Deep Graph Neural Networks for Numerical Simulation of PDEs

Wenzhuo Liu

université PARIS-SACLAY

Deep Graph Neural Networks for Numerical Simulation of PDEs

Réseaux de neurones sur graphes pour la simulation numérique des EDPs

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique
Référent : Faculté des sciences d'Orsay

Thèse préparée dans les unités de recherche LISN (Université Paris-Saclay, CNRS) et Inria Saclay-Ile-de-France (Université Paris-Saclay, Inria),
sous la direction de Marc SCHOENAUER, directeur de recherche INRIA,
le co-encadrement de Mouadh YAGOUBI, responsable de projet R&D à IRT-SystemX, entreprise

Thèse soutenue à Paris-Saclay, le 19/04/2023, par

Wenzhuo LIU

Composition du Jury

Membres du jury avec voix délibérative

Alexandre ALLAUZEN Professeur, ESPCI	Président
François JOUVE Professeur, Université Paris-Diderot	Rapporteur & Examinateur
Patrick GALLINARI Professeur, Sorbonne Université	Rapporteur & Examinateur
Anne SERGENT Maître de conférences, LISN	Examinatrice

THESE DE DOCTORAT
NNT : 2023UPASG032

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Réseaux de neurones sur graphes pour la simulation numérique des EDPs
Mots clés: Apprentissage profond, graph neural networks, EDPs

Résumé: Les équations aux dérivées partielles (EDP) sont un outil essentiel de la simulation numérique pour modéliser des systèmes complexes. Cependant, la résolution de ces équations avec une grande précision nécessite généralement un coût de calcul élevé. Ces dernières années, les algorithmes d'apprentissage profond ont reçu un intérêt croissant pour l'apprentissage à partir d'exemples, et pourraient être utilisés comme substituts des méthodes d'analyse numérique, en appliquant directement les techniques d'apprentissage supervisé à des bases de données de solutions connues, car une fois le modèle neuronal appris, l'inférence des solutions a un coût marginal. De nombreux problèmes subsistent cependant, que cette thèse de doctorat tente de résoudre. La thèse se concentre en particulier sur trois défis majeurs dans l'application des méthodes d'apprentissage profond aux EDP : la gestion des maillages non structurés, qui peut difficilement se faire en utilisant les techniques de traitement d'images, sources d'immenses succès en apprentissage profond ; les problèmes de généralisation, en particulier pour des données hors-distribution par rapport aux données d'apprentissage ; et les coûts de calcul élevés pour générer ces données d'apprentissage. Nos trois contributions sont fondées sur les Réseaux de Neurones sur Graphes (GNNs) : un modèle hiérarchique inspirées des méthodes multi-grilles de l'analyse numérique ; le méta-apprentissage pour améliorer les performances sur les données hors distribution ; et l'apprentissage par transfert entre des ensembles de données multifidélité pour réduire le temps de génération des données d'apprentissage. Ces approches sont validées expérimentalement sur différents systèmes physiques.

Title: Deep Graph Neural Networks for Numerical Simulation of PDEs
Keywords: Deep learning, Graph Neural Networks, PDEs

Abstract: Partial differential equations (PDEs) are an essential modeling tool for the numerical simulation of complex systems. However, their accurate numerical resolution usually requires a high computational cost. In recent years, deep Learning algorithms have demonstrated impressive successes in learning from examples, and their direct application to databases of existing solutions of a PDE could be a way to tackle the excessive computational cost of classical numerical approaches: Once a neural model has been learned, the computational cost of inference of the solution on new example is very low. However, many issues remain that this Ph.D. thesis investigates, focusing on three major hurdles: handling unstructured meshes, which can hardly be done accurately by simply porting the neural successes on image processing tasks; generalization issues, in particular for Out-of-Distribution examples; and the too high computational costs for generating the training data. We propose three contributions, based on Graph Neural Networks, to tackle these problems: A hierarchical model inspired by the multi-grid techniques of Numerical Analysis; The use of Meta-Learning to improve the performance of Out-of-Distribution data; and Transfer Learning between multi-fidelity datasets to reduce the computational cost of data generation. The proposed approaches are experimentally validated on different physical systems.

Maison du doctorat de Université Paris-Saclay
2e étage, aile ouest, École normale supérieure Paris-Saclay
4 avenue des Sciencs
91190 Gif-sur-Yvette, France

Synthèse

Les équations aux dérivées partielles (EDPs) constituent un outil essentiel dans le domaine de la simulation numérique, permettant de modéliser des systèmes complexes de manière efficace et précise. Ces équations sont utilisées dans une multitude de domaines, allant de la physique fondamentale à l'ingénierie et à l'économie, permettant de décrire des phénomènes naturels et complexes. Cependant, obtenir des solutions précises à ces équations peut souvent entraîner des coûts de calcul prohibitifs. En effet, la résolution d'EDP à haute résolution nécessite une puissance de calcul considérable.

Dans ce contexte, les techniques d'apprentissage profond ont attiré une attention considérable au cours des dernières années. Ces algorithmes, capables d'apprendre de manière autonome à partir d'exemples, ont le potentiel de servir de substituts aux méthodes d'analyse numérique traditionnelles. Par exemple, il est possible d'appliquer directement des techniques d'apprentissage supervisé à des bases de données de solutions connues à des EDPs. Une fois le modèle neuronal entraîné, l'inférence des solutions peut être effectuée à un coût marginal. Cela signifie que les techniques d'apprentissage profond peuvent offrir une alternative attrayante aux méthodes conventionnelles de résolution d'EDPs, en particulier lorsque les ressources de calcul sont limitées.

La thèse se concentre en particulier sur trois défis majeurs dans l'application des méthodes d'apprentissage profond aux EDP :

1. La plupart des méthodes d'apprentissage profond sont conçues pour fonctionner avec des données structurées, comme des images, qui peuvent être facilement manipulées avec des techniques standard de traitement d'images. Cependant, les maillages non structurés, qui sont couramment utilisés dans les simulations numériques, présentent des défis uniques qui rendent difficile leur traitement avec des méthodes d'apprentissage profond traditionnelles.

2. La capacité de généralisation des réseaux neuronnes est limitée, en particulier lorsqu'il s'agit de données hors-distribution par rapport aux données d'apprentissage. Les méthodes d'apprentissage profond peuvent avoir du mal à fournir des résultats précis lorsque les données de test sont significativement différentes de celles sur lesquelles le modèle a été formé.
3. Les coûts de calcul élevés pour générer ces données d'apprentissage. Bien que l'inférence à partir de modèles neuronaux formés puisse être relativement peu coûteuse, l'entraînement initial des modèles peut être coûteux en termes de temps et de ressources de calcul.

Dans cette thèse, nos contributions est de proposer différentes approches visant à résoudre les trois problèmes mensionnés.
1. Nous introduisons un modèle hiérarchique inspiré des méthodes multigrilles de l'analyse numérique, basé sur les Réseaux de Neurones sur Graphes (GNNs). D'une part, les GNNs peuvent être appliqués directement aux maillages non structurés, et d'autre part, le modèle hiérarchique peut aider à extraire les features utiles.
2. Nous introduisons une perspective de méta-apprentissage des tâches de problèmes physiques et appliquons une approche de méta-apprentissage basée sur l'optimisation pour améliorer la performance des modèles appris sur les points de données hors distribution.
3. Nous proposons une approche basée sur l'apprentissage par transfert pour transférer les connaissances préalables des données de maillage grossier aux données de maillage à haute résolution. L'apprentissage par transfert contribue à réduire la taille du jeu de données nécessaire sur les maillages à haute résolution, conduisant à une réduction globale du temps de calcul sur la génération de données.

Nous validons ces trois approches expérimentalement sur divers problèmes physiques tels que la dynamique des fluides, l'élasticité et les équations de Poisson. Cependant, les approches proposées sont très générales et peuvent être appliquées à une variété de problèmes au-delà de ceux testés.

Acknowledgements

I would like to express my sincere gratitude to all those who have supported me throughout my Ph.D. thesis.

First of all, I want to thank my supervisor and advisor, Marc Schoenauer and Mouadh Yagoubi, whose guidance and mentorship were invaluable. Their expert insights and feedback played an essential role in my research.

I am also grateful to my family and friends for their unwavering support, especially during difficult times. Their belief in me and their encouragement have given me the strength to persevere and succeed.

I would like to acknowledge the colleagues and peers who have collaborated with me, Ahmadali Tahmasebimoradi, Balthazar Donon, Chetra Mang, David Danan, Emmanuel Menier, Jean-Patrick Brunet, Milad Leyli-abadi, Romain Barbedienne, and Sara Yasmine Ouerk, sharing their knowledge, expertise, and resources. Working together has been a joy and has greatly enriched my work.

I would also like to express my appreciation to the administrative staff and support services at the institution, whose hard work and dedication have been integral to the success of my studies.

Finally, I would like to thank the reviewers and examinators François JOUVE, Patrick GALLINARI, Alexandre Allauzen, and Anne Sergent, whose constructive feedback helped me to improve the quality of my work. Your insights and suggestions have been invaluable.

Once again, I am deeply grateful to everyone who has contributed to my journey. Your support and encouragement have made a tremendous impact, and I will always remember your kindness and generosity.

Contents

List of Figures

List of Tables

1 Introduction

Nowadays, numerical simulation represents an essential tool in designing and managing real-world systems, thanks to its lower cost compared to direct experimental testing on the system to be designed. (see Figure 1.1) Many industrial applications have benefited from the contributions of numerical simulation to improve the performance of systems. The mathematical modeling of the numerical simulation use often partial differential equations (PDEs) to model continuous phenomena and describe the behavior of real-world complex systems, be they physical(e.g., mechanics, biology,.etc) or artificial (e.g, finance, networks,.etc).

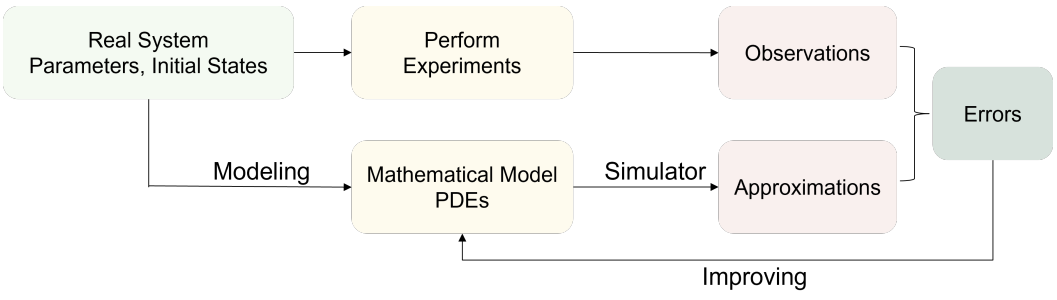


Figure 1.1: Numerical Simulation for real systems

However, it is usually impossible to derive a PDE's solution in some analytic form. The numerical analysis encompasses different numerical simulation

methods that are widely used to approximate the solutions of PDEs. One of the most famous methods used to deal with this limitation is the Finite Element Method (FEM): This approach discretizes the domain into *meshes* and computes approximated values of the quantities of interest on each node or cell of the mesh. These approaches can predict the behavior of the systems, generally with known error bounds. However, in order to be accurate enough for operational use, these simulations come at a high computational cost for complex systems.

In recent years, and in particular, since the rise of Deep Learning approaches to solve computer vision or Natural Language Processing problems in the 2010s, there has been a rapid growth in the use of machine learning algorithms to solve problems from different domains where the numerical analysis approaches are hard to design, or too expensive to compute accurately. Deep neural networks have become the most popular approach due to their ability to solve complex tasks, outperforming existing numerical analysis algorithms when large-scale data are available.

In such context, the overall goal of the present thesis is to study the ability to use Deep Learning algorithms to reduce the computational cost of FEM-based resolution methods.

Partial Differential Equations

Studies on PDEs date back to the 18th century, when scientists such as Euler, Lagrange, d'Alembert, and Laplace [1–3] describe physical laws in mechanics as PDEs. Since the 19th century, some methods have been proposed to calculate analytical solutions for certain particular PDEs with appropriate boundary conditions. Separation of variables, method of characteristics and integral transform, etc. [4–6] are essential methods to find the exact solutions of PDEs. In the meantime, scientists like Poincaré, show interest in studying general properties of PDEs, existence, uniqueness, and regularity [7–9] for example.

In the early 20th century, there has been a rise in the development of the numerical analysis of PDEs. The finite difference methods (FDM) [10], the finite element methods (FEM) [11], and the finite volume (FVM) methods [12] are three common numerical techniques. They involve discretizing the domain and the PDEs, and solving the resulting discrete problem to obtain approximate numerical solutions. On a practical side, in engineering, almost all PDEs are

solved today by numerical analysis approaches. These approaches can predict the physical behavior of the systems accurately, however, at a high computational cost for complex systems.

Among the techniques proposed to overcome this drawback, multi-grid [13] and model order reduction [14] stand out. Multi-grid methods use a hierarchy of discretizations to accelerate the convergence of basic iterative methods. Model order reduction aims to simplify the mathematical model itself, such as the state space dimension or degrees of freedom, to reduce the time complexity. Both Multi-grid and Reduced Order Modeling(ROM) have been demonstrated to be effective in reducing computational time. However, it is important to note that these techniques may not be sufficient to fully address the computational challenges associated with large-scale PDEs.

Machine Learning for PDEs

Back in the 50s, some simple algorithms have been proposed, pioneering machine learning research [15]. Machine learning algorithms automatically detect patterns from experience by the use of large amounts of data and later use the uncovered patterns to predict outcomes. There are various types of models in the field of machine learning. Common algorithms include linear regression, decision trees, support vector machines, random forest, naive Bayes, etc. [16–20]. Later on, the deep learning algorithms come to the stage. The term multi-layer neural networks were introduced in the 60s [21]. Later on, the backpropagation algorithm and its general use in neural networks were proposed [22] to train deep learning models. In the 2010s, boosted by the rapid development of specialized hardware like GPUs, training a deep learning model became feasible. Since then, deep neural networks have become the most popular approach due to their ability to solve complex tasks.

Using machine learning algorithms to solve PDEs has received particular attention since mid-2010. Regarding the learning algorithms applied, we can categorize these researches into two main categories, data-driven approaches to accelerate computation time and unsupervised methods to solve high dimensional PDEs.

Some scientists have proposed data-driven methods (see Figure 1.2), making full use of labeled data accumulated by numerical solvers, known as the *training set*. Some works [23, 24] are based on ROM model, using deep learning methods

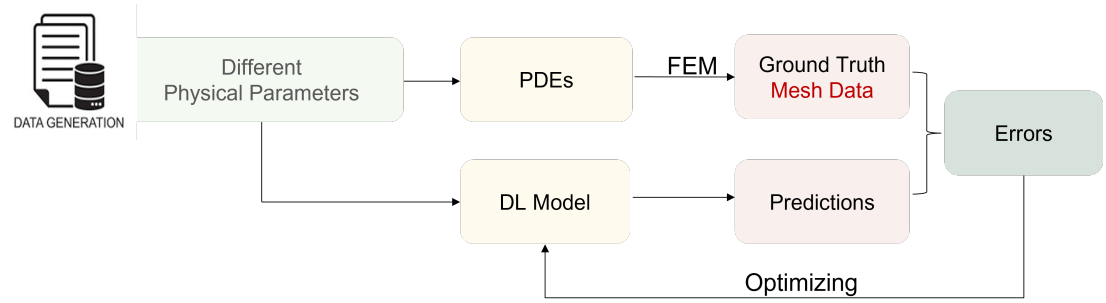


Figure 1.2: Data-driven methods to solve PDEs

to correct the result from ROM. Others aim to construct a model to solve a set of PDEs of the same type. The inputs of the model are physical quantities that define a specific instance of a given PDE. By analyzing labeled data from the training set, these methods learn a function that maps the input to learned features to the desired solution. Thanks to their capacity to capture spatial features, Convolutional Neural networks(CNN) are the most widely used model structure due to their tremendous successes in image analysis. Many works [25–28] applied CNNs to solve PDEs. For instance, [25] constructs a convolutional model to approximate electromagnetic problems by solving Poisson's equation on a square domain. Training data is generated by a FEM solver using a regular mesh, resulting in data living in a Euclidean space, which ensures the applicability of CNNs.

Spatial phenomena at different scales can be captured more efficiently by using specific network architectures, such as the U-Net architecture [29]. For instance, [26] utilizes the U-Net model to solve Reynolds Averaged Navier-Stokes (RANS) flow problems on airfoil shapes. The generated data on the unstructured mesh is first projected on structured grids as images before training. Compared with traditional methods, the CNN models can indeed reduce the overall computational time to compute the solution of new instances. They either use a structured mesh to discretize the physical domain such that image-like data is generated directly or apply interpolation to convert mesh data into structured grids.

In recent years, many attempts have been made to construct a deep learning model based on *graph neural networks* (GNNs) that can be applied directly to mesh data instead of projection into structured grids. [30] discussed fluid

flow field problems on different irregular geometries. It considers CFD data as a set of points (called point clouds) and applies the PointNet[31] architecture specially designed for such a data type. [32] combines graph neural networks with a traditional CFD solver (run on a coarse mesh) to accelerate fluid flow prediction on a much finer mesh.

On the other hand, unsupervised learning methods are designed to solve the cases where current numerical solutions on PDEs are inefficient for problems with high dimensions or complex geometry. A significant difficulty for such problems is meshing. On the one hand, forming a mesh is costly for complex geometric issues. On the other hand, it becomes infeasible in high-dimensional space. In such cases, unsupervised learning algorithms are proposed to avoid mesh construction. The basic idea of such methods is to train a model $f_\theta(x)$ to simulate the solution $u(x)$ of a specific PDE. Based on universal approximation theorems [33], neural networks have strong expressive power. Any continuous functions can be approximated by neural networks with only one hidden layer, provided it is large enough.

The loss function is directly defined by physical quantities without using any training set. The solution $u(x)$ for a specific PDE is approximated by a deep neural network that is directly trying to satisfy the equation. After learning, by entering a variable x_0, the network will predict the value $u(x_0)$. [34] used fully connected layers to approximate the solution on complex geometry. [35] discussed the possibility of solving high-dimensional problems within 200 dimensions by neural networks. The most famous work in such domain is the Physics-Informed Neural Network (PINN) [36]. The authors apply them to more challenging dynamic problems described by time-dependent nonlinear partial differential equations.

To date, solving PDEs with machine learning algorithms is still at an early stage. For data-driven methods, there are many issues yet to be resolved. We list three common problems encountered when adapting machine learning to PDEs in most cases:

- When dealing with problems defined on a geometrically complex domain, where the data does not live in Euclidian space, CNN approaches can lead to a significant interpolation error, in particular on the boundary of the actual domain. As an alternative, GNN-based models can handle mesh data directly, avoiding the need for structured grid projections. However,

the irregular nature of mesh data presents challenges in defining pooling operators, and to date, no multi-resolution GNNs like the U-Net architecture have been proposed yet.

- As the deep learning methods learn patterns directly from the data instead of studying problem-related constraints beyond the PDEs, they cannot really grasp the physics of the problem at hand. The lack of underlying physical laws leads to generalization issues: the predicted results can significantly lack any physical significance. Learned models often underperform on out-of-distribution samples. Generally speaking, the deep learning model can predict unseen examples that are close to the training set but perform poorly at solving new problems that significantly differ from the training examples.

- A complex neural network requires a large amount of data to learn latent patterns from PDEs. The most feasible method to collect datasets is to apply traditional solvers to numerical analysis. The accumulation of data on our physical problems becomes expensive and time-consuming with the use of numerical solvers.

Main Contributions

In the present thesis, we propose different approaches aiming to tackle the three issues above. The proposed approaches help to better adapt deep learning methods to PDEs. Our main contributions are threefold:

- We propose the hierarchical Graph models for the numerical simulation of PDEs and experimentally demonstrate the power of Graph models to handle PDEs in various physical domains [37].

- We introduce a meta-learning perspective of physical problem tasks and apply an optimization-based meta-learning approach to enhance the performance of the learned models on out-of-distribution data points [38].

- We propose a transfer learning based approach to transfer prior knowledge from coarse mesh data to high-resolution mesh data. Transfer learning helps to reduce the size of the dataset needed on high-resolution meshes, leading to an overall reduction of computation time on data generation [39].

We validate these three approaches experimentally on various physical problems such as fluid dynamics, elasticity, and Poisson's equations. However, the proposed approaches are very general and can be applied to different domains. This thesis consists of 7 chapters with two main parts:

- In PART I (Chap. 2 and 3), a comprehensive survey of the related existing studies and techniques in the literature is presented. Classical FEM methods are introduced first to provide a deeper understanding of the nature of approximately solving partial differential equations. Then, the fundamental concepts of Deep Learning are briefly presented in Chap. 3, sections 1 to 3. The following sections, 3 to 5, delve into the various connected machine learning techniques, such as graph neural networks, transfer learning, and meta-learning, that are used to solve PDEs.

- In PART II (Chap. 4 to 7), the methods we propose and the associated experimental results are discussed in detail, including a thorough analysis of the hierarchical Graph models and the meta-learning and transfer learning applied in PDEs. The experimental setup, data generation, and evaluation metrics used are also presented. The experimental results are discussed and compared with those of baseline models. Finally, the conclusion of the work is presented, including a summary of the main findings and contributions of the work and a discussion about the evaluation of the proposed methods, the limitations of this work, and potential directions for future research.

Part I

Background

2
Numerical Analysis

Contents

Partial Differential Equations (PDEs) are widely used to model physical systems. For example, the heat equation describes the diffusion of heat in a material and can be applied in the design of thermal management systems, such as heat exchangers for power plants or heat sinks for electronic devices. And the Navier-Stokes equations describing the motion of the fluid are commonly used in the analysis and design of fluid flow systems, such as the design of airfoils in aerodynamics. Numerical Analysis (Figure 2.1) is the branch of Mathematics that studies numerical simulations, deriving, theoretically analyzing, and proposing practical implementations of numerical methods. Common numerical analysis methods on PDEs involve discretizing the continuous system into a discrete problem, resulting in a set of algebraic equations, theoretically bounding the error

(difference between the solutions of these equations and the exact continuous solution of the PDE at hand), and practically solving these equations at the smallest possible computational cost. Several approaches have been proposed to reduce this computational cost, and the work presented in this dissertation will get inspiration from the so-called multi-grid methods that are based on the use of different granularity of discretization.

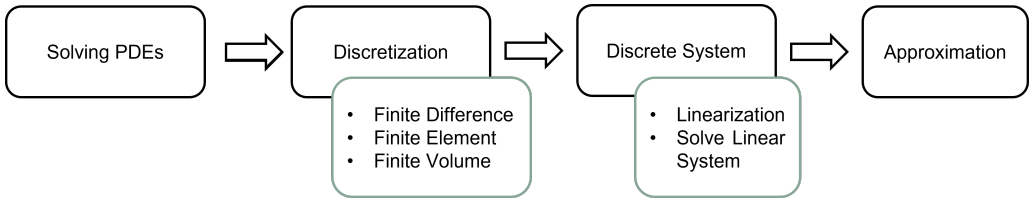


Figure 2.1: Numerical Analysis for PDEs

In this Chapter, we will rapidly survey some well-known discretization methods in PDEs, as well as some iterative methods used to solve the resulting discrete models. Finally, we will discuss some popular multi-grid methods that will be used later in this work. Interested readers are referred to the books [13, 40, 41] for more details on numerical analysis and multi-grid methods.

2.1 Discretization Methods

Systems in the physical world are discretized by subdividing the continuous physical domain into small segments. Common discretization schemes include the finite difference method (FDM), finite element method (FEM), and finite volume method (FVM).

Finite Difference Method is the simplest discretization method. The basic idea is to replace the derivatives in PDEs with the difference quotients that define the derivatives: because these quotients converge to the actual derivatives when the discretization step goes to zero, the solutions of the discretized equations will converge, under appropriate hypotheses, to those of the continuous equations.
Let us consider the one-dimensional case (in Figure 2.2) for the sake of simplicity. The basic finite difference method (FDM) divides the interval of [a,b] into n equal sub-intervals of length h=(b-a)/n. By Taylor's theorem, any C^2 continuous

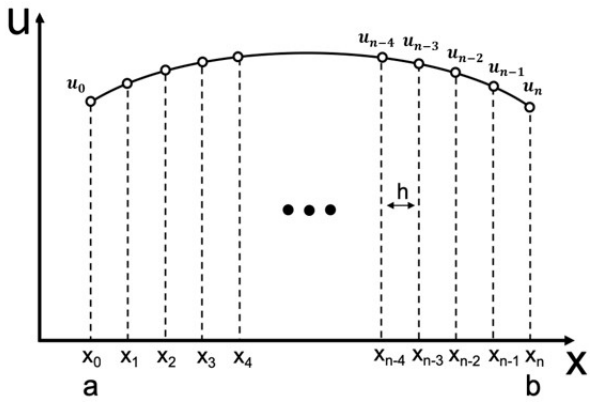


Figure 2.2: Finite difference Method

function u at the point x_i can be expanded as:

$$u(x_i + h) = u(x_i) + hu'(x_i) + O(h^2) \tag{2.1}$$

Denoting u_i the value $u(x_i)$, the derivative and the second derivative at x_i can be approximated by a second-order accurate scheme:

$$u'(x) \approx \frac{u_{i+1} - u_{i-1}}{2h}, \qquad u''(x) \approx \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} \tag{2.2}$$

We use the finite difference expressions above to replace the derivatives of $u(x)$ in PDEs to obtain the approximate discrete system consisting of algebraic equations.

The fact that FDM uses equal segments for discretization is a bottleneck when solving PDEs with complex geometry in multiple dimensions, as complex geometry usually requires higher resolution leading to a higher computational cost. Additionally, in certain physical problems, certain sub-domains may require greater attention, but with FDM, every grid is considered of equal importance. This lack of flexibility has motivated the development of FEM and FVM, which avoid regular grid subdivisions by using the PDEs' integral form.

Finite Element Method discretizes the PDEs by dividing the problem domain into small *elements*, i.e., a partition of the domain into polyhedrons. Figure 2.4 shows a 2D domain discretized in a triangular *mesh*. Meshes in 2D can also be made of quadrangles, while 3D meshes are generally built from tetrahedrons, pentahedrons, or hexahedrons elements.

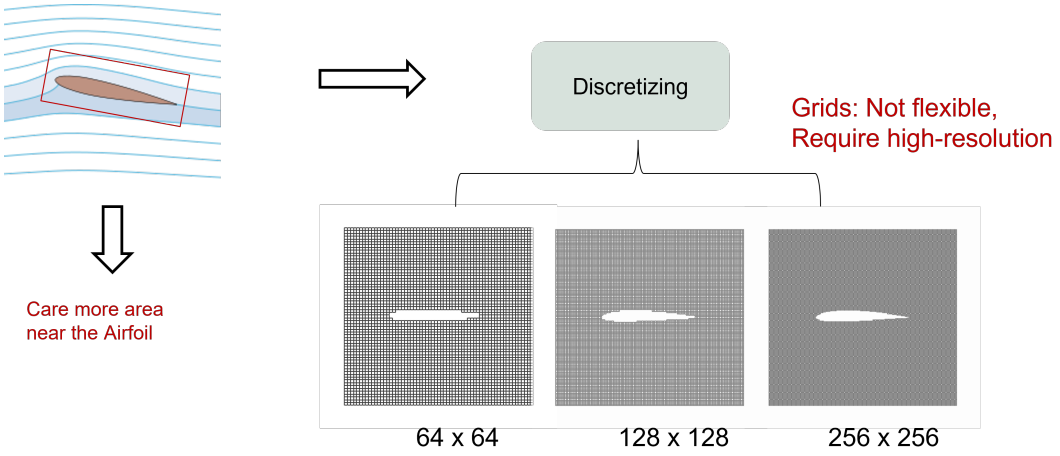


Figure 2.3: Problems existing in FDM

On each element, the values of the solution at given points of the elements, the "degrees of freedom" (dof) of the elements, are chosen as unknown, and the PDE, written in variational form, is applied using a specific basis of the approximation space, with small supports. The resulting set of discrete equations is the algebraic system whose solution is the sought approximation of the PDE at hand.

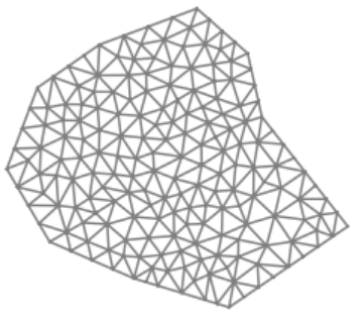


Figure 2.4: A Mesh

According to [40], the FEM approach is made of the following 5 steps:

1. Preprocessing/meshing: subdividing the problem domain into finite elements by constructing a mesh.

2. Element formulation: development of sub-equations for each element by using the weak form or variational equations form of the PDEs.

3. Assembly: obtaining the equations of the entire system from the equations of individual elements.

4. Solving the entire system.

5. Postprocessing: determining quantities of interest from the unknown of the discrete system (e.g., stresses and strains in solid mechanics) and visualizing the approximate solution.

The FEM computes an approximation u_i of the quantity of interest $u(x)$ on each mesh degree of freedom i. The final approximate solution u can be expressed as

$$u(x) = \sum_{i}^{N} u_i \Phi_i(x),$$

where $\Phi_i(x)$ is the basis function corresponding to dof i.

In the FEM, the accuracy of the solution directly depends on the discretization finesse (maximal size of all elements, directly linked to the number of elements when using modern mesh generators), and meshes with thousands to millions of nodes are routinely used in practice to obtain a reasonably accurate solution. Meanwhile, the computation cost increases with the number of elements as the number of equations from the assembled system increases. There hence exists a trade-off between accuracy and computational cost. Choosing a proper mesh is one of the most critical components of FEM so that an optimal balance can be achieved between accuracy and computational time.

Finite Volume Method is mainly used in the field of fluid dynamics. Similar to FEM, the finite volume method(FVM) uses a mesh to subdivide continuous domains into smaller subdomains. The FVM applies the divergence theorem on each element (cell) of the mesh, which converts volume integrals containing divergence terms into surface integrals. Contrary to FDM or FEM, which approximate the solution using nodal values, it constructs approximations of the solution within the cells of the cells.

All three methods end up solving one (or several) system(s) of linear equations to compute an approximate numerical solution of the PDE at hand. And for all three methods, these linear systems are sparse, and the equation for an unknown u_i involves only a few neighbors of point i. Overall, FDM is mostly used for geometries that can be discretized by structured grids (e.g., rectangles), while

FEM and FVM are more suitable for complex domains. As FVM is based on the integral formulation of a conservation law, it is mainly used to solve PDEs in fluid dynamics, which involves fluxes of the conserved variable. In this dissertation, we are only interested in general PDEs defined on a complex geometry where FDM is not applicable.

2.2 Solving Discrete Linear Systems

The discretization methods convert the continuous PDEs into a set of algebraic equations. The discrete equations of a linear PDE are also linear. These linear systems can be efficiently solved by iterative techniques. In the case of non-linear PDEs, linearization schemes are required to convert the non-linearity into a sequence of linear systems: a sequence of linearized equations is solved iteratively, converging to the solution of the non-linear system. This section reviews some iterative methods for solving linear systems and commonly used linearization schemes for handling non-linear systems.

2.2.1 Iterative Methods

Iterative methods are used to solve large and sparse linear systems. They generate a sequence of approximations $x^{(k)}$ such that $x^{(k)} \to x$ when $k \to \infty$ if x is the solution of the linear system at hand.
Let $Ax = b$ denote a system of linear equations, where:

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}, \qquad \mathrm{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathrm{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

The basic iterative methods split the matrix A into two parts $A = M - K$, where M is any non-singular matrix, "easy" to invert. The iterative methods is then defined by:

$$Mx^{(k+1)} = Kx^{(k)} + b, \tag{2.3}$$

that is, for each iteration, the new approximation $x^{(k+1)}$ is calculated by:

$$x^{(k+1)} = M^{-1}Kx^{(k)} + M^{-1}b \tag{2.4}$$

Thus, the algorithm [41] is given in Algorithm 1. The key to the success of iterative methods is the choice of split matrices M and K. $M^{-1}K$ and $M^{-1}b$ should be easily calculated. In the following, we outline two common iterative

Algorithm 1 Basic iterative method for solving a linear system

1: Choose a starter guess $x^{(0)}$
2: **for** $k = 0, 1, 2, ...$ until the convergence criterion is satisfied **do**

$$x^{(k+1)} = M^{-1}Kx^{(k)} + M^{-1}b$$

3: **end for**

methods, the Jacobi [42] and Gauss–Seidel [43] algorithms.

Jacobi Method decomposes the matrix A into a diagonal component $M = D$ and the rest part $K = -L - U$, where L is the lower triangular part and U is the upper triangular part of A.

$$\mathrm{D} = \begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 \\ 0 & a_{22} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{bmatrix}, \qquad \text{and L+U} = \begin{bmatrix} 0 & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & 0 & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & 0 \end{bmatrix}$$

Since D is a diagonal matrix, its inverse D^{-1} can be easily computed. The solution is then computed iteratively via: $x^{(k+1)} = -D^{-1}(L + U)x^{(k)} + D^{-1}b$. The element-based formula is thus:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij}x_j^k) \tag{2.5}$$

The algorithm of the Jacobi method is given in Algorithm 2.

Gauss–Seidel Method lets $M = D + L$ and $K = -U$, so that the iterative scheme can be expressed as:

$$x^{(k+1)} = -(D + L)^{-1}Ux^{(k)} + (D + L)^{-1}b$$

We compute the elements of $x^{(k+1)}$ sequentially using forward substitution since $D + L$ has a lower triangular structure. The computation of $x^{(k+1)}$ uses the elements of $x^{(k+1)}$ that have already been computed. The algorithm of the Gauss-Seidel method is given in Algorithm 3.

Algorithm 2 The Jacobi Method

Choose a starter guess $x^{(0)}$
for $k = 0, 1, 2, ...$ until convergence **do**
    for $i = 1, 2, ..., n$ **do**
        s = 0
        for $j = 1, 2, ..., n$ **do**
            if $j \neq i$ **then**
                $s = s + a_{ij} x_j^{(k)}$
            end if
        end for
        $x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - s)$
    end for
end for

Algorithm 3 The Gauss-Seidel Method

Choose a starter guess $x^{(0)}$
for $k = 0, 1, 2, ...$ until convergence **do**
    for $i = 1, 2, ..., n$ **do**
        s = 0
        for $j = 1, 2, ..., n$ **do**
            if $j < i$ **then**
                $s = s + a_{ij} x_j^{(k+1)}$
            else if $j > i$ **then**
                $s = s + a_{ij} x_j^{(k)}$
            end if
        end for
        $x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - s)$
    end for
end for

2.2.2 Linearization Schemes

The linearization schemes turn non-linear systems obtained from nonlinear PDEs into a sequence of linear systems that can be solved using any of the iterative methods mentioned above. We discuss here two schemes, Picard Iteration [44], and Newton–Raphson method [45].

Picard Iteration is an easy linearization scheme of handling the non-linearity given a system of non-linear equations. It can be applied when the non-linear

system is of the form $A(x)x = b(x)$. The idea, at iteration k, is to simply use the previous solution $x^{(k)}$ to replace x in the nonlinear terms and to solve the resulting linear equation for $x^{(k+1)}$:

$$A(x^{(k)})x^{(k+1)} = b(x^{(k)}) \tag{2.6}$$

Newton–Raphson method is designed to find the minimum of a function $f(x)$. At each iteration k, x_k is updated as:

$$\nabla f(x)(x_{k+1} - x_k) = -f(x) \tag{2.7}$$

At the algebraic Level, for example, given a non-linear PDE: $-\nabla(q(x)\nabla x) = f$,

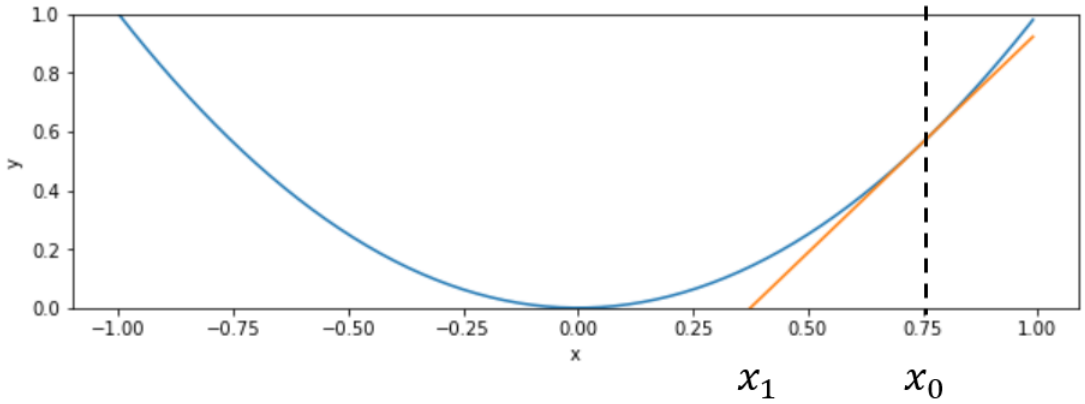


Figure 2.5: Newton-Raphson method on a polynomial function

we express the discrete variational problem as:

$$F_i(x_1, \ldots, x_N) \equiv \sum_{j=1}^{N} \int_\Omega \left(q \left(\sum_{\ell=1}^{N} x_\ell \phi_\ell \right) \nabla \phi_j x_j \right) \cdot \nabla \hat{\phi}_i \, \mathrm{d}x = 0, \quad i = 1, \ldots, N \tag{2.8}$$

The Newton–Raphson method to update x is:

$$\sum_{j=1}^{N} \frac{\partial}{\partial x_j} F_i(x_1^k, \ldots, x_N^k)\delta x_j = -F_i(x_1^k, \ldots, x_N^k), \quad i = 1, \ldots, N \tag{2.9}$$

$$x_j^{k+1} = x_j^k + \omega \delta x_j, \quad j = 1, \ldots, N \tag{2.10}$$

We solve the linear equation to get δx_j for further update.

