POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

SCIENCES
SORBONNE
UNIVERSITÉ

ONERA
THE FRENCH AEROSPACE LAB

Accelerating Convergence of Linear Iterative Solvers using Machine Learning

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica

Author: **Luca Saverio**

Student ID: 990172
Advisor Politecnico di Milano: Prof. Nicola Parolini
Advisor Sorbonne Université: Prof. Corrado Maurini
ONERA Supervisors: Emeric Martin, Jorge Nuñez, Florent Renac

Academic Year: 2022-2023

Abstract

This thesis explores the application of Machine Learning techniques to accelerate iterative numerical methods, with a particular focus on the Generalized Minimal RESidual (GMRES) method, for solving arbitrary invertible linear systems. By training on the GMRES convergence behaviour obtained on previous linear systems in a sequence, the main goal of this work is to predict an adequate initial guess for each system of the sequence. This thesis is organized as follows: first, Machine Learning is applied to simple problems, such as the Laplace equation, where the right-hand side is modified at each iteration, and to the Advection-Diffusion problem with a time-dependent right-hand side. The matrix or operator does not change over the sequence. The model is trained using online learning techniques and the prediction of the initial guess results in a significant speed-up of the GMRES convergence on the considered linear systems. In the second part of the thesis, Neural Networks are applied to more complex and stiff systems, starting from the Navier-Stokes equations: the flow around a cylinder, the flow around a NACA0012 airfoil and the Taylor-Green Vortex test case are solved. The results of the classical GMRES algorithm are then compared to the ones of the Machine Learning scheme. The effectiveness of this approach is evaluated and compared to traditional methods, in terms of number of matrix-vector products to satisfy user parameters driving the stopping of the iterative solver. Considerations about gain in time taking into account the cost of the coupling and the sensitivity of certain parameters on the ML strategy performance are also addressed. Overall, this thesis demonstrates the potential of Machine Learning to improve the efficiency and accuracy of iterative numerical methods, particularly in the context of solving complex mathematical problems.

Keywords: GMRES, Neural Networks, Graph Neural Networks, CPU, GPU, Machine Learning, CFD, Performance, Pytorch

Abstract in lingua italiana

Questa tesi esplora l'applicazione delle tecniche di Machine Learning per accelerare i metodi numerici iterativi, con particolare attenzione al metodo del residuo minimo generalizzato, per la risoluzione di sistemi lineari arbitrariamente invertibili. L'obiettivo principale è quello di prevedere un'ipotesi iniziale adeguata all'algoritmo GMRES addestrando su precedenti sistemi lineari della sequenza. Questa tesi è organizzata come segue: in primo luogo, l'apprendimento automatico viene applicato a problemi semplici, come l'equazione di Laplace, in cui il lato destro viene modificato ad ogni iterazione per le stesse matrici, e al problema di Avvezione-Diffusione con un vettore lato destro dipendente dal tempo. Il modello viene addestrato utilizzando tecniche di apprendimento *online* mantenendo fissa la matrice o l'operatore. I risultati mostrano una significativa accelerazione nella convergenza. Nella seconda parte della tesi, le Reti Neurali vengono applicate a sistemi più complessi e rigidi, partendo dalle equazioni di Navier-Stokes: vengono risolti il flusso attorno ad un cilindro, il flusso attorno ad un profilo alare NACA0012 e il caso ddel vortice di Taylor-Green. I risultati del classico algoritmo GMRES vengono quindi confrontati con quelli dello schema di Machine Learning. L'efficacia di questo approccio viene valutata e confrontata con i metodi tradizionali, in termini di numero di prodotti matrice-vettore per raggiungere la convergenza del sistema lineare. Vengono inoltre affrontate considerazioni sul guadagno nel tempo con l'accoppiamento e la sensibilità di alcuni parametri sulle prestazioni dello schema ML. Nel complesso, questa tesi dimostra il potenziale del Machine Learning per migliorare l'efficienza e l'accuratezza dei metodi numerici iterativi, in particolare nel contesto della risoluzione di problemi matematici complessi.

Parole chiave: GMRES, Reti Neurali, Reti Neurali per Grafi, CPU, GPU, Machine Learning, Fluido-Dinamica Computazionale, Rendimento, Pytorch

Résumé en français

Ce mémoire porte sur l'utilisation des techniques d'apprentissage automatique pour accélérer les solveurs itératifs, en particulier l'algorithme GMRES (Generalized Minimal RESidual), dans le cas d'une suite de systèmes linéaires creux sans propriétés particulières. La phase d'apprentissage se base sur la convergence GMRES obtenue sur les premiers systèmes linéaires de cette suite. L'objectif principal de ce travail est de prédire pour chacun des systèmes restants de la suite une estimation pertinente de la solution. Dans un premier temps, l'apprentissage automatique est appliqué à des problèmes simples, tels que l'équation de Laplace, où le second-membre est modifié à chaque itération, et l'équation d'advection-diffusion, où le second-membre va dépendre du temps. La matrice ou l'opérateur ne change pas au cours de la suite. Le modèle est entraîné à l'aide de techniques d'apprentissage *online* et le fait de prédire une solution initiale plus adéquate du système entraîne une accélération significative de la convergence GMRES sur les systèmes linéaires considérés. Dans un second temps, les réseaux de neurones sont appliqués pour accélérer la résolution de systèmes plus complexes et rigides qui sont régis par des équations de Navier-Stokes: la simulation d'un écoulement autour d'un cylindre, la simulation d'un écoulement autour d'un profil aérodynamique NACA0012 et enfin le cas-test du tourbillon de Taylor-Green. Les résultats obtenus avec l'algorithme GMRES classique sont ensuite comparés à ceux obtenus avec la stratégie de résolution avec apprentissage automatique mise en place. L'efficacité de cette stratégie est évaluée et comparée aux méthodes traditionnelles, en termes de nombre de produits matrice-vecteur nécessaire pour satisfaire les paramètres utilisateur pilotant l'arrêt du solveur itératif. Des considérations sur le gain en temps prenant en compte le coût du couplage et l'influence de certains paramètres sur les performances de la stratégie proposée sont également abordées. Globalement, les travaux de ce stage démontre le potentiel des techniques d'apprentissage automatique pour améliorer l'efficacité et la précision des solveurs itératifs, notamment dans le contexte de la résolution de problèmes mathématiques complexes.

Keywords: GMRES, Réseaux Neuronaux, Réseaux Neuronaux Graphiques, CPU, GPU, Apprentissage Automatique, CFD, Performance, PyTorch

Contents

Introduction

Iterative Numerical Methods (INMs) are widely used to solve complex mathematical problems in various scientific and engineering applications. However, these methods can be computationally expensive and time-consuming, especially when dealing with large-scale systems. To overcome this challenge, researchers have been exploring the use of Machine Learning (ML) techniques to accelerate the convergence of INMs [20].

ML algorithms can learn patterns from large amounts of data and exploit them to make informed predictions or decisions. In the context of INMs, ML models can learn the relationships between the input parameters and the convergence behavior of the iterative solver. By using these models to predict the optimal parameters for the iterative solver, the convergence rate can be significantly improved, leading to faster and more efficient computations.

The Generalized Minimal Residual (GMRES) method is a widely used INM for solving non-symmetric, non-definite positive large linear systems of equations. It was introduced in 1986 by Saad and Schultz [33] as an extension of the Minimal Residual (MINRES) Method to non-symmetric matrices.

However, the GMRES method can be computationally expensive, especially for large-scale problems. One way to accelerate the convergence of GMRES is the use of preconditioners, which are operators that transform the original linear system into a more easily solvable form.

This thesis studies the possibility of using ML techniques to predict an optimal initial guess to then be fed into a GMRES solver, aiming to obtain an acceleration in the convergence.

The use of ML for accelerating INMs is an active area of research, with promising results in various fields such as fluid dynamics, structural mechanics, and computational electromagnetics. As the field of ML expands and continues to advance, it is plausible to expect further developments and improvements in this area, making it possible to look forward to faster and more accurate solutions to complex mathematical problems.

This study will be presented in five chapters: the first chapter will introduce the GMRES method and underline the importance of an effective initial guess, the second chapter will

present the main concepts of Machine Learning that were used for this work, the third chapter will focus on detailing the developed algorithm, finally the last two chapters will present the results obtained on simple and representative test cases.

ONERA: The French Aerospace Lab

This work has been developed at the National Office for Aerospace Studies and Research, in French Office National d'Études et de Recherches Aérospatiales (ONERA), the main French research center in the aeronautics, space and defense sector. ONERA was founded in 1946 [24] and it is placed under the supervision of the Ministry of the Armed Forces and it employs around 2000 people, including a majority of researchers, engineers and technicians. ONERA has significant test and calculation resources, and in particular the largest fleet of wind tunnels in Europe. It is composed of three main divisions: Defence, Aeronautics and Space Program.
ONERA's mission is to develop and guide research activities in the aerospace field, while designing, developing and deploying the resources required to conduct this research. Moreover, it aims to disseminate, in collaboration with the authorities or organisations responsible for scientific and technical research, the results of said research at national and international levels. Therefore, promoting their use by the aerospace industry and, where appropriate, facilitating their application outside the aerospace field.
The Technical and Programs Department (DTP) guarantees state expertise and carries out ONERA's studies and research relating to its main purposes, aeronautics, space and defence, via the seven scientific departments that make it up:

- DAAA - Aerodynamics, aeroelasticity, acoustics;
- DEMR - Electromagnetism and Radar;
- DMAS - Materials and structures;
- DMPE - Multi-physics for energetics;
- DOTA - Optics and associated techniques;
- DPHY - Physics, instrumentation, environment, space;
- DTIS - Information Processing and Systems.

This work was developed in the DAAA/NFLU department. NFLU develops numerical methods for solving Navier-Stokes equations, in particular space-time numerical schemes by finite volume approach and Discontinuous Galerkin, such as the Aghora code.

1 | Solving Linear Systems with the GMRES Method

Contents

Chapter 1 introduces the Iterative Generalized Minimal RESidual (GMRES) method, the primary iterative technique used in this thesis. The chapter presents the theoretical foundations and algorithmic principles of GMRES, highlighting its effectiveness in solving complex numerical problems. Additionally, the significance of selecting appropriate initial guesses in iterative methods is explored through numerical experiments, emphasizing the crucial role of thoughtful initialization strategies for efficient convergence. This chapter serves as a crucial foundation for subsequent sections, offering essential insights into the strengths and challenges of the GMRES method.

1.1. Iterative Methods for System Solution

In the field of computational mathematics, a system of partial differential equations can be approximated to a linear system in the form:

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{1.1}$$

where A represents the matrix of coefficients, of dimension $n_1 \times n_2$, $\boldsymbol{b}$ is the second member vector, or right hand side (RHS), of size n_1 and $\boldsymbol{x}$ represents the vector of unknowns to be determined, of size n_2. In this thesis all the problems considered are going to be

square, therefore A is going to be of size $n \times n$, while both $\boldsymbol{b}$ and $\boldsymbol{x}$ are going to be of size $n = n_1 = n_2$ and only real sparse linear systems will be considered.
In order to solve a linear system one could use different methods, for instance direct methods are methods which give the exact solution to the system in a finite number of steps, minus a rounding error produced by the limited capabilities of the machine, such as Gaussian elimination or the LU and Cholesky decomposition [12]. When the size of the matrix becomes larger, however, direct methods are no longer affordable due to memory requirements, therefore a more acceptable solution is to use iterative methods. Classic iterative methods are the Jacobi method, the Gauss-Seidel method and Krylov subspace-based methods [12, 32].
The basic idea behind iterative methods is to start with an initial guess for the solution and then repeatedly update the guess based on certain calculations. These calculations typically involve matrix-vector multiplications and vector operations. The process continues until a convergence criterion is reached, indicating that the approximation of the solution satisfies a certain level of accuracy.
Iterative methods can be advantageous in certain situations. They can be computationally efficient for large sparse systems where direct methods may be impractical due to the memory requirements. Additionally, iterative methods can be more flexible when dealing with systems that are ill-conditioned or have specific properties.
However, it is important to note that iterative methods may not always converge or may converge slowly. Preconditioning is then required and is used in order to attempt to improve the spectral properties of the operator of the system by clustering as much as possible the eigenspectrum of the initial system [5]. However, a tradeoff has to be found between numerical efficiency of the approach and costs to construct and apply the preconditioner. The convergence of an iterative method depends on the properties of the matrix involved in the linear system. Therefore, it is crucial to analyze the convergence behavior and select appropriate iterative methods accordingly.
This thesis will focus on the use of the GMRES method.

1.2. The Generalized Minimal RESidual Method

The Generalized Minimal RESidual method (GMRES) [10, 11, 32] is a classic iterative method which does not require any particular form of the matrix and is perfectly applicable to non symmetric and non definite positive systems. The method is considered iterative since at each step a new approximation of the solution of the system $\boldsymbol{x}$ is computed. However, the algorithm is designed so that at most n iterations are necessary to reach the solution $\boldsymbol{x}$ in exact arithmetic: GMRES could thus be qualified as a semi-exact

method.

One of the major advantages of the GMRES algorithm is the fact that the update on the iterated $\boldsymbol{x_i}$ (which is supposed to be an approximation of $\boldsymbol{x}$) is carried out in order to minimize the residual $\boldsymbol{r_i} := \boldsymbol{b} - A\boldsymbol{x_i}$ associated, in the Eulerian norm, and this with respect to all the other possible approximations on the space considered. The considered space being chosen so as to contain all the previous search spaces, we obtain the property that the residual only decreases (in norm) over the iterations. Stated otherwise, it is possible to say that this algorithm guarantees that at each new iteration, the newly computed $\boldsymbol{x_i}$ is "better" than the previous one (in the sense that the norm $\|\boldsymbol{r_i}\| := \|\boldsymbol{b} - A\boldsymbol{x_i}\|$ is smaller). In practice, the GMRES method will converge in far fewer steps than n, which makes it all the more interesting (and viable in a context of limited computer memory storage).

Consider an initial vector $\boldsymbol{x_0}$, which will serve as the starting point for our iterative method, in most practical cases, $\boldsymbol{x_0} = \boldsymbol{0}$. The associated initial residual is $\boldsymbol{r_0} := \boldsymbol{b} - A\boldsymbol{x_0}$. This method iteratively constructs a Krylov subspace, $\mathcal{K}_m(A, \boldsymbol{r_0})$, this operation is performed by generating an orthonormal basis of $\mathcal{K}_m(A, \boldsymbol{r_0})$ through the use of the Arnoldi procedure. The j-th Krylov space ($j \in \mathbb{N}^*$) generated by A and $\boldsymbol{r_0}$ is the vector subspace:

$$\mathcal{K}_j(A, \boldsymbol{r_0}) := \mathcal{L}\left\{\boldsymbol{r_0}, A\boldsymbol{r_0}, A^2\boldsymbol{r_0}, ..., A^{j-1}\boldsymbol{r_0}\right\} \tag{1.2}$$

and if $j = 0$, the Krylov space will be the space generated by the null vector.

The principle underlying the definition of Krylov space is based on the Cayley-Hamilton theorem and the notion of minimal polynomial [15]:

Theorem 1.1 (Cayley-Hamilton Theorem)**.** *Given $A \in \mathbb{R}^{n\times n}$, invertible, there exists a polynomial of degree less than or equal to n which cancels A. Therefore, there exists a polynomial $\mathcal{P}$ of degree less than or equal to n such that:*

$$A^{-1} = \mathcal{P}(A)\ . \tag{1.3}$$

It is this polynomial that the Krylov space seeks to reach, while multiplying it by the residual vector $\boldsymbol{r_0}$. Indeed, an element of Krylov space can well be written as a linear combination of $A^l\boldsymbol{r_0}$, that is to say as a polynomial in A multiplied by $\boldsymbol{r_0}$. As the Krylov space increases in size, it is expected in practice that one can more reliably reach the quantity $A^{-1}\boldsymbol{r_0}$ and thus solve our problem. Considerations related to the eigenvalues of the matrix of the system are also taken into account in the Krylov spaces (cf. [15] for more details).

The idea then is to construct vector by vector a basis of Krylov space. However, in practice, the vectors resulting from the calculations of $A^l \boldsymbol{r_0}$ will tend to be close to the linear dependence, which makes their use less robust, in particular for the resolution of large systems. Therefore, it is preferable or even necessary to generate an orthonormal basis of the considered Krylov space and it is done by the Arnoldi's procedure. The principle of Arnoldi's orthonormalization process is the same as that of a Gram-Schmidt algorithm, by adapting its formulation to our Krylov space. In Arnoldi's process, exact arithmetic is assumed, however, with the presence of round-off, it is not always reliable. The classical Gram-Schmidt method (CGS) is not very robust numerically, since the vectors obtained have a fairly strong loss of orthogonality [32]. The modified Gram-Schmidt principle (MGS) is therefore preferred, although it also happens that it is no longer sufficient: a reorthogonalization is then sometimes employed by repeating the orthogonalization step before normalizing. More than one reorthogonalization seems unnecessary according to some theoretical results [32]. Arnoldi's algorithm is written in detail in Algorithm 1.1 and MGS with only one orthogonalization step will be the default in this work.
By fixing the size of the Krylov subspace as m it is possible to limit the increasing cost in memory and the computational time of the procedure. Starting from an initial vector (denoted r_0 for more generality) and after m stages of the Arnoldi process, we obtain $m+1$ orthonormal vectors $\boldsymbol{v_1}, \boldsymbol{v_2}, ..., \boldsymbol{v_{m+1}}$ which constitute the columns of a matrix V_{m+1}, where $\boldsymbol{v_1} := \frac{\boldsymbol{r_0}}{\|\boldsymbol{r_0}\|}$. Note that in case of "*breakdown*" (*i.e.*, when a new vector obtained by the process is zero and it is not possible to divide it by its norm), this means that we have reached the maximum size of the Krylov space and thus reached the best approximation of A^{-1} by this method. It happens then it is not possible to perform m steps, but a lower number of steps. Even if they are rare, these "*breakdowns*" are to be taken into account in the implementations because they make it possible on the one hand to avoid useless calculations while convergence is reached, and on the other hand prevent the program from crashing when it was a very favorable case.
Noting that $(\cdot, \cdot)$ represents the classic scalar product.

Algorithm 1.1 One step of the Arnoldi Algorithm

Input A, V, $\bar{H}$, $\boldsymbol{v_1}$, j, m, ε
Output V, $\bar{H}$, m

$\boldsymbol{v_{j+1}} = A\boldsymbol{v_j}$
for $i = 1, ..., j$ **do**
  $h_{i,j} = (\boldsymbol{v_i}, \boldsymbol{v_{j+1}})$
  $\boldsymbol{v_{j+1}} = \boldsymbol{v_{j+1}} - h_{i,j}\boldsymbol{v_i}$
end for
$h_{j+1,j} = \|\boldsymbol{v_{j+1}}\|$
if $h_{j+1,j} \leq \varepsilon$ **then**
  $m = j$
else
  $\boldsymbol{v_{j+1}} = \boldsymbol{v_{j+1}}/h_{j+1,j}$
end if

Arnoldi's process allows us to write the following relation after m steps:

$$AV_m = V_{m+1}\bar{H}_m \ , \tag{1.4}$$

where $\bar{H}_m$ is an upper Hessenberg matrix of dimensions $(m+1) \times m$, whose coefficients store the inner products between the previous orthonormal vectors of the Krylov basis and the intermediate version of the new vector under construction. The interest of this relation is to be able to work without the multiplication by A once the matrix $\bar{H}_m$ has been obtained, especially since the latter is of small size (m is in practice very small compared to n).
After performing the m-th Arnoldi step, we are led to consider a Krylov space $\mathcal{K}_m(A, \boldsymbol{r_0})$. Starting from an initial iteration $\boldsymbol{x_0}$, the *search space* of an approximation $\boldsymbol{x_m}$ of the solution $\boldsymbol{x}$ will be:

$$\boldsymbol{x_0} + \mathcal{K}_m(A, \boldsymbol{r_0}) \ . \tag{1.5}$$

The principle of GMRES, as its name indicates, is to determine the unique $\boldsymbol{x_m}$ of the search space, such that it is the vector having the minimum norm $\|\boldsymbol{r_m}\| := \|\boldsymbol{b} - A\boldsymbol{x_m}\|$ on the search space. In other words, we are looking for the vector $\boldsymbol{y} \in \mathbb{R}^m$ such that:

$$\boldsymbol{x_m} = \boldsymbol{x_0} + V_m\boldsymbol{y} \tag{1.6}$$

and such that

$$\boldsymbol{r_m} = \underset{\boldsymbol{x_m} \in \mathbb{R}^n}{\arg\min} \|\boldsymbol{b} - A\boldsymbol{x_m}\| \,. \tag{1.7}$$

Now, $\|\boldsymbol{b} - A\boldsymbol{x_m}\| = \|\boldsymbol{b} - A\boldsymbol{x_0} - AV_m\boldsymbol{y}\| = \|\boldsymbol{r_0} - AV_m\boldsymbol{y}\|$ and according to the Arnoldi relation, $\|\boldsymbol{r_0} - AV_m\boldsymbol{y}\| = \|\boldsymbol{r_0} - V_{m+1}\bar{H}_m\boldsymbol{y}\|$. Moreover, $\boldsymbol{v_1} = \frac{\boldsymbol{r_0}}{\|\boldsymbol{r_0}\|}$ and it is orthogonal with all other $\boldsymbol{v_i}$, so is $\boldsymbol{r_0}$. So,

$$\|\boldsymbol{r_0} - V_{m+1}\bar{H}_m y\| = \|V_{m+1}(V_{m+1}^T\boldsymbol{r_0} - \bar{H}_m\boldsymbol{y})\| = \|V_{m+1}^T\boldsymbol{r_0} - \bar{H}_m\boldsymbol{y}\| = \|\boldsymbol{c} - \bar{H}_m\boldsymbol{y}\| \tag{1.8}$$

with $\boldsymbol{c} := ||\boldsymbol{r_0}||\boldsymbol{e_1} := \beta\boldsymbol{e_1}$ and $\boldsymbol{e_1}$ the first vector of the canonical basis.

It is possible to write:

$$\boldsymbol{r_m} = \boldsymbol{b} - A\boldsymbol{x_m} = \boldsymbol{b} - A(\boldsymbol{x_0} + V_m\boldsymbol{y}) = \boldsymbol{r_0} - AV_m\boldsymbol{y} \overset{1.4}{=} \beta\boldsymbol{v_1} - V_{m+1}\bar{H}_m\boldsymbol{y} = V_{m+1}(\beta\boldsymbol{e_1} - \bar{H}_m\boldsymbol{y}) \,. \tag{1.9}$$

Therefore, since the smallest the norm of the residual the more accurate the solution is, one could search for a solution by minimizing the functional:

$$J(\boldsymbol{y}) \equiv \|\boldsymbol{b} - A\boldsymbol{x}\| = \|\boldsymbol{c} - \bar{H}_m\boldsymbol{y}\| \,. \tag{1.10}$$

Therefore, one should solve:

$$\boldsymbol{y} = \underset{\tilde{\boldsymbol{y}} \in \mathbb{R}^m}{\arg\min} \|\boldsymbol{c} - \bar{H}_m\tilde{\boldsymbol{y}}\| \,. \tag{1.11}$$

Equation (1.11) is a least squares problem since the matrix $\bar{H}_m$ has one more row than columns. A usual solution method consists in determining the QR factorization of $\bar{H}_m$, with $Q \in \mathbb{R}^{(m+1)\times(m+1)}$ orthogonal and $R \in \mathbb{R}^{(m+1)\times m}$ upper triangular. So:

$$\|\boldsymbol{c} - \bar{H}_m\tilde{\boldsymbol{y}}\| = \|Q(Q^T\boldsymbol{c} - R\tilde{\boldsymbol{y}})\| = \|Q^T\boldsymbol{c} - R\tilde{\boldsymbol{y}}\| \,. \tag{1.12}$$

Algorithm 1.2 details the classic GMRES algorithm.

In order to obtain better solutions and not to halt the computation only after m iterations

Algorithm 1.2 GMRES(m, tol)

Input A: squared matrix of dimensions (n, n), $\boldsymbol{b}$: r.h.s. vector of the system of dimensions $(n, 1)$, $\boldsymbol{x_0}$: initial candidate of the solution vector of the system of dimensions $(n, 1)$, m: dimension of the Krylov subspace $\mathcal{K}_m(A, \boldsymbol{r_0})$

Output $\boldsymbol{x_m}$: approximated solution of the system

1: Compute $\boldsymbol{r_0} = b - A\boldsymbol{x_0}$, $\beta := \|\boldsymbol{r_0}\|$, and $\boldsymbol{v_1} := \boldsymbol{r_0}/\beta$
2: **for** $j = 1, ..., m$ **do**
3: Compute $\boldsymbol{w_j} := A\boldsymbol{v_j}$
4: **for** $i = 1, ..., j$ **do**
5: $h_{i,j} := (\boldsymbol{w_j}, \boldsymbol{v_i})$
6: $\boldsymbol{w_j} := \boldsymbol{w_j} - h_{i,j}\boldsymbol{v_i}$
7: **end for**
8: $h_{i+1,j} = \|\boldsymbol{w_j}\|$
9: **if** $h_{j+1,j} == 0$ **then**
10: $m := j$ and **go to** 14
11: **end if**
12: $\boldsymbol{v_{j+1}} = \boldsymbol{w_j}/h_{j+1,j}$
13: **end for**
14: Define the $(m + 1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{i,j}\}_{1 \leq i \leq m+1, 1 \leq j \leq m}$
15: Solve the least squares problem by computing $\boldsymbol{y_m}$, the minimizer of $\|\beta\boldsymbol{e_1} - \bar{H}_m\boldsymbol{y}\|$ and $\boldsymbol{x_m} = \boldsymbol{x_0} + V_m\boldsymbol{y_m}$, with $V_m = [\boldsymbol{v_1}|\boldsymbol{v_2}|...|\boldsymbol{v_m}]$
16: Compute the solution $\boldsymbol{x} = \boldsymbol{x_0} + V_m\boldsymbol{y}$

the Restarted GMRES algorithm was developed [32]. Restarted GMRES is an extension of GMRES which introduces an outer cycle to periodically restart the optimization process. Algorithm 1.3 shows in detail the Restarted GMRES (RGMRES) algorithm.

Algorithm 1.3 RGMRES(m, tol)

Input A: squared matrix of dimensions (n, n), $\boldsymbol{b}$: r.h.s. vector of the system of dimensions $(n, 1)$, $\boldsymbol{x_0}$: initial candidate of the solution vector of the system of dimensions $(n, 1)$, m: dimension of the Krylov subspace $\mathcal{K}_m(A, \boldsymbol{r_0})$, N_{max}: maximum number of restarts

Output x_m: approximated solution of the system

1: Compute $\boldsymbol{r_0} = b - A\boldsymbol{x_0}$, $\beta := \|\boldsymbol{r_0}\|$, and $\boldsymbol{v_1} := \boldsymbol{r_0}/\beta$
2: Generate the Arnoldi basis and the matrix $\bar{H}_m$ using Algorithm 1.1 starting from v_1
3: Compute $\boldsymbol{y_m}$ the minimizer of $\|\beta\boldsymbol{e_1} - \bar{H}_m\boldsymbol{y}\|$ and $\boldsymbol{x_m} = \boldsymbol{x_0} + V_m\boldsymbol{y_m}$
4: If satisfied or N_{max} is reached then **break**, else set $\boldsymbol{x_0} := \boldsymbol{x_m}$ and **go to** 1

1.3. The Initial Guess of Iterative Methods

The effectiveness of iterative methods depends on the initial guess $\boldsymbol{x_0}$ used as a starting point for the iterative process. While the choice of initial guess may appear arbitrary,

it significantly influences the convergence behavior of iterative linear solvers. A more favorable initial guess can reduce the number of iterations required to achieve convergence, thus enhancing computational efficiency.
By selecting an initial guess that is closer to the true solution, the number of iterations required to reach convergence can be significantly reduced. Several techniques can help in obtaining an improved initial guess, including utilizing previous solutions, incorporating problem-specific information, or employing preconditioners to modify the system matrix.

1.3.1. Recycling the Previous Solution in Time-Dependent Simulations

Using the term recycling, in the field of iterative methods for solving linear systems obtained from time-dependent problems, one refers to the utilization of the solution obtained from the previous time iteration as an initial guess for the current iteration. This approach aims to accelerate the convergence of the iterative solver and reduce the overall computational cost. While the initial guess for the first iteration is typically an arbitrary or zero vector.
This operation is possible because linear systems in consecutive time steps are often related or similar, making the previous solution a reasonable starting point for the current iteration. By leveraging the information contained in the previous solution, such as the approximate structure of the problem or the dominant modes of the system, the recycling technique can significantly improve the convergence behavior of the iterative method.
This process is repeated for subsequent time steps, with each recycled solution providing an improved initial guess for the corresponding linear system. By recycling the solution from the previous time iteration, the iterative solver benefits from the information already captured in the previous solution, leading to faster convergence and reduced computational effort.

1.4. Graph Representation of Matrices

Each matrix $A = \{a_{ij}\}$ can be represented as a graph, specifically the connectivity of the graph is generated by the adjacency matrix of A, defined as Adj: $adj_{ij} = 1$ if and only if $|a_{ij}| > 0$. Subsequently, a self-loop is added to each node. The non diagonal elements of A can also be assigned to the corresponding edges as edge features, while the diagonal elements are attached to the corresponding self-loops. The input and result of this algorithm are shown in Figure 1.1. This procedure is defined because it serves the purpose of utilizing matrix information through a graph structure, which can then be use

in the implementation of a Graph Neural Network, see Section 2.3.3 and Section 3.4.2.

