpha}^{VB}(\boldsymbol{u}_i) = \sum_{j=1}^{N} K_{\varepsilon,\alpha}^{VB}(\boldsymbol{u}_i, \boldsymbol{u}_j)$, we can obtain the normalized kernel

$$\widehat{K_{\varepsilon,\alpha}^{VB}}(\boldsymbol{u}_i, \boldsymbol{u}_j) = \frac{\widehat{K_{\varepsilon,\alpha}^{VB}}(\boldsymbol{u}_i, \boldsymbol{u}_j)}{q_{\varepsilon,\alpha}^{VB}(u_i)}$$

and the weighted Laplacian for this formulation becomes

$$L_{\varepsilon,\alpha}^{VB}(\boldsymbol{u}_i, \boldsymbol{u}_j) = \frac{\widehat{K_{\varepsilon,\alpha}^{VB}}(\boldsymbol{u}_i, \boldsymbol{u}_j) - \delta_{ij}}{\varepsilon\rho(\boldsymbol{u}_i)^2}$$

2.3 Algorithmic implementation in the MSolve software

The code for implementing the variable-bandwidth diffusion maps algorithm can be found in https://github.com/mgroupntua/MSolve.MachineLearning [1]. In particular, the C# class DiffusionMapsAlgorithm.cs in the MGroup.MachineLearning folder implements the aforementioned procedure for an input data set. An example illustrating the use of this class is provided in the MGroup.MachineLearning.Tests folder, called DMAPexample.cs.

In this particular example, an initial data set is considered which consists of 2000 points in R^2, generated from a 2-dimensional Gaussian distribution centered at zero with covariance $C = 0.04I_2$. Using the syntax outlined below, a new object called DMAP from the DiffusionMapsAlgorithm class is generated, taking as input from the user a specified set of variables. Then the method ProcessData() applies the DMAP algorithm and computes the member variables DMAP.DMAPeigenvalues[·] and DMAP.DMAPeigenvalues[·].

- `dataSet` : the initial data set
- `numberOfKNN`: number of k-nearest neighbors used in the evaluation of the kernel $K_{\varepsilon,\alpha}^{VB}(\boldsymbol{u}_i, \boldsymbol{u}_j)$
- `numberOfKDE`: number of k-nearest neighbors required to estimate the kernel parameter ε
- `differentialOperator`: 1 – Laplace Beltrami operator, 2- generator of grad systems
- `numberOfEigenvectors`: The number of eigenvectors requested by the user

```
DiffusionMapsAlgorithm DMAP = new DiffusionMapsAlgorithm(dataSet,
numberOfKNN, NNofKDE, differentialOperator, numberOfEigenvectors);

DMAP.ProcessData();
```

The data used in this particular example are shown in figure 1, while figure 2 depicts the first 10 non-trivial DMAP eigenvalues.

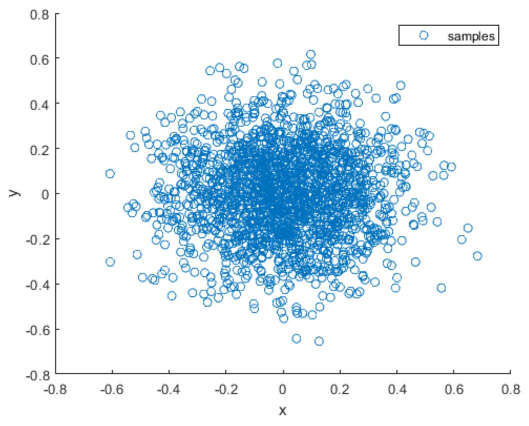

Figure 1: initial data samples

[1] The code has originally been submitted in the repo https://github.com/YiannisKalogeris/MSolve.MachineLearning

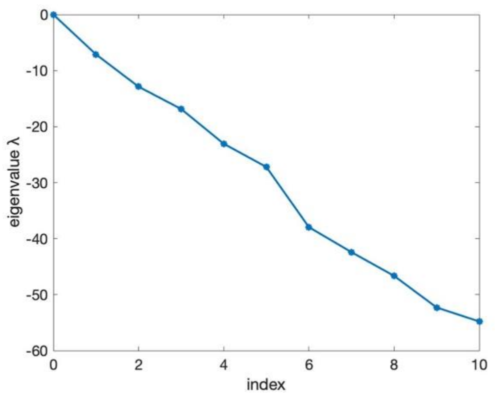

Figure 2: The first 10 diffusion map eigenvalues

2.4 Surrogate modeling scheme using diffusion maps and neural networks

Consider the modeling of a parametrized physical system governed by partial differential equations:

