

UNIVERSITÀ
DI PAVIA



Università
della
Svizzera
italiana

UNIVERSITÀ DEGLI STUDI DI PAVIA
UNIVERSITÀ DELLA SVIZZERA ITALIANA

JOINT PHD PROGRAM IN COMPUTATIONAL MATHEMATICS AND DECISION SCIENCES
XXXIV CYCLE

A Neural Network approach for the generation of Transfer Operators in Multilevel Solvers

Advisor:
Prof. Rolf KRAUSE

PhD Dissertation of:
Claudio TOMASI
matr. 470004

Academic year 2020-2021

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Claudio Tomasi
Pavia, 12 January 2022

Abstract

In this thesis, we investigate the combination of Multigrid methods and Neural Networks, starting from Finite Element discretizations of Partial Differential Equations. Multigrid methods are among the fastest numerical methods employed to solve elliptic equations. They use different levels of approximation in a multilevel hierarchy to compute the solution. The keypoint is to appropriately define transfer operators to transfer information between these different levels. These operators are crucial for fast convergence of Multigrid, but they are generally unknown.

Here, we propose Neural Network models for learning transfer operators, and we build a multilevel hierarchy based on the output of the predictive model. After a preliminary study in one-dimensional scenarios, we define our training set by extracting information from geometry and operator matrices. We take the features from the mass matrix and the target from the L^2 -projection. Then, we customize the model loss function in order to include knowledge about the transfer operators: in this way, our network solves a constrained problem, forcing some domain properties on the predictions. The application of this model in a Multigrid context results in good convergence, motivating the passage to two-dimensional problems. Given the increased complexity of the data, we first investigate the accuracy of the predictions, by testing it with different network architectures and with different combinations of parameters. We focus on the study of convergence, where we compare our strategy with existing Multigrid methods. More specifically, we consider the Semi-Geometric Multigrid and the Algebraic Multigrid.

A big issue that needs to be faced is the constraint given by feedforward Neural Networks of working only with fixed input and output dimensions. Therefore, we implement and compare different solution to address this problem, by extending the node patches when the neighborhood of a node has less nodes than expected, and by decomposing our feature extraction when the node patch is bigger than the network input. In order to validate this procedure in more general settings, we test our method using several geometries, considering structured

and unstructured grids, but without the need of using a specific implementation for each grid.

In the last part of this work, we focus on problems with variable diffusion coefficients, where stiffness information is added in the training process. This strategy allows to achieve faster convergence than using transfer operators based only on geometric data.

Future discussion should be devoted to the extension of this Neural Network approach to three-dimensional scenarios and to the construction of grid operators for an automatic definition of multilevel solvers, allowing a portable solution in scientific computing.

Contents

Contents	iv
List of Figures	vii
List of Tables	ix
Introduction	1
1 Multigrid	5
1.1 Model Problem and FE Discretization	6
1.1.1 Weak Formulation	6
1.1.2 Discrete Formulation	7
1.2 Mesh	7
1.2.1 Grid Generation	8
1.2.2 Refinement	9
1.3 Multigrid Idea	9
1.3.1 Error Smoothing procedure	10
1.3.2 Coarse Grid Correction	12
1.4 The Multigrid Algorithm	13
1.4.1 Two-grid Method	14
1.4.2 Multigrid Components	14
1.4.3 Multigrid method	15
1.4.4 Multigrid Cycles	17
1.5 Transfer operators	19
1.5.1 Geometric Transfer Operator	19
1.5.2 AMG Transfer Operator	20
1.5.3 L^2 -Projection	22
1.6 Convergence and Efficiency	26

2 Deep Learning	29
2.1 Machine Learning Methods	30
2.1.1 Learning Algorithms	30
2.1.2 Capacity, Overfitting, Underfitting	32
2.1.3 Preprocessing	33
2.1.4 Hyperparameters and Validation Set	33
2.1.5 Stochastic Gradient	34
2.2 Deep Feedforward Networks	35
2.2.1 Architecture	36
2.2.2 Backpropagation	38
2.2.3 Regularization Methods	39
3 Neural Multigrid	43
3.1 Training Set	44
3.1.1 Class of examples	46
3.1.2 One-dimensional Records	47
3.1.3 Two-dimensional Records	48
3.1.4 Methodology Details	50
3.2 Model Training	54
3.2.1 Regularization	54
3.2.2 Model Details	55
3.2.3 Hyperparameters Tuning	56
3.3 NMG Implementation	56
3.3.1 Mesh Extension	57
3.3.2 Virtual Extension	58
3.3.3 Adding information from A_h	60
4 Numerical Experiments	63
4.1 Technical Specifications	64
4.1.1 MacBook Pro	64
4.1.2 Cluster	64
4.2 Libraries and Coding	65
4.3 Numerical Results	65
4.3.1 1D	66
4.3.2 2D	70
Conclusion	83
Bibliography	85

Figures

1.1 Example of conforming and nonconforming grids	8
1.2 Refinement procedures applied on a mesh.	10
1.3 Smoothing effect of the Gauss–Seidel method.	11
1.4 Error on fine mesh projected onto a coarse mesh.	12
1.5 Different MG cycles, increasing the number of levels.	17
1.6 Solution of a test problem on different levels of a multilevel hierarchy	18
1.7 Classic coarsening procedure employed in AMG.	23
1.8 Intersection between 2D elements	25
2.1 Example of Neural Network.	36
2.2 Example of perceptron.	37
2.3 Early Stopping	40
3.1 Example of one-dimensional mesh	44
3.2 Example of two-dimensional mesh	45
3.3 Example of 2D mesh used to extract training set records.	48
3.4 2D meshes considering three different classes of records	50
3.5 Flow chart: training set creation	51
3.6 Overlapping between 1D basis functions.	53
3.7 Result of the mesh extension procedure.	58
3.8 Flow chart: Neural MG	61
4.1 Heatmap of 1D model comparison	66
4.2 Prediction accuracy 1D NN	67
4.3 Convergence of NMG in 1D increasing degrees of freedom	68
4.4 Comparison between NMG and SGMG 1D	69
4.5 Training set generation time	70
4.6 Heatmap of 2D model.	71
4.7 Prediction accuracy 2D NN comparison	72

4.8 Heatmap of 2D model after hyperparameter tuning.	74
4.9 Convergence of NMG in 2D increasing degrees of freedom.	75
4.10 Comparison between NMG and SGMG in 2D - Structured Grid . . .	76
4.11 Structured square mesh.	77
4.12 Comparison between MG methods and NMG, solving a problem with a variable diffusion coefficient	78
4.13 Unstructured mesh of a circle.	79
4.14 Comparison between NMG and SGMG - Unstructured Grid: Circle	79
4.15 Unstructured mesh of a circle with circular holes.	80
4.16 Comparison between NMG and SGMG - Unstructured Grid: Circle with holes.	80
4.17 Unstructured mesh of a circle with a hole - Adaptive refinement. .	81
4.18 Comparison between NMG and SGMG - Unstructured Grid: Circle with a hole, adaptive refinement	81

Tables

4.1	Convergence comparison of NMG against SGMG using ten grids	
	on a 1D problem	70
4.2	Hyperparameter Tuning	73
4.3	Numerical experiments summary	82

Introduction

The discretization of partial differential equations (PDEs) gives rise to large systems of equations. When reaching a three-dimensional scenario, more than one million of unknowns or degrees of freedom (dofs) are not unusual. Applying direct methods for solving systems of this size, results in prohibitively expensive computations, other than the need of very large amount of storage. Therefore, iterative methods are extensively used in treating numerically PDEs. A big disadvantage of this kind of methods is that the amount of work does not remain proportional to the number of unknowns: the time needed to solve a problem grows more rapidly than the size of the problem.

Multigrid (MG) methods overcome this barrier, being also among the most successful strategies for the solution of large system of equations arising from discretized elliptic equations. The works of Brandt [13] and Hackbusch [37, 39] can be considered the milestones forming the historical breakdown, giving rise to the MG theory. Today MG is used in a wide range of fields where PDEs are solved numerically, providing a robust convergence theory, in which a mesh size independent error decreasing is given [11, 93].

The main idea behind the Multigrid method is to combine different level of approximation into a multilevel hierarchy to compute the solution. The algorithm operates as an iterative procedure: in each level, we reduce the error using a smoother and then we move to the lower level. Once the lowest level is reached we move upward, bringing to the upper levels the corrections provided by the lower ones. More rigorous definitions and analysis of the MG procedure are well treated in [15, 78, 86, 90]. In order to transfer information between the different levels, we need to use the so-called transfer operators. Therefore, a crucial point to reach fast convergence is a correct definition of these operators, but they are generally problem-dependent. Unless the multilevel hierarchy consists of nested meshes, the computation of the transfer operators is very expensive, and domain knowledge is always required. Strategies to define and compute these operators in non-nested meshes hierarchies are studied in [25, 26, 85] and [51], where the latter gives a focus on a parallel implementation to obtain a

speed up in the computations. On the other hands, there are strategies that put their attention on the algebraic problem, considering only the system of equations independently of the underlining meshes. These procedures are known as Algebraic Multigrid (AMG) methods. For a general introduction and implementation refer to [20, 72, 82, 83]. These methods have a wide application, but they do not hold the optimal convergence property of the classic MG.

The ever-increasing application of Machine Learning (ML) as support for methods in scientific computing, makes it a natural solution to be employed in the definition of transfer operators, reducing the costs of their construction. ML methods, and specifically Deep Learning, are models that aim to learn from data. Amongst the large amount of references in the literature on this topic, we report [62], which gives a simple introduction and implementation of NNs. For more mathematical details and more complicated model architectures one can refer to [8, 22, 34, 45, 60]. Starting from shallow Neural Networks (NNs), useful to provide predictions related to very simple tasks, nowadays deep learning is applied in a wide range of scenarios, e.g., image classification, speech recognition, drug discovery, and so on. For a complete overview on the evolution of deep learning models refer to [77]. These systems consist of multiple connected layers and find complex structures in large datasets, during their optimization.

Therefore, employing these models to predict transfer operators can provide a good speed up in terms of computational time, removing the need of domain knowledge. Different approaches tried to move in this direction. In [36], Greenfield et al. propose a NN model between PDEs and the operators, specifically the prolongation matrices, for a broad class of 2D diffusion problems. This solution presented an improved convergence when tested against the widely used Black Box MG [21, 23], which select operator-dependent prolongation operators. Another approach is presented in the work of Katrutsa et al. [47]: here the transfer operators, both restriction and prolongation, are optimized while minimizing the spectral radius of the iteration matrix of a given problem, which depends on those two matrices. As an alternative, the method proposed by Luz et al. [59] focus on learning the AMG transfer operators: they employ Graph NNs as learning models (see, e.g., [76]) to learn optimal sparse operators, given in input classes of sparse symmetric positive definite (and semi-definite) matrices. Their aim is to produce a network that solve any linear system of equations with a matrix drawn from that class. Last but not least, the work of He and Xu [42] develops a unified model, that recovers some convolutional NNs [5, 65] for image classification and MG methods for solving discretized partial differential equations (PDEs), based on connections between the two kind of methodologies.

In this work, we propose a methodology based on deep NNs to define transfer operators based on the concept of L^2 -projection, a natural way to prolongate information from a coarse level to a finer mesh. Our strategy is to start from a simple hierarchy consisting of two levels, a fine and a coarse mesh and focus on the geometry. The information arising from the mesh, after a Finite Element (FE) discretization, composes the inputs of our training set. In details we take these data from the mass matrix assembled on the fine mesh. Then we compute the correct transfer operator overlapping fine and coarse grids, to retrieve the outputs of our dataset. Considering several cases of different resolution (thus, with different dofs), we gradually construct a global training set, ready to be employed in a learning algorithm. As a support tools for implementing NNs models we adopt the Tensorflow framework [1, 2]. During the training phase we add to the loss function a priori knowledge about properties of transfer operators in order to improve the generalization properties of the model. This allows our method to results not only in predictions close to the expected results, but also in good convergence once applied in a MG context. In the numerical experiments, after a preliminary studies of model parameters, we compare the predicted transfer operator against the computed L^2 -projection both applied in a MG solver on the same test equation. Further tests on unstructured mesh present our method against AMG, showing their differences in terms of convergence rate.

This thesis is organized as follows. In Chapter 1 we introduce the MG method. Starting from the discretization of an elliptic model problem through the FE method, we define the linear system of equation that we aim to solve. Then we give a brief overview on the concept of triangulation, focusing on procedures to create and refine meshes. After this preliminary discussion, we introduce the idea behind the MG method, explaining its two main components and the algorithmic framework to implement them. Finally, in Section 1.5 we direct our attention to the definition of transfer operator, distinguishing between the operator in case of nested meshes, AMG transfer operator, and the L^2 -projection. For the latter, we discuss the algorithm behind the computation of mesh intersection, in order to get the resulting matrix. To close the chapter, we briefly discuss the convergence of the MG method and its efficiency in terms of computational costs. Chapter 2 is devoted to ML methods, in particular to NN models. We employ the classic formalism to explain the goal of a learning algorithm applied on a training set. We focus on the definition of loss function and how its minimization affect the prediction capacity of the model, followed by an overview of the issues that one must take into account when designing these systems. Then, we go into details in the NN model, its structures and the minimization algorithms em-

ployed to optimize it. The last Section is dedicated to the improvements through regularization methods. In Chapter [3](#) we land on the Neural Multigrid (NMG) method, the goal of this work. Here, we make use of the concepts of the previous chapter to introduce and define the main components of our methodology. First, we explain how we define the training set, both in 1D and two 2D, seeing in details what we take from the mass matrix and the transfer operator. An important part is devoted to the critical aspects one need to consider when creating the several scenarios from which we retrieve our dataset examples. Then, we pass to the model training, explaining the custom loss function applied to the algorithm, the complexity of the architecture and the correct definition of the network parameters. Finally, we explain the method implementation, with an eye on the treatment of different patterns inside the geometry and how to address them. The numerical results follow in Chapter [4](#), where first a technical specifications on the machine architectures employed is given, followed by a presentation of the main libraries used in coding the Neural MG method. Then we present the results in 1D and 2D, first discussing some preliminary experiments for tuning the various parameter of the NN models, and then in terms of CPU time spent to obtain the predicted transfer operators and the convergence rate when we use these operators in a MG solver.

Chapter 1

Multigrid

In this chapter, we present the main aspects of the Multigrid (MG) method, one of the fastest methods for solving elliptic partial differential equations. The idea behind this method, in contrast with the classic iterative schemes, is to use different procedures for high and low frequency parts [11, 15]. We use iterative strategies, such as Jacobi and Gauss-Seidel [75] for smoothing the error, i.e., for removing its high frequency components. After a few iterations, we use a coarse grid to deal with the low frequency components. This strategy needs to transfer data or information between the grids. Therefore, a crucial point for reaching fast convergence is the definition of suitable transfer operators. Different approaches give rise to different transfer operators, some defined through information coming from the geometry, and others directly from the operator matrices obtained after the problem discretization.

Introducing an elliptic model problem we propose a discretization with the Finite Element Method (FEM) [11, 14, 19, 57, 66, 69, 97]. Once we obtain the arisen linear system of equations to be solved, we give a brief overview on triangulations strategies, to then focus on the main components of MG method: smoothing procedure and coarse grid correction. In Section 1.4 we present the actual algorithm, explaining how the main components are used together to obtain a fast convergence in solving a discretized problem. First, we introduce the simple two-grid method, which only considers a fine and a coarse mesh. Then, we generalize the procedure on a hierarchy of more levels, applying the two-grid method recursively to reach the solution. We conclude Section 1.4 briefly describing different MG cycles, showing the schemes obtained increasing the number of levels in the hierarchy. An overview of the concept of transfer operator follows, referring to interpolation operators in case of Geometric MG, L^2 -projection that is usually employed for the Semi-Geometric MG method and transfer operators

computed algebraically, with a focus on the Algebraic MG method. We explain how to define them, providing some examples. To conclude the chapter, we present a brief overview on the convergence of the method, and the efficiency of its implementation.

1.1 Model Problem and FE Discretization

Let us consider the following elliptic problem:

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega, \end{cases} \quad (1.1)$$

where $\Omega \subset \mathbb{R}^d$ is a bounded domain with Lipschitz continuous boundary $\partial\Omega$ and $f \in L^2(\Omega)$. For discussions on the analytical solution of the boundary-value problem, refer to [74].

1.1.1 Weak Formulation

Consider the test function $v \in V = H_0^1(\Omega)$. We multiply (1.1) by v and integrate over Ω :

$$-\int_{\Omega} \Delta u v \, d\Omega = \int_{\Omega} f v \, d\Omega.$$

Integrating by parts the left-hand side, we obtain

$$\int_{\Omega} \nabla u \nabla v \, d\Omega = \int_{\Omega} f v \, d\Omega.$$

Problem (1.1) is then reformulated as follows

$$\text{find } u \in V : \int_{\Omega} \nabla u \nabla v \, d\Omega = \int_{\Omega} f v \, d\Omega \quad \forall v \in V. \quad (1.2)$$

We can rewrite the weak formulation (1.2) in a more compact way:

$$\text{find } u \in V : a(u, v) = F(v) \quad \forall v \in V, \quad (1.3)$$

where a is a continuous symmetric elliptic bilinear form:

$$a(u, v) : V \times V \rightarrow \mathbb{R}, \quad a(u, v) = \int_{\Omega} \nabla u \nabla v \, d\Omega,$$

and F is a continuous linear functional:

$$F(v) : V \rightarrow \mathbb{R}, \quad F(v) = \int_{\Omega} f v \, d\Omega.$$

1.1.2 Discrete Formulation

Let V_h be a family of spaces depending on h such that

$$V_h \subset V, \quad \dim(V_h) = N_h \quad \forall h > 0.$$

The approximate version of problem (1.3) takes the form

$$\text{find } u_h \in V_h : a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h. \quad (1.4)$$

Let us denote with $\{\varphi_j, j = 1, \dots, N_h\}$ a basis of V_h . Since all the function in V_h are linear combinations of the basis, we get

$$\text{find } u_h \in V_h : a(u_h, \varphi_i) = F(\varphi_i) \quad i = 1, \dots, N_h. \quad (1.5)$$

The same works for u_h , being in V_h ,

$$u_h(\mathbf{x}) = \sum_{j=1}^{N_h} u_j \varphi_j(\mathbf{x}),$$

where u_j are unknown coefficients.

Equation (1.5) becomes

$$\text{find } u_h \in V_h : \sum_{j=1}^{N_h} u_j a(\varphi_j, \varphi_i) = F(\varphi_i) \quad i = 1, \dots, N_h. \quad (1.6)$$

Let us denote with A the so-called stiffness matrix with elements $a_{ij} = a(\varphi_j, \varphi_i)$, and with \mathbf{f} the vector with components $f_i = F(\varphi_i)$. Then solving problem (1.6) is equivalent to solve the linear system

$$A\mathbf{u} = \mathbf{f}, \quad (1.7)$$

where \mathbf{u} is the vector of coefficients u_j .

1.2 Mesh

Let us recall the polygonal domain Ω . We consider a non-overlapping partition of Ω into elements T , and we call it *triangulation* \mathcal{T}_h . The discretized domain

$$\Omega_h = \text{int} \left(\bigcup_{T \in \mathcal{T}_h} T \right)$$

defined as internal part of the union of elements of \mathcal{T}_h coincides with Ω . Here, $\text{int}(A)$ indicates the internal part of the set A , corresponding to the region obtained by excluding the boundary from A .

When $d = 1$ our elements are simple sub-intervals; if $d = 2$ we have triangles. Unless explicit ambiguity occurs, we will refer to the above definition equivalently as mesh or grid.

1.2.1 Grid Generation

Let us consider a one-dimensional scenario, i.e., Ω being an interval $[a, b] \subset \mathbb{R}$. For simplicity, let us use a sub-interval partition using a constant step-size h . We choose the number of elements N and we define $h = \frac{b-a}{N}$. Let us introduce the points $x_i = a + ih$, $i = 1, \dots, N$; these points are called vertices. In more general case, we would use non-uniform partitions of interval $[a, b]$ where we define a spacing function \mathcal{H} , where $\mathcal{H}(x)$ represents the spacing in correspondence of point x .

Let us now consider the two-dimensional case. Given a polygonal domain $\Omega \subset \mathbb{R}^2$, we can associate it with a partition \mathcal{T}_h of polygons. Given two distinct elements, their interior does not overlap. Furthermore, we admit only *conforming* triangulations. An example of conforming and non-conforming triangulations is reported in Figure 1.1. Formally, given two elements T_1 and T_2 of a mesh, if their intersection $F = T_1 \cap T_2$ is non-empty, then F is either a vertex or a whole edge. Also for the multidimensional case ($d > 1$), we use h to indicate the spacing of the mesh. Let us call $h_T := \text{diam}(T)$, $\forall T \in \mathcal{T}_h$, where $\text{diam}(T) = \max_{x,y \in T} |x-y|$. We set $h = \max_{T \in \mathcal{T}_h} h_T$.

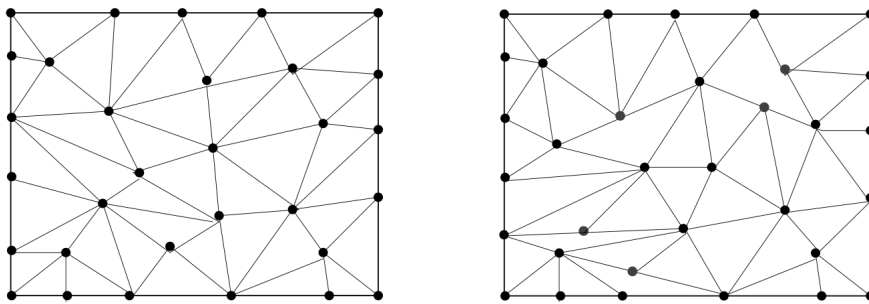


Figure 1.1. Example of conforming (left) and nonconforming (right) grid. Picture taken from [69].

Usually we distinguish between *structured* and *unstructured* grids. A structured grid has a regular connectivity, i.e, is a mesh where the inner nodes have the same number of elements around them. This kind of mesh basically use quadrangular or triangular elements and are characterized by the fact that access to the vertices adjacent to a given node is immediate.

On the contrary, for unstructured grids we have an irregular connectivity. They offer a greater flexibility both from the viewpoint of a triangulation of domains of complex shape and for the possibility to locally refine the grid. Usually they consists of triangles.

One of the most common strategy used to generate unstructured grids is the *Delaunay triangulation* [18, 55]. For a given set of vertices, a Delaunay triangulation is a triangulation such that the circumscribed circle to each triangle contains no vertex in its inside. This kind of triangulation is the one maximizing the minimum angle of the grid triangles.

1.2.2 Refinement

Mesh refinement is a strategy to increase the accuracy of the solution of a discretized problem arising from a PDE. It works as a iterative procedure applied to the single elements. The simplest solution is refining by bisection, which essentially divides the existing elements in half. Usually the longest edge bisection is applied, where we select the edge with the greater length. Another strategy is mid-point refinement, which takes each triangle, compute the mid-point of each edge and create four new elements. Figure 1.2 shows both the refinements, starting from an initial mesh on the left, with at the center the result of bisection and the mid-point refinement strategy on the right.

When we need to increase the accuracy of the solution only in certain regions of the domain, we use the *adaptive refinement*. It increases the precision of the numerical computation looking at the requirements of a specific problem in specific areas.

1.3 Multigrid Idea

Multigrid methods [78, 84, 86, 90] come from the observation that classical iterative methods result in error smoothing. The issue then is how to modify these methods to make them effective on all error components. Starting from the fact that performing some preliminary iterations on a coarse grid improves the initial guess, we can think more carefully about the implication of this coarse grid. In

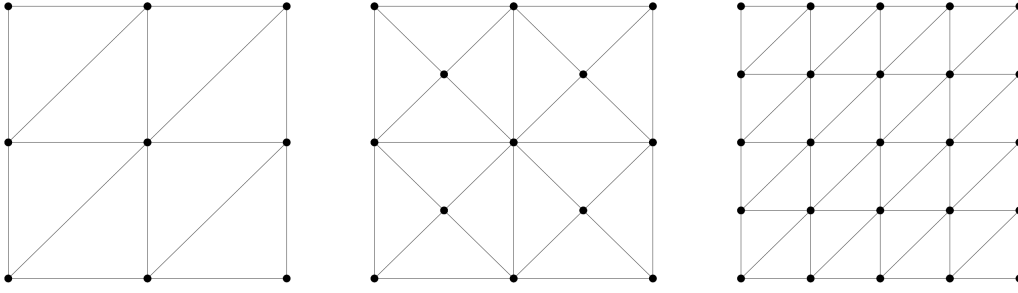


Figure 1.2. Refinement procedures applied on a mesh (left): refinement by bisection (center) and mid-point refinement(right).

passing from fine to coarse grid, we see more oscillatory components of the error: smooth modes on a fine grid look less smooth on a coarser grid. Therefore, when smoothing begins to stall, it is better to move on a coarser grid.