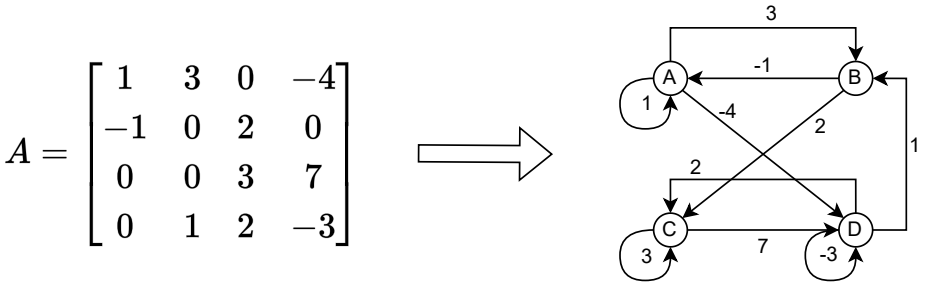


Figure 1.1: Generation of a graph from a matrix.

This conversion was also performed by [36] to exploit matrix features in the prediction of the best suited preconditioner for a linear system.

2 | Machine Learning and Neural Networks

Contents

Chapter 2 provides an introduction to Machine Learning, covering the core concepts and its importance in data-driven problem-solving. It discusses two primary learning paradigms, offline and online learning, highlighting their respective data processing and adaptability approaches. Additionally, the chapter defines various Neural Network types and their applications, facilitating informed choices for experimentation. Finally, the concept of the loss function and its role in guiding model optimization during training is explained. These foundational insights pave the way for the subsequent chapters, where Machine Learning techniques are applied and evaluated.

2.1. Introduction to Machine Learning

Machine Learning (ML) is a subfield of artificial intelligence (AI) that focuses on the development of algorithms and models that enable computers to learn and make predictions

or decisions without being explicitly programmed to. It provides computational systems the ability to automatically learn and improve from experience, allowing them to handle complex tasks and data-driven problems.
At its core, ML revolves around the idea of creating mathematical models that can learn patterns and relationships from data. By analyzing and processing large amounts of data, these models can uncover insights, make predictions, and make data-driven decisions.
The fundamental goal of ML is to enable computers to learn from data and generalize that knowledge to new, unseen examples. This process involves two main components: training and inference. During the training phase, a ML model is exposed to a labeled dataset, where it learns from the provided examples and adjusts its internal parameters or structure to capture the underlying patterns in the data. Once the model is trained, it can be deployed for inference, where it makes predictions or decisions based on new, unseen data.
Deep learning (DL) is a subset of ML that uses artificial Neural Networks (NNs) to mimic the learning process of the human brain. DL consists of multiple hidden layers in an artificial NN. This approach tries to model the way the human brain processes light and sound into vision and hearing. NNs are introduced and defined in Section 2.3.
ML algorithms can be categorized into different types, including supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning.

- Supervised Learning: the model learns from labeled examples, where the input data is paired with corresponding target labels or outcomes. This type of learning is used for tasks such as classification, regression, and sequence labeling;
- Unsupervised Learning: it involves learning patterns and structures from unlabeled data. The goal is to discover inherent relationships, clusters, or hidden patterns within the data. Unsupervised learning algorithms are commonly used for tasks like clustering, dimensionality reduction, and anomaly detection;
- Semi-supervised Learning: it combines elements of both supervised and unsupervised learning, where the model learns from a combination of labeled and unlabeled data. This approach is beneficial when labeled data is limited or expensive to obtain;
- Reinforcement Learning: it involves training agents to interact with an environment and learn optimal actions through a trial-and-error process. The agent receives feedback in the form of rewards or punishments based on its actions, enabling it to learn and improve its decision-making abilities over time. Reinforcement learning is often applied to problems in robotics, game playing, and control systems.

ML algorithms can be either offline or online learning algorithms. In offline learning, the

algorithm is trained on a pre-collected batch of data, and the training process does not take into account any new data that may be received during deployment. In contrast, online learning involves the continuous training of a model on incoming data, allowing it to adapt to new information and make more accurate predictions.
ML has a wide range of practical applications across various domains. It is used in areas such as image and speech recognition, natural language processing, recommendation systems, fraud detection, financial modeling, medical diagnosis, autonomous vehicles, and many more.
In recent years, advancements in computing power, the availability of large datasets, and breakthroughs in algorithms, such as deep learning, have accelerated the progress and adoption of ML. These developments have enabled the creation of highly sophisticated models capable of handling complex tasks and achieving state-of-the-art performance in various domains.
As ML continues to evolve, researchers and practitioners explore new techniques and algorithms to address challenges such as interpretability, fairness, robustness, and scalability. The field holds immense potential for transforming industries, driving innovation, and shaping the future of technology.
Apart from NNs, several other models exist for implementing ML. In this thesis NNs are preferred thanks to recent advancements in GPU-based implementations of backpropagation algorithms that have made them much more efficient.

2.2. Different Learning Modes

2.2.1. Offline Learning

Offline learning, also known as batch learning or batch-mode learning, is a ML paradigm where the learning algorithm is trained on a pre-collected batch of data before it is deployed for prediction or inference tasks. In offline learning, the entire dataset, often referred to as the training set, is available from the start and is used to train the model. The offline learning process typically consists of the following steps:

1. Data Collection: the training dataset is gathered by collecting and assembling relevant examples or instances that represent the problem domain. This dataset should be diverse, representative, and cover a wide range of scenarios that the model is expected to encounter during deployment.
2. Data Preprocessing: the collected dataset is processed to ensure its quality, con-

sistency, and suitability for training. This step may involve tasks such as cleaning the data, handling missing values, normalizing or scaling features, and performing feature engineering to extract relevant information.

3. Model Training: once the data preprocessing is complete, the training algorithm is applied to the entire dataset. The model is built by learning the underlying patterns and relationships within the training data. The goal is to create a model that can generalize well to unseen data and make accurate predictions or classifications.

4. Model Evaluation: after the model is trained, it needs to be evaluated to assess its performance and effectiveness. This is typically done by using a separate portion of the dataset called the validation set or test set. The model's predictions on this set are compared to the known true values to measure metrics such as accuracy, precision, recall, or F1 score. These metrics provide an indication of how well the model is likely to perform on new, unseen data.

5. Model Deployment: once the model has been evaluated and deemed satisfactory, it can be deployed for making predictions or classifications on new, unseen instances. The trained model takes input data and produces the desired output, based on the patterns it has learned during training.

One of the main characteristics of offline learning is that it assumes the availability of the entire training dataset before the learning process begins. This implies that offline learning algorithms do not consider new data that arrives after the training phase. Therefore, offline learning may not be suitable for applications where the data distribution changes over time or where a continuous stream of data needs to be processed.
Overall, offline learning provides a way to train models using a fixed dataset, making it well-suited for scenarios where data is static or does not change significantly during deployment.

2.2.2. Online Learning

Online learning is particularly useful in scenarios where the data distribution may change over time, or when there is a large and continuous stream of data. One example of this is in financial applications, where stock prices or other financial data are constantly changing and new data is continuously being generated. Another example is in natural language processing, where the language and vocabulary used by people are constantly evolving.
One of the key differences between online and offline learning is the way in which data is processed. In offline learning, the entire dataset is typically loaded into memory and processed in batches. This approach can be computationally intensive and time-consuming,

particularly when dealing with large datasets. In contrast, online learning algorithms process data in real-time, as it arrives, making them more efficient and scalable.
Another key difference between online and offline learning is the way in which the model is updated. In offline learning, the model is typically trained once on a fixed dataset, and the resulting model is used for prediction. In contrast, online learning algorithms update the model in real-time as new data arrives, allowing the model to adapt to changes in the data distribution.
Online learning algorithms also typically require less memory and computational resources than offline learning algorithms. This is because they only need to store and process a small subset of the data at any given time, rather than the entire dataset.
Therefore, online learning in ML allows for the continuous training of models on incoming data, making them more adaptable to changes in the data distribution. Online learning algorithms differ from offline learning algorithms in the way they process data, update the model, and require fewer computational resources.

2.3. Neural Networks

Neural Networks (NNs) are a fundamental concept in the field of ML and DL. They are powerful computational models inspired by the structure and functioning of the human brain, composed of interconnected nodes called artificial neurons or simply neurons.
At its core, a NN consists of multiple layers of interconnected neurons, forming a network-like structure. The most common type of NN is the feedforward NN, where information flows in one direction, from the input layer through one or more hidden layers to the output layer.
Each neuron in a NN receives input from the previous layer's neurons, processes the information, and then produces an output. These outputs are then passed as inputs to the neurons in the next layer, and this process continues until the final output is generated. The connections between neurons are represented by weights $\boldsymbol{w} = \{w_j\}_{j=1}^{n}$ and a bias $\boldsymbol{b} = \{b_j\}_{j=1}^{n}$, which determine the strength and influence of each connection. Each node j of a layer can be represented by a combination of a scalar product and activation function z, such that:

$$y_j = z(\boldsymbol{w} \cdot \boldsymbol{x} + b_j) \ , \tag{2.1}$$

where $\boldsymbol{x} = \{x_j\}_{j=1}^{n}$ is the input vector of the node and y_j is the j-th scalar component of the output vector $\boldsymbol{y}$ corresponding to the j-th node of the considered layer.

During the training phase of a NN, the weights are adjusted iteratively to minimize the difference between the network's predicted output and the desired output. This adjustment is achieved through a process called backpropagation, which calculates the gradient of the network's performance with respect to the weights and updates them accordingly using optimization algorithms like gradient descent. This operation is made possible by Automatic Differentiation, which is set of techniques to evaluate the partial derivative of a function specified by a computer program.
The hidden layers of a NN enable it to learn complex and abstract representations of the input data. By adjusting the weights, NNs can discover intricate patterns and relationships within the data, making them capable of tasks such as classification, regression, and even more advanced tasks like image and speech recognition.
One of the significant advantages of NNs is their ability to learn and generalize from large amounts of data. This property, known as *learning from examples*, allows neural networks to make predictions or decisions on new, unseen data based on patterns learned during training.
NNs have witnessed tremendous success in various domains, including computer vision, natural language processing, speech recognition, and recommendation systems, among others. Their versatility and ability to handle complex problems have made them a central component of modern ML and AI applications.
It's worth noting that neural networks come in different architectures and variations, such as Dense Neural Networks (DNNs), Convolutional Neural Networks (CNNs) used largely for image processing, Graph Neural Networks (GNNs), each designed to excel in specific tasks and data types.
Overall, NNs are a crucial and powerful tool in the field of ML, enabling computers to learn from data and recognize patterns.

2.3.1. Dense Neural Networks

Dense Neural Networks (DNNs) have been a cornerstone in the field of ML, providing effective solutions for a wide range of tasks. DNNs excel at processing sequential and tabular data.
DNNs, also known as feedforward NNs, are composed of multiple layers of interconnected neurons, as shown in Figure 2.1. The architecture follows a sequential flow of information, with each neuron in a layer connected to all neurons in the subsequent layer. The input data propagates through the network in a forward direction, without any recurrent connections. DNNs leverage the power of activation functions, weight parameters, and deep layer structures to learn complex representations and perform various tasks.

The architecture of a DNN typically comprises an input layer, one or more hidden layers, and an output layer. Neurons in each layer are fully connected to neurons in the subsequent layer, resulting in a dense connectivity pattern. The hidden layers, with their nonlinear activation functions, allow DNNs to capture intricate patterns and relationships within the data. Different architectures, such as Multilayer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs), have been developed to address specific types of data and tasks.

Training DNNs involves optimizing an objective function through a process known as backpropagation. Backpropagation calculates the gradients of the loss function with respect to the network parameters, enabling the update of weights using gradient descent optimization algorithms such as the ADAM (ADAptive Moment Estimation) algorithm. The availability of large-scale labeled datasets, advancements in optimization techniques, and the use of regularization methods, such as dropout and weight decay, have contributed to the success of training DNNs.

The evaluation of DNNs involves assessing their performance on specific tasks using appropriate metrics, such as accuracy, precision, recall, and F1 score [38]. Datasets with ground truth labels enable the training and evaluation of DNN models. Experimental analyses help researchers understand the impact of hyperparameters, network architectures, and training strategies on the performance of DNNs.

While DNNs have demonstrated exceptional performance, they face several challenges. Overfitting, where the model fails to generalize well to unseen data, remains a concern. The interpretability and explainability of DNNs have also raised questions, as the complexity of their architectures often leads to black-box decision-making. Adapting DNNs to handle limited labeled data, addressing the computational requirements of training large-scale models, and enhancing their robustness against adversarial attacks are areas of ongoing research. The exploration of novel architectures, such as transformers and generative models, further expands the potential applications of DNNs.

DNNs provide a powerful framework for processing sequential and tabular data, offering exceptional performance in various domains. Their dense connectivity pattern, activation functions, and deep layer structures enable them to learn complex representations and solve a wide range of tasks.

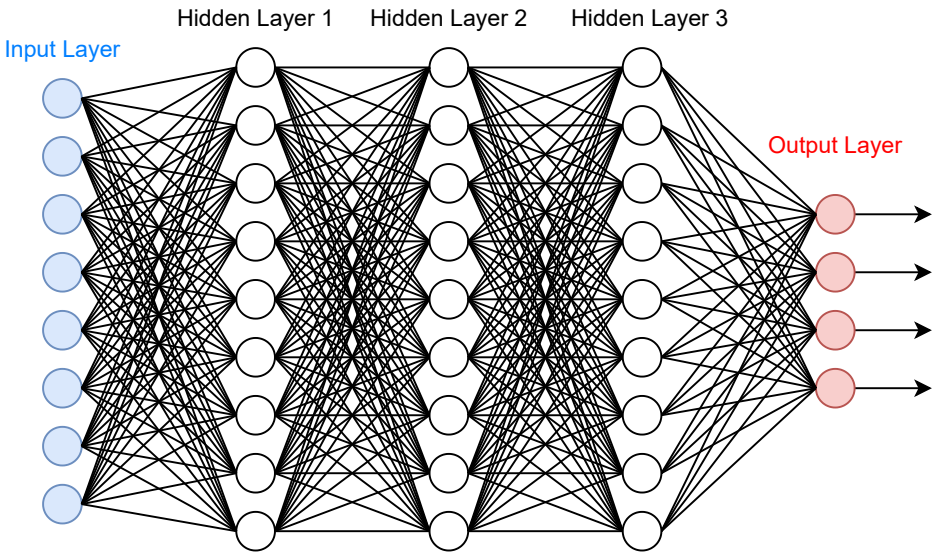


Figure 2.1: General structure of a Dense Neural Network.

2.3.2. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [25] have revolutionized the field of computer vision and have become a fundamental tool for image and video analysis. These DL models are specifically designed to capture spatial and hierarchical patterns in data, making them highly effective in tasks such as image classification, object detection, and image segmentation.

CNNs are a class of DNNs that excel in processing grid-like data, such as images and videos. Inspired by the organization of the visual cortex in humans and animals, CNNs employ convolutional layers that extract local spatial features from the input data. By leveraging shared weights and pooling operations, CNNs can efficiently learn hierarchical representations, enabling robust and accurate analysis of visual data.

The architecture of a CNN typically consists of convolutional layers, pooling layers, fully connected layers, and an output layer. Convolutional layers employ convolution operations, applying filters to the input data to extract relevant features. Pooling layers reduce the spatial dimensions of the feature maps, capturing the most salient information. There are two main types of pooling applied to CNNs: average and max pooling [39]. Fully connected layers further process the extracted features, learning high-level representations and making final predictions. The depth and width of the network, as well as the size of filters and pooling regions, can vary based on the complexity of the task and the available computational resources.

CNNs are trained using large-scale labeled datasets through a process known as backprop-

agation. During training, the network learns to optimize an objective function, typically a loss function, by adjusting the weights and biases. Gradient descent optimization algorithms, such as Stochastic Gradient Descent (SGD) [30] and its variants, are commonly employed to update the parameters of the network. Regularization techniques, such as dropout and weight decay, help prevent overfitting and improve generalization performance.

As mentioned earlier, CNNs have been widely applied in computer vision tasks, as shown in Figure 2.2, where the geneneral structure of a CNN is also presented. Image classification, where CNNs assign labels to images based on their content, has seen remarkable advancements with models like AlexNet, VGGNet [34] and ResNet [13]. Object detection involves localizing and classifying objects within images, and CNN-based architectures such as R-CNN, Fast R-CNN, and YOLO have achieved state-of-the-art performance. Image segmentation, which involves pixel-level classification, has also benefited from CNNs, with models like U-Net and Mask R-CNN being widely used. CNNs have found applications in other domains as well, including natural language processing, speech recognition, and bioinformatics.

While CNNs have achieved remarkable success, challenges still exist. Training deep CNNs requires large amounts of labeled data, which may not always be available. Addressing the computational requirements of training large-scale models and reducing the memory footprint are ongoing research areas. Improving the interpretability and explainability of CNNs remains a challenge, as the complex hierarchical representations learned by deep networks can be difficult to interpret. Advancements in transfer learning, domain adaptation, and model compression are expected to shape the future of CNNs.

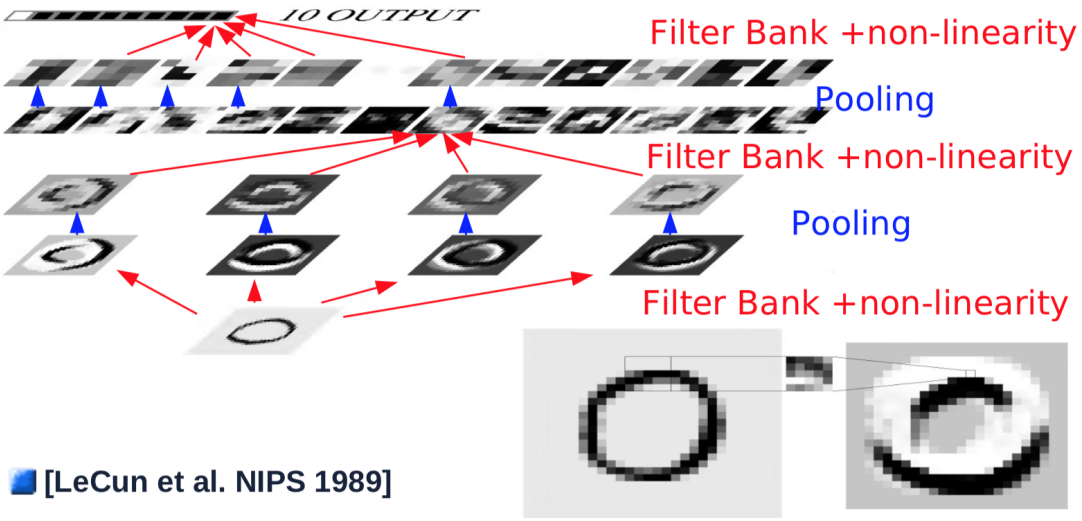


Figure 2.2: Scheme representing the general architecture of CNNs. Image taken from [14, 19]

2.3.3. Graph Neural Networks

Graph Neural Networks (GNNs) [40] have gained significant attention in recent years as a powerful framework for learning and reasoning on graph-structured data. With the rise of complex and interconnected datasets, such as social networks, citation networks, and molecular structures, the need to effectively model and analyze graph data has become increasingly crucial. Traditional ML approaches often struggle to handle the irregularity and varying sizes of graphs, making GNNs an appealing solution.

GNNs represent a class of NNs specifically designed to process graph-structured data. They leverage the connectivity and relationships between nodes in a graph to learn informative representations and perform tasks such as node classification, link prediction, and graph classification. GNNs build upon traditional neural networks, incorporating mechanisms that capture the local and global graph structure.

The architecture of a GNN consists of multiple layers, each performing message passing and aggregation operations to update node representations. The process typically involves the exchange of information between neighboring nodes, allowing nodes to capture the characteristics of their local graph neighborhoods. Various GNN architectures have been proposed, including Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), GraphSAGE, and Graph Isomorphism Networks (GINs). These architectures differ in terms of the aggregation functions, propagation rules, and attention mechanisms employed.

The general framework of Graph Neural Networks consists of a graph-in/graph-out architecture, meaning that this architectures accept a graph $G = (V, E)$, where $V = \{v_i\}_{i=1}^{N}$ and $E = \{e(v_i, v_j) : v_i, v_j \in V\}$ are the sets of nodes and the set of edges, as input, consisting of an adjacency matrix A together with the information loaded into nodes (e.g. nodes' features matrix X), edges and global-context, and returns a graph with the same connectivity structure of the input together with a progressively transformed embedding of the carried information. This behaviour can be observed in Figure 2.3. From this perspective a GNN can be defined as an optimizable transformation applied to one or multiple attributes of the graph (nodes, edges, global-context) preserving graph structure, so it is a permutation invariant transformation.

Training GNNs involves optimizing a specific objective function, typically through a combination of supervised, semi-supervised, or unsupervised learning techniques. Supervised learning on graphs requires labeled nodes or edges to learn node representations, while semi-supervised learning leverages a small portion of labeled data combined with unlabeled data. Unsupervised learning on graphs aims to capture latent graph structures without explicit supervision. GNNs utilize optimization algorithms and regularization

techniques to update the model parameters and prevent overfitting.
GNNs have demonstrated their efficacy across a wide range of domains. In social network analysis, GNNs can capture community structures, detect anomalies, and predict missing links. Recommendation systems benefit from GNNs by leveraging user-item interaction graphs to provide personalized recommendations. Knowledge graph completion tasks involve predicting missing relations between entities, and GNNs excel at capturing complex relational patterns. GNNs also find applications in bioinformatics and drug discovery, where they can analyze molecular graphs, predict molecular properties, and aid in drug design.
While GNNs have shown promise, several challenges remain. Scalability and efficiency are important concerns, as the computational complexity of GNNs grows with the size of the graph. Robustness and generalization are also crucial, as GNNs may struggle with handling noisy or incomplete graph data.
Figure 2.3 shows the idea behind GNNs. A graph is represented by $G = (V, E)$, the input of the network is $X \in \mathbb{R}^{N \times F}$, the array of graph features and $H \in \mathbb{R}^{N \times d_{out}}$ is the output array of the network.

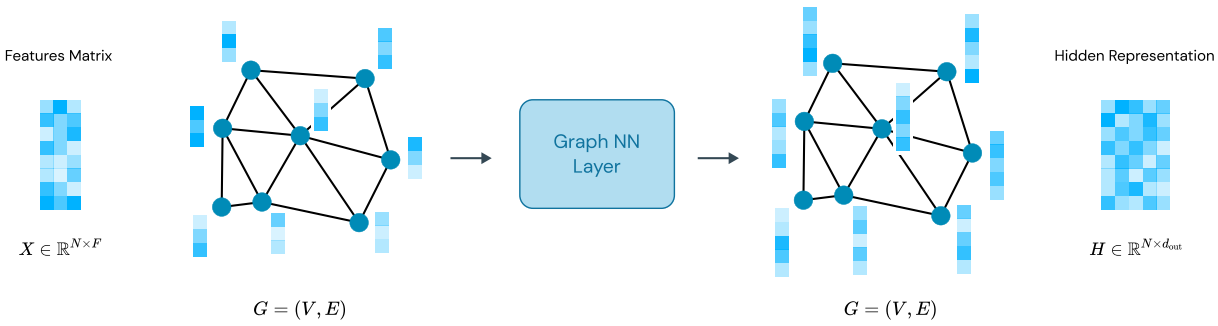


Figure 2.3: Scheme representing the idea behind GNNs. Extracted from [1]

2.4. Activation Function

In order to introduce non-linearities in a NN activation functions are used. In this thesis the main activation functions are introduced [16]:

1. Sigmoid: the sigmoid function is one of the special functions in the DL field, thanks to its simplification during back propagation. As it is possible to observe in Figure 2.4:
   - it ranges from [0,1];
   - it is not zero centered;

- it contains the exponential operation, therefore it is computationally expensive.

The main problem faced when using this function is because of saturated gradients, as the function ranges between 0 to 1, the values might remain constant and thus the gradients will have very small values. Resulting in almost no change applying gradient descent.

2. Hyperbolic Tangent (tanh): the hyperbolic tangent also has the following properties:
   - it ranges between [-1,1];
   - it is zero centered.

   This function can be considered as a good example in case when the input is greater than 0, so the gradients obtained will either be all positive or negative, which can lead to explosion or vanishing issue, thus usage of the hyperbolic tangent can be a good thing. However, it still faces the problem of saturated gradients.
3. Rectified Linear Unit Activation Function (ReLU): ReLU is the most commonly used activation function, because of its simplicity during backpropagation and since it is not computationally expensive. It has the following properties:
   - it doesn't saturate;
   - it converges faster than some other activation functions.

   Still, one could face an issue of dead ReLU, for instance, if: $\boldsymbol{w} > 0$, $\boldsymbol{x} < 0$. So, $\text{ReLU}(\boldsymbol{w} \cdot \boldsymbol{x}) = 0$, always.
4. Leaky ReLU: Leaky ReLU can be used as an improvement over ReLU. It has all properties of ReLU, plus it will never have a dead ReLU problem.
5. Exponential Linear Units (ELU): ELU is also a variation of ReLU, with a better value for $\boldsymbol{x} < 0$. It also has the same properties as ReLU along with:
   - no Dead ReLU Situation;
   - closer to zero mean outputs than Leaky ReLU;
   - more computations because of exponential function.
6. Maxout: Maxout has been introduced in 2013. It has the property of linearity in it. So, it never saturates or dies. But is expensive as it doubles the parameters.

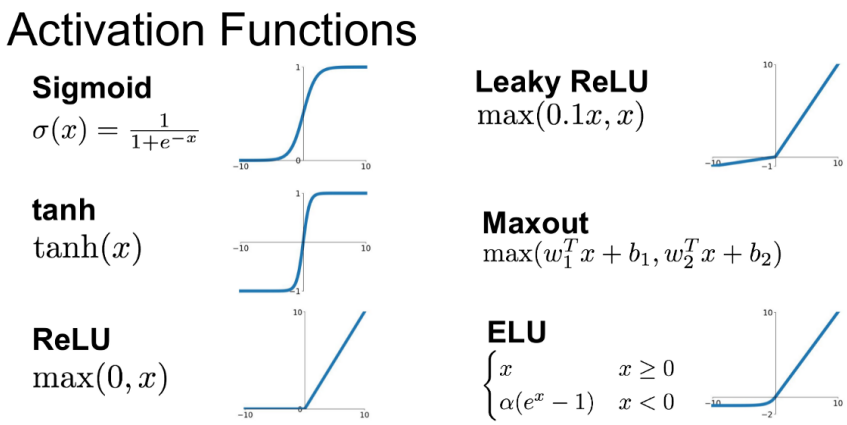


Figure 2.4: Different activation functions. Extracted from [16].

In this thesis ELU will be used as activation function since it has shown the best results.

2.5. The Minimization Problem

Finally, one crucial component in training NNs is the loss function. The evaluation of the loss function plays a crucial role in training and optimizing models. The loss function quantifies the discrepancy between the predicted outputs of the model, on a batch of n elements, $\{\hat{\boldsymbol{y}}^{(i)}\}_{i=1}^{n}$ and the actual target values $\{\boldsymbol{y}^{(i)}\}_{i=1}^{n}$. By measuring this discrepancy, the loss function provides a numerical representation of how well the model is performing. The choice of a specific loss function depends on the nature of the ML task. Different types of problems, such as classification, regression, or sequence generation, require different loss functions tailored to their respective objectives.
In regression problems, such as this case, where the aim is to predict continuous or numeric values, popular loss functions include:

- Mean Squared Error (MSE) Loss: MSE measures the average squared difference between the predicted and true values. It penalizes large errors more than smaller ones and is commonly used in tasks such as predicting housing prices or stock market trends.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)})^2 \,. \tag{2.2}$$

- Mean Absolute Error (MAE) Loss: MAE computes the average absolute difference between the predicted and true values. It provides a more robust measure of error

compared to MSE and is less sensitive to outliers.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)}| \,. \tag{2.3}$$

During training, the model's parameters are iteratively updated to minimize the chosen loss function. This process is typically achieved using optimization algorithms like gradient descent [30] which calculate the gradients of the loss function with respect to the model's parameters. By iteratively adjusting the parameters in the direction that minimizes the loss, the model gradually improves its predictions.
For instance, backpropagation is a fundamental algorithm used in training neural networks. It is a form of supervised learning that enables the network to adjust its parameters iteratively based on the errors in its predictions. The primary goal of backpropagation is to minimize the loss function, which measures the discrepancy between the predicted outputs and the true targets.
In this explanation a feedforward neural network with N_l layers will be considered. The input to the network is denoted as $\boldsymbol{x}$, and the output is denoted as $\hat{\boldsymbol{y}}$.
Each layer i of the network is associated with a set of trainable parameters, represented by a two-dimensional tensor $W_i = \{w_{ijk}\}_{j,k=1}^{n}$, denoted as weights, and $\boldsymbol{b_i} = \{b_{ij}\}_{j=1}^{n}$ the bias vector. The output of each layer is obtained through a series of transformations and activations. The output of layer i is denoted as $\boldsymbol{l_i}$ and $\boldsymbol{l_0} = \boldsymbol{x}$.
The forward pass in the neural network involves propagating the input $\boldsymbol{x}$ through each layer to obtain the final prediction $\hat{y}$. This process can be mathematically represented as:

$$\boldsymbol{l_i} = z(W_i\, \boldsymbol{l_{i-1}} + \boldsymbol{b_i}) \,, \tag{2.4}$$

where $z(\cdot)$ represents the activation function applied element-wise to the input.
During the training process, one computes the loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$, following the formula obtained by Equation (2.2) or Equation (2.3), which measures the error between the predicted output $\hat{\boldsymbol{y}}$ and the true target $\boldsymbol{y}$.
The key idea behind backpropagation is to compute the gradients of the loss function with respect to the network's parameters. This process starts from the output layer and moves backward through the layers, hence the name *backpropagation*.
The gradient of the loss function with respect to the parameters in layer i is denoted as ∇_{W_i} and $\nabla_{\boldsymbol{b_i}}$.
The gradients are computed using the chain rule of calculus. For the output layer (N_l), the gradients are computed as follows:

$$\nabla_{W_{N_l}} = \frac{\partial L}{\partial \boldsymbol{l_{N_l}}} \cdot \frac{\partial \boldsymbol{l_{N_l}}}{\partial W_{N_l}} \,, \tag{2.5}$$

$$\nabla_{\boldsymbol{b_{N_l}}} = \frac{\partial L}{\partial \boldsymbol{l_{N_l}}} \cdot \frac{\partial \boldsymbol{l_{N_l}}}{\partial \boldsymbol{b_{N_l}}} \,. \tag{2.6}$$

Then, for each layer i $(N_l - 1, N_l - 2, ..., 2)$, the gradients are recursively computed using the chain rule:

$$\nabla_{W_i} = \frac{\partial L}{\partial \boldsymbol{l_i}} \cdot \frac{\partial \boldsymbol{l_i}}{\partial W_i} \,, \tag{2.7}$$

$$\nabla_{\boldsymbol{b_i}} = \frac{\partial L}{\partial \boldsymbol{l_i}} \cdot \frac{\partial \boldsymbol{l_i}}{\partial \boldsymbol{b_i}} \,. \tag{2.8}$$

After computing the gradients for all layers, the network's parameters are updated using an optimization algorithm, such as stochastic gradient descent (SGD) or its variants like ADAM or Adagrad [6, 9, 18, 30]. The ADAM algorithm is a stochastic gradient descent method, meaning it replaces the actual gradient by an estimate calculated from a random subset of the data (batch). It combines the benefits of both adaptive learning rates and momentum, while Adagrad adapts the learning rates of each parameter individually based on their historical gradients. These adaptive methods enable faster convergence and better handling of different parameter scales. The gradients are used to update the weights and biases of each layer, aiming to minimize the loss function and improve the model's performance.