$$\frac{\partial u(\boldsymbol{x},t;\boldsymbol{\theta})}{\partial t} + \mathcal{L}[u(\boldsymbol{x},t;\boldsymbol{\theta})] = f(\boldsymbol{x},t;\boldsymbol{\theta}), \qquad \boldsymbol{x} \in \Omega, t \in [0,T], \boldsymbol{\theta} \in \Theta$$
$$\mathcal{B}[u(\boldsymbol{x},t;\boldsymbol{\theta})] = b(\boldsymbol{x},t;\boldsymbol{\theta}), \qquad \boldsymbol{x} \in \partial\,\Omega, t \in [0,T], \boldsymbol{\theta} \in \Theta$$
$$\mathfrak{C}[u(\boldsymbol{x},0;\boldsymbol{\theta})] = c(\boldsymbol{x};\boldsymbol{\theta}), \qquad \boldsymbol{x} \in \Omega, \boldsymbol{\theta} \in \Theta$$

where $u(\boldsymbol{x},t;\boldsymbol{\theta})$ is the (scalar) field of interest, $\mathcal{L}$ is a general differential operator that involves spatial derivatives and $f(\boldsymbol{x},t;\boldsymbol{\theta})$ is a source field. Also, $\mathcal{B}$ is the operator for the boundary conditions defined on $\partial\Omega$, $\mathfrak{C}$ is the operator for the initial conditions of the problem at $t = 0$ and $\boldsymbol{\theta} \in \Theta$ is the vector of the problem's parameters.

The discrete solution to the above set of equations for a given parameter instance $\boldsymbol{\theta}$ is obtained using numerical discretization techniques, such as the finite element method. In the context of parametric simulations, a set of parameter samples $\{\boldsymbol{\theta_j}\}_j^N$ is initially generated and the discretized solution vectors $\boldsymbol{u_j}(t_k) \in \mathbb{R}^d$ corresponding to each parameter instance $\boldsymbol{\theta_j}$ and time instance t_k are obtained through the numerical solution of the partial differential equation.

Based on the above, the numerical solution of the PDE can be regarded as a nonlinear mapping from the parametric space of $\mathbb{R}^d$. However, for a large-scale ($d \gg 1$) nonlinear dynamic problem, solving the corresponding numerical model can take significant computational resources and memory requirements. An intuitive approach to reduce this cost would be to exchange this mapping with another one that will be adequately accurate and much faster to compute. For this task, an feed-forward neural network, or FFNN for short trained over a reduced set of system solution would seem an appealing choice. Nevertheless, a FFNN with such high-dimensional output translates to a massive number of network parameters to be calibrated and the associated computational cost and memory demands would render this process unprofitable.

Our approach to overcome this issue is by employing the DMAP algorithm in order to reduce the dimensionality of the model's output, which will allow us to efficiently train a FFNN that will map points from the parametric space to their reduced representation in the DMAP space. Next, another mapping needs to established from the DMAP space to the high-dimensional solution space that will reconstruct the system's solutions from their reduced representations. For this purpose, in our investigation we chose the Laplacian Pyramids interpolation algorithm. The steps for the algorithmic implementation of the proposed procedure are outlined below.

1. Generate N samples of the vector of random parameters $\boldsymbol{\theta}$ and perform N simulations. Collect the solutions at N_t time increments to obtain the $d \times N_{tr}$ matrix of solutions (snapshots) $\boldsymbol{U} = [\boldsymbol{u_1}, \boldsymbol{u}_2, \dots, \boldsymbol{u}_{N_{tr}}]$, where $N_{tr} = N \times N_t$, N_t being the total number of time steps in the problem.

2. Given the matrix $\boldsymbol{U}$ of the N_{tr} distinct solutions, utilize the machinery provided by the diffusion maps algorithm to a identify a low dimensional representation $\boldsymbol{u} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^n$, with $n \ll d$.
3. Train a FFNN using $\{\boldsymbol{e}_i\}_{i=1}^{N_{tr}} = \{(\boldsymbol{\theta}_j, t_k)\}_{1\le j\le N, 1\le k\le N_t}$ as input and the DMAP coordinates $\{z_i\}_{i=1}^{N_{tr}}$ as output.
4. For a new vector of parameter values $\boldsymbol{e} = (\boldsymbol{\theta}, t)$ utilize the previously trained FFNN nonlinear mapping to obtain its image in the DMAP space as $FFNN(\boldsymbol{e}) = \boldsymbol{z} \in \mathbb{R}^n$.
5. Solve the pre-image problem with the Laplacian pyramid interpolation scheme to obtain the solution $\boldsymbol{u} = LP(\boldsymbol{z}) \in \mathbb{R}^d$

In the above, steps 1-3 constitute the offline (training) phase of the surrogate, while steps 4-5 are the online phase and these can be repeated for as many parameter instances as required for the purposes of the analysis. A schematic representation of the proposed surrogate modeling strategy is given in figure 3.