As can be seen in both algorithms above, each iteration requires solving a linear system, and any iterative methods presented in Section 2.2.1 can be used.

2.3 Multi-grid Methods

Even though the basic iterative methods, such as Jacobi and Gauss-Seidel presented above, are the most efficient methods to solve sparse linear systems, when handling meshes with high resolutions involving a large number of linear equations, solving such a system will be time-consuming. Therefore, multi-grid algorithms have been developed to decrease the overall computational cost of the discretized solution. The main idea is to compute approximate solutions to the problem at hand on meshes of different granularities. The steps on coarse meshes are fast and help to unveil the low-frequency pattern of the solution, while fine meshes refine the solutions, removing unwanted spatial oscillations. There are several common steps for multi-grid algorithms:

1. **Relaxation**: Apply a few iterations of the Gauss–Seidel or Jacobi method to obtain an approximation.

2. **Interpolation**: Multigrid algorithms use solutions on coarse meshes as an initial guess to accelerate the convergence for linear systems on fine meshes. This requires some mechanism for transferring information between resolutions. A common process used in numerical analysis is called interpolation. These interpolation algorithms are applied directly in multigrid.

3. **Correction**: Correct the approximation on the fine mesh based on the error obtained on the coarse mesh.

Consider a fine mesh Ω_{fine} and a coarse mesh Ω_{coarse}, a typical cycle of a multigrid method is the following:

- Apply a few iterations of an iterative method on the linear system $Ax = f$, defined on Ω_{fine}, to obtain an approximation v^{fine} of the solution;

- Compute the residual $r = f - Av^{fine}$;

- Interpolate the residual from Ω_{fine} to Ω_{coarse};

- Solve the residual equation $Ae = r$ on the coarse mesh Ω_{coarse}, to obtain an approximation of the error e^{coarse};

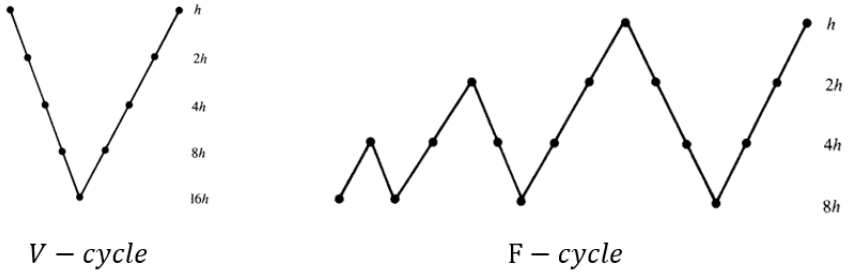


Figure 2.6: V-Cycle and F-cycle schemes

- Interpolate the error e^{coarse} to the fine mesh Ω_{fine};
- Correct the approximation $v^{fine} \leftarrow v^{fine} + e^{fine}$

There are various multigrid schemes that apply different cycles based on the loop above. Two of them have been inspirational in this work, the V-cycle, and the F-cycle. V-cycle algorithm starts from the finest mesh and maps down to

Algorithm 4 V-Cycle Scheme on a uniform grid of spacing h

Require: A set of resolutions of meshes on the same domain Ω, from fine to coarse: Ω_h, Ω_{2h}, Ω_{4h}, Ω_{8h}, ...

1: **for** $\ell = 1, 2, 4, ..., L$ **do** until the coarsest mesh L

2: Relax the linear system with iterative methods:

$$A^{\ell h} u^{\ell h} = f^{\ell h}$$

3: Calculate the residual error $r^{\ell h} = f^{\ell h} - A^{\ell h} u^{\ell h}$

4: Interpolate r on next mesh scale: $f^{2\ell h} = I_{\ell h}^{2\ell h} r^{\ell h}$

5: **end for**

6: **for** $L = ..., 4, 2, 1$ from the coarsest until the finest mesh **do**

7: Correct the approximation on each mesh level:

$$v^{\ell h} = v^{2\ell h} + I_{2\ell h}^{\ell h} v^{\ell h}$$

8: Relax on $A^{\ell h} u^{\ell h} = f^{\ell h}$ with an initial guess $v^{\ell h}$

9: **end for**

the coarsest mesh then works its way back to the finest mesh. Different from V-cycle Scheme, F-cycle algorithm begin with coarest mesh and joins nested iteration with the V-cycle.

Algorithm 5 F-Cycle Scheme on a uniform grid of spacing h

Require: A set of resolutions of meshes on the same domain Ω, from fine to coarse: Ω_h, Ω_{2h}, Ω_{4h}, Ω_{8h}, ...

1: **for** $\ell = 1, 2, 4, ..., L$ until the coarsest mesh L **do**
2: Initialize $f^{2\ell h} = I_{\ell h}^{2\ell h} f^{\ell h}$
3: **end for**
4: Solve the linear system on coarsest grid
5: **for** $L = ..., 4, 2, 1$ **do** from the coarsest mesh to the finest mesh
6: Call V-Cycle Scheme on $\Omega_{2\ell h}$
7: Correct the approximation on finer mesh

$$v^{\ell h} = v^{2\ell h} + I_{2\ell h}^{\ell h} v^{\ell h}$$

8: **end for**

The idea of multi-grid will be further discussed in Section 4 where the multi-grid algorithms will be combined with graph neural networks in order to construct multi-resolution graph models to approximate the solutions of PDEs. In particular, a critical issue will be to design ad hoc interpolation procedures for GNNs (Section 4), as no such generic procedure exists between graphs of different granularities.

2.4 Other methods to reduce simulation time

Besides multi-grid algorithms, other methods have been proposed to reduce the computational cost of numerical solutions of PDEs. This section briefly surveys these methods for the sake of completeness.

Model Reduction Method aims at decreasing the dimensionality of the model without sacrificing its accuracy to reduce the computational complexity of large-scale dynamical systems. The method generates a reduced-order model (ROM) that approximates the original high-fidelity model while capturing the essential dynamics of the original model in the meanwhile. There are different methods to construct the ROMs, such as Proper Orthogonal Decomposition, Balanced Truncation, Reduced Basis Method, and Empirical Interpolation Method. These methods find a low-dimensional subspace that captures the most important patterns of the system or selects the most relevant degrees of freedom. ROM is widely used in various engineering and scientific fields to reduce computational costs and improve the efficiency of the simulation.

Surrogate Modeling mimics the behavior of the complex model while being computationally cheaper to evaluate. This method is commonly used in engineering design when the outcome of interest cannot be easily measured or computed. Same to machine learning tasks, surrogate models use a data-driven approach. The exact inner workings of the system are not necessarily known; only the input-output relationships are important. The model is constructed by analyzing the output of the simulator for a limited number of carefully selected data points. This can be done using traditional machine learning algorithms like Support Vector Machine [18] or the Gaussian Mixture process. In this work, the proposed models can also be considered as a form of surrogate modeling.

The computational cost of numerically solving large-scale PDEs remains a significant hurdle in many fields of science and engineering. Techniques like multi-grid methods and reduced-order modeling are not fully capable of addressing these challenges. The cost of computation remains high. Traditional machine learning approaches through surrogate modeling are also limited in their ability to handle the complexity of physical systems, as they are typically limited to solving simple problems in low-dimensional spaces. Therefore, we propose utilizing deep learning methods to tackle such complex physical systems effectively.

3
Deep Learning

Contents

In this chapter, we briefly introduce some basic knowledge of deep learning. We start from machine learning basics, including fundamental statistical learning theory, and discuss common issues like underfitting and overfitting, hyper-parameters setting, and methods for model evaluation, which are general principles that run through machine learning. Secondly, we discuss the basic block of deep learning, Multi-Layer Perceptron, and review the commonly used structure for treating image data, Convolutional Neural Networks. Lastly, we outline the more advanced techniques that we have used during this thesis, namely graph neural networks (GNNs), meta-learning, and transfer learning.

Readers seeking a wider perspective on machine learning and deep learning are encouraged to read the (publically available) book [46] that covers more topics.

3.1 Machine Learning Basics

From [47], *Machine learning* is the study of algorithms that can automatically detect patterns in data and later use the uncovered patterns to predict future data. Machine learning algorithms[1] can be roughly divided into two categories depending on the nature of the available data.

Supervised Learning algorithms handle a dataset containing both features and the corresponding labels. The supervised learning algorithms can be considered as learning a function that maps vectors of input features to their labels, following the examples of the given sample dataset.

Unsupervised Learning algorithms learn to study the helpful properties of the dataset from their describing features. When it comes to deep learning, we usually want to learn the entire probability distribution that generated a dataset. One common task for unsupervised learning is clustering, which goal is to discover groups with similar feature patterns by their distributions.

We will only discuss in detail about supervised learning. In the following, we abusively refer to supervised learning when we talk about machine learning. We begin with the definition of a machine learning algorithm from the perspective of statistical learning theory and then proceed to discuss the main challenges faced in this field, namely underfitting and overfitting. Additionally, most learning algorithms require settings of their so-called hyper-parameters before training; we will explain how to use extra data to set these hyper-parameters. Finally, after finishing training a model, we will describe how to measure and compare the performances of different models.

3.1.1 Statistical Learning Theory

We consider an input space $\mathcal{X}$ (aka *feature space*) and an output space $\mathcal{Y}$ (aka *label space*). The pairs $(x, y) \in \mathcal{X} \times \mathcal{Y}$ are drawn from an unknown joint distribution function $P(x, y)$. In the perspective of statistical learning theory, the supervised

[1] we will only discuss here "Example-based Machine Learning", and not at all "Reinforcement Learning" [48], another domain of Machine Learning dealing with interaction-based learning.

learning process is that of choosing from a given set of functions, so-called machine learning *models*, $f(x;\theta) \in \mathcal{H}$: $\mathcal{X} \to \mathcal{Y}$, parameterized by $\theta \in \Theta$, for some parameter set Θ, the one that best estimates the conditional distribution $P(y|x)$. The set $\mathcal{H}$ of all candidate models is called the *Hypothesis Space*.

Risk Minimization The *risk* $\mathcal{R}(\theta)$ of a function $f(x;\theta)$ is defined as the expected loss:

$$R(\theta) = \int \mathcal{L}(y, f(x;\theta))dP(x,y) = \mathbb{E}[\mathcal{L}(y, f(x;\theta))] \tag{3.1}$$

where $\mathcal{L}$ denotes the loss function that measures the difference between $f(x;\theta)$ and y. The goal of a learning algorithm is to seek the function $f(x;\theta^*)$ that minimizes the risk $R(\theta)$ over the class of functions $f(x;\theta)$, for $\theta \in \Theta$.

$$\theta^* = \arg\min_{\theta} \mathcal{R}(\theta), \quad \theta \in \Theta \tag{3.2}$$

Empirical Risk Minimization In general, $R(\theta)$ cannot be computed exactly, as the joint distribution $P(x,y)$ is unknown. Given a training set $\mathcal{D} = \{(x_i, y_i), i = 1.., N\}$, a sequence of independent identically distributed (iid) features and labels (x,y), $R(\theta)$ can be approximated by the empirical loss over the training set $\mathcal{D}$, the so-called the *empirical risk*:

$$\hat{\mathcal{R}}(\theta) = \frac{1}{N}\sum_{i}^{N} \mathcal{L}(f(x_i;\theta), y_i) \tag{3.3}$$

Note that if the number of samples is small, the empirical risk will introduce a statistical uncertainty, resulting in a poor estimate of the expected risk.
The empirical risk minimization principle states the learning algorithm should choose a function $f(x;\hat{\theta})$ which minimizes the empirical risk $\hat{\mathcal{R}}(\theta)$.

$$\hat{\theta} = \arg\min_{\theta} \hat{\mathcal{R}}(\theta), \quad \theta \in \Theta \tag{3.4}$$

Estimation and Approximation Errors As described above, a learning algorithm chooses a function $f(.,\theta)$, that is, the parameter θ from a parameter

space Θ based on a limited number of examples.
Suppose that $f^* : \mathcal{X} \to \mathcal{Y}$ is a function that achieves the minimal risk among all possible functions.

$$f^* = \arg\min_{f \in \mathcal{H}} \mathbb{E}[\mathcal{L}(y, f(x))] \tag{3.5}$$

The function $f^*(x)$ is the ideal goal of supervised learning, while $f(x, \hat{\theta})$ is the best that can be obtained using learning algorithms. The error in terms of risk is :

$$\mathcal{R}(f^*) - \mathcal{R}(\hat{\theta}) = \underbrace{\mathcal{R}(f^*) - \mathcal{R}(\theta^*)}_{\varepsilon_{app}} + \underbrace{\mathcal{R}(\theta^*) - \mathcal{R}(\hat{\theta})}_{\varepsilon_{est}} \tag{3.6}$$

- ε_{app} is the *approximation error*, measuring how much inductive bias we have by restricting ourselves to the specific class of functions $f(.;\theta) \in \mathcal{H}, \theta \in \Theta$. This term is independent of the number of examples. Enlarging the class of chosen functions can help to decrease the approximation error.

- ε_{est} is the *estimation error*, which measures the effect of minimizing the empirical risk $\hat{R}$ instead of the expected risk R. The estimation error depends on the training examples and the complexity of the chosen class of functions. When the training set is not large enough, the empirical risk $\hat{R}$ can not approximate the expected risk R well, which leads to an increase of the estimation error. Choosing a very rich class of functions can reduce the approximation error, in the meanwhile, the estimation error might increase as the size of the space of possible solutions increases, making the search more difficult.

3.1.2 Underfitting and Overfitting

The ultimate goal of supervised learning is to identify a function that is able to predict the label of previoulsy unseen data. Such prediction is called *generalization*. The generalization error, also called the *test error* can be empirically estimated by measuring the empirical loss on a set of examples, called the *test set*, disjoint from the training set, but has been generated according to the same joint distribution $P(x, y)$.
There are hence two goals for a learning algorithm:

1. minimize the training error,

2. minimize the gap between the training error and the test error.

The model performance is measured by the two factors corresponding to two major pitfalls in machine learning: Underfitting and Overfitting. If the hypothesis space $\mathcal{H}$ (the set of functions to select under a learning algorithm) is too small, the approximation error ε_{app} will be large, and the model will struggle to fit the training set, which is known as *underfitting*. *Overfitting*, on the other hand, occurs when there is a great gap between the training error and test error, in particular when the estimation error ε_{est} is large. As stressed above, a too large hypothesis space or a too small size of training set can both lead to overfitting: the training data is learned quasi-"by heart", and the generalization performance is poor. A first way to control Overfitting and underfitting is a careful selection of $\mathcal{H}$.

Regularization is a common approach used to prevent the model from overfitting by adding constraints on the hypothesis space $\mathcal{H}$. The most commonly used method of regularization is simply adding some penalty term $r(\theta)$, typically some norm of θ, to the loss function. The new target of the minimization problem becomes:

$$J(\theta) = \frac{1}{N}\sum_{i}^{N} L(f(x_i, \theta), y_i) + \lambda r(\theta) \tag{3.7}$$

where λ is a hyper-parameter that controls the importance of the penalty term $r(\theta)$.

Typical regularization terms are LASSO [49] and RIDGE [50]. The first use L1-norm of θ as the regularizer $r(\theta) = ||\theta||_1$, and the latter consider the L2-norm to regularize the parameters $r(\theta) = ||\theta||_2$. By adding these terms, optimizers are encouraged to find a model of parameters with a small norm, i.e., a simpler model.

3.1.3 Hyper-parameters setting

Most machine learning algorithms have themselves parameters, called *hyper-parameters*, that should be set before running them to control their behavior. The choice of the hyper-parameters of a learning algorithm can greatly affect its performance, both in the quality of the solution and computational cost. Since

the learning algorithm aims at minimizing the generalization error, it is necessary to compare models associated with different hyper-parameters on unseen data to avoid overfitting.

Validation Set A *validation set* is yet another set of examples that are used to tune the hyper-parameters of the learning algorithm. It should follow the same distribution as the training set and the test set but must be drawn independently. A general procedure is to split the training set into two subsets. One subset is used to learn parameters θ of the model, and it is still called the training set. The other one is the validation set, used to estimate the generalization errors after training with different hyper-parameters and set the hyper-parameters accordingly.

Cross-Validation The *cross-validation* technique is commonly used in machine learning to select hyper-parameters. As stressed above, a dataset is split into three parts, a training set to learn the parameters θ, a validation set for hyper-parameters setting, and a test set for final evaluation. Typically, a validation set only takes a small percentage (less than 20%) of the dataset. If the number of samples on a validation set is small, it will inevitably introduce statistical uncertainty when estimating the error. Cross-validation is used to decrease this uncertainty.
K-fold cross-validation is the most commonly used. The training set is first randomly partitioned into k equal-sized subsets. At each step, a single subset is extracted as the validation set, and the remaining $k-1$ subsets are used to train the model, which is tested on the extracted validation set. The process is repeated k times so that all examples are used for both training and validation. The generalization error used to set the hyper-parameters is the average over the k folds of the performance on the validation set.

3.1.4 Model Evaluation

Once the hyper-parameters have been set, we will estimate the generalization error of the learned model by observing its performance on the test set. Ideally, the results should be averaged over several independent training+test sets, as in the k-fold cross-validation. The averaged results can reduce the irreducible uncertainty during the model training, such as noisy data and parameter initialization on

deep learning. However, because the number of available examples is limited, the learning procedure is repeated using different splits of the available training data. In the present work, we employ the k-cross validation to form k-training sets. After the k trials of training, the generalization error is then estimated by averaging the test errors on all trained models.

Statistical tests are statistical tools to decide whether the data at hand sufficiently support a particular hypothesis $\mathcal{H}$. The standard process of a statistical test is as follows: calculate from sample data a test statistic T, compute the p-value, the probability of sample data under the null hypothesis, based on the distribution of T, and finally decide whether to reject the hypothesis $\mathcal{H}$ with a given level of confidence from the p-value. The statistical tests can basically be divided into three categories based on the nature of $\mathcal{H}$, one-sample tests, two-sample tests, and paired tests. The one-sample tests are appropriate when comparing the sample data to a population from $\mathcal{H}$. Two-sample tests are mainly used to compare two random and independent sample data, each sampled from a different population. The purpose is to determine whether the two populations are statistically significantly different. Same as two-sample tests, paired-sample tests are designed to compare two samples of equal sizes. Rather than directly comparing two sets, the test computes the differences between each pair to create a new sample. Methods from one-sample tests are then applied to decide whether to reject or not the hypothesis $\mathcal{H}$.

In supervised machine learning, statistical tests are used combined with cross-validation to decide whether two machine learning algorithms (or two hyper-parameter settings of the same algorithm) are statistically significantly different. The $\mathcal{H}_0$ hypothesis is that the results of the two models are sampled from the same distribution. After applying k-fold cross-validation when training each model, we obtain k evaluation scores. These scores are considered as the sample data and used to calculate the test statistic T. In such cases, paired-sample tests are appropriate as sample data from the two models are paired and observed. Pair-sample tests can be conducted, among others, with the following methods: z-test [51], t-test [52], and Wilcoxon signed-rank test [53].

T-test is a type of statistical hypothesis test in which, under the null hypothesis,

the test statistic T follows a student t-distribution. It assumes that the distribution of the sample data is normal. Additionally, the sample variance follows a κ^2 distribution. Because of the central limit theorem, the normalized sum of N identical random variables converges toward a normal distribution as N goes to infinity, whatever the original variables. Therefore, the t-test is fairly robust against the assumption of normality. For model evaluation where the sample (X_i, Y_i) is described as paired measurements, we suppose that the differences $d_i = X_i - Y_i$ follow t-distribution. T-test is mostly used for small sample data, especially when the sample size is smaller than 30.

Algorithm 6 Paired t-test

Require: A paired sample test $\{(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)\}$

1: Compute the absolute difference $d = \{X_1 - Y_1, ..., X_n - Y_n\}$
2: Calculate the test statistic $T = \frac{\bar{d}}{s_d/\sqrt{n}}$, where $\bar{d}$ and s_d are the mean and standard deviation of the difference d, $s_d = \sqrt{\frac{\sum_i (d_i - \bar{d})}{n-1}}$
3: Produce a p-value from T under the assumption that the test statistic T follows a t-distribution with the freedom degree of n-1.

Z-test assumes that the distribution of the test statistic T can be approximated by a normal distribution under the null hypothesis. Same as t-test, z-test requires that the sample data follow a normal distribution. To perform a z-test, the sample size should be larger than 30, and the population deviation σ must be known. The latter is rarely satisfied in practice as σ is difficult to determine.

Algorithm 7 Paired z-test

Require: A paired sample test $\{(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)\}$, the population standard deviation of paired differences σ.

1: Compute the absolute difference $d = \{X_1 - Y_1, ..., X_n - Y_n\}$
2: Calculate the test statistic $T = \frac{\bar{d}}{\sigma/\sqrt{n}}$, where $\bar{d}$ of the difference d, and σ is the known population standard deviation.
3: Produce a p-value from T under the assumption that the test statistic T follows a normal distribution $\mathcal{N}(0, 1)$.

Wilcoxon Signed-Rank Test is a non-parametric statistical hypothesis test, i.e., a test that does not assume any particular form for the distribution of sampled data (unlike t-test and z-test that assume normality of the data sampled).

Algorithm 8 Wilcoxon Signed-Rank Test

Require: A paired sample test $\{(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)\}$

1: Compute the absolute difference $d = \{|X_1 - Y_1|, ..., |X_n - Y_n|\}$

2: Sort $\{|X_1 - Y_1|, ..., |X_n, Y_n|\}$

3: Use the sorted list to assign ranks $R_1, ..., R_n$. The rank of the smallest observation is 1, and the next smallest observation is 2, until we rank the largest as n.

4: Calculate the test statistic T as the signed-rank sum:

$$T = \sum_{i=1}^{N} sgn(X_i - Y_i)R_i$$

5: Produce a p-value from T under the null hypothesis

3.2 Deep Learning

This section briefly introduces the basic concepts of Deep Learning that have been used in the rest of this work. The basic block of deep neural networks, the multi-layer perceptron (MLP), is discussed initially. The optimization procedures that are utilized for training deep learning models, such as stochastic gradient descent, are also surveyed. Additionally, the commonly used initialization strategies, such as Xavier initialization and Kaiming initialization, are discussed. Finally, the section ends by presenting some common loss functions used for both classification and regression problems, such as cross-entropy and mean squared error, and also some feature scaling methods.

3.2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs), or simply *Neural Networks* (NNs) [2] [54, 55] are artificial systems inspired by biological neural networks. It is hoped that ANNs are capable of performing complex tasks like the human brain, where traditional algorithms barely succeed.

Neurons are the basic building block for Neural Networks. The design of a neuron is inspired by biological neurons found in biological brains. A neuron is a computational unit that takes as inputs a vector of real values $\mathbf{x} = (x_1, x_2, ..., x_n)$, computes their weighted sum using local *weights* $\mathbf{w} = (w_1, w_2, ..., w_n)$ and a bias term b. The weighted sum is called *activation*. It then applies to this activation a non-linear function g, known as the *activation function* to produce the output $y = g(w^T x + b)$.

Activation Function In general, the activation function g is non-linear, as such non-linearity allows the neural network to be more expressive than a purely linear model. The first activation function, used in the Boolean context, was the discontinuous *Heaviside function* [56]. With the development of continuous NNs, several activation functions have been proposed: the continuously differentiable Sigmoid activation was long used, as well as the tanh activation, allowing the gradient back-propagation algorithm, until Deep Learning popularized the

[2] the word "artificial" will be omitted from thereon, for NNs, neurons, etc.

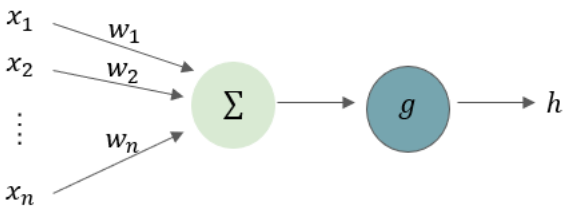


Figure 3.1: The architecture of an artificial neuron: $h = g(w^T x + b)$

continuous but not differentiable at 0 ReLU activation [57], to avoid gradient vanishing problems existing when using Sigmoid. (see Figure 3.2). However, they have sub-gradients, and gradient methods can nevertheless be applied: in the following, for simplicity, we will nevertheless talk about gradients. Later, leaky ReLU and ELU activations [58, 59] were proposed based on ReLU to solve the dying neuron problem, where many ReLU neurons only compute values of 0 when the inputs are negative. Leaky ReLU and ELU both add a small slope for negative input instead of 0 in ReLU.

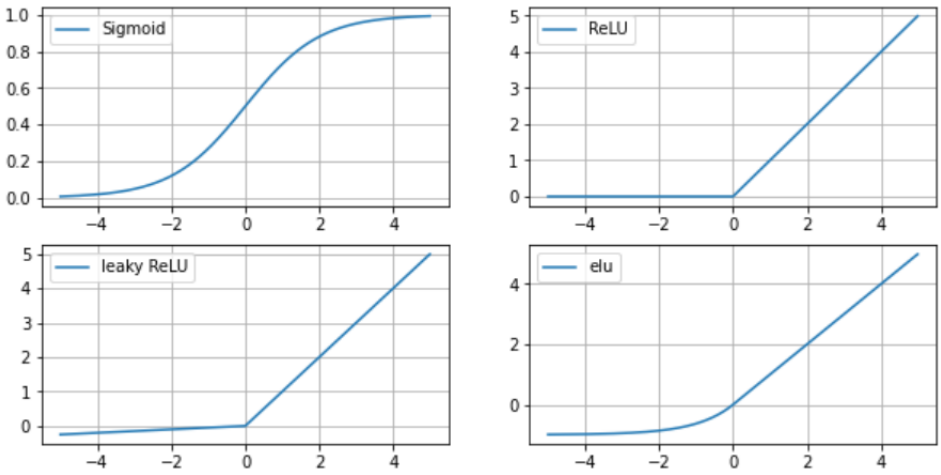


Figure 3.2: Some activation functions used by deep neural networks

Neural Networks are sets of neurons in which the outputs of some neurons are connected as inputs to other neurons. Some neurons also received some external inputs – the inputs of the NN. And the outputs of other neurons are tagged as the outputs of the network. A NN can hence be represented as a directed graph,

in which nodes represent artificial neurons and edges connect outputs and inputs between neurons. Based on the properties of the connection graph, ANNs can be classified into two categories, feedforward neural networks and recurrent neural networks.

Feedforward Neural Networks are networks without cycles in the directed graph of connections. Information only circulates through one direction, forward from the input neurons to the output neurons. Multi-Layer Perceptron (MLP) is the simplest feedforward neural network architecture. The neurons in MLP are organized in *layers*. MLP is nevertheless very expressive because of the use of a non-linear activation function: Hornik [33] proved a universal approximation theorem for 3-layers MLPs (one hidden layer between the inputs and the output of the network, see Figure 3.4) using sigmoid activations, i.e., they can approximate any continuous function to any precision provided they contain enough neurons in their hidden layer. The architecture of MLP is described in the figure 3.3

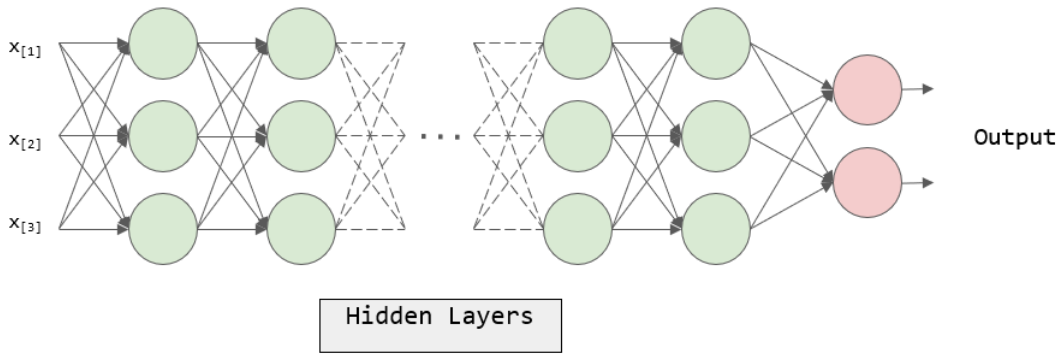


Figure 3.3: MLP Architecture

Recurrent Neural Networks In recurrent neural networks(RNNs), there exist cycles in the graph because of the *feedback* connections. The output from some neurons can impact the input from the same neurons by feedback paths. RNNs are commonly used to process a sequence of inputs with variable lengths, such as natural language processing.

3.2.2 Optimization procedure

Optimization is a crucial component of machine learning. As discussed in Section 3.1.1, machine learning algorithms involve solving an optimization problem, minimizing the empirical risk $\hat{\mathcal{R}}$ over the training set $\mathcal{D} = \{(x_1, y_1); (x_2, y_2); ...; (x_N, y_N)\}$:

$$\hat{\mathcal{R}}(\theta) = \frac{1}{N} \sum_{i}^{N} \mathcal{L}(f(x_i; \theta), y_i) \tag{3.8}$$

All supervised learning algorithms turn the learning problem into the problem of minimizing such a loss function (directly $\mathcal{R}$ or variants thereof, see Section 3.2.4). In deep learning, the non-linear activation functions causes loss functions to be nonconvex, making the global optimization very difficult. Nevertheless, deep learning models are usually trained using iterative gradient-based optimization algorithms, leading to an expectedly "good enough" local minimum. At each iteration, the current point moves following the opposite of the direction of the gradient:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^{t} - \eta \nabla \hat{\mathcal{R}}(\boldsymbol{\theta}^{t}) \tag{3.9}$$

where η is a positive scalar called the *learning rate*. η determines the step size. In practice, in order to enforce convergence, η is generally gradually decreased over time. In this part, we will discuss the *backpropagation* algorithm [22] to compute the gradient of differentiable loss functions, and introduce several optimization algorithms designed to train a deep learning model.

BackPropagation is widely used to calculate efficiently the gradient descent for feedforward neural networks. For simplification, let's consider the 3-layers MLP shown in Figure 3.4[3]. The forward propagation to compute the prediction $\hat{y}$ is :

$$h_1 = g(\mathbf{w}_1^T \mathbf{x}), \quad h_2 = g(\mathbf{w}_2^T \mathbf{x}), \quad \hat{y} = g(\mathbf{w}_3^T \mathbf{h}) \tag{3.10}$$

where $\mathbf{w}_i$ denotes the local weights on the neuron i, g is the activation function, and $\mathbf{h} = [h_1, h_2]$ is the hidden state. Moreover, we use $a_i = \mathbf{w}_i^T \mathbf{h}$ to represent the activation of each neuron. During the forward propagation, every activation a_i is

[3]In the historical terminology, the inputs x_i are considered as a first layer – hence the name

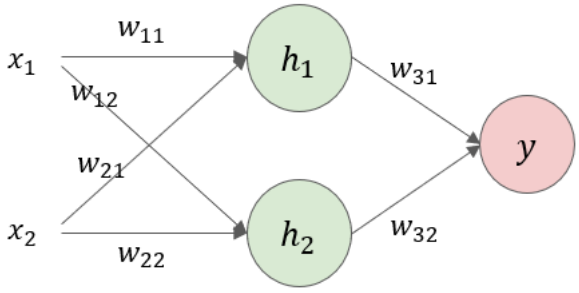


Figure 3.4: A three layers neural network

saved and will be further used to calculate the gradients.
The gradients w.r.t each weight are computed reversely by the *chain rule*. [60]:

$$\begin{gathered}
\frac{\partial \hat{R}}{\partial a_3} = \frac{\partial \hat{R}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_3} = \frac{\partial \hat{R}}{\partial \hat{y}} g'(a_3) \\
\frac{\partial \hat{R}}{\partial w_{31}} = \frac{\partial \hat{R}}{\partial a_3} \frac{\partial a_3}{\partial w_{31}} = \frac{\partial \hat{R}}{\partial a_3} h_1; \quad \frac{\partial \hat{R}}{\partial w_{32}} = \frac{\partial \hat{R}}{\partial a_3} h_2 \\
\frac{\partial \hat{R}}{\partial a_1} = \frac{\partial \hat{R}}{\partial h_1} \frac{\partial h_1}{\partial a_1} = \frac{\partial \hat{R}}{\partial a_3} g'(a_1) w_{31}; \quad \frac{\partial \hat{R}}{\partial a_2} = \frac{\partial \hat{R}}{\partial a_3} g'(a_2) w_{32} \\
\frac{\partial \hat{R}}{\partial w_{11}} = \frac{\partial \hat{R}}{\partial a_1} x_1, \frac{\partial \hat{R}}{\partial w_{21}} = \frac{\partial \hat{R}}{\partial a_1} x_2; \quad \frac{\partial \hat{R}}{\partial w_{12}} = \frac{\partial \hat{R}}{\partial a_2} x_1, \frac{\partial \hat{R}}{\partial w_{22}} = \frac{\partial \hat{R}}{\partial a_2} x_2
\end{gathered} \tag{3.11}$$

By observing the equations above, we can notice that the gradients on layer i can be efficiently computed from the gradients on layer $i+1$ and activation values recorded during forward propagation. The backpropagation, computing gradients iteratively from the last layer, avoids redundant calculations in the chain rule.

Stochastic Gradient Descent can be considered as a stochastic approximation of gradient descent optimization, and it is commonly used to solve deep learning optimizer problems. Computing the exact gradient $\nabla\hat{\mathcal{R}}$ can be expensive, especitally bevause deep learning also requires a large training set. SGD proposes to estimate the gradient using only a small number of examples (minibatch) from the training set. See Algorithm 9 for a straightforward implementation of SGD.

SGD with Momentum Sometimes, using SGD to optimize the target can be slow, as there is no way to ensure that SGD goes in the right direction at each iteration because of the noisy gradient approximator due to using mini-batches.

Algorithm 9 Stochastic Gradient Descent Updates [46]

Require: Learning rate schedule η_1, η_2, ...
Require: Initialize the parameters θ
1: **for** $k = 1, 2, ...$ until stop criterion met **do**
2: Sample a minibatch of m examples from the training set $\{(x_i, y_i)\}_{i=1}^{m}$
3: Compute the gradient estimation $\hat{\mathbf{g}} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i \mathcal{L}(f(x_i, \boldsymbol{\theta}), y_i)$
4: Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta_k \hat{\mathbf{g}}$
5: **end for**

The *SGD with momentum* algorithm was proposed to cope with this issue: it accumulates an exponentially decaying moving average of past gradients and continues to move into their directions, the *momentum* $\boldsymbol{\nu}$. The update rules of SGD with momentum become:

$$\boldsymbol{\nu} = \alpha\boldsymbol{\nu} - \eta\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(x_i, \boldsymbol{\theta}), y_i), \quad \boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{\nu} \tag{3.12}$$

Adam Optimizer The choice of learning rate η has a significant impact on the final model performance. Moreover, the loss is often sensitive to some directions of θ and less sensitive to others. Using a non-identical learning rate for each parameter can be more reasonable. In recent years, a number of optimizers have focused on adapting the learning rate of model parameters, such as AdaGrad[61] and RMSProp [62]. These algorithms assign each parameter an individual learning rate. AdaGrad keeps track of the per-parameter sum of squared gradients, which is used to scale the learning rate of each parameter. While RMSProp is a small modification on AdaGrad, that calculates the exponentially decaying moving average of squared gradients to normalize the learning rate to avoid making learning rates too small. Adam optimizer [63] is probably today the most widely used among all the existing adaptive optimization algorithms, and the one we have used throughout this thesis. It is detailed in Algorithm 10.

By default, Adam uses a decay rate of 0.9 on ρ_1 for the first moment estimate, 0.999 on ρ_2 for the second moment estimate, and a small constant of $1e-8$ on δ to prevent division by zero. These default values have been found to be effective and robust for a wide range of supervised learning problems. Similar to RMSProp and AdaGrad, the Adam optimizer stores, in addition, the exponentially decaying

Algorithm 10 Adam Algorithm [46]

Require: Learning rate η
Require: Exponential decay rates for moment estimates ρ_1, ρ_2
Require: Small constant δ used for numerical stabilization
Initialize the parameters θ ▷ (see Section 3.2.3)
Initialize two moments $\mathbf{s}, \mathbf{r}$ to 0
for $k = 1, 2, ...$ until stop criterion met **do**
    Sample a minibatch of m examples from the training set $\{(x_i, y_i)\}_{i=1}^{m}$
    Compute the gradient estimation $\hat{\mathbf{g}} = \frac{1}{m}\nabla_\theta \sum_i L(f(x_i, \theta), y_i)$
    Update $\mathbf{s} = \rho_1 \mathbf{s} + (1 - \rho_1)\hat{\mathbf{g}}$
    Update $\mathbf{r} = \rho_1 \mathbf{r} + (1 - \rho_2)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
    Correct the bias $\hat{\mathbf{s}} = \frac{\mathbf{s}}{1-\rho_1^k}$, $\hat{\mathbf{r}} = \frac{\mathbf{r}}{1-\rho_2^k}$
    Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta}$
end for

moving average of squared gradients. Adam also uses momentum to determine the moving direction.

Learning Rate Decay In practice, training neural networks by annealing the learning rate has proved helpful in many works and for several different network architectures, such as ResNet [64] and DenseNet[65]. Initially, the learning rate is set to a large value in order to accelerate the move toward some optimal region. It is then gradually decreased to fine-tune the local minimum. Several strategies have been proposed for learning rate decay:

1. Step Decay: Decay the learning rate by some multiplicative factor at a fixed interval of *epochs*, the number of training iterations over the dataset. The factor by which the learning rate is reduced, and the interval at which the reductions are applied, are two the hyperparameters of step decay. e.g.: In Figure 3.5, the learning rate is multiplied by a factor of 0.5 every 20 epochs.