The evaluation of the loss function occurs after each batch. The loss value represents the current performance of the model on the training data. Lower loss values indicate better alignment between the predicted outputs and the true target values.

Pytorch is able to perform automatic differentiation, precisely, it uses a dynamic computational graph. It offers automatic differentiation through backpropagation.

3 | Development of the Prediction Algorithm

Contents

Chapter 3 presents an in-depth explanation of the prediction algorithm used in this thesis, tracing its origins from a seminal work and elucidating its core principles. The chapter delves into the intricacies of the algorithm, providing comprehensive insights into its inner workings. Furthermore, it highlights the modifications and improvements made in the new implementation, paving the way for enhanced performance and adaptability. By offering a detailed account of the algorithm's evolution and customizations, this chapter sets the stage for subsequent testing and evaluation, showcasing the algorithm's applicability in the context of the thesis objectives.

3.1. Presentation of the Original Code

The ML algorithm developed in this thesis was inspired by the work described in: 'Accelerating GMRES with Deep Learning in Real-Time' written by Kevin Luna, Katherine Klymko and Johannes P. Blaschke [20]. In the paper the implementation of a ML accelerated GMRES solver in Python, and using Pytorch, is defined. The solver is then used

to accelerate the Poisson and Advection-Diffusion equation and different architectures of NNs are tested. The GitHub GMRES-Learning [21] contains two demos:

- 2D Poisson Problem;
- 2D Advection Diffusion Problem.

Thanks to these demos the potentials of accelerating the GMRES algorithm with ML wrappers is shown more clearly.
Specifically, the code used in the demos worked on problems of the form: $A\boldsymbol{x^i} = \boldsymbol{b^i}$, where A is a linear operator defined as a stencil operator for a n-cell 2D grid, one of which is shown in Listing 3.2.
A sequence of linear problems of the form $A\boldsymbol{x^i} = \boldsymbol{b^i}$ are solved over the course of a simulation. The goal here is to train a NN $N(\boldsymbol{b})$ in real-time as linear problems are solved by GMRES. This is accomplished by having the NN provide an initial guess $N(\boldsymbol{b^i}) = \boldsymbol{x_0^i}$ to GMRES. The training objective is that this initial guess then improves the rate of convergence of GMRES.
The first training of the model is performed after a certain *initial set* of problems is solved, that is done in order to generate a first dataset to train on. After the first training the model is retrained with each addition to the dataset. The size of the initial set N^{train} is an input of the algorithm, as shown in Listing 3.1, and can be chosen by the user.
The dataset used to train the model consists of RHS-solution pairs $\{(\boldsymbol{b^i}, \boldsymbol{x^i})\}$. However, unlike traditional deep learning approaches, the goal here is to train the network in real-time while data is being generated from the simulation. This naturally leads to an online supervised learning problem since at a given time $t < T$, only a finite number of $(\boldsymbol{b^i}, \boldsymbol{x^i})$ pairs are available. In order to ensure a high-quality dataset some time steps are discarded, while only 'high-quality' ones are kept.
This operation is performed by computing two quality metrics once a system is resolved. The first is the time to reach convergence, or to obtain the solution, starting from $\boldsymbol{x_0^i}$, $TOS(\boldsymbol{x_0^i})$, while the second metric is the residual at the end of the first restart, $E_\kappa(\boldsymbol{x_0^i})$. These two values are then compared with the averages of the previous p iterations $(M_p(TOS(\boldsymbol{x_0})),\ M_p(E_\kappa(\boldsymbol{x_0})))$. Now, if the new values are worse than the averages, specifically if both are greater, the system is saved into the dataset, as this indicates that the model is not apt to provide good predictions for this type of data.
Figure 3.1 and Figure 3.2 show the speed-up for each linear system obtained by the ML routine. For the i-th system the speed-up is computed as the ratio between the time to reach convergence of the classic GMRES algorithm and $TOS(\boldsymbol{x_0^i})$.

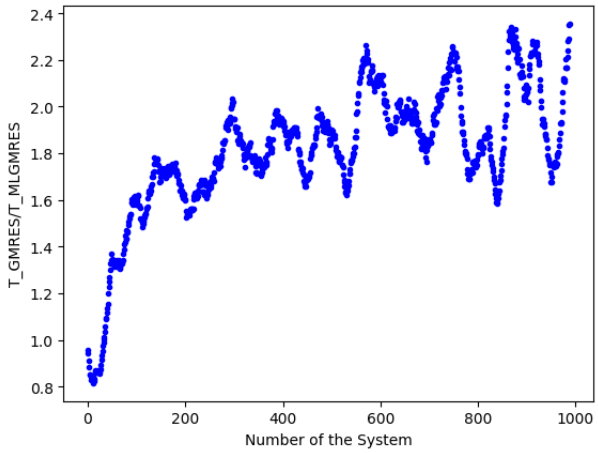


Figure 3.1: Iteration speed-up obtained in the `Demo` found at [21]. The problem considered is the Laplace equation for a grid of 20×20, with an initial set of 32 problems, the model is applied on a set of 1000 problems and is trained using CNNs.

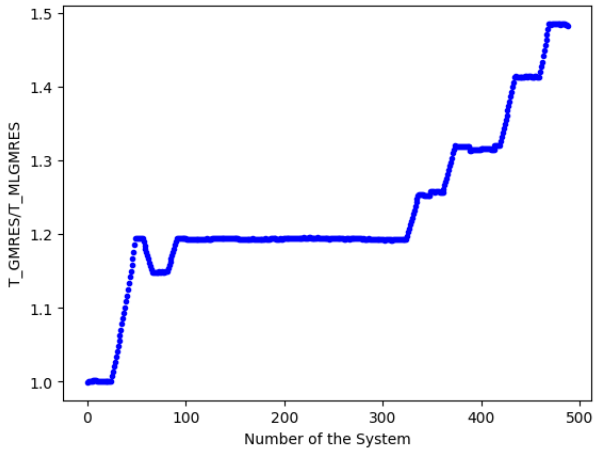


Figure 3.2: Iteration speed-up obtained in the `AdvectionDiffusion_Demo` found at [21]. The problem considered is the Advection-Diffusion equation for a grid of 20×20, with an initial set of 32 problems, the model is applied on a set of 500 problems and is trained using CNNs.

Following [20], the aim of this thesis is to accelerate the GMRES method by feeding it a better initial guess, predicted using ML. In the following, the algorithm is explained in detail, with all the modifications performed on the original code. In Chapter 4 the original simple test cases of [20] are taken into consideration in the new framework, studying also lager dimensions, exploring different NN architectures and also comparing the ML efficiency with respect to previously used methods. Furthermore, Chapter 5 tackles the application of the implemented algorithm on compressible CFD problems.

3.2. The Prediction Algorithm

First, the RHS vector is generated, either randomly for non-time-dependent problems or from the solution of the previous iteration. If the network is already trained, and therefore the index of the system $i > N^{train}$, the NN is evaluated using as input $\boldsymbol{b^i}$ to obtain as output a new initial guess $\boldsymbol{x_0^i} = N(\boldsymbol{b^i})$. Then new data is then fed into the GMRES code, with output $\boldsymbol{x_m^i}$, the approximated solution of the system. When the new solution is computed the metrics $(TOS(\boldsymbol{x_0^i}), E_\kappa(\boldsymbol{x_0^i}))$, defined in Section 3.1, are calculated and compared to the averages $(M_p(TOS(\boldsymbol{x_0})), M_p(E_\kappa(\boldsymbol{x_0})))$. In the case that the new metrics are below average, then the next system in the sequence is considered and therefore the process starts again by generating the new RHS b^{i+1}.
When, instead, the metrics are above average the system is added to the new batch of data and if the size of the new batch is equal to the retrain frequency f_r the data is check for orthogonality and linear independence, before being added into the training dataset and retraining the model. After these operations are performed the new RHS is generated and the process restarts once again, until the last system is solved.
The algorithm by [20] is shown in Figure 3.3 for the sake of completeness. Figure 3.4 represents one iteration of the prediction algorithm in a more compact manner.

There are two main versions of the algorithm. The first one is used for non-time-dependent problems and therefore the algorithm iterates on different right hand sides, see Section 4.1. The second version of the algorithm iterates on time and therefore it is used for time-dependent problems, see Section 4.2.
Both versions are detailed in Algorithm 3.1. It is important to remember that $1 \leq \kappa \leq m$ represents the number of iterations to end the first restart of the GMRES algorithm. Moreover, $d(\cdot, \cdot)$ defines a distance operation used in order to check the orthonormality of the different $\boldsymbol{b}$ vectors. This version of the algorithm is applied in the numerical simulations reported in Section 4.1.1.

3.3. Loss Function Evaluation

In the implemented model's training process, the loss function plays a crucial role in guiding the network to make accurate predictions. The loss function measures the discrepancy between the predicted initial guess, denoted as $\boldsymbol{x_0}$, and the actual solution computed by the GMRES algorithm, denoted as $\boldsymbol{x}$.
The loss function is defined as the Mean Squared Error (MSE) between the predicted initial guess and the true solution:

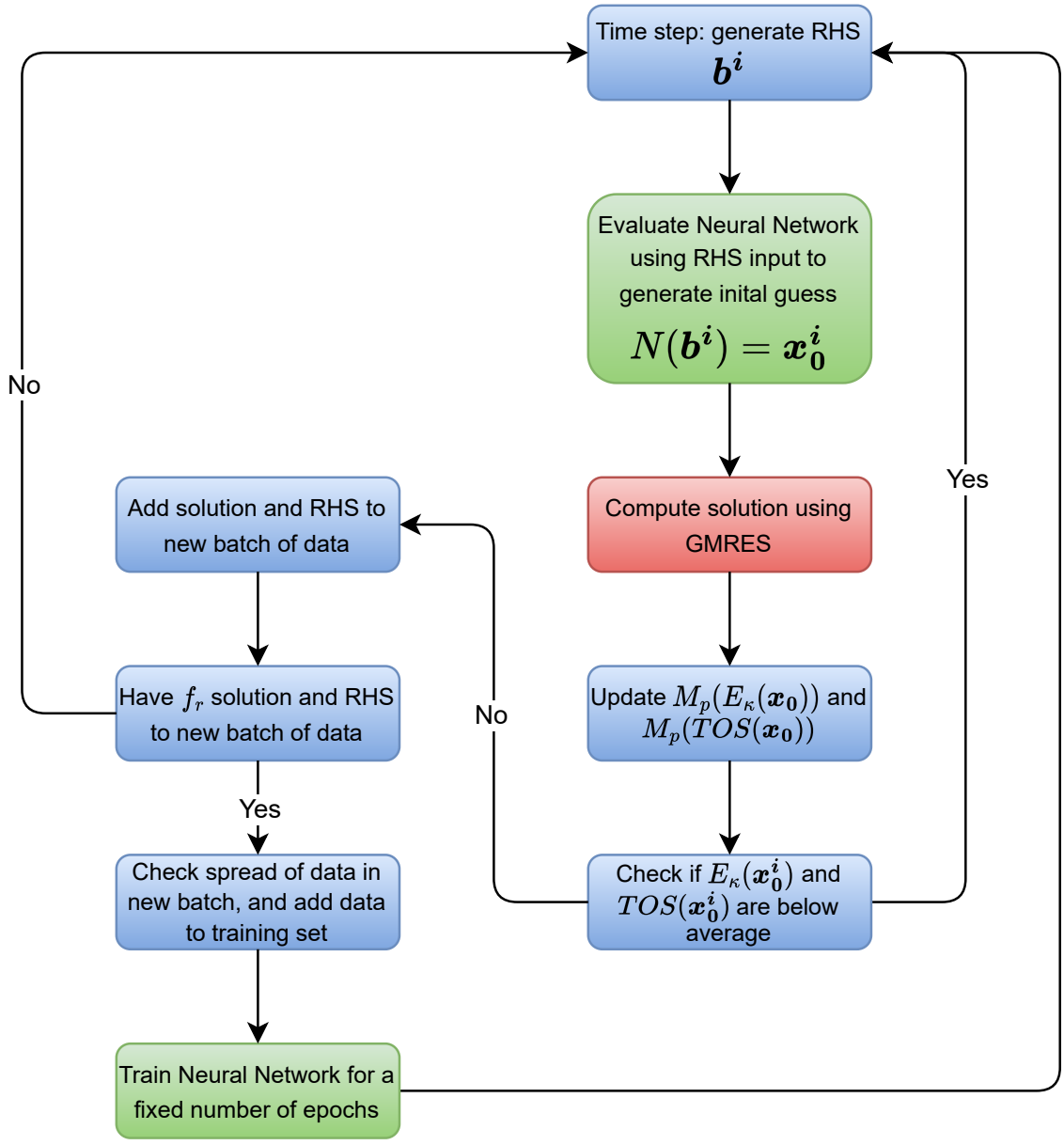


Figure 3.3: Scheme of the whole algorithm. In red the sections ran with CPU and in green the ones ran with GPU. The scheme was taken from [20].

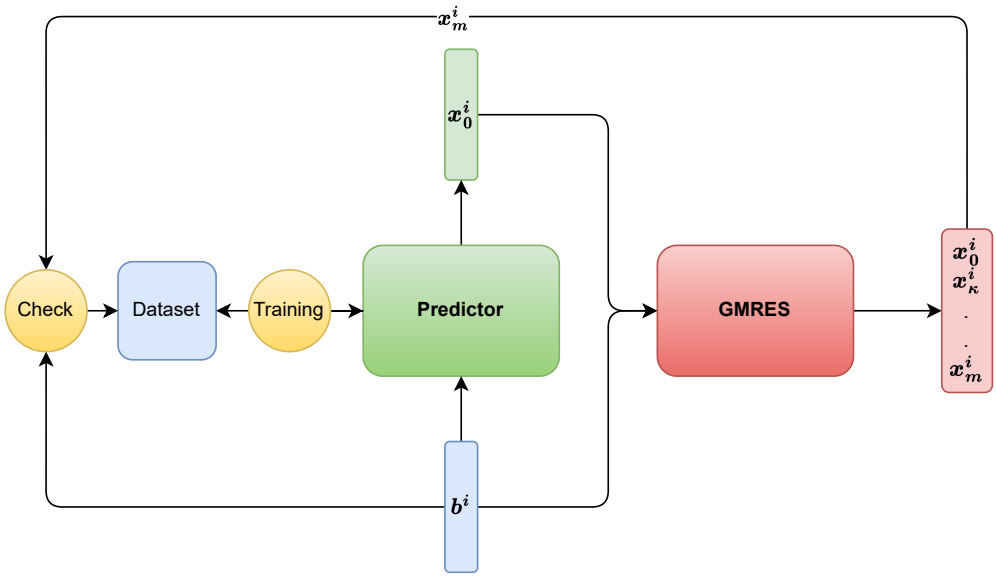


Figure 3.4: ML workflow of iteration i of the algorithm.

Algorithm 3.1 MLGMRES after the model has been trained

Simulation has reached iteration k (or time $t = t_{k-1}$)
Neural Network $N(\boldsymbol{b})$ has been trained using training set X_t
Set of candidate data pairs C_t is empty, the size of the new batch of data is $N_b = 0$
while $N_b < f_r$ **do**
  Compute RHS $\boldsymbol{b^k}$ as a random distribution on $[-1, 1]$ (or $\boldsymbol{b^k} = B(\boldsymbol{x_m^{k-1}})$)
  Use NN $\boldsymbol{x_0^k} = N(\boldsymbol{b^k})$
  Compute solution $\boldsymbol{x_m^k}$ = GMRES($\boldsymbol{x_0^k}$, $\boldsymbol{b^k}$)
  Compute performance metrics $E_\kappa(\boldsymbol{x_0^k})$ and $TOS(\boldsymbol{x_0^k})$
  Compute averages $M_p(\{E_\kappa(\boldsymbol{x_0})\})$ and $M_p(\{TOS(\boldsymbol{x_0})\})$
  if $E_\kappa(\boldsymbol{x_0^k}) < M_p(\{E_\kappa(\boldsymbol{x_0})\})$ and $TOS(\boldsymbol{x_0^k}) < M_p(\{TOS(\boldsymbol{x_0})\})$ **then**
    Proceed with the simulation. Set $k = k + 1$
  else
    Compute $d(\boldsymbol{b^k}, \boldsymbol{b})\ \forall\, \boldsymbol{b} \in C_t$
    if $d(\boldsymbol{b^k}, \boldsymbol{b})\ \forall\, \boldsymbol{b} \in C_t$ is acceptable **then**
      Add $(\boldsymbol{b^k}, \boldsymbol{x_m^k})$ to C_t
      Set $N_b = N_b + 1$
    end if
    Set $k = k + 1$ (and $t = t_{k-1} + \Delta t = t_k$)
  end if
end while

$$L(\boldsymbol{x_0}, \boldsymbol{x}) = \|\boldsymbol{x_0} - \boldsymbol{x}\|^2 = \|N(\boldsymbol{b}) - \boldsymbol{x}\|^2 \,, \tag{3.1}$$

where $\|\cdot\|$ denotes the Euclidean norm, also known as the L^2 norm, which measures the squared distance between $\boldsymbol{x_0}$ and $\boldsymbol{x}$. Minimizing this loss function encourages the network to provide initial guesses that are closer to the numeric solutions.
During each iteration of training, the model's trainable parameters are updated based on the gradients of the loss function with respect to these parameters. This process of computing the gradients and adjusting the parameters iteratively refines the network's ability to predict more accurate initial guesses.
In the implementation, an adaptive gradient descent optimization algorithm is used to efficiently update the parameters. In the implemented code the ADAM algorithm is applied.
By defining the loss function and employing these optimization techniques, our model learns to improve its predictions iteratively, achieving more accurate initial guesses ($\boldsymbol{x_0}$) for the GMRES algorithm, which leads to improved overall performance and convergence speed for the linear system solver.

3.4. Algorithmic Refinements and Code Modifications

In the given context, the goal was to integrate ONERA's version of the GMRES code into the existing codebase. The integration was accomplished by modifying the *Python decorator*, responsible for invoking the GMRES algorithm within the algorithm. In the programming context, a decorator is a design pattern that allows you to modify the functionality of a function by wrapping it in another function. The outer one is called the decorator, which takes the original function as an argument and returns a modified version of it. Python decorators are used to enhance the functionality of functions or methods in a non-intrusive manner. The original code by K. Luna et al. [20] employed such a decorator to facilitate the use of GMRES in certain computations.
The custom GMRES algorithm was designed to retain the core purpose and functionality of the original version. However, its internal implementation was better suited to fulfil the requirements of ONERA, specifically in order to tackle linear systems applied to vectors. By adjusting the Python decorator, it was ensured that all invocations of the GMRES algorithm within the algorithm were routed to the new implementation instead of the original one. Through this approach, the integrity of the existing code was maintained, still many modifications had to be performed.

```
1 InputDim  = dim
```

```
OutputDim = dim
# Number of samples to collect before using prediction from Neural
    Network:
Initial_set = 32

nn_predict = CNNPredictorOnline(InputDim, OutputDim, DenseNN_blocks)
trainer    = PredictorTrainer(nn_predict, Initial_set = Initial_set)

@timer
@cnn_predictorOnline_timed(trainer)
def MLGMRES(A,b,x0,n,m,e,max_restart,M,flagM,method_precond_right):
    return GMRES_ML(A,b,x0,n,m,e,max_restart,M,flagM,
    method_precond_right)
```

Listing 3.1: Definition of the MLGMRES function with the decorator defined in Appendix C Listing C.3.

3.4.1. Use of matrices and of Vectors

The original code used linear operators, defined as a python `function`, for instance, for the Laplace equation, it was defined as shown in Listing 3.2. These operators were then applied to $n \times n$ matrices and produced a RHS of size $n \times n$. This thesis, instead, applies the GMRES method to systems composed of matrices and vectors.

```
def appl_2d(op, x, Nx, Ny):
    Ax = np.zeros((Nx, Ny))
    for ix, iy in np.ndindex(Ax.shape):
        Ax[ix, iy] = op(x, ix+1, iy+1)
    return Ax

def mk_laplace_2d(Nx, Ny, bc="dirichlet", xlo=0, xhi=0, ylo=0, yhi=0):
    '''
    mk_laplace_2d(N, bc="dirichlet", xlo=0, xhi=0, ylo=0, yhi=0)

    Generates laplace operator as a stencil operation for a N-cell 2D
    grid, for
    given boundary conditionds.
    '''
    def build_bc_1(x_in):
        x_out = np.zeros((Nx + 2, Ny + 2))
        x_out[1:Nx+1, 1:Ny+1] = x_in[:, :]
        x_out[0,    :] = xlo
        x_out[Nx+1, :] = xhi
        x_out[:,    0] = ylo
```

```
20          x_out[:, Ny+1] = yhi
21          return x_out
22
23      def build_bc_2(x_in):
24          x_out = np.zeros((Nx + 2, Ny + 2))
25          x_out[1:Nx+1, 1:Ny+1] = x_in[:, :]
26          x_out[0,    1:Nx+1] = x_in[-1, :]
27          x_out[Nx+1, 1:Nx+1] = x_in[0,  :]
28          x_out[1:Nx+1,    0] = x_in[:, -1]
29          x_out[1:Nx+1, Ny+1] = x_in[:,  0]
30          return x_out
31
32      def laplace_2d(x, i, j):
33          return (-4*x[i, j] + x[i-1, j] + x[i+1, j] + x[i, j-1] + x[i, j
    +1])
34
35      if bc == "dirichlet":
36          op  = lambda x, i, j: laplace_2d(build_bc_1(x), i, j)
37      elif bc == "periodic":
38          op  = lambda x, i, j: laplace_2d(build_bc_2(x), i, j)
39      else:
40          raise RuntimeError(f"bc={bc} not implemented")
41
42      return lambda x: appl_2d(op, x, Nx, Ny)
```

Listing 3.2: Example of Linear Operator used in [20]. Extracted from `src_dir.linop.py` at [21].

Therefore, one preliminary modification to the code was to adapt the decorator to use sparse matrices, since when dealing with large systems of equations it is much more effective to store the data in sparse format. This was done by changing the $(\cdot)$ operator of a function with the `@ Python3` operator. Moreover, as detailed in Section 3.4.2 all the Neural Networks developed by [20] were to be modified in order to work with 1D vector data and not 2D.

3.4.2. New Neural Network Architectures

Since the Neural Networks defined in [20] were defined to work on different class of data and for problems of small dimension overall, it was pivotal to develop new models.
Three different architectures were implemented and tested:

1. Dense Neural Network based Model;

2. Convolutional Neural Network based Model;
3. Mixed model: Combination of a Convolutional NN to process the input vector $\boldsymbol{b}$ and a Graph NN to process the input matrix A.

The code used to define the new version of NNs is found in Appendix C.

Dense Neural Network based Model

The proposed DNN architecture comprises the following components and operations:

- Normalization: The features of the input data are normalized with respect to the norm. This normalization step ensures that the features are appropriately scaled for effective learning;
- Linear Layers and Activation: Following the normalization step, the network applies a series of linear layers to increase the size of the input vector. Each linear layer performs a linear transformation on the input, gradually expanding its dimensions. After each linear layer, an activation function is applied to introduce non-linearity into the network;
- Iterative Linear Layers and Activation: The number of iterations for the linear layers is determined based on the size of the initial vector. The network repeats the linear layer and activation function operations iteratively, increasing the size and complexity of the input vector. This iterative process allows the network to learn and capture higher-order representations from the data;
- Final Linear Layer: After the iterative linear layers and activation functions, the network performs one last linear layer to reshape the output vector to its original size.

This structure is detailed in Figure 3.5. The iterative application of these layers and activations enables the network to learn and represent the data in a progressively more expressive manner. The final linear layer reshapes the output vector to match the original size.

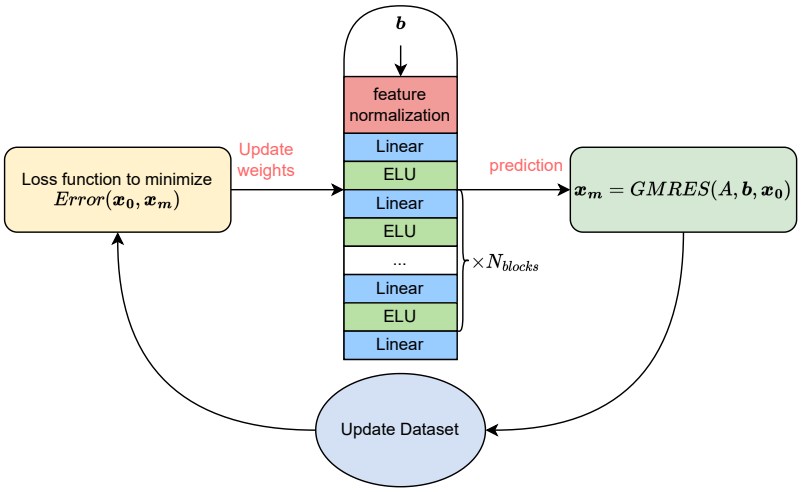


Figure 3.5: Structure of the Dense Neural Network used.

Convolutional Neural Network based Model

The proposed CNN architecture comprises the following components and operations:

- Normalization: The features of the input data are normalized with respect to the norm. This normalization procedure ensures that the features are appropriately scaled for effective learning by the network;
- Convolutional Layers, Activation, and Average Pooling: Each iteration of the network begins with a convolutional layer applied to the input data, followed by an activation function. The convolutional layer performs convolutions on the input, extracting pertinent features. The activation function introduces non-linearity, enabling the network to learn intricate patterns and relationships within the data. Subsequently, average pooling is applied, which reduces the spatial dimensions of the feature maps while retaining their depth;
- Iterative Convolution and Activation: Following each average pooling operation, a series of convolutional layers and activation functions are applied iteratively. The number of iterations for each vector is contingent upon the size of the initial vector. This iterative process facilitates the successive extraction and refinement of features from each vector, incorporating additional context and detail into the network's representation;
- Upsampling and Concatenation: Once the iterative convolution and activation steps are completed, the smaller-sized feature maps are upsampled to match the size of the original feature map (pre-pooling). This step ensures uniform spatial dimensions across all vectors for subsequent operations. Following upsampling, the feature maps

are concatenated together, through a summation. This concatenation procedure enables the network to amalgamate information from various levels of abstraction, as each feature map captures distinct spatial scales and patterns;

- Final Convolution Layer: Subsequently, a final convolutional layer is applied to the combined feature map. This layer further processes the aggregated information and produces the ultimate output of the network.

This structure is detailed in Figure 3.6. By employing iterative convolution and activation steps, the network enhances its capacity to extract intricate features from the data. The incorporation of normalization, pooling, upsampling, and concatenation facilitates the integration of information from multiple scales and levels of abstraction. Ultimately, the final convolutional layer refines the extracted features and generates the network's final output.

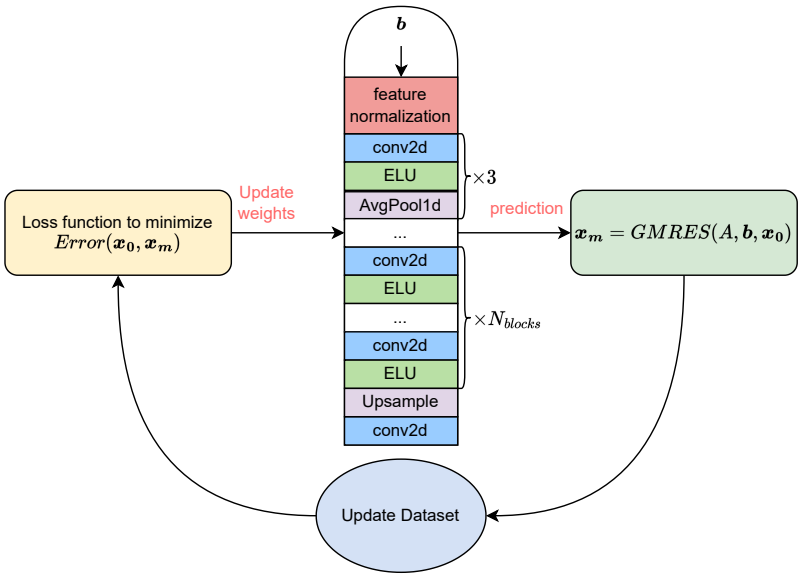


Figure 3.6: Structure of the Convolutional Neural Network used.