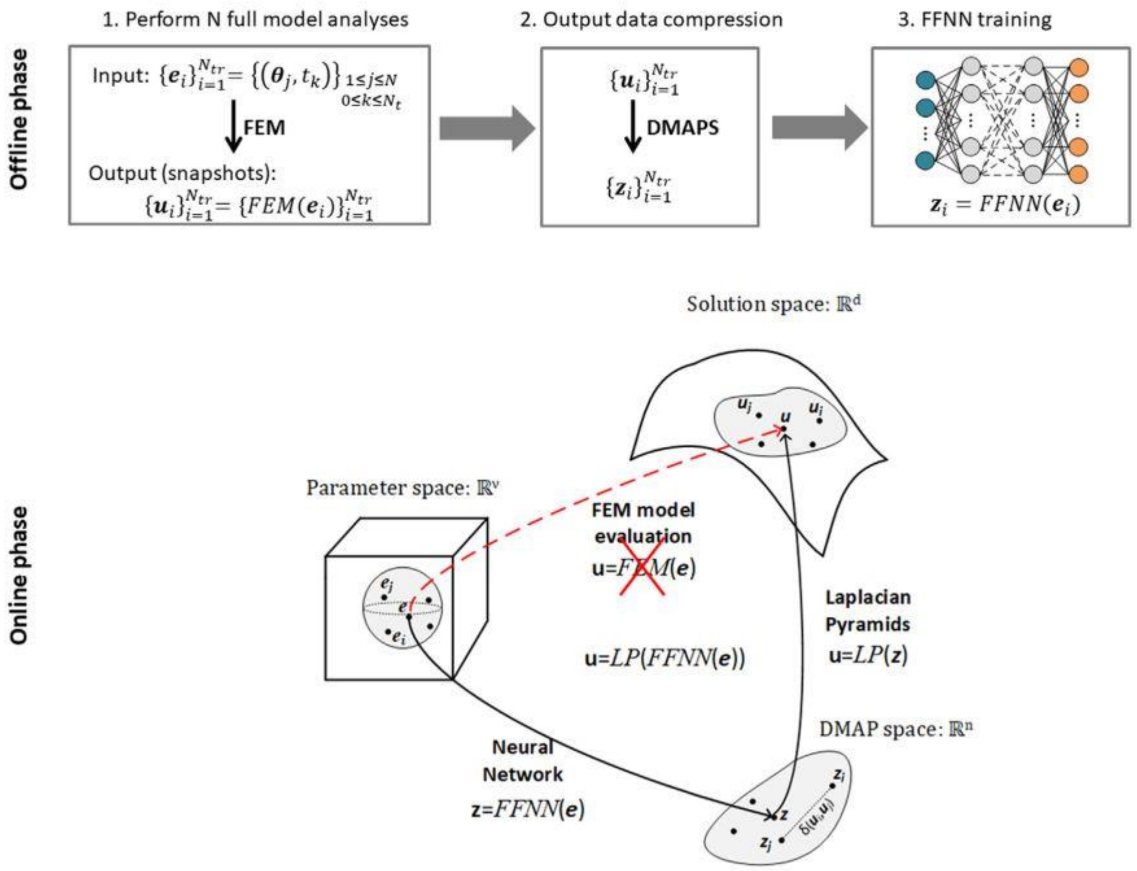
Offline phase
1. Perform N full model analyses
Input: $\{e_i\}_{i=1}^{N_{tr}} = \{(\theta_j, t_k)\}_{1\le j\le N, 0\le k\le N_t}$
FEM
Output (snapshots):
$\{u_i\}_{i=1}^{N_{tr}} = \{FEM(e_i)\}_{i=1}^{N_{tr}}$
2. Output data compression
$\{u_i\}_{i=1}^{N_{tr}}$
DMAPS
$\{z_i\}_{i=1}^{N_{tr}}$
3. FFNN training
$z_i = FFNN(e_i)$
Online phase
Solution space: $\mathbb{R}^d$
Parameter space: $\mathbb{R}^v$
FEM model evaluation
u=FEM(e)
u=LP($FFNN$(e))
Laplacian Pyramids
u=LP(z)
DMAP space: $\mathbb{R}^n$
Neural Network
z=$FFNN$(e)

Fig 3. Concept of the proposed surrogate modeling strategy

3 Convolutional autoencoders

3.1 Theoretical background

The Autoencoder (AE) concept is regarded as a neural network that learns from an unlabeled data set in an unsupervised manner. The aim of an AE is to learn a reduced representation for a set of data, known as encoding, and then learn how to reconstruct the original input from the encoded input with the minimum possible error. The latter part of the AE is called the decoder.