1.3.1 Error Smoothing procedure

The argument presented in this section follows the discussion in [86]. Error smoothing is one of the two basic principles of MG methods. Classical iterative schemes, like Jacobi and Gauss-Seidel, have a smoothing effect on the error of any approximation, when appropriately applied to discrete elliptic equations. Let us consider a two-dimensional domain $\Omega \subset \mathbb{R}^2$ and its FE discretization Ω_h . Let us consider the discrete Poisson equation

$$-\Delta_h u_h(x, y) = f_h(x, y). \quad (1.8)$$

We can write the error $e_h = e_h(x, y)$ as

$$e_h(x, y) = \sum_{k,l=1}^{n-1} \alpha_{k,l} \varphi_h^{k,l}(x, y), \quad (1.9)$$

where $\varphi_h^{k,l}(x, y) = \sin k\pi x \sin l\pi y$ are the discrete eigenfunctions of Δ_h .

As stated above, when we apply some steps of iterative methods, the error becomes smooth. This means that high frequency components in (1.9), i.e.

$$|\alpha_{k,l}| \sin k\pi x \sin l\pi y \quad \text{with } k \text{ or } l \text{ large}$$

becomes small after a few iterations, while the low frequency components, i.e.

$$|\alpha_{k,l}| \sin k\pi x \sin l\pi y \quad \text{with } k \text{ and } l \text{ small}$$

have not real changes.

Therefore, when we carry out the iterative (smoothing) procedure, the result is a clear reduction in the error as long as the latter contains highly oscillatory components. Once the error becomes smooth, the effect of the iterative method essentially stops. Figure 1.3 shows the effect on the error components of five iterations of Gauss–Seidel: Starting from the error related to the initial guess (top-left), we see an evident reduction in the first iterations, followed by a lower error decreasing until the fifth iteration (bottom-right).

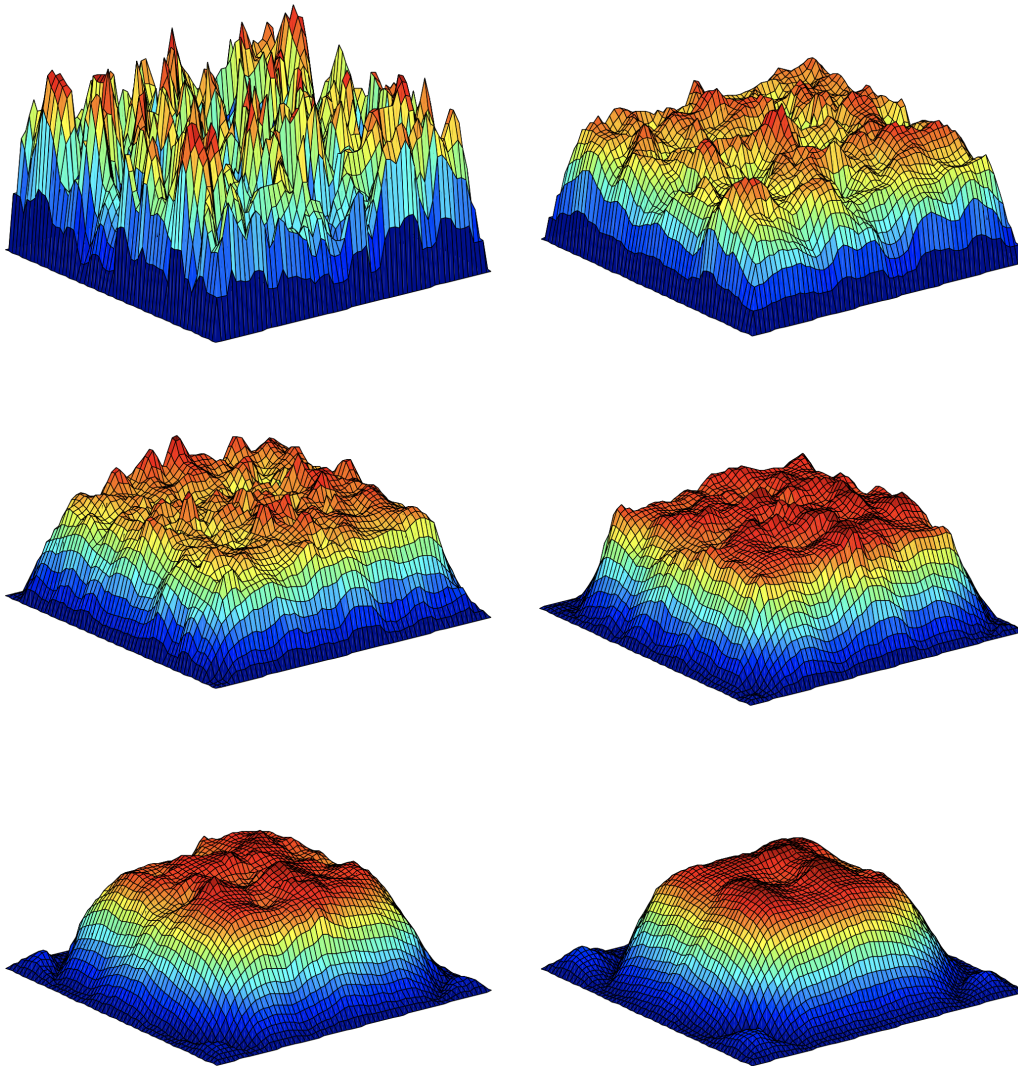


Figure 1.3. Illustration of the smoothing effect of the Gauss–Seidel method: error of the initial guess and the first five iterations. Picture taken from [25].

1.3.2 Coarse Grid Correction

After having carried out some smoothing steps, we remain with only smooth components of the error. A quantity that is smooth on a given grid can also be approximated on a coarser grid, through a suitable procedure. Then, a method for solving the problem on a coarser grid is less expensive, having less grid points. In Figure 1.4 we can better understand how a smooth quantity is approximated on a coarser grid. It is easy to see that what appear to be smooth on a fine grid becomes more oscillatory on the coarser one.

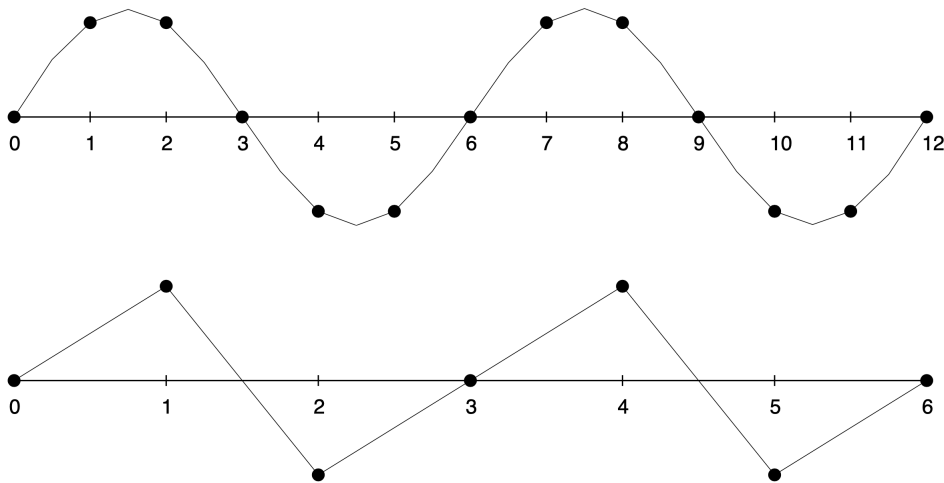


Figure 1.4. Error on fine mesh projected onto a coarse mesh. On the coarse grid there is a more oscillatory behavior than on the fine grid.

Let us consider the discrete elliptic problem

$$A_h u_h = f_h \quad (1.10)$$

on a mesh Ω_h , where $h = \frac{1}{n}$. Assume A_h to be invertible. Consider now a coarse grid Ω_H where $H > h$; usually $H = 2h$. Let u_h^i be the current approximation of the solution u_h . The error at the current iteration is

$$e_h^i := u_h - u_h^i.$$

Let us denote the *residual* by

$$r_h^i := f_h - A_h u_h^i.$$

We can now consider the residual equation

$$A_h e_h^i = r_h^i, \quad (1.11)$$

which is equivalent to equation (1.10) since $u_h = u_h^i + e_h^i$. If we approximate A_h by a simpler operator \hat{A}_h , the solution c_h of

$$\hat{A}_h c_h^i = r_h^i$$

gives a new approximation $u_h^{i+1} := u_h^i + c_h^i$.

The idea behind the coarse grid correction is to solve equation (1.11) using an appropriate approximation of A_h , namely A_H , on the coarse grid Ω_H . Thus, we replace the residual equation related to the fine grid with

$$A_H c_H^i = r_H^i, \quad (1.12)$$

where c_H^i and r_H^i are error and residual on the coarser grid, respectively. We consider the following formulas

$$r_H^i := R r_h^i,$$

$$c_h^i := P c_H^i.$$

The matrix R is called *restriction*, and it is used to restrict the residual on Ω_H ; matrix P is called *prolongation*, used to interpolate (or prolong) the correction on Ω_h . The choice of R and P has major influence on the rate of convergence.

Algorithm 1 presents a single step of the coarse grid correction strategy. Coarse grid correction is a nice procedure to deal with smooth error components, but if we take it as a standalone solution it is of no use, because it is a projector and hence it is not convergent.

1.4 The Multigrid Algorithm

The above arguments imply the necessity of combining the two procedures described, i.e., smoothing and coarse grid correction. Therefore, we present the so-called two-grid strategy, a simple case of the more general MG method. We focus on the main elements needed to solve a given problem with this procedure. Then, we pass to the actual MG method, a straightforward extension of the two-grid method, where the latter is used recursively until the coarsest level is reached. Finally, we briefly introduce the different MG cycles (schemes) that can be employed for solving a given problem.

Algorithm 1: Coarse Grid correction

1 Compute the residual on fine grid	$r_h^i = f_h - A_h u_h^i$
2 Restrict the residual	$r_H^i = R r_h^i$
3 Solve on Ω_H	$A_H c_H^i = r_H^i$
4 Interpolate the correction	$c_h^i = P c_H^i$
5 Compute new approximation	$u_h^{i+1} = u_h^i + c_h^i$

1.4.1 Two-grid Method

Let us consider a two-grid scenario, consisting of a fine and a coarse grid. First, we carry out some smoothing steps to dump the high frequency components of the error on the fine grid. Then we move to the coarse grid, approximating the remaining smooth parts. Here, we solve the coarse problem and go back on the fine grid interpolating the correction and computing a new current approximation. In the end, we apply other few steps of smoothing. Repeating this procedure, we create an iterative method. In general, we consider more than two grids, choosing the coarsest grid to be small enough to allow a direct solving after the restriction of the residual. We will focus on the actual MG algorithm ahead in this section.

We now present the complete two-grid procedure, showed in Algorithm [2](#). We use $\text{SMOOTH}^v(u_h^i, A_h, f_h)$ to indicates v smoothing steps, independent on the iterative scheme selected, which take the current iterate u_h^i , the operator A_h and the right hand-side f_h .

1.4.2 Multigrid Components

The choice of the components of this method has a strong influence on the resulting procedure. A correct choice of the coarse grid improves the multigrid strategy. The simplest approach is the standard coarsening, where we double the mesh size h in every direction. In d dimensions, we can find the number of coarse nodal points as

$$N_H \approx \frac{1}{2^d} N_h.$$

Other coarsening strategies do not depend directly on the fine mesh. The most important example is the Algebraic Multigrid (AMG), that uses algebraic relations in the operator matrix. We will talk extensively of this method in the later sections.

Another crucial point is the coarse grid operator A_H , which we can define as we

Algorithm 2: Two-grid Cycle

1 Pre-smoothing

Compute \bar{u}_h^i : apply ν_1 steps of smoothing to u_h^i

$$\bar{u}_h^i = \text{SMOOTH}^{\nu_1}(u_h^i, A_h, f_h)$$

2 Coarse Grid correction:

Compute the residual on fine grid

$$\bar{r}_h^i = f_h - A_h \bar{u}_h^i$$

Restrict the residual

$$\bar{r}_H^i = R \bar{r}_h^i$$

Solve on Ω_H

$$A_H c_H^i = \bar{r}_H^i$$

Interpolate the correction

$$c_h^i = P c_H^i$$

Compute new approximation

$$u_h^{i,\text{new}} = \bar{u}_h^i + c_h^i$$

3 Post-smoothing

Compute u_h^{i+1} : apply ν_2 steps of smoothing to $u_h^{i,\text{new}}$

$$u_h^{i+1} = \text{SMOOTH}^{\nu_2}(u_h^{i,\text{new}}, A_h, f_h)$$

do for the fine grid. However, several multigrid applications make use of the so-called *Galerkin coarse grid operator*

$$A_H := I_h^H A_h I_H^h, \quad (1.13)$$

with $I_h^H = R$ and $I_H^h = P$ being appropriate transfer operators representing restriction and prolongation operator, respectively. We will focus more on the choice of different grid transfer operators in Section [1.5](#).

1.4.3 Multigrid method

The two-grid method is a nice approach to illustrate how MG works, but it is of small use when applied on large-scale problems. When the number of nodal points increases, the relative coarse problem becomes computationally heavy to be solved by a direct solver.

Let us recall that we aim to find a correction c_H^i on the coarse grid. In order for the two-grid method to converge, it is not necessary that equation [\(1.12\)](#) is solved exactly. The idea behind MG is instead to replace c_H^i with an approximation \hat{c}_H^i . Therefore, we apply recursively the two-grid strategy: once we are on Ω_H , we employ a still coarser grid and move there to continue the current iteration. This is possible, since equation [\(1.12\)](#) is the same as the original equation [\(1.10\)](#). Of

Algorithm 3: Multigrid Cycle $u_k^{i+1} = \text{MG}(u_k^i, A_k, f_k, \nu_1, \nu_2)$

1 Pre-smoothing

Compute \bar{u}_k^i : apply ν_1 steps of smoothing to u_k^i

$$\bar{u}_k^i = \text{SMOOTH}^{\nu_1}(u_k^i, A_k, f_k)$$

2 Coarse Grid correction:

Compute the residual on fine grid

$$\bar{r}_k^i = f_k - A_k \bar{u}_k^i$$

Restrict the residual

$$\bar{r}_{k-1}^i = I_k^{k-1} \bar{r}_k^i$$

Compute \hat{c}_{k-1}^i of the residual equation on Ω_{k-1}

$$A_{k-1} \hat{c}_{k-1}^i = \bar{r}_{k-1}^i$$

by

· if $k = 1$ then use a direct solver
 · if $k > 1$ then $\hat{c}_{k-1}^i = \text{MG}(0, A_{k-1}, f_{k-1}, \nu_1, \nu_2)$

Interpolate the correction

$$\hat{c}_k^i = I_{k-1}^k \hat{c}_{k-1}^i$$

Compute correct approximation on Ω_k

$$u_k^{i,new} = \bar{u}_k^i + \hat{c}_k^i$$

3 Post-smoothing

Compute u_k^{i+1} : apply ν_2 steps of smoothing to $u_k^{i,new}$

$$u_k^{i+1} = \text{SMOOTH}^{\nu_2}(u_k^{i,new}, A_k, f_k)$$

course, we can apply this recursive procedure until we obtain a suitable coarsest grid, i.e., a grid with only a few degrees of freedom.

We now present a formal description of the multigrid strategy. Let us consider a sequence of grids $\{\Omega_{h_k}\}$, depending by a sequence of mesh sizes $\{h_k\}_{k=0}^L$:

$$\Omega_{h_L}, \Omega_{h_{L-1}}, \dots, \Omega_{h_0},$$

where Ω_{h_L} and Ω_{h_0} indicate finest and coarsest grid, respectively. In the following we use index k instead of h_k to lighten the notation. For each Ω_k we consider a restriction operator I_k^{k-1} , a prolongation operator I_{k-1}^k and a k -level stiffness matrix A_k . Therefore, the original equation (1.10) to be solved now reads

$$A_L u_L = f_L.$$

Having this in mind, we present the MG strategy in Algorithm 3. For completeness, we report in Figure 1.6 the solution of a test problem on grids of different approximations, to show how the accuracy of the solution changes when considering different mesh resolutions.

1.4.4 Multigrid Cycles

The procedure described in Algorithm 3 keeps restricting the problem until the coarsest grid, and then interpolate the approximate corrections until the finest level. A scheme of this kind takes the name of *V-cycle*.

Let us define a cycle index σ which indicates the number of recursive calls to the two-grid method. Of course, $\sigma = 1$ indicates a *V-cycle* scheme; when $\sigma = 2$ we refer to the so-called *W-cycle* scheme. Figure 1.5 shows the different cycles when we increase the number of levels: (a) shows the classic two-grid cycle; in (b) we have a three-grids hierarchy with $\sigma = \{1, 2, 3\}$; (c) presents four levels with cycle index $\sigma = 1$ on the left and $\sigma = 2$ on the right.

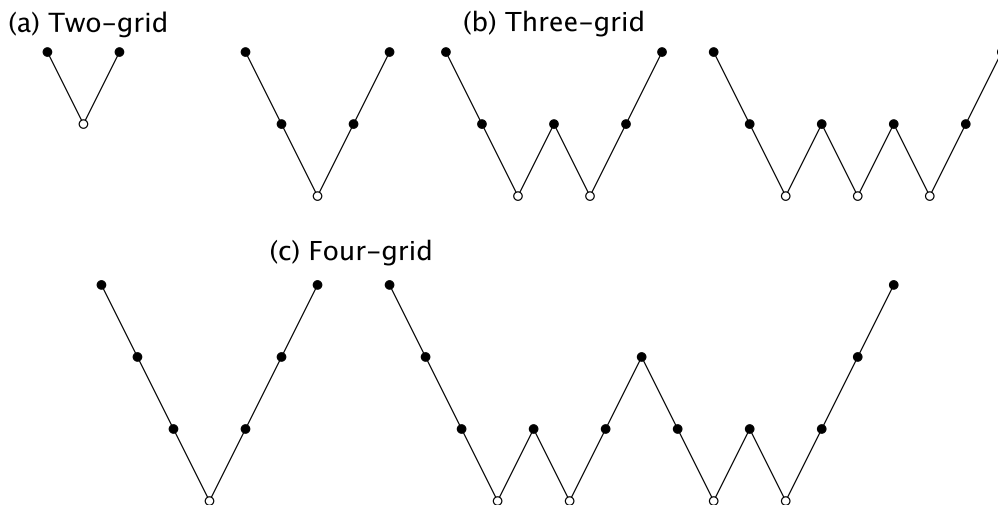


Figure 1.5. Different MG cycles, increasing the number of levels. The empty dots represent the coarsest grids. Three different hierarchies are considered: two, three and four levels; from the three-levels example, we show both V- and W-cycle.

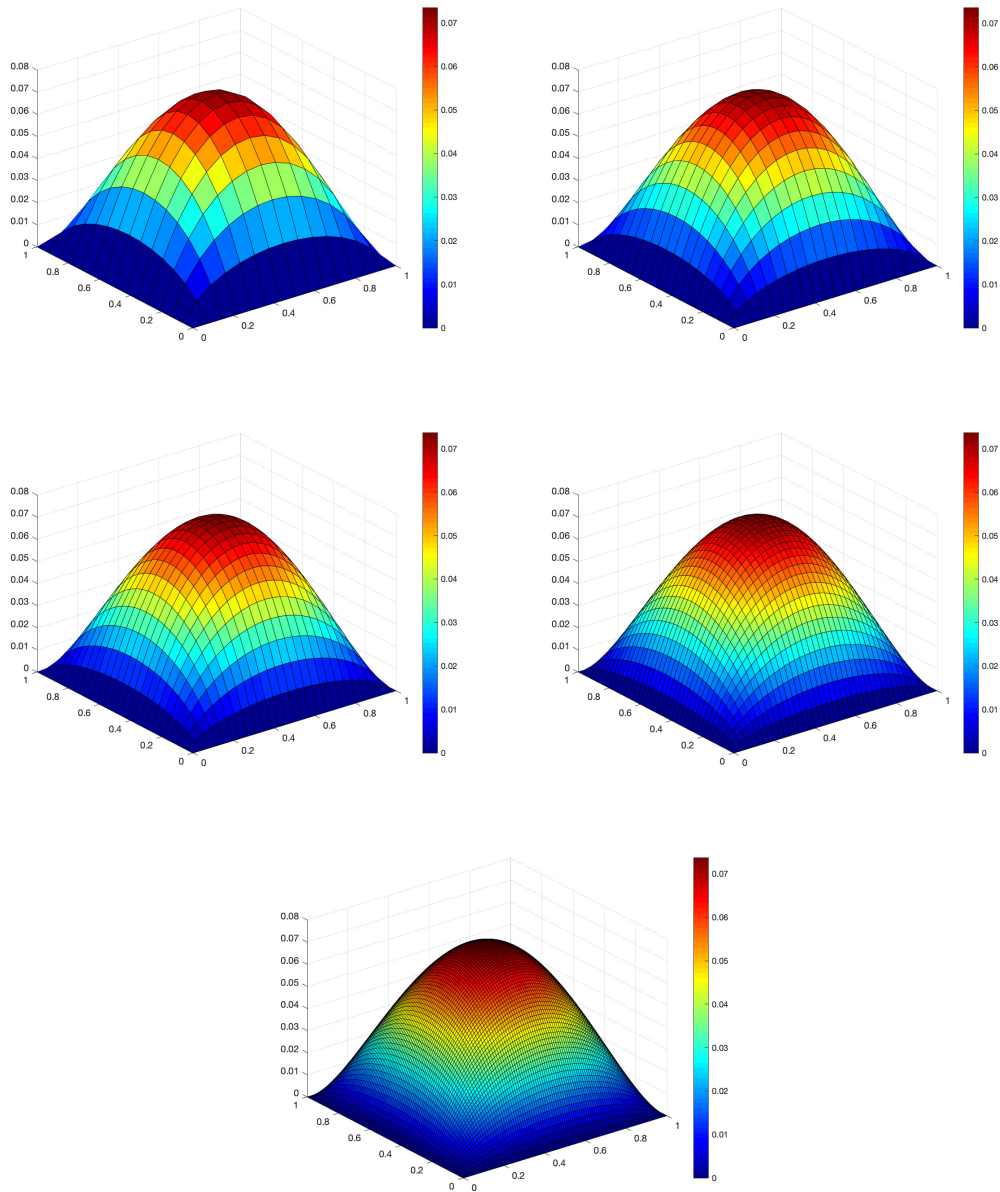


Figure 1.6. Solution of a test problem on different levels of the multilevel hierarchy. Starting from the coarsest level (top-left picture), we keep refining obtaining a better grid and therefore a more precise solution, until we reach the finest mesh (bottom picture).

1.5 Transfer operators

Let us return to the simple two-grid strategy. To perform the coarse grid correction we need to move between fine and coarse meshes, i.e. between Ω_h and Ω_H . The choice of the transfer operators for transport information between Ω_h and Ω_H has a major influence on the convergence rate. Furthermore, the definition of restriction and interpolation operators I_h^H and I_H^h is closely related to the coarse grid. We start this discussion introducing the transfer operators used in the Geometric multigrid method (GMG), easy to define but with the constraint of being optimal in case of nested grids, and therefore applicable in a few cases. Then we introduce the Algebraic Multigrid (AMG) approach, where we treat algebraic systems without a grid-oriented background: we only look at the operator matrix to retrieve connectivity information between nodes, forgetting the underlined geometry. Here, we briefly describe the application of AMG on problem with a variable diffusion coefficient, seeing how it can perform better than classic geometric transfer operators. Finally we discuss the L^2 -Projection as transfer operator, introducing the Semi-Geometric multigrid method (SGMG), which we can use for more general mesh scenarios.

1.5.1 Geometric Transfer Operator

A GMG approach operates on predefined grid hierarchies. Usually we obtain the hierarchy using simple refinement or coarsening strategies; the latter will be explained better in the next sections. This means that the refinement (coarsening) process is fixed and kept simple. When we fix the grid hierarchy, we require particular requirements on properties of the smoothing method employed, in order to have a fast convergence in combining smoothing and coarse grid correction. Let us consider a basis for Ω_h , i.e. $\{\varphi_i^h\}_{i=1}^{N_h}$, and let $\{\varphi_i^H\}_{i=1}^{N_H}$ be a basis for Ω_H . Since we obtain the grid hierarchy with a refinement (coarsening) process we can state $\Omega_H \subset \Omega_h$. Then, there exists a relation

$$\varphi_j^H = \sum_i a_{ji} \varphi_i^h, \quad (1.14)$$

where the coefficients a_{ji} compose the restriction matrix I_h^H of dimension $N_H \times N_h$. The examples of operators here reported are taken from [39], where they are discussed more in details. Let us now define some of the transfer operators used in this context. The simplest choice for the restriction operator is the *trivial injection*:

$$(R_{inj} r_h)(x) = r_h(x) \quad \forall x \in \Omega_H,$$

equations of the type:

$$Au = f \quad \text{or} \quad \sum_{j=1}^n a_{ij}u_j = f_i, \quad i = 1, \dots, n$$

In AMG there is no need of knowing a priori a multilevel hierarchy. Its construction is part of the algorithm itself, where we define also the coarsening process and the transfer operators. The only data required are the algebraic information contained in a given system of equations. For this reason, AMG and its range of applicability are more general, and it provides an attractive multilevel variant whenever GMG is difficult to apply.