Mixed Model

This architecture was inspired by and adapted from [36].

The final network architecture is a hybrid model that combines a CNN and a GNN. The structure is as follows:

1. CNN Component:

- Input: The input data $\boldsymbol{b}$ is fed into the CNN component, following the same architecture described in Figure 3.6.
- Normalization: The features of the input matrix are normalized, respecting the norm.
- Convolution and Activation: The normalized features undergo a series of convolutional layers and activation functions. These operations increase the feature size.
- Pooling: After each convolutional layer, average pooling is applied to reduce the feature map's spatial dimensions while preserving its depth.
- Iterative Convolution and Activation: The convolutional layers and activation functions are repeated for the same number of times as the convolutional operations after pooling. This iterative process enhances the network's ability to extract and refine features.
- Upsampling: The smaller-sized feature maps obtained through pooling are upsampled to match the original size, ensuring consistency in the spatial dimensions.
- Final Convolution Layer: The upsampled feature maps are combined and processed by one last convolutional layer.

2. GNN Component:

- Input: The input matrix A is transformed into a graph representation, where each node has a certain number of features, and the edges capture the connectivity of the matrix.
- Feature Normalization: The features of the graph are normalized, similar to the CNN component.
- Graph Convolutional Layers and Activation: The normalized features undergo a series of graph convolutional layers, specifically designed for processing graph-structured data. After each graph convolutional layer, an activation function is applied to introduce non-linearity.
- Iterative Graph Convolution and Activation: The graph convolutional layers and activation functions are iterated for the same number of times as the convolutional operations after pooling in the CNN component. This iterative process enables the GNN to extract and refine features from the graph data.

- Final Graph Convolution Layer: The features from the last graph convolutional layer are reshaped to a vector of size 1.

Once both branches of such architecture have finished, there is a combination operation: The output vectors from the CNN and GNN components are summed element-wise. The resulting vector represents the combined features from both branches of the network.
To finish with, here are the post-combination operations that have been implemented:

1. Convolution and Activation: The combined vector undergoes a convolutional layer and an activation function.
2. Average Pooling: Following the convolution and activation, average pooling is applied to reduce the vector's size while preserving important features.
3. Convolution and Activation: Another convolutional layer and activation function are applied to further process the pooled vector.
4. Upsampling: The pooled vector is upsampled to match the original size.
5. Final Convolution Layer: The upsampled vector is passed through one last convolutional layer.

This structure is detailed in Figure 3.7. The final network architecture, combining both the CNN and GNN components, leverages the strengths of both approaches. The CNN processes the input matrix through convolutions and pooling, capturing spatial information, while the GNN operates on the graph representation, capturing relational information between the matrix' features. By combining the outputs of both branches, the network can leverage the complementary strengths of both components to produce the final output.

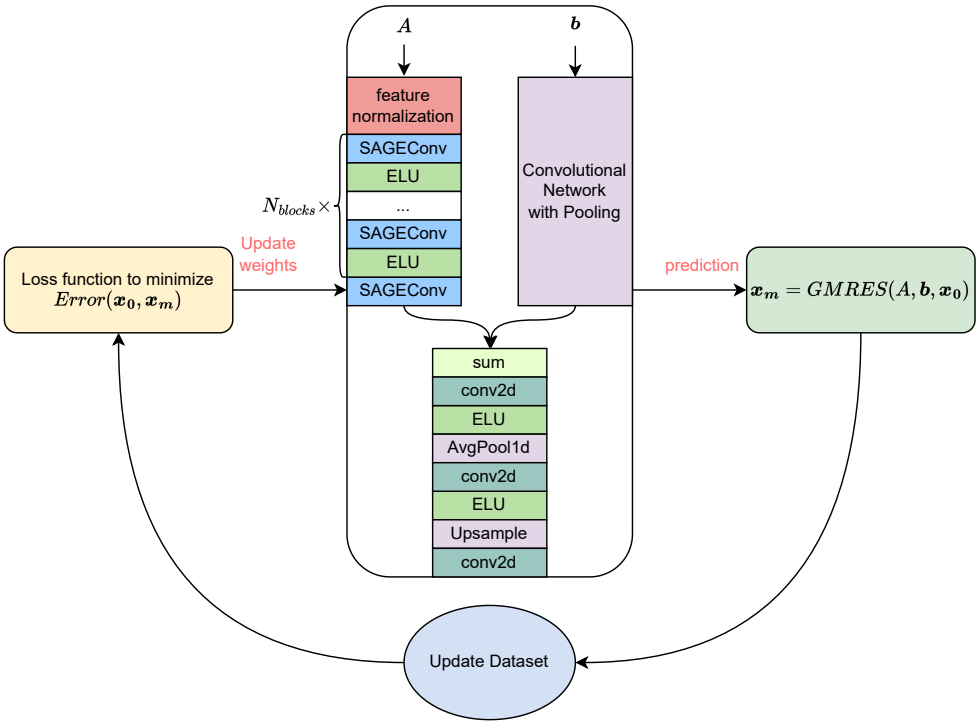


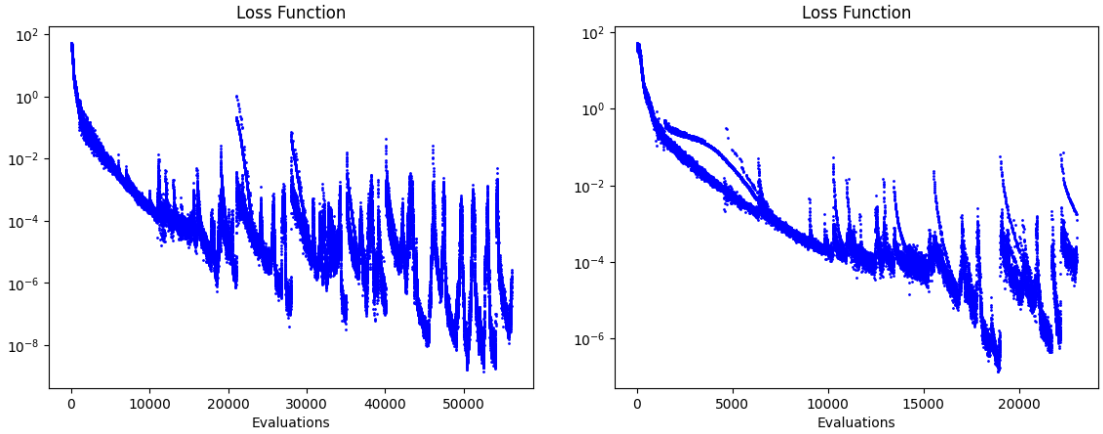
Figure 3.7: Structure of the Graph Neural Network used.

3.4.3. Generalization of the dataset expansion and retraining procedure

In the original code the model was retrained with every addition to the training dataset, therefore the retrain frequency f_r was fixed to 1. In order to decrease the computational cost of the simulations and to avoid overfitting the code was changed to accept different values for f_r. Specifically, the retrain frequency was defined such that it depended on the size of the training dataset. The idea behind this definition is that, whenever the simulation changes a lot with time, the model must retrain frequently to adapt the trainable parameters, however this would result in a large computational time dedicated to the training, therefore making different non-ML strategies more efficient. Moreover, retraining at each addition to the dataset could possibly result with overfitting on certain problems. To avoid these issues, f_r was fixed so that the larger the size of the dataset becomes, the less frequent the model would be retrained. This procedure showed great savings in real time, since, for a sequence of 1000 systems at the end of the simulation more than 200 systems could have been added to the dataset. Therefore, had the code remained the same, this would have resulted in hundreds of retrainings.

3.4.4. Early Stopping

Early stopping is a powerful technique employed in ML to optimize model performance. Its purpose is to prevent overfitting and enhance the generalization capabilities of a model. Overfitting occurs when a model becomes too complex and starts memorizing the training data instead of learning the underlying patterns. As a result, the model's performance on unseen data tends to suffer. Early stopping mitigates this problem by monitoring the model's performance during training and terminating the process at an optimal point.
During the training phase, a ML model iteratively adjusts its parameters to minimize a predefined loss function. This process aims to find the optimal configuration that best fits the training data. However, if the model is allowed to continue training for too long, it may start to overfit. Overfitting can be detected by observing the model's performance on a separate validation set, which consists of data not used for training.
Early stopping operates by stopping the training process when the performance starts to deteriorate consistently, indicating that the model's ability to generalize has reached its peak. The rationale behind early stopping is rooted in the bias-variance trade-off. The bias represents the error introduced by approximating a real-world problem with a simplified model, while the variance captures the model's sensitivity to fluctuations in the training data. Initially, as the model learns, both the bias and variance decrease. As training progresses, the bias continues to decrease while the variance begins to rise. At some point, the increasing variance leads to overfitting, causing the model's performance on unseen data to decline.
By stopping the training process early, the model's capacity to fit noise and irrelevant patterns in the training data is limited. Early stopping effectively strikes a balance between bias and variance, resulting in a model that generalizes well to unseen data. Moreover, early stopping helps reduce computational resources and training time, as the model is not allowed to converge fully before termination, this is especially important for the case of online learning, when the training operation could be performed a large number of times.



(a) No Early Stopping (b) Early Stopping

Figure 3.8: Comparison of the behaviour of the loss function with and without early stopping.

Hence, in the context presented, early stopping enables an approximate 60% reduction in evaluations. The code describing the implementation of early stopping is found in Listing C.1.

3.4.5. Gradient Clipping

In the field of ML, exploding gradients [2] is a term used to indicate when the gradients reach too large values during the training, making the model unstable. Similarly, vanishing gradients refer to the case of gradients getting too small values. These problems prevent the trainable weights of the network from changing values in a coherent way and therefore they result causing the model to be unable to learn from the training data.
If these issues arise, one possible solution is the use of gradient clipping. Gradient clipping is a technique that tackles exploding gradients. The idea of gradient clipping is very simple: If the gradient gets too large, it gets rescaled to keep it small. More precisely, if $\|\boldsymbol{g}\| \geq c$, then $\boldsymbol{g} \leftarrow c\frac{\boldsymbol{g}}{\|\boldsymbol{g}\|}$.

4 | Numerical Experiments on Simple Problems

Contents

Chapter 4 provides first validations for the proposed algorithm on simple test cases. The chapter begins with the investigation of the Laplacian equation with a series of random right hand sides, exploring the algorithm's efficacy in accelerating this classical mathematical problem. Subsequently, the algorithm's performance is evaluated in the context of the advection diffusion equation, providing insights into its ability to handle time-dependent

problems. Through comprehensive analysis and comparisons with established methods, this chapter demonstrates the algorithm's capabilities and adaptability, further validating its suitability for tackling diverse classic problems within the scope of the thesis.

4.1. Laplace Equation

First, the real-time deep-learning methodology is applied to the 1D Poisson problem. In particular, a sequence of problems is solved:

$$-\frac{\partial^2 u}{\partial x^2} = f^k \tag{4.1}$$

$$u|_{\partial\Omega} = 0 \tag{4.2}$$

where $\Omega = [-1, 1]$ and f^k are randomly generated for every index.
The discretized system is written as:

$$-\frac{U_{i+1} - 2U_i + U_{i-1}}{\Delta x^2} = f^k(x_i) \quad \forall\, i = 1, ..., n-1, \tag{4.3}$$

with

$$U_0 = 0, \quad U_n = 0 \quad \forall\, k = 1, ..., N_S. \tag{4.4}$$

Finally, the system is studied as:

$$A\boldsymbol{x}^{\boldsymbol{k}} = \boldsymbol{b}^{\boldsymbol{k}}\ , \tag{4.5}$$

where A is the matrix obtain with the discretization of the equation, $\boldsymbol{x}^{\boldsymbol{k}}$ is the n-dimensional vector of unknowns and $\boldsymbol{b}^{\boldsymbol{k}}$ is the n-dimensional vector such that $b_i^k = f^k(x_i)$. In order to recreate the same problem presented in [21], the RHS vector is taken as as a random dipole distribution on $[-1, 1]$, as detailed in Listing 4.1.

```
# Definition of a random RHS fro the Poisson problem
xloc      = np.random.uniform(xx[0], xx[-1])
xlocShift = np.random.uniform(-0.25, 0.25)
AmplitudeFactor  = np.random.uniform(1,10)
AmplitudeFactor2 = AmplitudeFactor*np.random.uniform(1,2)
sigma = 0.07*np.random.uniform(0.9,1.1)
```

```
b      = AmplitudeFactor*Gauss_pdf(xx,xloc,sigma) \
       + AmplitudeFactor2*Gauss_pdf(xx,xloc+xlocShift,sigma)
Field = np.random.normal(loc=0.0, scale=1.0, size=(dim,))
Field = AmplitudeFactor*np.random.normal(loc=0.0, scale=1.0, size=(dim,)
    )
b = b + Field

b = b * dx ** 2 # Finite difference grid spacing
```

Listing 4.1: Definition of the random RHS for the Laplacian testcase.

4.1.1. Results

The algorithm was tested on a set of 1000 systems each of size 20×20, fixing the residual tolerance to 10^{-10} and the Krylov space, $m = 4$, also in order to compare the results with the ones obtained by [20]. The simulation is run using the DNN model defined in Section 3.4.2, with 800 trainable parameters and with the initial set size fixed to 32 (*i.e.*, the training of the network, will wait until at least 32 systems are in the database to begin the online training procedure). At the end of the simulation the size of the dataset is equal to 234 (*i.e.*, 202 sets of $\boldsymbol{b}$ and $\boldsymbol{x}$, where added to the database since the system they represented was above the average with respect to the time to solution or the error, *i.e.*, see Section 3.2). It is possible to plot the residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) with respect to the number of iterations. Figure 4.1 shows the comparison between the values obtained by the classic GMRES method with the zeros vector as initial guess and the ML enhanced version.

Appendix A explains in detail the figures and how they were obtained.

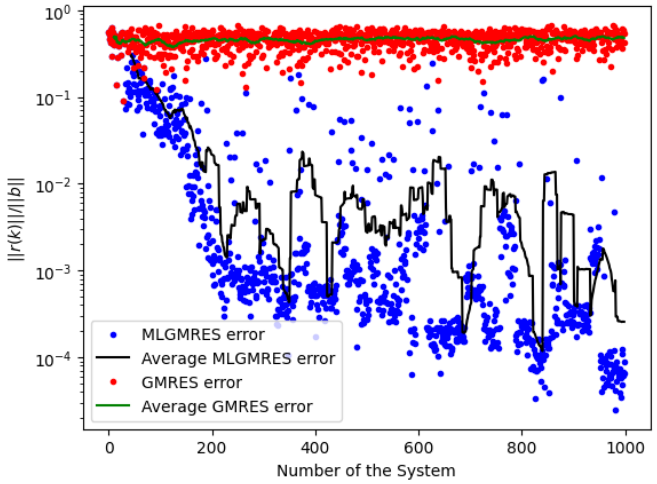


Figure 4.1: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) w.r.t. the number of systems (according to the x-axis label) of the Laplace equation, comparing GMRES to MLGMRES. Obtained by using $n = 20$, $N_S = 1000$, using an initial set of dimension 32 and DenseNN.

From Figure 4.1 one can see that the initial guesses predicted by the model are already closer at the first restart than for the classic algorithm. Moreover, it becomes evident that as the number of systems increases, there is a discernible learning process occurring, as evidenced by the decreasing residual, depicted by the black line in the figure.
Furthermore, one can take the last system of the simulation and see how the standard and ML-powered GMRES behave with respect to the norm of the normalized residual $\|\boldsymbol{r}\|/\|\boldsymbol{b}\| = \|\boldsymbol{b} - A\boldsymbol{x}\|/\|\boldsymbol{b}\|$, as shown in Figure 4.2. From the figure it is possible to observe that already the norm initial residual $\|\boldsymbol{r_0^{1000}}\| = \|\boldsymbol{b} - A\boldsymbol{x_0^{1000}}\|$ is much smaller than the initial residual with the classic choice of taking $\boldsymbol{x_0} = [0, ..., 0]^T$.

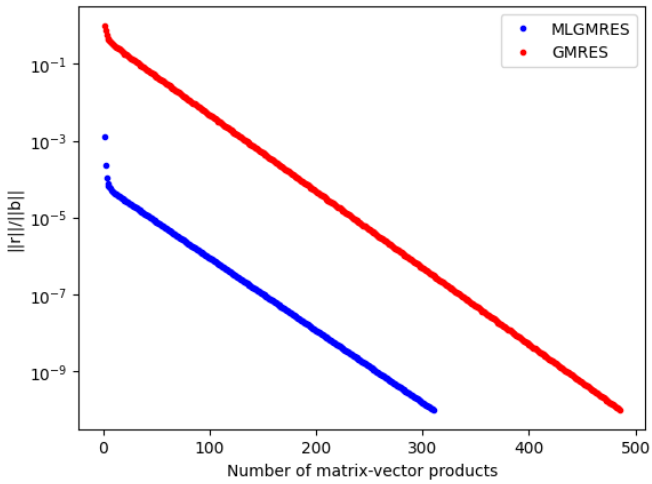


Figure 4.2: Behaviour of the norm of the normalized residual for the last system of the Laplace sequence w.r.t. the number of matrix-vector products.

When all the N_S are completed, it is also possible to observe the number of matrix-vector products needed to reach convergence for each single system solved by the GMRES method. Figure 4.3 displays the behaviour of the ML enhanced version of the GMRES code, confirming that indeed a learning is taking place, indeed the number of matrix-vector products decreases as the simulation goes on, after the first training is performed.

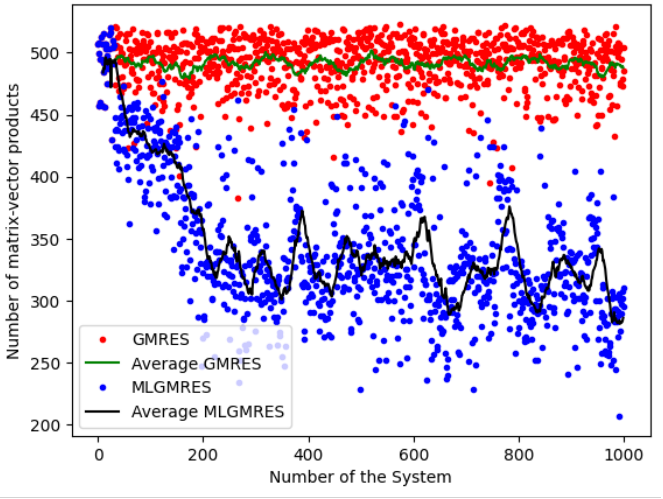


Figure 4.3: Number of matrix-vector products to reach convergence of the Laplace equation w.r.t. the number of systems.

Finally, one can observe the plot of the iterations' speed-up, as shown by Figure 4.4. The speed-up is computed as the ratio between the time taken to resolve the system by

the classic GMRES solver (T_GMRES) and the one taken by the MLGMRES solver (T_MLGMRES).

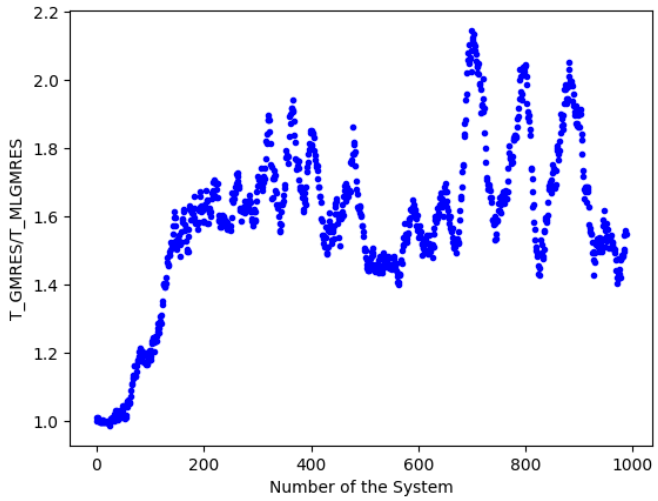


Figure 4.4: Iteration speed-up of the sequence of Laplace problems.

Figure 4.4 shows that the implemented algorithm recovered approximately the same speed-up observed by [20], considering that for the demo for the Laplace equation showed a speed-up of approximately 2.

4.2. The Time-Dependent Advection-Diffusion Equation

4.2.1. Homogeneous Equation

On $\Omega = [0, 1]$ the considered problem is:

$$\overbrace{\frac{\partial u}{\partial t}}^{(A)} + c\overbrace{\frac{\partial u}{\partial x}}^{(B)} = \nu\overbrace{\frac{\partial^2 u}{\partial x^2}}^{(C)} \tag{4.6}$$

$$u(0, x) = u^0(x) \tag{4.7}$$

$$u(t, 0) = l \tag{4.8}$$

$$u(t, 1) = r \tag{4.9}$$

One needs to specify the initial condition, in this case $u^0(x) = 0$, and the advective constant, $c = 1$. This study is performed with a constant diffusion coefficient $\nu = 1$.

Numerically, the time derivative (A) is discretized using backward Euler, the diffusion term (C) is discretized using Central Differences and the advection term ($\boldsymbol{b}$) is discretized using the Upwind Scheme.
The discretized system is written as:

$$\frac{U_i^k - U_i^{k-1}}{\Delta t} + c\frac{U_{i+1}^k - U_i^k}{\Delta x} = \nu\frac{U_{i+1}^k - 2U_i^k + U_{i-1}^k}{\Delta x^2} \quad \forall\, i = 1, ..., n-1,\ k = 2, ..., N_S, \tag{4.10}$$

with Δt the time step, Δx the spacial step and

$$U_0^k = l, \quad U_n^k = r \quad \forall\, k = 1, ..., N_S. \tag{4.11}$$

Finally, the system can be expressed in matrix form as:

$$A\boldsymbol{x^k} = \boldsymbol{b^k}\ , \tag{4.12}$$

where $\boldsymbol{x^k} = \boldsymbol{U^k} = \{U_i^k\}_{i=0}^n$ and $\boldsymbol{b^k}$ is the n-dimensional vector, such that $b_i^k = U_i^{k-1}$, explicit RHS.

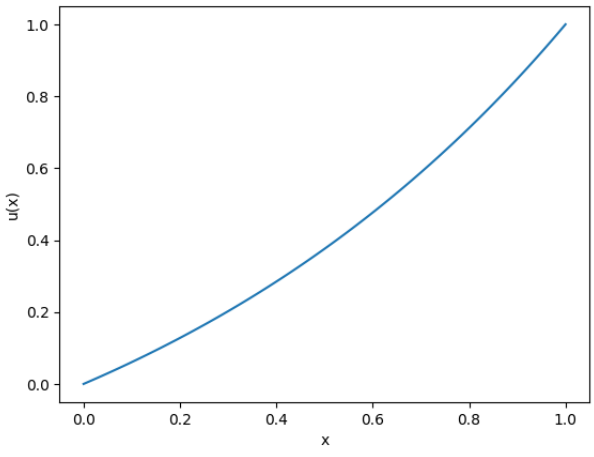


Figure 4.5: Steady State Solution for the Homogeneous Case.

4.2.2. Constant Source Term

On $\Omega = [0, 1]$ the considered problem is:

$$\overbrace{\frac{\partial u}{\partial t}}^{(A)} + c\overbrace{\frac{\partial u}{\partial x}}^{(B)} - \nu\overbrace{\frac{\partial^2 u}{\partial x^2}}^{(C)} = \overbrace{c\frac{\partial u_\pi}{\partial x} - \nu\frac{\partial^2 u_\pi}{\partial x^2}}^{(D)} \tag{4.13}$$

$$u_\pi(x) = x(1-x)\sin(N\pi x) \tag{4.14}$$

$$u(0,x) = u^0(x) \tag{4.15}$$

$$u(t,0) = l \tag{4.16}$$

$$u(t,1) = r \tag{4.17}$$

One needs to specify the initial condition, in this case $u^0(x) = 0$, and the advective constant, $c = 1$. This study is performed with a constant diffusion coefficient $\nu = 1$ and with $N = 10$.
Numerically, the time derivative (A) is discretized using backward Euler, the diffusion term (C) is discretized using Central Differences and the advection term (B) is discretized using the Upwind Scheme, while (D) is computed for each point of the discretization.

The discretized system is written as:

$$\frac{U_i^k - U_i^{k-1}}{\Delta t} + c\frac{U_{i+1}^k - U_i^k}{\Delta x} = \nu\frac{U_{i+1}^k - 2U_i^k + U_{i-1}^k}{\Delta x^2} + f_i \quad \forall\, i = 1, ..., n-1,\ k = 2, ..., N_S, \tag{4.18}$$

with

$$U_0^k = l, \quad U_n^k = r \qquad \forall\, k = 1, ..., N_S, \quad f_i = \left(c\frac{\partial u_\pi}{\partial x} - \nu\frac{\partial^2 u_\pi}{\partial x^2}\right)\Big|_{x=x_i}, \tag{4.19}$$

Finally, the system is studied as:

$$A\boldsymbol{x^k} = \boldsymbol{b^k} + \Delta t\boldsymbol{f}\ , \tag{4.20}$$

where $\boldsymbol{x^k} = \boldsymbol{U^k} = \{U_i^k\}_{i=0}^n$ and $\boldsymbol{b^k}$ is the n-dimensional vector such that $b_i^k = U_i^{k-1}$.

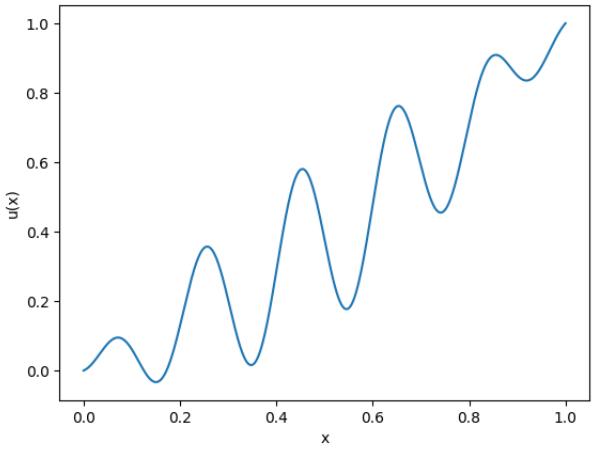


Figure 4.6: Steady State Solution for the Constant Non-Homogeneous Case.

4.2.3. Time-Dependent Source Term

Finally, on $\Omega = [0, 1]$ the considered problem is:

$$\overbrace{\frac{\partial u}{\partial t}}^{(A)} + \overbrace{c\frac{\partial u}{\partial x}}^{(B)} - \overbrace{\nu\frac{\partial^2 u}{\partial x^2}}^{(C)} = \overbrace{\frac{\partial u_\pi}{\partial t} + c\frac{\partial u_\pi}{\partial x} - \nu\frac{\partial^2 u_\pi}{\partial x^2}}^{(D)} \tag{4.21}$$

$$u_\pi(x, t) = x(1 - x)\sin(N\pi(x - t)) \tag{4.22}$$

$$u(0, x) = u^0(x) \tag{4.23}$$

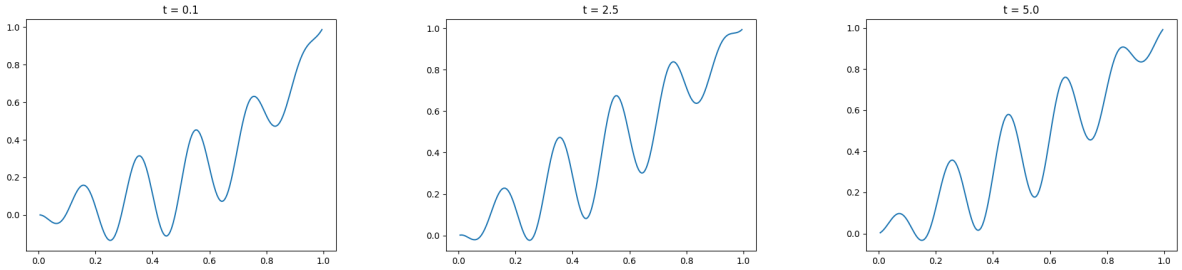
$$u(t, 0) = l \tag{4.24}$$

$$u(t, 1) = r \tag{4.25}$$

One needs to specify the initial condition, in this case $u^0(x) = 0$, and the advective constant, $c = 1$. This study is performed with a constant diffusion coefficient $\nu = 1$ and with $N = 10$.

Numerically, the time derivative (A) is discretized using backward Euler, the diffusion term (C) is discretized using Central Differences and the advection term (B) is discretized using the Upwind Scheme, while (D) is computed for each point of the discretization. The discretized system is written as in Equation (4.18).

Figure 4.7 presents the solutions of the considered problem at different time instances.



(a) At the first iteration the solution has not yet converged to the stable one.

(b) At time iteration 50 the solution is stable, but time-dependent.

(c) The solution is different to the one at $t = 2.5$

Figure 4.7: Time-Dependent Solutions of the Time-Dependent Source Case.