2. Exponential Decay: Decay the learning rate by some factor γ every epoch, the learning rate after t epochs is $\eta_t = \eta_0 \times \gamma^{t-1}$. The factor γ is a hyperparameter to set.

3. Automatic Decay: Dynamic learning rate reducing when the validation error has stopped improving, e.g.: multiply a factor 0.5 to the current learning

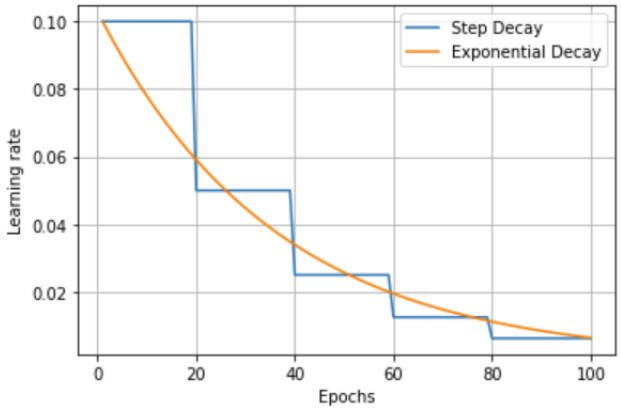


Figure 3.5: Different Learning Rate Decay Schedules

rate if no improvement on the validation error is observed for a period of 20 epochs.

3.2.3 Weight Initialization

Deep neural networks are trained by iterative methods, thus requiring the user to specify initial values of trainable parameters. Indeed, the initialization strongly influences the convergence and the final performance of the model. Initializing all the parameters with a constant value has experimentally proved to be detrimental. This leads all neurons to learn the same features during training. Too small initial weights will slow down the convergence, while too large initial weights lead to numerical divergence. The initial weights should be chosen carefully for efficient training. Two strategies are commonly used, Xavier [66] and Kaiming [67] initializations.

Xavier initialization is designed to ensure that the variances of inputs and outputs after each layer are similar. Given uniform distribution as an example, Xavier initialization limits the weights into a range:

$$w^{[l]} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n^{[l]}+n^{[l-1]}}}, +\frac{\sqrt{6}}{\sqrt{n^{[l]}+n^{[l-1]}}}\right) \tag{3.13}$$

where $n^{[l]}$ represents the number of neurons at layer l.

In practice, Xavier initialization works very well in cases where the activation

function is symmetric, such as Sigmoid.

Kaiming initialization is commonly used with ReLU activation functions (see Figure 3.2) :

$$w^{[l]} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n^{[l-1]}}}, +\frac{\sqrt{6}}{\sqrt{n^{[l-1]}}}\right) \tag{3.14}$$

3.2.4 Loss functions and Evaluation Metrics

The empirical risk $\hat{\mathcal{R}}(\theta) = \frac{1}{N}\sum_i^N \mathcal{L}(f(x_i, \theta), y_i)$ is based on a *loss function* $\mathcal{L}$, which measures how different the prediction $f(x_i, \theta)$ and the ground truth y_i are. On the other hand, the *evaluation metrics* are used to measure the performance of a trained model. In this Section, we will discuss some frequently used loss functions and metrics for both classification and regression problems.

Classification Problem In classification problems, the accuracy [68] is the most commonly used metric to measure the overall performance of the model. The accuracy is the proportion of accurate predictions within the whole sample set. However, the metric accuracy is not differentiable, and hence it is rarely used as a loss function. Other metrics, such as precision, recall, and F1-score, are also employed frequently, especially when datasets are imbalanced. The precision measures the proportion of correctly classified positive samples among overall samples predicted to be positive. Recall calculates the proportion of correctly classified positive samples among overall true positive samples. While F1-score is the harmonic mean of precision and recall to provide a balance between the two metrics. *Cross-entropy loss* [69], on the other hand, is a common loss function for classification problems. The notion of cross-entropy comes from information theory, and it compares the similarity of two probability distributions p and q. The definition is expressed as:

$$H(p, q) = -\mathbb{E}_p[\log q] \tag{3.15}$$

Back to a classification problem with C classes, the prediction $\hat{\mathbf{y}}_i \in [0, 1]^C$ represents the probabilities for the example $\mathbf{x}_i$ to belong to the existing classes $1, \ldots, C$, and the ground truth $\mathbf{y}_i$ on $\mathbf{x}_i$ is *one-hot* encoded, that is the value y_{ic}

of $\mathbf{y}_i$ at index c is 1 if and only if x_i belongs to the class c. The cross-entropy between the prediction and the actual probability distribution is:

$$H_i = -\sum_{c=0}^{C} y_{ic} \log \hat{y}_{ic} \tag{3.16}$$

where $\hat{y}_{ic}$ represents the score of the output $\hat{y}_i$ at the class c.
Besides being differentiable, the cross entropy is also convex. The convexity of a loss function is important, especially in the context of traditional machine learning algorithms, such as logistic regression. The cross-entropy loss in logistic regression ensures that the optimization problem is convex, meaning that the local minimum is also the global minimum.

Regression Problem For regression problems, the output y is a vector with continuous values. Mean Squared Error (MSE) [70] and Mean Absolute Error (MAE) [71] are the two most commonly used loss functions used for regression problems.

$$MSE = \frac{1}{N}\sum_{i}^{N}(y_i - f(x_i;\theta))^2 \tag{3.17}$$

$$MAE = \frac{1}{N}\sum_{i}^{N}|y_i - f(x_i;\theta)| \tag{3.18}$$

MAE computes the absolute distance between the ground truth and the network prediction, and MSE measures the variance of the residual. It is more sensible to outliers compared to MAE but may be less robust since the squaring of the error impose a higher emphasis on outliers.
There are several evaluation metrics especially designed for regression problems, such as the coefficient of determinant R^2 [72] and relative error [73]. They both take into consideration the order of magnitude of the target.

$$R^2 = 1 - \frac{\sum_i^N (y_i - f(x_i,\theta))^2}{\sum_i^N (y_i - \bar{y})^2} \tag{3.19}$$

where $\bar{y}$ is the mean of the ground truth. R^2 indeed calculates the ratio of MSE and the variation of the ground truth.
The relative error compares the error from model predictions and that achieved by a simple predictor. More specifically, the simple predictor is usually the average of the actual data. Here is an example of relative absolute error (RAE):

$$RAE = \frac{\sum_i^N |y_i - f(x_i,\theta)|}{\sum_i^N |y_i - \bar{y}|} \tag{3.20}$$

3.2.5 Feature Scaling

Feature Scaling normalizes the range of the features to avoid numerical issues and eventually accelerate the convergence of the gradient descent. Since the range of values for each feature can vary widely, the gradient descent used to optimize most machine learning algorithms works much more efficiently with feature scaling. There are several methods to scale the features.

Min-Max Scaling is the simplest scaling method to scale each feature x to the range $[0, 1]$ by min and max values on this feature:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.21}$$

Standardization convert values of each feature with zero mean and unit variance. This is the most commonly used method in machine learning, e.g., SVM, logistic regression, and NNs, etc.

$$x' = \frac{x - \tilde{x}}{\sigma} \tag{3.22}$$

with $\tilde{x}$ the mean and σ the variance of the feature x.

Scaling to unit length simply scales the component of a vector $\mathbf{x}$ of input features such that the norm of $\mathbf{x}$ equals to 1 after scaling.

$$\mathbf{x}' = \frac{\mathbf{x}}{||\mathbf{x}||} \tag{3.23}$$

3.2.6 Hyperparameters in Neural Networks

Neural Networks have a large number of hyperparameters that need to be adjusted to optimize their performance. According to the previous discussion of neural networks, hyperparameters can be roughly divided into four parts:

- **Structure/topology of the Neural Network**: Hyperparameters from the structure involve the number and the type of layers in the network, the number of neurons in each layer, the activation function used by each layer, etc. They determine the overall capacity of the network. Increasing

the number of layers and neurons can improve the ability of the model to adapt to the training data, but it also increases the risk of overfitting. Other topology parameters include skip connections, introduced in ResNet [64], and used in U-Net [29] that has been inspirational for this work (See Section 3.3.1).

- **Optimization**: The choice of the optimizer, together with its own associated hyperparameters, such as the learning rate, is crucial for training the network. Moreover, there are some other hyper-parameters to set up for learning rate decay strategies as illustrated (Figure 4.6). Taking the step decay as an example, hyperparameters include the multiplicative factor and the initial step size.
  The batch size, and the number of training examples used in each iteration of the learning also influence the training. In general, a larger batch size can provide more accurate estimates of the gradient of the loss function, which can lead to faster and more stable convergence of the learning algorithm. However, a larger batch size also requires more memory and computational resources, which can limit the size of the network or the amount of data that can be used for training. On the other hand, a smaller batch size can provide more noisy estimates of the gradient, which can make the learning algorithm more sensitive to the specific initialization of the weights and biases. This can lead to better generalization, but it can also make the learning process more costly and less stable.

- **Weight Initialization**: As discussed in Section 3.2.3, there are mainly two weight initialization strategies used nowadays, Xavier and Kaiming initialization. The two strategies are designed to make the variances of inputs and outputs after each layer similar to avoid gradient vanishing or exploding.

- **Loss function**: Different loss functions are designed for different types of problems (Section 3.2.4) and can provide different trade-offs between bias and variance. In general, the loss function should be chosen based on the specific data and problem at hand. It is important to select a loss function that is appropriate for the task and that provides a meaningful and effective measure of the error between the prediction and ground truth.

- **Other hyperparameters**: Several other variants of topology, optimization procedure, etc. have been proposed, like gradient clipping, batch-normalization layers, dropout, etc. [74–76], but have not been used in this work, and we will not detail them here.

Tuning these hyperparameters can be a challenging and time-consuming task, but it is an important part of building a successful neural network model.

3.3 Convolutional Neural Networks

The Convolutional Neural Networks (CNNs) have been proposed by Yann LeCun, and co-authors [77] back in 1989! CNNs are specially designed to handle images, i.e., data with grid-like topology. In this section, we will outline the structure and properties of CNNs. Moreover, we will discuss a famous architecture based on CNNs, UNet [29], designed for image segmentation tasks.

3.3.1 The Convolution Operator

The convolution of two functions X and W is defined as:

$$(X * W)(t) = \int_{-\infty}^{+\infty} X(\tau)W(t-\tau)d\tau \tag{3.24}$$

In images, X and W are both discrete, and the convolution operator becomes:

$$\begin{aligned}(X * W)(i,j) &= \sum_m \sum_n X(m,n)W(i-m, j-n) \\ &= \sum_m \sum_n X(i-m, j-n)W(m,n)\end{aligned} \tag{3.25}$$

In such context, X is often referred as the input and W as the kernel, or filter. In general, the kernel has small support, and this limits the summation above and gives the convolutional operator some nice properties when dealing with image datasets. Fully connected layers are not good at treating data with high-dimensional inputs such as images in consideration of the model complexity and memory requirements. Instead, in CNN layers, each neuron of a given layer is connected with only a small local region of the previous layer. The local connectivity of neurons helps to control the number of parameters for such high-dimensional data.

Moreover, the same filter is used at every location of the layer input: beyond limiting the number of weights, this ensures some invariance by translation of the whole process. We could expect that these filters can learn some local features at different locations.

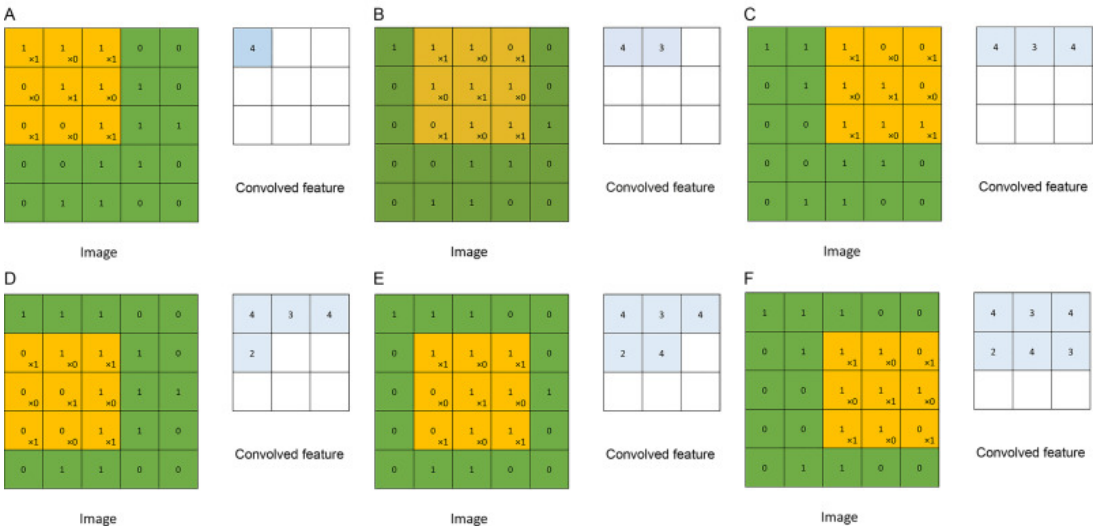


Figure 3.6: A Convolutional Operator (source from [78])

3.3.2 Pooling and Un-pooling

It is common to insert a pooling function after applying some CNN layers on images, as for instance in VGGNet [79]. The pooling function down-samples images to a smaller size. Due to the local connectivity, the filter receive information from a small region. The pooling layer helps filters to have a larger perspective of the input volume so that CNNs can extract hierarchical spatial features. In addition, the pooling layer can successively reduce the image size so as to reduce the computation time of CNNs The most common form of a pooling layer is

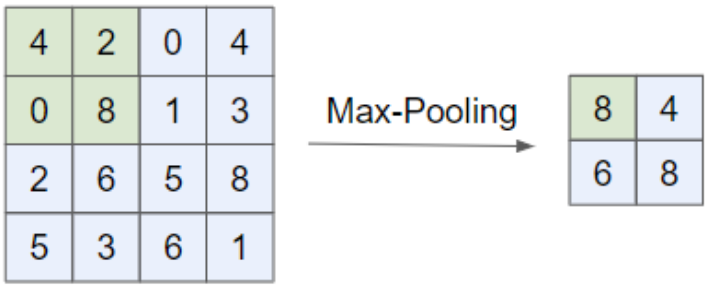


Figure 3.7: Pooling Operator with a filter of size 2×2 and a stride of 2

to apply a filter of size $(2, 2)$ with a stride of 2, as shown in Figure 3.7). The maximum or mean value of each region of size 2×2 is used to fill the corresponding pixel of the next layer. The process is repeated on the next non-overlapping region.

Contrary to the pooling operator, un-pooling up-samples the feature map to a higher resolution. Such operator is commonly used in image reconstruction tasks to recover the down-sampled feature maps. A simplest approach to achieve up-sampling is Nearest-Neighbor (In Figure 3.8). It copies values of the input image to the corresponding sub-region of the output image.

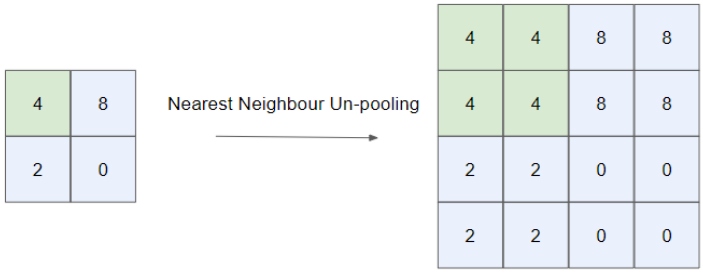


Figure 3.8: Nearest Neighbor un-pooling operator

3.3.3 The U-Net Architecture

For classification problems on images, the first successes were obtained with VGG [79], an architecture that alternates CNN and pooling layers until obtaining features of small dimensions. Later, ResNet [64], introducing *skip connections* to jump over one or more layers (Figure 3.9), was proposed to allow training very deep CNNs with impressive performance. Fully connected layers are then

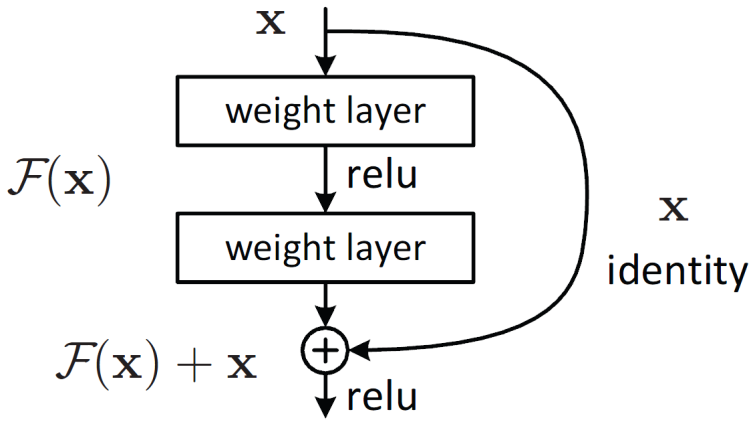


Figure 3.9: ResNet Architecture

used to compute the output (class scores).

However, for image reconstruction or image segmentation tasks, the output should be an image of same dimension than the input. In such context, many fully-convolutional models [80, 81] are proposed. In particuler, U-Net [29] is a CNN architecture specially designed for image reconstruction tasks. U-Net is one of the most famous encoder-decoder models, and was originally proposed for biomedical image segmentation. Figure 3.10 shows an example of U-Net architecture. The encoder part (on the left) regularly down-samples the image, applying a few CNN layers at each level to extract features at different granularity levels. The decoder (on the right) gradually up-samples these low-resolution features until reaching the original dimensions of the input. Furthermore, high-resolution features from the encoding part are directly linked together with the outputs of the up-sampled layers (horizontal arrows). These *skip connections* provide some extra information about features at all scales to the decoder.

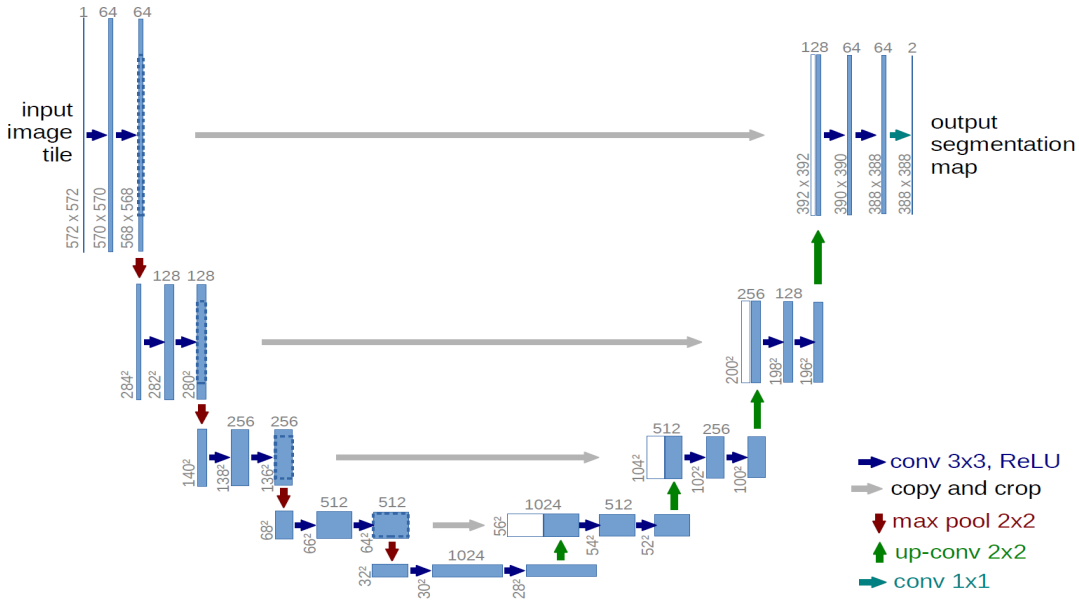


Figure 3.10: An example of U-Net Architecture [29]

An interesting remark in the context of the present work is that the U-Net architecture is very similar to the multi-grid method called V-cycle discussed in Section 2.3. They both use a hierarchical architecture where the information is transformed between different resolutions.

3.4 Graph Neural Networks

In the previous section, convolutional networks have been introduced. Due to their local connectivity and parameter sharing, they can effectively capture hidden patterns from data with grid-like topology. However, there are still an increasing number of applications involving data that is not grid-like, such as *graphs* that describe entities in relation with one another. For instance, molecules such as proteins are modeled as graphs in biology. Social Networks are represented by graphs. And at the heart of the work presented in this dissertation, the simulation of physical systems that are modeled by Partial Differential Equations (PDEs) involves data that is most often represented on a mesh structure, which can also be expressed as a graph. However, most of the early deep learning tools were designed to treat simple data types such as grids, sequence data, or fixed size vectors. Graphs are much more complex and harder to process by deep learning algorithms since graph nodes are connected in a non-ordered manner, and with a variable size of neighborhoods.

Motivated by the great success of CNNs on images, there are plenty of studies aiming at designing algorithms to define a convolutional operator on graph data, leading to the so called *graph neural networks* (GNNs). These algorithms can be divided into two categories, the *spectral-based* GNNs, and *spatial-based* GNNs. The first spectral-based GNN was proposed by [82], which defines a convolution operator in the Fourier space of a graph. From the convolution theorem [83], the convolution in the spatial domain is defined as the multiplication after Fourier transformation.

Since then, several works [84, 85] improved spectral-based GNNs. Meanwhile, some scientists attempt to construct the convolutions directly on the graph, called spatial-based GNNs. These approaches, such as [86–88], aggregate the feature information on each node from its neighbors. Moreover, there are several works made to solve problems with a set of points (aka point clouds). The PointNet [31] is the first neural network for point clouds. The basic idea is that each point feature is encoded and then aggregated to a global vector by a symmetric function. PointNet++ [89] improved the PointNet by introducing hierarchical structures to capture local features.

In this section, we will first describe the graph structure in detail with its properties and discuss how GNNs are designed to treat graph data. This work focuses exclusively on one type of GNNs, the message passing GNNs. For readers seeking a general presentation in this domain, we refer to the survey paper [90].

3.4.1 Data Set-Up

A graph is defined by a set of n nodes V and a list of edges E that describe the connections between two nodes, $G = (V, E)$. Let $v_i \in V$ denote a node and $e_{ij} = (v_i, v_j) \in E$ to denote an edge pointing from the nodes v_j to the node v_i. We represent the neighbourhood of node v_i as $N(v_i) = \{v_j \in V | (v_j, v_i) \in E\}$. For example, we can list all nodes $V = \{1, 2, 3, 4\}$ and edges $E = \{(1, 2), (1, 3), (2, 3), (3, 4)\}$ of the undirected graph in Figure 3.11 to represent this graph.

An alternative way to represent a graph is by using an adjacent matrix (in Figure 3.11). The adjacent matrix A is a matrix of size $n \times n$, where n is the number of nodes of the graph. $A_{ij} = 1$ only if there exists an edge e_{ij} in G, and $A_{ij} = 0$ if the two nodes v_i and v_j are not connected. The adjacent matrices are symmetrical for undirected graphs and are often extremely sparse in the real world.

Figure 3.11: Graph represented by adjacent matrix

Graph data can be attached to nodes or edges. For example, node attributes can represent the properties of a specific node, and edges can be assigned with weights to represent how strong the connection is. The matrix $X \in \mathbb{R}^{n \times d}$ denotes the property of each node by a vector $x_i \in \mathbb{R}^d$, and $X^e \in \mathbb{R}^{m \times c}$ denotes the edge attributes with $x^e_{ij} \in \mathbb{R}^c$ to describe the attribute on the edge e_{ij}.

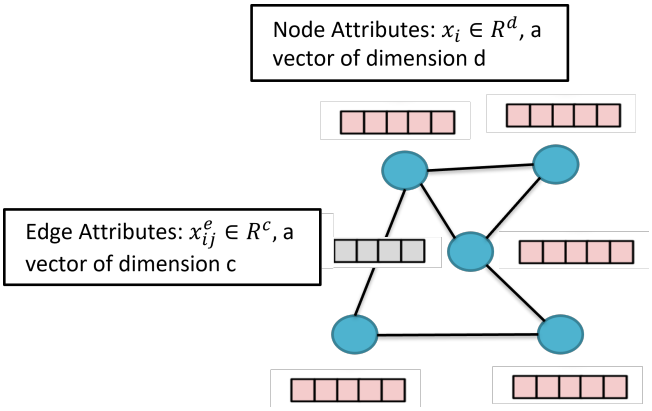


Figure 3.12: Graph Data

3.4.2 Challenges on graph data

Before the invention of Deep Learning, machine learning pipelines used handcrafted features such as node degrees, centrality, etc to train models on graph data. The choice of these handcrafted features greatly influences the final performance of the model. With the help of deep learning, the models are expected to better learn how to represent data on a graph, by alsolearning the features (end-to-end learning). As mentioned, the well-defined convolution operator of CNNs is designed for grid-like inputs and cannot be applied in general to graph data. To better learn on graphs, new operators should be defined based on some common properties of a graph.

Variable Input Size A natural way to define a neural network for graphs would be to simply use the adjacent matrix as input features, and feed it into a multi-layer perceptron. Such naive method, however, is not applicable for graphs of different sizes. When designing an architecture for graph data, the model should allow varying input sizes.

Interaction among Nodes An edge in a graph represents an interaction between two nodes, and these nodes are correlated through their neighborhoods. For example, in social networks, social connections can influence the individual characteristics of a person. Therefore, similar to CNNs, the model designed for graph data is expected to extract local structures from neighborhood.

Permutation equivariant A graph doesn't have a canonical order of the nodes meaning that a model needs to be equivariant to any permutation of the graph nodes: once inputs are permuted, the resulting objects will also be permuted consistently.

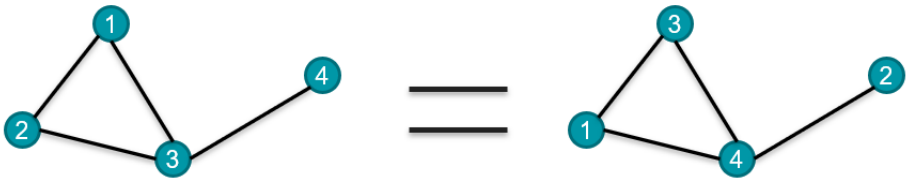


Figure 3.13: Permutation equivariant of the graph

A graph neural network should at least satisfy the three properties above. In next Section, we will outline a general framework of spatial-based GNNs, "Message-Passing Schema," which is designed to solve the above issues for graph data.

3.4.3 Message-Passing Schema

Let us first take a look at CNNs on image data: indeed, an image can be considered as simply a special case of graph data, by considering that there is an edge between neighboring pixels. Suppose that a single convolutional layer is applied with a filter of 3×3. What the convolutional layer does is taking an area of 3×3 from the image and apply some transformation to create a new pixel (Figure 3.14). In the perspective of graphs, the transformation simply multiplies edge weights to the corresponding neighbors and take a sum:

$$x_i^{t+1} = \sum_{j \in \mathcal{N}(i)} W_{ij} x_j^t \tag{3.26}$$

Each node receives information from its neighbors, from which it creates new features.

This is the basic idea of message-passing schema [91]. The general framework of message-passing treats the convolutional operator in a graph as a message-passing process in which information is passed along edges from one node to its

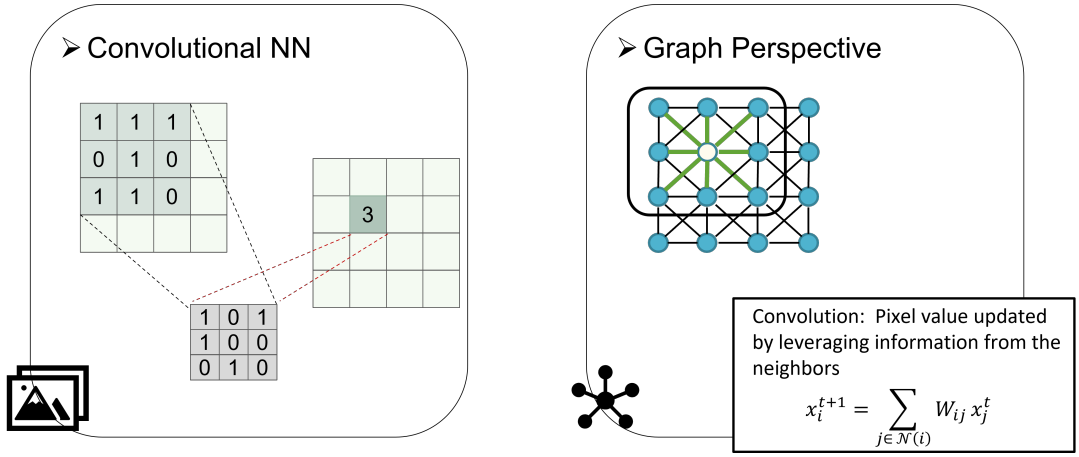


Figure 3.14: CNN operator is considered as a transformation of message in a graph

neighbors. The new feature on node v_i is calculated by aggregating information from its neighbors $v_j \in \mathcal{N}(i)$ using *permutation invariant functions*, i.e., functions that are invariant w.r.t. to permutations of node order, like Sum and Mean. The information propagation function is defined as:

$$x_i^{t+1} = \gamma(x_i^t, \sum_{j \in \mathcal{N}(i)} \phi(x_i^t, x_j^t, x_{ij}^e)) \tag{3.27}$$

where both γ and ϕ are parametric functions, MLPs, for example, and the Sum operator can be replaced by any permutation invariant functions accepting a variable number of variables. The architecture allows different graph sizes, and it

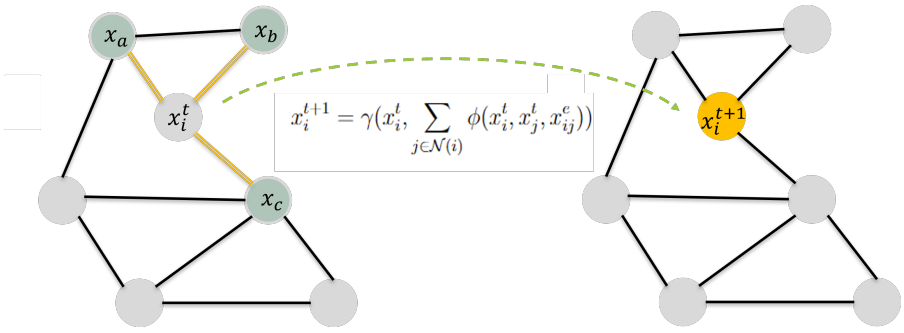


Figure 3.15: Message Passing Schema

also helps to extract local features by receiving neighbors' messages. Moreover, the message passing is permutation equivariant as it finally applies a permutation invariant function to aggregate neighbor's information in a node-wise manner.

To construct a graph model, one can apply the message passing layer multiple times. The more layers are utilized, the farther the information the nodes receive. Indicated in figure 3.16, by applying two message passing layers, the node 1 can receive messages from the node 4, which is not one of its neighbor: after applying k layers, the node gets information from k hops away.

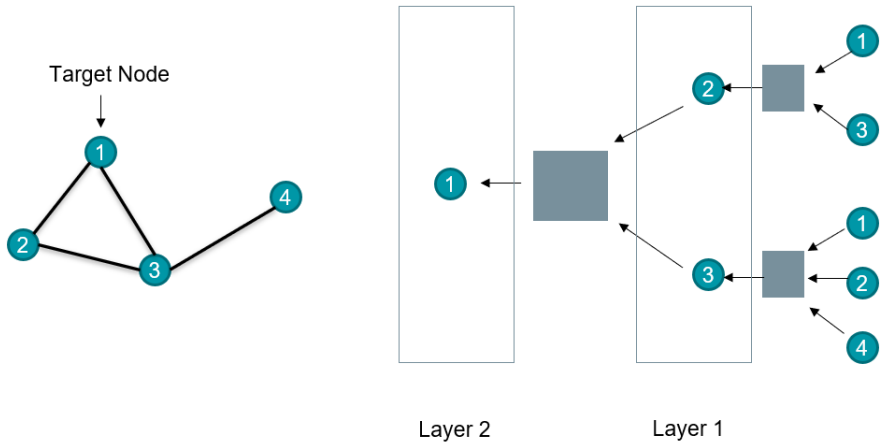


Figure 3.16: A message passing schema with two graph layers, the node 1 will finally receive information from the node 4 after a two-layer GNN.

Almost all recently proposed GNN algorithms can be expressed using a message-passing schema. They all share the idea of information propagation. In the following sections, we will provide an overview of several GNNs that have been proposed within the context of this message-passing framework.

Graph Convolutional Network [85] is a spectral-based GNN, in which the convolution operator is defined in the Fourier domain. Consider a graph represented by its adjacency matrix A; the graph convolutional network (GCN) first normalizes the adjacency matrix with self-loop $\hat{A} = A + I$ such that all rows sum to one. The normalization is achieved by simple matrix multiplication involving the diagonal normalizing matrix D : $D^{-\frac{1}{2}}\hat{A}D^{-\frac{1}{2}}$. GCN defines the convolution operator as:

$$X' = D^{-\frac{1}{2}}\hat{A}D^{-\frac{1}{2}}X\Theta \tag{3.28}$$

where X denotes the matrix of the feature attributes, and Θ represents the trainable parameters of the graph layer. Even though GCN is a spectral-based

GNN, it can still be written in the message-passing framework by taking its node-wise formulation:

$$x_i^{t+1} = \Theta^T \sum_{j \in \mathcal{N}(i) \cup i} \frac{x_j^t}{\sqrt{d_i d_j}}, \tag{3.29}$$

with d_i denoting the number of neighbours on node i by counting the self-loop. GCN is one of the simplest graph neural networks. However, the architecture cannot take into consideration edge attributes.

MoNet [88] assigns different aggregation weights to a node neighbours depending on the edge features. Weight functions are trained to map the edge features to the relative weight between two connected nodes. Consider a weighted graph $G = (V, E, \mathcal{V}, \mathcal{E})$, the basic idea of MoNet is to define a trainable function $\mathbf{w}_k$ that computes an edge weight w_{ij} from the edge features $\mathbf{e}_{ij}$. MoNet then defines the convolutional operator on node i as:

$$\mathbf{x}_i^{t+1} = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \frac{1}{K} \sum_{k=1}^{K} \mathbf{w}_k(\mathbf{x}_{ij}^e) \odot \mathbf{\Theta}_k \mathbf{x}_j^t \tag{3.30}$$

where $\mathbf{K}$ is the user-defined kernel size, $\mathbf{\Theta_k} \in \mathbb{R}^{M \times N}$ stands for the trainable matrix applying a linear transformation on the input data, $\odot$ is the element-wise product, and $w_k, k = 1, \ldots, K$ are trainable edge weights. Following [88], we choose Gaussian kernels defined as:

$$\mathbf{w}_k(\mathbf{x}_{ij}^e) = \exp(-\frac{1}{2}(\mathbf{x}_{ij}^e - \mu_k)^T \Sigma_k^{-1}(\mathbf{x}_{ij}^e - \mu_k)) \tag{3.31}$$

Both μ_k and Σ_k are trainable parameters representing the mean vector and covariance matrix of the Gaussian kernel. MoNet provides a framework to learn graphs with multi-dimensional edge features, that is, using a trainable function $\mathbf{w}$ to map the edge features to an edge weight. Some recent graph layers are based on the work of the MoNet. For example, [92] chooses the kernel function defined over the weighted B-Spline tensor product basis.

3.4.4 Graph pooling operators

CNN-based models utilize pooling operators to reduce the image size and generate smaller feature maps to extract hierarchical spatial features. The down-sampling strategy is also needed for graph models to coarsen the graph. However, defining a pooling operator is not straightforward for graph data. The pooling operator is not natural for graph structures, as it breaks the connections between nodes. Normally, to coarsen a graph, the preserved nodes should be carefully chosen after the pooling operator, then new edges between these nodes should be added if needed. Some earlier works [93, 94] proposed the idea of using eigen-decomposition to down-sample the graphs. More recent works, however, tend to apply neural networks to automatically learn how to coarsen graphs rather than using time-consuming eigen decomposition methods.

Top-K pooling [95, 96] uses a trainable projection vector to transform node features to the corresponding scores. The top k nodes with the highest scores are chosen as the remaining nodes of the new graph, but no additional connections are added between these nodes.

Diff pooling [97] is a differential pooling layer. The Diff pooling can generate hierarchical representations of graphs in an end-to-end fashion by learning a differentiable cluster assignment for nodes at each graph layer. The cluster assignment is learned by the following equation:

$$S^l = \text{softmax}(GNN_l(A^l, X^l)), \quad A^{l+1} = (S^l)^T A^l S^l \tag{3.32}$$

where X^l denotes the node features, and A^l represents the adjacency matrix. S^l is the learned assignment matrix. If edge features are scalar, they are expressed in adjacency matrix; otherwise, edge features are not considered.

Self-Attention graph pooling [98] uses a GNN to provide self-attention scores. Similar to the Top-k pooling, after getting the score of each node, the top k nodes with the highest scores are preserved to construct the new graph.

These coarsen operators take advantage of nodal features to select a subset of nodes from the graph. While [99] proposed an edge pooling operator, where a

score is attributed to each edge by a trainable function, and edges are contracted iteratively according to that score.

In Chapter 4, we will propose two hierarchical graph architectures based on a set of meshes with different scales. Different from the situation above, coarse meshes are generated by well-studied mesh generation algorithms. This will allow us to define pooling operators with k-nearest neighbors to transform the features from one mesh to the next, upward or downward.

3.5 Transfer Learning

Although deep learning methods have achieved big success in many domains, there are still some limitations when it comes to certain real-world applications. Collecting sufficient data to support the training process of deep learning models is often expensive or even infeasible. According to the statistical learning theory, large and diverse data is needed to reduce the estimation error so as to achieve broad generalization on unknown examples drawn from the same distribution as those in the training set. For example, the most famous database on computer vision is ImageNet which contains more than 14 million images covering more than 20 000 categories. However, in most cases, only a few data can be accessed, such as medical images or sport images, that can be viewed as different tasks, with images drawn from different distributions. Learning to solve all those different tasks from scratch with possibly insufficient data is impractical.