In particular, let $\boldsymbol{X}$ be a subset of $\mathbb{R}^{\mathrm{d}}$ with $\boldsymbol{x} \in \boldsymbol{X}$ denoting an element of the set. Then, the AE's encoder and decoder are defined as transition maps φ, ψ such that:

$$\varphi : \boldsymbol{X} \subseteq \mathbb{R}^{\mathrm{d}} \rightarrow \boldsymbol{H} \subseteq \mathbb{R}^{\mathrm{l}}$$
$$\psi : \boldsymbol{H} \subseteq \mathbb{R}^{l} \rightarrow \boldsymbol{X} \subseteq \mathbb{R}^{\mathrm{d}}$$
$$\varphi, \psi = argmin_{\varphi,\psi} \|\boldsymbol{X} - (\psi o \varphi)\boldsymbol{X}\|$$

with the dimension l typically being much smaller than d.

Now, let us consider the simplest case, where the encoder has only one hidden layer. It takes an input $\boldsymbol{x} \in \mathbb{R}^{\mathrm{d}}$ and sends it to $\boldsymbol{h} = \varphi(\boldsymbol{x}) \in \mathbb{R}^{\mathrm{l}}$ with $\boldsymbol{h} = \boldsymbol{\sigma}(\boldsymbol{Wx} + \boldsymbol{b})$, σ being an activation function (e.g. tanh or ReLU), $\boldsymbol{W}$ a weight matrix and $\boldsymbol{b}$ a bias vector. The image $\boldsymbol{h}$ of $\boldsymbol{x}$ is the latent or encoded representation of $\boldsymbol{x}$ and $\boldsymbol{H}$ is the latent or feature space.

The decoder's task is to establish the reverse mapping ψ that will reconstruct the input $\boldsymbol{x}$**,** given its latent representation $\boldsymbol{h}$**.** Again, considering a one-hidden layer, the reconstructed point $\widetilde{\boldsymbol{x}} = \psi(\boldsymbol{h})$ is given by: $\widetilde{\boldsymbol{x}} = \tilde{\sigma}(\widetilde{\boldsymbol{W}}\boldsymbol{h} + \widetilde{\boldsymbol{b}})$, with $\tilde{\sigma}, \widetilde{\boldsymbol{W}}, \widetilde{\boldsymbol{b}}$ being different than those of the encoder. Also, the network's architecture selected for the encoder can be different than that of the decoder and the number of hidden layers can be greater than one, leading to the so-called deep AEs.

AEs are trained by a back propagation algorithm, which is the most commonly used algorithm for the training of NNs. The training loss function becomes the reconstruction error between the input points $\boldsymbol{x}_i$ and their respective output $\widetilde{\boldsymbol{x}_\iota}$, that is:

$$L = \frac{1}{N}\sum_{i=1}^{N} \|x_i - \widetilde{x_\iota}\|$$

Despite their powerful dimensionality reduction properties, AEs face significant challenges when dealing with very high-dimensional inputs, due to the fact that the number of trainable parameters increases drastically with an increase in the input's dimensionality. In addition, AEs are not capable of capturing the spatial features of the input (e.g. when dealing with images) nor the sequential information in the input (e.g. when dealing with sequence data). To remedy these issues, a new type of AEs has emerged, that of convolutional autoencoders (CAEs). Similarly to AEs, CAEs also consist of an encoder and a decoder that are trained to minimize the loss function, but they are built from different layer types. Specifically, in CAEs the encoder part is built using a combination of convolutional layers, fully connected layers, pooling layers and normalization layers, while the decoder is built from deconvolutional layers and unpooling layers along with fully connected and normalization layers. Intuitively, CAEs can be viewed as extensions of ordinary AEs in the same way that CNNs are extensions of FFNNs.

Convolutional layers take as input a $n - D$ array $\boldsymbol{M}$ and apply a filter (a.k.a kernel) F of specified size to the elements of $\boldsymbol{M}$ in a moving window fashion. This process is schematically depicted in following figure.