4.2.4. Results of the Homogeneous Case

In order to test the implemented algorithm on a larger dimension n is set to 100. In this case, $N_S = 1000$ is the number of time iterations, hence, the resulting procedure corresponds to solving a set of 1000 systems each of size 100×100, fixing the tolerance to 10^{-10} and $m = 4$. This choice of parameters allows to compare the results with the ones obtained by [20], despite that the authors in [20] used a maximum dimension of $n = 40$. Moreover, $\Delta t = 0.05$ and $T = 50$. The simulation is run using the DNN model defined in Section 3.4.2, with 60000 parameters and with the initial set size fixed to 32. At the end of the simulation the size of the dataset is equal to 186. It is possible to plot the residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) with respect to the number of iteration. Figure 4.8 shows the comparison between the values obtained by the classic GMRES method with the zeros vector as initial guess and the ML enhanced version.

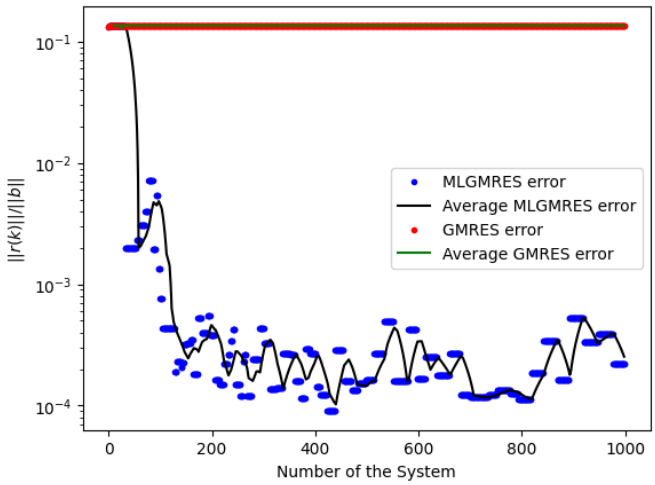


Figure 4.8: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) w.r.t. the number of systems of the homogeneous case. Obtained by using $n = 100$, $N_S = 1000$, using an initial set of dimension 32 and DenseNN.

From Figure 4.8 it is possible to acknowledge that also with a larger dimension and with a time-dependent problem the ML decorator is able to predict a better $\boldsymbol{x_0}$ in order to minimize the residual at the end of the first restart of the GMRES code.
In Figure 4.9 it is possible to see that the speed-up in this case is not as good as in the previous test, however it is still comparable to [20].

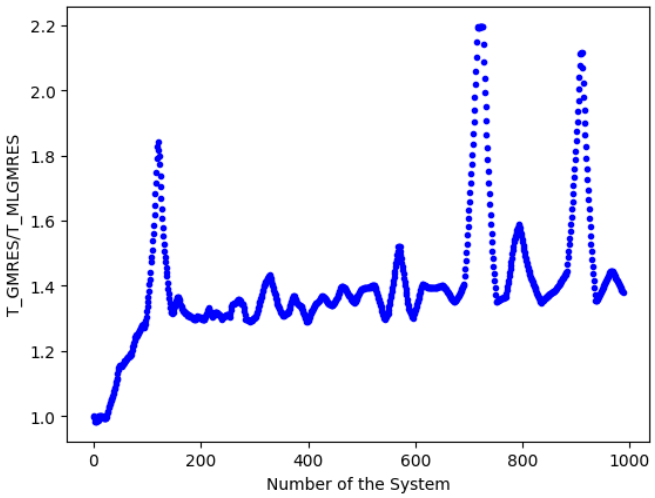


Figure 4.9: Iteration speed-up of the homogeneous case sequence.

4.2.5. Results of the Constant Non-Homogeneous Case

The ML algorithm is tested on a set of 1000 systems each of size 100×100, fixing the tolerance to 10^{-10} and $m = 4$. The simulation is run using the DNN model defined in Section 3.4.2, with 60000 parameters and with the initial set size fixed to 32. At the end of the simulation the size of the dataset is equal to 191.
The same figures of Section 4.2.4 are again plotted for the non-homogeneous case.
Figure 4.10 demonstrates that the implemented model is able to predict good initial guesses to feed into the GMRES method.

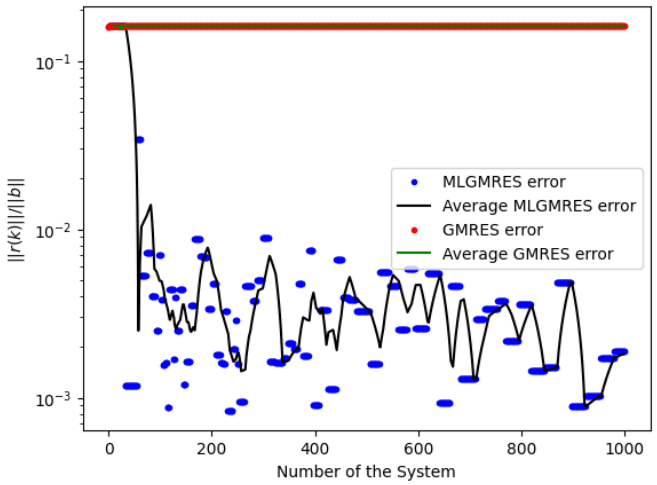


Figure 4.10: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x}_0^k)$) w.r.t. the number of systems of the constant non-homogeneous case. Obtained by using $n = 100$, $N_S = 1000$, using an initial set of dimension 32 and DenseNN.

In this case, the speed-up is worse than in the homogeneous case, however there is still a gain in time with respect to the zeros initial vector, as shown in Figure 4.11.

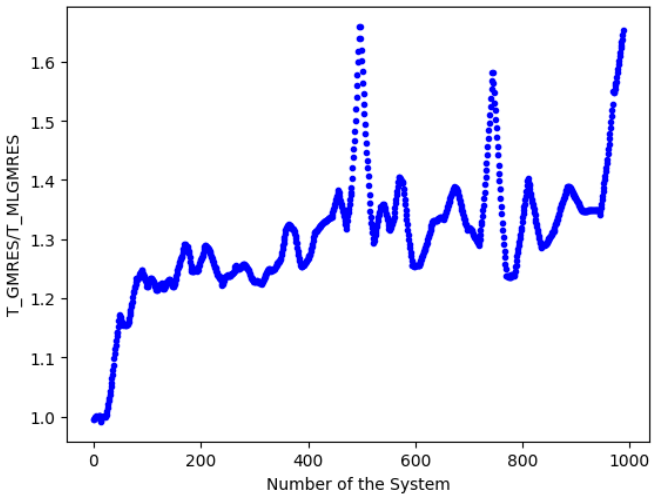


Figure 4.11: Iteration speed-up of the constant non-homogeneous case sequence.

4.2.6. Results of the Time-Dependent Non-Homogeneous Case

In order to test the ML algorithm on systems with a faster changing RHS, a time-dependent source term is introduced, as defined in Section 4.2.3. The test case consisted in a set of 1000 systems each of size 100×100, fixing the tolerance to 10^{-10} and $m = 4$. The simulation is run using the DNN model defined in Section 3.4.2, with 60000 parameters and with the initial set size fixed to 32. At the end of the simulation the size of the dataset is equal to 186.

Figure 4.10 demonstrates that the implemented model is able to predict good initial guesses to feed into the GMRES method.

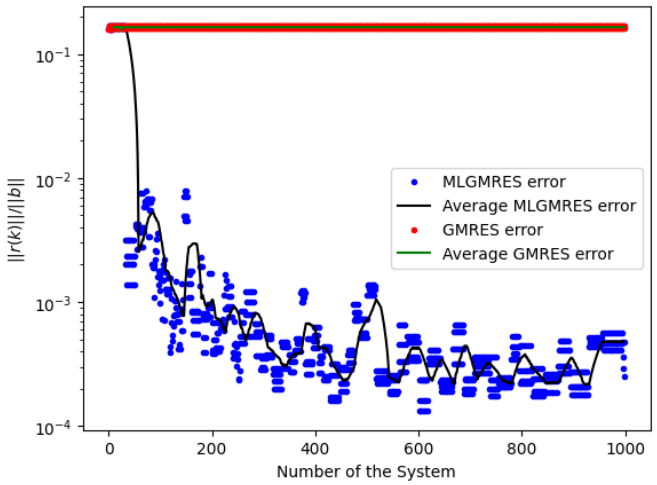


Figure 4.12: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x}_0^k)$) w.r.t. the number of systems of the time-dependent non-homogeneous case. Obtained by using $n = 100$, $N_S = 1000$, using an initial set of dimension 32 and DenseNN.

In this more complex case, the speed-up obtained is still lower than in the homogeneous case, however it is comparable to the results obtained by [20], showing an interesting gain in time with respect to the zeros initial vector, as shown in Figure 4.13.

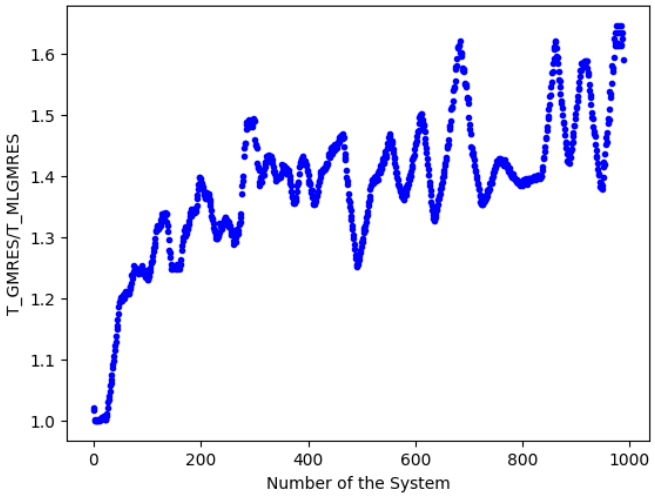


Figure 4.13: Iteration speed-up of the time-dependent non-homogeneous case sequence.

4.2.7. Using an Increasing Time Step to Generate Stiffer Systems

To increase problem stiffness while mitigating diagonal dominance, an approach is adopted that involves introducing an increasing time step. This technique aims to optimize the

numerical stability and accuracy of the system being analyzed. To accomplish this, a vector comprising ascending values is defined using the built-in function `linspace` from the `NumPy` library.
When dealing with stiff problems, conventional numerical methods might become inefficient, leading to inaccurate results or even instability in the computation. While, diagonal dominance means that the diagonal elements of a matrix dominate the off-diagonal elements in magnitude. Such dominance can occur in systems of equations, especially those arising from linearized partial differential equations or other complex mathematical models.
It is possible to observe the dependence of the system on Δt by rewriting it as:

$$\left(\frac{I}{\Delta t} + \frac{\partial R(\boldsymbol{U}^n)}{\partial \boldsymbol{U}^n}\right) \delta \boldsymbol{U}^{n+1} = \text{RHS} \,, \tag{4.26}$$

where $\delta \boldsymbol{U}^{n+1} = \boldsymbol{U}^{n+1} - \boldsymbol{U}^n$, $R(\boldsymbol{U})$ is the residual of the semi-discretized equation and RHS $= -R(\boldsymbol{U}^n) +$ Source Terms.
By introducing an increasing time step, we can effectively regulate the resolution at which the system evolves over time.
In this case, the range of values is defined to increase progressively, conforming to the requirements of the problem at hand. This newly created vector serves as a set of time steps that facilitate the temporal discretization of the problem, providing the numerical solver with the necessary information to accurately simulate the system's behavior over time.

4.2.8. Results of the Increasing Time Step Case

The implemented algorithm is tested on a set of 2000 systems each of size 100×100, fixing the tolerance to 10^{-10} and $m = 10$. The simulation is run using the DNN model defined in Section 3.4.2, with 60000 parameters and with the initial set size fixed to 32. At the end of the simulation the size of the dataset is equal to 340. The time step assumes values in the interval $[0.01, 5]$. The number of matrix-vector products needed to reach convergence for each single system solved by the GMRES method can be studied. Figure 4.14 displays the behaviour of the ML enhanced version of the GMRES code, confirming indeed that as the simulation advances, the network is learning to predict a better $\boldsymbol{x_0}$, indeed the number of matrix-vector products increases as the simulation goes on for both methods, since the systems become stiffer as the time step increases. Indeed, it is easy to see that the number of matrix-vector products of the implemented algorithm is always lower than the number needed by the classic method.

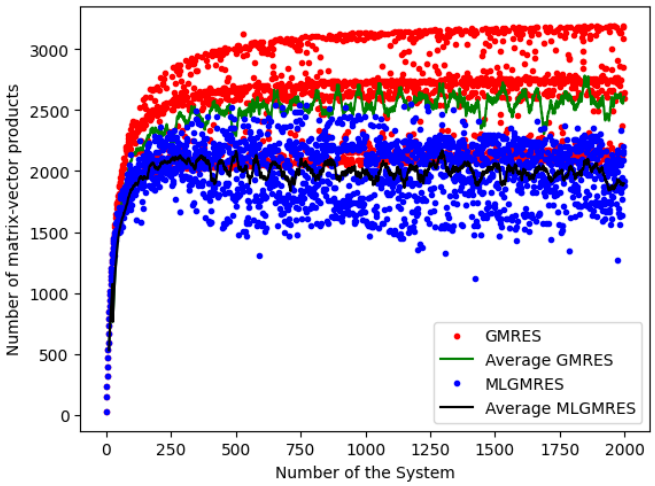


Figure 4.14: Number of matrix-vector products w.r.t. the number of systems.

Then, Figure 4.15 shows the comparison between the values of the normalized residual at the end of the first restart obtained by the classic GMRES method with the zeros vector as initial guess and the ML enhanced version. From the figure it is possible to observe the learning taking place, as the normalized residual plotted decreases as the simulation goes on.

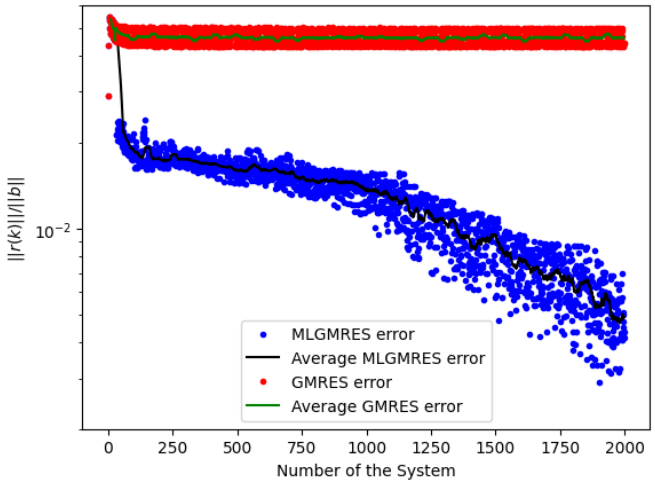


Figure 4.15: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) w.r.t. the number of systems.

Moreover, Figure 4.16 shows the behaviour of the normalized residual for the final system of the simulation with respect to the number of matrix-vector products. From the figure it is possible to observe that already the norm of the initial residual $\|\boldsymbol{r_0^1 000}\| = \|\boldsymbol{b} - A\boldsymbol{x_0^1 000}\|$ is a little smaller than the initial residual with the classic choice of taking $\boldsymbol{x_0} = [0, ..., 0]^T$, yet this small difference produces a much significant difference in convergence.

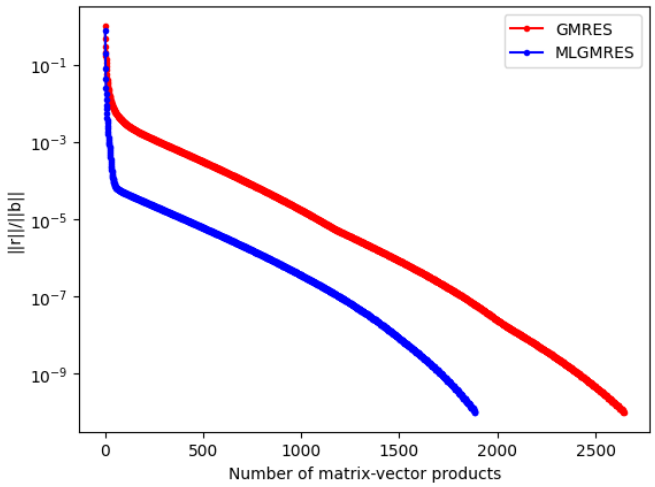


Figure 4.16: Behaviour of the norm of the normalized residual for the last problem w.r.t. the number of matrix-vector products.

Figure 4.17 shows the speed-ups with respect to the number of the system. Also in this stiffer case it is observed that the ML prediction results in much faster convergence, with approximately 30% of the time spent solving systems saved.

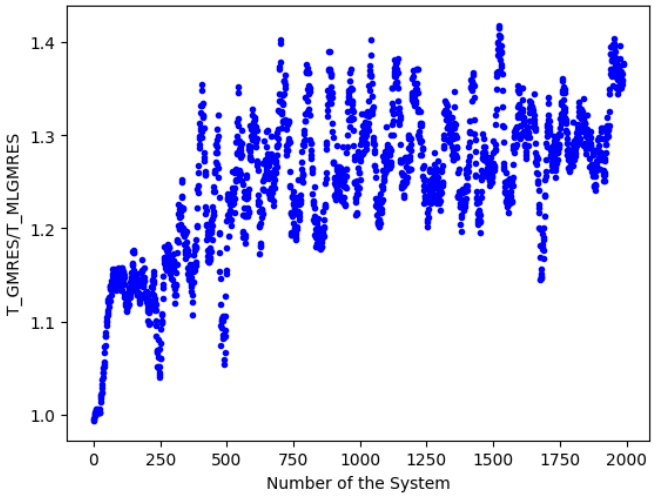


Figure 4.17: Iteration speed-up.

Finally, it is important to notice that in this case, as well as in the previously defined ones, the total times of the simulations using the ML enhanced code, considering all the GMRES computations, all the trainings of the model and all the other operations, are smaller than the total times of the simulations with the classic GMRES algorithm, in other words, the gain induced by using a ML-based initial guess for GMRES (training time included) supersedes the same GMRES implementation using an initial guess coming from

usual heuristics. For instance, this test case, run on the computing environment defined in Appendix D, finished with the times reported in Table 4.1.

Component	Runtime (s)	Percentage
Total non-decorated simulation	625.57	53.05
Total decorated simulation	553.63	46.95
Total simulation	1179.20	100

Table 4.1: Runtimes of the different components of the total simulation.

Therefore resulting with a 11.50% of real time saved by Machine Learning for this particular case and with respect to this heuristic. One could study also the different portions of the Machine Learning Simulation. As shown in Table 4.2.

Component	Runtime (s)	Percentage
MLGMRES (only GMRES time)	500.28	90.36
Training	44.27	8.00
Other	9.08	1.64

Table 4.2: Runtimes of the different components of the decorated simulation.

One must note that the summation of the percentages of the GMRES algorithm and of the training does not give 100%, that is because the simulation contains also the prediction times and other less relevant computations.

4.2.9. Results with GNNs

Since, until now the initial guess was predicted only from the RHS $\boldsymbol{b}$ vector, it is interesting to investigate what happens when the hybrid model defined in Section 3.4.2 is deployed, in other words, to see the effect of using matrix features to extrapolate the matrix importance in the choice of the initial guess. The features used are very simple to extract, the values on the principal diagonal of the matrix and the value of diagonal dominance, defined as:

$$dd_i = \frac{|a_{ii}|}{\sum_{i \neq j} |a_{ij}|} \,, \tag{4.27}$$

where a_{ij} is the element of A in position (i, j).
As detailed in Section 3.4.2, the implemented GNN-based architectures uses a combination

of GNN and CNN layers.
Considering that this test case has a matrix that changes only with regards to the time step, and considering that the implemented model uses CNNs and not DNNs, the results will very much differ from the previous section.
The implemented algorithm is tested on a set of 2000 systems each of size 100×100, fixing the tolerance to 10^{-10} and $m = 10$. The model has 540179 parameters and the initial set size is fixed to 32. At the end of the simulation the size of the dataset is equal to 105. The dataset's final size is significantly smaller compared to the previous case, due to the GNN model's training process being computationally intensive. Consequently, to add a problem to the dataset and undergo retraining, more stringent conditions were imposed. The time step assumes values in the interval $[0.01, 5]$. By observing Figure 4.18, indeed, it is understandable that the new method is not as efficient as previously shown, still the results show that the moving average of number of matrix-vector products mostly remains below the average of the classic method.

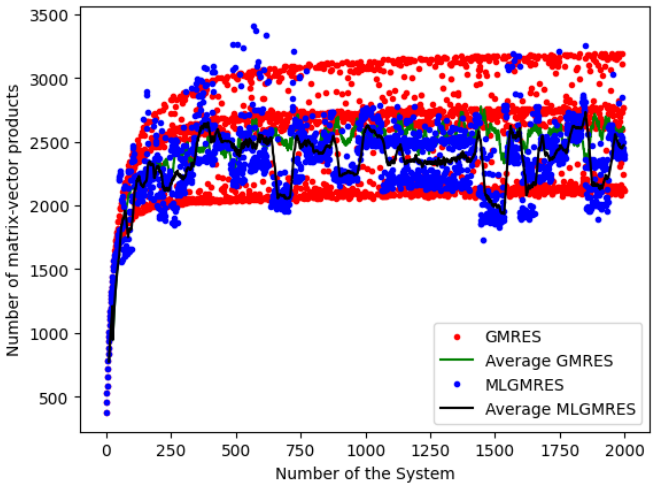


Figure 4.18: Number of matrix-vector products w.r.t. the number of systems. Obtained by using an initial set of dimension 32 and the hybrid NN architecture.

The speed-ups, shown in Figure 4.19, also confirm that the model was not optimal for the considered test case. Yet, it is anyway possible to observe better times with the ML enhancement.

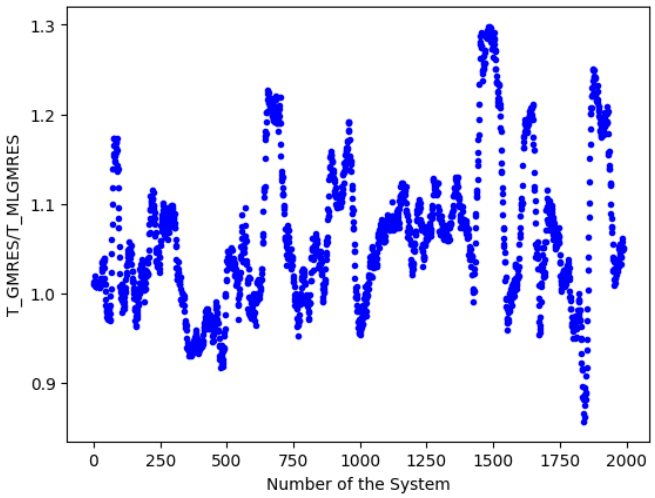


Figure 4.19: Iteration speed-up using the hybrid model.

From this results it is important to notice that, in test cases where the matrix does not change drastically, the use of matrix features using GNNs doesn't seem to improve model performance. In fact, quite to the contrary, the added trainable parameters seem to complexify the learning process of the network which would require a more detailed look at other trainable parameters and better tuning of the model to obtain better results. Moreover the use of Dense Neural Network, if applicable, is much more efficient that the use of Graph Neural Networks.

4.3. Recycling of the Previous Solution

In order to make a comparison between the ML enhanced algorithm and other more classical heuristics with regards to the initial guess used with GMRES, the recycling of the previous time step's solution, as defined in Section 1.3.1, is applied to strengthen the classic GMRES algorithm.
First, a comparison with the solution recycling method is performed on the Heat equation with the introduction of a time derivative, as defined in Equation (4.28):

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = f^k \tag{4.28}$$

$$u|_{\partial\Omega} = 0 \tag{4.29}$$

Then, the time-dependent Advection-Diffusion equation is taken into consideration, at first with the introduction of numerical noise in the RHS vector $\boldsymbol{b}$, in order to represent contributions from source terms that can be of a stochastic nature, while making

the system more complex to solve. Lastly, the noise is removed and the ML enhanced solver is compared to the solution recycling GMRES on the same test case as defined in Section 4.2.3.

4.3.1. Heat Equation

The simulation is performed using the same implementation defined in Section 4.1.1. After the first training is performed it is possible to observe in Figure 4.20 that the normalized residual at the end of the first restart with the predicted $\boldsymbol{x_0}$ is smaller in norm than the one recycling the previous solution. This shows that for time-dependent problems with a rapidly changing RHS the ML enhanced code can provide a better initial guess than classic heuristics.

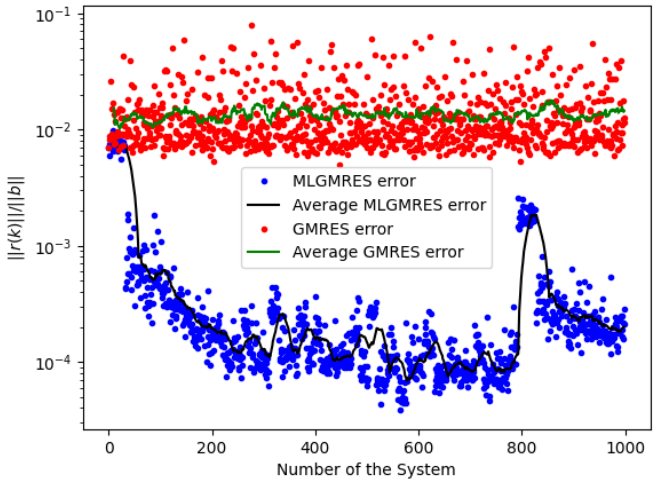


Figure 4.20: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) w.r.t. the number of systems of the Heat equation.

The capabilities of the algorithm are also shown by the computed speed-ups, plotted in Figure 4.21. In fact, the classic GMRES takes in average 20% more computational time to reach the sought convergence criterion.

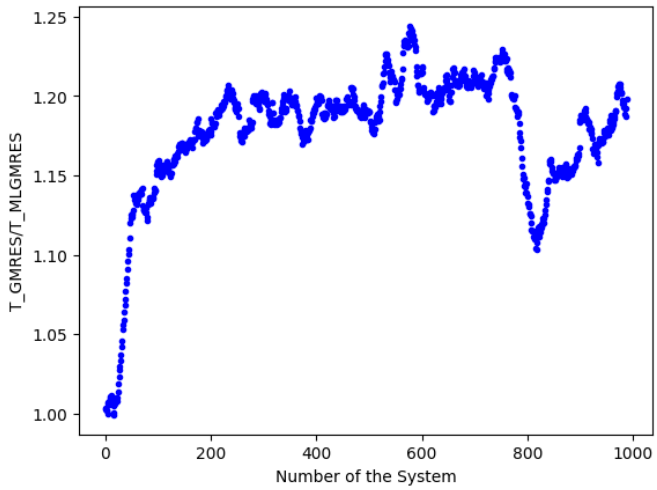


Figure 4.21: Iteration speed-up w.r.t. recycling the previous solution.

4.3.2. Time-Dependent Advection-Diffusion Problem with Increasing time step and Numerical Noise

The simulation is performed using the same implementation defined in Section 4.2.8, introducing a numerical error in the RHS vector. The time step assumes values in the interval $[2.5, 150]$.

After the first training is performed it is possible to observe in Figure 4.22 that the number of matrix-vector products with the predicted $\boldsymbol{x_0}$ is lower than the ones of the algorithm recycling the previous solution. This shows that also in this test case the ML enhanced code can provide a better initial guess than classic heuristics.

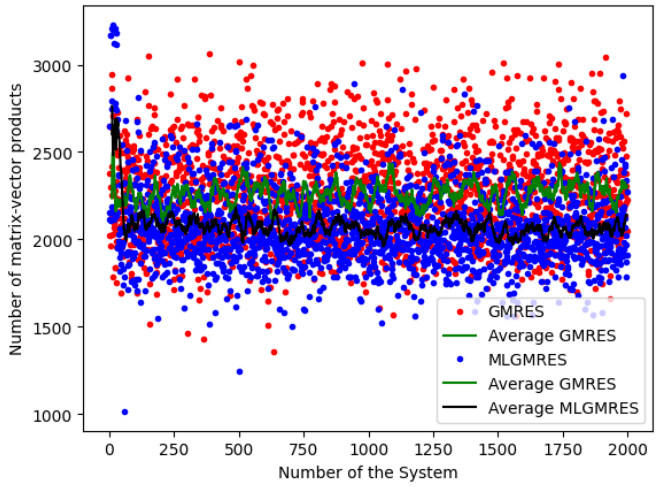


Figure 4.22: Number of matrix-vector products w.r.t. the number of systems.

The capabilities of the algorithm are also shown by the computed speed-ups, plotted in Figure 4.23. In fact, the classic GMRES takes in average 10% more computational time to reach the searched convergence criterion.

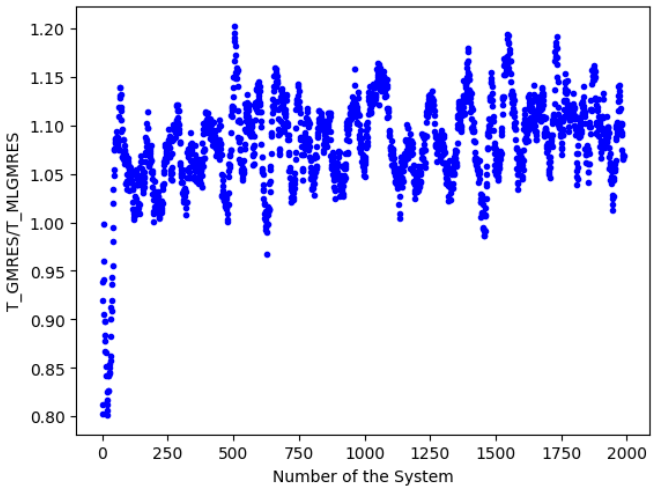


Figure 4.23: Iteration speed-up w.r.t. recycling the previous solution.

4.3.3. Advection Diffusion Problem with Increasing time step

Finally, the solution recycling method is applied to the non-perturbed Advection-Diffusion equation, still with an increasing time step. The time step assumes values in the interval $[2.5, 150]$.