Transfer Learning algorithms [100] is one way to handle such issues, where the knowledge gained while solving a task is re-used for a different but related task. Transfer Learning is widely used nowadays to solve text- and image-related tasks [101–103], especially in similar situations when only a few data samples are available or training. We will discuss in more detail the underlying ideas in transfer learning and the formal definition in this section. We refer the interested reader to the review papers [100, 104, 105], that provide a comprehensive understanding of the transfer learning domain.

3.5.1 Definitions

This Section is a quick introduction to Transfer Learning, following [100, 106].

A *domain* $D = \{\mathcal{X}, P\}$ consists of a feature space $\mathcal{X}$ and a distribution P over $\mathcal{X}$. A *learning task* $\mathcal{T}$ is defined by a *label space* $\mathcal{Y}$, and a model $f(x)$ that predicts a label y to all points in $\mathcal{X}$. The goal of the supervised learning task, as introduced in Section 3.1.1, is to identify the model $f(x)$ from an i.i.d. sample $X = \{x_i \in \mathcal{X}, i = 1, \ldots, n\}$ drawn following P, and the corresponding labels $y_i = f(x_i)$. Many learning algorithms learn, in fact, a conditional distribution $P(Y|X)$, such that the output of the model for a given $x \in \mathcal{X}$ is the probability distribution $P(y|x)$ for all y in $\mathcal{Y}$. More generally, the resolution of a supervised learning task

uses a so-called *loss function* $\mathcal{L}$ (See Section 3.1.1) whose minimization leads to an approximation of the model f, or of the conditional probability $P(.|x)$.

Standard supervised learning problems (Section 3.1.1) are concerned with solving one single learning task defined on one single domain: from a sample dataset drawn from one single distribution in which examples are independent of each other (i.i.d examples), the goal is to find a model that generalizes to any sample drawn from the same distribution. Transfer learning considers several *source tasks*, possibly defined on several domains, and for which the learning is supposed to be relatively easy (for instance, because of the availability of a large representative labeled sample set), and different *target tasks*, possibly defined on different domains, that would be hard to solve stand-alone (for instance, because only very few, if any, labeled examples are available), but for which the knowledge acquired on the source tasks can help (see Figure 3.17). More formally:

Transfer learning: Given m^S source domains and tasks $\{(D_i^S, \mathcal{T}_i^S), i = 1, \ldots, m^S\}$, and m^T target domains and tasks $\{(D_j^T, \mathcal{T}_j^T), j = 1, \ldots, m^T\}$, the basic idea of transfer learning is to use the knowledge from the learning processes run on the source tasks (usually the corresponding learned models) to improve the learning process on the target tasks compared to what would be learned using only the data of each target task. The sought improvement can be either in terms of the accuracy of the learned model or in terms of the computational cost of the learning itself – or both. Figure 3.17 is a schematic representation of a transfer learning setting with $m^S = 2$ and $m_T = 1$.

3.5.2 Transfer Learning Approaches

According to [100, 106], transfer learning approaches can be categorized into four cases, instance-based, feature-based, parameter-based, and relational-based.

Instance-based approaches [107–110] assume that by re-weighting, some data in the source domain can be reused for learning in the target domain. These approaches propose certain algorithms to re-weight and sample data from the source domain.

Feature-based approaches [111–113] learn a good feature representation for the

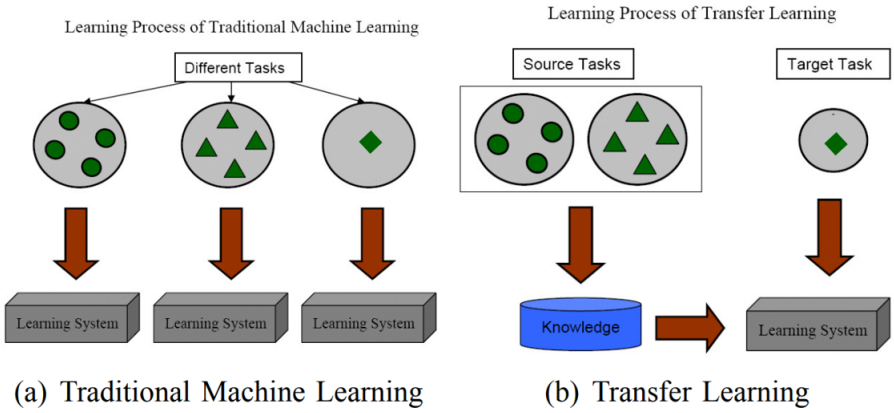


Figure 3.17: Difference between transfer learning and traditional machine learning.

target domain with the help of data in the source domain. The representative features are considered as the transferred knowledge. It involves minimizing the marginal and conditional distribution differences, preserving the properties or potential structures of the data, and finding the correspondences between features.

Parameter-based approaches assume that the source model and the target model share some parameters or prior-distribution of hyper-parameters. The parameter-based approaches are widely used in neural networks. A more detailed discussion will follow for the parameter-based topic.

Relational-based approaches [114–116] focus primarily on problems in relational domains. The logical relationships or rules learned in the source domain are transferred to the target domain.

3.5.3 Parameter-based Approaches

Parameter-based approaches are commonly used in transfer learning, especially with deep learning models. One benefit of deep learning is that when treating complex tasks such as image recognition, deep learning models tend to automatically learn feature representation at different levels [117]. Hand-crafted features used for traditional machine learning methods are no longer needed. As much as the source and the target tasks are related, the learned feature representation from the source domain can also help to solve the target task. Usually, parameter-based

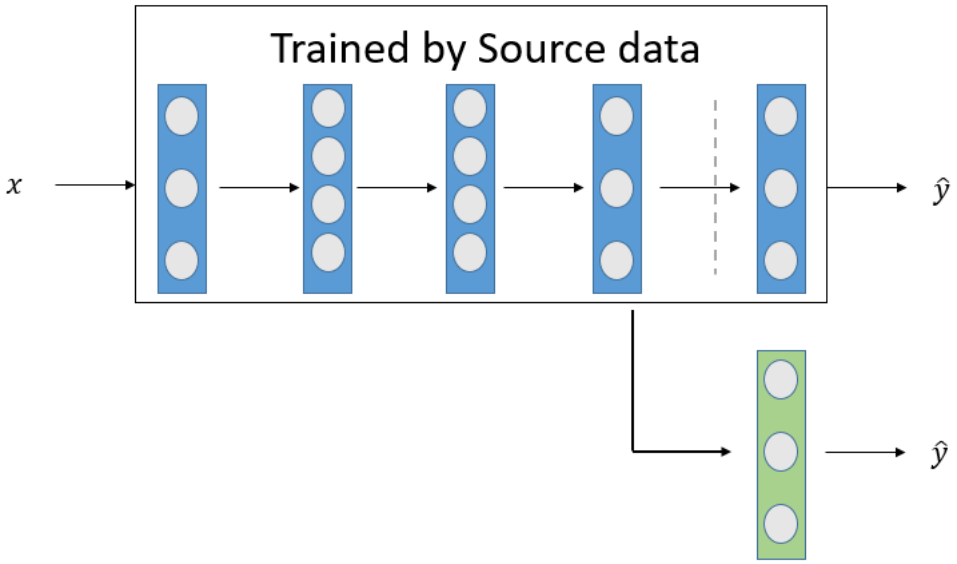


Figure 3.18: Parameter-based approach on deep learning model

approaches start by training a model using data from the source domain. To solve the target task, instead of training a new model from scratch, parameters of the pre-trained model are used as an initialization. Typically, for multi-layer neural models, the final layer from the pre-trained model is replaced by a randomly initialized layer to match the output dimension for the target task, as shown in Figure 3.18. Data from the target domain is then fed to retrain the model.

There are mainly two ways to retrain the model. The first one freezes the weights of the layers from the pre-trained model, considers these layers as a feature extractor, and only trains the last few layers: [103] is an example; it preserves the front layers of the model trained on the ImageNet dataset to compute latent representation features for images from other domains.

Another approach is to retrain the entire model but using a small learning rate. Incremental adaptation of the pre-trained features to the new data (also called fine-tuning) can potentially lead to meaningful improvements (see e.g., [101, 102] for NLP applications).

In chapter 6, we will propose an original transfer learning approach for the specific case of PDE approximation, where the source domain will be the data-

based simulation of the PDE on a coarse mesh, and the target domain will be the numerical approximation of the same PDE using a much finer mesh.

3.6 Meta Learning

Standard supervised learning problems introduced in Section 3.1.1 are concerned with solving one single task. Transfer learning, as surveyed in the previous Section 3.5, aims at using knowledge acquired after solving some source learning tasks better to solve some target tasks, a priori known.

As noted in previous Sections, standard supervised machine learning algorithms have difficulties in solving problems with only a few data accessible. Transfer Learning is one way to handle such issues when at least a few tasks can be easily learned: the transfer of knowledge makes it easier to solve new related tasks. However, when no source task is sufficiently easy to learn on its own, or when the goal is to solve any target task from a given distribution of tasks and not a few a priori known target tasks, Transfer Learning might not be efficient enough, or not even applicable, and this is where Meta-Learning can be a solution.

In this section, we introduce the concept of *Meta-Learning*, a technique to transfer learning methodology, and in particular, to solve problems with insufficient data. The basic idea is to learn an incomplete model that is selected (and optimized) for its ability to learn efficiently (quickly and accurately) several different tasks, known as *training tasks*. According to [118], given a set of tasks $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N\}$ with different data generation distributions and decision functions, meta-learning algorithms are designed to learn a general purpose but incomplete model that, in turn, will be able to rapidly learn specialized models for some new tasks (e.g., with very few samples).

One baseline naive approach for solving such a meta-learning problem involving multiple tasks would be to aggregate all data across the different tasks so as to reduce the problem to a single-task problem with a distribution equal to the mixture of the different initial distributions. Every task then shares the same model as single-task learning. But first, such an approach ignores the fact that data comes from different distributions, something that can easily hinder the search for a good model. And second, this assumes that all the tasks you want to solve are known beforehand, and this is not always the case in real applications. Indeed, if a new unknown task is to be solved, the model should be completely re-trained after augmenting the training set with the examples from the new

tasks. Another choice could be to train a different model for each task and to use Transfer Learning for new tasks. Unfortunately, this is not ideal, especially when there is only a small amount of data per task. The empirical risk is no longer a reliable approximation of the actual risk due to the lack of training examples.

Meta-learning provides a way to gain experience (aka *meta-knowledge*) over various tasks, and to use this experience to improve learning a new task using task-specific data. In the perspective of statistical learning, rather than seeking a function for a single task from scratch, the meta-knowledge, in a way, decrease the complexity of the hypothesis space so as to reduce the requirement of learning examples to obtain a good estimation error.

In this Section, we will briefly introduce some commonly used meta-learning algorithms. Readers interested in more details on meta-learning are referred to [119], and the paper [118].

3.6.1 Problem Set-Up

A Meta-learning dataset consists of a set of training tasks $\mathcal{T}_i, i = 1, \ldots, N$. For each task $\mathcal{T}_i$, a dataset $\mathcal{D}_\rangle$ of pairs (features, label) is available, sampled from a distribution $p(\mathcal{T}_i)$, with $\mathcal{D}_i = \{(x_1^i, y_1^i), \ldots, (x_k^i, y_k^i)\}$, as well as a loss function $\mathcal{L}_i$. In addition, each dataset $\mathcal{D}_i$ is split into a training and a test set: $\mathcal{D}_i = \{\mathcal{D}_i^{tr}, \mathcal{D}_i^{test}\}$. Formally, the meta-training set is: $\mathcal{D}_{meta}^{tr} = \{(\mathcal{D}_i^{tr}, \mathcal{D}_i^{test})\}_i^N$. Figure 3.19 shows an example of the meta-learning dataset in the image recognition context.

Meta-training: some *meta-knowledge* w is learned from $\mathcal{D}_{meta}^{tr}$. w differs for the different learning algorithms and will be discussed in detail in the next section. Task-specific parameters θ_i are then computed based on $\mathcal{D}_i^{tr}$ and the meta-knowledge w:

$$\theta_i = \arg\min_\theta \mathcal{L}_i(\mathcal{D}_i^{tr}, \theta, w) \tag{3.33}$$

During meta-training, meta-learning algorithms minimize the performance of all task-specific parameters θ_i on their respective test set $\mathcal{D}_i^{test}$, after some (partial) training for task $\mathcal{T}_i$:

$$\min_w \sum_i^N \mathcal{L}_i(D_i^{test}, \theta_i, w) \tag{3.34}$$

$$s.t. \quad \theta_i = \arg\min_\theta \mathcal{L}_i(\mathcal{D}_i^{tr}, \theta, w) \tag{3.35}$$

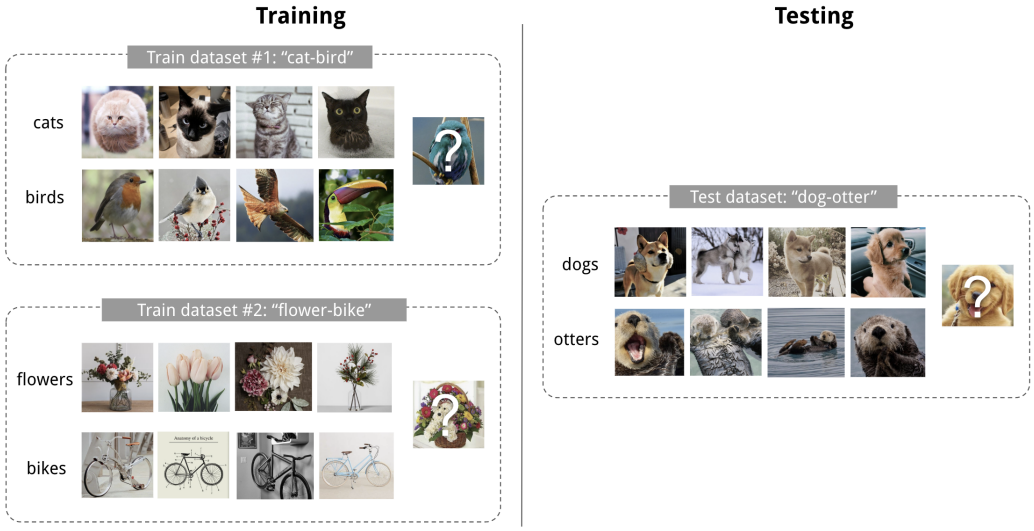


Figure 3.19: An Example of meta-learning dataset (Source from Pinterest)

Meta-testing: given a new meta-test task $\mathcal{T} = \{(\mathcal{D}^{tr}, \mathcal{D}^{test}), \mathcal{L}\}$ sampled from the same meta-distribution as the training tasks $\mathcal{T}_1, \ldots, \mathcal{T}_N$, it should be possible to learn the task-specific parameters θ for the new task $\mathcal{T}$ using $\mathcal{D}^{tr}$ and the meta-knowledge w. The performance of the whole meta-learning algorithm is then evaluated on the test set $\mathcal{D}^{test}$ (and possibly for several different new meta-test tasks $\mathcal{T}$).

3.6.2 Meta-learning Algorithms

There are mainly three categories of meta-learning algorithms based on how adaptation is performed in the training process: Model-based, metric-based, and optimization-based approaches.

Model-based approaches [120–122] train a neural network to directly predict the task-specific parameters θ_i given the corresponding training set D_i^{tr}:

$$\theta_i = f_w(D_i^{tr})$$

In this case, the meta-knowledge w is an entire model.
The model-based algorithms can be trained directly with standard supervised

learning. The target is to train a neural network f_w such that the output parameters θ represent an accurate predictor on every test set for each task:

$$w = \arg\min_{w} \sum_{i}^{N} \mathcal{L}_i(D_i^{test}, f_w(D_i^{tr})) \tag{3.36}$$

These approaches are very expressive and easy to combine with different learning algorithms. But they require a complex model to take the entire dataset as input, which challenges the optimization problem. Moreover, to train a complex model, they need sufficiently many training tasks, and this is rarely the case.

Metric-based approaches [123–126] usually employ non-parametric methods (e.g., k-nearest neighbors) to compute the prediction for each task. These methods are simple and efficient and work very well when tasks contain only a few data. The prediction over a task is approximated by simply comparing the input features with training samples and predicting the label of the matching training samples in classification problems or the weighted average in regression problems:

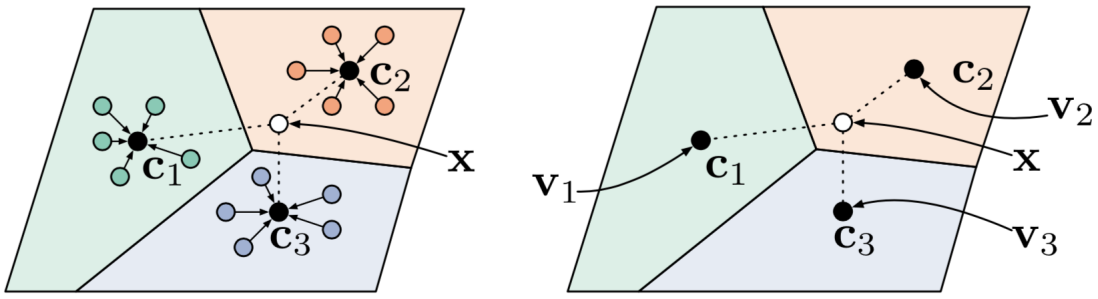
$$\hat{y} = \sum_{(x_i, y_i) \in \mathcal{D}_j^{tr}} k_w(x, x_i) y_i,$$

where k_w is a kernel function to compute the similarity score between two input features. Metric-based learning algorithms aim to learn a good similarity kernel k_w over $\mathcal{D}^{tr}$. The metric-based approaches are easy to optimize but they don't work well on larger tasks.

Optimization-based approaches [127–129] are based on the idea that the adaptation on a specific task is treated as an optimization problem. In other words, the task-specific parameters θ are computed through optimization, and the meta-parameters w serve as a prior, i.e., as an initialization of the optimization process. In the next section, we will discuss in more detail the well-known optimization-based approach MAML (Model-Agnostic Meta-learning).

3.6.3 Model-Agnostic Meta-learning

Model-Agnostic Meta-learning (MAML) [128] is an optimization-based meta-learning algorithm that explicitly optimizes model parameters for fast learning.



(a) Few-shot (b) Zero-shot

Figure 3.20: Architecture of a metric-based approach: prototypical-network, the network embeds a function f_w to encode each input to an M-dimensional space. The similarity between inputs is calculated by a pre-defined distance function

As mentioned, the meta-parameters w are trained to serve as a prior for the subsequent meta-test learning tasks. In fact, one successful usage of prior knowledge is from fine-tuning introduced in Section 3.5. In fine-tuning, the pre-trained model f_w from a similar task is considered as initialization, and used to train a new model using a few steps of gradient descent:

$$\theta = w - \eta \nabla_w \mathcal{L}(D^{tr}, w)$$

Indeed, fine-tuning works very well on small-scale datasets. Rather than learning from scratch, using an initialization from the pre-trained model can accelerate the training time and also avoid overfitting. This is one way to solve meta-learning problems. Meta-training dataset is used to get a pre-trained model, and new tasks are solved by simply fine-tuning the pre-trained model during meta-test time. Unfortunately, for cases where only small samples are available in test tasks, fine-tuning is not very effective.

Inspired by such fine-tuning approaches, MAML aims to find a partial model with parameters w, which can be easily fine-tuned for all tasks, even with small amounts of data. At meta training time, for each training task $\mathcal{T}_i$, the fine-tuning process is used to obtain the task-specific parameters θ_i and evaluate the task-specific model f_{θ_i} by samples from the test set D_i^{test}. The optimization target in this case is:

$$\min_w \sum_{\mathcal{T}_i} \mathcal{L}_i(D_i^{test}, \theta_i) \tag{3.37}$$

θ_i is obtained by fine-tuning w on D_i^{tr}: If only one step of gradient descent is applied, θ_i is expressed as:

$$\theta_i = w - \alpha \nabla_w L_i(D_i^{tr}, w) \tag{3.38}$$

By replacing θ_i with the formula above, the optimization problem becomes:

$$\min_w \sum_{\mathcal{T}_i} \mathcal{L}_i(D_i^{test}, w - \alpha \nabla_w L_i(D_i^{tr}, w)) \tag{3.39}$$

In the more general case where that there are N update steps of gradient, the meta loss function is then written as:

$$\min_w \sum_{\mathcal{T}_i} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i^N}) = \min_w \sum_{\mathcal{T}_i} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i^{N-1} - \alpha \nabla_\theta L_{\mathcal{T}_i}(f(\theta_i^{N-1}))}) \tag{3.40}$$

In other words, MAML learns a set of parameters w in such a way that it is easily fine-tuned for each known learning task. It is then expected that it can also be easily adapted to all test tasks with a few steps of fine-tuning. MAML is described in Algorithm 11.

Algorithm 11 Model-Agnostic Meta-learning Algorithm

Require: Sample a set of tasks $\{\mathcal{T}_1, ..., \mathcal{T}_n\}$ from the task distribution.
1: Set learning rate parameters η_1, η_2
2: Initialize the meta parameters θ
3: **for** every training task $\mathcal{T}_i$ **do**
4: Sample disjoint dataset (D_i^{tr}, D_i^{test}) from $\mathcal{T}_i$
5: Evaluating gradient descent on θ with D_i^{tr}
6: Compute adapted parameters: $\theta_i = w - \eta_1 \nabla_w \mathcal{L}(D_i^{tr}, w)$
7: **end for**
8: Update the meta-parameters $w = w - \eta_2 \nabla_w \sum_{\mathcal{T}_i} \mathcal{L}_i(\theta_i, D_i^{test})$

In fact, MAML tends to extrapolate better than metric-based and model-based approaches, as the procedure of adaptation at test-time corresponds to an actual optimization method. However, the update of the meta-parameters θ requires second-order derivatives. As a result, MAML has high computation cost and is memory-intensive. The bi-level optimization problem also introduces instabilities when training the meta-learner. Some works try to mitigate these instabilities: [130, 131] automatically learn inner learning rate η_1, [132, 133] optimize only a subset of parameters in inner-loop. The state-of-the-art of such works is, to the best of our knowledge, MAML++ [134], stabilizing the meta-learner by redesigning the loss function

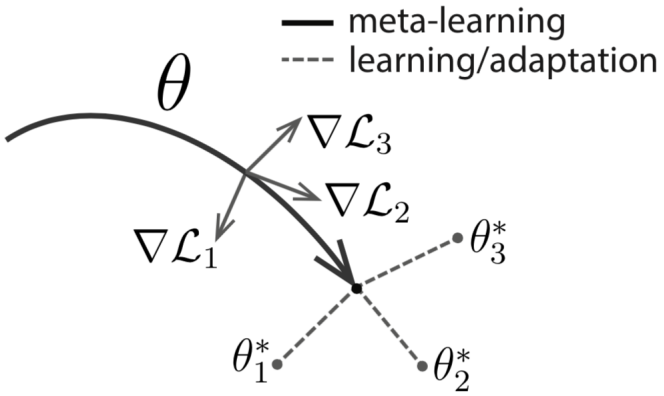


Figure 3.21: Diagram of MAML

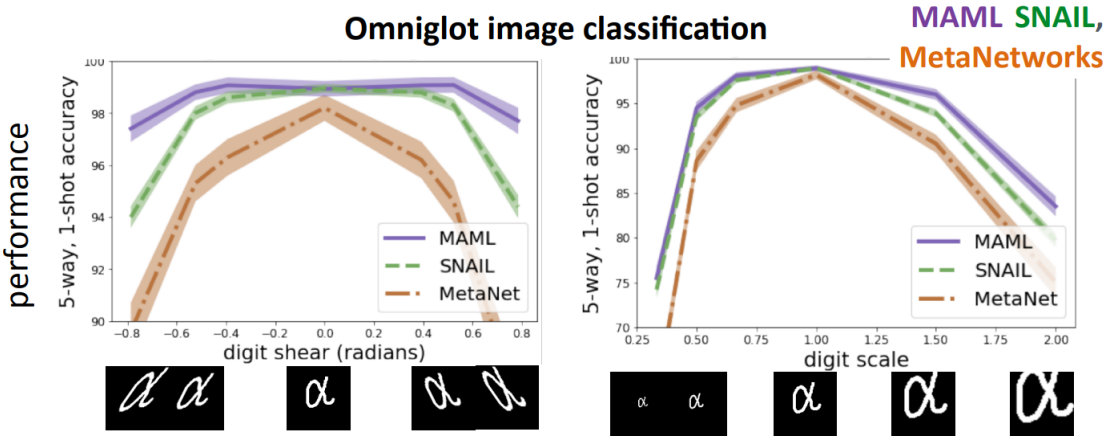


Figure 3.22: Comparison of model performance on extrapolation tasks between MAML and the two model-based methods SNAIL and MetaNet. These meta-learners are trained by the Omniglot dataset, and tested on out-of-distributions tasks where images are simply sheared or scaled. Source from [135]

3.6.4 MAML++

According to [134], MAML can become very unstable when training with large update steps. For instance, Figure 3.23 displays the instability of MAML during the meta-training on the Omniglot dataset. In order to mitigate these instabilities, [134] proposed a multi-step loss optimization method (MSL). Instead of minimizing only the loss from the last step, the idea is to minimize the weighted sum of the

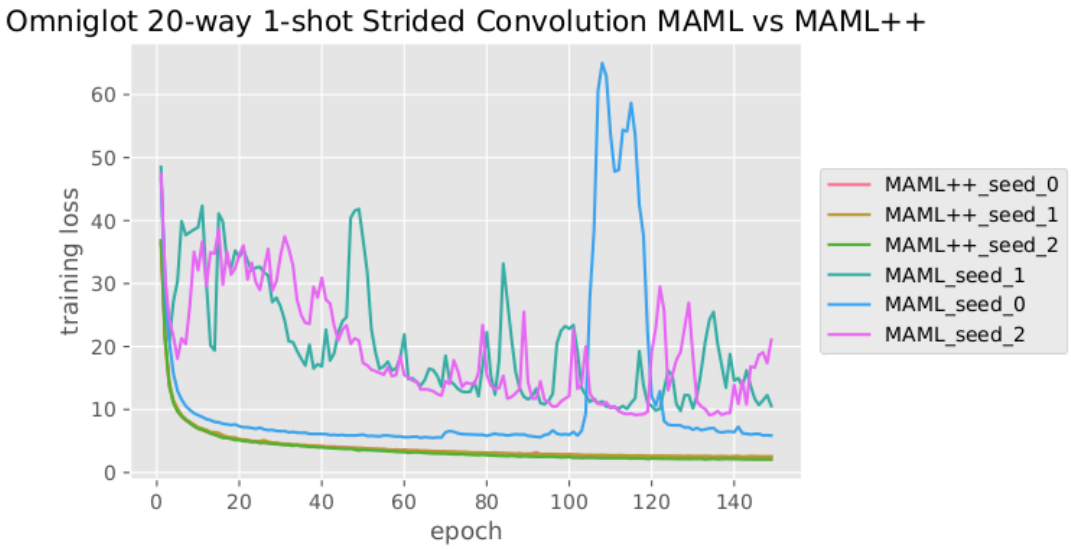


Figure 3.23: Instability of MAML and stability of MAML++ on Omniglot dataset, from [134]

losses at every updating step j, where the weights w_j are pre-defined by users.

$$\mathcal{L}_{meta} = \sum_{\mathcal{T}_i} \sum_{j=1}^{N} w_j \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i^j}) \tag{3.41}$$

Figure 3.23 shows three training curves of MAML and three (almost undistinguisahble) training curves for the improved MAML++ on the Omniglot dataset: MAML++ is clearly much more stable and displays a better convergence (in both accuracy and speed) than the original MAML.

In chapter 5, we will propose a meta-learning approach for airfoil simulation problems to enhance the model's performance on out-of-distribution data.

Part II

Contributions

4

Graph Neural Networks for PDEs

Contents

In Chapter 1, we discussed some research on applying CNNs to solve PDEs thanks to their tremendous successes in image analysis. Typically, data generated on unstructured meshes is first projected on structured grids, then trained by a CNN-based model. However, in the real world, physical problems have

complex geometric domains, and such a CNN approach can lead to a significant interpolation error, especially on the boundary of an actual domain. We will discuss this in more detail later in this chapter.

Graph Neural Networks (GNNs) introduced in Section 3.4 are ML methods that handle data living on graphs. GNNs aim to reproduce the locality properties of CNNs on graph data. As the mesh data can be considered as a special graph structure, GNNs allow us to construct a deep learning model that can be applied directly to mesh data instead of projection into structured grids. In recent years, many attempts have been made to construct a GNN model to study mesh data: [30] discussed fluid flow field problems on different irregular geometries. It considers CFD data as a set of points (called point clouds) and applies the PointNet[31] architecture specially designed for such a data type. [32] combines graph neural networks with a traditional CFD solver (run on a coarse mesh) to accelerate fluid flow prediction on a much finer mesh; [136] proposed a framework for learning mesh-based simulation problems with graph neural networks. Unlike these methods that apply GNNs directly on a fine mesh, we propose a hierarchical structure to extract both global and local features from mesh data. We propose generic up- and down-sampling procedures for GNNs, taking advantage of meshes of different granularities. From thereon, inspired by the multi-grid methods in the numerical field [13], we introduce two such architectures for GNNs in the context of PDE simulations: the Graph U-Net, based on the famous U-Net [29] proposed for image segmentation, and the novel Multi-Grid Multi-Input model.

This Chapter is organized as follows: In Section 4.1, we introduce the multi-grid architectures Graph U-Net and Graph MGMI, based on a hierarchy of meshes on the domain of the PDE. In Section 4.2, we describe the experimental use cases used to validate our proposed models: non-linear Poisson equations and Navier-Stokes equations for airfoil simulation. In Section 4.3 and 4.4, we present the results of our experiments on non-linear Poisson's equations and Navier-Stokes equations for airfoil simulation, respectively, which validate the effectiveness of the proposed approach.

4.1 Multi-Grid GNNs

Given a partial differential equation (PDE) defined on a domain Ω, the goal of this work is to use neural networks as a replacement for numerical solvers to simulate the PDE. To do this, we first generate sample data on the domain using numerical solvers. This dataset is then used to train a neural network to predict solutions for new samples of the PDE. However, when simulating PDEs, complex geometries and unstructured meshes are often inevitable. The finite element method (FEM) or finite volume method (FVM) mentioned in Chapter 2.1 are commonly used to discretize PDEs by constructing a mesh on the domain Ω. And except in specific cases (Ω is similar to a quadrangle), this mesh will be irregular, i.e., not topologically equivalent to a grid. As a result, the data generated from numerical simulations is no longer structured as a regular grid image. In order to effectively handle this type of data with Deep Neural Networks, we will use Graph Neural Networks, as already mentioned. However, whereas multi-grid approaches can be straightforwardly ported to CNNs, their implementation in GNNs is still missing.

In this section, we propose hierarchical graph-based approaches for learning approximate numerical solutions of partial differential equations (PDEs) on unstructured meshes. Our approach addresses the up- and down-sampling issues of graph neural networks (GNNs) by constructing a hierarchy of meshes with increasing complexity. This allows us to learn both global and local features of the solution on the mesh data.

4.1.1 GNNs for Mesh data

When it comes to unstructured mesh data, a naive solution is to embed the complex domain into a regular rectangle domain and to proceed by interpolation to project data on the unstructured mesh onto the grid. From there on, CNNs can be applied straightforwardly to treat the resulting Euclidean data. However, using pixel representation has some shortcomings [137]:

- Due to the data projection between mesh and grid, pixelization decreases the accuracy of mesh data, especially for physical problems posed on complex geometric domains. Such a CNN-based approach can lead to a significant interpolation error, in particular on the boundary of the actual domain.

- Pixels often require higher resolution to have a similar expressive power than a mesh when the geometry is complex. The donut domain, which will be discussed in Section 4.3 is an example. Grids are not good at describing complex boundaries.
- Moreover, it is common to mask the interior area in the pixel-wise approach. Some computational effort of the CNN is wasted on the unused area.

The use of GNNs becomes crucial when it comes to unstructured meshes, as GNNs can operate directly on data from non-Euclidean space. As a matter of fact, a mesh is naturally a particular graph architecture. Formally, any 2D mesh structure[1] can be expressed as $M = (V, E, \mathcal{E})$. The set $v_i \in V$ denotes the nodes of the mesh, $e_{ij} = (v_i, v_j) \in E$ if both v_i and v_j belong to one element of the mesh. Moreover, the attached attributes $\mathcal{E} \in \mathbb{R}^{m\times 2}$ denote the pseudo-coordinates for every edge: The attribute $x_{ij}^e \in \mathbb{R}^2$ on edge e_{ij} is defined as the difference of the coordinates of the two nodes v_i and v_j.

$$x_{ij}^e = p_j - p_i \tag{4.1}$$

where p_i and p_j are the coordinates of node v_i and node v_j respectively.
Any type of data can be defined on the mesh, as some node attribute in $\mathcal{V}$.

MoNet GNN Some basic concepts of GNNs are outlined in Section 3.4. Most of them are based on the idea of message-passing, leveraging features from the node's neighbors to create new information. In this work, we use the mixture model network MoNet as GNN structure introduced in Section 3.4.3.

4.1.2 Multi-Resolution Approaches

As discussed in Section 2.3, many multi-grid algorithms have been proposed in the context of numerical simulation. The main idea is to compute approximate solutions to the problem at hand on meshes of different resolutions. The steps on coarse meshes are fast and help to unveil the global features of the solution, while fine meshes refine the solutions, removing unwanted spatial oscillations but at a

[1]For the sake of simplicity, we will only consider in this work P1 finite elements in 2D, for which the dof are the values at the nodes. The same work can be done for other types of 2D or 3D meshes, with more complex notations, but will not be discussed here.

higher computational cost. As a matter of fact, the idea of multi-grid has already been used in the neural network framework in the context of CNNs for image analysis and is based on pooling (down-sampling) and up-sampling layers that merge or expand rectangle patches of the image. Among well-known examples are Autoencoders, and the famed U-Net architecture (Section 3.3.3), which adds to the reduction/reconstruction structure some "horizontal" connection between downstream and upstream layers of the same dimension.

The multi-grid concept can be extended to the GNN framework easily. Instead of using down-sampling (pooling) operators to automatically coarsen the mesh, we create, over the domain at hand, a hierarchy of meshes of increasing complexity. However, different from the CNN context, no obvious down-sampling (pooling) operators exist when it comes to GNNs. The pooling operators discussed in Section 3.4.4 focus on the selection of nodes to construct a coarse graph – but this is unnecessary for mesh data. By simply generating a set of meshes with different coarsenesses, we can easily define operators that transform the features from one mesh to the next, up- or downward.

Mesh Hierarchy There are many options to create meshes to subdivide the problem domain. In fact, a considerable amount of effort is undertaken to design mesh structures to achieve high-order accuracy in numerical analysis. Thanks to the full-developed studies on mesh construction, we can easily create hierarchies of meshes without using graph pooling operators discussed in Section 3.4.4.
Delaunay triangulation [138] is a commonly used algorithm to create a mesh on a given domain. It attempts to maximize the minimum of all the angles of the triangles in the triangulation to avoid sliver triangles. New nodes are gradually inserted into the triangulation and connected with their neighbors under several rules. Delaunay triangulation divides the whole domain uniformly, avoids narrow triangles, and retains geometric properties with limited nodes. The software CGAL [139] uses Delaunay triangulation to ensure good mesh quality, and allows for the efficient computation of sets of meshes at different scales using user-defined criteria.
Nevertheless, several approaches have been proposed to mesh coarsening. For example, the *incremental decimation* method aims to reduce the number of points by preserving the specific properties of the original mesh as much as possible.

It sequentially removes one vertex or edge with the smallest changes until the given criteria are satisfied. Such methods are quite expensive compared with Delaunay triangulation which directly computes the coarse mesh. As a result, in this work we utilize Delaunay triangulation to generate a set of independent meshes at different scales in order to reduce generation time.

Sampling Operator We propose a sampling operator to convert data between two meshes $\mathcal{M}_1$ and $\mathcal{M}_2$ that simply uses the k-nearest interpolation proposed in PointNet++ [89]. Let z be a node from $\mathcal{M}_1$, and assume its k nearest neighbors on $\mathcal{M}_2$ are $(x_1, \ldots, x_k)$. Let f represent some node feature. The interpolated feature $f(z)$ is defined from those of the x_i as:

$$\mathbf{f}(\mathbf{z}) = \frac{\sum_{i=1}^{k} w(x_i)\mathbf{f}(x_i)}{\sum_{i=1}^{k} w(x_i)}, \text{ where } w(x_i) = \frac{1}{||z - x_i||_2} \tag{4.2}$$

Based on these operators, both up- and down-sampling operators can be defined, and it is then straightforward to define multi-resolution architectures in the context of PDE simulation.

The U-Net Achitecture

First, we propose a simple adaptation of the U-Net architecture [29], where each block is a MoNet block, followed by one sampling operator as described above. The "horizontal" connections that characterize U-Net are added as well. During encoding, the model starts capturing local features from small neighborhoods. Pooling layers will down-sample the data from finer to coarser mesh, where neurons have spatially larger receptive fields. We repeat the above steps until we reach the coarsest mesh where features have a good representation of the whole domain.

As noted, the U-Net architecture is quite similar to the V-cycle scheme on Multi-grid algorithms (See Section 3.3.3), which starts from the finest mesh and samples down to the coarsest mesh, then works its way back into the finest mesh.

The MGMI Architecture

We now propose a completely original architecture called Multi-Grid Multi-Input (MGMI), as displayed in Figure 4.1. Here, we will illustrate it on the example from the non-linear Poisson's equation that will be introduced in Section 4.2.1.