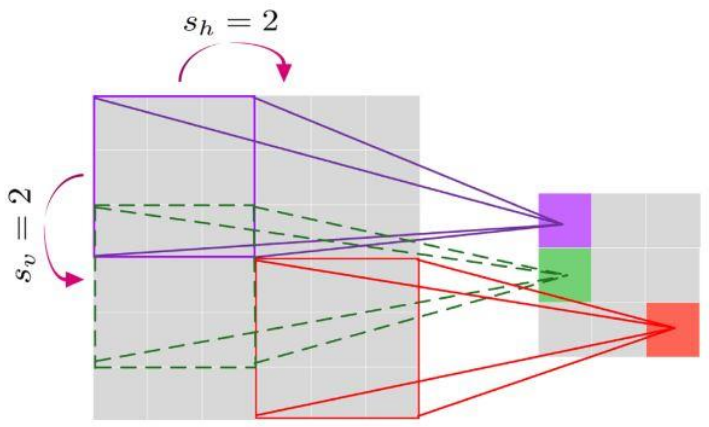

FIG. 4: Schematic representation of a $2-D$ convolutional filter with strides $s_h = 2$ and $s_v = 2$

To better clarify this process, let us consider a $2-D$ array $\boldsymbol{M} = [m_{ij}]$ and its encoded version $\boldsymbol{M}^{enc} = [\mu_{ij}]$, called feature map, which is obtained after applying a filter $\boldsymbol{W} = [w_{ij}]$ of size $f_h \times f_w$, moving with vertical stride s_v and horizontal stride s_h. The element μ_{ij} of $\boldsymbol{M}^{enc}$ is given by the equation:

$$\mu_{ij} = \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} m_{i'j'} \cdot w_{uv} + b \;\; with\; i' = i \times s_v + u,\; j' = j \times s_h + v$$

where b is the bias term and w_{uv} is the element of the filter $\boldsymbol{W}$ that gives the connection weight between elements of $\boldsymbol{M}^{enc}$ and the element of $\boldsymbol{M}$ within the respective window. This layer architecture is significantly more economical than that of a fully connected layer since the parameters involved are only the $f_h \times f_w$ elements of the filter $\boldsymbol{W}$ and the bias term b. The filter parameters do not require to be manually defined, instead the convolutional layer will learn the most appropriate filter for the task. Also, a convolutional layer can have multiple filters, in which case it outputs one feature map $\boldsymbol{M}_k^{enc}$ per each filter k. We shall write $\boldsymbol{M}^{enc} = ConvNN(\boldsymbol{M})$ to denote the application of several convolutional layers, with multiple filters each, to an array $\boldsymbol{M}$.

On the other hand, a deconvolutional layer performs the reverse operation of convolution, called deconvolution, and it is used to construct decoding layers. Their function is to multiply each input value by a filter elementwise. For instance a 2D $f_h \times f_w$ deconvolution filter maps a 1×1 spatial region of the input to an $f_h \times f_w$ region of the output. Thus, the filters learned in the deconvolutional layers create a basis used for the reconstruction of the inputs' shape, taking into consideration the required shape of the output. As before, a deconvolutional layer can have multiple filters, while several deconvolutional layers can be stacked for building deep architectures. The decoding procedure can be represented as: $\boldsymbol{M} = DeconvNN(\boldsymbol{M}^{enc})$.

Based on the above, the CAE's architecture consists of convolutional, deconvolutional and dense layers and is typically used for dimensionality reduction and reconstruction purposes. In practice, the CAE's encoder uses a number of convolutional layers to compress the input and once the desirable level of reduction has been achieved, the encoded matrix is flattened into a vector. Then, a dense layer is employed to map this vector to its latent representation. In the reverse direction, the decoder starts by taking the latent representation and transforming it into a vector through a denser layer. Subsequently, the input reconstruction is achieved by the deconvolutional layers. Thus, the loss for CAEs becomes:

$$L = \frac{1}{N} \sum_{i=1}^{N} \| \boldsymbol{M_i} - \widetilde{\boldsymbol{M_\iota}} \|$$

where $\boldsymbol{M}_i$ denotes the input arrays used for training and $\widetilde{\boldsymbol{M_\iota}} = DeconvNN(ConvNN(\boldsymbol{M}))$ the corresponding CAE's output. A schematic representation of a deep CAE is presented in figure 5.