By, investigating the number of matrix-vector products required in order to resolve each linear system, one can notice, as depicted in Figure 4.24, that until half of the simulation, around system 1000, the average number of products required by the implemented solver is higher than the number required by using as initial guess the solution of the previous time iteration. However, the figure also presents a decreasing trend with the black line, and an increasing one with the green line. In fact, after a certain number of systems the ML algorithm requires less iterations to reach convergence, in average.

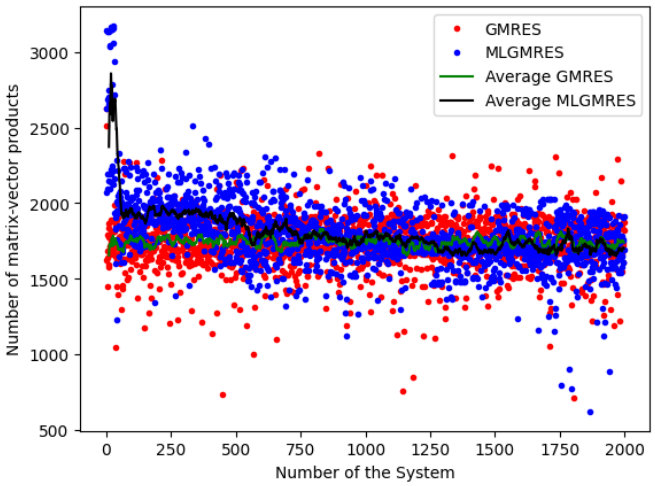


Figure 4.24: Number of matrix-vector products w.r.t. the number of systems.

The behaviour interpreted in Figure 4.24 is also sustained by Figure 4.25. In fact it is possible to see that the residual at the end of the first restart of recycling solver keeps approximately constant at around 10^{-2}, while the one of the ML solver shows indeed that the learning produces better results with each new training.

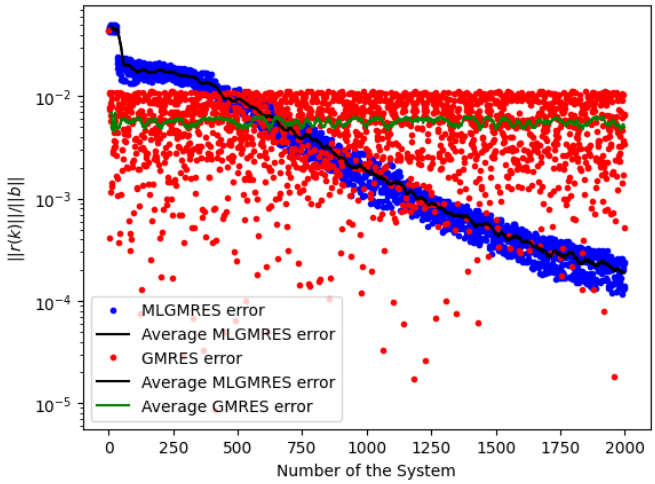


Figure 4.25: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x}_\mathbf{0}^k)$) w.r.t. the number of systems.

Finally, the speed-ups show that indeed, before the training, feeding a vector of zeros in much worse than recycling the last computed solution, as the first values in Figure 4.26 are below 1. However, as the simulation continues the same time as the recycling version is reached and even reduced by the ML counterpart, as showed in Figure 4.26.

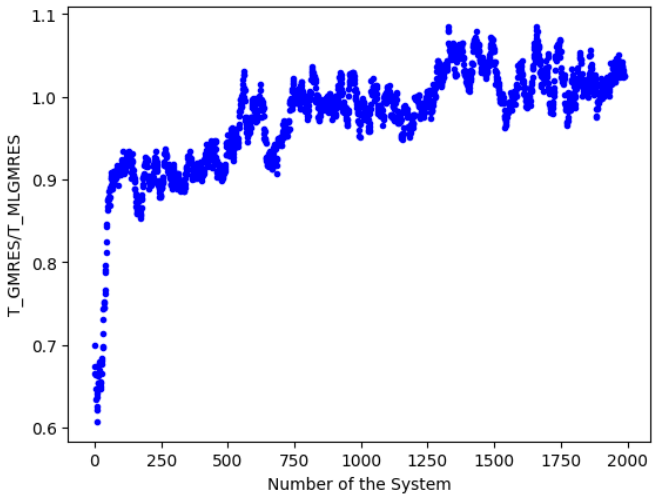


Figure 4.26: Iteration speed-up w.r.t. recycling the previous solution.

5 | Numerical Experiments on Representative Test Cases

Contents

Chapter 5 applies the prediction algorithm to representative test cases governed by the compressible Navier-Stokes equations and discretized in space using the Discontinuous Galerkin method. The chapter outlines the extraction of the problems, introduces the Discontinuous Galerkin method briefly, and discusses the time discretization strategy. Three significant test cases are presented: the subsonic flow around a cylinder, the transonic flow around a NACA0012 airfoil, and finally the Taylor-Green vortex simulation. The algorithm's efficacy in handling complex fluid dynamics scenarios is demonstrated, showcasing its applicability in realistic simulations. This chapter validates the algorithm's potential for achieving the thesis objectives.

5.1. Problem Extraction from CFD Cases using DG Discretization

High-fidelity simulation of turbulent compressible flows in aerodynamics often implies a large number of discretization elements to tackle complexity arising from physics and geometry. Furthermore, the physical modeling of more and more complex phenomena and industrial demand for accurate solutions lead to address increasingly stiff problems. This work mainly focuses on a typical Computational Fluid Dynamics (CFD) application: the research of the fixed point of the equations by means of an inexact Newton method in the contest of steady-state computation. Several 2D test-cases with Navier-Stokes (N-S) or Reynolds-Averaged Navier-Stokes (RANS) modeling are considered with a spatial DG discretization method. In a first step, the N-S model is presented, then the DG method and the time discretization, finally the *Aghora* CFD code is introduced and the test cases are described.

5.1.1. Navier-Stokes equations for gas dynamics discretization

Flow problems governed by the compressible N-S equations for gas dynamics are now considered, by using the DG method (see [29] for details). Let $\Omega \subset \mathbb{R}^d$ be a bounded domain where d is the space dimension (typically 2 or 3). Consider the following problem:

$$\partial_t \boldsymbol{u} + \nabla \cdot \boldsymbol{f}_c(\boldsymbol{u}) - \nabla \cdot \boldsymbol{f}_v(\boldsymbol{u}, \nabla \boldsymbol{u}) = 0, \quad \text{on } \Omega \times (0, +\infty) \tag{5.1}$$

with initial condition $\boldsymbol{u}(\cdot, 0) = \boldsymbol{u}_0(\cdot)$ in Ω and appropriate boundary conditions prescribed on $\partial\Omega$. The vector

$$\boldsymbol{u} = \begin{pmatrix} \rho \\ \rho \boldsymbol{v} \\ \rho E \end{pmatrix} \tag{5.2}$$

represents the conservative variables with ρ the density, $\boldsymbol{v} = (u, v, w)^T$ the velocity vector and $E = e + |\boldsymbol{v}|^2/2$ the total specific energy where e denotes the internal specific energy. The nonlinear convective and diffusive fluxes in Equation (5.1) are defined by

$$\boldsymbol{f}_c(\boldsymbol{u}) = \begin{pmatrix} \rho u & \rho v & \rho w \\ \rho u^2 + p & \rho uv & \rho uw \\ \rho uv & \rho v^2 + p & \rho vw \\ \rho uw & \rho vw & \rho w^2 + p \\ \rho Hu & \rho Hv & \rho Hw \end{pmatrix} \tag{5.3}$$

and

$$\boldsymbol{f}_v(\boldsymbol{u}, \nabla \boldsymbol{u}) = \begin{pmatrix} 0 \\ \tau \\ \boldsymbol{v}^T \tau - \boldsymbol{q}^T \end{pmatrix}, \tag{5.4}$$

where p is the pressure, defined by an equation of state of the form $p = (\gamma - 1)\rho e$, with γ the ratio of specific heats, $H := E + \frac{p}{\rho}$ is the total specific enthalpy, $\boldsymbol{q} = -k\nabla T$, with k the thermal conductivity and T the temperature, is the heat flux vector of Fourier's heat conduction law, $\tau = \mu\,(-\frac{2}{3}(\nabla \cdot \boldsymbol{v})\boldsymbol{I} + \nabla\,\boldsymbol{v} + \nabla\,\boldsymbol{v}^T)$ is the viscous stress tensor and $\mu = \mu(T) = \mu(T_s)\sqrt{\frac{T}{T_s}}\left(\frac{1+\frac{C_s}{T_s}}{1+\frac{C_s}{T}}\right)$ is the kinematic viscosity defined by Sutherland's law as a function of the reference temperature T_s with $C_s = 110.4K$ [29].
By applying the time mean operator:

$$\overline{\boldsymbol{v}} = \frac{1}{T}\int_{t_0}^{t_0+T} \boldsymbol{v}(\boldsymbol{x}, t)\, dt \tag{5.5}$$

to the system in Equation (5.1) and by introducing the decomposition $\boldsymbol{v} = \overline{\boldsymbol{v}} + \boldsymbol{v}'$ one can obtain the RANS equations [27]. Since, these equations contain the Reynolds stress tensor, representing the turbulent fluctuations in the flow, ones needs a turbulence model to close the system. In this thesis the RANS are coupled with the one-equation turbulence model of Spalart-Allmaras [31], which couples the system defined by Equation (5.1) with Equation (5.6), in order to define the eddy viscosity ν_t.

$$\begin{aligned} \frac{\partial \nu_t}{\partial t} + \boldsymbol{v} \cdot \nabla \nu_t - c_{b1}(1 - f_{t2}) S_t \nu_t = \\ - \left[c_{w1} f_w - \frac{c_{b1}}{\kappa^2} f_{t2}\right] \left(\frac{\nu_t}{d}\right)^2 + \frac{1}{\sigma}\left[(\nabla \cdot ((\nu + \nu_t)\nabla \nu_t)) + c_{b2}|\nabla \nu_t|^2\right] \end{aligned} \tag{5.6}$$

All the details and additional definitions can be found at [31].

5.1.2. Discontinuous Galerkin Discretization

The principle of the Discontinuous Galerkin (DG) approach is based on a division of the domain Ω into a regular mesh Ω_h, without overlapping or empty element, h being defined as a characteristic size of the elements (i.e. a measure of the elements is chosen and h will be the maximum among all the elements of the mesh). We will denote κ a given element of Ω_h.
A finite-dimensional space $\mathcal{V}_h^p$ is introduced, defined as:

$$\mathcal{V}_h^p = \{\phi \in L^2(\Omega_h) \mid \phi_{|\kappa} \circ F_\kappa \in \mathcal{P}^p(I^d), \forall \kappa \in \Omega_h\}, \tag{5.7}$$

where $\mathcal{P}^p(I^d)$ is the space of polynomials in d variables of degrees at most p, and defined on the unit cube with $I = [-1; 1]$ (F_κ representing a bijection between I^d and κ). Each physical element κ is the image of I^d through the mapping F_κ. The numerical solution of Equation (5.1) is sought under the form

$$\boldsymbol{u}_h(x,t) = \sum_{l=1}^{N_p} \phi_\kappa^l(x) \boldsymbol{U}_\kappa^l(t), \quad \forall x \in \kappa, \kappa \in \Omega_h, \forall t \geq 0, \tag{5.8}$$

where $(\mathbf{U}_\kappa^l)_{1 \leq l \leq N_p}$ are the degrees of freedom (DOFs) in the element κ. The subset $(\phi_\kappa^1, \ldots, \phi_\kappa^{N_p})$ constitutes a hierarchical and orthogonal modal basis of $\mathcal{V}_h^p$ restricted onto the element κ and N_p is its dimension.
The semi-discrete form of Equation (5.1) reads: find $\mathbf{u}_h$ in $[\mathcal{V}_h^p]^{d+2}$ such that for all v_h in $\mathcal{V}_h^p$ we have

$$\int_{\Omega_h} v_h \partial_t \boldsymbol{u}_h dx + \mathcal{L}_c(\boldsymbol{u}_h, v_h) + \mathcal{L}_v(\boldsymbol{u}_h, v_h) = 0 \, . \tag{5.9}$$

The formulation of the discretizations of the convective and diffusive fluxes $\mathcal{L}_c(\boldsymbol{u}_h, v_h)$ and $\mathcal{L}_v(\boldsymbol{u}_h, v_h)$ as well as the details of the calculations are not presented in this report since they would be too far from the main subject, namely the resolution of linear systems. More details are provided in [3] and [4].

5.1.3. Time Discretization

Once the previous steps have been implemented, we obtain a semi-discrete formulation of the basic problem, the spatial discretization having been carried out: it therefore remains to discretize the temporal dimension of the equations. For this, a (potentially

non-constant) time step $\Delta t^{(n)} > 0$ is chosen, such that $t^{(n+1)} - t^{(n)} = \Delta t^{(n)}$ and $t^{(0)} = 0$.

Several choices of diagrams of implicit temporal integration are then possible, like Euler, Crank-Nicolson or Runge-Kutta for example. For the sake of clarity, the time discretization through the Backward Euler scheme is introduced:

$$\frac{1}{\Delta t^{(n)}} \boldsymbol{M} \left(\boldsymbol{U}^{(n+1)} - \boldsymbol{U}^{(n)} \right) + \boldsymbol{R}(\boldsymbol{U}^{(n+1)}) = 0 \, , \tag{5.10}$$

where:

- $\boldsymbol{M}$ is the diagonal mass matrix, resulting from the discretization by Galerkin method.
- $\boldsymbol{U}^{(n)}$ is the vector of the DOFs for the conservative variables studied, and on all the elements of the domain. The exponent indicates the time step considered.
- $\boldsymbol{R}$ is the residual vector function defined by the discretization of the convective and diffusive fluxes. It is evaluated at $\boldsymbol{U}$.

The Jacobian matrix of the residual is then computed in an approximate way. This yields the following equation:

$$\boldsymbol{A}^{(n)} \delta \boldsymbol{U}^{(n+1)} = -\boldsymbol{R}(\boldsymbol{U}^{(n)}) \, , \tag{5.11}$$

where:

- $\delta \boldsymbol{U}^{(n+1)} := \boldsymbol{U}^{(n+1)} - \boldsymbol{U}^{(n)}$ is the change on $\boldsymbol{U}$ from one time step to the next.
- $\boldsymbol{A}^{(n)} := \frac{1}{\Delta t^{(n)}} \boldsymbol{M} + \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{U}} \Big|_{\boldsymbol{U}=\boldsymbol{U}^{(n)}}$ is called implicit matrix (with $\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{U}} \Big|_{\boldsymbol{U}=\boldsymbol{U}^{(n)}}$ the Jacobian of $\boldsymbol{R}$ at point $\boldsymbol{U}^{(n)}$).

In this form, the search for the approximate solution $\boldsymbol{U}^{(n+1)}$ at time $n+1$ is reduced to solving a system of linear equations, since the values of $\boldsymbol{U}$ at previous times are known. The matrix of the system as well as the second member are brought to change with each step of time but are known at the time of the resolution. One could rewrite this sequence of equations in the more classical form:

$$A^{(n)} \boldsymbol{x}^{(n)} = \boldsymbol{b}^{(n)} \, . \tag{5.12}$$

Due to the nature of the problem, the resulting matrices are real, non-symmetric, non-positive-definite, with a block-wise structure and a symmetric pattern.
The time step is limited locally according to a CFL-like condition:

$$\Delta t_\kappa^{(n)} = CFL\Big(\frac{h_\kappa}{\lambda_{max}(\boldsymbol{u})},\ \frac{\rho h_\kappa^2}{2\frac{\mu}{Pr}}\Big), \tag{5.13}$$

where the value of the Courant number CFL is defined as a function of a user-defined parameter CFL_{in} and the maximum eigenvalue of convective fluxes, $\lambda_{max}(\boldsymbol{u})$, is evaluated from the mean value of the numerical solution in the element κ and $Pr = 0.72$ is the Prandtl number. For unsteady computations, a global time step is used and set $CFL = CFL_{in}$ and $\Delta t^{(n)} = \min_{\kappa\in\Omega_h} \Delta t_\kappa^{(n)}$, while for steady-state computations a local time step and the pseudo-transient continuation technique are used [8] where the CFL number is related to norms of the residuals according to the following relations:

$$CFL = \min\left(\tfrac{1}{r}, CFL_{in}\right), \tag{5.14}$$

$$r = \max\left(\frac{\|\boldsymbol{R}(\boldsymbol{U}^{(n)})\|}{\|\boldsymbol{R}(\boldsymbol{U}^{(0)})\|}, \frac{\|\boldsymbol{R}(\boldsymbol{U}^{(n)})\|_\infty}{\|\boldsymbol{R}(\boldsymbol{U}^{(0)})\|_\infty}\right). \tag{5.15}$$

A user-defined constant time step may also be imposed throughout the computation. In general, the computation can be stopped either when a given number of physical time steps N_T is reached, or when a given physical time is reached, or when the global L^2-norm of the explicit residuals is below a given tolerance ε_{res}.

5.1.4. The Aghora Code

The research code developed at ONERA, called ***Aghora***, offers a parallel implementation of the DG method presented in this chapter. Several turbulence models are available, and since the first versions of the code, many improvements have been added in order to take into account new models. *Aghora* is written in Fortran 2003 and comprises approximately 130,000 lines of code, spread over several Fortran modules, which offers great flexibility and good maintainability of the code. Calls to Intel's Math Kernel Library (MKL) are made, in particular for BLAS and LAPACK functions usual in linear algebra (matrix-vector products, resolution of eigenvalue problems, descent or rise method for solving triangular linear systems, etc.).

In the presented contex, the core of the code is required to solve a series of linear systems of the form $A^{(i)}x^{(i)} = b^{(i)}$, and a substantial part of *Aghora*'s computation time (for a given simulation) is spent solving these linear systems. Currently, it is the preconditioned, with the block ILU(0) algorithm, GMRES algorithm [26] with restart which is used in the solver, with the flexible variants or with deflation FGMRES, GMRES-DR. These iterative methods have been the subject of several efficiency studies in a DG or

Finite Volume contexts [17, 22, 28, 29]. The flexible version of the code is devoted to the need of accuracy, therefore, it is not required for steady-state calculations, where a rough approximation is enough.
In the context of this research endeavor, the GMRES and Predictor Codes were executed using the Python programming language. To generate the matrices of interest and their corresponding right-hand side vectors, the computational framework of *Aghora* was employed.

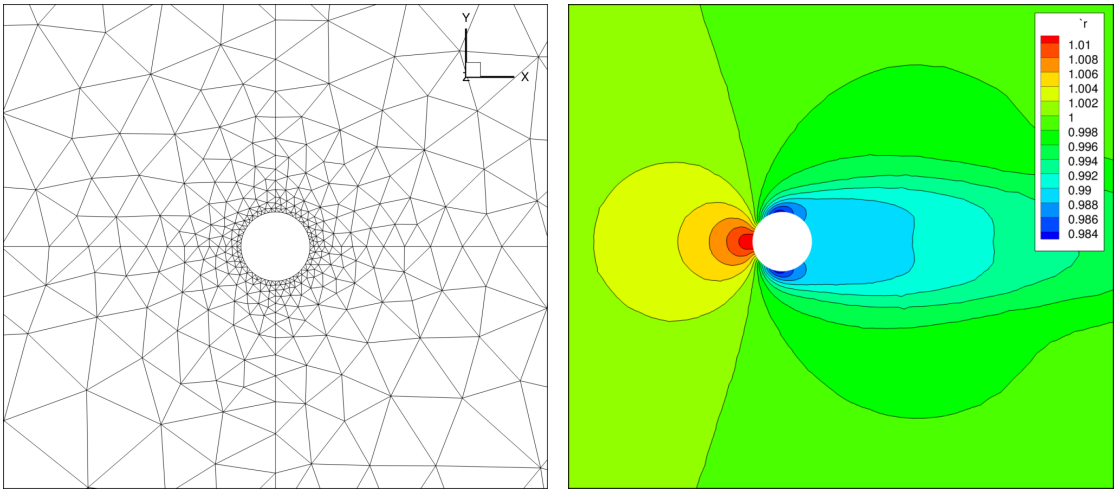
5.1.5. About the Dimension of the Systems and Neural Networks

Due to the substantially larger dimensions of the upcoming test cases compared to the previously examined ones, employing the previously tested DNN-based model is impractical. The reason is that the DNN model demands a considerable number of parameters, which exceeds the available memory capacity of the ONERA system, Appendix D. Consequently, utilizing the DNN-based approach for testing the forthcoming systems is not feasible within the current computational constraints.
To address this issue and facilitate the evaluation of the larger-scale systems, an alternative approach will be adopted. Specifically, the testing will be conducted using the CNN-based architecture, as introduced and described in Section 3.4.2. Unlike the DNN model, the CNN architecture is known for its ability to efficiently process and extract relevant features from high-dimensional data, making it more suitable for handling the increased complexity of the upcoming test cases.
By leveraging the CNN-based model, we aim to maintain the quality and accuracy of the evaluations while mitigating the computational requirements, thus ensuring that the testing process remains feasible and within the capabilities of the ONERA system's memory constraints. This decision allows us to effectively explore and assess the performance of the model on the larger systems without compromising the reliability of the results.

5.2. Laminar Flow around a Cylinder at Low Reynolds Number

Understanding the flow characteristics around a cylinder is of great importance in various engineering applications. This work focuses on a 2D laminar flow over a cylinder using a DG approach. The fluid dynamics are modeled using the RANS equations [27], with the one-equation turbulence model of Spalart-Allmaras [31]. The flow operates at a Mach

number (M_∞) of 0.15 and a Reynolds number (Re) of 80.
The computational domain focuses on the 2D flow around a cylinder. The mesh discretization comprises 1028 triangular elements. The flow operates at a laminar state, and the specified Mach and low Re ensure a subsonic flow. These values do not produce an unsteady wake, therefore, a steady solution is searched.
The spatial discretization is performed using the DG method, with a modal basis and order of the discretization equal to 3 and 4.
The convective flux is approximated using the Roe numerical flux, which handles shock waves and discontinuities accurately. The viscous flux is treated using the BR2 method [3, 7], ensuring accurate representation of the viscous effects near the cylinder surface.
The implicit Backward Euler scheme is employed for time integration, using a local time step. This time discretization method allows for larger time steps while maintaining numerical stability, enabling efficient simulations of the flow evolution over time.



(a) Illustration of the unstructured mesh used. (b) Steady-state density field.

Figure 5.1: Cylinder test case.

5.2.1. Results of the Cylinder Test Case

The implemented algorithm is tested on a set of 98 systems each of size 30840×30840, with 3621600 non-zeros, fixing the GMRES residual tolerance to 10^{-3}, $m = 50$ and imposing a maximum of 2 restarts. The simulation is run using the CNN model, with 1500929 parameters and with the initial set size fixed to 32. At the end of the simulation the size of the dataset is equal to 42.
In this specific test case, convergence is not reached within the researched tolerance without the use of a preconditioner, therefore it would not make sense to analyse the speed-ups

result. Still it is of interest to observe the behaviour of the residuals.
Taking into consideration a system solved after the first training is performed, specifically system number 50, it is possible to observe how the predicted $\boldsymbol{x_0}$ results with better decreasing rate for the normalized residual, with respect to the classic GMRES algorithm. This fact is shown in Figure 5.2.

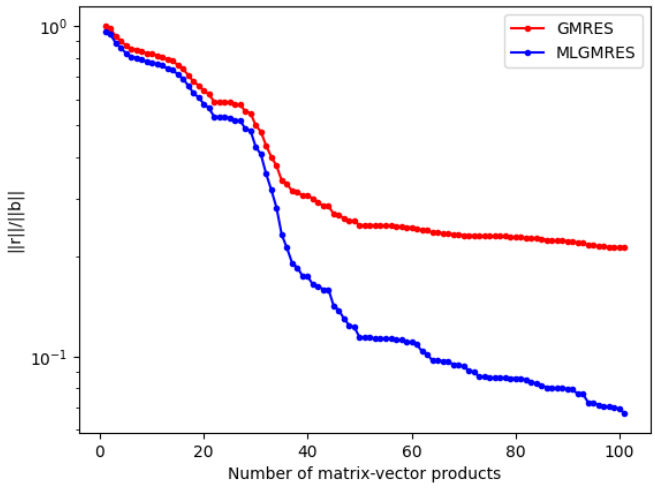


Figure 5.2: Behaviour of the norm of the normalized residual w.r.t. the number of matrix-vector products for system 50 (after training) of the cylinder test case.

Studying the last system, it is interesting to see the values of the norm of the normalized residual at the end of the two restarts. Indeed, the ML algorithm presents better values than the non enhanced version. This is quite interesting since it was possible to reach better convergence values without the use of a preconditioner for complex and large linear systems.

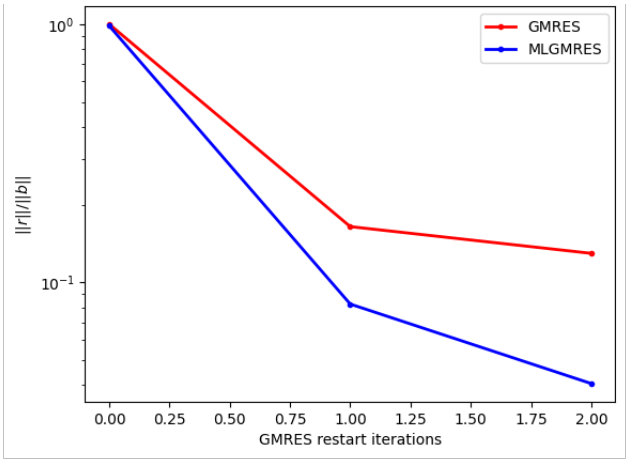


Figure 5.3: Behaviour of the normalized residual w.r.t. the number restarts for the last problem of the cylinder test case.

Finally, Figure 5.4 shows the behaviour of the residual at the end of the first restart for each system of the simulation. It can be easily extracted that after the training the values of $E_\kappa(\boldsymbol{x_0^k})$ are better using a ML predicted initial guess.

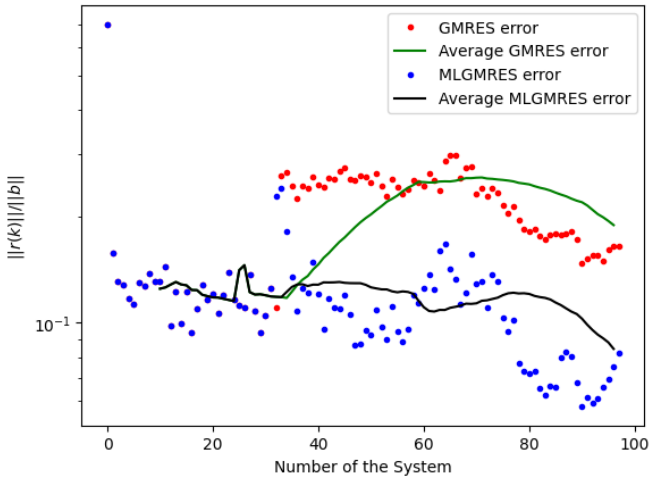
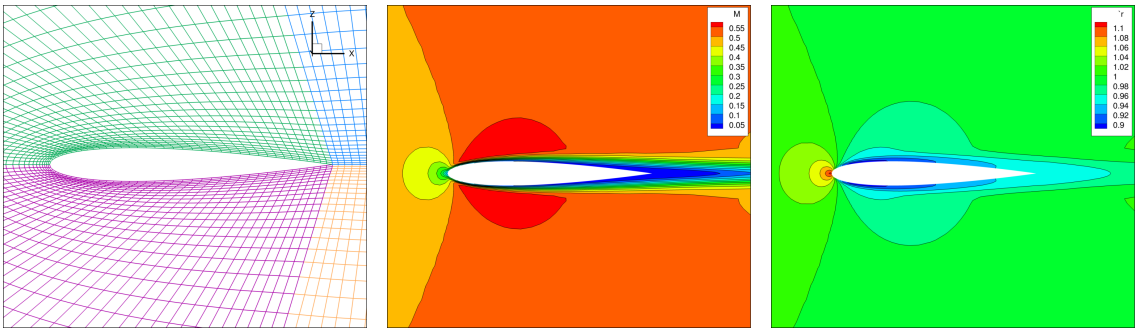


Figure 5.4: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) w.r.t. the number of systems.

5.3. Laminar Flow around a NACA0012 airfoil

The study of aerodynamics around airfoils is crucial for various engineering applications, and the NACA0012 airfoil is a standard benchmark case [35]. This work focuses on a 2D laminar flow around a NACA0012 airfoil. The fluid dynamics are modeled using the Navier-Stokes equations, capturing the behavior of the flow with $M_\infty = 0.5$, with an angle

of attack (AoA) set to 0 degrees, and a Re of 5000.
The mesh discretization comprises 1600 quadrangular elements, providing sufficient resolution to accurately capture the flow features near the airfoil surface. The spatial discretization is performed using the DG method with modal basis and a polynomial degree equal to 1 (order 2). Furthermore, to enhance computational efficiency, Reduced Basis techniques are employed, reducing the number of DOFs required for accurate representation.
The implicit Backward Euler scheme is employed for time integration.
The convective flux is approximated using the Roe numerical flux, which effectively captures shock waves and discontinuities in the flow. On the other hand, the viscous flux is handled using the BR2 method, providing accurate treatment of the viscous effects near the airfoil wall and in the wake.
In order to solve the laminar compressible N-S equations at the wall the adiabatic no-slip condition is enforced.



(a) Illustration of the mesh used. (b) Steady-state Mach number field. (c) Steady-state density field.