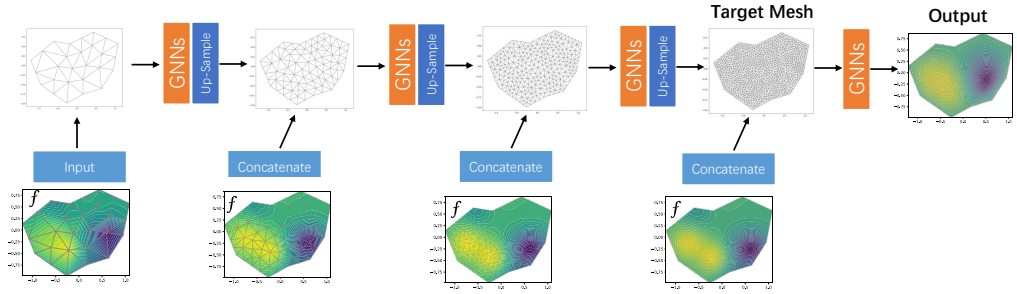


Figure 4.1: The MGMI architecture: Coarse-to-fine meshes are linked with up-sampling operators, and the right-hand side is input to the NN at all different resolutions.

The external force function f is considered as an input, and the function u is what we want to predict. Only upsampling operators are used: the model starts from the coarsest mesh, and the input is the projection of f on this mesh. Our approach is inspired by the F-cycle scheme from multi-grid algorithms, which starts at the coarsest mesh and progresses to finer meshes until the target mesh is reached. After a GNN block, an upsampling operator adapts the output to the next mesh, and a projection of f on the current mesh concatenated with the previous output is again fed to the next GNN block. The process repeats until reaching the finest mesh (4 different levels will be used throughout this work). This should allow the features of different granularities to be discovered gradually, from global to local features.

This architecture takes advantage of the hierarchy of meshes from the input perspective, feeding the different dimensions with ad hoc samples of the input f as well. Note that a similar strategy with the U-Net architecture (i.e., adding scaled f inputs at all mesh levels) did not make any significant difference.

4.2 The Problems

The experimental validation of the two multi-resolution architectures proposed in the previous section will be made on two physical problems associated with two different underlying PDEs. This section will first introduce a simple electrostatics problem. Next, the more challenging problem of simulating Navier-Stokes equations around airfoils will be introduced. In this section, we will describe the two systems in detail, explain how these problems are solved with traditional numerical methods, and finally establish machine learning scenarios to address these challenges and experiment with the two multi-resolution architectures that have been proposed in the previous section.

4.2.1 A nonlinear Poisson Equation

Poisson's equation is an elliptic PDE that arises in many physical problems, including electromagnetism, heat transfer, and fluid dynamics.

In the field of electrostatics, Poisson's equation is used to determine the electric potential V given a charge density ρ. In electrostatics, the field around charges $\vec{D}$ is described by Gauss' law:

$$\nabla \vec{D} = \rho \tag{4.3}$$

By introducing the constitutive relation $\vec{D} = \epsilon \vec{E}$, where ε is the permittivity of the medium and $\vec{E}$ is the electric field, we can rewrite the above equation as:

$$\nabla(\varepsilon \vec{E}) = \rho \tag{4.4}$$

As the electric field $\vec{E}$ can be expressed as the gradient of the electric potential V, $\vec{E} = -\nabla V$, by replacing $\vec{E}$ with V, the electrostatic problem is modeled by Poisson's equation:

$$\nabla(\varepsilon \nabla V) = -\rho \tag{4.5}$$

If the medium is homogeneous, ε is constant and 4.5 becomes the linear Poisson's equation. However, permittivity ε is generally not constant, and may vary depending on various factors depending on the position within the medium. In

this work, we will study a more complex non-linear Poisson's equation defined on a domain Ω, with boundary Γ, where ε is a function depending on the electric field:

$$\begin{aligned} -\nabla(D(u,x)\nabla u(x)) &= f(x) \ in \ \Omega \\ u(x) &= g(x) \ on \ \Gamma_D \\ \nabla u(x) \cdot n &= 0 \ on \ \Gamma_N \end{aligned} \tag{4.6}$$

where $u(x)$ is the function to solve representing the electric potential V, and $D(u,x)$ is the permittivity depending on u and the position x. The charge density is represented by $f(x)$. $g(x)$ is the Dirichlet boundary condition, defined on the boundary domain $\Gamma_D \subset \partial\Omega$. Given these functions f, D, and g, the value of interest u is obtained by solving the non-linear Poisson's equation.

Finite Element Approach

In this Section, we briefly recall how to numerically solve the non-linear Poisson's equation 4.6 with the finite element method algorithm presented in Section 2.1. FEM converts the nonlinear Poisson's equation into a sequence of linear systems that are solved using Picard iteration (In Section 2.2.2).

By simply replacing u in the non-linear terms $D(u,x)$ with a known solution from the previous iteration, the Poisson's equation becomes:

$$-\nabla(D(u^k,x)\nabla u^{k+1}(x)) = f(x) \tag{4.7}$$

In iteration $k+1$, the new solution u^{k+1} is computed by solving the linear system above. Iteratively, u^{k+1} will converge to the real solution of the Non-linear Poisson's equation.

Denoting $a(x)$ the term $D(u^k,x)$, the linear equation (4.7) becomes the classical Poisson's equation:

$$-\nabla(a(x)\nabla u(x)) = f(x) \tag{4.8}$$

The finite element method first converts the equation above to a variational problem by multiplying the test function v on each side and integrating the whole

equation over the domain Ω. The variational problem is defined as follows: Find a solution $u \in V$ such that:

$$\begin{aligned} -\int_\Omega (\nabla(a(x)\nabla u(x)))v dx = \int_\Omega f v dx \quad \forall v \in \hat{V} \\ V = u \in H^1(\Omega) : u = g \ on \ \Gamma_D \\ \hat{V} = v \in H^1(\Omega), v = 0 \ on \ \Gamma_N \end{aligned} \tag{4.9}$$

To discretize the variational problem, the two function space V and $\hat{V}$ are discretized. The discrete problem becomes: find $u_h \in V_h \subset V$ such that:

$$-\int_\Omega \nabla(a\nabla u_h)v_h dx = \int_\Omega f v_h dx, \ \forall v_h \in \hat{V_h} \subset \hat{V} \tag{4.10}$$

By using Green's identity for the integration, the problem equals to:

$$\int_\Omega a\nabla u_h \nabla v_h dx = \int_\Omega f v_h dx \tag{4.11}$$

There are a lot of choices on V_h. Generally, V_h is a space of piecewise polynomial functions. Assume that $\{\Phi_j\}_{j=1}^N$ is a basis for the subspace V_h and $\{\hat{\Phi}_i\}_{i=0}^N$ is a basis for the subspace $\hat{V}$, N represents the dimension of the two spaces. u_h can be represented as a combination of the basis Φ_j:

$$u_h = \sum_j u_j \Phi_j \tag{4.12}$$

By varying the function v over its basic functions, the finite element discretization problem is obtained:

$$\sum_j u_j \int_\Omega a\nabla\Phi_j \nabla\hat{\Phi}_i dx = \int_\Omega f\hat{\Phi}_i dx, \ \forall i \in 1, 2, .., N \tag{4.13}$$

The solution u_h is then computed by solving the linear problem:

$$Au = b \tag{4.14}$$

where $A_{ij} = \int_\Omega a\nabla\Phi_j\nabla\hat{\Phi}_i dx$ depending on the form of basis functions, $b_i = \int_\Omega f\hat{\Phi}_i dx$, and $U_j = u_j$. The assembled matrix A is a sparse matrix. If and only if the nodes i and j share an edge on the mesh or $i = j$, A_{ij} is a non-zero value.

A common way to solve such a linear system is the Jacobi method mentioned in Section 2.2.1. Successive approximations are used to improve the solution at each step:

$$u_i^{t+1} = \frac{1}{A_{ii}}(b_i - \sum_j A_{ij} u_j^t) \tag{4.15}$$

The method starts with an initial guess u_0, and the sequence u_i after each iteration will converge to the solution u. From the expression, we observe that the computation of the new approximation u_i^{k+1} only requires u_j^k where j is the neighbor of the node i. The solution process for the non-linear Poisson's equation involves applying the Jacobi method iteratively with different values of A_{ij} and b_i.

As previously discussed, $A_{ij}, B_{ij} = 0$ if j doesn't share an edge of the triangulation with i on the mesh. The iterative process on node i is indeed aggregating information from its neighbors j, which can be viewed as a special graph neural layer.

Machine Learning Scenarios on Poisson's Equations

Even though the FEM can perfectly approximate the solution of electrostatic problems using the steps above, it takes high computation costs, especially when tens of thousands of problems are to be solved. Once given a new problem, FEM reconstructs the discrete model and solves new linear systems despite the fact that these problems share the same physical law. In such a context, machine learning can be extremely helpful. Machine learning algorithms learn hidden patterns from experience gained from solving numerous problems of the same type and use them to efficiently compute solutions to unseen problems. We propose three machine learning scenarios on Poisson's equations of increasing difficulty.

Various charge density f: In the first scenario, we consider a set of electrostatic problems with different charge density f but the same domain Ω and boundary conditions. We will use a machine learning model to predict the solution u given a new function f.

Different Dirichlet conditions g: The second scenario adds the additional challenge of different Dirichlet conditions g, requiring the model to be able to

handle distinct input functions f and g.

Unfixed Physical Domain Ω: The third and most challenging scenario involves solving problems with varying physical domains Ω, charge density f, and boundary conditions g. This scenario is the most general case, as it includes the most variables and requires the model to be able to adapt to different problem domains.

In our study, we begin by examining the simplest case of the electrostatics problem. If the deep learning model performs well in this simple case, we will gradually increase the complexity of the problem. By starting with the simplest case and gradually complicating it, we can better understand the capabilities of deep learning in solving more complex problems.

4.2.2 Airfoil Flow Simulation

In the field of computational fluid dynamics (CFD), the turbulence flow around an airfoil is a well-studied physical problem. The goal is to compute the velocity and pressure distributions of the flow around the airfoil when it is immersed in a fluid such as air. Reynolds-Averaged Navier-Stokes Equations (RANs) can be used to model the complex physical system.

However, traditional CFD solvers that use the finite volume method mentioned in Section 2.1 often have high computational costs, especially when dealing with complex non-linear equations like RANs.

CFD Background

Reynolds-averaged Navier–Stokes equations [140] (RANS equations) are time-averaged equations of motion for fluid flow. The simulation of the airfoil flow system with RANs equations is described by:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{4.16}$$

$$u_j \frac{\partial u_i}{\partial x_j} = \frac{\partial}{\partial x_j}[-p\delta_{ij} + (v + v_t)(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})] \tag{4.17}$$

where $U = (u_x, u_y)$ and p are velocity and pressure, the unknown variables to identify, ν is the fluid viscosity considered as a constant depending on the fluid

property, and ν_t is the eddy viscosity. ν_t can be further solved by Spalart–Allmaras equation [141], a commonly used model. The model is expressed as follows:

$$\begin{aligned}
\nu_t = \tilde{\nu} f_{v1}, \quad f_{v1} = \frac{\chi^3}{\chi^3 + C_{v1}^3}, \quad \chi := \frac{\tilde{\nu}}{\nu} \\
u_j \frac{\partial \tilde{\nu}}{\partial x_j} = C_{b1}[1 - f_{t2}]\tilde{S}\tilde{\nu} + \frac{1}{\sigma}\{\nabla \cdot [(\nu + \tilde{\nu})\nabla\tilde{\nu}] + C_{b2}|\nabla\tilde{\nu}|^2\} \\
- \left[C_{w1} f_w - \frac{C_{b1}}{\kappa^2} f_{t2}\right] \left(\frac{\tilde{\nu}}{d}\right)^2 + f_{t1}\Delta U^2 \\
\tilde{S} \equiv S + \frac{\tilde{\nu}}{\kappa^2 d^2} f_{v2}, \quad f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}} \\
f_w = g\left[\frac{1 + C_{w3}^6}{g^6 + C_{w3}^6}\right]^{1/6}, \quad g = r + C_{w2}(r^6 - r), \quad r \equiv \frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2} \\
f_{t1} = C_{t1} g_t \exp\left(-C_{t2}\frac{\omega_t^2}{\Delta U^2}[d^2 + g_t^2 d_t^2]\right) \\
f_{t2} = C_{t3} \exp\left(-C_{t4}\chi^2\right) \\
S = \sqrt{2\Omega_{ij}\Omega_{ij}} \\
\Omega_{ij} = \frac{1}{2}(\partial u_i/\partial x_j - \partial u_j/\partial x_i)
\end{aligned}$$

where σ, χ, and all quantities denoted with a "C" are constants specific to the model, calibrated through experimentation. The Spalart-Allmarasa equation to model eddy viscosity combined with RANs equations forms a system of four PDEs in two-dimensional space. This system is typically solved using the Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) algorithm, which assumes that the fluid is incompressible.

Machine Learning Scenario on Airfoil Flow

The airfoil flow system is characterized by two physical quantities: the Angle-of-Attack (AoA) and the Mach number. The AoA describes the angle between the airfoil and the incoming fluid flow, while the Mach number is a dimensionless quantity representing the ratio of the flow velocity past the airfoil to the local speed of sound. Only subsonic cases with Mach numbers smaller than 0.3 are considered in this study. Although these two quantities are low-dimensional, the airfoil simulation is a complex problem as the two parameters can lead to diverse fluid flow behaviors.

In this work, we aim to construct a machine learning model that can accurately predict the behavior of fluid flows around an airfoil given its associated mesh and the AoA and Mach number. We will evaluate the performance of the proposed model by applying it to a set of flow problems defined on different airfoil shapes and with various physical parameters. This will allow us to assess the ability of the model to generalize to complex physical systems.

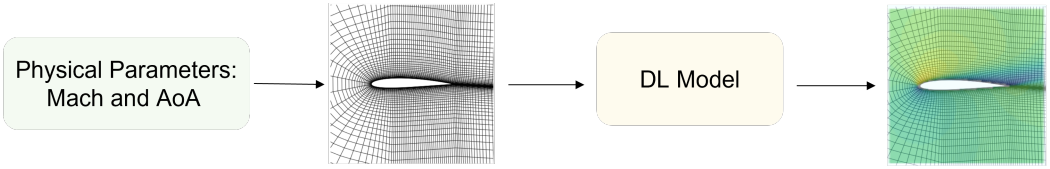


Figure 4.2: Airfoil Flow Scenario

4.3 Validation on Poisson's Equation

This section presents some significant results obtained on Poisson's equation that validate the proposed multi-grid GNN approaches.

Note that several other experimental campaigns on the Poisson's equation have been published in our ICANN'21 paper "Multi-resolution Graph Neural Networks for PDE approximation" [37], that addresses the same nonlinear Poisson's equations under different machine learning scenarios (and in particular different geometric domains). These published results are similar to those presented here, but the ones presented below allow a more unified view of the overall results.

We will use here the three scenarios for Poisson's equation discussed in Section 4.2.1. For each scenario, the problem statement and data collection are first introduced. After proper hyper-parameters tuning, the models are trained and finally compared to some baseline approaches.

4.3.1 Various charge density f

In scenario 1, a set of electrostatics problems with different charge densities f are given on the same domain. We will learn a model to predict u given the input function f. We fix $D(u,x) = 1 - u(x) + u(x)^2$ and apply a constant Dirichlet condition. The problem becomes:

$$\begin{aligned} \nabla((1 - u(x) + u(x)^2)\nabla u(x)) + f(x) = 0 \ in\ \Omega \\ u(x)|_{\Gamma_D} = 1 \\ \nabla u(x) \cdot n = 0 \ in\ \Gamma_N \end{aligned} \tag{4.18}$$

The domain is the "doughnut" $\Omega = \{(x,y) \in \mathbb{R}^2 | 0.09 \leqslant x^2 + y^2 \leqslant 1\}$ and apply the Dirichlet condition on the outer boundary $\Gamma_D = \{(x,y) \in \mathbb{R}^2 | x^2 + y^2 = 1\}$.

Data Preparation

To construct the learning dataset, we vary the function $f(x)$, and use the FEM solver *Fenics* to solve each problem in turn, i.e., to compute the approximated nodal values of the solution u on the mesh.

Mesh Generation To construct the hierarchical models, four scales of unstructured mesh (in Figure 4.3) are generated by *CGAL* for the same domain Ω, with respectively 32, 105, 384, and 1478 nodes. *Fenics* is applied on the finest mesh with 1478 nodes to obtain some high-quality approximations, considered as the "ground truth" in the following. These irregular triangle sub-domains can better

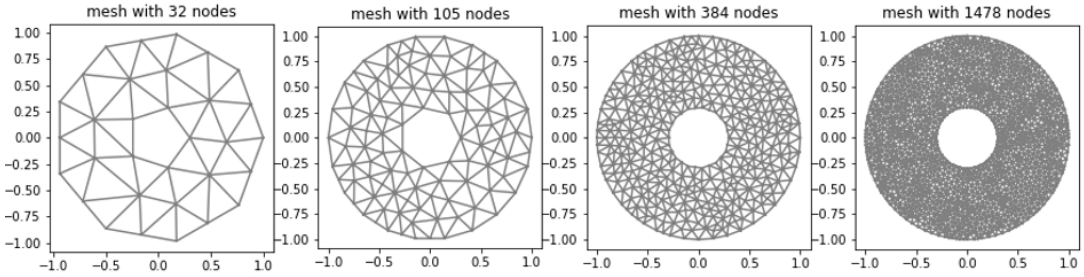


Figure 4.3: Different resolution of meshes on Ω

capture the geometry of Ω than a structured mesh consisting of regular rectangles (shown in Figure 4.4).

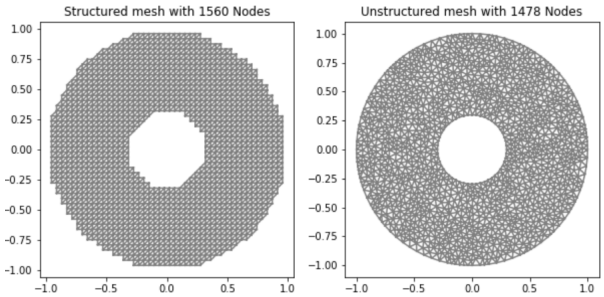


Figure 4.4: A comparison of structured (left) and unstructured (right) meshes with similar resolutions

Input function In Equation (4.18), the source terms f are generated randomly as a linear combination of eight isotropic Gaussian functions. This results in 32 control parameters: the coordinates of the means of the Gaussian functions and their standard deviations, and their respective weights in the linear combination. These parameters are chosen uniformly from domain-dependent intervals. 8 000 data are generated as the training set.

$$f(x) = \sum_{i=1}^{8} \frac{C_i}{\sigma_i} exp(\frac{(x - x_i)^2 + (y - y_i)^2}{2\sigma_i^2})$$

These 32 parameters are sampled under uniform distributions as follows: the weight $C_i \sim \mathcal{U}(-5, 5)$, the coordinate of the means $(x_i, y_i) \sim \mathcal{U}([-1, 1]^2)$, finally the standard deviation $\sigma_i \sim \mathcal{U}(0.1, 1)$.

Inputs and Outputs for GNNs As expressed in Section 4.1.1, the mesh data can be converted to a graph with node et edge features. The input function f defined on the continuous domain Ω is converted into node features by taking the values of the function f at each node. The output of the GNN model is the nodal values of the solution u on the mesh. The model performs node-level prediction with a graph structure and node content information as inputs plus the geometrical edge features described in Section 4.1.

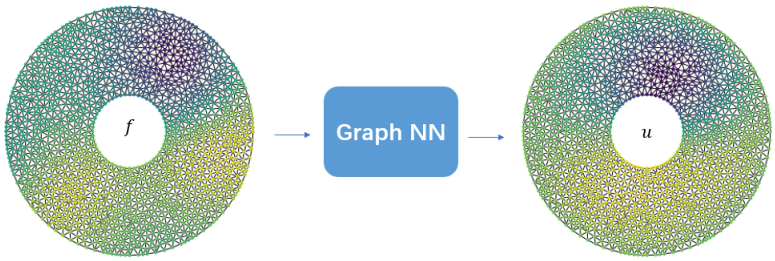


Figure 4.5: Input and output function are expressed in a format of graph

Hyper-parameters tuning

To ensure that our model has a high level of performance, we carefully consider several key hyperparameters. These include the overall structure of the model, including the number of graph layers and channels, as well as the initial value and decay strategy of the learning rate. We also carefully select an appropriate loss function. A 5-fold cross-validation is applied for hyper-parameters tuning to reduce the uncertainty. At each fold, the entire dataset is separated into a training set with 6 400 samples and a validation set containing 1 600 samples. Then hyper-parameters will be selected due to the validation errors across the five folds. Moreover, the training epochs are fixed as 1 000. After then, the model with the best performance is evaluated on a test set containing 2 000 examples.

Learning Rate Decay Taking Graph U-Net as an example, the learning rate

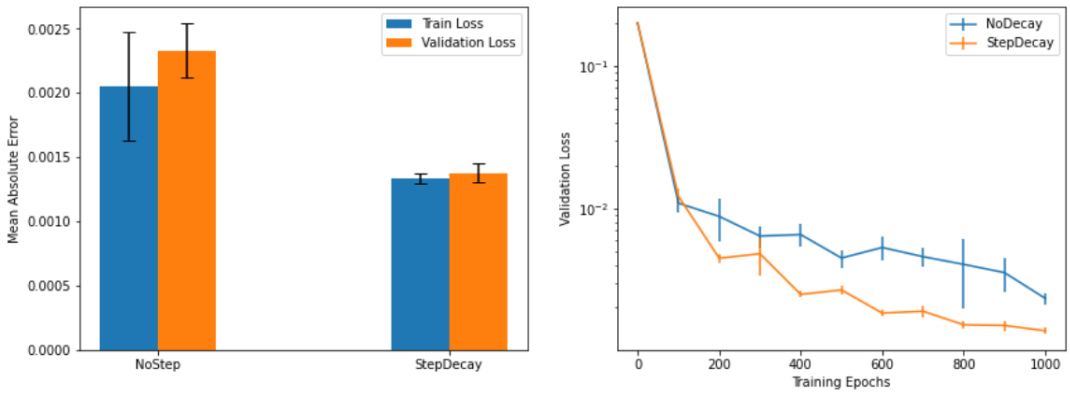


Figure 4.6: Comparison of model performance between applying or not learning rate decay

is one of the most important hyperparameters to consider. A small learning rate can increase the probability that the model will get stuck in a bad local minimum or saddle point, while a large learning rate may prevent the model from converging. We found that using a learning rate decay strategy, as discussed in Section 3.2.2, is helpful for training neural networks. A large learning rate at the beginning can accelerate convergence and help the model escape saddle points while gradually reducing the learning rate prevents us from missing local minima. In our experiments, we use a step decay strategy that reduces the learning rate by half every 200 epochs.
When adjusting the learning rate decay, we kept the other hyperparameters fixed: The graph model contains 2 layers and 48 channels for each GNN block, and we train the model using the Adam optimizer with an initial learning rate of 0.001.

A comparison of training the model with and without learning rate decay is shown in Figure 4.6. The model performance is vastly improved by applying the step learning rate decay. Moreover, the training process becomes more stable.

Decaying Strategy There are a variety of learning rate decay strategies (see Section 3.2.2). We evaluate the performance of models trained with different strategies, including step decay, exponential decay, and automatic decay. The other hyperparameters remain the same. From Table 4.1, step decay, exponential decay, and automatic decay have similar performance, but step decay is the most stable among different validation sets. Therefore, we will use step decay as our

Strategies	StepLR	ExponentialLR	AutomaticLR
MAE	0.0014 ± 0.0001	0.0014 ± 0.0003	0.0015 ± 0.0002

Table 4.1: Validation error using different learning rate decay strategies

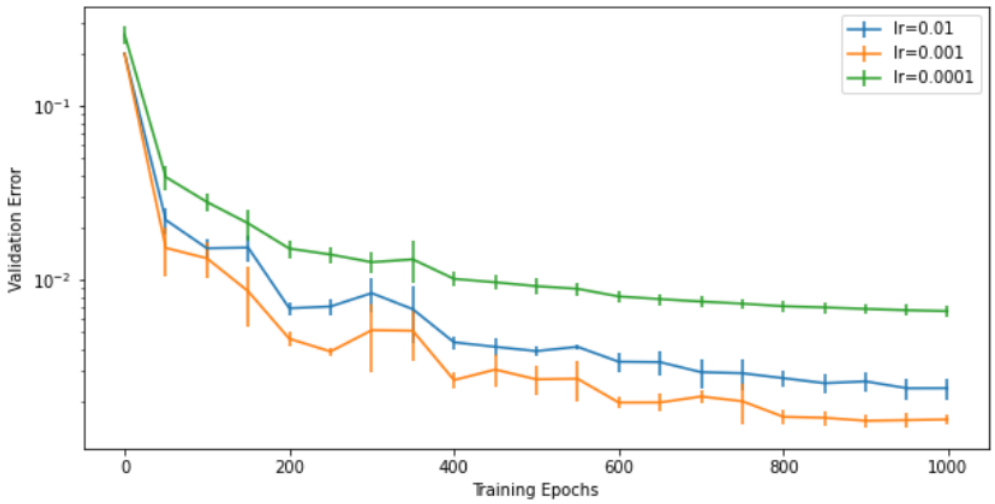


Figure 4.7: Comparison of models trained with different initial learning rate decay

strategy to anneal the learning rate for future experiments.

Initial Learning Rate In order to further improve the performance of our model, we must carefully consider the choice of the initial learning rate for our learning rate decay strategy. In our experiments, we tested a range of learning rates, including 0.0001, 0.001, and 0.01, on the graph model containing 2 layers and 48 channels for each GNN block and found that the model trained with a learning rate of 0.001 yielded the best results. As shown in Figure 4.7, this model outperformed the others, achieving a higher level of accuracy and stability during the training process. Therefore, we will use a learning rate of 0.001 in combination with our step decay strategy for all future experiments.

Model Structure Another important factor that can impact the performance of our model is the choice of its structure. A model with a simple structure may not be able to capture the complex patterns present in the training data, while a complex model may be prone to overfitting and require more resources

to train. In our experiments, we use a Graph U-Net model with four different mesh resolutions. This model consists of a series of graph blocks and sampling operators. Each graph block C is composed of multiple graph layers followed by an "elu" activation function (see Section 3.2). For example, a block $C = (4, 128, 5)$ would consist of four graph layers, each with 128 hidden channels and a kernel size of 5. After each graph block, a sampling operator is applied to transform the data between two mesh levels. The sampling operator has one hyperparameter, the number of nearest neighbors n. After some preliminary experiments, we set $n = 6$ for all our experiments. Finally, a graph layer is applied to map the high-dimensional features to the solution space.

From Table 4.2, we found that the model with 128 channels and 2 layers and that with 48 channels and 4 layers on each block have similar performance on validation sets. In this case, we prefer to choose the simplest model, as it has fewer trainable parameters and thus a reduced risk of overfitting. After tuning these crucial hyper-parameters, the best model for Graph U-Net is that with 48 channels and 4 layers trained with Step Decay every 200 epochs from an initial learning rate 0.001.

Table 4.2: Comparison models of different structures

	Num. of Channels	Num. of Parameters (1e5)	MAE(1e-3)
Layers=2	32	1.1	1.8 ± 0.3
	48	2.5	1.4 ± 0.1
	64	4.4	1.2 ± 0.1
	128	17	**1.0 ± 0.1**
Layers=4	32	2.1	1.2 ± 0.3
	48	4.7	**1.0 ± 0.3**
	64	8.3	1.4 ± 0.5
	128	3.3	1.6 ± 0.8

Loss Function Mean absolute error (MAE) and mean squared error (MSE) (see Section 3.2.4) are commonly used as loss functions for regression problems. MAE measures the average absolute difference between the predicted values and the true values, while MSE measures the average squared difference. One advantage

of MAE is that it is more robust to outliers, as it does not amplify the effects of individual errors as much as MSE. However, MSE can penalize large errors more heavily, which can be beneficial in some cases. In our experiments with the U-Net model, we tested both loss functions and found the results as follows:

Table 4.3: Comparison of Loss functions

Loss Function	Evalutation Metrics (1e-2)		
	MAE Error	RMSE Error	Max Absolute Error
MAE	1.0 ± 0.3	2.2 ± 0.7	5.8 ± 1.7
MSE	1.2 ± 0.2	2.1 ± 0.3	4.7 ± 1.2

The choice of loss functions has relatively little effect on the performance of the model. We take MSE loss as the loss function by taking into consideration of Max Error.

We use the same strategy for hyperparameter tuning in Graph-MGMI and focus on the model structure. All models were trained using step decay on an initial learning rate of 0.001. Through our experiments, we found that a model structure with 48 channels and 8 layers per block yielded the best results.

Baseline Models

In this experiment, we compare the performance of our hierarchical graph models to two baseline approaches. The first baseline is based on a pixelization method, where we project all mesh data onto a pixel grid using interpolation or extrapolation and then apply standard convolutional neural network (CNN) layers directly to the pixelated data (as illustrated in Figure 4.8). This simplifies the problem into an image reconstruction task, and the prediction from the CNN model is then mapped back to the original mesh space. However, as shown in Figure 4.8, the preprocessed image data does not accurately represent the solution and its underlying geometry. The second baseline is a pure GNN model that operates only on the finest mesh scale and directly predicts the solution from the input. This model does not incorporate any of the hierarchical structures. All the hyperparameters of these baselines have been set according to a procedure similar to the one described above for our multi-grid architectures; see values below.

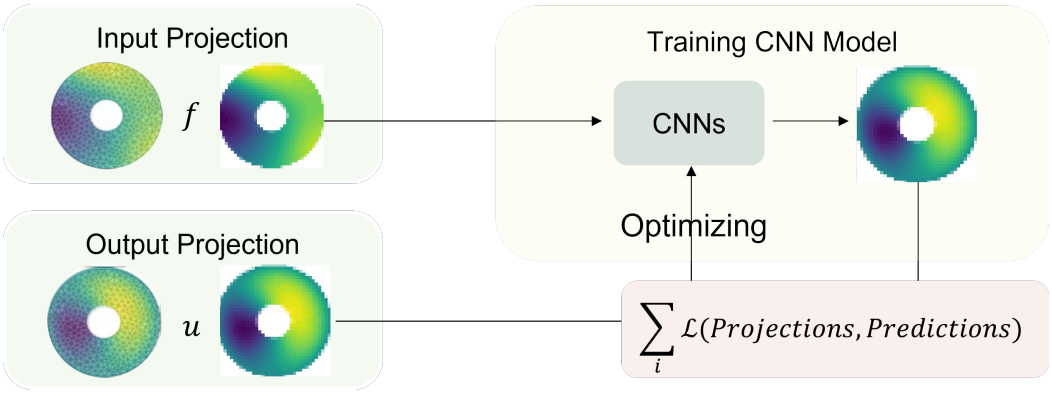


Figure 4.8: CNN Baseline model Strategy

Training the Models

After completing the hyperparameter tuning process, we evaluate the performance of the model using 10-fold cross-validation on the dataset with the best-chosen hyperparameters. Models trained on the different folds are then evaluated on a test set.

The activation function used between consecutive layers is the exponential linear unit (ELU) function. We initialize all models using the Kaiming distribution (as described in Section 3.2.3), which is recommended when using the ELU or rectified linear unit (ReLU) activation functions. To prevent overfitting and ensure that the models are fully trained, we use early stopping during training. The Adam optimizer is used in combination with a step decay scheduler, with the learning rate halved every 500 epochs. All models are trained with a batch size of 100 (several previous experiments showed that the results are not very sensitive to the batch size and that the limiting factor is the size of the available memory on the GPUs).

We train a *Graph U-Net* model with 48 hidden channels and 33 graph layers, using an initial learning rate of 0.001. The *Graph MGMI* model also has 33 layers and 48 hidden channels for each layer and is trained using an initial learning rate of 0.0005. The *Pure Graph* model has the same number of layers and hidden channels as the other two models and is trained using an initial learning rate of 0.001. Finally, the *CNN* model is trained on the projected image-like dataset, using a U-Net architecture to solve the image reconstruction task. This model has 33 layers with 32 hidden channels and is trained using an initial learning rate of 0.0005.

Evaluation

In addition to evaluating the performance of our model on a test set with the same distribution as the training set, we also assess its ability to generalize to out-of-distribution (OoD) samples. In practice, the training and test sets are rarely sampled from the same distribution, despite the fundamental assumption of machine learning. We, therefore, expect our model to perform well on OoD samples underlying the same physical rules as the training samples. To this end, we create three OoD test sets with different input distributions:

- Exponential Set: We generate new examples by the same distribution used when generating the training set. The function f can be expressed as a linear combination of eight exponential functions.
- Sine Set: The function f is expressed as a trigonometric function. We have

  $$f = C\sin(\frac{2\pi}{T}[(x-x_0)^2 + (y-y_0)^2]),$$

  where the parameters C, T, x_0, y_0 are randomly sampled from uniform distributions. $C \sim \mathcal{U}[0, 10]$, $T \sim \mathcal{U}[1, 5]$, and $(x_0, y_0) \sim \mathcal{U}([-0.5, 0.5] \times [-0.5, 0.5])$.
- Polynomial set: The function f is represented as a polynomial function. The expression is as follows:

  $$f = A_1x^2 + A_2y^2 + B_1x + B_2y + C,$$

  the distributions of the five parameters above are: $A_1, A_2 \sim \mathcal{U}[-5, 5]$, $B_1, B_2 \sim \mathcal{U}[-10, 10]$, and $C \sim \mathcal{U}[-5, 5]$

Each of these test sets contains 2 000 examples.

Results on Exponential Set: As shown in Table 4.4, the results of all algorithms on the exponential set indicate that Multi-resolution models outperform pure graph models. Furthermore, the training time for pure graph models is also longer in the context of graph-based approaches.
As shown in Figure 4.9, Graph-based models demonstrate superior performance compared to CNN-based models. However, it should be noted that CNN models

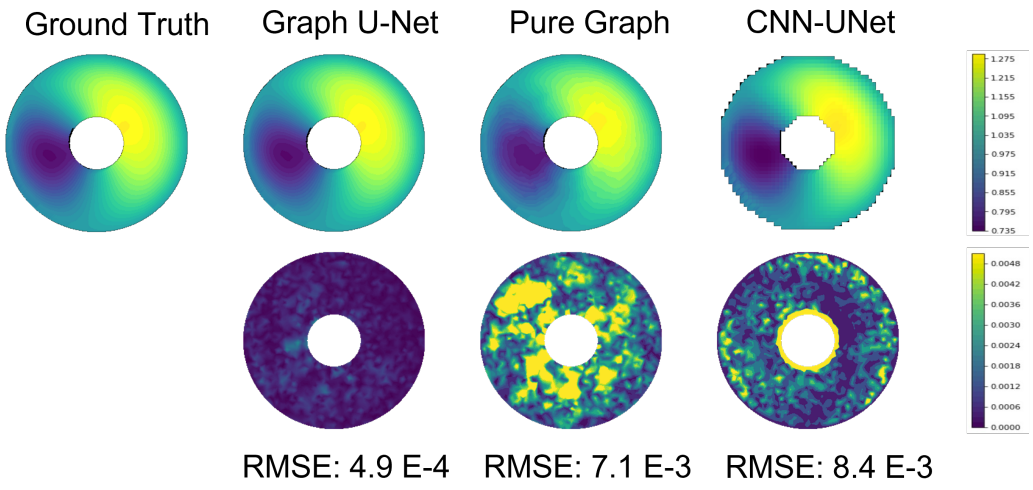


Figure 4.9: Predictions of a new Poisson's equation by three models

exhibit strong performance on regular grids, and the observed discrepancy can be attributed to interpolation error. Furthermore, errors in the CNN-based model tend to concentrate near the boundary of the domain, which may be due to the limitations in the geometry representation of regular grids. In terms of graph-based approaches, it appears that Graph U-Net slightly outperforms Graph MGMI in the case of donut-shaped data.

Table 4.4: Results on the exponential set

Model	Evalutation Metrics				
		n-th percentile (1e-3)			
	RMSE(1e-3)	10	50	90	R-Square
Graph U-Net	1.17 ± 0.19	0.40 ± 0.06	0.66 ± 0.11	1.72 ± 0.29	0.99999 ± 0.00001
Graph MGMI	1.85 ± 0.30	0.38 ± 0.06	0.63 ± 0.09	2.38 ± 0.37	0.99997 ± 0.00002
Pure Graph	15.28 ± 3.57	3.69 ± 0.95	7.58 ± 2.20	24.39 ± 6.37	0.99791 ± 0.00090
CNN	17.17 ± 0.02	12.92 ± 0.02	16.24 ± 0.03	21.70 ± 0.03	0.99603 ± 0.00001

Results on out-of-distribution test sets: From Table 4.5 and 4.6, even when presented with examples from the Sine and Polynomial sets that differ significantly from those in the training set, the models are able to predict the output solution u accurately. In particular, the two hierarchical graph models exhibit strong

performance. In terms of out-of-distribution test sets, our proposed multi-resolution models demonstrate the ability to solve these problems effectively.