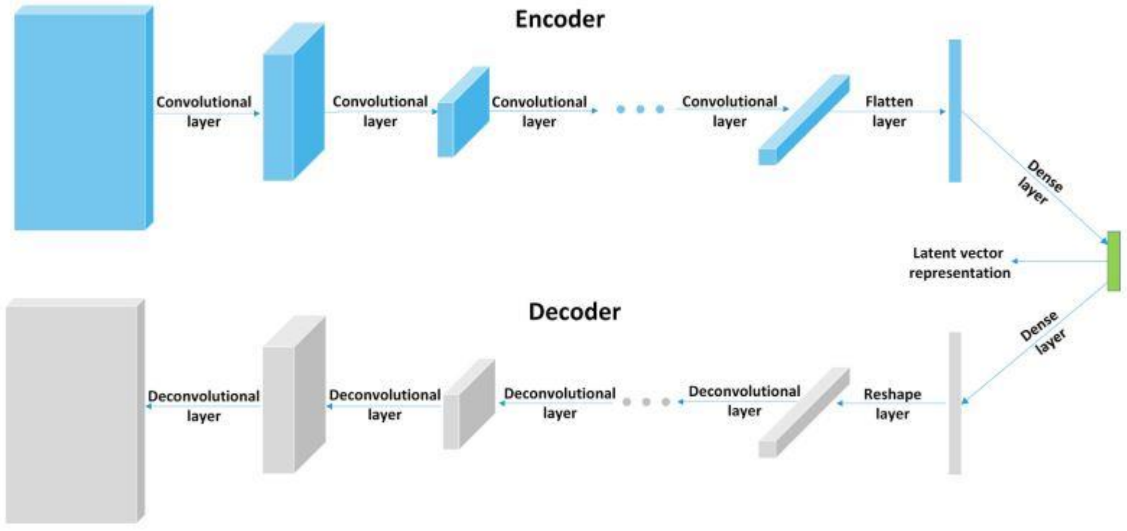

FIG.5: Schematic representation of a deep convolutional autoencoder

Lastly, aside of convolutional, deconvolutional and dense layers, two other important layer types often employed in CAEs are those of pooling and unpooling. Pooling layers are quite similar to convolutional layers in the sense that they downsample the input in order to decrease its size, however, they do not involve any trainable parameters. Their goal is to reduce the computational load, the memory usage, and the number of parameters. Common types of pooling layers include the max pooling layer and the average pooling layer. The first outputs the maximum value from the portion of the input covered by the filter and all other inputs are neglected. Accordingly, average pooling layers return the average from the portion of the input. On the other hand, unpooling layers perform the reverse operation of pooling and their aim is to reconstruct the original size of each rectangular patch. These operations are schematically depicted in figure 6.

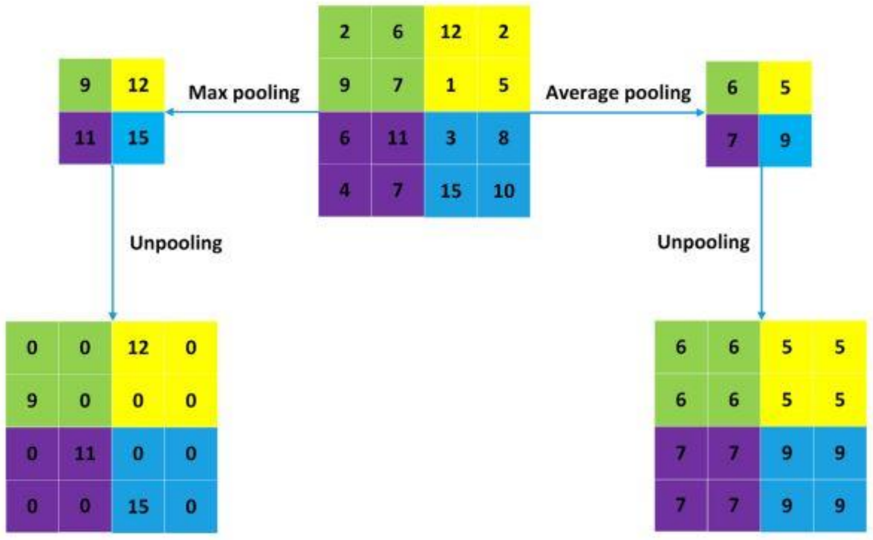

FIG. 6: Examples of pooling and unpooling

3.2 Algorithmic implementation in the MSolve software

The code regarding the Convolutional Autoencoders is located in https://github.com/mgroupntua/MSolve.MachineLearning. Specifically, the C# class **ConvolutionalAutoencoder** in the folder **NeuralNetworks** of the project **MGroup.MachineLearning.Tensorflow** can be readily used to implement a CAE.