Figure 5.5: NACA0012 test case.

5.3.1. Results of the NACA0012

The implemented algorithm is tested on a set of 431 systems each of size 19200×19200, with 1128960 non-zeros, fixing the GMRES residual tolerance to 10^{-3}, $m = 50$ and imposing a maximum of 2 restarts. The simulation is run using the CNN model, with 1253889 parameters and with the initial set size fixed to 32. At the end of the simulation the size of the dataset is equal to 126.
By observing the behaviour of the norm of the normalized residual, it is important to note that Figure 5.6 shows that the norm of the initial residual $\|\boldsymbol{r_0}\|$ computed by the ML enhanced algorithm is greater than the one computed by the classic algorithm. Even if this is the case, the final value demonstrates that the learning does result with better

estimates. This phenomenon is quite peculiar, however, it could be a result of the fact that the studied system is non-symmetric, therefore the related functional to optimize is not convex, resulting with a more complex behaviour of the residual.

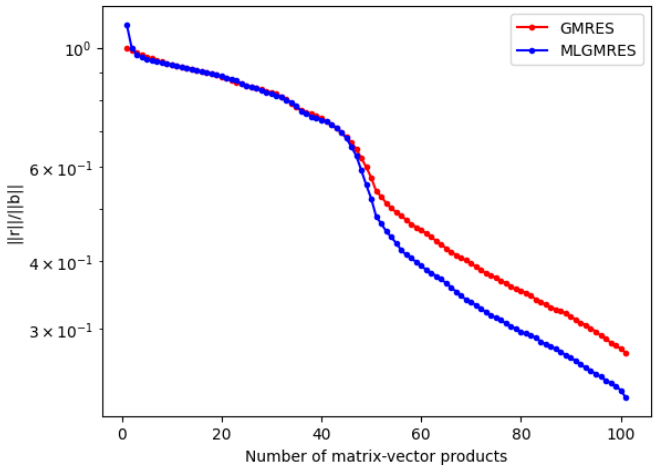


Figure 5.6: Behaviour of the norm of the normalized residual for the last system of the NACA0012 test case w.r.t. the number of matrix-vector products.

Additionally, it is interesting to investigate the values of the norm of the normalized residual at the end of the two restarts. Indeed, the ML algorithm presents better values than the non enhanced version. This is quite interesting, considering the first value observed, and it also confirms the conclusions made in Section 5.3.1, specifically that it was possible to reach better convergence values without the use of a preconditioner for complex and large linear systems.

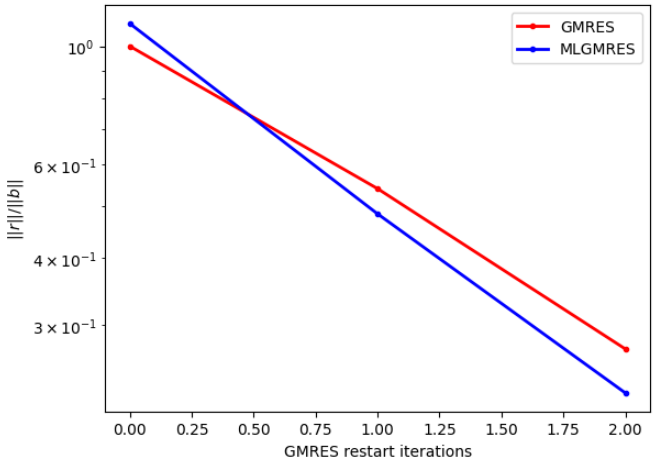


Figure 5.7: Behaviour of the norm of the normalized residual for the last system of the NACA0012 test case w.r.t. the number restarts.

Figure 5.8 shows the behaviour of the residual at the end of the first restart for each system of the simulation. This time it is quite complicated to recognize the difference between the two values, therefore, a further study could be to observe the behaviour of the residual at the end of the GMRES algorithm and therefore after two restarts.

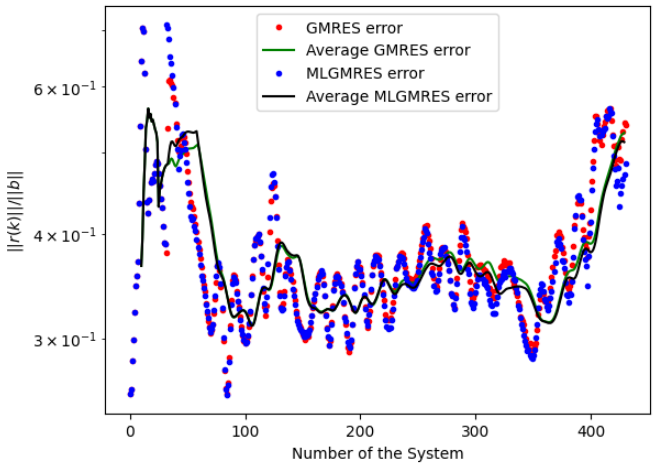


Figure 5.8: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) w.r.t. the number of systems.

Figure 5.9 shows the behaviour of the residual at the end of the second and last restart for each system of the simulation. Now it is easier to see that the values of the ML enhanced version of the algorithm are better than the ones of the classic implementation. Specifically, the two average lines have a similar behaviour, still the black line is mostly below the green one, confirming that the training does in fact produce a smaller value of the last residual and therefore a smaller final error. Probably, performing a longer simulation, with additional trainings of the NN, would improve the results.

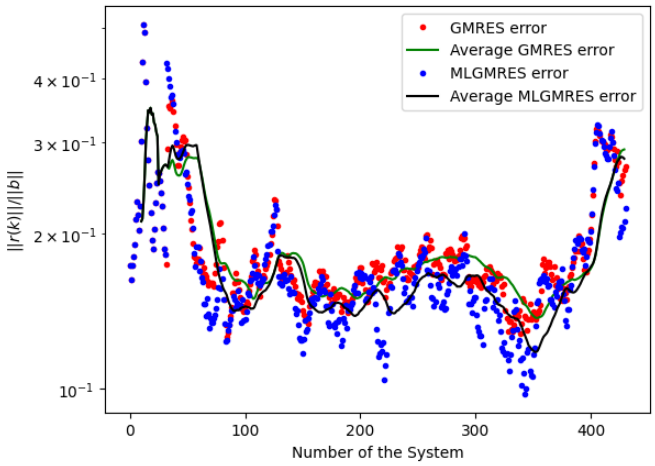


Figure 5.9: Behaviour of the norm of the normalized residual after the second restart w.r.t. the number of systems.

5.4. Taylor-Green Vortex

The Taylor-Green Vortex (TGV), a well-known benchmark problem in fluid dynamics, serves as an ideal test case for studying turbulent flows. This work focuses on a 3D turbulent flow within a box with periodic boundary conditions, utilizing Direct Navier-Stokes. The periodicity allows the simulation to evolve without any external influences, maintaining a constant total energy level throughout the decay process.
The simulation employs Direct Numerical Simulation (DNS) of a flow with M_∞ set at 0.1 and models the fluid dynamics using the Navier-Stokes equations. The computational domain consists of a Cartesian mesh of 64 hexahedral elements. The simulation employs the fourth-order Rosenbrock-like scheme (ROS44) for time discretization. This choice guarantees high temporal accuracy and stability, enabling a precise representation of the flow's transient behavior during the decay process. This case, being outside of the considered framework, is used as a last validation of the ML approach.

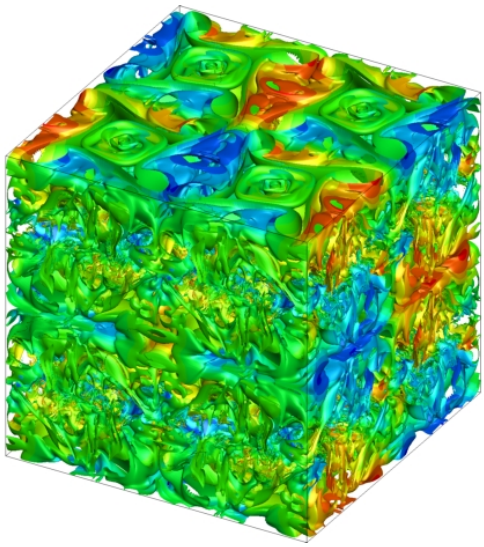


Figure 5.10: Visualisation of the solution of the Taylor-Green Vortex.

5.4.1. Results of the Taylor-Green Vortex

As final test case, the implemented algorithm is tested on a set of 4000 systems each of size 2560×2560, with 716800 non-zeros, fixing the GMRES residual tolerance to 10^{-3}, $m = 50$ and imposing a maximum of 2 restarts. The simulation is run using the CNN model, with 265729 parameters and with the initial set size fixed to 32. At the end of the simulation the size of the dataset is equal to 695.
This final test case does reach tolerance without the use of a preconditioner. Therefore, all the previous metrics can be analyzed.
First, Figure 5.11 shows the number of matrix-vector products needed to reach the desired tolerance. It is possible to see that, since after the first training, and except from a very small number of systems, the number of products to reach convergence with the optimized $\boldsymbol{x_0}$ is smaller than with the classic $\boldsymbol{x_0}$.

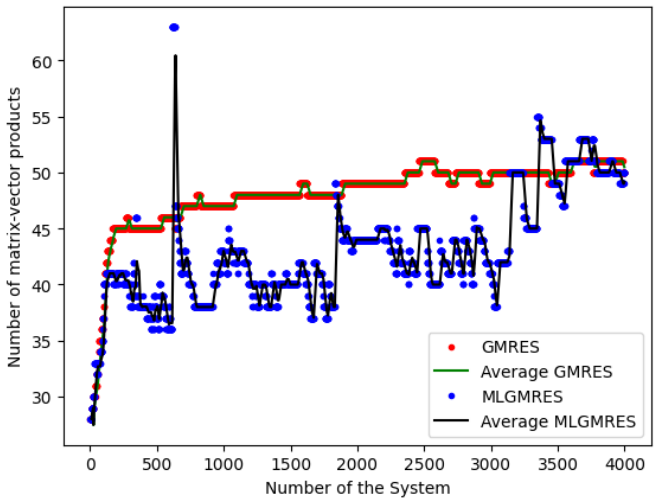


Figure 5.11: Number of matrix-vector products w.r.t. the number of systems of the TGV test case.

Figure 5.12 shows instead the speed-ups, since this time it is an interesting result to investigate. It can be seen that the value of the speed-ups is mostly above 1, showing that indeed the time to reach convergence with the ML implementation is smaller than without. Despite our expectations for speed-up improvements, there are instances where we observe speed-up values below 1, indicating sub-optimal performance. However, the retraining mechanism of the model comes to the rescue in such situations. This retraining process works diligently to address the issues, ultimately resulting in better speed-up values.

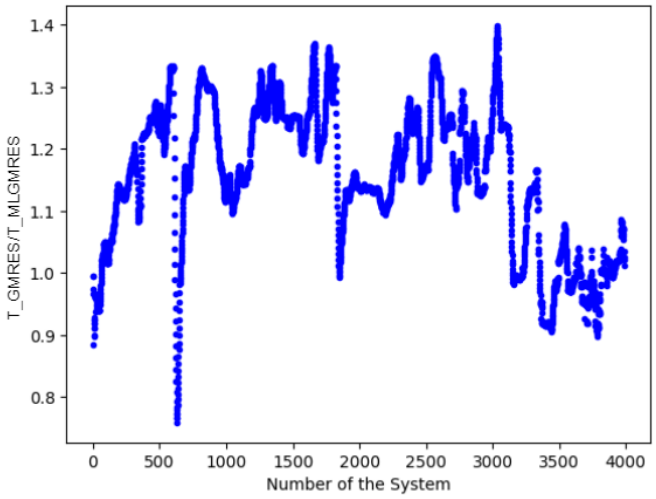


Figure 5.12: Iteration speed-up.

5.4.2. Results using other NN-based Approaches

Results with Offline Learning

Another interesting investigation is to observe what happens when applying an Offline Learning approach to the test case. Since, as observed in the previous section, having a good behaviour for a certain number of systems does not necessarily mean that the model will continue to work perfectly, one could study what happens without retraining the system and thus understand the interest of using an online approach.
The structure of the test case is precisely the same as in the previous section, with the exception that the training is performed after 2000 systems are solved. Then, the model is trained on this very large dataset and it is finally applied to a test set of 400 systems.
The plot of the number of matrix-vector products required to reach convergence, shown in Figure 5.13, shows an interesting behaviour. Even though the system is not retrained, there is a decrease in the number of products.

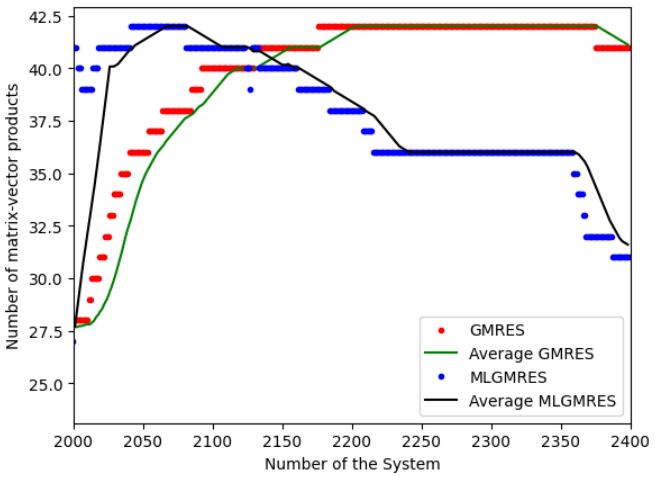


Figure 5.13: Number of matrix-vector products w.r.t. the number of systems of the test set.

The figure shows only the systems solved after the training.
Then, the speed-ups are shown. Figure 5.14 shows the moving average of the speed-ups, and it confirms the behaviour of the previous one. The time to solve the systems with the ML prediction decreases, resulting in better values of the speed-ups, reaching at the end of the simulation a value of approximately 1.4.

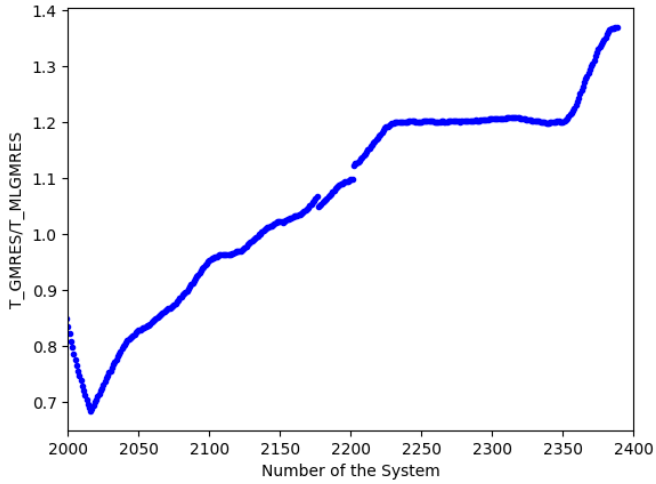


Figure 5.14: Iteration speed-up of the test set.

Results with GNNs

Finally, since Section 5.4.1 has shown great promise, as last investigation it is of interest to investigate the same TGV case using the information provided by GNNs, in order to analyze the influence of matrix features, the same defined in Section 4.2.9, on the model efficiency. The size of the Krylov subspace is changed to $m = 30$, in order to better study the value of the residual at the end of the first restart. The model is composed of 1320899 trainable parameters and with the initial set size fixed to 32. At the end of the simulation the size of the dataset is equal to 102.

By studying the behaviour of $E_\kappa(\boldsymbol{x_0})$, it is possible to see that the values of the ML prediction solver are below the values of the normal implementation. Moreover, it is possible to observe that, even having decreased m, some values are very close to the imposed tolerance.

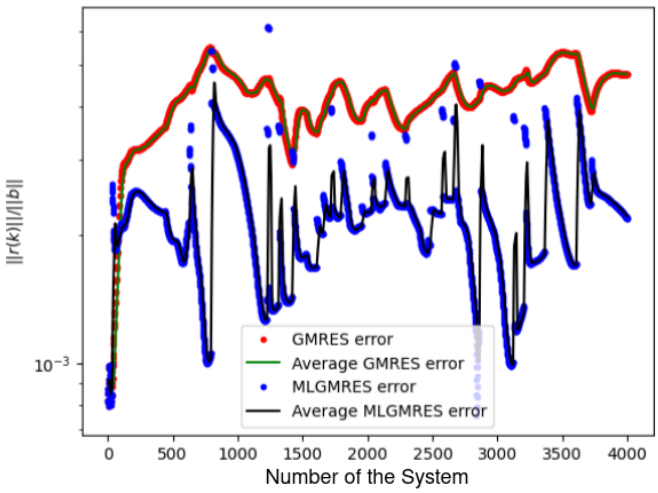


Figure 5.15: Residual at the end of the first restart ($E_\kappa(\boldsymbol{x_0^k})$) w.r.t. the number of systems. Obtained by using an initial set of dimension 32 and the hybrid NN architecture.

Finally, observing the plot of the speed-ups in Figure 5.16, it is possible to see that better values are obtained with respect to the case where only CNNs where being used. This confirms the assumption that capturing matrix features in the model is essential to obtain better results.

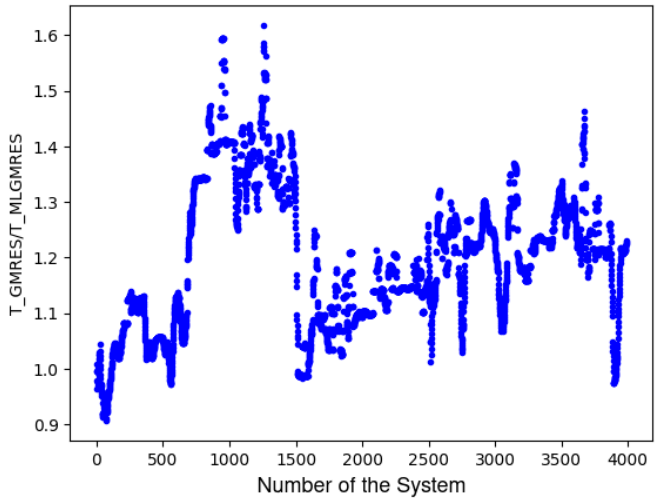


Figure 5.16: Iteration speed-up using the hybrid model.

However, when considering the total times, the one of the ML simulation is much larger than the time spent by the classic GMRES method. This is a result of the fact that the trainings of the GNN-based model require large computational times, resulting in a longer simulation and also CNNs seem to be more optimized within Pytorch.

Conclusion

This thesis has presented an approach, based on Machine Learning techniques, that aims to deliver an adequate initial guess for the iterative Krylov methods in the contest of a series of linear systems, where A and $\boldsymbol{b}$ can change significantly. Furthermore, one of the main concerns of this work is to explore its application to representative cases arising from compressible flows in compressible computational fluid dynamics.
First, the Iterative Generalized Minimal RESidual (GMRES) method was introduced, highlighting the importance of the initial guess for iterative algorithms. Then, the focus was moved to the topic of Machine Learning, understanding the two main learning paradigms, offline and online, and defining various neural network types (DNN, CNN, GNN). The concept of the loss function was elucidated, essential in guiding model optimization during training.
Subsequently, the prediction algorithm was presented, showcasing where the inspiration was taken from and explaining its evolution and modifications done for tackling a wide range of simple problems, including the Laplacian equation and the Advection-Diffusion equation. Furthermore, the proposed ML-based algorithm is compared to a classic heuristic used to obtain an initial guess, which consists in recycling the previous solution during an time-iteration scheme. The results obtained are quite promising. The ML predicted initial guesses accelerate the convergence for most of the considered cases. Moreover, the algorithm demonstrated to have a better behaviour with respect to the recycling algorithm for the advection-diffusion equation with an increasing time step.
Finally, the algorithm's application was extended to representative test cases governed by the Navier-Stokes and RANS equations thanks to tests carried out on extracted matrices from the Aghora code. The DG method was introduced, along with the Backward-Euler method for time integration. The algorithm's ability was tested for solving the flow around a cylinder with low Reynolds number, the flow around a NACA0012 airfoil and finally the Taylor-Green vortex test case was studied. The results obtained demonstrate that the developed algorithm leads to shorter times to reach the stopping conditions of the GMRES solver, unfortunately, the training of the network causes the computational time and memory constraints for large systems to be still prohibitive with the current

implementation. Hopefully, advancements of algorithms and GPU hardware could lead to address these issues in the future.

While we have achieved interesting results, this work also opens avenues for future research. Further exploration of hybrid approaches, combining iterative methods and Machine Learning, could lead to even more robust and efficient solvers for fluid dynamics simulations. Additionally, investigating larger-scale problems and exploring new architectures for neural networks may unveil even greater potential for computational efficiency.

On a final note, the findings presented in this thesis contribute to the advancement of Machine Learning techniques to accelerate iterative methods for computational fluid dynamics. The interesting findings in the current work pave the way for continued innovation in the field of fluid dynamics simulations, bringing closer the goal of more accurate, efficient, and scalable computational methods for real-world engineering applications.

Bibliography

[1] P. F. Antonietti, N. Farenga, E. Manuzzi, G. Martinelli, and L. Saverio. Agglomeration of Polygonal Grids using Graph Neural Networks with applications to Multigrid solvers. https://doi.org/10.48550/arXiv.2210.17457, 2022.

[2] A. Bajaj. Understanding Gradient Clipping. https://neptune.ai/blog/understanding-gradient-clipping-and-how-it-can-fix-exploding-gradients-problem, 2023. Accessed: 2023-07-03.

[3] F. Bassi, S. Rebay, G. Mariotti, S. Pedinotti, and M. Savini. A high order accurate discontinuous finite element method for inviscid and viscous turbomachinery flows. In *Turbomachinery - Fluid Dynamics and Thermodynamics, European Conference, 2*, pages 99–108, 1997. ISBN 90-5204-032-X. URL https://www.tib.eu/de/suchen/id/tema%3ATEMAM97071562579.

[4] F. Bassi, A. Crivellini, S. Rebay, and M. Savini. Discontinuous Galerkin solution of the Reynolds-Averaged Navier–Stokes and k - ω turbulence model equations. *Computers & Fluids*, 34:507–540, 05 2005. doi: https://doi.org/10.1016/j.compfluid.2003.08.004.

[5] M. Benzi. Preconditioning Techniques for Large Linear Systems: A Survey. *Journal of Computational Physics*, 182(2):418–477, 2002. ISSN 0021-9991. doi: https://doi.org/10.1006/jcph.2002.7176. URL https://www.sciencedirect.com/science/article/pii/S0021999102971767.

[6] S. Bock, J. Goppold, and M. G. Weiß. An improvement of the convergence proof of the ADAM-Optimizer. *CoRR*, abs/1804.10587, 2018. URL http://arxiv.org/abs/1804.10587.

[7] L. Botti and L. Verzeroli. BR2 discontinuous Galerkin methods for finite hyperelastic deformations. *Journal of Computational Physics*, 463:111303, 2022. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2022.111303. URL https://www.sciencedirect.com/science/article/pii/S0021999122003655.

[8] A. Crivellini and F. Bassi. An implicit matrix-free Discontinuous Galerkin solver for

viscous and turbulent aerodynamic simulations. *Computers & Fluids*, 50(1):81–93, 2011. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2011.06.020. URL https://www.sciencedirect.com/science/article/pii/S0045793011002088.

[9] J.-K. Fang, C.-M. Fong, P. Yang, C.-K. Hung, W.-L. Lu, and C.-W. Chang. AdaGrad Gradient Descent Method for AI Image Management. In *2020 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-Taiwan)*, pages 1–2, 2020. doi: https://doi.org/10.1109/ICCE-Taiwan49838.2020.9258085.

[10] V. Frayssé, L. Giraud, S. Gratton, and J. Langou. Algorithm 842: A Set of GMRES Routines for Real and Complex Arithmetics on High Performance Computers. *ACM Trans. Math. Softw.*, 31(2):228–238, jun 2005. ISSN 0098-3500. doi: 10.1145/1067967.1067970. URL https://doi.org/10.1145/1067967.1067970.

[11] V. Frayssé, L. Giraud, and S. Gratton. Algorithm 881: A Set of Flexible GMRES Routines for Real and Complex Arithmetics on High-Performance Computers. *ACM Transactions on Mathematical Software*, 35:1–12, 2008. doi: https://doi.org/10.1145/1377612.1377617.

[12] G. H. Golub and C. F. van Loan. *Matrix Computations*. JHU Press, fourth edition, 2013. ISBN 1421407949 9781421407944.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. https://doi.org/10.48550/arXiv.1512.03385, 2015.

[14] C. Ick, S. Tamba, Z. Lei, and H. Tang. ConvNet Evolutions, Architectures, Implementation Details and Advantages. https://atcold.github.io/pytorch-Deep-Learning/en/week03/03-2/, 2020. Accessed: 2023-07-13.

[15] I. C. F. Ipsen and C. D. Meyer. The Idea Behind Krylov Methods. *The American Mathematical Monthly*, 105(10):889–899, 1998. doi: https://doi.org/10.1080/00029890.1998.1200498. URL https://doi.org/10.1080/00029890.1998.12004985.

[16] S. Jadon. Introduction to different activation functions for deep learning. *Medium, Augmenting Humanity*, 16, 2018. URL https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092.

[17] M. Jadoui, C. Blondeau, E. Martin, F. Renac, and F. X. Roux. Comparative study of inner–outer Krylov solvers for linear systems in structured and high-order unstructured CFD problems. *Computers and Fluids*, 244:105575, 2022.

[18] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization, 2017.

[19] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Handwritten Digit Recognition with a Back-Propagation Network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pages 396–404. Morgan Kaufmann, 1989. URL https://dl.acm.org/doi/10.5555/109230.109279.

[20] K. Luna, K. Klymko, and J. P. Blaschke. Accelerating GMRES with Deep Learning in Real-Time. https://doi.org/10.48550/arXiv.2103.10975, 2021.

[21] K. Luna, K. Klymko, and J. P. Blaschke. GMRES-Learning, 2021. URL https://github.com/ML4FnP/GMRES-Learning.

[22] E. Martin and F. Renac. Parallel algebraic solvers for high-order Discontinuous Galerkin methods for compressible turbulent flows, 2019. Slides ENUMATH.

[23] ONERA. Le calcul scientifique intensif. https://www.onera.fr/fr/les-moyens-de-calcul-intensifs, 2023. Accessed: 2023-07-03.

[24] ONERA. ONERA: The Frech Aerospace Lab. https://www.onera.fr/en/identity, 2023. Accessed: 2023-08-20.

[25] K. O'Shea and R. Nash. An Introduction to Convolutional Neural Networks. *ArXiv e-prints*, 11 2015. URL https://doi.org/10.48550/arXiv.1511.08458.

[26] P.-O. Persson and J. Peraire. Newton-GMRES Preconditioning for Discontinuous Galerkin Discretizations of the Navier–Stokes Equations. *SIAM*, 30, 01 2008. doi: https://doi.org/10.1137/070692108.

[27] S. B. Pope. *Turbulent Flows*. Cambridge University Press, 2000. doi: https://doi.org/10.1017/CBO9780511840531.

[28] F. Renac. Discrétisation implicite en temps des équations de Navier-Stokes pour une méthode Galerkin Discontinue. Mise en oeuvre dans le code Aghora. RT 6/2013 DMFN. Technical report, ONERA, 2016.

[29] F. Renac, M. de la Llave Plata, E. Martin, J. B. Chapelier, and V. Couaillier. *Aghora: A High-Order DG Solver for Turbulent Flow Simulations*, pages 315–335. Springer International Publishing, Cham, 2015. ISBN 978-3-319-12886-3. doi: https://doi.org/10.1007/978-3-319-12886-3_15. URL https://doi.org/10.1007/978-3-319-12886-3_15.

[30] S. Ruder. An overview of gradient descent optimization algorithms, 09 2016.

[31] N. C. Rumsey. The Spalart-Allmaras Turbulence Model. https://turbmodels.larc.nasa.gov/spalart.html, 2020. Accessed: 2023-07-03.

[32] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003. URL https://www-users.cse.umn.edu/~saad/IterMethBook_2ndEd.pdf.

[33] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986. doi: 10.1137/0907058. URL https://doi.org/10.1137/0907058.

[34] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, 2015. doi: https://doi.org/10.48550/arXiv.1409.1556.

[35] R. Swanson and S. Langer. Comparison of NACA 0012 Laminar Flow Solutions: Structured and Unstructured Grid Methods, 01 2016.

[36] Z. Tang, H. Zhang, and J. Chen. Graph Neural Networks for Selection of Preconditioners and Krylov Solvers, 2022. URL https://openreview.net/forum?id=tMlBpP1I3Bt.

[37] TOP500. TOP500 List - June 2017. https://www.top500.org/lists/top500/list/2017/06/, 2017. Accessed: 2023-07-03.

[38] G. Varoquaux and O. Colliot. Evaluating machine learning models and their diagnostic value. In O. Colliot, editor, *Machine Learning for Brain Disorders*. Springer, June 2023. URL https://hal.science/hal-03682454.

[39] A. Zafar, M. Aamir, N. Mohd Nawi, A. Arshad, S. Riaz, A. Alruban, A. K. Dutta, and S. Almotairi. A Comparison of Pooling Methods for Convolutional Neural Networks. *Applied Sciences*, 12(17), 2022. ISSN 2076-3417. doi: 10.3390/app12178643. URL https://www.mdpi.com/2076-3417/12/17/8643.

[40] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph Neural Networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. ISSN 2666-6510. doi: https://doi.org/10.1016/j.aiopen.2021.01.001. URL https://www.sciencedirect.com/science/article/pii/S2666651021000012.