Table 4.5: Results on the Sine set

Model	Evalutation Metrics				
		n-th percentile (1e-3)			
	RMSE(1e-3)	10	50	90	R-Square
Graph U-Net	5.62 ± 0.71	0.47 ± 0.08	1.34 ± 0.22	7.27 ± 0.94	0.99119 ± 0.00218
Graph MGMI	15.6 ± 2.5	0.63 ± 0.09	2.57 ± 0.30	22.8 ± 3.7	0.95063 ± 0.02852
Pure Graph	49.6 ± 22.1	8.2 ± 2.8	20.6 ± 6.5	68.4 ± 25.1	0.90660 ± 0.02454
CNN	19.4 ± 1.0	15.6 ± 0.1	17.6 ± 0.1	21.7 ± 1.5	0.93394 ± 0.00390

Table 4.6: Results on the Polynomial set

Model	Evalutation Metrics				
		n-th percentile (1e-3)			
	RMSE(1e-3)	10	50	90	R-Square
Graph U-Net	0.85 ± 0.14	0.52 ± 0.08	0.79 ± 0.13	1.15 ± 0.20	0.99999 ± 0.00001
Graph MGMI	1.02 ± 0.14	0.58 ± 0.06	0.92 ± 0.11	1.40 ± 0.22	0.99997 ± 0.00001
Pure Graph	14.3 ± 6.6	7.0 ± 2.6	12.1 ± 5.1	20.7 ± 10.3	0.99612 ± 0.00298
CNN	16.1 ± 0.1	13.0 ± 0.1	15.4 ± 0.1	19.6 ± 0.1	0.99344 ± 0.00001

Statistical significance For all pairwise comparisons between test errors, we performed a Wilcoxon signed-rank statistical test (see Section 3.1.4) with 95% confidence on the results of the ten models obtained through the 10-fold procedure. The results indicate that all differences between the multi-resolution models and the baselines are statistically significant.

Evaluation of the three types of test sets has demonstrated that these hierarchical graph models can improve prediction accuracy on test sets with different source terms compared to the Pure Graph model. Additionally, the CNN model with projection performs worse than graph models for problems defined on complex physical domains due to interpolation errors.

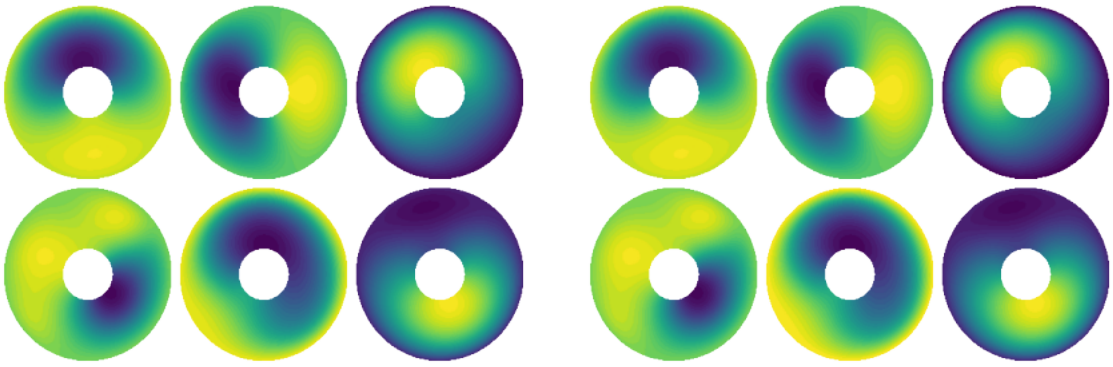


Figure 4.10: Visualization on the exponential test set. Left: Prediction by Graph U-Net. Right: Ground Truth

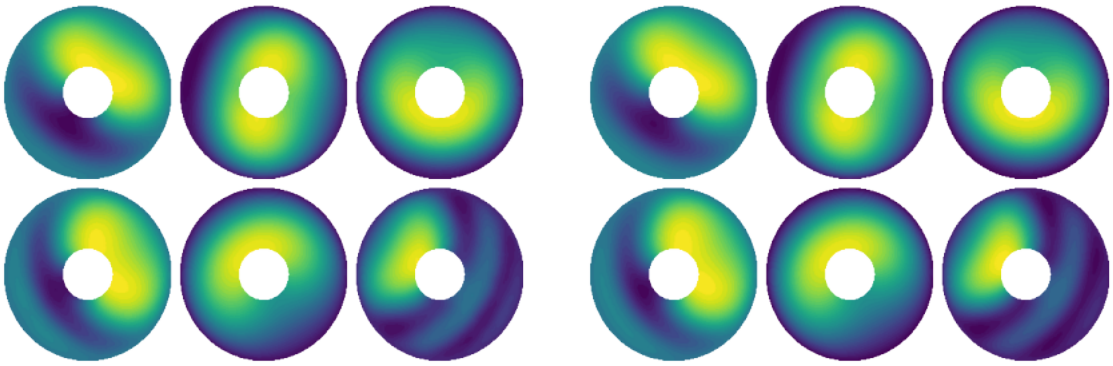


Figure 4.11: Visualization on sine test set. Left: Prediction by Graph U-Net. Right: Ground Truth

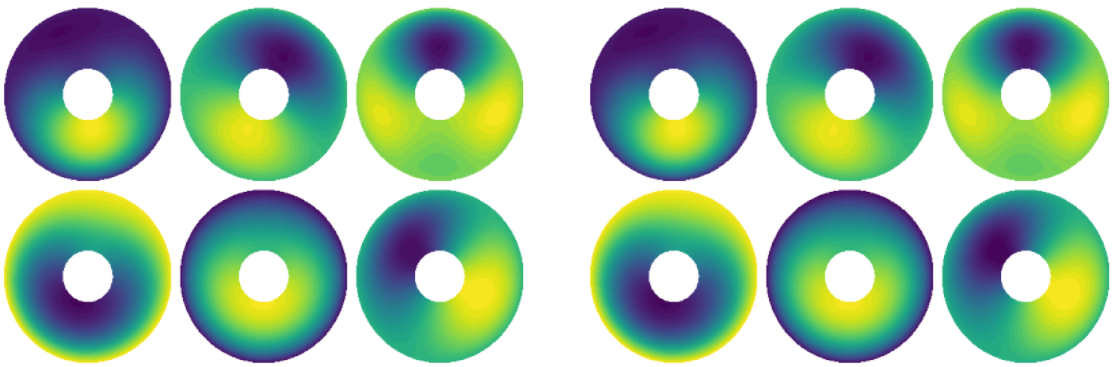


Figure 4.12: Visualization on polynomial test set. Left: Prediction by Graph U-Net. Right: Ground Truth

4.3.2 Different Dirichlet Condition

In this scenario, not only does the charge density f vary, but the Dirichlet condition g also varies. We will explore whether our proposed architectures are able to solve this more complex problem effectively.

Data Generation

Similar to the first scenario, we vary the external force f using a combination of exponential functions. Additionally, for each problem, we also change the Dirichlet condition g defined on the outer boundary.

$$g(x, y) = sin(\frac{2\pi x}{T}),$$

where $T \sim \mathcal{U}[1, 3]$. We generate 4 000 examples as the training set by randomly sampling parameters to create distinct input functions f and g.

Inputs to the GNN

Because this scenario involves two input functions, the external force f defined in Ω, and the Dirichlet condition g defined on the outer boundary of $\partial\Omega$, the input layer had to be re-scaled. The nodal feature v_i at node i is defined as $v_i = (f_i, g_i, h_i)$, where f_i is the value of function f at node i, and $g_i = g(i)$ is the value of function g at node i if the node is located on the outer boundary, or 0 otherwise. h_i is a binary variable, which takes 1 if the node is on the outer boundary of $\partial\Omega$, and 0 otherwise. This variable serves as an indicator of whether a given node is located on the Dirichlet boundary.

Training models

The results from Scenario 1 indicate that the CNN baseline model performed poorly due to high interpolation errors between the mesh and grids. Additionally, the Pure Graph model, which did not incorporate a hierarchy, performed significantly worse than the other models. In this section, we only focus on the two hierarchical graph models: Graph U-Net and Graph MGMI. Both models have 48 channels and 33 layers. As before, we use the Adam optimizer to train the models. Graph U-Net has an initial learning rate of 10^{-3}, while Graph MGMI is

trained with an initial learning rate of 5×10^{-4}. Every 500 epochs, the learning rate is halved. We apply 10-fold cross-validation on the training set and compute the mean and standard deviation of the test errors.

Results

We generated 1 000 new examples with same feature distributions as the test set. Errors are recorded on Table 4.7

Table 4.7: Results on Non-linear Poisson's equations with different Dirichlet conditions

Model	Evalutation Metrics				
		n-th percentile (1e-3)			
	RMSE(1e-3)	10	50	90	R-Square
Graph U-Net	2.20 ± 0.28	1.03 ± 0.15	1.59 ± 0.23	3.14 ± 0.40	0.99998 ± 0.00001
Graph MGMI	2.51 ± 0.26	0.90 ± 0.08	1.44 ± 0.14	3.48 ± 0.30	0.99998 ± 0.00001

As the results from scenario 1, both models are capable of solving the problems with different Dirichlet conditions. Graph U-Net is slightly better than Graph MGMI.

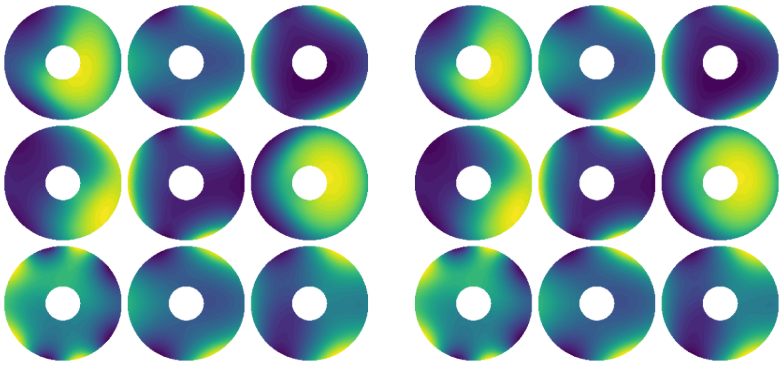


Figure 4.13: Some predictions by Graph U-Net. Left: Prediction, Right: Ground Truth

4.3.3 Variable Domain Ω

The graph neural networks have the capacity to solve nonlinear Poisson's equations with different source terms f. In this part, we will study a set of more complex problems. These Poisson's equations are defined on different physical domains Ω.

We generate polygonal domains by sampling 10 random vertices and setting a vertex at a random radius at each step while walking along a circle at a random angular step. This allows us to create a wide range of diverse domains.

$$\triangle\theta_i = U(\frac{2\pi}{n} - \epsilon, \frac{2\pi}{n} + \epsilon)$$

$$\theta_i = \theta_{i-1} + \frac{1}{k}\triangle\theta_i, k = \frac{\sum \triangle\theta_i}{2\pi}$$

$$r_i = clip(\mathcal{N}(r_{ave}, \sigma), 0, 2r_{ave})$$

where θ_i and r_i define the angle and radius of each point relative to the center of the circle. The random angular step $\triangle\theta_i$ is chosen from a uniform distribution U. r is sampled from a Gaussian distribution, and the $clip(., 0, 2r_{ave})$ thresholds the radius r_i into a range from 0 to $2r_{ave}$. n is the number of vertices of the polygon. The two parameters ϵ and σ control how irregular the polygon is. ϵ decides whether or not the vertices are uniformly spaced angular-wise around the circle, and σ controls how large the points can vary from the circle of radius r_{ave}.

We choose the average radius as 1. Figure 4.14 presents some polygons from the dataset. This scenario aims to evaluate the ability of our models to

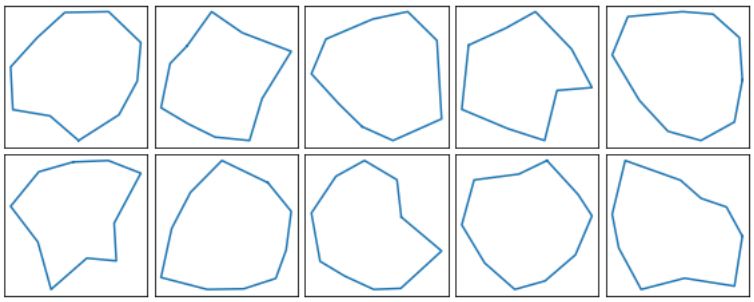


Figure 4.14: Some examples of polygons for scenario 3

solve problems defined on different geometric domains. Each training sample

is defined on a different polygonal domain (and hence a different mesh), also involving a different f and a Dirichlet condition g. We created a totally 4 000 data points as the training set.

Model Evaluation

We train the hierarchical models with 10-folds cross-validation. The test set consists of 1 000 new polygon problems sampled from the same distribution as the training set. Results are listed in Table 4.8. The R-square values indicate that the graph models exhibit strong performance on problems defined on diverse graphs, although the performance is slightly worse than on the static graph in the Donut problem.

Table 4.8: Results on polygon problems

Model	Evalutation Metrics				
		n-th percentile (1e-2)			
	RMSE(1e-2)	10	50	90	R-Square
Graph U-Net	1.43 ± 0.06	0.68 ± 0.02	1.12 ± 0.04	2.07 ± 0.10	0.99933 ± 0.00005
Graph MGMI	1.49 ± 0.10	0.54 ± 0.03	1.02 ± 0.05	2.24 ± 0.16	0.99930 ± 0.00009

Besides the polygon test set with the same distribution, we present two out-of-distribution experiments. We take the trained Graph U-Net as an example to discuss the capacity of model generalization on out-of-distribution sets.

Mesh complexity The first experiment (in Table 4.9) examines the impact of the number of nodes in the meshes. In the training set, the target meshes had an average of 1049 nodes, with a range of 900 to 1,200. We generate two test sets with the number of nodes in the ranges $[1,200, 1,600]$ and $[1,600, 2,200]$.

Domain shape Whereas the training set was made of polygons with 10 vertices, this second OoD experiment concerned shapes made with 5 and 20 vertices, keeping the number of mesh nodes approximately the same.
As expected, the performance degrades as we move away from the training distribution. However, the graph model is still able to maintain its performance on these out-of-distribution test sets.

Table 4.9: Test Errors on out-of-distribution sets

	Graph U-Net		Graph MGMI	
Dataset	RMSE(1e-2)	R-Square	RMSE(1e-2)	R-Square
Nodes 1200-1600	1.74 ± 0.09	0.99898 ± 0.00010	1.95 ± 0.16	0.99876 ± 0.00018
Nodes 1600-2200	3.01 ± 0.20	0.99703 ± 0.00037	4.35 ± 0.39	0.99440 ± 0.00089
5 Vertices	1.36 ± 0.06	0.99937 ± 0.00005	1.58 ± 0.08	0.99916 ± 0.00008
20 Vertices	1.72 ± 0.07	0.99907 ± 0.00007	1.62 ± 0.05	0.99920 ± 0.00005

4.4 Airfoil Flow Simulation

The Machine Learning scenario on Airfoil Flow problems (Section 4.2.2) is to construct a model which can predict the flows around an airfoil given its shape (with associated mesh) and the two flow parameters, Angle of Attack (AoA) and Mach number (Mach). To achieve this, we will train a model on a set of airfoil flows obtained with a standard numerical solver on a range of airfoils, AoA, and Mach. Based on the previous experiments on Poisson's equations, we discuss solely the Graph U-Net model in this case.

4.4.1 Data Generation

We generate a database of cases with various airfoil shapes, AoAs, and Machs. The CFD solver `OpenFOAM` is used to generate ground truth on C-grid meshes [142] designed for airfoil shape.

NACA Airfoil Generation To create a set of diverse airfoil shapes, we generate 80 NACA 4-digit airfoils characterized by their camber C, the position of their maximum camber P, and their thickness T [143].

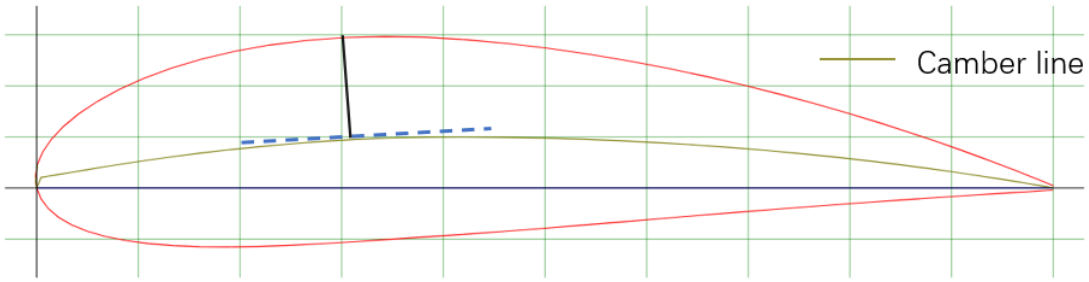


Figure 4.15: An example of NACA airfoil

The NACA airfoil section is drawn from its camber line and its thickness distribution, measured orthogonally to the camber line (Figure 4.15), that are themselves determined by the three parameters: the camber C, the position of maximum camber P, and the thickness T. The camber line is described by a function that is parameterized by C and P:

$$y_c = \begin{cases} \frac{C}{P^2}(2Px - x^2), & 0 \leq x < P \\ \frac{C}{(1-P)^2}(1 - 2P + 2Px - x^2), & P \leq x \leq 1 \end{cases}$$

While the thickness distribution is decided by a polynomial function:

$$y_t = \frac{T}{0.2}(a_0 x^{0.5} + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4)$$

,where $\{a_i | i = 0, .., 4\}$ are constants. These parameters allow us to control the shape of the airfoil.

To generate a diverse set of NACA airfoils, we uniformly sample the three parameters from uniform distributions: $C \sim \mathcal{U}[0, 0.09]$, $P \sim \mathcal{U}[0.4, 0.6]$, and $T \sim \mathcal{U}[0.1, 0.3]$. Examples of the generated airfoil shapes are shown in Figure 4.16.

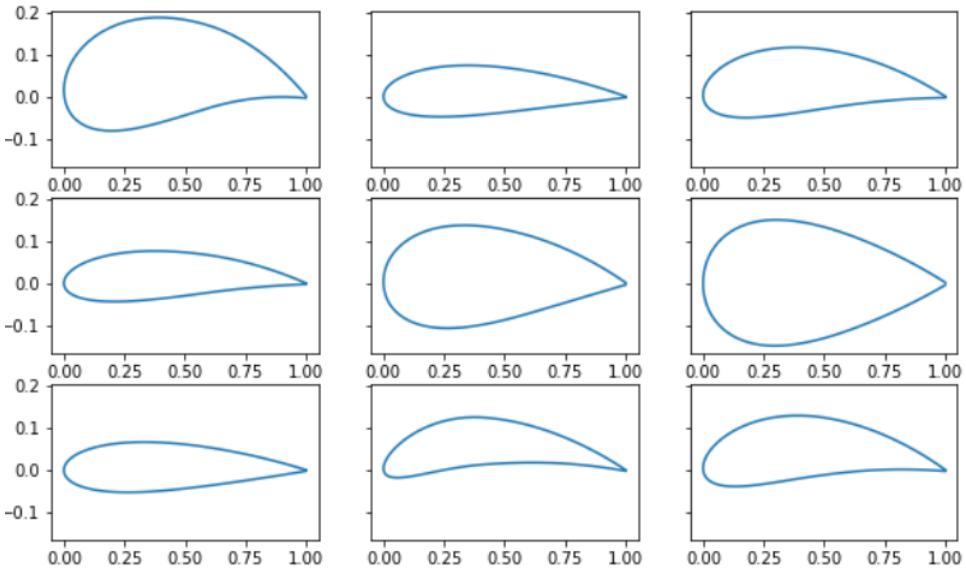


Figure 4.16: Examples of NACA airfoil shapes

Automatic Mesh Generation Each NACA airfoil is automatically meshed in a C-grid quadrilateral format using the algorithm provided by [142]. The algorithm creates a C-grid mesh given some inputs concerning mesh size, the domain, and the coordinates describing the discretized airfoil shape. Figure 4.17-left displays such a mesh. C-grid meshes are widely used for CFD analysis of an airfoil. In most cases, these meshes give a better convergence of flow over the airfoil than the Delaunay triangulation.

Ground Truth To generate ground truth data, we use the open-source CFD

software `OpenFOAM` to solve a set of airfoil flow problems. The flow around the airfoil is modeled using the Reynolds Averaged Navier-Stokes equations, and the Spalart-Allmaras equation is used to solve for the eddy viscosity μ_t, as detailed in Section 4.2.2.
For each airfoil, we uniformly sample the two quantities, AoA and Mach number, from their respective ranges: AoA $\sim \mathcal{U}[-22.5, 22.5]$ degrees and Mach $\sim \mathcal{U}[0.03, 0.3]$. This generates a total of 3 200 examples, with 40 examples for each of the 80 airfoils.

Data Pre-processing Data Generation by CFD solvers requires a large computational domain where the approximation of CFD simulation is calculated. An appropriate distance between the airfoil object and the borders of the computational domain should be large enough so that the boundary conditions assigned to the outer domain don't affect the quality of the flow simulation around the airfoil. However, only the area close to the airfoil is of interest to the engineer, and hence, it is not necessary that the ML model computes predictions for areas that are far away from the airfoil, where the flow is approximately equal to the incoming flow. We can instead focus on a small domain close to the airfoil, as shown in Figure 4.17. This allows us to save computational resources and improve the accuracy of ML predictions. Additionally, the output quantities (velocity

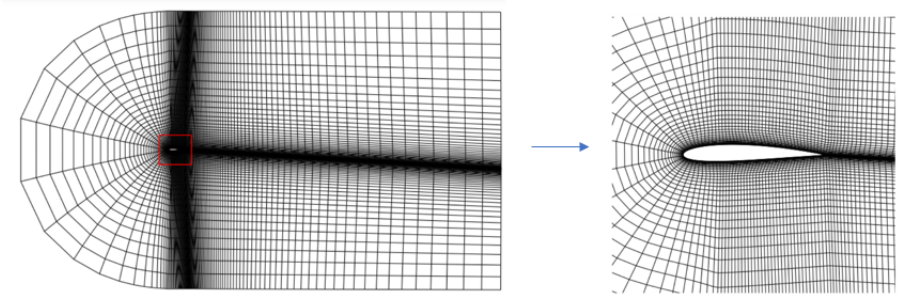


Figure 4.17: Left: Domain used by `OpenFOAM`, Right: the inference domain for Deep Learning

u and pressure p) are normalized relative to the magnitude of the freestream velocity to make them dimensionless, i.e.,

$$\bar{u} = u/||u_0||, \quad \bar{p} = p/||u_0||^2$$

.

Network Inputs and Outputs A two-dimensional quadrilateral mesh can be represented as $M = (P, C)$, where $P \in R^{N \times N}$ is a collection of N mesh nodes with coordinates (x, y), and $C = (i_1, j_1, k_1, l_1), ..., (i_M, j_M, k_M, l_M))$ is a set of M quadrilateral cells, each represented by nodal indices. In our case, the CFD solver `OpenFOAM` uses finite volume methods to discretize the PDE. The simulation results provide an approximate value of the output quantities at the center of each cell. The dof of the discretization (i.e., the centroids of each cell) are the nodes of the graph, and two centroids i and j are connected if their corresponding cells are adjacent (i.e., share an edge).
The initial freestream condition $v_0 = (v_{0,x}, v_{0,y})$ depending on AoA and Mach is considered as the two input channels of the deep learning model. The outputs have three channels representing nodal values of the normalized pressure p and the normalized velocity (v_x, v_y), coordinate of the flow velocity in the x, y system. For each node of the graph, we assign the same vector $(v_{0,x}, v_{0,y})$ as nodal features. The output with three channels denotes the prediction of (p, u_x, u_y) on each graph node.

4.4.2 Hyper-parameters Tuning

We mainly discuss two hyper-parameters, the number of layers and the channel size of each block in this case. The dataset is separated into the training set, Validation set 1, and Validation set 2.

Training and Validation Sets The dataset of 3200 examples is split into three parts:

- **Training set**: 64 different NACA airfoils with 30 examples for each airfoil.
- **Validation set 1**: the same 64 training airfoils containing 10 other samples for each airfoil, used to measure the performance of the trained model on known airfoils.
- **Validation set 2**: 16 new NACA airfoils with 40 examples each, the first 20 examples are used to update the initial model, and the other 20 are used to evaluate the model performance.

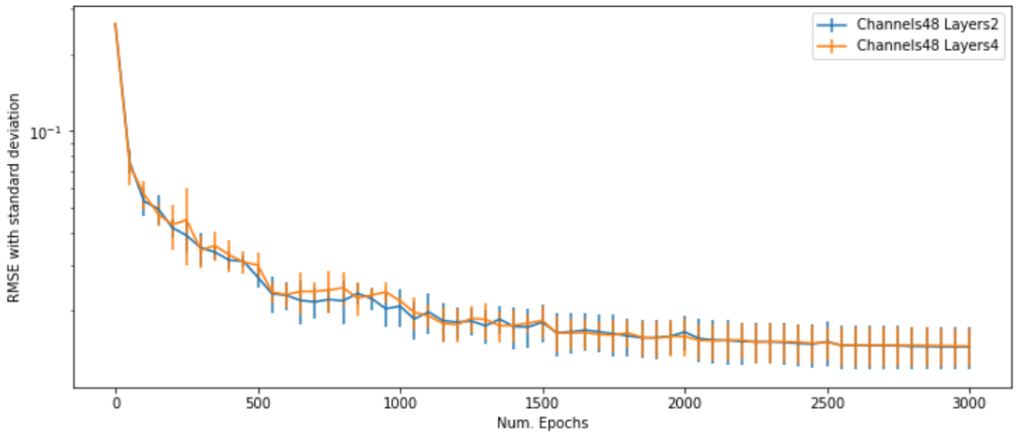


Figure 4.18: RMSE error on validation sets w.r.t. number of epochs. The two models have very similar convergence curve which signifies the lighter model takes less training time to reach to the same validation error.

We are not only interested in the performance of the model on unknown airfoils, but also on airfoils shown in the training set but with different flow parameters (AoA, Mach).

Layers	Channels	Num. Weights	RMSE Val1	RMSE Val2
2	48	250k	0.0069	0.0110
2	64	445k	0.0065	0.0129
4	24	119k	0.0080	0.0140
4	48	473k	0.0061	0.0130
8	24	231k	0.0087	0.0155

Table 4.10: Hyper-parameters tuning on Model Structure

The results of the hyperparameter search in Table 4.10 show that the three models (2, 64), (2, 48), (4, 48) have similar performance, and better than the others. We further apply 5-cross validation to analyze the stability of these three models. From table 4.11, the two models (4, 64) and (2, 48) have similar performance, while the model (2, 64) is less stable compared with the others. Combined with Figure 4.18, we choose the lighter model by taking into consideration the training time.

After choosing the best model structure, we train the model with 10-fold cross-validation.

(Layers, Channels)	(2, 48)	(4, 48)	(2, 64)
Validation Set1	0.0065 ± 0.0003	0.0062 ± 0.0006	0.0066 ± 0.0006
Validation Set2	0.0142 ± 0.0026	0.0143 ± 0.0024	0.0166 ± 0.0050

Table 4.11: Hyper-parameters tuning on Model Structure with 5-cross validation

4.4.3 Experimental Results

We use three different test sets to validate the performance of our Graph-UNet model and we will mainly discuss the generalization error w.r.t. airfoil shapes. For the Ao and Mach, we use the same distribution as for the training set.

- **Flow Interpolation test set**: We generate 10 more examples for each training airfoil. For evaluation, we can directly use the training data to update each task and make further predictions for new examples.
- **Shape Interpolation test set**: It contains 20 airfoils which are generated using the same distributions of NACA parameters. 20 samples are created on each airfoil with a total of 400 data.
- **Out of Distribution test set** (OoD) – Thinner Airfoils: By changing the range of distribution on T and P, we can create airfoils that are thinner and more irregular ($P \in [20, 80]$ and $T \in [5, 10]$). We generated 20 thinner airfoils, with 20 new examples for each airfoil by varying AoA and Mach.

As for evaluation metrics, we calculate root mean squared error and R-Square Score for all the test sets. Moreover, we also compute the k-th percentile, a score in which a given percentage k falls, with k = 10, 50, 90.

Table 4.12: Results on Airfoil Flow problems

Test Sets	Evalutation Metrics				
		n-th percentile (1e-2)			
	RMSE(1e-2)	10	50	90	R-Square
Flow Interpolation	0.71 ± 0.03	0.48 ± 0.03	0.61 ± 0.03	0.89 ± 0.04	0.9991 ± 0.0001
Shape Interpolation	1.58 ± 0.15	0.61 ± 0.06	1.08 ± 0.09	2.36 ± 0.26	0.9962 ± 0.0006
Out-of-Distribution	6.22 ± 1.66	1.55 ± 0.18	3.60 ± 0.53	10.55 ± 3.41	0.9299 ± 0.0334

By comparing the predicted results with the ground truth in Table 4.12, we can conclude that the model has good performance on samples from the training set and the interpolation set but performs poorly on the out-of-distribution test set. This is a frequent disadvantage of data-driven methods, as they tend to learn patterns from the training data rather than understanding the underlying physical constraints. The error distribution on the out-of-distribution set is less

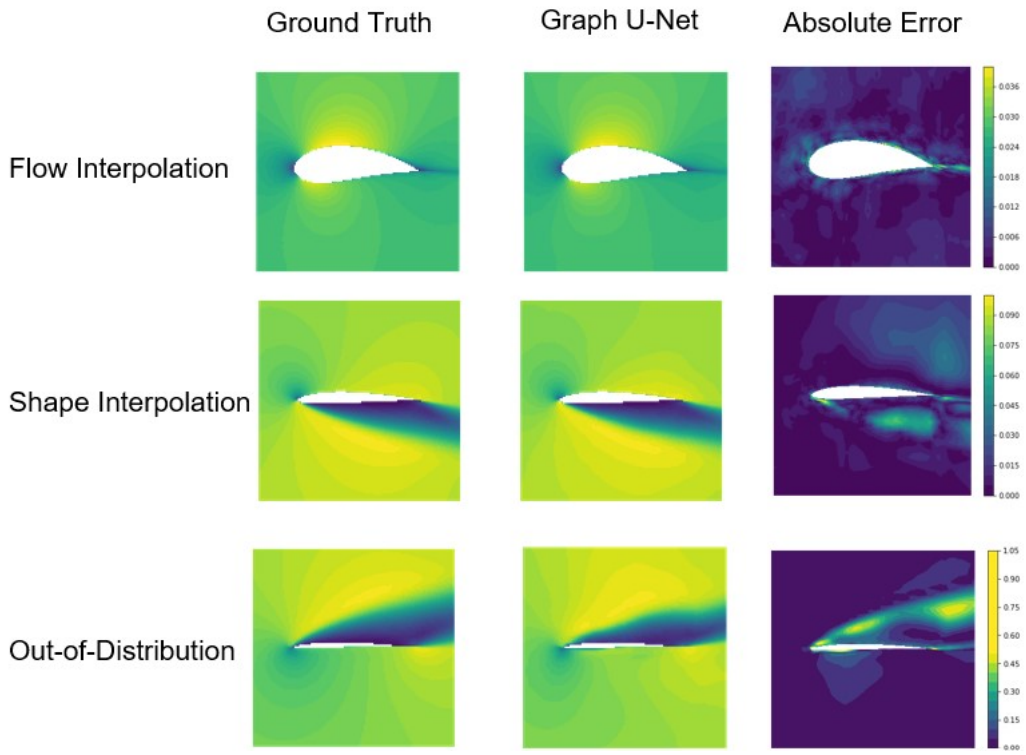


Figure 4.19: Predictions from three test sets

centered and has a larger variance, as shown in Figure 4.20. Finally, we randomly plot a set of predictions from all three test sets, we can see that the model is able to accurately predict the behavior of the flow for samples from the training and interpolation sets, but it fails to do so for the out-of-distribution set (Figure 4.27).

Errors on surface of the airfoils In our previous discussions, we have extensively analyzed the total RMSE across the entire domain of inference. However, in practical applications, the emphasis may often be placed more on the

surface errors surrounding the airfoil. For a more comprehensive comparison, we plot the RMSE errors on surface in Table 4.13. It can be observed that across

Table 4.13: Results on the surface of Airfoil Flow problems

RMSE (1e-2)	Evalutation	
Test Sets	Surface Errors	Volume Errors
Flow Interpolation	1.20 ± 0.12	0.71 ± 0.03
Shape Interpolation	2.32 ± 0.31	1.58 ± 0.15
Out-of-Distribution	11.20 ± 2.48	6.22 ± 1.66

all three test sets, the surface errors marginally exceed the volume errors. Despite this, the model maintains a commendable level of performance, particularly for the first two test sets.