ConvolutionalAutoencoder Constructor

ConvolutionalAutoencoder(normalization, optimizer, lossFunc, encoderLayers, decoderLayers, epochs, batchSize, seed, classification)

normalization: normalization option for the preprocessing stage of the CAE (e.g. MinMaxNormalization, ZScoreNormalization)

optimizer: optimization algorithm used for the CAE training (e.g. Adam, SGD, RMSProp)

lossFunc: loss function which will be minimized in the optimization procedure (e.g. MeanSquaredError, MeanAbsoluteError)

encoderLayers: network architecture regarding the layers of the encoder (e.g Convolutional2DLayer, MaxPooling2DLayer)

decoderLayers: network architecture regarding the layers of the decoder (e.g ConvolutionalTranspose2DLayer, UpSampling2DLayer)

epochs: total number of optimization iterations in terms of processing the full dataset

batchSize: total number of samples that will be processed in a single backpropagation step (default: 32)

seed: (*optional*)seed of the random number generator (default: null)

classification: controls the nature of the prediction i.e. for classification or regression (default: false)

ConvolutionalAutoencoder Methods

Train(trainX, testX)

trainX: train dataset (*remark*: *for an AE the input and output are identical*)

testX: (*optional*)test dataset (default: null)

method objective: to train a CAE with the provided train dataset

ValidateNetwork(testX)

testX: test dataset

method objective: to validate a trained CAE with the provided test dataset

MapFullToReduced(initialStimuli)

initialStimuli: collection of samples in the initial/full representation

method objective: to calculate the representation of the provided samples in the reduced space

MapReducedToFull(reducedStimuli)

reducedStimuli: collection of samples in the reduced/latent representation

method objective: to calculate the representation of the provided samples in the initial space

EvaluateResponses(initialStimuli)

reducedStimuli: collection of samples in the initial/full representation

method objective: to calculate the representation of the provided samples in the initial space after performing the full CAE procedure i.e. encoding and decoding

SaveNetwork(netPath, weightsPath, normalizationPath)

netPath: path of the operating system where the CAE architecture is saved to

weightsPath: path of the operating system where the CAE trained parameters are saved to

normalizationPath: path of the operating system where the normalization information is saved

method objective: to save the necessary information of after the training of a CAE, so it can be used in the future or shared

LoadNetwork(netPath, weightsPath, normalizationPath)

netPath: path of the operating system where the CAE architecture is loaded from

weightsPath: path of the operating system where the CAE trained parameters are loaded from

normalizationPath: path of the operating system where the normalization information is loaded from

method objective: to load the necessary information of an already trained CAE

ConvolutionalAutoencoder Example

class ConvolutionalAutoencoderTest

class objective: to illustrate the training accuracy of a CAE in the case of image classification. Specifically, it verifies the correct implementation and the accuracy of the CAE that has been trained on a subset of the MNIST dataset. Both the reduced representation and the reconstruction of the images are monitored.

3.3 Surrogate modeling scheme using convolutional autoencoders and neural networks

In a similar fashion to section 2.4, let us consider again the modeling of a parametrized physical system governed by partial differential equations:

$$\frac{\partial u(\boldsymbol{x},t;\boldsymbol{\theta})}{\partial t} + \mathcal{L}[u(\boldsymbol{x},t;\boldsymbol{\theta})] = f(\boldsymbol{x},t;\boldsymbol{\theta}), \qquad \boldsymbol{x} \in \Omega, t \in [0,T], \boldsymbol{\theta} \in \Theta$$
$$\mathcal{B}[u(\boldsymbol{x},t;\boldsymbol{\theta})] = b(\boldsymbol{x},t;\boldsymbol{\theta}), \qquad \boldsymbol{x} \in \partial\,\Omega, t \in [0,T], \boldsymbol{\theta} \in \Theta$$
$$\mathfrak{C}[u(\boldsymbol{x},0;\boldsymbol{\theta})] = c(\boldsymbol{x};\boldsymbol{\theta}), \qquad \boldsymbol{x} \in \Omega, \boldsymbol{\theta} \in \Theta$$

By applying spatial and temporal discretization techniques, the complete time-history response of the system is given by $\boldsymbol{U}(\theta) = \left[\boldsymbol{u}(\boldsymbol{\theta},t_1), \boldsymbol{u}(\boldsymbol{\theta},t_2), \ldots, \boldsymbol{u}\left(\boldsymbol{\theta},t_{N_t}\right)\right] \in \mathbb{R}^{d\times N_t}$, where N_t is the number of time increments in the temporal discretization.

The surrogate modeling approach proposed in this section is based on the powerful dimensionality reduction capabilities of CAEs. To this purpose, the PDEs are solved with the classic numerical procedure for a small, yet sufficient number, N, of parameter values in order to obtain a data set of time history matrices $\{\boldsymbol{U}_i\}_{i=1}^{N}$. The CAE (encoder and decoder) is trained over this data set minimizing the reconstruction mean square error. The encoded representation of each time history solution matrix $\boldsymbol{U}_i$ is a low dimensional vector $\boldsymbol{z}_i \in \mathbb{R}^l$ ($l \ll d$), which allows a FFNN to be trained accurately and efficiently in order to construct a mapping between the PDEs parametric space and the encoded solution space. It should be mentioned that the optimal architecture and hyperparameters of the CAE and FFNN are typically obtained via a trial and error procedure. After the training phase is completed, the proposed surrogate scheme works as follows. For a new input parameter vector, the encoded vector representation of the time history solution matrix is calculated by the FFNN and, subsequently, the entire time history matrix is delivered by the CAE's decoder. This way a large number of additional simulations can be performed at minimum computational cost.