The main difference with GMG is that the latter employs fixed grids, and therefore it needs to ensure a good interplay between smoothing and coarse grid correction. On the contrary, AMG fixes the smoother and enforces an efficient interplay with coarse grid correction choosing the coarser levels and interpolation appropriately. Another difference is the setup cost: in GMG we already have a hierarchy of meshes and pre-defined transfer operators, we only consider the solution phase. The algebraic approach needs to analyze the problem, to construct the coarse levels and the operators, and only then we can start the solution phase.

The coarsening process is fully automatic. We present a simple AMG algorithm, introduced by Beck [7]. Other strategies can be taken into account, e.g., the Ruge-Stüben method [72], which takes into account also the notation of strong connection between nodes, the Smoothed Aggregation method [87] and a coarsening procedure that uses Gaussian processes [35].

Let us consider the re-written system

$$A_h u^h = f^h \quad \text{or} \quad \sum_{j \in \Lambda_h} a_{ij}^h u_j^h = f_i^h, \quad i \in \Lambda_h, \quad (1.16)$$

where Λ_h denotes the index set $\{1, \dots, n\}$. In order to derive the coarse-level system from (1.16), we split Λ_h in two disjointed subsets: $\Lambda_h = C^h \cup F^h$; C^h denotes the variables contained in the coarse level, F^h is its complementary set. Thus, we define the coarse level Λ_H to correspond to C^h . Without entering into much details, we report a classical strategy for creating the F^h and C^h sets. We use the operator matrix to obtain a connection matrix S , which makes us understand the connections between nodes. We put all the node indexes into a set of remaining nodes R . We then apply the following procedure:

1. Choose an index i from R ; usually it corresponds to the smallest value in R .

2. Add i in C^h and remove it from R
3. Find all neighbors of i inside S , i.e. all those nodes k such that $S(i, k) \neq 0$
4. Put all nodes k in F^h and remove them from R
5. If there are still nodes remaining in R restart from 1.

As a post-process, we check for any fine-fine node connections which do not have a common coarse node. If any such nodes are found, one of the fine nodes will be made a coarse node. Now we can define the transfer operators. We fill the prolongation operator row-wise. If i is a coarse node, then the i th row of the operator correspond to the identity. On the contrary, if i is a fine node, then we compute the weighs, based on its connection to the coarse nodes. At the end of this process, we set the restriction operator to be the inverse of the prolongation operator, and we can thus solve the given linear system using the MG strategy. In Figure [1.7](#) we see the coarsening process previously illustrated, where the red dots indicate the coarse nodes, while the blue ones goes into the fine set.

We conclude the AMG discusion by pointing out its applicability to a certain class of equations. Let us consider the following PDE

$$-\operatorname{div}(k(\mathbf{x})\nabla u(\mathbf{x})) = f(\mathbf{x}), \quad (1.17)$$

where $k(\mathbf{x})$ is the diffusion coefficient. In case $k(\mathbf{x})$ is a constant function, the above equation can be written as Equation [\(1.1\)](#). Since they are based on geometry, GMG methods do not take into account this coefficient, and they could degrade if the diffusion coefficient is highly variable. On the contrary, AMG only consider the stiffness matrix, that actually contains data on $k(\mathbf{x})$. Therefore, AMG reaches convergence faster when applied on this kind of problems.

1.5.3 L^2 -Projection

We now define a multilevel framework based on non-nested meshes. The notation and calculations used here are reviewed from [\[85\]](#). Let us consider a domain $\Omega \subset \mathbb{R}^2$ and the space $V = H_0^1(\Omega)$. Let V^ϕ and V^φ be two finite-dimensional subspace of V , spanned by the basis $\{\phi_i\}_{i=1}^n$ and $\{\varphi_j\}_{j=1}^m$, respectively. Let u be a member of V^ϕ . The best approximation in the L^2 -norm of u onto V^φ is given by the orthogonal projection u^* . Then u^* must satisfy

$$(u - u^*, v)_{L^2} = 0 \quad \forall v \in V^\varphi, \quad (1.18)$$

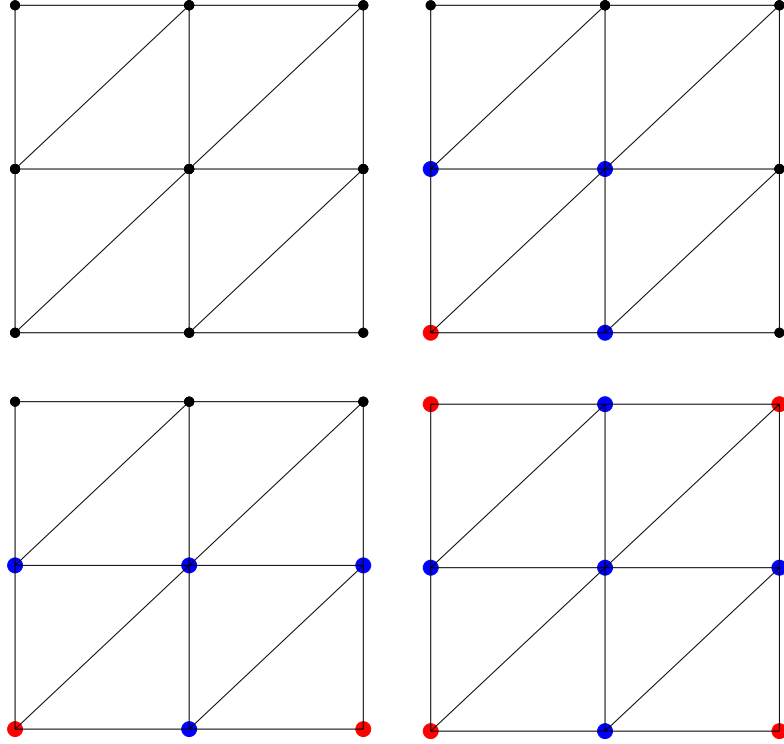


Figure 1.7. Classic coarsening procedure employed in AMG: the coarse nodes are shown in red, while the fine nodes are in blue.

where $(x, y)_{L^2} = \int_{\Omega} xy \, d\Omega$ represents the L^2 -inner product.

Since u is a member of V^{ϕ} , we can write it as a linear combination of its basis, i.e. $u = \sum_{j=1}^n u_j \phi_j$. The same goes for u^* which we write as $u^* = \sum_{j=1}^m u_j^* \varphi_j$. Therefore, rewriting (1.18) as a linear combination of basis functions, yields to

$$\sum_{j=1}^m (u_j^* \varphi_j, \varphi_i)_{L^2} = \sum_{j=1}^n (u_j \phi_j, \varphi_i)_{L^2}, \quad i = 1, \dots, m.$$

This allows us to define the L^2 -projection in terms of a linear system. Let us set $\mathbf{u} = (u_1, \dots, u_n)^{\top}$. Then we have

$$M\mathbf{u}^* = B\mathbf{u}, \quad (1.19)$$

where $\mathbf{u}^* = (u_1^*, \dots, u_n^*)^\top$ are the unknown coefficients. M represents the so-called mass matrix, where its entries are $M_{ij} = \int \varphi_i \varphi_j$ for $i, j = 1, \dots, m$. B is a rectangular coupling matrix (projection matrix), whose entries are $B_{ij} = \int \phi_j \varphi_i$, for $i = 1, \dots, m$ and $j = 1, \dots, n$. From (1.19) we can easily derive a construction for the projection operator:

$$\mathbf{u}^* = Q\mathbf{u},$$

where $Q := M^{-1}B$ is the actual discrete L^2 -projection.

As a last remark, it is worth mentioning that when we follow the above argument in case of nested meshes, the L^2 -projections assumes the characteristics of the geometric interpolation operator.

Computing the Operator. Here we focus on the computation of the coupling operator B , the trickiest part of the procedure. We need first to limit our finite element spaces to use only Lagrange basis functions. Furthermore, we only consider meshes generated through Delaunay triangulation. Let \mathcal{T}^ϕ and \mathcal{T}^φ denotes the triangulations related to V^ϕ and V^φ , respectively.

We consider the L^2 -inner product between the basis functions $\phi \in V^\phi$ and $\varphi \in V^\varphi$, as stated in the expression for computing B . Let \mathcal{I}_ϕ be the set of indexes such that $i \in \mathcal{I}_\phi$ if $T_i \subset \text{supp}(\phi)$; define \mathcal{I}_φ similarly. Given that each basis function is nonzero only on neighboring elements, the inner product $(\phi_j, \varphi_i)_{L^2}$ will be nonzero if and only if there exists $T_i \in \mathcal{T}^\phi$ and $\hat{T}_j \in \mathcal{T}^\varphi$ such that $T_i \cap \hat{T}_j \neq \emptyset$, where $i \in \mathcal{I}_\phi$ and $j \in \mathcal{I}_\varphi$. More intuitively, if two elements $T \in \mathcal{T}^\phi$ and $\hat{T} \in \mathcal{T}^\varphi$ do not intersect, the product between the basis functions related to their vertices must be zero. On the contrary, if they intersect the product must be nonzero only over the region of intersection.

In the following we describe an example of procedure to assemble the B matrix, which is detailed described in [51]. Let us consider the two triangulations \mathcal{T}^ϕ and \mathcal{T}^φ . We proceed as follows:

- Determine all pairs of intersecting elements $\langle T_i, \hat{T}_j \rangle$ such that $T_i \in \mathcal{T}^\phi$ and $\hat{T}_j \in \mathcal{T}^\varphi$. An example of algorithm to carry out this operation can be found in [30].
- For each pair $\langle T_i, \hat{T}_j \rangle$ we compute the intersection I and we mesh it.
- We define the quadrature point for integrate in the intersecting region I .
- Finally, we compute the local element-wise contributions using numerical quadrature, obtaining the operator B .

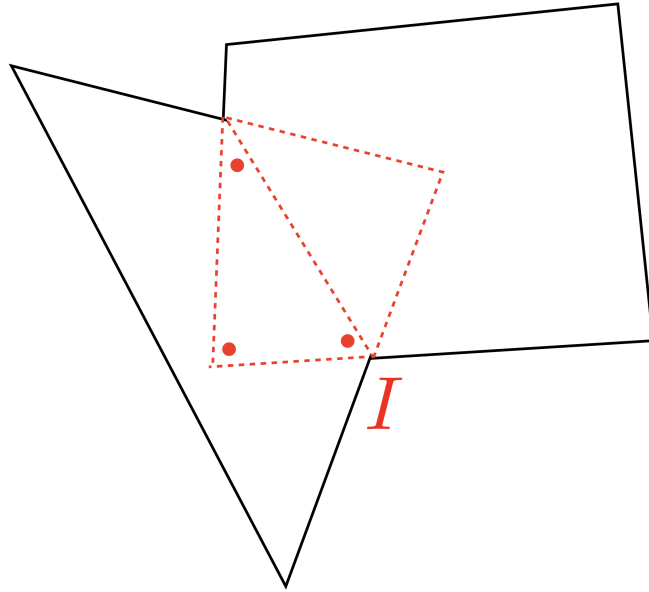


Figure 1.8. Intersection between 2D elements. picture taken from [51]

Figure 1.8 presents an example of intersection between elements. We overlap the elements finding their intersection region I , then we mesh it and generate the quadrature points for the following numerical integration.

Now, in order to obtain the actual transfer operator Q , we need to invert the mass matrix M . The inversion of a matrix is generally expensive, and in the case of the mass matrix, the resulting M^{-1} is a dense matrix. Therefore, the computation of Q itself becomes computationally expensive. To address this issue, we define the transfer operator by lumping the mass matrix M . In numerical practice, this results in lighter computations from which we get a sparse operator. An operator of this kind is called L^2 -quasi-projection, which is extensively defined in [25, 26].

Finally, a remark on the computational costs of determining the intersections. Let us consider, for simplicity, a two-dimensional scenario. A trivial solution is to check every element of T^ϕ with every element of \mathcal{T}^φ . If both meshes have N elements, this operation requires $\mathcal{O}(N^2)$ in terms of computational time. Therefore we adopt a smarter solution:

- Consider the number of nodes of the smaller mesh, i.e., N_p .
- Generate a square grid with $\sqrt{N_p}$ points in each dimension, so that in total we would have N_p nodes. The grid nodes must be evenly spaced. Each element of the grid is a square, to which we associate an index. Since the

grid is regular, given a point we can say in constant time the square index in which it belongs.

- For each node x^ϕ of T^ϕ , we find the grid square index i in which it belongs, and we associate i with x^ϕ .
- We do the same for each T^φ .
- For each grid square i , we intersect those elements of the two meshes, which have nodes belonging to i .

This has a complexity of $\mathcal{O}(N_p)$, thus is linear in the number of nodes of the mesh. This gives a great speed up need to intersect several meshes. Of course this argument can be extended in 3D, where instead of a grid consisting of square elements we consider cube elements.

1.6 Convergence and Efficiency

MG methods are among the fastest methods to solve elliptic PDE. Here we briefly present the efficiency and convergence of this class of method. As asserted in [11] MG iterations provide a relative error decreasing independent on the mesh size h . In order to obtain this result, various convergence proofs (starting from the work of Hackbusch [39]) focus on two separate parts: a *smoothing property* is combined with an *approximation property*. The first is related to the iterative method employed as smoother, and reads

$$\|S^\nu(u_h - u_h^0)\|_{H^1} \leq \frac{h^{-1}}{\nu} \|u_h - u_h^0\|_{L^2},$$

where S^ν indicates ν steps of the smoother S , $\|\cdot\|_{H^1}$ the H^1 -norm and $\|\cdot\|_{L^2}$ the L^2 -norm. The second part describes the interaction of the different levels of the MG hierarchy and can be expressed as:

$$\|u_h - u_H\|_{L^2} \leq 2h \|u_h\|_{H^1}.$$

In other words, the latter indicates the quality of the coarse grid correction. Without entering in details, convergence proofs use these two expressions and produces an estimate of a relative error decreasing independent of h . This states that the method converges and its convergence rate does not depend on the mesh size. For a more rigorous explanation and in depth analysis, refer to [12, 38, 92, 93].

The rate of convergence for MG iterations is bounded by a value independent of

the level. This means that the number of iterations for solving a given equation up to a certain accuracy is bounded from above by a constant not dependent on the number of the levels, i.e., one needs on each grid level essentially the same number of iterations to solve the problem. This behavior is in contrast to the classical iteration schemes, where the number of iterations increases, when the grid is refined. Let N_l be the number of degrees of freedom of a problem related to level l . It can be shown that the number of operations per multigrid cycle behaves like $\mathcal{O}(N_l)$. Since the number of multigrid cycles for solving the linear system is bounded uniformly, i.e., independently of level l , it follows that also the solution of the linear system requires $\mathcal{O}(N_l)$ operations.

Chapter 2

Deep Learning

The problem of retrieving representation of data is fundamental nowadays [8, 54, 68, 96]. In this chapter, we focus on Deep Learning (DL), a particular class of Machine Learning (ML) algorithms. DL allows computational models consisting of multiple layers to learn the representation of information with multiple levels of abstraction. Employing these systems, we can define solutions for speech and object recognition [24, 29], image classification [58], drug discovery [17] and for many other domains. DL models find complex structures in large datasets by optimizing its internal parameters, used to compute the representation in each layer, taking in input the representation in the previous layer. The goal of these methods is the construction of a pattern-recognition system that transforms the raw data (e.g. pixel values of an image) into a suitable internal representation of feature vectors from which the learning model could detect patterns in the input.

Here, we present the artificial Neural Network (NN) model, a common supervised learning method widely used for classification and regression tasks. We give an initial overview on ML methods, distinguishing between unsupervised [6, 32] and supervised [50, 79] approaches, beginning with a definition of learning algorithm. Discussing their ability to learn from data, we see the input and output structure related to such a method. Then, we focus on the approximation of a target function, essential to the correct optimization of these algorithms. As support to reach this goal, we introduce the concept of loss function, which aims to measure the prediction error. A section dedicated to the learning method properties follows, where we give the main definitions of generalization abilities, overfitting and underfitting. We briefly talk about dataset normalization, useful to make the method more stable. Before passing to the actual NNs, we discuss the parameters of the model not dependent on the learning itself, which control the algorithm behavior. Ending the section, we describe the stochastic gradient

descent method, used for the model optimization. Then we focus on NN models in Section 2.2, giving the basic concepts and going into details discussing their structure and the learning algorithms. We describe both the forward step and the backpropagation of the error. To end the chapter, we introduce regularization methods, employed to increase the NN ability to perform well on the non-training examples.

2.1 Machine Learning Methods

A ML algorithm is an algorithm that is able to learn from data [34, 60]. A system of this kind is said to learn from experience, with respect to some task, if its ability to perform that task improves with experience. ML allows to consider tasks which are generally difficult or expensive to solve with fixed programs. We define a task in terms of how the ML system should process an example. Depending on the learning algorithm we consider the examples differently: in case of unsupervised learning, we aim to discover patterns, without any pre-existing expected results related to the inputs; on the contrary, supervised learning works with both input and expected output, in order to learn the map between them. In the following, we always refer to supervised learning methods.

2.1.1 Learning Algorithms

We define an example as a couple (\mathbf{x}, \mathbf{y}) , where $\mathbf{x} \in \mathbb{R}^n$ is the feature set and $\mathbf{y} \in \mathbb{R}^m$ is the target. A feature is a measured quantity coming from some object or event that we want the learning system to process. Each entry $x_i \in \mathbf{x}$ represents a single feature. For example, considering an image as the input to the learning algorithm, its features are the values of the pixels. On the other hand, the target is the expected result associated to a given features set. Usually, before defining the actual feature set, there is a process of feature selection, which simplifies a ML problem by choosing which subset of the available features should be used. It is desirable to reduce the number of features to both reduce the computational cost of modeling and to improve the performance of the model. More details on this process are described in [46, 49].

ML methods allow us to tackle several kinds of task. Classification is one of the most common operation we perform in supervised learning. The algorithm is asked to output in which category some input belongs to. We ask the model to approximate a function $f: \mathbb{R}^n \rightarrow \{1, \dots, k\}$, where k is the number of possible categories. A classic example is object recognition, where we input images and

get from the system a numeric code corresponding to the identified object. Another task broadly addressed, is regression [28]. We ask the system to predict continuous values given the input. The model is asked to output a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$. When $m > 1$ we talk of multi-target regression [89]. An example is the prediction of house prices for a specific real estate market. In the following, we mostly refer to the task of regression when talking about learning algorithms.

In order to evaluate the learning abilities of a specific algorithm, we define a measure of its performance. Usually the measure directly depends on the task we aim to solve. For an algorithm which learn to perform classification, we use the *accuracy* of the model. Accuracy is simply the proportion of the examples correctly predicted. For regression, we adopt the Mean Squared Error (MSE) metric to measure how well our model is performing.

Let us consider the simple example of linear regression. Our goal is to define a model that takes a vector $\mathbf{x} \in \mathbb{R}^n$ in input and predict a scalar y . Let y_{pred} be the prediction of our model. Then we define the output as

$$y_{\text{pred}} = \mathbf{w}^\top \mathbf{x},$$

where $\mathbf{w} \in \mathbb{R}^n$ is a vector of parameters. The parameters are the values which define the actual behavior of our model. We call \mathbf{w} the vector of *weights*. The weights determine how each feature in \mathbf{x} affects the predictions. The above expression describes our task: we want to predict y by predicting $y_{\text{pred}} = \mathbf{w}^\top \mathbf{x}$.

Now we only need a measure of the model performance. We consider another set of examples that are not used for training the model, namely the *test set*. Let m be the cardinality of this set. Also for the test set we have both features and target. We define the following metric to measure the goodness of our system:

$$\mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y}) = \frac{1}{m} \sum_i (\mathbf{y}_{\text{pred}} - \mathbf{y})_i^2,$$

which can be written as

$$\mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y}) = \frac{1}{m} \|\mathbf{y}_{\text{pred}} - \mathbf{y}\|_2^2. \quad (2.1)$$

We call $\mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y})$ the *loss function* (or cost function). This measure decreases to 0 when $\mathbf{y}_{\text{pred}} = \mathbf{y}$. Therefore, to design a ML algorithm, we need to optimize the weights \mathbf{w} in a way that reduces $\mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y})$. More details on the minimization of the loss function will follow in Section 2.1.5. As a last remark, it is worth

noting that we usually consider more sophisticated models with an additional parameters, i.e., an intercept term b :

$$y_{\text{pred}} = \mathbf{w}^T \mathbf{x} + b.$$

This makes the mapping from features to predictions an affine function. The intercept term is often called *bias*.

2.1.2 Capacity, Overfitting, Underfitting

The crucial point in ML is defining and training models that perform well on previously unseen examples [34, 61]. The ability to provide good results on unobserved inputs is called *generalization*. When we compute some error measure on the training set, to then reduce this error we are apply a classic optimization technique. In addition, in ML we aim also to reduce the generalization error (or test error), i.e., the error measured on new inputs. We make use of a test set to compute the generalization error. Usually, the test set consists of records that are collected separately from the training set. Therefore, we train the model minimizing the training error, but we actually care about the test error. In order to improve the model performance and being able to reduce the test error, we need to make assumptions on how both training and test set are collected. We generate training and test data using a probability distribution over datasets. Then, the examples in these datasets must be independent from each other, and the training and test set must be identically distributed, drawn from the same probability distribution.

The ML method optimizes its parameters using the examples in the training set, and then it considers new sampled examples for the test set to compute the generalization error. This implies that the test error is greater than or equal to the training error. We then consider two factors to determine the goodness of a learning algorithm: making the training error small, and reduce the gap between training and test error. To these two challenges correspond, respectively, *underfitting* and *overfitting* (for further reading refer to [27, 34, 41]). We experiment underfitting when the model training does not provide a low training error. Overfitting happens when the gap between training and test error is too large. Focusing on the model's *capacity*, we can control if the algorithm is likely to overfit or underfit. The capacity of a model is its ability to fit a wide variety of functions. Intuitively, low capacity models struggles to fit the training set; high capacity models, on the contrary, could overfit, memorizing training set properties that do not serve them well on test data.

In order to control the model's capacity we choose its *hypothesis space*, i.e., the

set of functions that the algorithm is allowed to select as solution. Let us consider again the linear regression examples, i.e, $y_{\text{pred}} = b + wx$. Its hypothesis space consists of all linear function of its inputs. We can increase the model capacity by including also polynomials in the hypothesis space: $y_{\text{pred}} = b + w_1x + w_2x^2$. This remains a function of the inputs, thus we can still train the model as before.

ML methods have good results when their capacity is appropriate for the complexity of a specific task. Models with insufficient capacity result in a low ability to solve complex tasks. Model with high capacity are good in solving complex tasks, but they may overfit if the capacity is higher than the task complexity.

2.1.3 Preprocessing

A central point in learning from data is feature normalization. Scaling the input values improves the numerical stability of the model. Furthermore, it may speed up the training process. We focus on two strategies for normalize the data: *Max-Min* normalization and standardization.

Max-Min normalization is the simplest method, and consists in rescaling the *features* range in a target range, usually $[0, 1]$. If we call \mathbf{x} our *features* set, then

$$\mathbf{x}' = \frac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})},$$

where \mathbf{x}' indicates the scaled set, \min and \max two functions which return a vector with the minimum and maximum value of each *feature*, respectively.

Standardization rescales the distribution of *features* values making the mean 0 and the standard deviation 1. It can be though as centering the values. We rescale the dataset as

$$\mathbf{x}' = \frac{\mathbf{x} - \mu(\mathbf{x})}{\sigma(\mathbf{x})},$$

where $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ represent the mean value and standard deviation of the *features* set, respectively. Standardization can be of help when we have input values with different scales.

2.1.4 Hyperparameters and Validation Set

Models in ML learn to optimize their parameters to improve their predictions and their generalization ability. On the contrary, there are certain parameters that are not adapted by the models themselves, but they are set by the model architect. These settings are called *hyperparameters*. Through these properties we can control the behavior of the learning algorithm. If we consider the regression task, an

example of hyperparameter is the degree of the polynomial we select, which acts as capacity hyperparameter. Another example can be a parameter which control penalty terms inside the loss function, as we will see in Section [2.2.3](#).

In order to learn hyperparameters, we need a *validation set* of examples that is not used by the training algorithm. This set of examples is not related to the test set we discussed above: it is important that the examples in the test set are not used to make choices about the model, including the hyperparameters. Therefore, we construct the validation set from the training data, splitting the training sets into two disjointed subsets. One of the two is used to learn the model parameters, the other is our validation set, that we use to estimate the generalization error after the training allowing to update the hyperparameters accordingly.

2.1.5 Stochastic Gradient

Most ML algorithms aim to minimize a cost function $y = f(\mathbf{w})$ during the learning process. We are interested in improving the value \mathbf{w} such that it minimizes y . The derivative of f is therefore useful to minimize $f(\mathbf{w})$ because it tells how to change the values of the input \mathbf{w} to make small improvements in y . In the context of ML, we usually optimize functions with many local minima that are not optimal. When the input is multidimensional, the optimization becomes really difficult to be carried out. Therefore, we settle for finding a low value of f , even if it is not minimal.