A | Typical Result Plots

Contents

This chapter, included in the appendix, delves into the visual representation of the research findings and introduces the moving average designed to enhance the effectiveness of presenting the results, therefore aiding in comprehending complex trends but also providing a clearer understanding of the data patterns. Through this chapter, the commitment to ensuring clarity and precision in conveying the research results is demonstrated.

A.1. Moving Average

In this thesis, in order to present clearer result patterns, with respect to the scattered and messy data obtained by the simulations, a moving average is defined. The definition is extracted and made more general from [20, 21]. Listing A.1 details the definition.

```
def moving_average(a, n, N):
    """
    Args:
        a (array): Array over which the average is to be computed.
        n (int): index of the last element of a.
        N (int): Interval over which the average is computed.
    """

    if n == 0:
        return 0
    elif n < N:
        Window=int(math.ceil(0.5*n))
    else:
        Window=N
    return np.sum(a[-Window-1:-1])/Window
```

Listing A.1: Definition of Moving Average found in `src_dir.util.py` at [21].

Listing A.2 shows how to use the function to compute a moving average.

```
# Compute moving average of array
```

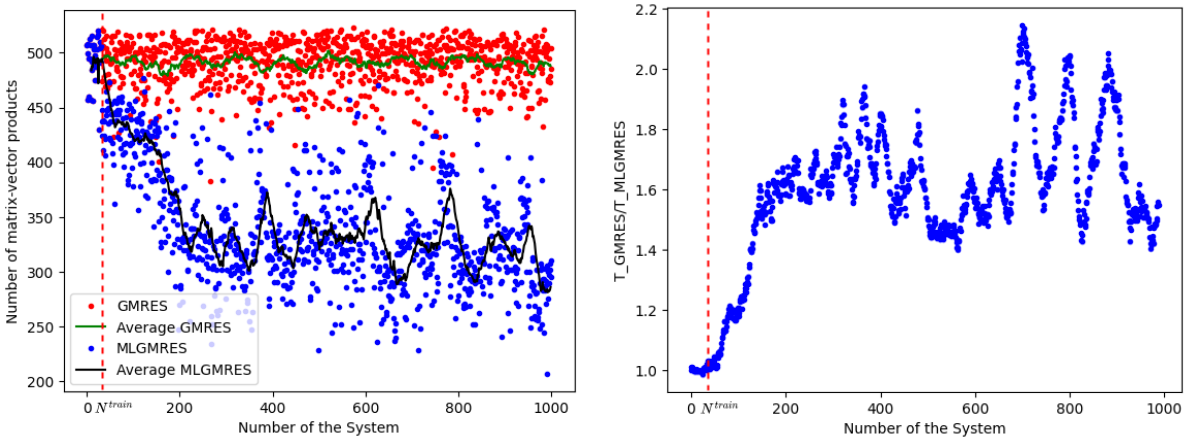
```
array_AVG = np.zeros(len(array))
N = 25

for j in range(0, len(array)):
    array_AVG[j] = moving_average(np.asarray(array[:j]), j, N)
```

Listing A.2: Call of Moving Average.

A.2. Typical Plots using the Moving Average

The results are presented using figures obtained thanks to the `Matplotlib` Python library. The figures represent important metrics to extract the efficiency of the prediction algorithm, such as the rate of convergence, the normalized residual of each system at the end of the first restart, the total number of matrix-vector products required to reach convergence, in regards with defined user parameters, and the speed-up in the resolution of a system with respect to the classic GMRES method. In order to better observe the trend of the Online Learning in the plots with respect to the number of system the moving average is also shown, alongside the complete data, as one can observe in Figure A.1a. Furthermore, the figures depicting the speed-up of the new algorithm show only the computed moving average, such as in Figure A.1b. In this example the two figures also present a dashed red line, representing the position of the last system before the first training is performed by the model. This is to highlight the difference between before of the training and after it. In this case $N^{train} = 32$.



(a) Number of matrix-vector products to reach convergence w.r.t. the number of systems.

(b) Iteration speed-up.

Figure A.1: Plots with moving averages.

B | Time Analysis with respect to the Architecture and the Dimension

Contents

This chapter, included in the appendix, conducts a comprehensive comparison of CPU and GPU performance, both with and without the integration of Machine Learning techniques. Execution times for the implemented algorithms on CPU and GPU platforms are evaluated, revealing potential speed-ups achieved through parallelization and hardware acceleration in a transparent manner. Additionally, the impact of Machine Learning on algorithm performance is examined. Furthermore, the chapter analyzes the speed-up trends with respect to the system dimension of the algebraic problem, providing insights into the algorithm's scalability and efficiency.

B.1. Comparing Times on CPU and GPU with and without ML

Central Processing Unit (CPU) and Graphics Processing Unit (GPU) are both types of processors, but they have different architectures and are designed to handle different types of tasks.
A CPU is the primary processor in a computer, responsible for executing instructions and performing calculations for the operating system and applications. It has a few powerful

processing cores that can handle a wide range of tasks, including running software, managing input/output operations, and communicating with other components.
On the other hand, a GPU is a specialized processor that is designed to handle graphics-intensive tasks, such as rendering images, videos, and animations. It has thousands of smaller processing cores that can work in parallel to perform complex mathematical calculations and manipulate large amounts of data quickly.

B.1.1. Computing Environment

The tests have all been run on GPU rtx6000.

File	**CPU** (s)	**GPU** (s)
Spiro Torch DEMO	699.60	44.076

Table B.1: Time to solve the code `autoencoder_mnist_spiro.py` using CPU and GPU.

B.1.2. Machine Learning Training on the Laplace Equation

Solver	**CPU** (s)	**pip GPU** (s)	**spack GPU** (s)
GMRES Demo	1.1663	1.0146	1.0234
MLGMRES Demo	0.6207	0.2411	0.3167
GMRES	0.1271	0.1174	0.1148
MLGMRES	0.0484	0.0425	0.0394

Table B.2: Times to solve a linear system with the GMRES algorithm compared with the ML strategy using DNNs and the times of the Demo by [20] [21], run on CPU and GPU, with $n = 20$, $N_S = 1000$ and an initial set of dimension 32.

Solver	**CPU mean** (s)	**pip GPU mean** (s)	**spack GPU mean** (s)
GMRES Demo	1.2165	0.8935	0.8835
MLGMRES Demo	0.7747	0.4906	0.5472
GMRES	0.1225	0.1134	0.1155
MLGMRES	0.0855	0.0802	0.0791

Table B.3: Average times to solve a linear system with the GMRES algorithm compared with the ML strategy using DNNs and the times of the Demo by [20] [21], run on CPU and GPU, with $n = 20$, $N_S = 1000$ and an initial set of dimension 32.

Solver	CPU (s)	GPU (s)	CPU mean (s)	GPU mean (s)
GMRES	50.196	51.782	46.242	49.760
MLGMRES	36.406	40.980	46.092	49.163

Table B.4: Times to solve a linear system (maximum and average values) with the GMRES algorithm compared with the ML strategy using DNNs, run on CPU and GPU, with $n = 200$, $N_S = 100$ and an initial set of dimension 16.

B.1.3. Using the Python Profiler

The following tests have been performed with Dense Neural Networks to observe the full speed-up of the GPU with respect to the CPU.

Equation and Dimension	Total CPU (s)	Total GPU (s)
Laplace Equation $n = 20$	62.5	60.6
Laplace Equation $n = 100$	180	140
Advection Diffusion Equation $n = 100$	263	149

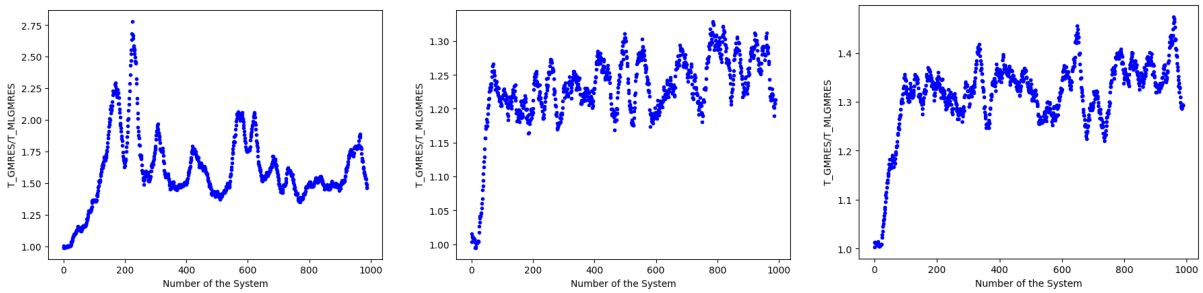
Table B.5: Full time of the Python decorator, without the time of solving the GMRES system.

Equation and Dimension	Training CPU (s)	Training GPU (s)
Laplace Equation $n = 20$	59.6	58.1
Laplace Equation $n = 100$	118	81.6
Advection Diffusion Equation $n = 100$	248	134

Table B.6: Time of the training of the Python decorator.

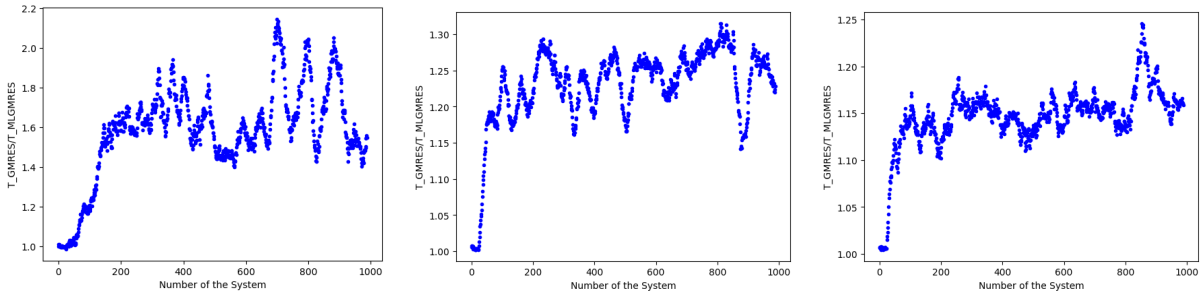
B.2. Comparing Speed-ups with respect to the Dimension

In order to study the generalization properties of the solver one can apply it to different sizes of the system, using also different models of Neural Networks, and observe the different speed-ups which are obtained.
These simulations were performed on the Laplace equation testcase.



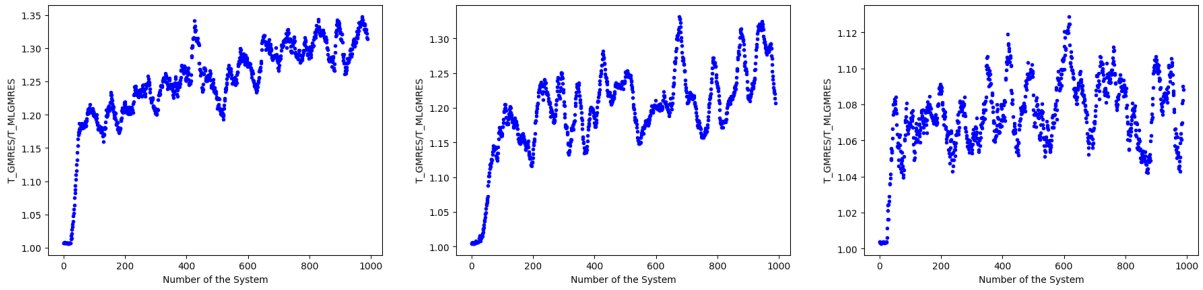
(a) Dense NN (b) Convolutional NN (c) ResNet

Figure B.1: Plots of the speed-up with systems 10×10.



(a) Dense NN (b) Convolutional NN (c) ResNet

Figure B.2: Plots of the speed-up with systems 20×20.



(a) Dense NN (b) Convolutional NN (c) ResNet

Figure B.3: Plots of the speed-up with systems 40×40.

C | Implemented Code

Contents

This chapter, included in the appendix, provides implementation details of the predictor code and the neural networks utilized in the thesis. It offers insights into the structure and key components of the predictor code. Additionally, the chapter explores the architectures and configurations of the implemented neural networks. By providing a comprehensive overview of the predictor code and neural networks, this chapter offers valuable information for readers seeking a deeper understanding of the technical aspects underpinning the thesis's methodology and computations.

C.1. Parts of the Code of the Prediction Algorithm

```
class EarlyStopping:
    """Early stops the training if validation loss doesn't improve after a given patience
    ."""
    def __init__(self, patience=7, delta=0):
        """
        Args:
            patience (int): How long to wait after last time validation loss improved.
                            Default: 7
            delta (float): Minimum change in the monitored quantity to qualify as an
    improvement.
                            Default: 0
        """
        self.patience = patience
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.delta = delta
```

```

    def __call__(self, val_loss, model):

        score = -val_loss

        if self.best_score is None:
            self.best_score = score
        elif score < self.best_score + self.delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.counter = 0
```

Listing C.1: Definition of Early Stopping.

```
    def predict(self, x):
        # inputs need to be [[x_1, x_2, ...]] as floats
        # outputs need to be numpy (non-grad => detach)
        # outputs need to be [y_1, y_2, ...]
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        a1 = torch.from_numpy(x).unsqueeze_(0).float().to(device)
        a2 = np.squeeze(
                self.model.forward(
                    a1).detach().cpu().numpy()
            )
        return a2
```

Listing C.2: Prediction Code.

```
def cnn_predictorOnline_timed(trainer):

    def my_decorator(func):
        name = func.__name__

        @wraps(func)
        def speedup_wrapper(*args, **kwargs):

            # Get problem data:

            A, b, x0, *etc = args

            # Use predictor to generate initial guess:
            b_norm, b_Norm_max = prob_norm(b)
            if trainer.A_CHANGING:
                if isinstance(A, np.matrix) or isinstance(A, scipy.sparse._csc.csc_matrix
    ) or isinstance(A, np.ndarray) or isinstance(A, scipy.sparse._csr.csr_matrix):
                    Ab_norm, Ab_Norm_max = prob_norm(A @ b)
                    pred_x0 = trainer.predict(A, A @ b, x0, Ab_norm*Ab_Norm_max)
                else:
                    Ab_norm, Ab_Norm_max = prob_norm(A(b))
                    pred_x0 = trainer.predict(A, A(b), x0, Ab_norm*Ab_Norm_max)
            else:
                pred_x0 = trainer.predict(A, b, x0, b_norm*b_Norm_max)

```

```
            # Run function (and time it)
            tic = time.perf_counter()
            out = func(A, b, pred_x0, *etc)
            toc = time.perf_counter()

            # Check on the number of outputs of the function
            out1 = False
            if isinstance(out, tuple):
                target, *other = out
            else:
                target = out
                out1 = True

            # Pick out solution from output list
            sol = target[-1]

            # Write diagnostic data (error and time-to solution) to list
            IterTime = (toc-tic)
            trainer.write_diagnostics(IterTime, A, target, b)

            # Add problem to the training set
            trainer.add_single(sol, b, b_norm*b_Norm_max)

            if out1:
                return target
            else:
                return target, *other

        speedup_wrapper.__signature__ = signature(func)

        return speedup_wrapper

    return my_decorator
```

Listing C.3: Decorator Code.

C.2. Definition of the Neural Networks

C.2.1. Definition of the Dense Neural Network

```
class DenseNN(torch.nn.Module):


    def __init__(self, D_in, D_out):

        super(DenseNN, self).__init__()
        device        = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        self.n_blocks = D_in // 20 - 1
        self.dimHid       = D_in
        self.Lin_in   = torch.nn.Linear(D_in, self.dimHid, bias=False)
        self.Lin_out  = torch.nn.Linear(self.dimHid, D_out, bias=False)
        self.act      = nn.ReLU()
        self.blocks   = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])
```

```
14
15      def make_block(self):
16          return nn.Sequential(
17              nn.Linear(self.dimHid, self.dimHid, bias=False),
18              self.act
19          )
20
21
22      def forward(self, x):
23
24          x = self.act(self.Lin_in(x))
25
26          for block in self.blocks:
27              x = block(x)
28
29          x = self.Lin_out(x)
30
31          return x
```

Listing C.4: Dense Neural Network.

C.2.2. Definition of the Convolutional Neural Network

```
1  class CNN(nn.Module):
2      def __init__(self, D_in, D_out):
3          super(CNN, self).__init__()
4          device      = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
5
6          self.D_in = D_in
7
8          self.convIn = nn.Conv2d(
9                        in_channels=1,
10                       out_channels=16,
11                       kernel_size=(1,3),
12                       stride=1,
13                       padding=(0,1),
14                       dilation=1
15                      )
16
17         self.convGrow = nn.Conv2d(
18                       in_channels=16,
19                       out_channels=64,
20                       kernel_size=(1,3),
21                       stride=1,
22                       padding=(0,1),
23                       dilation=1
24                      )
25
26         self.convSame = nn.Conv2d(
27                       in_channels=64,
28                       out_channels=64,
29                       kernel_size=(1,3),
30                       stride=1,
31                       padding=(0,1),
32                       dilation=1
```

```
                    )

        self.convShrink = nn.Conv2d(
                     in_channels=64,
                     out_channels=16,
                     kernel_size=(1,3),
                     stride=1,
                     padding=(0,1),
                     dilation=1
                    )

        self.convOut = nn.Conv2d(
                     in_channels=16,
                     out_channels=1,
                     kernel_size=(1,3),
                     stride=1,
                     padding=(0,1),
                     dilation=1
                    )

        self.kernel_size = max(self.D_in // 40, 2)
        self.n_blocks = max(self.D_in // 20, 1)
        #self.n_blocks = max(self.D_in // 100, 1)
        #self.n_blocks = max(self.D_in // 500, 1)
        self.AVG_big = torch.nn.AvgPool1d(self.kernel_size, stride=self.kernel_size)
        self.AVG = torch.nn.AvgPool1d(2, stride=2)
        self.Upsample = torch.nn.Upsample(mode='linear', size=(D_in,))

        self.act = nn.ELU()

        self.blocks1 = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])
        self.blocks2 = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])
        self.blocks3 = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])
        self.blocks4 = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])

    def make_block(self):
        return nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(1,3), stride=1,
    padding=(0,1), dilation=1),
            self.act
        )

    def forward(self, x):

        def AVGPOOL(x, AVG):
            x = x.squeeze(2) # shape (batch_size, 1, dim)
            x = AVG(x)
            x = x.unsqueeze(2) # shape (batch_size, 1, 1, dim)
            return x

        def UPSAMPLE(x):
            x = x.squeeze(2) # shape (batch_size, 1, dim)
            x = self.Upsample(x)
            x = x.unsqueeze(2) # shape (batch_size, 1, 1, dim)
            return x

```

```
        x1 = x.unsqueeze(1).unsqueeze(1) # shape (batch_size, 1, 1, dim)

        x1 = self.act(self.convIn(x1))

        #z2 = AVGPOOL(x1, self.AVG_big)
        z2 = AVGPOOL(x1, self.AVG)

        x1 = self.act(self.convGrow(x1))
        z2 = self.act(self.convGrow(z2))

        x1 = self.act(self.convSame(x1))
        z2 = self.act(self.convSame(z2))

        z3 = AVGPOOL(z2, self.AVG)

        x1 = self.act(self.convSame(x1))
        z2 = self.act(self.convSame(z2))
        z3 = self.act(self.convSame(z3))

        z4 = AVGPOOL(z3, self.AVG)

        for block1, block2, block3, block4 in zip(self.blocks1, self.blocks2, self.
    blocks3, self.blocks4):

            x1 = block1(x1)
            z2 = block2(z2)
            z3 = block3(z3)
            z4 = block4(z4)

        x1 = self.act(self.convShrink(x1))
        z2 = self.act(self.convShrink(z2))
        z3 = self.act(self.convShrink(z3))
        z4 = self.act(self.convShrink(z4))

        x2 = UPSAMPLE(z2)
        x3 = UPSAMPLE(z3)
        x4 = UPSAMPLE(z4)

        x = x1 + x2 + x3 + x4

        x = self.convOut(x)

        return x.squeeze(1).squeeze(1) # shape (batch_size, dim)
```

Listing C.5: Convolutional Neural Network.

C.2.3. Definition of the Graph Neural Network

```
class GNN(nn.Module):
    def __init__(self, D_in, D_out, num_features, nnz):
        super(GNN, self).__init__()
        device      = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

        self.D_in = D_in
        self.D_out = D_out
```

```
        self.num_features = num_features
        self.nnz = nnz

        # CNN

        self.convIn = nn.Conv2d(
                     in_channels=1,
                     out_channels=16,
                     kernel_size=(1,3),
                     stride=1,
                     padding=(0,1),
                     dilation=1
                    )

        self.convGrow = nn.Conv2d(
                     in_channels=16,
                     out_channels=64,
                     kernel_size=(1,3),
                     stride=1,
                     padding=(0,1),
                     dilation=1
                    )

        self.convSame = nn.Conv2d(
                     in_channels=64,
                     out_channels=64,
                     kernel_size=(1,3),
                     stride=1,
                     padding=(0,1),
                     dilation=1
                    )

        self.convShrink = nn.Conv2d(
                     in_channels=64,
                     out_channels=16,
                     kernel_size=(1,3),
                     stride=1,
                     padding=(0,1),
                     dilation=1
                    )

        self.convOut = nn.Conv2d(
                     in_channels=16,
                     out_channels=1,
                     kernel_size=(1,3),
                     stride=1,
                     padding=(0,1),
                     dilation=1
                    )

        self.kernel_size = max(self.D_in // 40, 2)
        self.n_blocks = max(self.D_in // 500, 1)
        self.AVG_big = torch.nn.AvgPool1d(self.kernel_size, stride=self.kernel_size)
        self.AVG = torch.nn.AvgPool1d(2, stride=2)
        self.Upsample = torch.nn.Upsample(mode='linear', size=(D_in,))

```

```
64          self.act = nn.ELU()
65
66          self.blocks1 = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])
67          self.blocks2 = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])
68          self.blocks3 = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])
69          self.blocks4 = nn.ModuleList([self.make_block() for _ in range(self.n_blocks)])
70
71          # GNN
72
73          self.hidden_units = 8
74
75          self.convGNN1 = SAGEConv(self.num_features, self.hidden_units, aggr='mean',
    node_dim=1)
76          self.convGNN2 = SAGEConv(self.hidden_units, 1, aggr='mean', node_dim=1)
77
78          # Mixing
79
80          self.linMIX1 = nn.Linear(self.D_out * 2, self.D_out, bias=False)
81
82          self.linMIX2 = nn.Linear(self.D_out, self.D_out, bias=False)
83
84      def make_block(self):
85          return nn.Sequential(
86              nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(1,3), stride=1,
    padding=(0,1), dilation=1),
87              self.act
88          )
89
90      def forward(self, x, feats, edge_index):
91
92          # CNN
93
94          def AVGPOOL(x, AVG):
95              x = x.squeeze(2) # shape (batch_size, 1, dim)
96              x = AVG(x)
97              x = x.unsqueeze(2) # shape (batch_size, 1, 1, dim)
98              return x
99
100         def UPSAMPLE(x):
101             x = x.squeeze(2) # shape (batch_size, 1, dim)
102             x = self.Upsample(x)
103             x = x.unsqueeze(2) # shape (batch_size, 1, 1, dim)
104             return x
105
106         x1 = x.unsqueeze(1).unsqueeze(1) # shape (batch_size, 1, 1, dim)
107
108         x1 = self.act(self.convIn(x1))
109
110         #z2 = AVGPOOL(x1, self.AVG_big)
111         z2 = AVGPOOL(x1, self.AVG)
112
113         x1 = self.act(self.convGrow(x1))
114         z2 = self.act(self.convGrow(z2))
115
116         x1 = self.act(self.convSame(x1))
117         z2 = self.act(self.convSame(z2))
```

```
118
119         z3 = AVGPOOL(z2, self.AVG)
120
121         x1 = self.act(self.convSame(x1))
122         z2 = self.act(self.convSame(z2))
123         z3 = self.act(self.convSame(z3))
124
125         z4 = AVGPOOL(z3, self.AVG)
126
127         for block1, block2, block3, block4 in zip(self.blocks1, self.blocks2, self.
    blocks3, self.blocks4):
128
129             x1 = block1(x1)
130             z2 = block2(z2)
131             z3 = block3(z3)
132             z4 = block4(z4)
133
134         x1 = self.act(self.convShrink(x1))
135         z2 = self.act(self.convShrink(z2))
136         z3 = self.act(self.convShrink(z3))
137         z4 = self.act(self.convShrink(z4))
138
139         x2 = UPSAMPLE(z2)
140         x3 = UPSAMPLE(z3)
141         x4 = UPSAMPLE(z4)
142
143         x = x1 + x2 + x3 + x4
144
145         x = self.convOut(x)
146
147         x = x.squeeze(1).squeeze(1)
148
149         # GNN
150
151         feats = self.act(self.convGNN1(feats, edge_index[0]))
152
153         feats = self.convGNN2(feats, edge_index[0])
154
155         feats = feats.squeeze(2)
156
157         # Mixing
158
159         x = torch.cat([x,feats],1)
160
161         x = self.linMIX1(x.float())
162
163         x = self.act(x)
164
165         x = self.linMIX2(x)
166
167         return x # shape (batch_size, dim)
```

Listing C.6: Graph Neural Network.

D | Calculation resources at ONERA

This chapter, included in the appendix, offers a comprehensive, but brief, description of the calculation resources utilized at ONERA. It provides detailed insights into the computational infrastructure, hardware specifications, and software environment employed for conducting the numerical simulations presented in the thesis. By outlining the calculation resources used at ONERA, this chapter offers readers valuable context and transparency regarding the computational environment, ensuring reproducibility and facilitating further research in the field of fluid dynamics simulations.

(a) SATOR. (b) SPIRO.

Figure D.1: ONERA's supercomputers.

ONERA has high-quality intensive computing resources, allowing the development of massive code. parallel activities, for research or production purposes (for contracts with industrial partners For example). Two supercomputers are thus available [23], each having a targeted objective:

- the SATOR computer is dedicated to intensive scientific production with LINPACK performance of 579.2 TFlops/s (which earned it the 343rd place in the Top500 in June 2017 [37]), enabled thanks to its 43600 processing cores (17360 Broadwell, 7040

Skylake and 19200 Cascade Lake) and its memory RAM of 182.5 TB;

- the SPIRO calculator is dedicated to development and exploration activities for new computing architectures. With a more heterogeneous architecture (Intel, AMD, GPU and ARM), SPIRO has 225 computing nodes equipped with Intel Broadwell (BRW) processors and 24 equipped with Intel Skylake (SKL). The interconnection network is different depending on the partitions and types of affected processors, with a Gigabit Ethernet (1 Gbit/s) network or an Intel Omnipath network (OPA) at very high speed (100 Gbit/s).

List of Figures

List of Tables

Listings

List of Symbols

Symbol	Description
A	Matrix of a linear system
a_{ij}	Element in position (i, j) of A
$\lvert \cdot \rvert$	Absolute value
$\boldsymbol{b}$	RHS of a linear system
C_t	New batch of data
dd	Diaginal dominance
Δt	Time step of a time discretization
Δx	Spatial step of a space discretization
E	Total specific energy
$\boldsymbol{e_j}$	j-th vector of the canonical basis
$E_\kappa(\boldsymbol{x_0})$	Residual at the end of the first restart of GMRES starting form $\boldsymbol{x_0}$
f_r	Retrain frequency
$\bar{H}_m$	Hessenberg matrix
$\mathcal{K}_m(A, \boldsymbol{r_0})$	Krylov space generate from A and $\boldsymbol{r_0}$
L	Loss function
m	Size of the Krylov subspace
$M_p(\cdot)$	Moving average of p elements
n	Size of a linear system
N_b	Number of elements in the batch
$N(\boldsymbol{b})$	Neural Network N applied to $\boldsymbol{b}$
N_{max}	Maximum number of iterations for the GMRES algorithm
N_S	Total number of systems of a simulation
$\mathbb{N}^*$	Set of positive numbers $(\mathbb{N} \setminus \{0\})$
N_{train}	Size of the initial set
$\lVert \cdot \rVert$	Eulerian norm

$\boldsymbol{r}$	Residual
$R(\boldsymbol{U}^n)$	Residual of the semi-discretized equation
ρ	Density
T	Final time of a simulation
$TOS(\boldsymbol{x_0})$	Time to reach convergence with GMRES starting from $\boldsymbol{x_0}$
$\boldsymbol{v}$	Velocity vector
$\boldsymbol{x_0}$	Initial guess of the GMRES algorithm
$\boldsymbol{x_m}$	Solution of the GMRES algorithm
X_t	Training set
W	Trainable weights
$z(\cdot)$	Activation function
Ω	Domain

Acknowledgements

I would like to express my heartfelt gratitude towards Emeric Martin, Jorge Nuñez and Florent Renac for the opportunity to conduct my internship at ONERA and for all the help you gave me during the work. Your guidance, support and knowledge-sharing throughout the internship have been crucial to me. Working under your mentorship has been an enriching experience, which I believe will have a lasting impact on my personal and professional growth. I am deeply grateful to IFPEN for their invaluable contributions through insightful discussions and thoughtful recommendations. My journey would not have been as enlightening without their engagement. Moreover, I wish to extend my heartfelt appreciation to Dr. Kevin Luna and Dr. Johannes Blaschke for providing me the foundations for my work. Their unwavering availability to address my queries has been a testament to their commitment to research.

Desidero esprimere la mia profonda gratitudine al Professor Nicola Parolini per l'incessante supporto che mi ha offerto durante l'intero processo di redazione di questa tesi. La sua costante disponibilità a confrontarsi con me attraverso costruttivi colloqui e i suoi preziosi consigli sono stati fondamentali per la stesura di questo lavoro. In aggiunta, desidero estendere il mio sentito ringraziamento al Professor Corrado Maurini, il quale ha svolto un ruolo assolutamente fondamentale nel corso del mio percorso di doppia laurea a Parigi.