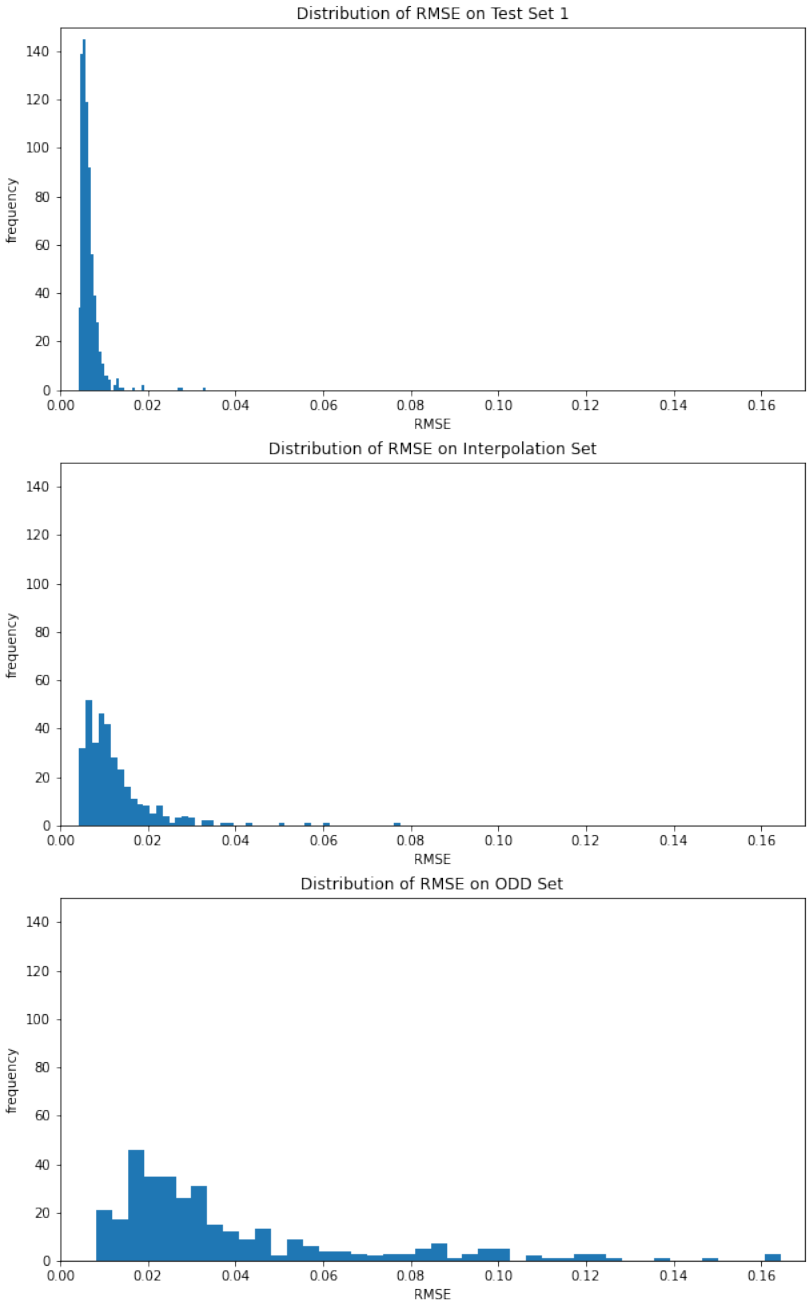


Figure 4.20: Comparison of RMSE distribution on three test sets

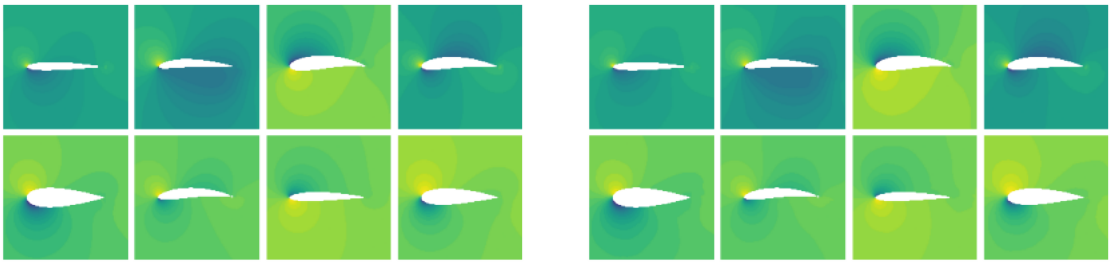


Figure 4.21: A set of predictions on flow field Pressure from Flow Interpolation Test Set.
Left:Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

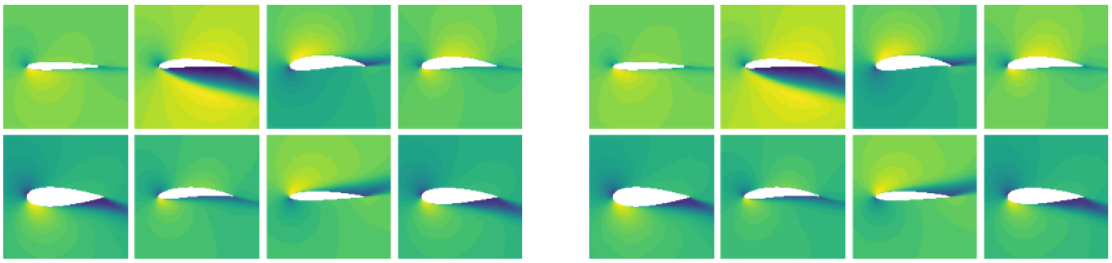


Figure 4.22: A set of predictions on flow field Velocity along axis x from Flow Interpolation Test Set.
Left: Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

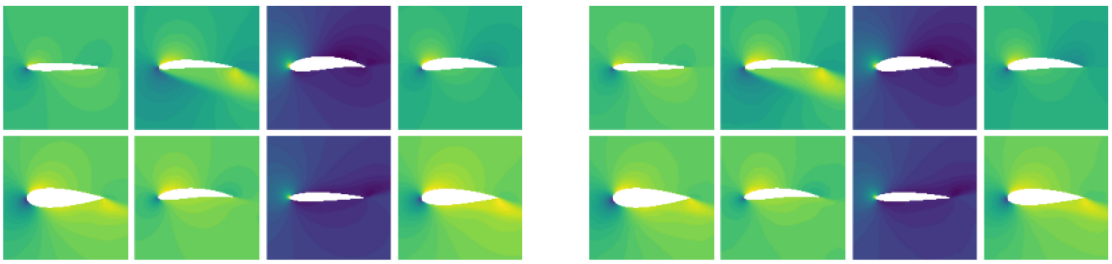


Figure 4.23: A set of predictions on flow field Velocity along axis y from Flow Interpolation Test Set.
Left: Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

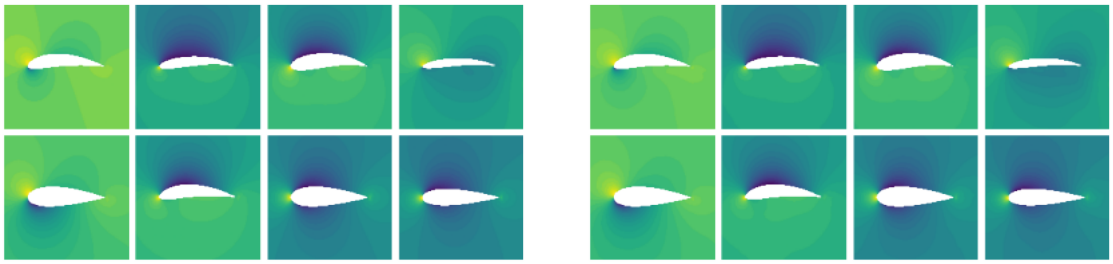


Figure 4.24: A set of predictions on flow field Pressure from **Shape Interpolation Test Set**. Left: Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

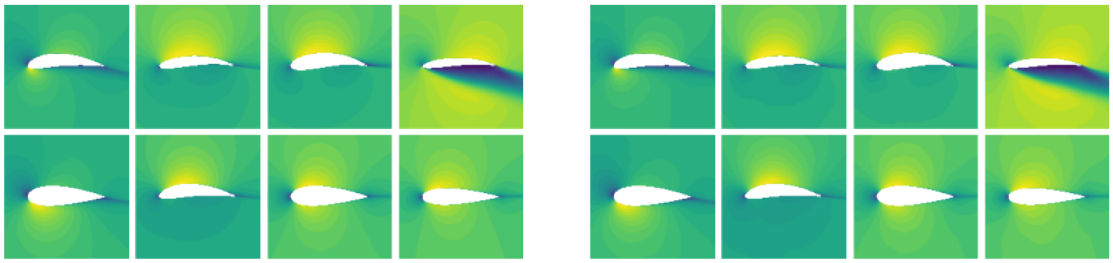


Figure 4.25: A set of predictions on flow field Velocity along axis x from **Shape Interpolation Test Set**. Left: Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

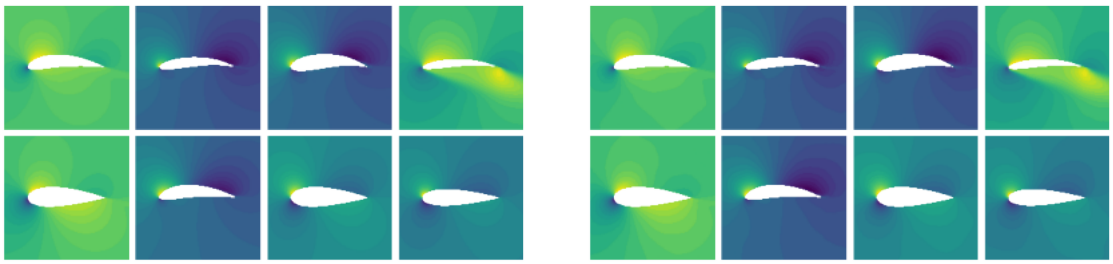


Figure 4.26: A set of predictions on flow field Velocity along axis y from **Shape Interpolation Test Set**. Left: Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

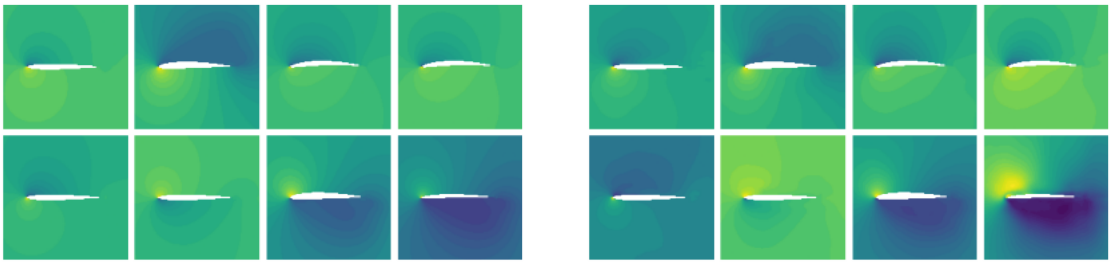


Figure 4.27: A set of predictions on flow field Pressure from **Out-of-Distribution Test Set**. Left: Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

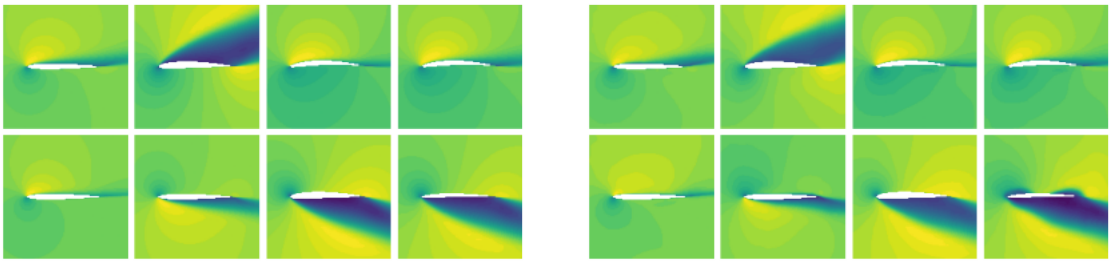


Figure 4.28: A set of predictions on flow field Velocity along axis x from **Out-of-Distribution Test Set**. Left: Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

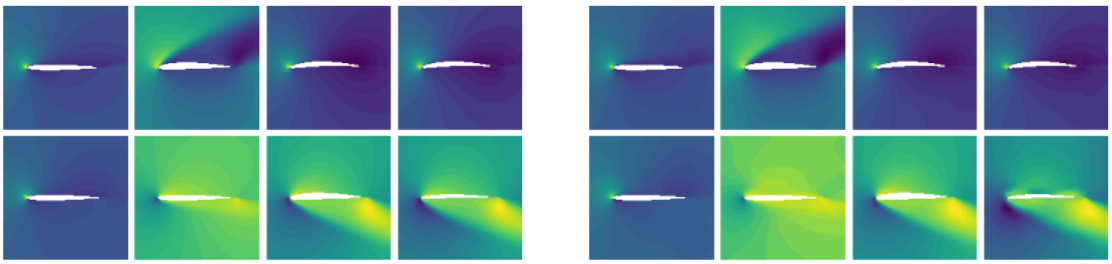


Figure 4.29: A set of predictions on flow field Velocity along axis y from **Out-of-Distribution Test Set**. Left: Simulation by OpenFOAM (Ground Truth), Right:Prediction from Graph U-Net

NACA0012 Validation Case

Even though studies on airfoil flow problems last over decades, the Navier-Stokes equations on an airfoil still remain challenging to solve. Simulations from CFD solvers do not ensure an accurate prediction of flow fields. We will hence perform a validation from experimental data on the NACA0012 profile, for which high-quality experimental results are accessible: Ladson tripped data [144] measured by NASA, is publically available. This data set is considered to be the most appropriate for comparison with fully turbulent CFD forces at a *Reynolds* number of 6 million [145], a dimensionless number comparing how much the fluid is moving around (inertial forces) to how much the fluid is being slowed down by internal friction (viscous forces). In our case, a Reynolds number of 6 million corresponds a Mach number of 0.15 approximately. We will thus compare the CFD approximation from `OpenFOAM` and the prediction from our deep learning model to the real experimental data from Ladson. We consider two experimental cases where Mach $= 0.15$ and $AoA = 0^o$ or 10^o to show the high reliability of the deep learning model in predicting flow fields. In such conditions, the initial freestream along the two axes x and y, $v_0 = (51.4815, 0)$ (for the case of $AoA = 0^o$) $v_0 = (50.6994, 8.9397)$ (for the case of $AoA = 10^o$), are considered as the inputs of our deep learning model.

NACA0012 Airfoil is a member of NACA 4-digits airfoils with the max camber $C = 0$, the max camber position $P = 0$ and the thickness $T = 12\%$. The mathematical equation to describe the profile of NACA0012 is written as:

$$y = \pm 0.594689181 \cdot (0.298222773 \cdot \sqrt{x} - 0.127125232 \cdot x$$
$$-0.357907906 \cdot x^2 + 0.291984971 \cdot x^3 - 0.105174606 \cdot x^4)$$

Apparently, the NACA0012 case doesn't follow the distribution we used to generate training airfoil profiles. It can be considered as an OoD problem.

Pressure Coefficient C_p is a non-dimensional number. It describes the relative pressures in a flow field:

$$C_p = \frac{p - p_\infty}{\frac{1}{2}\rho_\infty U_\infty^2}$$

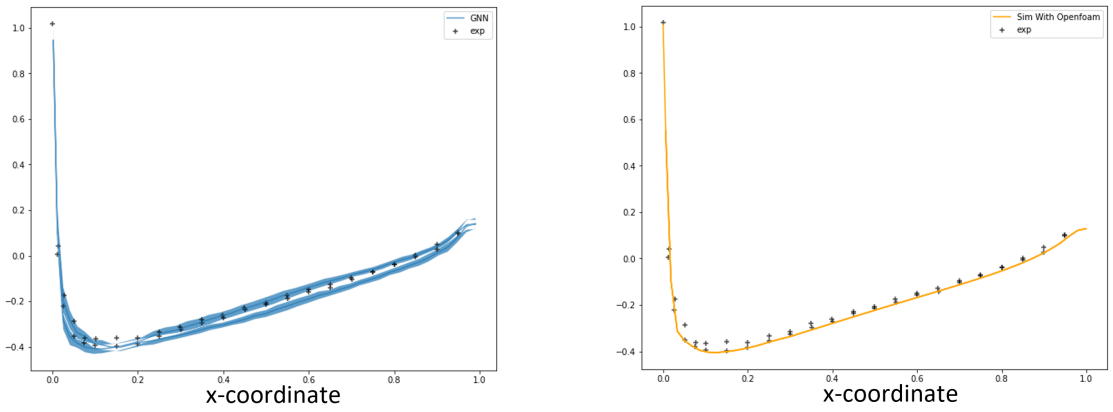


Figure 4.30: Pressure Coefficient on NACA0012 for $AoA = 0^o$

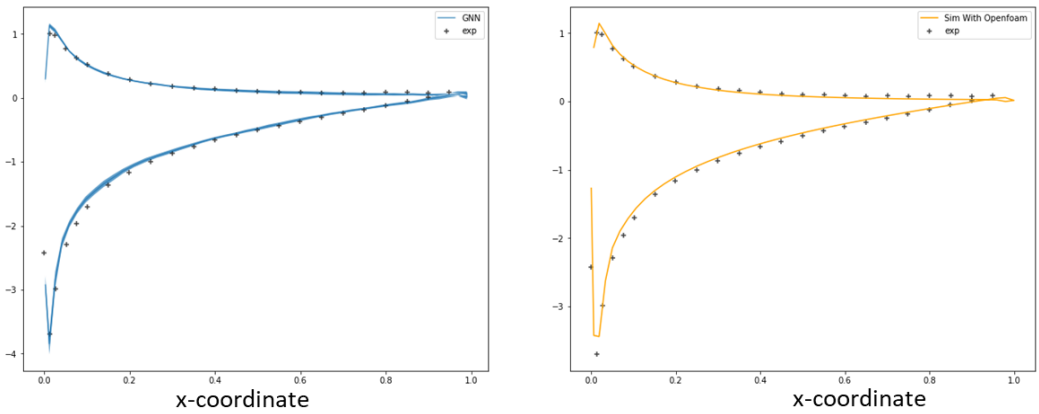


Figure 4.31: Pressure Coefficient on NACA0012 for $AoA = 10^o$

We calculate C_p at every point of the airfoil surface using the prediction of the pressure.

As shown in Figure 4.30 and 4.31, the prediction by our Graph model is very close to the measured experimental data. Deep learning is capable of predicting the flow field with high accuracy.

4.5 Computational Cost

In our experiments, we use a single GPU Nvidia A100 to compute the inference computational cost of the graph-based model, and compare it to that of numerical solvers on Intel(R) Xeon(R) Silver 4108 CPU. Note that this does not take into account the learning cost.

Table 4.14: Time computation Comparison

Time (s)	Donut			
	Senario 1	Senario 2	Polygon	Airfoil Flow
Graph U-Net	0.77	0.78	0.88	0.28
Graph MGMI	0.74	0.75	0.77	\
Numerical Solver	59.15	61.45	47.1	1542

Non-linear Poisson Equations We use a batch size of 100 to solve 1 000 unknown PDEs with our graph models on the three scenarios and compare them to that of *FEniCS* solver. For problems on fixed mesh, the computation time of neural networks is about 80 times faster than *FEniCS*.
When considering problems on variable polygon domains, the sampling operators slow down the graph-based approaches, making the computation time about 50 times faster than *FEniCS*. Also, Graph MGMI has fewer down-sampling operators than Graph U-Net, and hence allows slightly faster prediction.

Airfoil Flow problems are much more costly than the nonlinear Poisson's equation. To calculate the time computation of the traditional CFD solver `OpenFOAM`, we fix the solver relative tolerance as 1E-4. The solver will stop when the ratio of current to initial residuals falls below the solver's relative tolerance. We report the computation cost of solving a batch of 80 new problems.
The experimental results in Table 4.14 show that the Graph U-Net model is much more efficient than the CFD solver `OpenFOAM`. When solving complex physical systems, deep learning models have tremendous advantages in time consumption of the inference part.

Finally, note that this work studied a simple problem, though nonlinear, for which FEM solvers are relatively fast. The advantage of graph-based inference for complex PDEs (e.g., 3D CFD) would be even more significant.

4.6 Graph Neural Networks for PDEs: Conclusions

The present Chapter introduced multi-resolution graph-based approaches to learning PDE solutions on unstructured meshes, addressing the up- and down-sampling issues of GNNs spatially based on a hierarchy of meshes of increasing complexity. The models work with mesh-based simulations on various physical domains, including electrostatics and aerodynamics.

By bypassing the projection on a regular mesh and the use of standard CNNs, these approaches avoid the resulting interpolation errors. Furthermore, our experiments have shown that these hierarchical models improve the prediction accuracy on test sets compared to a simple GNN model that only uses the finest mesh. Most importantly, these models largely accelerate the computation time compared to the classical numerical solvers, especially for complex physical cases like Navier-Stokes CFD simulations.

Furthermore, whereas Out-of-Distribution generalization is satisfactory w.r.t. the source characteristics and the mesh complexity for electrostatics problems, in aerodynamics, further work is needed to decrease the dependency w.r.t. the airfoil profile outside the strict bounds of the training distribution. This is the subject of the next Chapter, based on recent advances in Meta-Learning.

5

Meta Learning Algorithms for Airfoil Flow Simulation

Contents

As demonstrated in the previous Chapter, neural networks are capable of accurately simulating complex physical systems and can reduce computational costs when compared to traditional numerical solvers. Through the use of deep learning methods, these models are able to learn patterns directly from data, allowing for more efficient and effective solutions to PDEs. However, a major limitation of these approaches is their tendency to suffer from poor generalization performance on out-of-distribution (OoD) samples, as the underlying physical laws are not explicitly incorporated into the learning process.

In order to address this issue, a number of researchers have proposed the use of hybrid models that combine deep learning with computational fluid dynamics (CFD) solvers. [32] takes advantage of the approximated solutions from CFD solvers on coarse meshes, using them as input features for deep learning models to make super-resolution predictions in aerodynamics. [146] uses machine learning to correct errors in cheap simulations on coarse meshes from traditional solvers. These models obtain extra information from the coarse-grained simulations to reduce generalization errors. While these hybrid models are able to improve the accuracy of predictions on OoD samples, they are often less efficient than pure machine learning methods.

In particular, we will focus on the application of these methods to airfoil flow problems, where our previous work has shown that deep learning models are effective at solving problems involving airfoil shapes from the training set, but may struggle with new and significantly different airfoil configurations.

5.1 Generalization issues on OoD data

In Section 4.2.2, we evaluated the performance of a deep learning model on a set of NACA airfoils and found that it struggled to generalize to OoD samples. Specifically, we trained the model on a dataset of 64 NACA airfoils and tested it on three separate test sets: A set of additional samples derived from the training airfoils, a set of interpolated airfoils generated using the same distribution of NACA parameters, and a set of thinner airfoils that significantly differ from the training distribution. While the model performed well on the first two test sets, it demonstrated poor performance on the OoD samples.

One strategy to consider in order to improve the accuracy of the model when making predictions on an OoD airfoil is to use transfer learning. If we have access to a small dataset $\mathcal{D}^{new}$ of pairwise data from the OoD airfoil, we can leverage the knowledge learned by the pre-trained model and fine-tune it using this new data. This can help to speed up training process and prevent overfitting, as the pre-trained model serves as an initialization for the fine-tuning process. By applying a few steps of gradient descent on the new dataset, we can potentially improve the model performance can be potentially improved on the OoD airfoil.

Algorithm 12 Transfer Learning Algorithms to solve OoD problems

Require: A pre-trained model f_θ
Require: New out-of-distribution airfoils we wish to deal with $\{\mathcal{A}_1, ..., \mathcal{A}_n\}$.
1: **for** every OoD airfoil $\mathcal{A}_i$ **do**
2: Sample a few data on $\mathcal{A}_i$ by changing physical quantities to create $\mathcal{D}_i$
3: Set step size parameters α_i
4: Evaluating a few steps of gradient descent on θ with D_i
5: Compute fine-tuned parameters: $\phi_i = \theta - \alpha_i \nabla_\theta \mathcal{L}(\theta, D_i)$
6: **end for**
7: The finetuned model f_{ϕ_i} will be used to deal with problems defined on $\mathcal{A}_i$

While fine-tuning can be effective in many cases, it may not be as effective when we have limited data available for the new task. This has been observed in studies of fine-tuning language models for text classification [101] (in Figure 5.1), where the performance of the fine-tuned model was found to be significantly better than a model trained from scratch but was less effective when the dataset

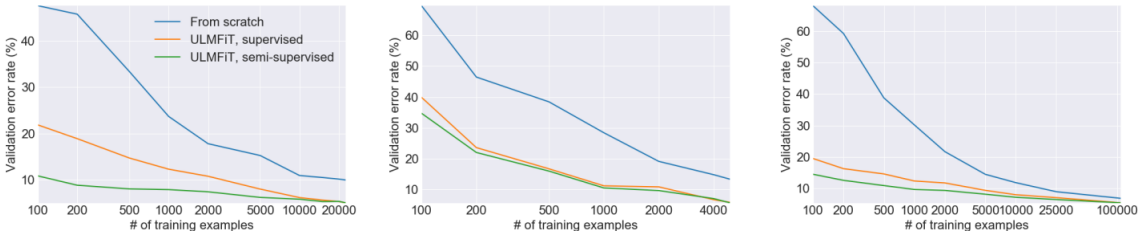


Figure 5.1: Validation error rates for supervised and semi-supervised transfer model ULMFiT vs. training from scratch with different numbers of training examples on datasets IMDb, TREC-6, and AG (from left to right).

contained fewer than 100 examples The process of collecting data for fine-tuning a model on OoD airfoils can be resource-intensive, as it involves running numerical simulations more than 100 times to generate new problem instances. This can be a significant burden, especially when the model needs to be fine-tuned for multiple OoD airfoils. To address this issue, rather than transfer learning, we propose to use meta-learning to train a meta-learner capable of adapting to new tasks or domains using a small amount of data. Meta-learning, or learning to learn, presented in Section 3.6, involves training a model on a (meta-)distribution of tasks from the same meta-distribution such that it can quickly adapt to new tasks by leveraging its past experience. In the following section, we will discuss how meta-learning can be applied to improve the performance of a data-based approach on OoD airfoils.

5.2 A meta-learning perspective

Introduced in Section 3.6, meta-learning provides a way to gain meta-knowledge over various tasks (meta-knowledge) and to use this knowledge to learn a new task using few task-specific data.

We formulate the airflow problem over various airfoils as a meta-learning problem, where each set of examples defined on a single airfoil shape is treated as a separate task. Our goal is to learn a meta-learner that is able to adapt to new tasks, i.e., to unseen airfoil shapes, using only a small amount of task-specific data (results of simulation on that shape). To achieve this, we propose to use the model-agnostic meta-learning (MAML) (see Section 3.6.3) to learn a meta-learner that can then be easily fine-tuned.

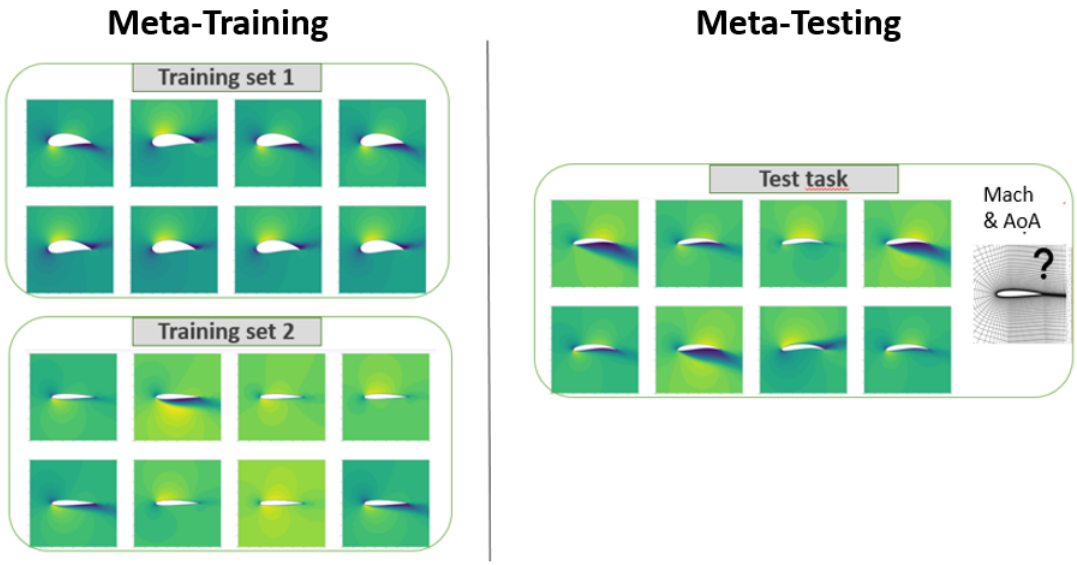


Figure 5.2: Airfoil Dataset for MAML algorithms: examples defined on the same airfoil are considered as a single task.

Recall that MAML is an optimization-based algorithm in meta-learning that find the initial parameters to enable quick adaptation to new tasks with small amounts of data.

In airfoil flow problems (see Figure 5.3), MAML++[1] is used to learn an incomplete model f_{θ_0}, called a *meta-learner*, that need to be fine-tuned with a

[1]the improved version of MAML, see Section 3.6.4, that we will abusively name MAML in the following

few steps of gradient descent, for any task, both from the meta-training set and the meta-test set. During meta-test time, new tasks from the meta-test set, with small amounts of data points, are treated similarly: the meta-leraner is updated with a few steps of gradient descent on the task-specific examples. The updated model f_{θ_i} is then employed to predict further unseen exampes of this specific task.

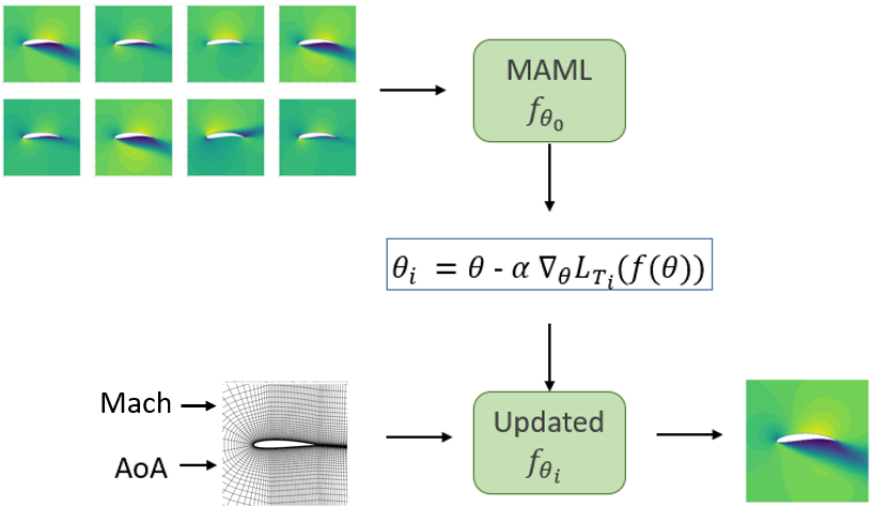


Figure 5.3: Prediction diagram of MAML algorithm for new tasks with a step of gradient descent

5.3 Experiments

In this Section, we revisit the airfoil simulation problems discussed in Section 4.4 and formulate it as a meta-learning problem.

5.3.1 Dataset and Baseline

As said, one task here corresponds to one airfoil shape. The Meta-Dataset is the same as in Section 4.4: It is made of 80 tasks (the same NACA airfoils) with 40 examples of various AoA and Mach numbers. As before, this dataset is split into three parts:

- **Meta-Training set**: 64 different tasks (NACA airfoils) with 30 examples for each airfoil.
- **Meta-Validation set 1**: the same 64 tasks (NACA airfoils) containing 10 new samples for each airfoil, used to measure the performance of the trained model on the airfoil already seen.
- **Meta-Validation set 2**: 16 new tasks (NACA airfoils) with 40 examples for each airfoil, the first 20 examples are used to update the initial model, and the other 20 are used to evaluate the model performance within MAML.

MAML results will be compared with those of the baseline presented in Section 4.4, where we consider the dataset as one single task and train a global model. For a fair comparison, both the baseline and MAML model use the same architecture, described in Section 4.4.

5.3.2 Hyper-parameter tuning

As for the baseline model (Section 4.4), we chose the hyper-parameters of model structure using 5-cross validation folds. Here we focus on the impact of the inner learning rate of MAML.

During meta-training, the meta-learner is updated using Adam optimizer with an initial learning rate of 5E-4 and step decay by a factor of 0.5 every 500 epochs. The number of inner gradient update steps is set to 3, and the first

20 examples of each task $\mathcal{T}_i$ are considered as the train set $\mathcal{D}_i^{tr}$ used for inner gradient updates, while the remaining examples are used as $\mathcal{D}_i^{test}$ to compute the meta-loss $\mathcal{L}$ for meta-update.

Different from the baseline model, MAML requires a few examples to update the meta-learner once given a new task. To treat problems with airfoils already seen in the training set, we can use examples from the training set to update the model. While for new tasks (in the Meta-Validation set 2), new examples (with the corresponding solutions) must be generated to update the model. First, validation errors are computed by using 3 steps of gradient updates, the same number in Meta-training. Moreover, the authors of the original paper of MAML have noted that using more gradient steps during Meta-testing may continuously improve the performance of an updated model without overfitting. To further investigate the impact of the inner learning rate, we conduct additional experiments using 10 steps to update the model. This will allow us to thoroughly examine the influence of the inner learning rate on the performance of the MAML model. As shown in Table 5.1, the experimental results indicate that increasing the

Table 5.1: RMSE error on both Validation sets of different inner learning rate

RMSE(1e-2)	3 Inner Steps		10 Inner Steps	
Inner Lr	Val. 1	Val. 2	Val. 1	Val.2
Lr=0.01	0.82 ± 0.05	1.33 ± 0.23	$\mathbf{0.80 \pm 0.04}$	$\mathbf{1.20 \pm 0.17}$
Lr=0.001	0.83 ± 0.06	1.32 ± 0.16	0.82 ± 0.05	1.29 ± 0.14

number of update steps can lead to improved performance of the MAML model. Moreover, the choice of inner learning rate does not appear to have a significant influence on the performance of the MAML model when using 3 steps to update the model. When using 10 update steps, we find that the model performs best when using an inner learning rate of 0.01. The inner learning rate will be fixed as 0.01 for the following experiments.

5.3.3 Results

After using 5-fold cross-validation to choose the proper hyper-parameters, we evaluate the models with different test sets. To better compare the baseline model and MAML, we apply 10-fold cross-validation folds.

Three meta-test sets will allow us to evaluate the generalization of the models with respect to airfoil shapes. For the flow parameters AoA and Mach number, we use the same distribution as in training set in order to more accurately highlight the performance of the models on OoD tasks.

- **Flow Interpolation Test Set**: We generate 10 more examples for each meta-training task (airfoil shape). For evaluation, we can directly use the training data to update each task and make further predictions for new examples.
- **Shape Interpolation set**: This set contains 20 new airfoils that are generated using the same distributions of NACA parameters as the training set. Each airfoil is treated as a separate meta-test task, with 50 examples in total. We use the first 20 examples to update the MAML model and the remaining 30 examples to evaluate the model performance.
- **Out of Distribution set** – Thinner Airfoils: By changing the range of distribution on T and P, we can create airfoils that are thinner and less regular ($P \in [20, 80]$ and $T \in [5, 10]$). These test tasks are more challenging than those in the training set, with each task consisting of 50 different examples.

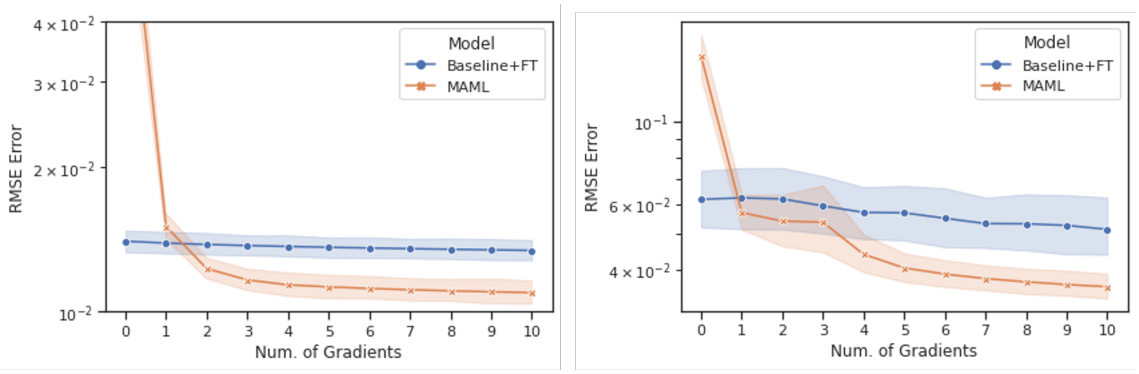
Our meta-model is evaluated after fine-tuning on each task and compared with the baseline model on the three meta-test sets described above. MAML models require additional data every time a new task is presented. In contrast, the baseline model is applied directly to unseen airfoils. However, such extra data (the few labeled examples available for the new tasks/airfoils) can help not only MAML but maybe also the baseline model: To be fair to the baseline, the same number of additional gradient descent steps are used to finetune the baseline model on each new tasks before evaluation, in the same way as for MAML: this is the column labeled "Baseline + FT" in Table 5.2. We use RMSE as the metric

Table 5.2: Evaluation Results on different test sets with 10 gradient updates

	RMSE(1e-2)		
Test Sets	MAML	Baseline	Baseline + FT
Flow Interpolation	0.79 ± 0.08	**0.71 ± 0.03**	0.71 ± 0.03
Shape Interpolation	**1.18 ± 0.14**	1.58 ± 0.15	1.48 ± 0.14
Out-of-Distribution	**3.69 ± 0.28**	6.22 ± 1.66	5.35±1.62

for model performance.

Moreover, for all pair-wise comparisons between test errors on MAML and Baseline, we performed a Wilcoxon signed-rank statistical test with 95% confidence, and the differences between all pairs are statistically significant. From the results in Table 5.2, it turns out the MAML model outperforms the baseline on the interpolation and OoD test sets. By using just a few data points and gradient steps, MAML models can quickly adapt to new tasks without overfitting. Compared to the baseline, the MAML model shows much better performance on OoD Meta-test tasks, less similar to the meta-training tasks. Interestingly, adding some fine-tuning to the baseline does improve its performance, but the results remain far below those of the MAML approach.



(a) Shape Interpolation (b) Out-of-Distribution

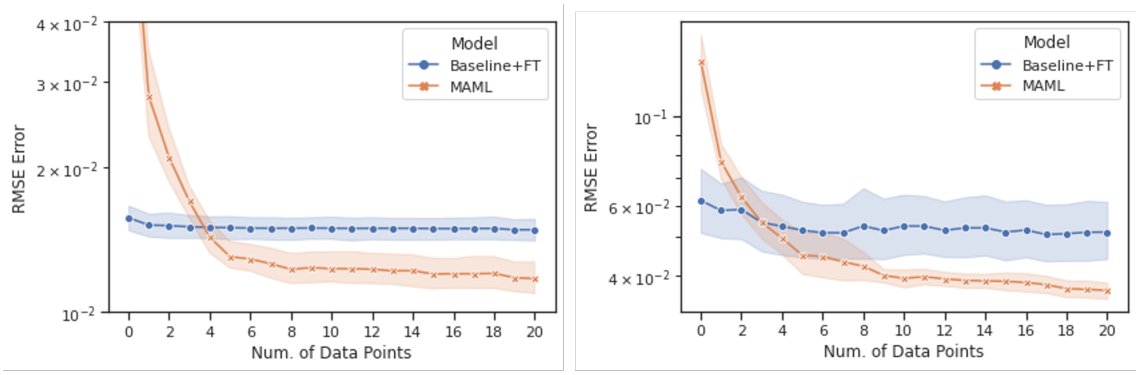
Figure 5.4: Sensitivity w.r.t. the number of gradients on Shape Interpolation Test Set (a) and Out-of-Distribution Test set (b). The MAML model is improved a lot with extra gradients and continues to improve. While the baseline with finetune doesn't have significant improvement

Sensitivity w.r.t. the number of gradient updates Figure 5.4 shows that

with small amounts of data, the model learned with MAML is able to adapt quickly to new tasks and continues to improve without overfitting. Again, we can see that fine-tuning indeed improves the performance of the baseline, but not as much as the model trained with MAML.

Discussion on number of Examples As previously mentioned, during the meta-training phase, 20 examples were used for inner gradient updates. At the meta-test time, the same number of examples is used to update the meta-learner for test tasks in the first experiment.

We now investigate the impact of the number of points used for test tasks on the performance of the MAML model. Specifically, the MAML model is fine-tuned using 10, 5, and 1 examples, each with 10 gradient updates, in order to understand how the number of examples used for test tasks affects the model performance. The results shown in Table 4 indicate that decreasing the number



(a) Shape Interpolation (b) Out-of-Distribution

Figure 5.5: RMSE w.r.t. number of examples on Shape Interpolation Test Set (a) and Out-of-Distribution Test set (b)

.

of examples used to update the meta-learner has a negative impact on the model performance. While the MAML model still performs well on new tasks and outperforms the baseline model when using smaller numbers of examples. From Figure 5.5 it is unable to make reasonable predictions on new tasks (i.e., at least as good as the baseline) on new meta-tasks with using less than 4 examples: at least 4 examples are necessary for the Shape Interpolation set, while 3 are sufficient on the OoD set, where the baseline is far less efficient. This suggests

Table 5.3: Results using a different number of examples to update the model

RMSE(1e-2)		Num. Examples			
	Models	10	5	1	Before Update
Shape Int.	MAML	1.21 ± 0.14	1.33 ± 0.10	3.91 ± 1.25	10.34 ± 3.60
	Baseline+FT	1.49±0.12	1.50±0.13	1.52±0.13	1.58±0.15
OoD	MAML	4.00 ± 0.30	4.19 ± 0.33	6.70 ± 0.92	14.44 ± 3.49
	Baseline+FT	5.44±1.39	5.51±1.72	6.07±1.48	6.43±1.79

that while the MAML model is able to adapt to new tasks using only a small number of examples, it nevertheless requires more than one example in order to perform well on these tasks: one example is, in particular, insufficient to reach the performance of the baseline.

5.4 Meta-Learning for Air Flow Simulation: Conclusions

In this Chapter, we have presented a meta-learning approach to address airfoil flow problems. By utilizing the Model-Agnostic Meta-Learning (MAML) algorithm, we have trained a meta-learner that is capable of adapting to new tasks with only a small number of examples.

Our experimental results show that the meta-learner consistently outperforms the baseline model, which is trained classically, once and for all using the whole dataset, in terms of its ability to generalize to out-of-distribution (OoD) airfoil shapes – and this, even if this baseline model is fine-tuned with the same small amount of data for the new tasks than the MAML meta-learner. Overall, we claim that the meta-learning approach represents a promising solution for addressing the challenges of generalization in airfoil flow problems.