A widely used method for iteratively find a minimal point is the method of gradient (or steepest) descent [\[71\]](#). Intuitively, we decrease the value of f by moving in the direction of the negative gradient. If we start with a value of the inputs \mathbf{w}_0 , our first gradient descent iteration reads as

$$\mathbf{w}_1 = \mathbf{w}_0 - \eta \nabla f(\mathbf{w}_0),$$

where $\eta > 0$ is the learning rate, used to determine the size of the step.

As far as concern ML methods, we need to compute the optimization of the loss function using large training sets, that are necessary to reach good generalization. Of course, large training sets mean expensive computations, and therefore a heavy slow-down of the training algorithm.

Let us consider the loss function $\mathcal{L}(\mathbf{y}_{\text{pred}}^{(i)}, \mathbf{y}^{(i)})$, where $\mathbf{y}_{\text{pred}}^{(i)}$ is the prediction corresponding to the i th input $\mathbf{x}^{(i)}$ and $\mathbf{y}^{(i)}$ is the i th target in the training set. The cost function often decomposes as a sum over training examples of some per-example loss function. Therefore, being N the size of the training set, the

gradient descent algorithm requires to compute

$$\frac{1}{N} \sum_{i=1}^N \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{y}_{\text{pred}}^{(i)}, \mathbf{y}^{(i)}).$$

The complexity of this operation is $O(N)$, which becomes prohibitively when the training set has billions of records. Therefore, we employ the so-called *Stochastic Gradient Descent* (SGD) method [9, 10]. The idea behind this method is to use a small set of examples instead of all the training set. On each iteration of the method, that in this context takes the name of *epoch*, we sample *mini-batch* of records of size N' drawn from the training set. Usually N' is relatively small and it is fixed as N grows. Now, the gradient computation becomes

$$\mathbf{g} = \frac{1}{N'} \nabla_{\mathbf{w}} \sum_{i=1}^{N'} \mathcal{L}(\mathbf{y}_{\text{pred}}^{(i)}, \mathbf{y}^{(i)}).$$

We then update the parameters using

$$\mathbf{w}_{i+1} \leftarrow \mathbf{w}_i - \eta \mathbf{g}.$$

Although SGD has good performance in optimizing a cost functions, there are challenges to face, related to the magnitude of the updating in each iterations and the learning rate applied. In order to deal with these issues, there are some gradient descent optimization algorithm that can help us. One of the best methods is the *Adaptive Moment Estimation* (Adam), which inherits its idea from AdaGrad and RMSProp. For a more detailed discussion refer to [48, 71, 94, 95].

2.2 Deep Feedforward Networks

Deep Feedforward Networks are one of the most common methods in supervised ML. The aim of a network is to approximate some function f . In the example of a classifier, $y = f(\mathbf{x})$ maps a vector input \mathbf{x} to a category y . A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \mathbf{w})$, where \mathbf{w} are the parameters to learn so that they provide the best function approximation. We call these models 'feedforward' because the information flows from the input \mathbf{x} through the intermediate computations used to define f and finally to the output \mathbf{y} . The output does not go back into the model. Unless explicitly ambiguity occurs, through this thesis we denote with the generic term Neural Networks (NNs) this particular kind of

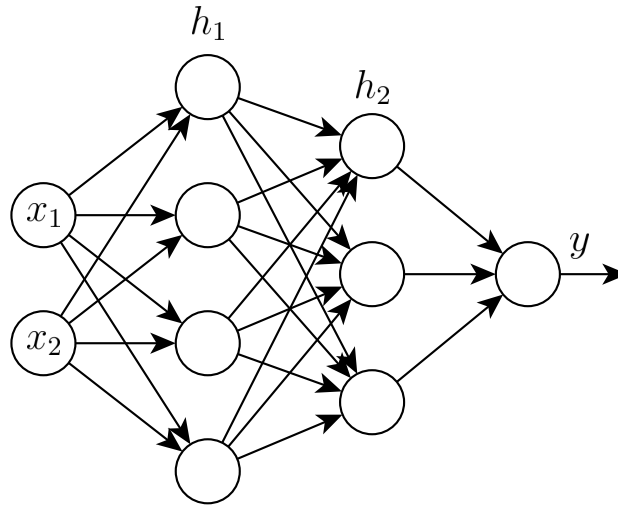


Figure 2.1. Example of Neural Network as an acyclic directed graph.

methods.

We represent NNs with direct acyclic graphs, which describe how different functions are composed together to generate the predictions.

In Figure [2.1](#) we see an example of NN. On the left we have the so-called input layer, while on the right we have the output layer. The layers in the middle, i.e., h_1 and h_2 , are called hidden layers.

The records used to train a NN specify what the output layer should do for each input \mathbf{x} , i.e., it must produce a result close to the given target. On the contrary, we do not specify the behavior of the other layers through the data. The training algorithm specifies how to use these layers for approximating the target function.

We proceed this section giving an overview of architecture and components of a NN, focusing on the predictions computation obtained through the activation functions of the several layers. Then, we present the backpropagation method, based on a local information transfer backwards from output to input, employed to update the network parameters during the loss function minimization.

2.2.1 Architecture

Let us consider the simplest neural network model, i.e. a perceptron, reported in Figure [2.2](#). Here, the output y is given by the applying an activation function

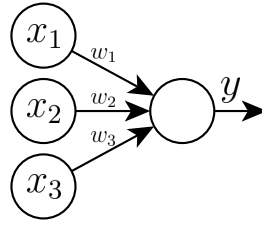


Figure 2.2. Example of perceptron.

σ on the weighted sum of the inputs, i.e.,

$$y = \sigma\left(\sum_{i=1}^3 w_i x_i + b\right),$$

where b indicates the bias. Several perceptrons stacked in different layers form a deep NN, as the one shown in Figure 2.1. Each neuron of a layer is connected to all the neurons of the next layer through weighted synapses. Each layer takes in input the result of the previous layer computation and apply an activation function on each of its neurons. The combination of these activation functions makes the model able to approximate complex target functions. In a model with two hidden layers, with respectively n_1 and n_2 neurons, we have, for each output neuron k :

$$y_k = \sigma_2\left(\sum_{j=1}^{n_2} w_{kj}^{(2)} \sigma_1\left(\sum_{i=1}^{n_1} w_{ji}^{(1)} x_i + b^{(1)}\right) + b^{(2)}\right).$$

Here, the weights related to each layer l are grouped together into a matrix $\mathbf{W}^{(l)}$, where the generic element $w_{ij}^{(l)}$ represents the weight connecting the j th neuron of layer $l - 1$ to the i th neuron of layer l .

Activation Functions. A crucial point in designing a NN model is the choice of the hidden units to use in the hidden layers, i.e., which activation functions are better for the specific task we are considering. Designing hidden units is an active area of research (see [4, 70]) and there are not definitive guiding theoretical principles. Recently, the most common choice of hidden unit is the so-called *Rectified Linear Unit* (ReLU). For further reading refer to [3, 91]. These units use the activation function

$$\sigma(z) = \max\{0, z\}.$$

Other choices of hidden units are possible: prior to the introduction of ReLU, most models used the logistic sigmoid function or the hyperbolic tangent function. For a comparison between different activation functions, refer to [63].

2.2.2 Backpropagation

In Section 2.1.5 we introduced the gradient descent algorithm to minimize a cost function in a general ML algorithm. Now, we aim to find a technique for evaluating the gradient of $\mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y})$ for a NN. This can be done using a local message passing strategy, in which the information travel forward and backward through the network. For this reason, we call it *backpropagation* ([44, 73]). This training method works as an iterative procedure which adjusts the weights in a sequence of steps. At each step, we distinguish two parts. First, we evaluate the derivatives of the loss function with respect to the weights. In this stage we propagate the error backwards through the network. In the second part, we use the derivatives to compute the weight adjustments.

In the following, we present the backpropagation algorithm for a general NN. Let us consider a loss function computed on a arbitrary example n of the training set, i.e., $\mathcal{L}(\mathbf{y}_{\text{pred}}^{(n)}, \mathbf{y}^{(n)})$. For simplicity, let us denote it with $\mathcal{L}^{(n)}$. Each unit of the network computes a weighted sum of its input

$$a_j = \sum_i w_{ji} z_i, \quad (2.2)$$

where z_i is the output of a unit i that sends a synapses to unit j , and w_{ji} is the weight associated to that synapses. We then apply an activation function on a_j :

$$z_j = \sigma(a_j). \quad (2.3)$$

One or more of variables z_i in (2.2) could be an input, as well as a unit j in (2.3) could be an output.

We now consider the evaluation of the derivative of $\mathcal{L}^{(n)}$ with respect to a weight w_{ji} . We note that $\mathcal{L}^{(n)}$ depends on w_{ji} only via a_j . Therefore, by applying the chain rule for partial derivatives we get

$$\frac{\partial \mathcal{L}^{(n)}}{\partial w_{ji}} = \frac{\partial \mathcal{L}^{(n)}}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}. \quad (2.4)$$

Let us introduce the notation

$$\delta_j \equiv \frac{\partial \mathcal{L}^{(n)}}{\partial a_j}. \quad (2.5)$$

From (2.2) we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \quad (2.6)$$

Using (2.5) and (2.6) into (2.4) we get

$$\frac{\partial \mathcal{L}^{(n)}}{\partial w_{ji}} = \delta_j z_i. \quad (2.7)$$

Hence, in order to compute the derivatives, we need to compute δ_j for each hidden and output units, and then apply (2.7). For the output unit, the error is simply the distance between the output value $\mathbf{y}_{\text{pred}}^{(n)}$ and the target $\mathbf{y}^{(n)}$. For evaluating the δ 's for hidden units we use again the chain rule,

$$\delta_j \equiv \frac{\partial \mathcal{L}^{(n)}}{\partial a_j} = \sum_k \frac{\partial \mathcal{L}^{(n)}}{\partial a_k} \frac{\partial a_k}{\partial a_j}, \quad (2.8)$$

where the sum consider all the units k to which unit j sends connection. Finally, substituting the definition of δ given by (2.5) in (2.8) and use (2.2) and (2.3), we get the following backpropagation formula

$$\delta_j = \sigma'(a_j) \sum_k w_{kj} \delta_k. \quad (2.9)$$

Equation (2.9) tells that the value of δ for a particular unit can be computed by propagating the δ 's backwards from units ahead in the network.

2.2.3 Regularization Methods

A crucial point in ML is improving the ability of an algorithm to perform well on non-training examples. Other than increasing or decreasing the model capacity manipulating its hypothesis space, we can give the learning algorithm a preference for one solution over another. In order to do so, we add penalty terms in the loss function. This addition, together with other ways of expressing preferences for different solutions, is called *regularization*. More specifically, we refer to regularization when we apply any modification to a learning algorithm intended to reduce its generalization error.

A classic example is the inclusion of weight decay (or L^2 regularization) [52] into the cost function. Instead of considering the standard loss function $MSE(\mathbf{y}_{\text{pred}}, \mathbf{y})$ we add a criterion that expresses a preference on the choice of the weights, i.e.,

$$\mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y}) = MSE(\mathbf{y}_{\text{pred}}, \mathbf{y}) + \lambda \mathbf{w}^\top \mathbf{w},$$

where λ is a value which controls the strength of our preference for small weights. Therefore, when we minimize $\mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y})$, we aim to obtain a choice of

weights that make a trade-off between fitting the data and being small. More in general we denote a regularized loss function as

$$\mathcal{L}_p(\mathbf{y}_{\text{pred}}, \mathbf{y}) = \mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y}) + \alpha p(\mathbf{w}),$$

where $\alpha \in [0, \infty)$ is an hyperparameter weighting the contribution of the penalty term p .

A consequence of using the regularized cost function is that we can minimize a function subject to constraints. We only need to add the constraints to the loss function as penalty terms. For example, if we want a constraint $p(\mathbf{w})$ to be less than a constant k , we add the penalty term $\alpha(p(\mathbf{w}) - k)$.

Additionally, to directly manipulate the loss function, there are other regularization strategies that can be employed. To end the chapter, we describe two methods, which allow the model to better generalize its predictions: early stopping and dropout.

Early Stopping. During the training phase of a NN model, we often observe that the training error keeps decreasing over time, while the validation error, after decreasing for some time, begins to rise again, as shown in Figure 2.3.

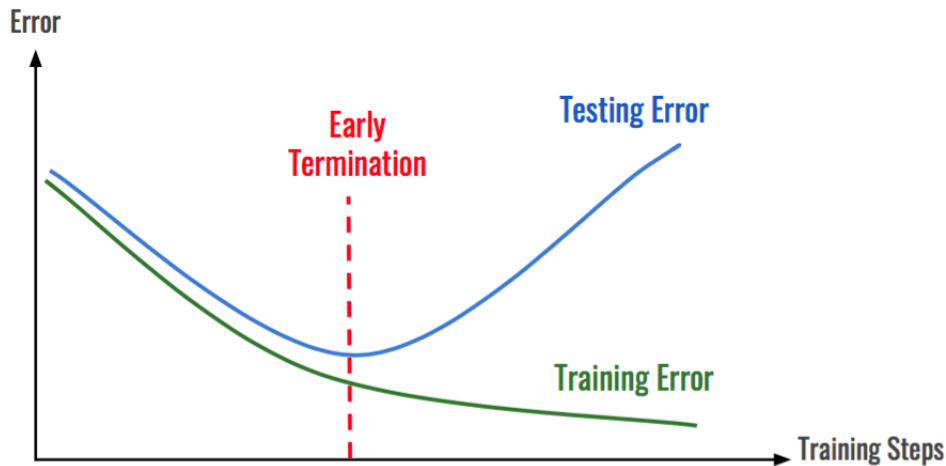


Figure 2.3. Early Stopping point¹

This means that after a while, our model starts to overfit, preferring training examples and losing its ability to perform well when applied on the validation set. Therefore, we can obtain a model with better validation error (and hopefully

¹Ananda Mohon Ghosh (2020, February 9). Early Stopping with PyTorch to Restrain your Model from Overfitting. <https://medium.com>

with better test error), by returning to the parameters setting at the point in time with the lowest validation set error. We keep track of the model parameters each time the validation error improves. When the training phase terminates, we go back to the best set of parameters, rather than the latest state of the model. Early stopping is a simple form of regularization, that requires almost no change in the training algorithm. More details in applying this strategy can be found in [16, 67].

Dropout. When we are dealing with NNs with a large number of parameters, overfitting becomes a serious issue to be addressed. Given the large number of parameters, these models are slow to be used, and this makes inefficient the combination of predictions resulted from different model architectures. Dropout provides a computationally inexpensive solution to this problem (see [80, 81]). It can be thought as training and evaluating multiple models on each test example. The main idea is to randomly drop non-output units, together with their connections, during the training phase. This results in training a collection of thinned networks which share the weights values. Each unit has a probability to be included during a specific training step. This probability acts as hyperparameter and it is fixed before the training begins.

Chapter 3

Neural Multigrid

In this chapter, we introduce a ML approach in the context of MG methods: we propose a NN model able to predict transfer operators. In particular we aim to obtain the L^2 -projection. We give a definition for patch of a node, both in a geometrical context, thus on the mesh, and in the discrete operator. The extraction of the patches allows the construction of a training set, used to optimize the model parameters to then predict a suitable transfer operator. We define the procedure to obtain a correct data distribution, focusing on the correct geometric (mesh) scenarios. To make the NN able to predict appropriate values for a multigrid solver procedure, we focus on the definition of the loss function to be minimized. We add a priori knowledge about properties of the transfer operators, in order to improve the generalization abilities of the learning algorithm. Once the model is optimized, we employ it as a black box for solving linear system of equations in a MG fashion. We call this newly defined procedure Neural Multigrid (NMG) method.

NMG works as follows: we apply a coarsening procedure on the operator matrix, which contains the connection information of the nodes; for each identified coarse node, we retrieve its information and we use it to ask the NN to predict a part of the transfer operator. Once the full operator is defined, we can start with the MG solving procedure.

This chapter starts with a discussion on the methodology behind the creation of the training set. We consider one- and two-dimensional scenarios: we provide the definition of node patch in both cases, presenting their differences. After the definition of *features* and *target*, we explain the subdivision of examples into classes: this concept is crucial to avoid overfitting and produce suitable transfer operators. A brief description of the extraction of examples follows, with a focus on 2D geometries, which need more attention during the scenarios creation.

Then, a section is dedicated to the procedure details, to better explain our choices and decisions in building this strategy.

Before addressing our attention on the actual implementation of NMG, in Section 3.2 we explain the training of the model, where the custom loss function obtained through regularization is defined. Finally, the last section illustrates the main coding aspects that must be taken into account, to use our method in general cases and with more general and complicated grids.

3.1 Training Set

In order to construct a model employable for any problem size, i.e., a method independent of the mesh dimension, we need to localize the data that compose the training set. If a specific problem is considered in its global form, we would need several models, each created ad-hoc for a specific dimension. Therefore, we consider local parts of the problems, focusing on the concept of neighborhood of a node. Given a subset of pre-identified coarse nodes, obtained with a coarsening procedure, we extract a record for each of them. Let j be a coarse node. A record contains information that comes from the patch of node j , that we indicate with $\rho(j)$; here, $\rho(j)$ is the set consisting of node j together with its neighbors. We define the patch-size (or patch-dimension) of a node j to be the cardinality of the set $\rho(j)$.



Figure 3.1. Example of one-dimensional mesh.

Let us consider the one-dimensional scenario reported in Figure 3.1. We adopt the convention of increasing nodes indexes from left to right. In this example, $\rho(j) = \{j - 1, j, j + 1\}$. In practice, we define $\rho(j)$ directly from the mass matrix M_h , considering the set of those nodes $i \neq j$ such that $M_h(j, i) \neq 0$. The one-dimensional case is trivial, but the above definition holds also in the two-dimensional scenario. Let us refer to Figure 3.2 where the nodes are indicated with numbers, for simplicity. The patch of node 30 is highlighted, namely $\rho(30) = \{30, 14, 23, 28, 36, 46, 48\}$.

Let us now focus on the records. For each coarse node j , we define a couple (*features*, *target*), using the information related to $\rho(j)$. For each node $k \in \rho(j)$,

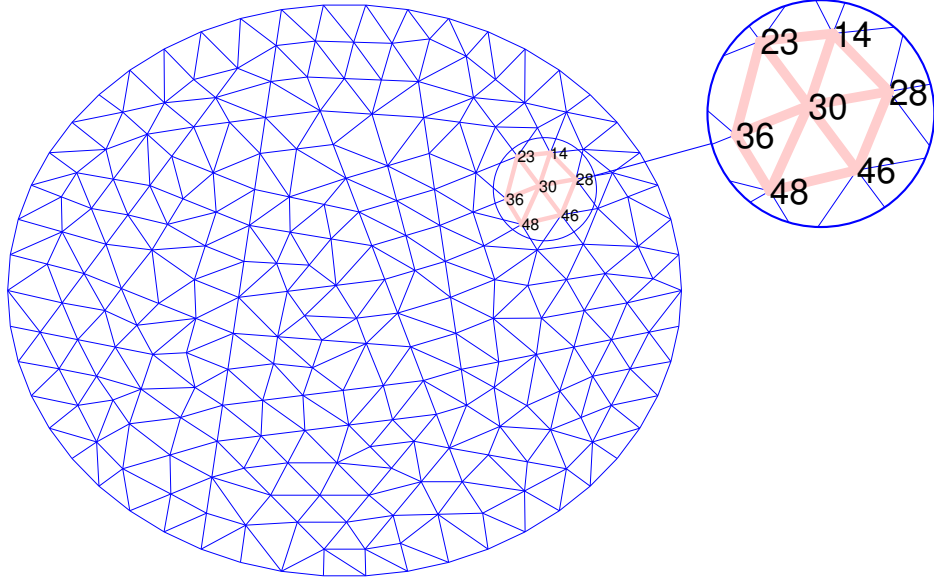


Figure 3.2. Example of two-dimensional mesh: here we zoom on a small portion of the mesh in order to underline the patch of a specific node.

we define the *features* and *target* as the non-zero entries of the rows k of M_h and the coupling matrix B_h , respectively. More details on the choices of the records components will follow in Section [3.1.4](#).

In general, for each coarse node j , we define the *features* as the triplet

$$\left(M_h(j, j), \Gamma_{M_h}^r(j), \mathcal{N}(j) \right),$$

where

- $\Gamma_{M_h}^r(j) := \{M_h(j, k) : k \in \rho(j) \setminus \{j\}\}$,
- $\mathcal{N}(j) := \{M_h(k, w) : k \in \rho(j) \setminus \{j\} \wedge w \in \rho(k) \setminus \{j\}\}$.

In order to define the *target*, we need more focus on the operator B_h .

Let N_h be the number of nodes in a mesh; then $B_h \in \mathbb{R}^{N_h \times N_H}$, where $N_H (< N_h)$ corresponds to the number of coarse nodes. We introduce two sets, C and F , which contain the indexes of coarse and fine nodes, respectively. In this context, F only contains those nodes which are not present in the C , i.e., F is the

complement of C . We set a map, $\gamma(j)$, which converts the indexes in C to the indexes enumeration of the coarse mesh. Let us consider Figure 1.4, focusing only on the two grids and ignoring the error functions. Applying $\gamma(\cdot)$ on nodes of the fine grid belonging to C , e.g., $\{4, 6, 10\}$, we obtain their indexes on the coarse grid, i.e., $\gamma(\{4, 6, 10\}) = \{2, 3, 5\}$. Furthermore, we define a function $\rho_C(j) := \rho(\gamma(j)) \setminus \{\gamma(j)\}$, which returns the coarse indexes of the neighbors of fine node j . To obtain them, we consider $\rho(j)$ and extend it, taking recursively $\rho(k)$ for all the neighbors k of j . From the resulting set, we extract only the coarse nodes, obtaining $\rho_C(j)$. Returning again on Figure 1.4, if we consider fine node 6, then $\rho_C(6)$ consists of coarse nodes 2 and 4.

Now we can define the *target* as the quadruplet

$$\left(B_h(j, \gamma(j)), \Gamma_{B_h}^c(j), \Gamma_{B_h}^r(j), \Gamma_{B_h}^r(\rho_C(j)) \right),$$

where

- $\Gamma_{B_h}^c(j) := \{B_h(k, \gamma(j)) : k \in \rho(j) \setminus \{j\}\},$
- $\Gamma_{B_h}^r(j) := \{B_h(j, k) : k \in \rho_C(j)\},$
- $\Gamma_{B_h}^r(\rho_C(j)) := \{B_h(k, z) : z \in \rho_C(k) \wedge k \in \rho_C(j)\}.$

It is important to always maintain the same order of the values in the *features* and *target* fields. For examples, in the *features* $M_h(j, j)$ must be always in the first position. This is crucial for the actual implementation of the custom loss function, as we will see ahead in the chapter.

3.1.1 Class of examples

In order to allow the NN to learn, we provide a large number of examples. These examples must come from different meshes, in terms of elements dimension and dofs, in order to consider values of different magnitude when filling the training set. Furthermore, we need several different examples to be sure of avoiding overfitting, occurring when the model predictions correspond too closely or exactly to a particular set of data. Thus, we define classes of examples, and we choose a fixed amount of records for each class. This allows us to create an unbiased training set, without preferring some classes over others. The definition of class is related to the mesh from which we extrapolate the records. We set a number of elements N ; all the records coming from meshes with N elements belong to class C_N . Then we pass to the next class, increasing N , and we repeat the generation

and extraction procedure. How we pass from a class to the next is crucial for reaching good accuracy in the training phase, as we will see for both one- and two-dimensional scenarios.

We consider different examples of two grids hierarchies, where each fine mesh is associated to only one coarse mesh. Doing so, we obtain an actual function which relates fine and coarse grids. This function is the one that the NN model has to approximate during its optimization. Since we consider several examples for the same class C_N , we need to make them different from each other, to avoid duplication inside the dataset. To reach this result, we take the fine mesh, and we move the nodes along the edges by a random quantity proportional to the step size h . This produces different elements, and consequently different records. Since NN models allow only fixed input and output dimension, we define distinct models for 1D and 2D scenarios.

3.1.2 One-dimensional Records

The records related to one-dimensional meshes are extracted from scenarios obtained by coarsening: starting from a random generated fine mesh, we decide which nodes to keep for defining the coarse grid. Here, $\rho(j)$ consists only of j itself, together with its left and right neighbors. For each coarse node, we take the information on $\rho(j)$ from M_h and B_h , following the strategy explained in Section [3.1](#), to define each example. In summary, we adopt the following steps:

1. Given a domain $[a, b] \subset \mathbb{R}$, define a uniform mesh with exactly N elements, i.e., with a step-size $h = \frac{b-a}{N}$.
2. Perturb randomly the $N - 1$ internal nodes: we sample a uniform distribution in $[l, r] \subset (-\frac{h}{2}, \frac{h}{2})$, obtaining a collection $\{v_j\}_{j=1}^{N-1}$ of values, one for each node j . Then, we shift these nodes applying $j = j + v_j$.
3. Split the nodes in coarse and fine.
4. For each coarse node j , extract data related to $\rho(j)$ from M_h (features) and from B_h (target).
5. When enough records are generated, increase N .
6. Restart from 1.