The steps for the algorithmic implementation of the proposed procedure are outlined below.

1. Generate N samples of the vector of random parameters $\boldsymbol{\theta}$ and perform N simulations. Collect the solutions in three-dimensional array $N \times d \times N_t$, where d is the number of degrees of freedom and N_t the number of time increments.
2. Train a CAE over the N time history solution matrices $\boldsymbol{U_i} \in \mathbb{R}^{d\times N_t}$, collected at step 1, to obtain the encoded low dimensional vector representations $\boldsymbol{z}_i \in \mathbb{R}^l$ of these matrices along with the reconstruction map.
3. Train a FFNN to establish a mapping from the parameter space to the low dimensional encoded space, that is $\boldsymbol{z_i} = FFNN(\boldsymbol{\theta_i})$.
4. For a new vector of parameter values $\boldsymbol{\theta}$ utilize the trained FFNN to obtain the encoder vector representation $\boldsymbol{z} \in \mathbb{R}^l$ of the solution matrix.
5. The CAE's decoder is used to produce the solution matrix $\boldsymbol{U}$ based the encoded representation $\boldsymbol{z}$ of the previous step.

In the above, steps 1-3 constitute the offline (training) phase of the surrogate, while steps 4-5 are the online phase and these can be repeated for as many parameter instances as required for the purposes of the analysis. A schematic representation of the proposed surrogate modeling strategy is given in figures 7 and 8.

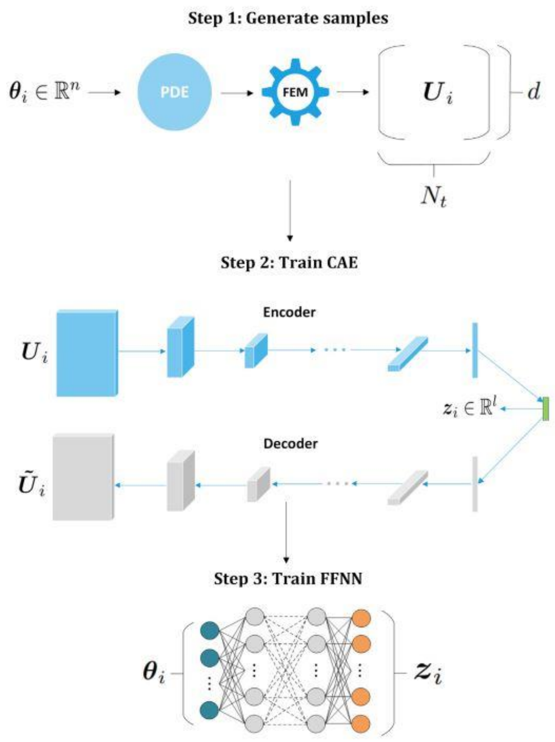

FIG.7: Offline phase of the proposed surrogate modeling method

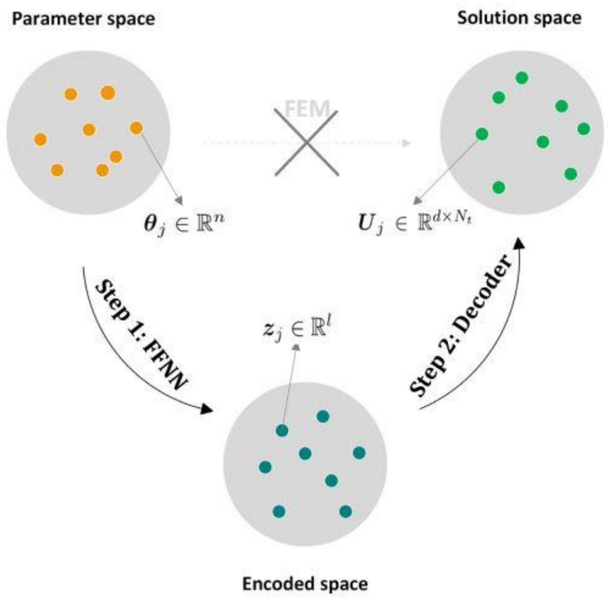

FIG. 8: Online phase of the proposed surrogate modeling method