Given that each node has the same patch-size, the generation and extraction processes work fine. We will see for the two-dimensional case how to deal with nodes with different patch-sizes.

3.1.3 Two-dimensional Records

In the NNs context, each record must have the same number of features and target, since this kind of models have a fixed dimension in input and output. Therefore, dealing with two-dimensional meshes becomes more difficult when we need to build a training set. Let us consider again Figure 3.2; there is not some restriction on the patch-size of the nodes.

As starting point, we consider grids where the nodes have a fixed patch-dimension. We will focus on dealing with different patch-sizes in Section 3.3.1. The left picture in Figure 3.3 shows a mesh with nodes which have a fixed patch-

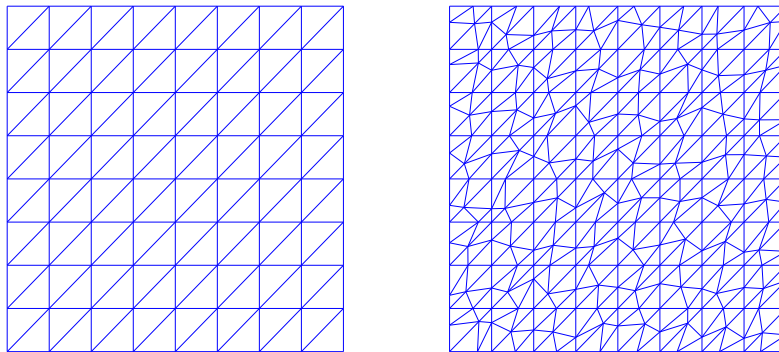


Figure 3.3. Example of 2D mesh used to extract training set records: on the left we have the fixed coarse mesh, on the right the result of the refinement and shifting procedure.

size. Of course, the boundaries have less neighbors, but for the moment we focus on internal nodes. Using this kind of structure, each record has always the same number of features and target. The training set is then constructed extracting information from grids that follow this rule.

Contrary to the 1D case, here, to create the multilevel scenario we generate the coarse mesh and proceed by refinement. After refining, we shift the nodes randomly, creating elements of different shapes. The result is shown in Figure 3.3 on the right.

A crucial point is to find a correct distribution of data in the training set, in terms of magnitude of the values. Since the NN should not prefer some examples - thus, some classes - over others, we need an even filling of the dataset. As a first approach, we relate the concept of class to the refinement procedure. When we need to pass to a new class, we refine our mesh, obtaining a new coarse mesh

rom which we start to generate new examples. Applying a training algorithm on these data results in a poor ability of approximation and a large prediction error, making a NN model unfit to work in a MG setting. The refinement procedure makes the number of elements to scale by a factor of 2 (bisection) or 4 (mid-point). In terms of domain of training examples this means that the initial classes of records, i.e., C_N with N small, are close to each other, while their distance grows when N increases. This turns out to produce an uneven training set, without a good balance in terms of data distribution. For this reason we need a linear increasing in the number of elements. If the classes of examples are evenly spaced in terms of domain, the network does not prefer some classes over others. Therefore, a second approach changes the definition of class, independent of the concept of refinement: we start from a number N , and we create a mesh having exactly N elements; once we extract enough records, we pass to the next class, increasing N by a constant factor K , and create new grids with $N + K$ elements. In details, the generation procedure works as follows:

1. Given a domain $[a, b] \times [c, d] \subset \mathbb{R}^2$, define a coarse mesh with exactly N elements. To reach this result, we start by considering rectangular elements instead of triangles. Therefore, we need a mesh with $N_r = \frac{N}{2}$ elements. We factorize N_r as a product of two integer values, h_e and v_e , that indicate the number of edges in which we uniformly divide $[a, b]$ and $[c, d]$, respectively. Using these values, we create our grid, and then divide each rectangle into two triangles.
2. Refine the obtained mesh, using mid-point refinement or longest-edge bisection. The refinement strategy must be the same for each class C_N .
3. Perturb each new node by a random quantity, sampled from a uniform distribution in $[l, r] \subset (0, 1)$: let us consider, for example, the longest-edge bisection procedure, applied on the N_r edges, each joining the points (x_1^j, y_1^j) and (x_2^j, y_2^j) , $j = 1, \dots, N_r$. The normal application of bisection would result in a new point for each edge j , namely $p_j = (\frac{1}{2}(x_1^j + x_2^j), \frac{1}{2}(y_1^j + y_2^j))$. Here, we consider the collection $\{v_j\}_{j=1}^{N_r}$ of random generated value and define the new point as

$$p_j = (v_j(x_1^j + x_2^j), (1 - v_j)(y_1^j + y_2^j)).$$

This strategy produces a series of coarse grids as the ones in Figure [3.4](#), where we start with $N = 32$ (on the left) and we choose $K = 32$. Following this simple procedure, the resulting training set is effectively unbiased and with a good distribution of the examples. A learning algorithm applied on these data produces

the expected good approximation.

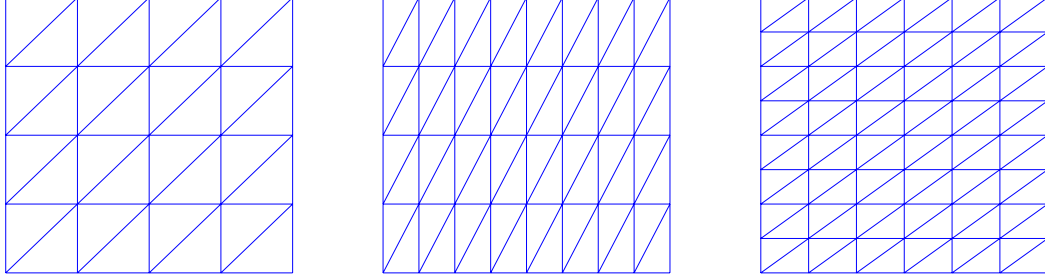


Figure 3.4. Changing class of meshes: on the left we start with 32 elements, and then we change class two times, with $K = 32$. This produces a grid of 64 elements (center) and then a mesh of 96 elements.

To conclude the description of the training set creation, we report in Figure 3.5 an activity diagram to better present how we create the scenarios and extract the data we need for the model to learn. The black dot represents the starting point; the diamonds represent conditional statements.

3.1.4 Methodology Details

This section is dedicated to present the details of our method. In particular, we discuss the motivations in defining *features*, *target*, the geometries considered and additional information that could be useful to produce a working model.

Definition of *features* and *target*: Our goal is to produce a transfer operator applicable in a multigrid contest. If we extract the *target* directly from the transfer operator, i.e. the L^2 -projection, the neural network is not able to learn. We know that for computing the transfer operator, we multiply the inverse of M_h (or its lumped version) with B_h . Therefore, we decide to predict B_h and only then we compute the actual transfer operator.

Regarding the *features*, the choice is between A_h and M_h , the operators we usually obtain after a FE discretization. Using only the stiffness matrix, the trained NN results in low accuracy and a poor property of approximate the components of B_h . Selecting M_h in input makes the model work. We find the reason in the definition of two operators: the stiffness matrix has gradient information, while the mass matrix has direct information about the geometry. Since the coupling operator is computed through the quadrature on intersections between elements, there exist a relation between M_h and our *target*. Lastly, we select values coming

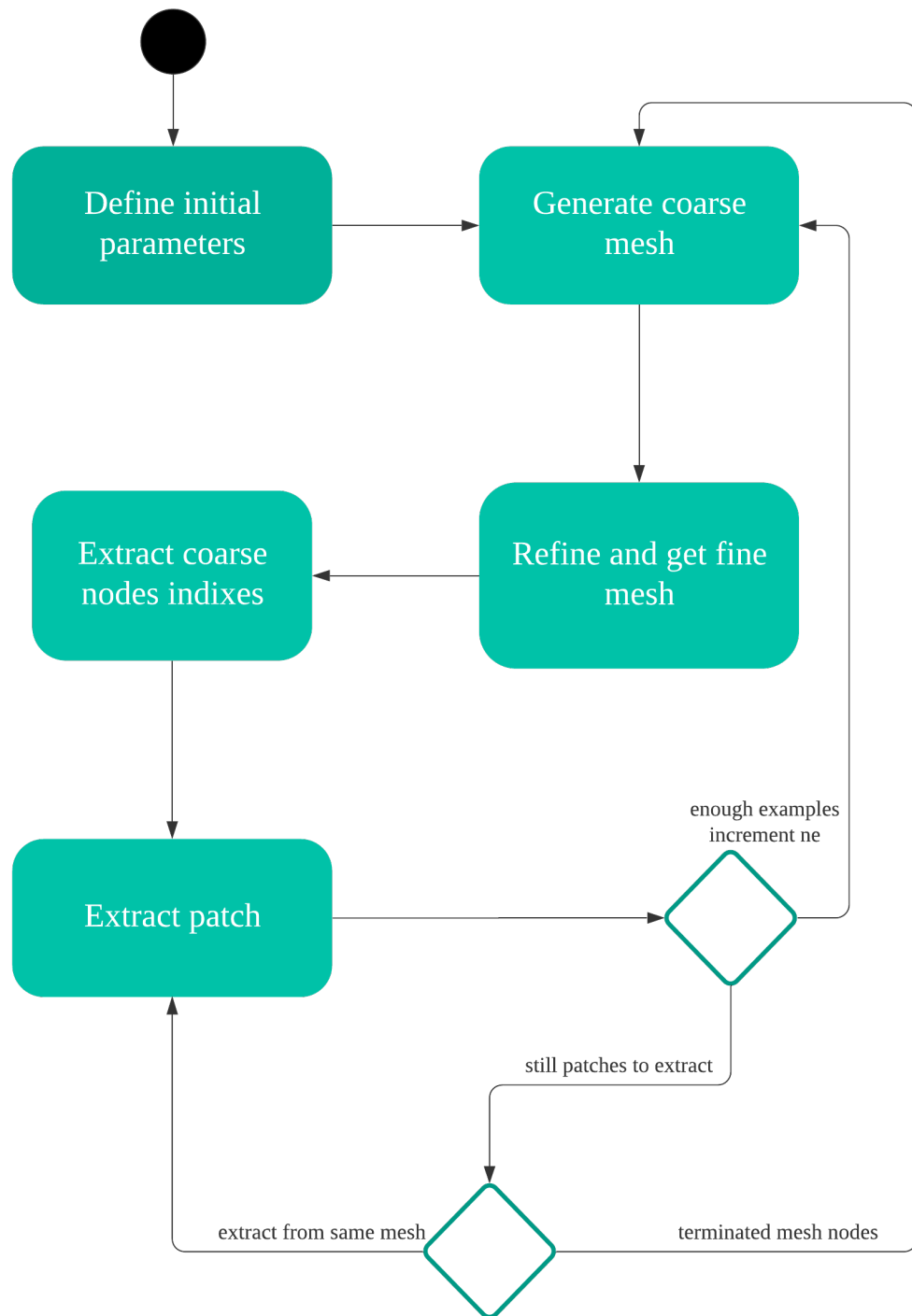


Figure 3.5. Flow chart of training set creation.

from both mass and stiffness matrix. This solution allow our model to better deal with variable diffusion coefficients, since this information is inside A_h . More details on this implementation will follow in Section [3.3.3](#).

Relation fine-coarse: A NN model, as other ML supervised methods, aims to approximate a function. Thus, we need to fill the training set with couples (*features*, *target*) that provide this kind of relation. As first approach, to generate different scenarios, we define fine and coarse mesh independently. This strategy produces uncorrelated *features* and *target*, from which the neural network is not able to learn. Therefore, we define a second procedure for creating the several scenarios: we start from a coarse (or fine) grid and proceed by refinement (or coarsening), to always have a relation between the two meshes.

Patch learning: In order to predict parts of B_h and then obtain the actual transfer operator, we consider several possibilities. A simple attempt is to consider only rows of B_h as *target*. Regarding the input, we consider the rows of M_h separately, without any distinction between fine and coarse nodes. This procedure does not have the necessity of paying attention on the filling procedure, since for each prediction we fill a single row in B_h . The creation of the training set is easier, but the results after the model optimization are not promising, providing a really slow convergence in a multigrid context. The main reason behind the poor results is the lack of information about the coarse nodes: predicting rows puts the focus on the fine nodes, considering only a part of the coarse nodes data in each *target*. Therefore, to give more focus to the coarse nodes, we redefine the *target* as a column in B_h . Consequently, we consider patches of M_h to take into account sets of fine nodes at a time. These grouped fine nodes are the ones transferring their information to the target coarse node, identified by that specific column of B_h . The accuracy of the NN gets better, leaving only issues when the operator is applied in a multigrid algorithm. Since we want our model to preserve some properties of the transfer operator, we enlarge the parts of B_h considered, including also rows of the coupling operator. These new data have a main role in the optimization of our custom loss function, as we will explain in the next section. This addition makes the predictions suitable for the multigrid method.

Choice of the classes: A main point is defining how to pass from a class to the next, and how many examples for each class we need. If we choose a small K and a large amount of example for each class, we have a lot of examples close to each other; this can make the network preferring a subset of examples over others, causing overfitting and decreasing its generalization capability. On the other hand, if K is large and we have only a few examples for each class, the model

is not able to approximate the target function, resulting in a poor accuracy in the predictions. Therefore, the choice of these values is crucial in making the network suitable as a solution for multigrid solvers. In 2D, for example, we find a working choice of parameters, selecting $K = 32$ and a thousand of examples for each class.

High correlation between records: When we randomly move the nodes in creating a mesh we transform all the elements so that their shapes differ. The values of the *features* highly depend on how the random component is handled. A small movement of the node is allowed, but does not create much difference between the mesh elements. A large movement can create elements with very acute angles (from 2D ahead); this can cause a large overlap between basis functions of neighboring nodes, resulting in similar values inside M_h . In terms of training set this translates in having different records with high correlated features, causing possible overfitting during the model optimization. Figure 3.6 shows a 1D example of overlapping between basis function. Considering nodes $j-1$ and j , the overlap between their relative basis function, φ_{j-1} and φ_j , is the underlined gray area.

Predict distances: In studying how to improve the model accuracy, we also take into account the distances between the nodes in a single patch. For each coarse nodes j , we take information of $\rho(j)$ together with the Euclidean distance from j to each of its neighbors. We aim, with these new data, to add information about the positions of the neighbors of j and therefore to create an ordering between the nodes in the patch. The neural network shows no real improvement, making these new data useless to reach our goal.

Ordering of features: In 1D, the actual order of the nodes follows the enumer-

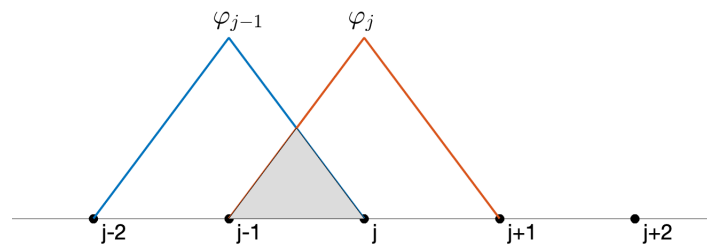


Figure 3.6. Overlapping between 1D basis functions: we indicate it with the gray area.

ation on the mesh, and we always know the position of a neighbor in respect to a specific coarse node. From the two-dimensional scenario, we can have different enumeration for the same mesh, and we lose the ordering information of the nodes in a patch. Following this argument, we try to put more “stability” in each record, fixing an ordering for each node based on its position inside the patch. Once obtained an ordering, we can also remove duplicates inside the *features* field: it can happen that two nodes k and w , belonging to the same patch of node j , are neighbors themselves; thus, when we extract entries from M_h we create a duplicate, taking both $M_h(k, w)$ and $M_h(w, k)$. The resulting training set is less complex and its application allows a reduction in the number of parameters of the network, although we spent more time in preparing the input. The predictions accuracy is not affected by these changes, which is why we prefer to ignore the ordering issue, making the patch extraction faster.

3.2 Model Training

A NN model optimizes its parameters during the training phase. This optimization process aims to reduce the prediction error and get better results when we test the model on never seen examples. To optimize the parameters we select a suitable loss function to be minimized. Since our goal is to produce continuous values, performing a regression, we employ the MSE cost function. This results in predictions close to the actual targets, but when we compose the full transfer operator and we employ it as part of a MG method, we obtain slow convergence. Therefore, the loss function needs to consider also properties of the operator, in order to make the model closer to the correct approximation that we want to reach. Regularization helps us overcome this issue, reducing the hypothesis space and allowing the NN to choose a better function to approximate. We consider penalty terms related to the domain knowledge. These terms force constraints during the training phase, in order to respect properties that the transfer operator must satisfy.

3.2.1 Regularization

During the construction of the training set we take, for each coarse node j , information of $\rho(j)$ from M_h and B_h . We use these data to define the penalty terms. Let us denote M_j, B_j and Q_j to be the j th rows of M_h, B_h and Q , respectively. We define the j th predicted and actual rows of Q as

$$Q_j^{\text{pred}} = \frac{1}{\sum M_j} B_j^{\text{pred}}, \quad Q_j^{\text{true}} = \frac{1}{\sum M_j} B_j^{\text{true}}. \quad (3.1)$$

We know that the predicted transfer operator must preserve constants (more details can be found in [25]). Furthermore, given that we can compute the rows of Q for each record during the training phase, we also require that Q_j^{pred} is as close as possible to Q_j^{true} . Hence, we consider the following penalty terms to specialize our loss function:

$$\|Q_j^{\text{pred}} \cdot \mathbb{1}_H - \mathbb{1}_h\|_2, \quad \|Q_j^{\text{pred}} - Q_j^{\text{true}}\|_2, \quad (3.2)$$

where $\|\cdot\|_2$ denotes the Euclidean norm, $\mathbb{1}_H$ and $\mathbb{1}_h$ the all-ones vectors of dimensions N_H and N_h , respectively. We then define, for all the nodes $k \in \rho(j)$,

$$p_k = \frac{1}{\alpha} \|Q_k^{\text{pred}} \cdot \mathbb{1}_H - \mathbb{1}_h\|_2 + \frac{1}{\beta} \|Q_k^{\text{pred}} - Q_k^{\text{true}}\|_2, \quad (3.3)$$

where $0 < \alpha, \beta < 1$.

We can now define the new loss function as

$$\mathcal{L}(y_{\text{true}}, y_{\text{pred}}) = \text{MSE}(y_{\text{true}}, y_{\text{pred}}) + \sum_k p_k. \quad (3.4)$$

3.2.2 Model Details

We use a classic splitting for our dataset: 20% for test and 80% for training, where the latter is divided again in 20% for the validation set and the remaining for the training phase. To have reliable results for a wide range of problem dimension, we need around 250 thousands examples. We try both the two approaches in the training set pre-processing explained in Section 2.1.3; given that the standardization brings better accuracy in the predictions, our dataset is then rescaled using this strategy. For both one- and two-dimensional models we adopt Adam as optimizer, and we initialize the weights in our models using a normal distribution. Regarding the architecture, we will discuss it in details in Chapter 4, testing the accuracy provided by various model architecture. Finally, considering regularization strategies, good results has been noticed by putting a dropout layer just before the output layer. Furthermore, we apply the early stopping procedure, waiting ten epochs before ending the training process.

3.2.3 Hyperparameters Tuning

The creation and employment of the training set during a model optimization, provides the optimal values of the network parameters. They are part of the equation to be minimized in order to obtain better accuracy in approximating the target. On the contrary, hyperparameters are set manually, to help the model optimization during the training phase. There are two main important hyperparameters: the learning rate and the regularization parameters. We also consider the batch size and a *reduce_lr* variable. The latter makes the learning rate smaller when the learning stagnates after a certain amount of epochs. The most common strategies for tuning these hyperparameters are *Grid search* and *Random search*. The first use a preset list of values for each hyperparameter and the model is evaluated for every combination of the values in this list. In *Random search*, we don't provide a preset list of hyperparameters. Instead, we give the searcher a distribution for each hyperparameter. The search algorithm tries random combinations of values to find the best one. For large sets of hyperparameters, random search is a lot more efficient.

We adopt the *Grid search* strategy, testing different combinations of hyperparameters. The results make us choose the best model setting to obtain better accuracy in the predictions, and they will be presented in Chapter 4.

3.3 NMG Implementation

In the above sections, we focused on making the problem local, extracting only parts of the global problem, and put them in a dataset. When we extract patches from M_h and B_h we follow a certain order to fill the features and target fields. Let us consider two neighbors of a coarse node j , namely k and w . If we consider first node k , then its information goes in the features field before w , and the same must happen for the target field. During the construction of the training set this is easily respected.

Once we have a working model, able to predict parts of B_h , we also take into account where the prediction must be put inside the full coupling operator. Thus, additionally to the patches information, we fill a structure to save the location of the predicted values. We are able to find these indexes looking directly in M_h , enlarging the node patch and intersecting it with the coarse node list, as explained in Section 3.1, i.e., we need to look at $\rho_c(j)$.

Following the indexes structure we compose the global B_h , from which we get the transfer operator Q . Doing so, we are able to use it inside the MG method,

Algorithm 4: Extract Patches

1 Find coarse nodes	$C = \text{coarsening}(M_h)$
2 Prepare empty lists	$\text{patches} = [], \text{indices} = []$
3 For each c_{node} in C :	
4 Initialize arrays to -1	$\rho(i), \text{idx}_i = \text{array}(-1)$
5 Extract patch info and indices	$n_{info}, n_{idx} = \text{extract}(M_h, c_{node})$
6 Insert patch into array	$\rho(i).\text{insert}(n_{info})$
7 Insert indices into array	$\text{idx}_i.\text{insert}(n_{idx})$
8 Append full patch to list	$\text{patches}.\text{append}(\rho(i))$
9 Append full indices to list	$\text{indices}.\text{append}(\text{idx}_i)$

and study the resulting convergence.

In order to create the multilevel hierarchy we recursively apply this procedure to define coarser problems: at each level we predict the transfer operator Q and we apply it in the *Galerkin operator*, i.e., $M_H = Q^\top M_h Q$. M_H will be a new input for our NN, to produce a transfer operator to pass information to an even coarser grid.

Algorithm 4 presents a simple procedure for extracting the patches information together with the positions to respect when we fill the coupling operator. For each coarse node we prepare two arrays containing only -1 . In 1D they are entirely overridden, since we always have the same patch-size to deal with. On the contrary, in 2D, based on the neighborhood of the coarse nodes, we could end up with -1 values after the procedure. This indicates missing information, which must be addressed to obtain full patches for each examined node. In order to obtain this missing data, as a first attempt we extend the mesh, as we explain more in details in the next Section. Furthermore, in order to create a more flexible procedure independent on the mesh structure, we implement a virtual extension method, presented in Section 3.3.2.

3.3.1 Mesh Extension

As we mentioned before, in 2D we deal with different patch-sizes. Therefore we need a strategy to overcome the non-fixed structure of these node patches. Let us start by considering the boundary nodes of simple meshes, like the one in Figure 3.3 on the left. To define the records for the training phase, we consider the internal nodes, so that each patch has the same dimension. Thus, when we create the NN input for nodes near the boundary, we need to add information.

The first solution requires an extension of the mesh on the boundaries. The

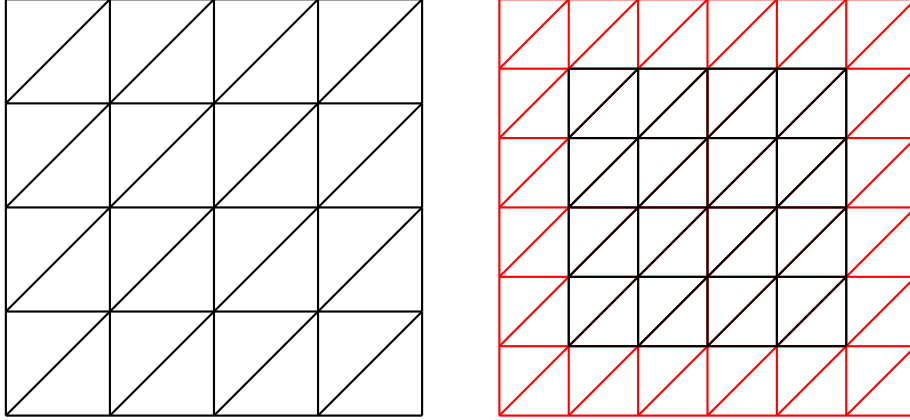


Figure 3.7. Mesh extension procedure: on the left we have the starting mesh, on the right we have the result of the extension, where in red we denote the new nodes and elements.

idea is to make the boundary nodes of the original mesh, internal nodes in a new extended grid. The procedure is simple: for each boundary node, we create a new point outside the mesh. Figure 3.7 shows the result of this procedure, where in black we denote the original mesh, and in red the extended components.

During the prediction of the coupling operator B_h we need to consider both these meshes. A first procedure runs on the original mesh, and for those nodes with smaller patches, i.e., near the boundaries, we remain with missing data in the *features* list. This “holes” are then filled by a second run of the same procedure on the extended mesh. Algorithm 5 present this strategy: after the mesh extension, we replace the values -1 that we initialized in the previous algorithm, and we put the newly computed values from the extended mass matrix.

Now that we have a full patch structure, we ask the NN model to compute the predictions. Since we stored in the list *indices* the positions that must be filled by the predictions, we simply ignore those related to the extended mesh, and finally obtain the coupling operator B_h .

3.3.2 Virtual Extension

Mesh extension allows our NN model to work also for boundary nodes. The transfer operator can be composed and applied in creating the multilevel hierarchy. However, this strategy presents two problems to be addressed. First, it requires heavy computations: when the problem dimension increases, we need to create several new nodes and elements for extending the grid, and compute a

Algorithm 5: Mesh Extension

1 Extract patches from M_h	$patches = \text{extract_patches}(M_h)$
2 Find boundary nodes	$border = \text{boundary}(mesh)$
3 For each border edge:	
4 Create new rectangular element el	
5 Divide el into two triangles and add them to $mesh_{ext}$	
6 Compute mass matrix M_h^{ext} on extended mesh	
7 Extract patches from M_h^{ext}	$patches_{ext} = \text{extract_patches}(M_h^{ext})$
8 For each $patch$ in $patches$:	
9 Replace values -1 with $patches_{ext}$ values	

bigger mass matrix, as well as completing the patches extraction. Furthermore, it only works for patches that have less neighbors than the fixed input of the NN. Let P be the fixed patch-size of the nodes used for training the model. When we extend the mesh, we can only address those nodes having less than P neighbors.

Let us start by addressing the first issue. We need to fill the missing part of the *features* field, with data somehow connected to the values inside the mass matrix. Let us consider a structured mesh obtained by mid-point refinement. We compare patches of an internal node i with a patch of a boundary node j . We notice that the values $M_h(i, w), w \in \rho(i)$, result from scaling the value $M_h(i, i)$ by a factor f_1 . The same happens for the values $M_h(j, z), z \in \rho(j)$, with respect to $M_h(j, j)$, but with a different scaling factor $f_2 < f_1$. Thus, depending on the patch-size of the considered node, we need a certain scaling factor f^* . Let us remark that the unknown values we are looking for are just fictitious mass matrix entries needed to fill all the *features* field. The NN output strictly related to them are ignored when assembling B_h . Therefore, even if using the factor f^* to scale $M_h(j, j)$ results in approximations, we obtain reasonable values, of the same order of magnitude of $\rho(j)$, and that is enough to make our method work.

Let us move to the second point. Let us consider an unstructured mesh, obtained through Delaunay triangulation. There could be a node w with $P + 1$ neighbors. We need a strategy to obtain the components of B_h considering all the $P + 1$ neighbors. The model input accepts nodes with maximum P neighbors. Therefore, we proceed by considering two subsets of the neighborhood of w , both with P neighbors. More precisely, we exclude a node in each subset, and define two different NN input for node w . Once we have the two predictions, the components related to the nodes in common between the subsets are averaged, while the ones present in only one subset are entered normally.

Algorithm 6: Virtual Extension

```

1 For each  $j$  in  $C$ :
2   IF  $|\text{neigh}(j)| < P$  :
3     Find scaling factors            $f^* = \text{scaling}(|\text{neigh}(j)|)$ 
4     Create new nodes                $\text{node}_v = \text{create}(M_h(j, j), f^*)$ 
5   IF  $|\text{neigh}(j)| > P$  :
6     Find nodes to add               $\text{additional} = |\text{neigh}(j)| - P$ 
7     For  $i$  in  $\{1, \dots, \text{additional}\}$ :
8       Extract neighborhood subset   $\text{set} = \text{neigh}(j)(i : i + P)$ 
9       Extract patch related to  $sub$ 

```

Finally, we consider a generic node with $P + Z$ neighbors. In this case, we define $Z + 1$ subsets which give rise to $Z + 1$ predictions, which we put in B_h in the same way as for simpler $P + 1$ case. Algorithm 6 reports the main steps to reach this goal. For each coarse node we verify if we need to “add” or “remove” nodes. We did not report the part related to a correct patch dimension, that stays the same as explained in the previous algorithms.

Figure 3.8 shows an activity diagram describing the main behavior of our method: after an initial pre-processing in which we either load or define a problem, we proceed in creating the multilevel hierarchy and then to the solver phase.

3.3.3 Adding information from A_h

In 3.1.4 we underline the poor results obtained by a given model, if trained with data coming from the stiffness matrix. Although we solve the issue focusing on the mass matrix, we do not consider information about the diffusion of a specific problem, since it is specified in the equation and not related to the geometry. Let us consider Equation (1.17); given different non-constant functions $k(\mathbf{x})$, our method will always produce the same transfer operator, since it ignores the diffusion coefficients. Therefore, we define a new model, following the same workflow illustrated along this chapter, such that it would include also data from the stiffness matrix. The training set creation now takes into account also values from A_h , following the same indexes as for M_h . The implementation of Neural MG considers also patches in A_h when constructing the model input. We consider this matrix before the application of the boundary conditions when we extract its values. In Chapter 4 we report a test using this last defined NN, showing the convergence when applied on a problem with a variable diffusion coefficient.

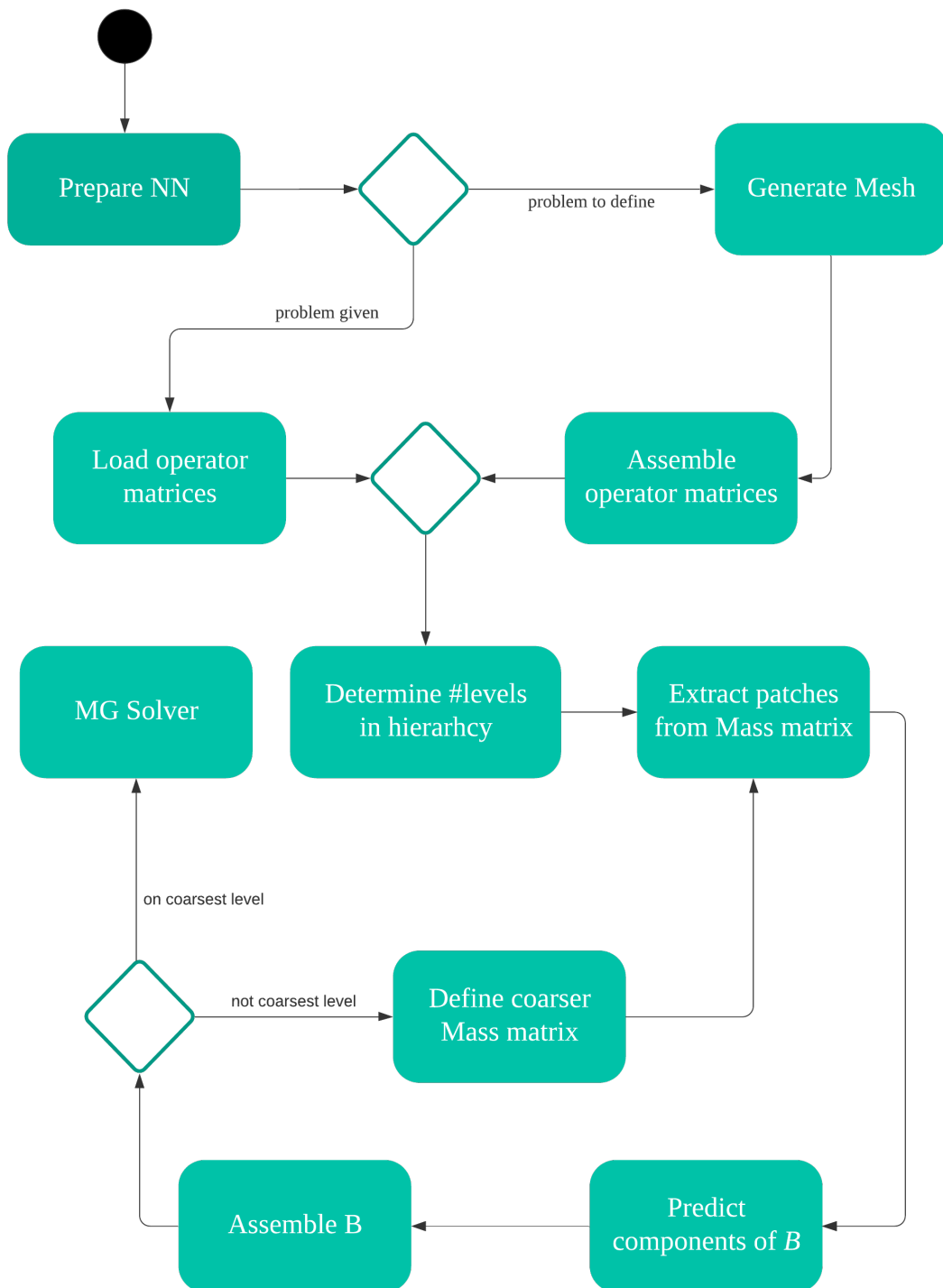


Figure 3.8. Flow chart of Neural MG.

Chapter 4

Numerical Experiments

In this chapter, we present the performance of the NMG method, introduced in the previous chapter. First, we give an overview of the machines employed for running the numerical experiments, and the libraries which supported the implementation of our method. Then, we present the actual numerical tests. We distinguish the results between one- and two-dimensional scenarios. For both of them, we start with a preliminary study, comparing the prediction errors given by different architectures, in terms of number of layers and neurons. Subsequently, we study the predictions while changing the dataset employed during the training phase: we consider a badly distributed one, showing the incapacity of the network to approximate the desired function, and then an evenly filled set, showing the related goodness when tested on examples not used during the training. Finally, we test the model in a MG context, using the NN to provide transfer operators actually employed to restrict and prolong data between grids. We start testing the model on one-dimensional scenarios, reporting its behavior when applied on a set of problems with increasing dimension, and then against the SGMG method, focusing on the CPU time spent to assemble the transfer operators. Then, we try out the two grid and the more general MG method. Therefore, we test NMG in two-dimensional scenarios. Here, we have to deal with more complex data and models. Thus, we start by finding the correct values of hyperparameters. After the test on CPU time and convergence using problems of increasing dimension we consider two kinds of comparison for the NMG method: against SGMG for structured grids, and against AMG for unstructured grids. Regarding the latter experiments, we consider the Poisson equation as model problem. We are aware that algebraic systems arising from more complex problems will require an accurate choice of the smoother. However, since this Thesis focuses on comparing the application of different transfer operators,

we do not perform a pre-processing for choosing the best smoothing strategy, and we select the standard Gauss-Seidel for each MG method tested. For structured grids, we also report a test using a model which takes into accounts also the stiffness matrix, as explained in the previous chapter. We show a convergence comparison between AMG, SGMG, NMG and NMG with the addition of stiffness data, when applied on a Poisson problem with variable diffusion coefficient.

4.1 Technical Specifications

In this section, we describe the hardware used to carry out the various experiments and computations required to define and test our method.

4.1.1 MacBook Pro

Since running on a cluster is expensive, when there is no need to have produce a large amount of data, we use a MacBook Pro. The MacBook Pro has the following hardware: a CPU Intel Core i9 2.9 GHz with 6 cores, and a RAM of 16 GB, 2400 MHz DDR4. We employ this machine for tests on training set creation, correlation studies between records, comparison of different dataset distribution and small problems simulation to have a first look on the state of our method.

4.1.2 Cluster

The heavy computations are carried out on the cluster of the Institute of Computational Science¹. We distinguish two type of computations: the training of the NNs and the tests of MG methods using the predicted transfer operators. In addition, a small portion of tests is addressed to the hyperparameter tuning, testing several combinations of values. The nodes considered for grid search and MG tests are the ones named Xeon Phi nodes, which in total include 8 different usable nodes. The detailed hardware installed on these nodes is:

- **CPU:** 2 x Intel Xeon E5-2650 v3 @ 2.30GHz, 20 (2 x 10) cores
- **RAM:** 128GB DDR4 @ 2133MHz
- **HDD:** 1 x 1TB SATA 6Gb

¹<https://intranet.ics.usi.ch/HPC>

The training phase is carried out on nodes specific for GPU computations. Their hardware is equivalent to the one above, with in advance two kind of GPU (depending on the specific node selected):

- **GPU:** 1 x NVIDIA GeForce GTX 1080, Founders Edition 8GB GDDR5, 2560 CUDA cores
- **GPU:** 2 x NVIDIA GeForce GTX 1080 Ti, Titan 11GB GDDR5X, 3584 CUDA cores

4.2 Libraries and Coding

We present here, without entering too much in details, the main libraries we employ to implement the Neural MG method. Our framework² is written in *Python3*. It handles the numerical solver part of the method: it implements smoother procedures, mesh definitions, direct and iterative solvers, and the three MG methods we use in this context: GMG, SGMG and NMG, which are coded directly in the library. The main libraries we use as support are *NumPy* [40] for arrays and matrices math, and *SciPy* [88] for the sparse objects. When we need to use AMG we make use of *pyAMG* [64], a framework of Algebraic Multigrid solvers for Python. NMG, as well as the training algorithms outside of our library, uses *Tensorflow* [1, 2, 31] to interface with NN models. This framework provides a simple and transparent control over the networks, allowing to customize loss functions, optimizations methods and gradient computations. Regarding the creation of the training set, we adopt *MATLAB*. With it, we implement a small suite for FE discretization: it allows mesh definition, coarsening, refinement, and assembling of operator matrices and right-hand side. For details on FEM implementations, one can refer to [33, 53, 56]. Furthermore, this library provides functions to compute intersections between grids, coupling operators and the actual L^2 -projection. Together, these modules allow a simple implementation for creating different training set, based on parameters given as input. Lastly, we generate the unstructured grids employed for the tests with *FreeFem++* [43], a *C++* PDE solver for non-linear multi-physics systems in 1D, 2D, 3D and 3D border domains.

4.3 Numerical Results

In this section, we show the main results obtained with our strategy, both in 1D and 2D scenarios. Since the 1D model has less parameters and it is simpler to

²https://bitbucket.org/ctomasi/learn_multigrid/src/master/

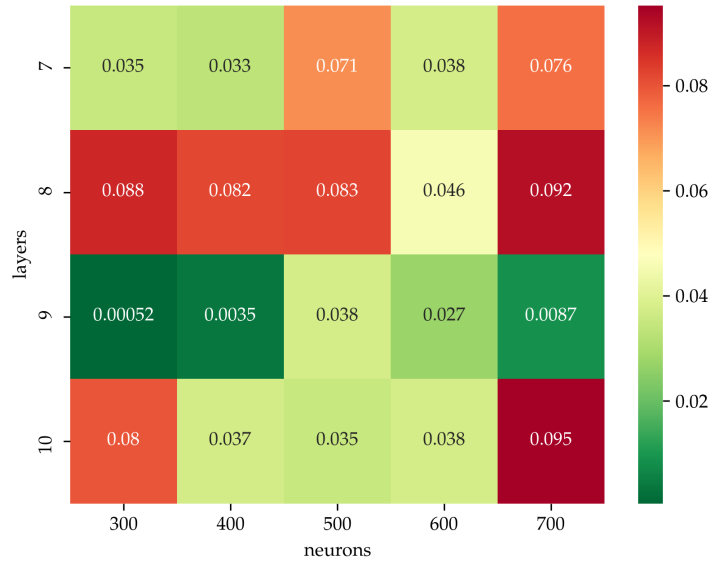


Figure 4.1. Heatmap of 1D model: comparison between different architectures to find the best solution in terms of validation error.

handle, we put only a few important tests to show the success of the proposed method. Presenting the two-dimensional part, we first show the result of the grid search algorithm for hyperparameter tuning, in order to define a starting point for the NNs construction. Then, we report a comparison between NMG and other existing MG methods, showing their differences in computational time and convergence. Regarding the latter we distinguish between structured and unstructured mesh. For the structured ones, we test against the SGMG method. For unstructured grids, we test NMG against AMG. All the plots regarding convergence and CPU time comparisons adopt the logarithmic scale for the y-axis.

4.3.1 1D

First, we present the preliminary results which guarantee the applicability of our method. They include the variability of the network loss value while changing the structure of the model, and the prediction accuracy while changing the tested training set. Figure 4.1 reports a comparison between different network architectures, showing the validation loss value for each of them. We consider an amount of layers between seven and ten, and a number of neurons for each layer between 300 and 700. Each value at the intersections (n^*, l^*) represents the prediction error obtained with a model with l^* layers, each one with n^* neurons. Smaller values mean better predictions for the network, and are colored in

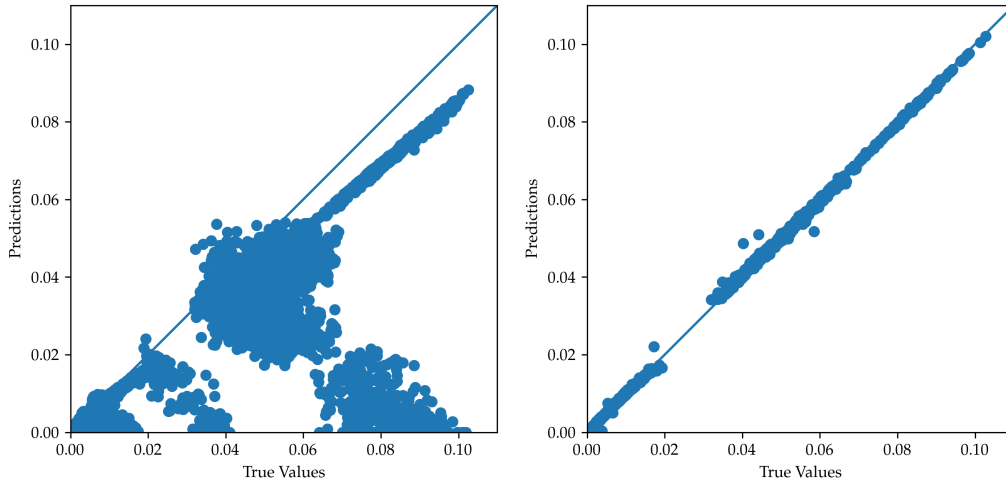


Figure 4.2. Prediction accuracy 1D NN: on the left a network trained on a baldly defined dataset; on the right the training is on a correctly distributed dataset.

green; the bigger error values are showed in red.

On the other hands, in terms of prediction accuracy, we present in Figure 4.2 a comparison between two models: one defined on a poorly constructed training set, with an uneven amount of examples for each class (left), and the other referring to a correctly defined dataset (right). This shows how crucial is the dataset definition. In both plots, each point is defined by a couple given by the actual value together with the predicted value. Of course, more points are close to the diagonal and lower is the prediction error.

Finally, we pass to the convergence of the method. In order to consider our procedure suitable for an application in general cases, we need it to reach convergence in the same number of iterations, independently from the degrees of freedom of a specific problem. Therefore, considering only the NMG method, we keep increasing the dimension of a test problem to solve with our strategy, retrieving the iterations needed to reach a residual below $1e-09$. The pre- and post-smoothing phases in the MG algorithm employ Gauss-Seidel as iterative procedure, and we use three steps before and after the coarse grid correction. We adopt as test problem the Poisson equation, i.e., $-\Delta u = f$, which we discretize and then solve. Figure 4.3 shows the number of iterations needed for NMG to reach convergence, increasing the problem size. The convergence rate is between 0.006 and 0.01, where with convergence rate we mean the ratio with whom the norm of the residual decreases from one iteration to the next, i.e., $\frac{\|r_{i+1}\|_2}{\|r_i\|_2}$.

The second convergence test shows the computational time needed to create

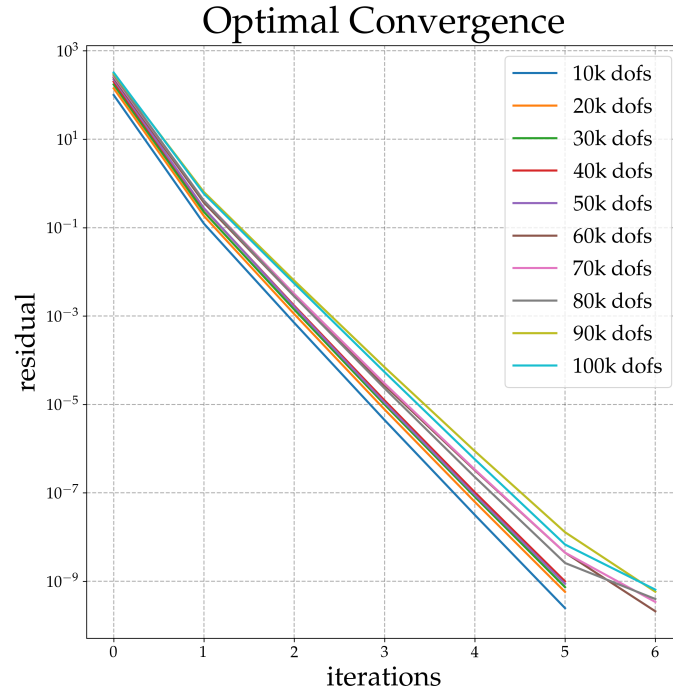


Figure 4.3. Convergence of NMG in 1D, showing the number of iterations needed for solving test problems of increasing dimension.

the transfer operator. We compare the time needed to predict the transfer operator against the computation of the actual L^2 -projection, increasing the problem dimension. Figure 4.4 (left) shows this comparison. For our method we take into account patches extraction, predictions, assembly of B_h and computation of the operator. For computing the actual L^2 -projection, we consider the intersections definition between fine and coarse meshes, triangulation for each intersecting polygon and numerical integration. We see that our method is faster, providing the transfer operator in a time of smaller order of magnitude than the one computed through intersections and quadrature. Regarding our strategy, different components can be optimized. First, each patch extraction can be carried out independently from each other. Therefore, we can make this part parallel, saving computational time. Another aspect to consider is the redundancy of the extraction procedure of some nodes: coarse node not far from each other could have patches with a non-empty intersection. In other terms, given two fine nodes, they could belong to two different patches, thus we only need to consider them only one time when we extract their information from the mass matrix.

Finally, we test our method against the SGMG method on an example of around 100 thousands degrees of freedom. First we compare the two strate-

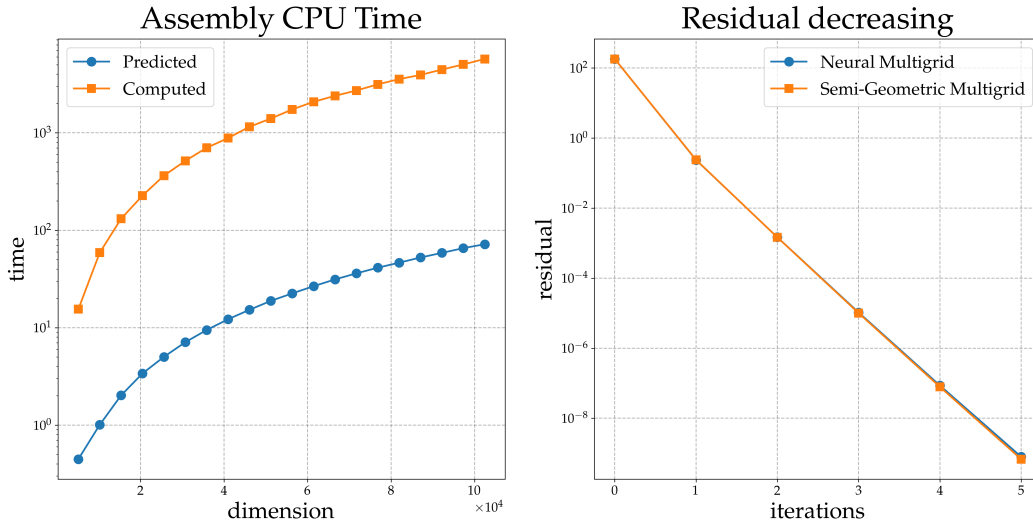


Figure 4.4. Comparison between NMG and SGMG: on the left CPU time comparison between the prediction and the computation of the transfer operator; on the right convergence comparison of NMG against SGMG using two grids method, applied on a 1D problem with 100 thousands degrees of freedom. Both plots have the y-axis in logarithmic scale.

gies using only a two-grid method. Figure 4.4 (right) shows this comparison. We see the two lines representing the residual decreasing of both methods being almost the same. This result allows us to pass to the general MG method with hierarchies consisting of several levels of approximation. The fine test problem considered is the same as in the two-grid method. We report the results of the methods on a hierarchy of ten levels, only in tabular form (Table 4.1), since both the residual norms are too similar to visualize their comparison in a plot.

Summarizing, we see our method resulting in a good compromise between computational time and convergence. Even in a simple one-dimensional scenario, using a NN for predicting the transfer operator instead of actual computing it provides a great speed up. While the classic computation is basic on the geometry, and therefore requires intersection research and quadrature, our approach only depends on the problem dimension, since it produces a prediction for each coarse node identified. Therefore, the presented algorithm for predicting transfer operators scales linearly with the degrees of freedom of the specific problem taken into account.

Regarding the convergence, we have good results for both the simple two-grid method and the general MG method. This makes us confident of pursuing the study of this methodology in the two dimensional scenario.

Iteration	NMG	SGMG
0	2.26e02	2.26e02
1	2.95e-01	3.02e-01
2	2.05e-03	2.14e-03
3	1.97e-05	2.00e-05
4	2.89e-07	2.89e-07
5	5.43e-09	5.42e-09
6	2.20e-10	2.01e-10

Table 4.1. Convergence comparison of NMG against SGMG using ten grids, applied on a 1D problem with around 100 thousands degrees of freedom.

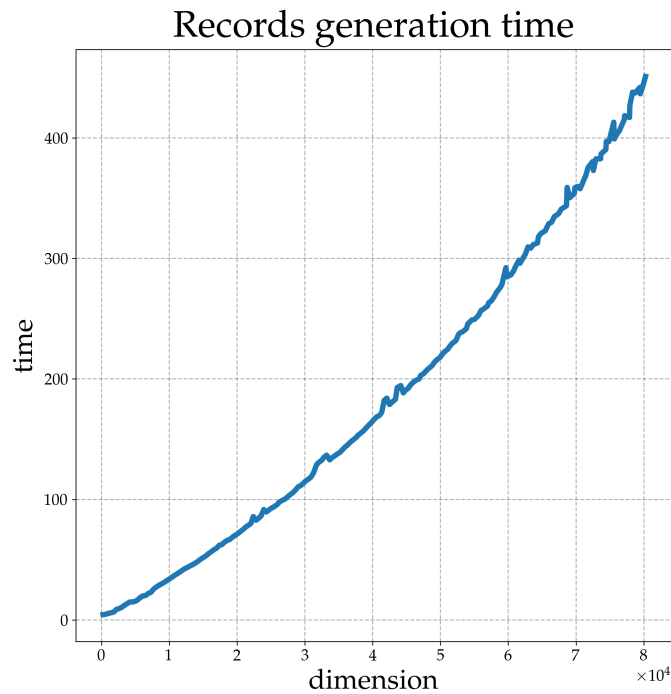


Figure 4.5. CPU time spent in generating a specific training set, showing how the time grows, when the degrees of freedom increase.

4.3.2 2D

Before discussing the actual numerical tests, in Figure [4.5](#) we report the time spent in constructing a working training set, showing how the required time increases when we increase the degrees of freedom of the problem. Each point in

the plot represents the time spent for generating a certain class of examples. We start with the first classes of examples, taking around four seconds for each, until we reach the bigger dimensions, that take around seven minutes to be generated. In this specific training set, we consider 624 different classes, starting from 81 dofs until 80 thousands, for a total generation time of 32.2 hours.

Regarding the actual numerical results, as in 1D, we first report the preliminary tests in constructing the best distributed dataset, and the NN architecture that provide better prediction accuracy. Studying these characteristics, we do not aim to already find the best model, but only a starting point. In Figure 4.6 we show the heatmap related to the error, comparing the different network architectures. We maintain the same coloring as before, showing in green the better results, and in red the bigger errors. Compared to the previous case, here we have a range of layers between 11 and 16, and the number of neurons between 300 and 1200. Thus, we need a more complex architecture to reach suitable results in terms of accuracy. This is shown also inside the heatmap, with error values generally bigger than the one in 4.1. Since we add information, given that the patch-sizes of nodes are bigger in 2D, it is natural to obtain worse results.

Figure 4.7 shows a comparison between two 2D training sets used during a training algorithm: on the left we consider an uneven dataset with badly distributed data; on the right we have a correctly distributed training set. It is clear from these plots, that also in 2D a good data definition is crucial to generate

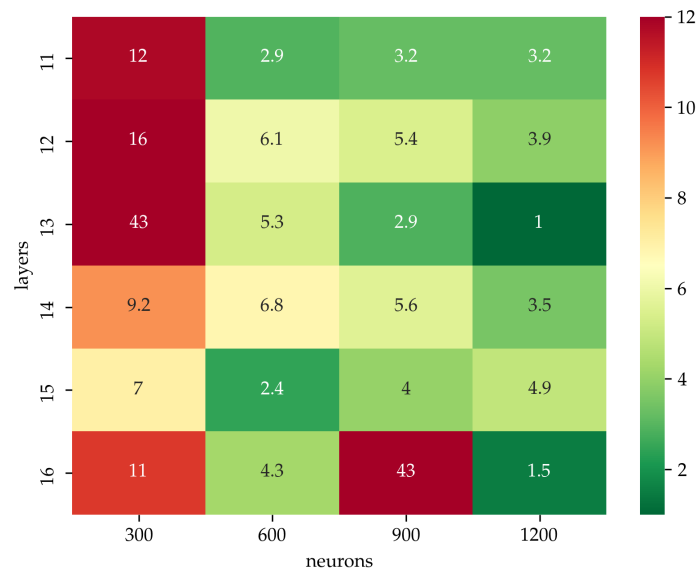


Figure 4.6. Heatmap of 2D model: comparison between different architectures to find the best solution in terms of validation error.

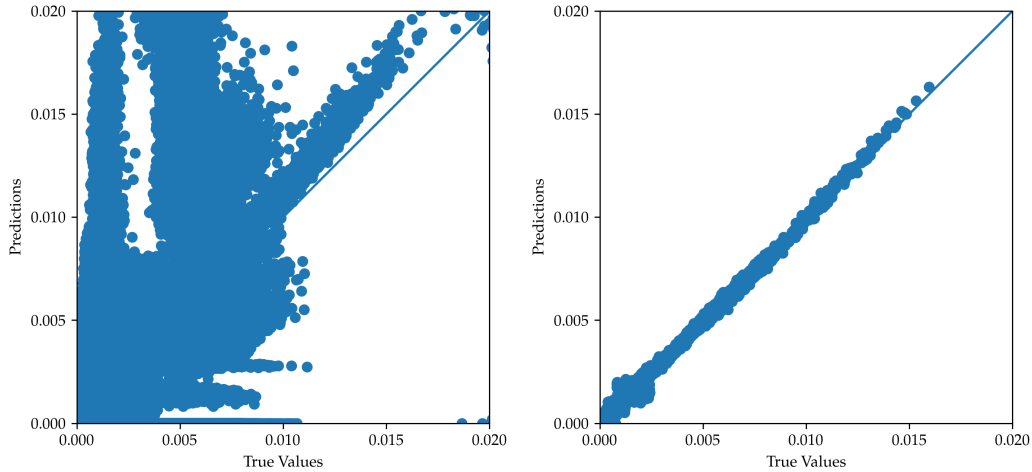


Figure 4.7. Prediction accuracy 2D NN: on the left a network trained on a baldy defined dataset; on the right the training is on a correctly distributed dataset.

working models.

After having selected the best architecture and training set, we pass to adjust the various network parameters. As mentioned at the beginning of Section 4.3, in 2D we deal with several more data when defining the NN model. Therefore, we perform a parameters tuning using the grid search algorithm. The considered parameters are α, β , i.e., the penalty terms in the custom loss function, the initial learning rate of the optimization algorithm, the *batch-size*, amount of random taken examples used for a single training step (or epoch), and the *reduce_lr* parameter. The latter helps the training algorithm when the optimization is stucked on a certain result for a fixed number of epochs. It is a scalar that is multiplied with the learning rate, to obtain a new smaller learning rate with whom continue the optimization of the model. Table 4.2 shows the main results of this tuning. We do not report the entire list of combination tried out, which is of great length, but only a few combinations to show the sensitivity of the model, when we change these parameters. Since some combinations give the same results in term of validation loss and validation MSE, we group them together, using the interval notation $[a, b]$.

From Table 4.2, it is clear that the best solution is the one reported on the last line, with a validation loss of 0.97. Therefore, we adopt that combination of values for setting the model hyperparameters during the learning algorithm. Once the setting is complete, we compare again the different architectures. The result of the comparison is presented as a heatmap in Figure 4.8. We changed

α	β	lr	reduce_lr	batch_size	val_loss	val_mse
1	1e-03	1e-04	[0.5, 0.9]	[32, 128]	1313.83	1.4e-07
1	1e-03	1e-05	0.5	128	3.39	1.06e-09
1	1e-03	1e-05	0.7	128	17.07	1.5e-09
1	1e-03	1e-04	0.9	64	3.38	5.34e-10
1	1e-03	1e-05	0.5	32	1.03	1.68e-10
1	1e-03	1e-05	0.9	32	5.36	4.42e-10
1	1e-03	1e-04	[0.5, 0.9]	[32, 128]	7600	1.67e-08
1	1e-03	1e-05	0.7	128	25.72	1.62e-09
1	1e-03	1e-05	0.5	64	5.37	9.73e-10
1	1e-03	1e-05	0.5	32	10.56	1.58-e09
1e-02	1e-03	1e-04	[0.5, 0.9]	[32, 128]	1313.8	1.4e07
1e-02	1e-03	1e-05	0.5	128	1.76	2.03e-10
1e-02	1e-03	1e-05	0.9	128	4.03	4.9e-10
1e-02	1e-03	1e-05	0.5	64	0.97	1.42e-10

Table 4.2. Hyperparameter Tuning

the ranges of layers and neurons number: we do not consider anymore models with 300 neurons per layer, since employing that amount results in poor approximation. Furthermore, we test a larger number of layers, now going from 11 to 19. We can see that with a correct choice of hyperparameters, the overall performance of the various architecture gets better.

The NN that gives the best approximations consists of 17 layers and 1200 neurons per layer. It results in good predictions, but it brings a big complexity, with around 20 millions of parameters. Therefore, we prefer to employ lesser complicated models, even if they provide an higher prediction error.

Finally, having a working model, we test the convergence of the two-dimensional Neural MG method. First, we study the convergence rate, increasing the size of a specific problem. The pre- and post-smoothing phases consist of three steps of Gauss-Seidel. The test problem considered is again the Poisson equation. Each fine level mesh in this test is a structured mesh, for simplicity in the comparison. The results are presented in Figure 4.9. We see that our method reaches convergence in 6 - 7 iterations, independently of the degrees of freedom. Reaching bigger dimensions, we notice that the convergence rate increases, passing from 0.03 for small problems, to 0.08 for bigger ones. This is due to the range of examples considered during the training phase. If we generate two-grid scenario examples until one hundred thousands dofs, the tested

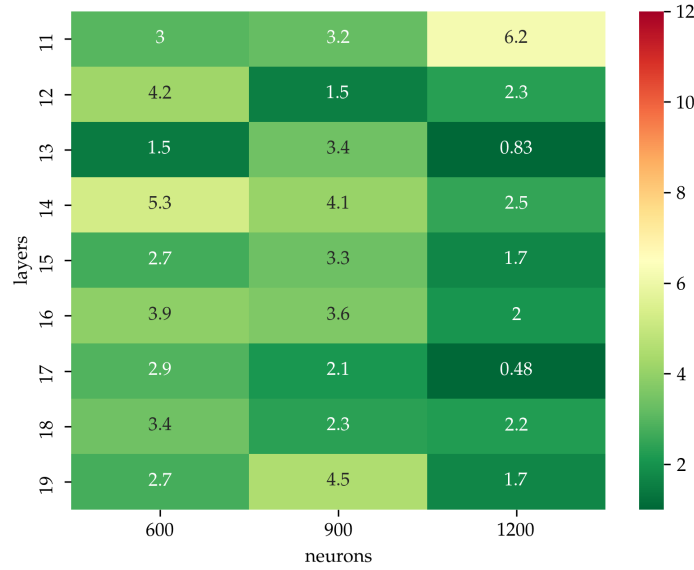


Figure 4.8. Heatmap of 2D model: comparison between different architectures after having set the model hyperparameters, to find the best solution in terms of validation error.

examples of a bigger dimension will result in a worse convergence rate, since they provide values outside the training set domain. Nevertheless, the method provides a small number of iteration even in bigger problems.

Then, we test our method against the other MG methods. First, we compare the CPU time spent in providing the transfer operators. We compare the computation of the operator predicted against the L^2 -projection calculation. The results are shown in Figure 4.10 (bottom). Also in 2D, we see our method to gain an order of magnitude. Furthermore, the rate between the two CPU times is not constant, and grows bigger while increasing the problem dimension.

Subsequently, we compare NMG against SGMG in examples of structured meshes. Figure 4.11 shows an example of the kind of grids considered. The last part of the Chapter is dedicated to show the results of our method applied on unstructured meshes and tested against the AMG method; more in details, we configure AMG to work with the Ruge-Stüben method, using the LU decomposition provided by *SciPy* as direct method on the coarsest grid. For the unstructured grid tests, we report grid pictures with a few elements, to show their structure, while for each actual test we indicate the actual amount of dofs used.

Structured We compare the methods first on a two-grid scenario, and then applied on a hierarchy of five levels. The result presented are obtained on a

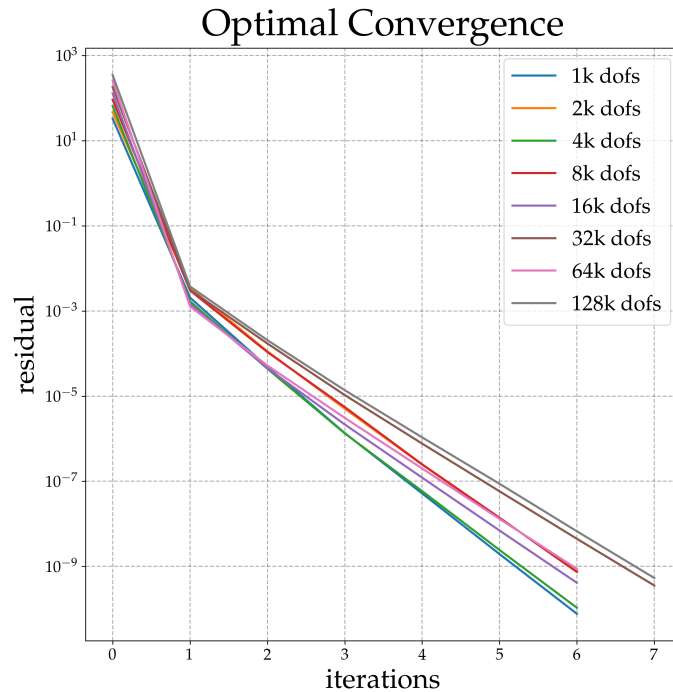


Figure 4.9. Convergence of NMG in 2D, showing the number of iterations needed for solving test problems of increasing dimension.

test problem of 100'651 degrees of freedom. The kind of meshes considered are structured square grids as the one in Figure 4.11. Figure 4.10 shows the comparison of the two methods. On the top-left picture, we have the two-grids convergence. Our method perform well with only two levels: it reaches convergence in five iterations, one less than the SGMG method. On the top-right plot, we have the convergence on a hierarchy of five meshes. Here, the NMG method reaches convergence in six iterations, with a similar residual decreasing as the SGMG method.

Figure 4.12 presents a last test on the same structured grid. Our model problem is a Poisson equation with a variable diffusion coefficient. In this specific case $k(\mathbf{x}) = x^2 + y^2$, where $\mathbf{x} = [x, y]$. The plot reports Neural MG and SGMG reaching convergence in ten iterations, while the other two methods are faster. AMG only takes four steps, NGM with the addition of stiffness information six steps, showing an improvement from the initial NMG.

Given the promising results, we pass to unstructured grids examples, where the virtual extension algorithm presented in Section 3.3.2 is employed to address the problem of the different patch-sizes of nodes.

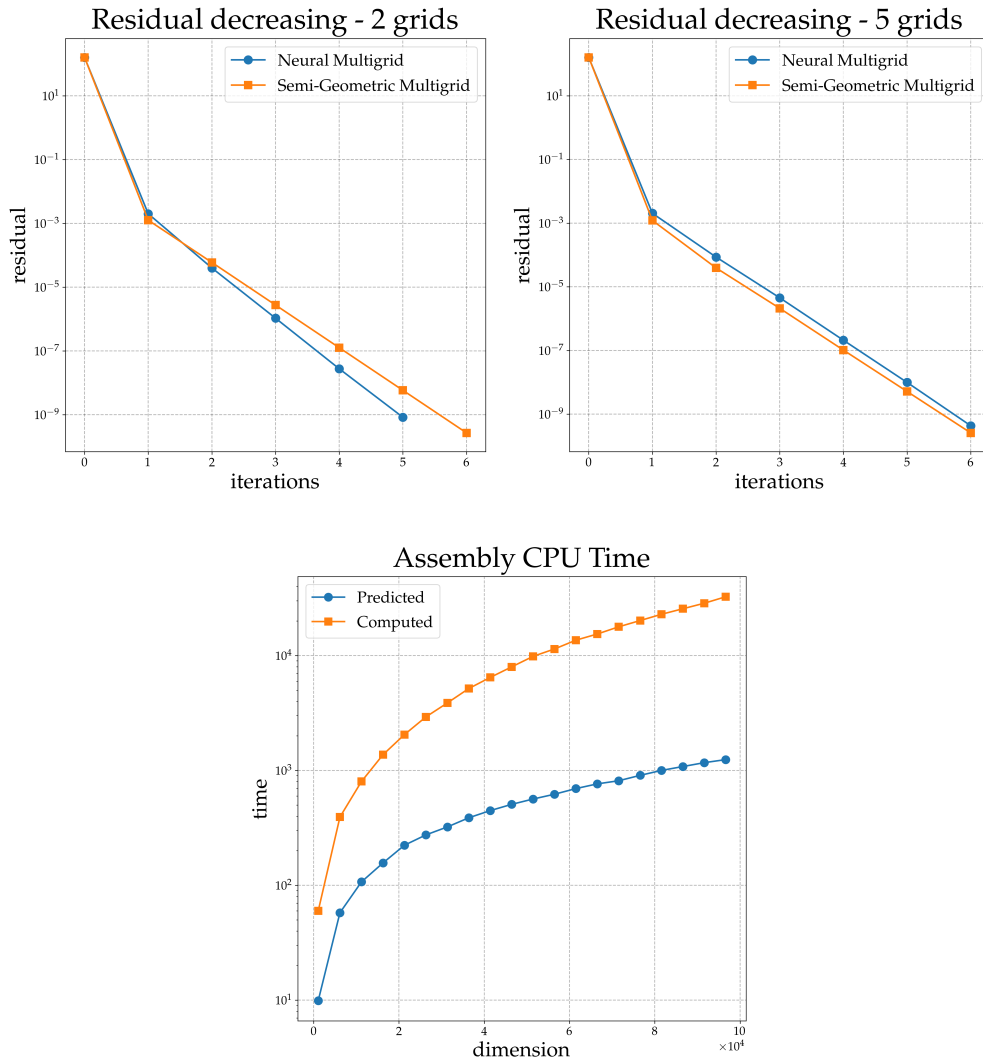


Figure 4.10. Comparison between NMG and SGMG: convergence comparison using two grids method (top-left) and a hierarchy of five levels (top-right), applied on a 2D structured grid with 100'651 degrees of freedom. On the bottom, CPU time comparison between prediction and computation of the transfer operator, increasing the dofs.

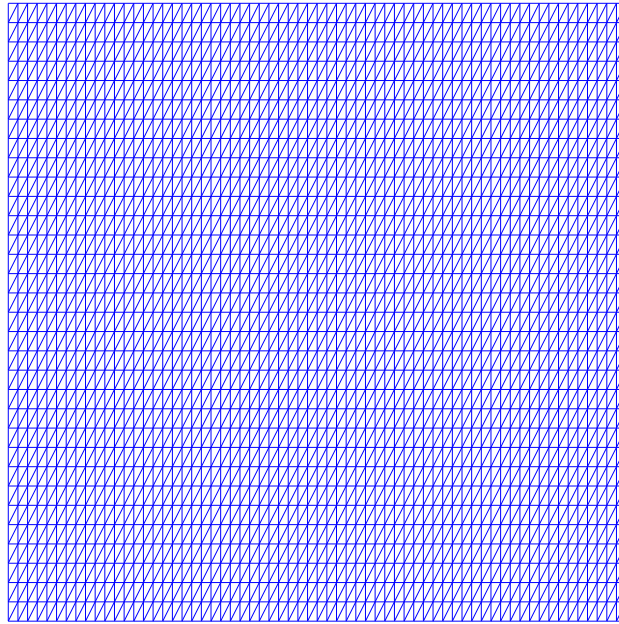


Figure 4.11. Structured square mesh.

Unstructured We consider different meshes of different complexity. The number of degrees of freedom is around 100 thousands. The grids used in the following tests are generated with *FreeFem++* [43]. We consider three examples of grid: a circle with 104'997 dofs, in Figure 4.13, a circle with holes with 107'934 dofs, in Figure 4.15 and a circle with a hole, obtained through an adaptive refinement procedure with 106'322 dofs, in Figure 4.17. For the three tests, we consider first a two-grid scenario and then a hierarchy of five levels. Let us start with the simple circle. We see the convergence comparison in Figure 4.14. On the left we have the convergence of the two-grid method. Our strategy gains a couple of iteration against AMG. Increasing the number of levels, we have on the right the comparison on five levels: NMG reaches convergence in 17 iterations, three steps more than before. On the contrary, AMG pass from 16 to 26 iterations.

Then, we test on the holed circle mesh. Again we test our NMG method against AMG. The resulting convergence comparison is shown in Figure 4.16, distinguishing between two-grid (left) and five levels (right). We can see a similar trend as in the previous example, with NMG reaching convergence in 15 and 16 iterations. Instead, the AMG method takes 17 iterations with two grid, and 34 using the five levels.

Concerning the last test, the convergence of the two methods is presented in

Figure 4.18. With this kind of mesh, our procedure reaches convergence in 14 and 19 iterations, when tested on a two and a five levels hierarchy, respectively. AMG takes 15 steps applying the two-grid method, and grows to 24 iterations for reaching convergence when applied on five levels.

In summary, we tested our methodology with several examples. The numerical results show the robustness of our method, both with problems of increasing degrees of freedom and also on hierarchy of increasing size. Especially in the scenario of the circle with holes, we only need a single additional iteration to reach convergence, passing from two to five levels, while AMG takes several more. Table 4.3 reports the convergence tests, where for each mesh, we report the number of iterations to reach convergence and the average of the convergence rate, considering the different hierarchy configurations. We do not consider here the variable diffusion coefficient problem, being it the only one solved with four different MG methods.

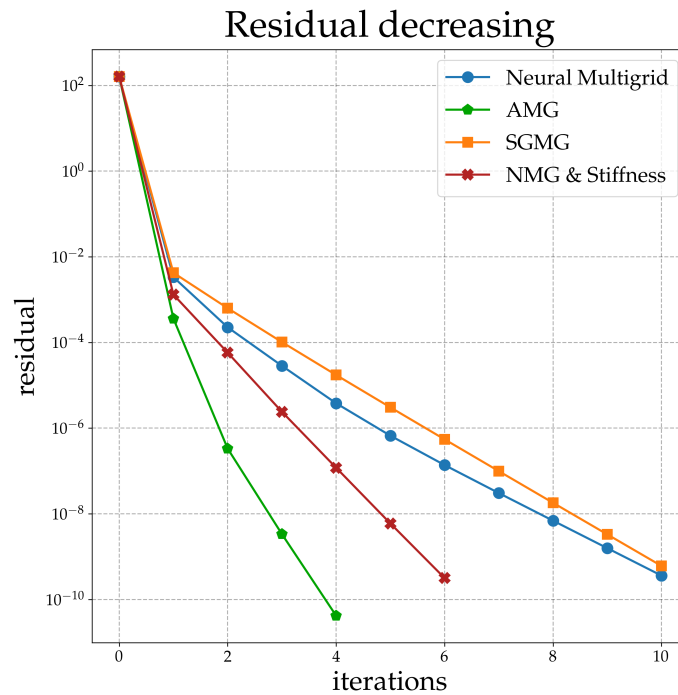


Figure 4.12. Convergence comparison of NMG, AMG, SGMG and NMG with the addition of data from the stiffness matrix, using two grids method, applied on a 2D structured grid with 100'651 degrees of freedom. The problem considers a variable diffusion coefficient.

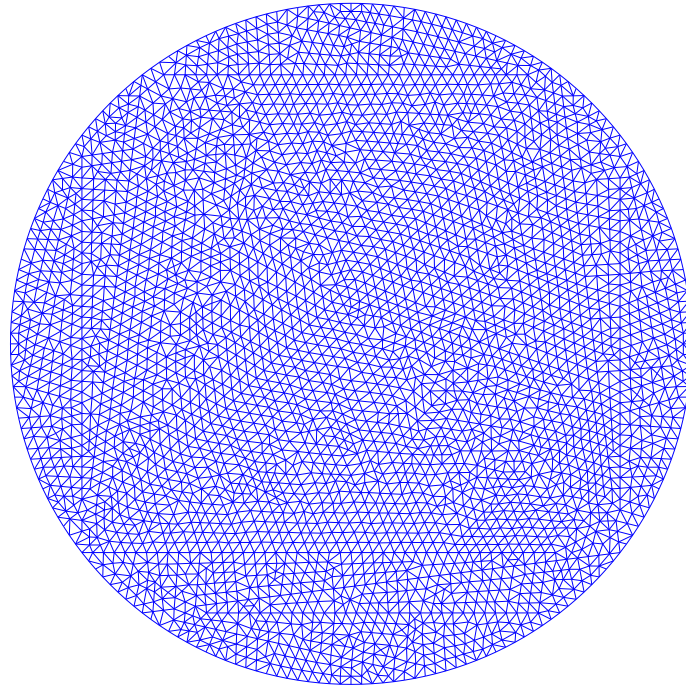
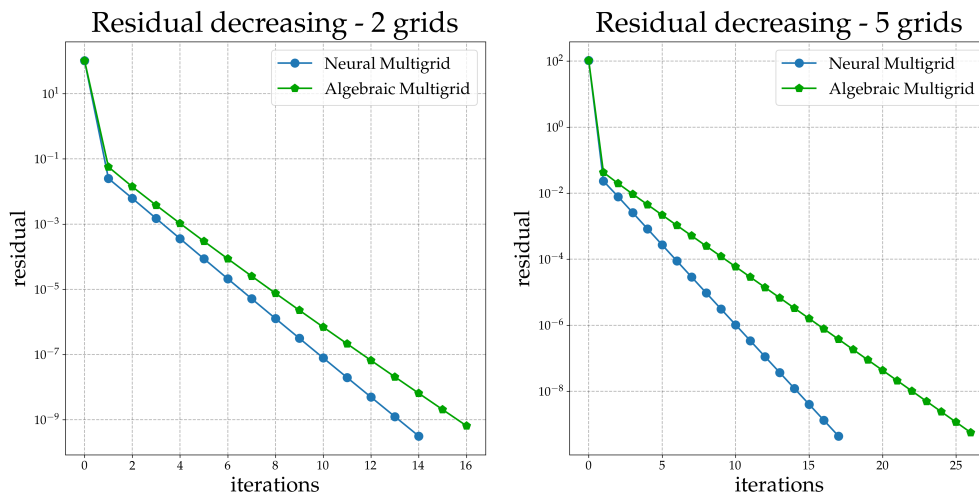


Figure 4.13. Unstructured mesh of a circle.

Figure 4.14. Convergence comparison of NMG against AMG using two grids method (left) and a hierarchy of five levels (right), applied on a 2D unstructured grid of a circle, with $104'997$ degrees of freedom.

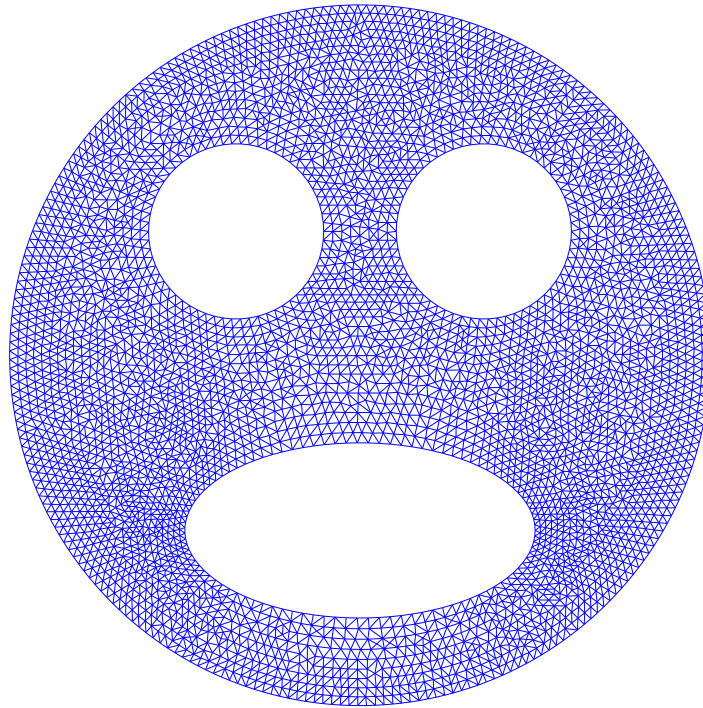


Figure 4.15. Unstructured mesh of a circle with circular holes.

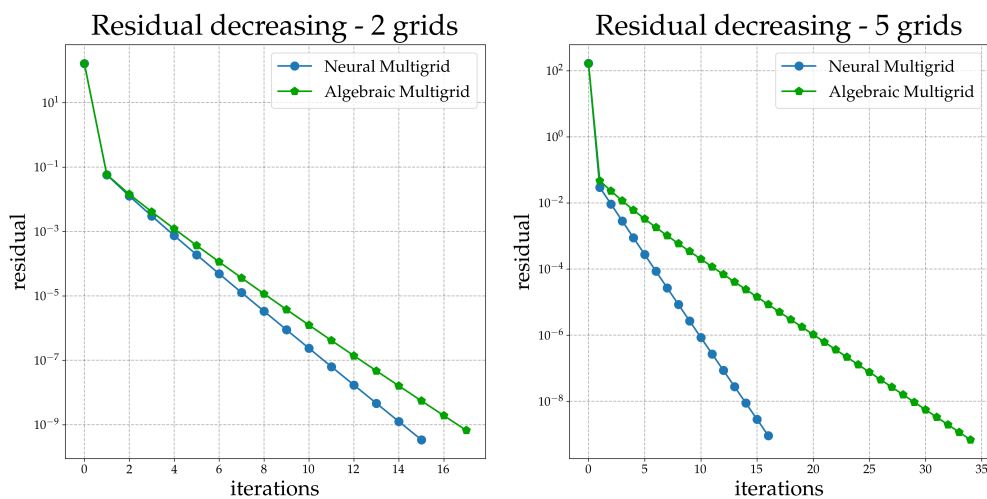


Figure 4.16. Convergence comparison of NMG against AMG using the two grids method (left) and a hierarchy of five levels (right), applied on a 2D unstructured grid of a circle with circular holes, with $107'934$ degrees of freedom

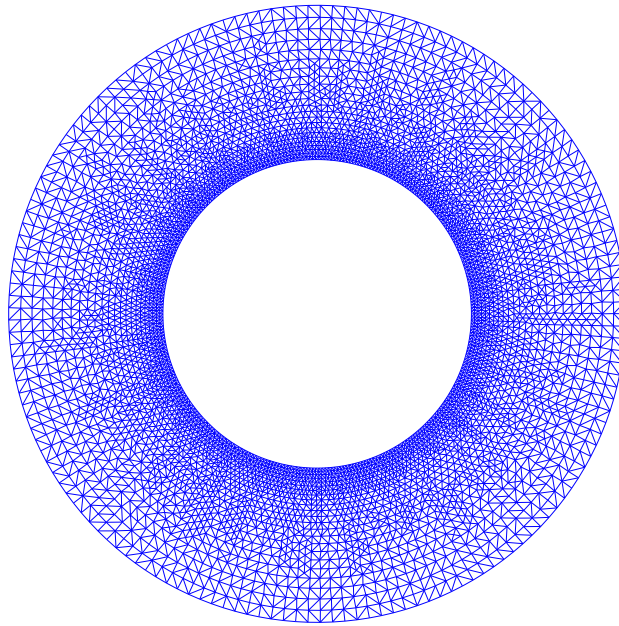


Figure 4.17. Unstructured mesh of a circle with a hole - Adaptive refinement.

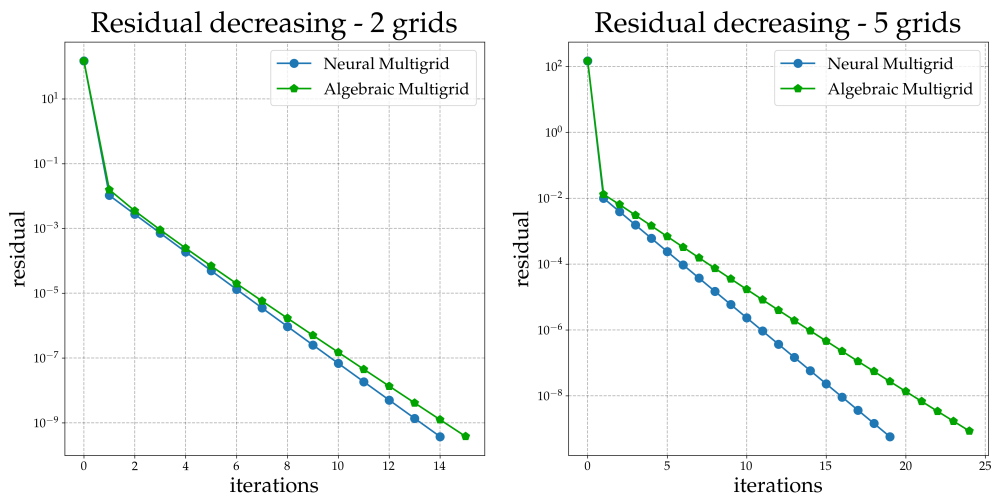


Figure 4.18. Convergence comparison of NMG against AMG using the two grids method (left) and a hierarchy of five levels (right), applied on a 2D unstructured grid of a circle with a hole, obtained through adaptive refinement, with 106'322 degrees of freedom


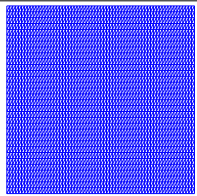
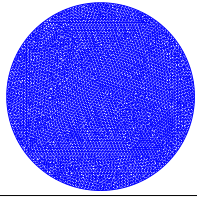
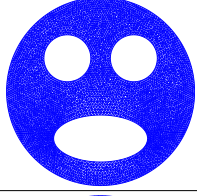
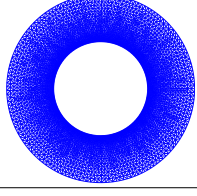
	Mesh type	#dofs	#levels	#iterations			Convergence Rate (AVG)		
				SGMG	NMG	AMG	SGMG	NMG	AMG
1D		100'000	2	5	5	-	0.0073	0.0077	-
			10	6	6	-	0.017	0.018	-
2D		100'651	2	6	5	-	0.046	0.025	-
			5	6	6	-	0.047	0.046	-
		104'997	2	-	14	16	-	0.24	0.29
			5	-	17	26	-	0.32	0.48
		107'034	2	-	15	17	-	0.26	0.32
			5	-	16	34	-	0.31	0.58
		106'322	2	-	14	15	-	0.27	0.28
			5	-	19	24	-	0.39	0.49

Table 4.3. Numerical experiments summary: for each dimension, for each kind of mesh and for each hierarchy tested, the table reports the number of iterations needed to solve the problem and the average of the convergence rates. These tests are related to the Poisson equation with a constant diffusion coefficient.

Conclusion

In this thesis, we presented the definition of a methodology for creating transfer operators with the aid of Neural Network models. The final aim of this work was to provide a working operator that would give rise to a fast convergence in reaching the solution of a specific problem, when employed in a Multigrid context. Therefore, starting from one-dimensional scenarios, we focused our attention on the construction of a training set containing domain information, together with the target operator. After several attempts, we defined the *features* as components extracted from the mass matrix M_h , while the *target* is part of the coupling operator B_h . Furthermore, each dataset example refers to a single coarse node patch. Using these ingredients, we were able to construct an initial working model, that showed good results when applied as part of a MG solver. This made us confident in generalizing the methodology to bigger dimensions. Thus, we investigated the two-dimensional scenarios, giving enough background to extend our method to a N -dimensional case. In 2D we had to address several issues, having more complicated geometries to taken into account. Initially we emulated the previous case when creating the various records for the training set. Unfortunately, this did not result in employable NN models. Changing the data distribution of the records solved the problem, and provided working models. The other main difference with the 1D scenario, was the variability of the patch-sizes, major problem we had to deal with, given that NNs allow only a fixed dimension for input and output. Implementing an extension algorithm, we dealt with those nodes having a smaller patch-size than the training examples. On the contrary, for those nodes with more neighbors than the ones considered for training, we needed to divide patches in subsets, and considering each of them distinctly, as explained in details in Section [3.3.2](#). At this point, our methodology provided a working model also in the two-dimensional case. Hence, we tested it on problems of increasing dimension, to check its generalization abilities. Obtained convergence in few iterations independently of the problem size, we focused on different geometries, to test the flexibility of the model. We considered first structured meshes, and then unstructured grids. We obtained promising

results in both cases, especially for the unstructured ones, with our Neural MG method reaching convergence before the Algebraic MG, providing a strong robustness when the number of levels in the multigrid hierarchies increased. A brief mention regards also to the results obtained on the variable diffusion problem, where the introduction of stiffness information inside the training algorithm allowed our method to improve its convergence.

Given the results, future works should be devoted on the definition of a model to be adopted in three-dimensional scenarios. The workflow follows the 2D case, with an obvious increasing complexity due to the larger neighborhoods of each node. Therefore, the difference would be only the computational time spent in creating examples and defining more complex neural networks, with a number of parameters suitable for those kind of records. Another aspect to consider is the increasing of the dataset domain, i.e., taking into account a wider range of scenarios, until a number of degrees of freedom acceptable for 3D problems. Additionally, the overall performance of the method could be improved further, investigating more on the data distribution of the examples and improving the architecture and the hyperparameters settings, considering more combinations of values during the preliminary tuning of the model. Furthermore, the inclusion of stiffness data in the model showed the possibility to consider also information related to a specific equation, in addition to the geometry. Thus, forthcoming works will follow this line of research, to create better model and improving the convergence of the NMG method.

Bibliography

- [1] M. Abadi, A. Agarwal, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [2] M. Abadi, P. Barham, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, Nov. 2016. USENIX Association.
- [3] A. F. Agarap. Deep Learning using Rectified Linear Units (ReLU). *arXiv preprint arXiv:1803.08375*, 2018.
- [4] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi. Learning Activation Functions to Improve Deep Neural Networks. *arXiv preprint arXiv:1412.6830*, 2014.
- [5] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a Convolutional Neural Network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017.
- [6] H. B. Barlow. Unsupervised learning. *Neural computation*, 1(3):295–311, 1989.
- [7] R. Beck. Graph-Based Algebraic Multigrid for Lagrange-Type Finite Elements on Simplicial Meshes. Technical Report SC-99-22, ZIB, Takustr. 7, 14195 Berlin, 1999.
- [8] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2006.
- [9] L. Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

-
- [10] L. Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.
- [11] D. Braess. *Finite elements: Theory, fast solvers, and applications in solid mechanics*. Cambridge University Press, 2007.
- [12] D. Braess and W. Hackbusch. A new convergence proof for the multigrid method including the V -cycle. *SIAM Journal on Numerical Analysis*, 20(5):967–975, 1983.
- [13] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390, 1977.
- [14] S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer, third edition, 2008.
- [15] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2000.
- [16] R. Caruana, S. Lawrence, and L. Giles. Overfitting in Neural Nets: Back-propagation, Conjugate Gradient, and Early Stopping. *Advances in neural information processing systems*, pages 402–408, 2001.
- [17] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke. The rise of deep learning in drug discovery. *Drug discovery today*, 23(6):1241–1250, 2018.
- [18] S.-W. Cheng, T. K. Dey, and J. R. Shewchuk. *Delaunay mesh generation*. Chapman & Hall/CRC Computer and Information Science Series. Chapman & Hall/CRC, Boca Raton, FL, 2013.
- [19] P. G. Ciarlet. *The finite element method for elliptic problems*, volume 40 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002.
- [20] A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, G. N. Miranda, and J. W. Ruge. Robustness and scalability of algebraic multigrid. volume 21, pages 1886–1908. 2000. Iterative methods for solving systems of algebraic equations (Copper Mountain, CO, 1998).

-
- [21] P. M. de Zeeuw. Matrix-dependent prolongations and restrictions in a black-box multigrid solver. *Journal of Computational and Applied Mathematics*, 33(1):1–27, 1990.
- [22] M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.
- [23] J. E. Dendy, Jr. Black box multigrid. *Journal of Computational Physics*, 48(3):366–386, 1982.
- [24] L. Deng, G. Hinton, and B. Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8599–8603. IEEE, 2013.
- [25] T. Dickopf. On multilevel methods based on non-nested meshes. *PhD Thesis*, 2010.
- [26] T. Dickopf and R. Krause. Weak information transfer between non-matching warped interfaces. In *Domain decomposition methods in science and engineering XVIII*, volume 70 of *Lect. Notes Comput. Sci. Eng.*, pages 283–290. Springer, 2009.
- [27] T. Dietterich. Overfitting and Undercomputing in Machine Learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.
- [28] S. Dreiseitl and L. Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5-6):352–359, 2002.
- [29] A. Eitel, J. T. Springenberg, L. Spinello, M. Riedmiller, and W. Burgard. Multimodal deep learning for robust RGB-D object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 681–687. IEEE, 2015.
- [30] M. J. Gander and C. Japhet. An Algorithm for Non-Matching Grid Projections with Linear Complexity. In *Domain decomposition methods in science and engineering XVIII*, pages 185–192. Springer, 2009.
- [31] A. Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O’Reilly Media, 2019.

-
- [32] Z. Ghahramani. Unsupervised Learning. In *Summer School on Machine Learning*, pages 72–112. Springer, 2003.
- [33] M. S. Gockenbach. *Understanding and implementing the finite element method*. Society for Industrial and Applied Mathematics (SIAM), 2006.
- [34] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2016.
- [35] H. Gottschalk and K. Kahl. Coarsening in algebraic multigrid using Gaussian processes. *Electron. Trans. Numer. Anal.*, 54:514–533, 2021.
- [36] D. Greenfeld, M. Galun, R. Basri, I. Yavneh, and R. Kimmel. Learning to Optimize Multigrid PDE Solvers. In *International Conference on Machine Learning*, pages 2415–2423. PMLR, 2019.
- [37] W. Hackbusch. On the Multi-Grid Method Applied to Difference Equations. *Computing*, 20(4):291–306, 1978.
- [38] W. Hackbusch. On the convergence of multi-grid iterations. In *Beiträge zur Numerischen Mathematik Band 9*, pages 213–239. 1981.
- [39] W. Hackbusch. *Multi-Grid Methods and Applications*, volume 4. Springer Science & Business Media, 2013.
- [40] C. R. Harris, K. J. Millman, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.
- [41] D. M. Hawkins. The Problem of Overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.
- [42] J. He and J. Xu. MgNet: a unified framework of multigrid and convolutional neural network. *Science China. Mathematics*, 62(7):1331–1354, 2019.
- [43] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.
- [44] R. Hecht-Nielsen. Theory of the Backpropagation Neural Network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.
- [45] A. K. Jain, J. Mao, and K. M. Mohiuddin. Artificial Neural Networks: A Tutorial. *Computer*, 29(3):31–44, 1996.

- [46] M. Karagiannopoulos, D. Anyfantis, S. Kotsiantis, and P. Pintelas. Feature Selection for Regression Problems. *Proceedings of the 8th Hellenic European Research on Computer Mathematics & its Applications, Athens, Greece, 2022*, 2007.
- [47] A. Katrutsa, T. Daulbaev, and I. Oseledets. Black-box learning of multi-grid parameters. *Journal of Computational and Applied Mathematics*, 368:112524, 12, 2020.
- [48] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [49] K. Kira and L. A. Rendell. A Practical Approach to Feature Selection. In *Machine learning proceedings 1992*, pages 249–256. Elsevier, 1992.
- [50] S. B. Kotsiantis. Supervised machine learning: a review of classification techniques. *Informatica. An International Journal of Computing and Informatics*, 31(3):249–268, 2007.
- [51] R. Krause and P. Zulian. A parallel approach to the variational transfer of discrete fields between arbitrarily distributed unstructured finite element meshes. *SIAM Journal on Scientific Computing*, 38(3):C307–C333, 2016.
- [52] A. Krogh and J. A. Hertz. A Simple Weight Decay Can Improve Generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.
- [53] M. G. Larson and F. Bengzon. *The finite element method: theory, implementation, and applications*, volume 10 of *Texts in Computational Science and Engineering*. Springer, 2013.
- [54] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
- [55] D. T. Lee and B. J. Schachter. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer and Information Sciences*, 9(3):219–242, 1980.
- [56] J. Li and Y.-T. Chen. *Computational partial differential equations using MATLAB®*. Textbooks in Mathematics. CRC Press, Boca Raton, FL, [2020] ©2020. Second edition of [2460670].

-
- [57] G.-R. Liu and S. S. Quek. *The Finite Element Method: a Practical Course*. Butterworth-Heinemann, 2013.
- [58] D. Lu and Q. Weng. A survey of image classification methods and techniques for improving classification performance. *International journal of Remote sensing*, 28(5):823–870, 2007.
- [59] I. Luz, M. Galun, H. Maron, R. Basri, and I. Yavneh. Learning Algebraic Multigrid Using Graph Neural Networks. In *International Conference on Machine Learning*, pages 6489–6499. PMLR, 2020.
- [60] T. M. Mitchell et al. *Machine Learning*. 1997.
- [61] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-Based Generalization: A Unifying View. *Machine learning*, 1(1):47–80, 1986.
- [62] M. A. Nielsen. *Neural Networks and Deep Learning*, volume 25. Determination press San Francisco, CA, 2015.
- [63] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation Functions: Comparison of Trends in Practice and Research for Deep Learning. *arXiv preprint arXiv:1811.03378*, 2018.
- [64] L. N. Olson and J. B. Schroder. PyAMG: Algebraic multigrid solvers in Python v4.0, 2018. Release 4.0.
- [65] K. O’Shea and R. Nash. An Introduction to Convolutional Neural Networks. *arXiv preprint arXiv:1511.08458*, 2015.
- [66] D. W. Pepper and J. C. Heinrich. *The Finite Element Method: Basic Concepts and Applications with MATLAB, MAPLE, and COMSOL*. CRC press, 2017.
- [67] L. Prechelt. Early Stopping - but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.
- [68] J. Qiu, Q. Wu, G. Ding, Y. Xu, and S. Feng. A survey of machine learning for big data processing. *EURASIP Journal on Advances in Signal Processing*, 2016(1):1–16, 2016.
- [69] A. Quarteroni. *Numerical models for differential problems*, volume 2 of *MS&A. Modeling, Simulation and Applications*. Springer-Verlag Italia, Milan, 2009.

-
- [70] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for Activation Functions. *arXiv preprint arXiv:1710.05941*, 2017.
- [71] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [72] J. W. Ruge and K. Stüben. Algebraic multigrid. In *Multigrid methods*, volume 3 of *Frontiers Appl. Math.*, pages 73–130. SIAM, Philadelphia, PA, 1987.
- [73] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.
- [74] S. Salsa. *Partial differential equations in action: From modelling to theory*, volume 99 of *Unitext*. Springer, third edition, 2016.
- [75] T. Sauer. *Numerical Analysis*. Pearson, second edition, 2012.
- [76] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.
- [77] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.
- [78] V. V. Shaǐdurov. *Multigrid methods for finite elements*, volume 318 of *Mathematics and its Applications*. Kluwer Academic Publishers Group, 1995.
- [79] A. Singh, N. Thakur, and A. Sharma. A review of supervised machine learning algorithms. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1310–1315. IEEE, 2016.
- [80] N. Srivastava. Improving Neural Networks with Dropout. *University of Toronto*, 182(566):7, 2013.
- [81] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 15:1929–1958, 2014.
- [82] K. Stüben. Algebraic multigrid (AMG): experiences and comparisons. *Applied Mathematics and Computation*, 13(3-4):419–451, 1983.

-
- [83] K. Stüben. A review of algebraic multigrid. volume 128, pages 281–309. 2001. Numerical analysis 2000, Vol. VII, Partial differential equations.
- [84] K. Stüben and U. Trottenberg. Multigrid methods: fundamental algorithms, model problem analysis and applications. In *Multigrid methods (Cologne, 1981)*, volume 960 of *Lecture Notes in Math.*, pages 1–176. Springer, 1982.
- [85] R. A. Thompson. Galerkin projections between finite element spaces, 2015.
- [86] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, Inc., San Diego, CA, 2001.
- [87] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. volume 56, pages 179–196. 1996.
- [88] P. Virtanen, R. Gommers, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.
- [89] W. Waegeman, K. Dembczyński, and E. Hüllermeier. Multi-target prediction: a unifying view on problems and methods. *Data Mining and Knowledge Discovery*, 33(2):293–324, 2019.
- [90] P. Wesseling. *An introduction to multigrid methods*. Pure and Applied Mathematics (New York). John Wiley & Sons, Ltd., Chichester, 1992.
- [91] B. Xu, N. Wang, T. Chen, and M. Li. Empirical Evaluation of Rectified Activations in Convolutional Network. *arXiv preprint arXiv:1505.00853*, 2015.
- [92] J. Xu. Iterative methods by space decomposition and subspace correction. *SIAM Review. A Publication of the Society for Industrial and Applied Mathematics*, 34(4):581–613, 1992.
- [93] H. Yserentant. Old and new convergence proofs for multigrid methods. In *Acta numerica, 1993*, Acta Numer., pages 285–326. Cambridge Univ. Press, Cambridge, 1993.
- [94] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
- [95] Z. Zhang. Improved Adam Optimizer for Deep Neural Networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–2. IEEE, 2018.

- [96] G. Zhong, L.-N. Wang, X. Ling, and J. Dong. An overview on data representation learning: From traditional feature learning to recent deep learning. *The Journal of Finance and Data Science*, 2(4):265–278, 2016.
- [97] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The finite element method: its basis and fundamentals*. Elsevier/Butterworth Heinemann, seventh edition, 2013.