6

Multi-Fidelity Transfer Learning: from Coarse to Fine

Contents

Experimental results presented in previous Chapters have demonstrated the ability of data-based models to solve PDEs with low computational costs at inference time, however using at training time large datasets (even though this assumption can be mitigated by adopting some Meta-Learning approach whenever possible, see Chapter 5). The dataset is gathered using classical numerical solvers such as the finite element method (FEM) or the finite volume method (FVM). As discussed in Section 2.1, these methods discretize the physical domain into small elements, somehow project the PDE on each element in turn, and solve the resulting system of discrete equations. The accuracy of the FEM/FVM

solution depends on the size of the mesh, i.e., the number of elements. Generally, fine meshes (i.e., with a large number of elements) are needed to reach an acceptable accuracy, and one important challenge in the data-based approach is the computational cost of generating such large training datasets. This is particularly true for complex phenomena that require large deep networks, which in turn require large training datasets. Additionally, the total error of the trained model is the sum of the training error of the network and the numerical error of the samples in the training set, which should be accurate enough solutions, hence obtained using very fine meshes.

To alleviate this challenge, we propose MFT, a Multi-Fidelity Transfer learning approach that uses two meshes of different granularity: On the coarse mesh, FEM/FVM approximate solutions of the PDE at hand can be computed at low cost – but only poorly approximating the solution of the PDE. On the fine mesh, on the other hand, the FEM/FVM solutions are good approximations of the solution of the PDE, but are very costly to compute, making it unrealistic to generate enough samples to train an accurate-enough deep model. The goal of the work presented in this Chapter is to train such an accurate-enough deep model using only a small dataset of highly accurate solutions, but leveraging the result of the training on the many samples computed on the coarse mesh, in combination with principles from Transfer Learning: the model trained on the large dataset generated on the coarse mesh is used as a starting point for the learning process on the small dataset of accurate solutions.

6.1 Multi-fidelity Transfer Learning

In the following, we aim at learning a data-based model for numerically solving a given PDE associated with various input quantities $\boldsymbol{x}$ that govern the system on a given domain. To do so, we utilize a FEM/FVM numerical solver on two different meshes of the same domain, a coarse mesh $\mathbf{M}^c$, and a fine mesh $\mathbf{M}^f$. The solver thus computes so-called low-fidelity solutions at a low cost on the coarse mesh $\mathbf{M}^c$ and high-fidelity solutions at a higher cost on the fine mesh $\mathbf{M}^f$. Our goal is to train a deep neural network model that can predict accurate, high-fidelity solutions for any given input using as few high-fidelity samples as possible but taking advantage of as many low-fidelity samples as needed.

6.1.1 Prediction on the Coarse Mesh

The first step involves generating a dataset $\mathbf{D}^c$ of low-quality solutions by applying the FEM/FVM solver to the coarse mesh $\mathbf{M}^c$ using a representative set of input values $\boldsymbol{x}$. A deep model f_c is then trained on $\mathbf{D}^c$ (details below). As many samples as necessary can be computed efficiently and at a low cost, and the loss function is the mean squared error between the network output and the numerical solution in $\mathbf{D}^c$, which is treated as the ground truth at this stage. Even in the ideal case where the error of the trained network is close to zero, thanks to a very large dataset $\mathbf{D}^c$, the accuracy of the learned model is not satisfactory, as the solutions in $\mathbf{D}^c$ are poor approximations of accurate solutions.

6.1.2 Transfer of low-fidelity knowledge on the Fine Mesh

The second step of the process begins with the low-fidelity model f_c, which has been fully trained on the low-fidelity dataset $\mathbf{D}^c$. It also involves the dataset $\mathbf{D}^f$ of high-fidelity solutions. However, $\mathbf{D}^f$ is assumed to be too small to allow for direct training of a sufficiently accurate deep learning model. Drawing inspiration from the concept of Transfer Learning (detailed in Section 3.5), which utilizes the knowledge acquired while solving a task to improve the learning of a related but distinct task, we propose to leverage the knowledge contained within f_c to enhance the learning of a deep model using the small dataset $\mathbf{D}^f$.

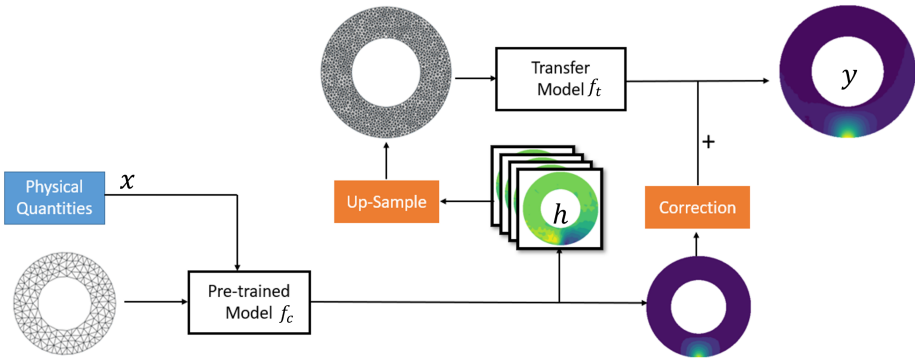


Figure 6.1: Diagram of the multi-fidelity transfer learning approach on Wheel Contact problem. Note that the network is asked to predict the correction w.r.t. the projection of the prediction by the low-fidelity model f_c rather than directly the deformation y.

Latent Feature Extraction The basic assumption here is that the pre-trained model f_c has learned a meaningful representation of the problem and can be viewed as a feature extractor for the high-fidelity task. For each input $\boldsymbol{x}$, the vector $\boldsymbol{h}$, outputs of all neurons before the last layer of f_c, can be viewed as a vector of latent features of the problem. It is first upsampled to the fine mesh $\mathbf{M}^f$ using k-nearest neighbors interpolation (see Section 4.1.2), and then used as additional inputs for the high-fidelity learning, as illustrated in Fig. 6.1.

Transfering Using the high-fidelity dataset $\mathbf{D}^f$ and additional input $\boldsymbol{h}$, we train a transfer model f_t using again the MSE loss. The final output is:

$$y = f_t(\boldsymbol{x}, \boldsymbol{h}) = f_t(\boldsymbol{x}, \text{UpSample}[f_c(\boldsymbol{x})])$$

6.2 Case Studies

The **MFT** approach is experimentally validated on two different PDEs, describing respectively the flow around an airfoil and the contact of a tire on the road.

Airfoil Flow: This problem, described in detail in Section 4.4, is that of the incompressible fluid flow around an airfoil, modeled by the Reynolds Averaged Navier Stokes equations. Flow fields on 80 distinct 2D NACA airfoils are simulated by the CFD solver `OpenFOAM`. Inputs are the Mach number $\in$ [0.03, 0.3], and the Angle-of-Attack (AoA) $\in$ [-22.5, 22.5] (in degrees). The target variable are the velocity $\mathbf{v} = (v_x, v_y)$ and the pressure p. The low-fidelity dataset is generated with meshes of approx. 500 nodes and the high-fidelity dataset uses meshes with around 3800 nodes.

Wheel Contact: As a specific use case of Michelin, the goal is to predict the deformation field $\mathbf{u} = (u_x, u_y)$ of a tire on a fixed 2D wheel under some external force $\mathbf{f}$ describing the action of the road on the tire while the car is driving. The FEM solver used is `GetFEM++` [147]. The input quantities are the mechanical properties of the rubber used to make the tire: Young's modulus $E \in [5, 7] \times 10^6$ and Poisson's ratio $\nu \in [0.38, 0.495]$; the magnitude $A \in [0.1, 0.5]$ and angle $\alpha \in [-0.78, -2.35]$ of the external forces; and the friction coefficient of the road $\mu \in [0.5, 0.8]$. The 2D linear plane strain elasticity equation is used to model the problem, with the following simplifying hypotheses:

1. The displacements of the material particles are much smaller than any relevant dimension of the tire.

2. The inner rim of the wheel is considered rigid.

3. The floor in contact with the wheel is flat and rigid.

In this context, the formulation of the wheel contact problem is as follows: $\boldsymbol{\nu}$ represents the normal vector of the boundary Γ, u and $\boldsymbol{\Pi}$ denote the Piola-Kirchoff displacement and stress fields, respectively. Moreover, we denote by u_ν and u_τ the normal and tangential components, $\boldsymbol{\Pi}_\nu$ and $\boldsymbol{\Pi}_\tau$ the normal and tangential stresses on Γ. $W(\mathbf{F})$ is a function representing hyperelastic internal

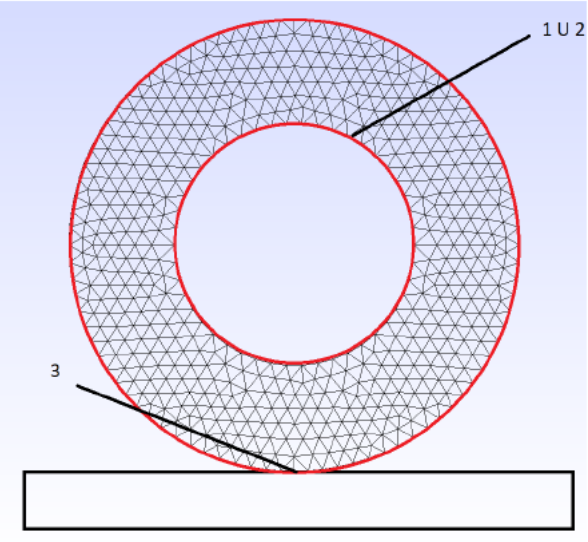


Figure 6.2: Physical Domain for Wheel Contact Problem

energy density depending on $\mathbf{F}$, the deformation gradient. f_0 and f_2 denote a volumetric force density and a surface traction force density, respectively. Finally, g represents the normal distance between the wheel and the foundation, and μ is the friction coefficient of the road.

$$\mathbf{\Pi} = \partial_{\mathbf{F}} W(\mathbf{F}) \quad \text{in} \quad \Omega, \tag{6.1}$$

$$\operatorname{Div} \mathbf{\Pi} + \boldsymbol{f}_0 = \mathbf{0} \text{ in } \Omega, \tag{6.2}$$

$$\boldsymbol{u} = \boldsymbol{u}_d \quad \text{on} \quad \Gamma_1, \tag{6.3}$$

$$\mathbf{\Pi}\boldsymbol{\nu} = \boldsymbol{f}_2 \quad \text{on} \quad \Gamma_2, \tag{6.4}$$

$$u_\nu \leq g, \quad \Pi_\nu \leq 0, \quad (u_\nu - g)\,\Pi_\nu = 0 \quad \text{on} \quad \Gamma_3, \tag{6.5}$$

$$\begin{cases} \|\Pi_\tau\| \leq \mu\,|\Pi_\nu|\,, \\ -\Pi_\tau = \mu\,|\Pi_\nu|\,\frac{u_\tau}{\|u_\tau\|} \text{ if } u_\tau \neq 0. \end{cases} \quad \text{on} \quad \Gamma_3\,. \tag{6.6}$$

Equation (6.1) represents the hyperelastic constitutive law of the material. Equation (6.2) is the equilibrium equation. The two conditions (6.3) and (6.4) represent the displacement and boundary conditions, respectively. Finally, (6.5) and (6.6) describe the rubbing contact condition.
The low-fidelity dataset is generated on a fixed coarse mesh with 504 nodes, while the high-fidelity dataset uses a fixed mesh with 3398 nodes.

6.3 Experiments

In the following, we evaluate the MFT approach on the two use cases introduced in previous Section 6.2.

Datasets For each domain, we create two meshes of different scales: a coarse mesh $\mathbf{M}^c$ and a fine mesh $\mathbf{M}^f$. The low-fidelity dataset $\mathcal{D}^c$ consists of 2,000 samples generated on $\mathbf{M}^c$, while the high-fidelity dataset $\mathcal{D}^f$ on $\mathbf{M}^f$ consists of 400 samples. While it would have been ideal for experiments with various numbers of samples, including 400, 300, 200, and 100, due to the time constraints of the Ph.D. research, it was not feasible to conduct other experiments.

Baselines We compare MFT with three baselines. An interpolation model **LF-Int** predicts the outputs by simply up-sampling the output of model f_c. Model **HF-400** is directly trained on the 400 high-fidelity samples without any knowledge transfer. Finally, in order to be fair with the high-fidelity-only approach, we add some high-fidelity samples to $\mathcal{D}^f$ to compensate for the computational cost of creating $\mathcal{D}^c$. Model **HF-ST** is trained on this extended high-fidelity dataset.

Hyper-parameter setting The hyperparameters are tuned using the same process as Chapter 4. Models are trained to minimize the mean square error (MSE) between the output and ground truth. We utilize the Adam optimizer to minimize this loss function. Moreover, a step decay strategy is applied during training, where the learning rate is halved every 500 epochs. The GNN architecture is the graph U-Net described in Section 2.3, consisting of a series of graph blocks and sampling operators. A graph block C contains multiple graph layers, each of which is followed by an activation function "elu". Each block C is parametrized by the number of layers l, a channel factor c, and a kernel size k. After each GNN block, a sampling operator is applied to convert data between both mesh levels. The sampling operator has one hyper-parameter, and the number of nearest neighbors n is set again to 6 for all our experiments. Finally, a graph layer is applied to map high-dimensional features onto the solution space.
For the airfoil flow problem, in order to train the model f_c which predicts the solution on a coarse mesh $\mathbf{M}^c$, we down-sample once $\mathbf{M}^c$, and up-sample it back to $\mathbf{M}^c$. A total of three blocks $C = (4, 128, 5)$ are applied. The transfer model f_t

on high-resolution meshes down-samples progressively three times in the encoding part and recovers the mesh resolution with three up-sampling operators during decoding. f_t contains seven GNN blocks $C = (2, 48, 5)$.
For the wheel contact problem, the model f_c contains three mesh levels and 5 GNN blocks $C = (4, 48, 5)$. For the transfer model f_t, same as that on airfoil flow, there are seven GNN blocks $C = (2, 64, 5)$.

Training Both the coarse model f_c and the transfer model f_t use the Graph U-Net architecture described above. Standard 10-fold cross-validation is applied when training f_t to assess the robustness of the approach, and we report the averages and standard deviations of test errors over the different folds. Models are trained on a single Nvidia A100 GPU, and each training takes from 3 to 6 hours.

Evaluation We create 800 new samples on $\mathbf{M}^f$ for each task to form the ultimate test set and evaluate the results of all approaches using the rooted mean squared error (RMSE) metric.

Results Table 6.1 reminds the characteristics of the datasets (right side) and the error on the test set described above (left). The latter clearly demonstrates that the MFT model significantly outperforms the three baselines on both physical problems. The performance of the interpolation baseline LF-Int shows that the transfer model MFT largely improved the predictions based on the pre-trained model f_c. Meanwhile, the prior knowledge extracted from $\mathcal{D}^c$ does help the prediction on fine meshes compared with HF-400, which has only access to the high-fidelity dataset $\mathcal{D}^f$. Moreover, the comparison between HF-ST and MFT indicates that the multi-fidelity datasets containing many more low-resolution samples are still more informative than a pure high-fidelity dataset with the same computational budget. Furthermore, it seems more stable, as it displays a much lower variance. The MFT model combines the benefit of the low-fidelity dataset for fast generation with the high-fidelity simulations to create accurate ground truth.

Figure 6.3 and 6.4 show examples of the predictions made by the MFT model for the airfoil flow and Wheel Contact problems. The figures demonstrate

Table 6.1: Rooted mean squared error (RMSE) for both Airfoil Flow and Wheel Contact problem (right); dataset composition for each discussed model (left); and generation time for each sample in the last two rows.

	Datasets		Results	
Models	$\mathcal{D}^c$	$\mathcal{D}^f$	Airfoil Flow (e-2)	Wheel Contact (e-4)
LF-Int	2 000	\	9.81	27.82
HF-400	\	400	2.98 ± 0.30	6.62 ± 0.25
HF-ST	\	400 + 127/240	2.43 ± 0.39	4.80 ± 0.19 [1]
MFT	2 000	400	$\mathbf{1.95 \pm 0.05}$	$\mathbf{4.57 \pm 0.10}$
Generation	Airfoil Flow		$\sim$ 2.33s	$\sim$ 36.7s
Time (each)	Wheel Contact		$\sim$ 0.20s	$\sim$ 1.64s

the ability of the MFT model to effectively transfer knowledge from the low-fidelity dataset to the high-fidelity dataset, resulting in accurate predictions for both problems.

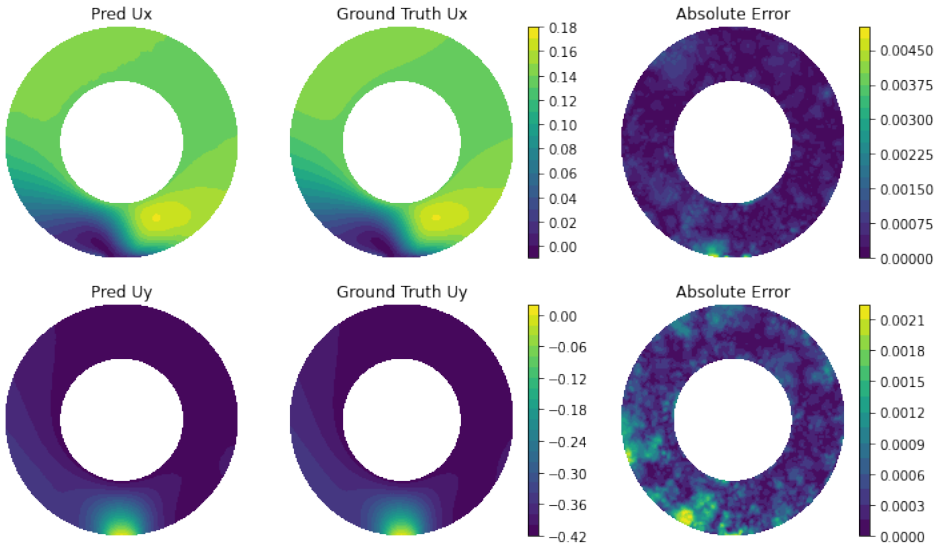


Figure 6.3: An example of wheel contact prediction

[1] We generated 127 additional high-fidelity examples on fine meshes for the Airfoil Flow problem, and 240 for Wheel Contact problem.

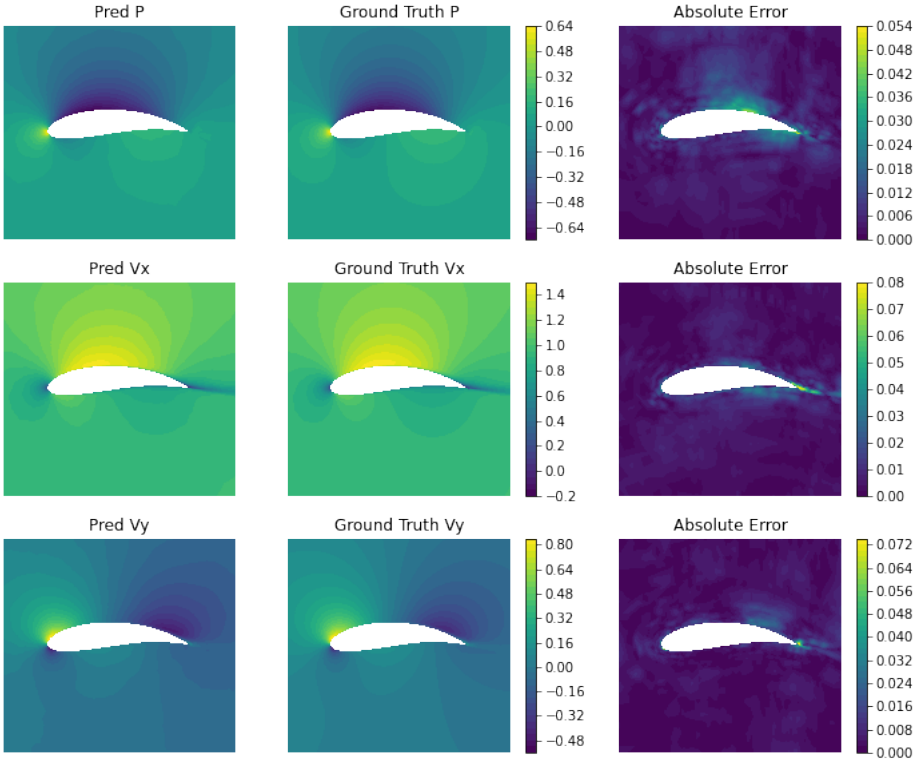


Figure 6.4: An example of airfoil flow prediction

Discussion on size of high-fidelity dataset

In this section, we will discuss the experiments with various numbers of samples for the high-fidelity dataset to analyze better the capacity of the proposed MFT approach. The low-fidelity dataset always has 2 000 examples, while the high-fidelity dataset contains 400, 320, 240, 160, and 80 examples separately.

Figure 6.5 presents a comparison of the performance of the proposed MFT model and the HF-ST model, each with varying sizes of high-fidelity datasets. (The HF-ST model is always trained with the extended high-fidelity only dataset to compensate for the computational cost of creating the low-fidelity dataset) The plot reveals that the HF-ST model is considerably more sensitive to the number of examples incorporated in the high-fidelity dataset. In contrast, the MFT model demonstrates a superior level of stability in relation to the impact of the number of high-fidelity dataset examples when compared to the HF-ST model. This outcome is quite reasonable, given that the MFT model can always leverage valuable information derived from the extensive low-fidelity dataset.

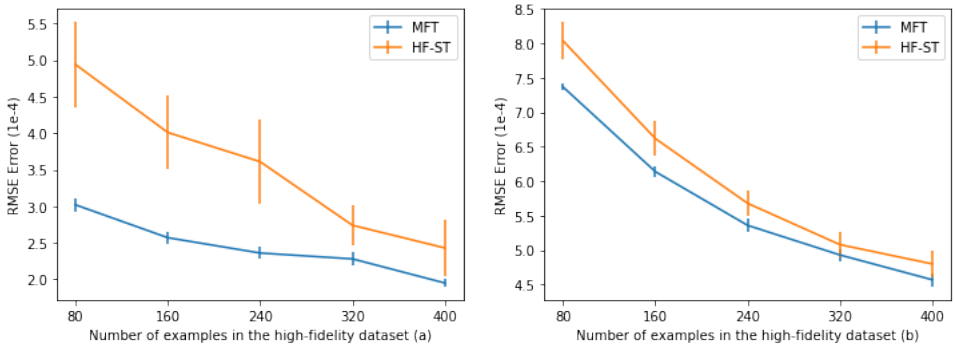


Figure 6.5: Errors comparison of the different number of high-fidelity examples. (a) airfoil flow problem; (b) wheel contact problem

6.4 Challenges on the Higher-Resolution Mesh

The proposed MFT approach is highly adaptable and can easily be applied to even higher-resolution meshes than above with additional high-fidelity datasets. In this Section, we demonstrate the ability of the MFT model to adapt to higher-resolution meshes by further refining the mesh and generating examples of even higher quality.

Same as in the previous experiments, we can transfer prior-knowledge learned to solve the higher-fidelity task. The transfer model learned from the previous experiments is considered as the new pre-trained model. In such a way, the new dataset on higher-resolution meshes with the two datasets $\mathcal{D}^h$ and $\mathcal{D}^f$ used in the previous experiment, from all three mesh-scales, contribute to training process, resulting in improved performance and accuracy.

Datasets Higher resolution meshes than past experiences are created to generate 160 new samples. The higher-fidelity dataset $\mathcal{D}^h$ uses meshes $\mathcal{M}^h$ with around 15000 nodes for the airfoil flow problem and a fixed mesh with 9015 nodes generated on the same domain for the wheel contact problem.

The baseline The **MFT** model will be fairly compared with the **HF-ST** model, the improved high-fidelity-only approach (see Section 6.3).

Evaluation We create 400 new samples on $\mathbf{M}^h$ for each problem to form the test set with high-quality and evaluate the results of all approaches using the rooted mean squared error (RMSE) metric.

Table 6.2: Rooted mean squared error (RMSE) for both Airfoil Flow and Wheel Contact problem (left); dataset size for each discussed model (right)

		Results	
$\mathcal{D}^h$	Models	Airfoil Flow (e-2)	Wheel Contact (e-4)
0	LF-Int	6.50	13.27
160 + 91/180	HF-ST	6.61 ± 0.22	26.2 ± 1.7
160	MFT	$\mathbf{5.90 \pm 0.10}$	$\mathbf{7.09 \pm 0.04}$

From Table 6.2, it can be observed that the results are consistent with the findings of previous section. Compared to table 6.1, the predictions on higher

resolution meshes become worse, which is due to the fact that the simulation from OpenFOAM becomes more accurate, leading to a large difference in $\mathcal{D}^f$ and $\mathcal{D}^h$. The results indicate that the proposed Multi-Fidelity Transfer (MFT) approach is able to effectively adapt to higher resolution meshes, once additional high-fidelity datasets are created.

6.5 Multi-Fidelity Transfer Learning: Conclusions

This Chapter introduced MFT, a multi-fidelity transfer learning model, to use Machine Learning approaches (namely GNNs) to numerically solve PDEs on fine meshes efficiently. MFT benefits from a large amount of low-fidelity data to extract some prior-knowledge, and then transfer it to predict high-fidelity solutions. The training process on fine meshes can then be achieved by using only a small training set. The experimental results on two complex physical problems are the first proof of the concept that MFT can solve PDEs accurately when a small number of high-fidelity samples is available.

7
Contributions and Further Work

We conclude this dissertation by summarizing our contributions and sketching a few research directions that have emerged from this work.

Contributions

It has become increasingly popular over the last few years to use deep neural networks to solve PDEs, as a way to accelerate computation time with data-driven methods or compensate for mesh-based solvers limitations when mesh decomposition becomes infeasible. However, such approaches still remain in their infancy. We have discussed throughout this thesis three common difficulties when using deep learning to approximate the solutions of PDEs: Model design to treat mesh data; Generalizing issues on OoD problems; And the cost of collecting data. Our main contribution was to develop some model architectures to start alleviating these hurdles.

We started by designing the first multi-resolution GNN approaches to learning the solutions of PDEs on unstructured meshes. On the one hand, graph neural networks can directly handle data living on unstructured meshes to reproduce the locality properties of CNNs. On the other hand, multi-resolution architectures extract hierarchical spatial features and accelerate message passing across the whole mesh.

Our second contribution was to adopt a meta-learning perspective in order to revisit deep learning scenarios on PDEs. The physical problems are decomposed into multiple tasks on the basis of their geometric domains. By porting the use of MAML, an optimization-based meta-learning approach, we enhanced model performance on Out-of-Distribution data points.

Last, in order to reduce the computational cost of data gathering, we turned to leverage low-quality data obtained on coarse meshes, which is almost costless, to help produce accurate solutions on fine meshes with very few solutions obtained directly on fine meshes. Transfer learning was thus introduced as a tool to solve tasks with insufficient high-accuracy training sets. By transferring knowledge from coarse meshes, transfer learning quickly solved problems on finer meshes using much less high-quality data than before.

Experimental results on various problem domains have shown the effectiveness of the proposed approaches: our proposed hierarchical GNN architectures avoid the resulting interpolation errors compared to CNN models, thus improving the quality of the predictions. The multi-resolution models based on a hierarchy of meshes largely outperform the baseline graph model without hierarchy. However, the experiences on model evaluation highlighted the generalization issue on OoD problems. We validated our Meta-learning approach on airfoil simulation tasks. The meta-learner demonstrated strong performance in cases outside the strict bounds of the training distribution. The two experiments on transfer learning showed that low-quality data on coarse meshes can be considered as the source domain and used to improve the predictions on high-resolution meshes.

Further work

Continuing our research in the field of deep learning for solving PDEs, there are several interesting directions that could be explored further.

Studies on 3D domains While the present work demonstrates the effectiveness of GNN-based models in solving PDEs, our current work is limited to simple simulated cases on 2D domains. In the real world, where physical problems are often defined on 3D domains with meshes with tens of thousands of nodes, predicting solutions can be more challenging and require larger amounts of data. Further investigation is needed to examine how these methods scale up to very

large meshes and to 3D domains. One approach is to directly train a model with the multi-resolution GNNs on 3D mesh data to predict the solutions from PDEs (Chapter 4).

Incorporating real experimental data The present study employs datasets generated by traditional numerical solvers, yet, directly utilizing data obtained from real experiments could enhance the model's representation of real-world systems and help to bypass the limitations of the PDE model itself. However, real-world measurements are often scarce and insufficient on their own to train a model. A possible approach could be to combine datasets obtained from simulations and real-world measurements to train the model. Another way could be incorporating the framework of the transfer learning method proposed in Chapter 6. Data from the simulation is considered as the low-fidelity dataset and can be further used to help the predictions on real systems. Alternatively, real-world measurements can be utilized to validate the model performance and ensure the accuracy and reliability of the model predictions.

Automatic Meshing To improve the proposed multi-resolution GNN models, it is possible that direct mesh generation algorithms to create a set of meshes of different scales employed in the current study may not be sufficient, as problems with different physical parameters may require different sampling methods. One possible way is to incorporate adaptive mesh refinement techniques [148] used in numerical analysis to refine the mesh after initial simulation. Specifically, after each iteration of the training process for deep learning models, areas with higher prediction errors can be identified and refined automatically to generate new mesh scales. By combining the proposed multi-resolution GNN models with adaptive mesh refinement, more accurate and efficient solutions can be achieved at a reasonable computational cost.

Conclusions

The use of deep learning to solve PDEs has seen great progress in recent years, but it is still in the early stages. The present thesis is an attempt to develop precise and efficient deep learning models. However, there are still many challenges that need to be overcome before we can fully benefit from these techniques for industry.

Nonetheless, the potential of deep learning for solving PDEs is immense. Further research and the development of more precise and efficient models could lead to significant breakthroughs in various industrial fields.

References

[1] Eric W Weisstein. *Euler-Lagrange Differential Equation. From MathWorld–A Wolfram Web Resource.* URL: `https://mathworld.wolfram.com/Euler-LagrangeDifferentialEquation.html`.

[2] *Linear Wave Equation. EqWorld: The World of Mathematical Equations.* URL: `http://eqworld.ipmnet.ru/en/solutions/lpde/wave-toc.pdf`.

[3] *Laplace Equation. Encyclopedia of Mathematics, EMS Press.* URL: `https://encyclopediaofmath.org/index.php?title=Laplace_equation`.

[4] LP Eisenhart. "Enumeration of potentials for which one-particle Schrödinger equations are separable". In: *Physical Review* 74.1 (1948), p. 87.

[5] Kim H Parker and CJH Jones. "Forward and backward running waves in the arteries: analysis using the method of characteristics". In: (1990).

[6] RI Nuruddeen et al. "A review of the integral transforms-based decomposition methods and their applications in solving nonlinear PDEs". In: *Palestine Journal of Mathematics* 7.1 (2018), pp. 262–280.

[7] Henri Poincaré. "Sur les équations aux dérivées partielles de la physique mathématique". In: *American Journal of Mathematics* (1890), pp. 211–294.

[8] Henri Poincaré. *Théorie du potentiel Newtonien: Leçons professées à la Sorbonne pendant le premier semestre 1894-1895.* Carré et Naud, 1899.

[9] Henri Poincaré. "Fonctions modulaires et fonctions fuchsiennes". In: *Annales de la Faculté des sciences de Toulouse: Mathématiques.* Vol. 3. 1911, pp. 125–149.

[10] Sergei Godunov and I Bohachevsky. "Finite difference method for numerical computation of discontinuous solutions of the equations of fluid dynamics". In: *Matematičeskij sbornik* 47.3 (1959), pp. 271–306.

[11] Junuthula Narasimha Reddy. *Introduction to the finite element method.* McGraw-Hill Education, 2019.

[12] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method.* Pearson education, 2007.

[13] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial.* SIAM, 2000.

[14] Wilhelmus HA Schilders, Henk A Van der Vorst, and Joost Rommes. *Model order reduction: theory, research aspects and applications*. Vol. 13. Springer, 2008.

[15] AL Samuel. "Some studies in machine learning using the game of checkers. 1959". In: *IBM Journal of Research and Development. http://dl. acm. org/citation. cfm* ().

[16] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[17] Yan-Yan Song and LU Ying. "Decision tree methods: applications for classification and prediction". In: *Shanghai archives of psychiatry* 27.2 (2015), p. 130.

[18] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297.

[19] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[20] Irina Rish et al. "An empirical study of the naive Bayes classifier". In: *IJCAI 2001 workshop on empirical methods in artificial intelligence*. Vol. 3. 22. 2001, pp. 41–46.

[21] Aleksei Grigor**********evich Ivakhnenko et al. *Cybernetics and forecasting techniques*. Vol. 8. American Elsevier Publishing Company, 1967.

[22] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[23] Emmanuel Menier et al. "Continuous Methods: Adaptively intrusive reduced order model closure". In: *arXiv preprint arXiv:2211.16999* (2022).

[24] Emmanuel Menier et al. "CD-ROM: Complemented Deep-Reduced order model". In: *Computer Methods in Applied Mechanics and Engineering* 410 (2023), p. 115985.

[25] Wei Tang, Tao Shan, et al. "Study on a Poisson's Equation Solver Based On Deep Learning Technique". In: *IEEE EDAPS*. 2017, pp. 1–3. arXiv: 1712.05559 [physics.comp-ph].

[26] Saakaar Bhatnagar, Yaser Afshar, et al. "Prediction of aerodynamic flow fields using CNNs". In: *Computational Mechanics* 64.2 (2019), pp. 525–545.

[27] Jonathan Tompson et al. *Accelerating Eulerian Fluid Simulation With Convolutional Networks*. 2017. arXiv: 1607.03597 [cs.CV].

[28] Nils Thuerey et al. "Deep learning methods for Reynolds-averaged Navier–Stokes simulations of airfoil flows". In: *AIAA Journal* 58.1 (2020), pp. 25–36.

[29] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[30] Ali Kashefi, Davis Rempe, and Leonidas J. Guibas. *A Point-Cloud Deep Learning Framework for Prediction of Fluid Flow Fields on Irregular Geometries*. 2020. arXiv: 2010.09469 [cs.LG].

[31] Charles R. Qi et al. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. 2017. arXiv: 1612.00593 [cs.CV].

[32] Filipe de Avila Belbute-Peres, Thomas D. Economon, and J. Zico Kolter. "Combining Differentiable PDE Solvers and GNNs for Fluid Flow Prediction". In: *37th ICML*. 2020. arXiv: 2007.04439 [cs.LG].

[33] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[34] Jens Berg and Kaj Nyström. "A unified deep ANN approach to PDEs in complex geometries". In: *Neurocomputing* 317 (Nov. 2018), pp. 28–41.

[35] Maziar Raissi. "Deep hidden physics models: Deep learning of nonlinear partial differential equations". In: *JMLR* 19.1 (2018), pp. 932–955.

[36] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational physics* 378 (2019), pp. 686–707.

[37] Wenzhuo Liu, Mouadh Yagoubi, and Marc Schoenauer. "Multi-resolution graph neural networks for pde approximation". In: *International Conference on Artificial Neural Networks*. Springer. 2021, pp. 151–163.

[38] Wenzhuo Liu, Mouadh Yagoubi, and Marc Schoenauer. *Meta-Learning for Airflow Simulations with Graph Neural Networks*. 2023. arXiv: 2306.10624 [cs.LG].

[39] Wenzhuo Liu et al. "Multi-Fidelity Transfer Learning for accurate data-based PDE approximation". In: *NeurIPS 2022-Workshop on Machine Learning and the Physical Sciences*. 2022.

[40] Daryl L Logan. *A first course in the finite element method*. Cengage Learning, 2016.

[41] Mats G Larson and Fredrik Bengzon. *The finite element method: theory, implementation, and applications*. Vol. 10. Springer Science & Business Media, 2013.

[42] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[43] Philipp Ludwig Seidel. *Ueber ein verfahren, die gleichungen, auf welche die methode der kleinsten quadrate führt, sowie lineäre gleichungen überhaupt, durch successive annäherung aufzulösen*. Vol. 11. Verlag d. Akad., 1873.

[44] Ernest Lindelöf. "Sur l'application de la méthode des approximations successives aux équations différentielles ordinaires du premier ordre". In: *Comptes rendus hebdomadaires des séances de l'Académie des sciences* 116.3 (1894), pp. 454–457.

[45] Carl T Kelley. *Solving nonlinear equations with Newton's method*. SIAM, 2003.

[46] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[47] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[48] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[49] Robert Tibshirani. "Regression shrinkage and selection via the lasso". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1 (1996), pp. 267–288.

[50] Arthur E Hoerl and Robert W Kennard. "Ridge regression: Biased estimation for nonorthogonal problems". In: *Technometrics* 12.1 (1970), pp. 55–67.

[51] DN Lawley. "A generalization of Fisher's z test". In: *Biometrika* 30.1/2 (1938), pp. 180–187.

[52] Student. "The probable error of a mean". In: *Biometrika* (1908), pp. 1–25.

[53] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. *Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test*. American Cyanamid Company, 1963.

[54] David E Rumelhart, James L McClelland, PDP Research Group, et al. *Parallel distributed processing*. Vol. 1. IEEE New York, 1988.

[55] Anil K Jain, Jianchang Mao, and K Moidin Mohiuddin. "Artificial neural networks: A tutorial". In: *Computer* 29.3 (1996), pp. 31–44.

[56] Eric W Weisstein. "Heaviside step function". In: *https://mathworld. wolfram. com/* (2002).

[57] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Icml*. 2010.

[58] Bing Xu et al. "Empirical evaluation of rectified activations in convolutional network". In: *arXiv preprint arXiv:1505.00853* (2015).

[59] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[60] Omar Hernández Rodrı**********guez and Jorge M Lopez Fernandez. "A semiotic reflection on the didactics of the chain rule". In: *The Mathematics Enthusiast* 7.2 (2010), pp. 321–332.

[61] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of machine learning research* 12.7 (2011).

[62] Geoffrey Hinton. *Lecture 6a Overview of mini-batch gradient descent.*

[63] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[64] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[65] Gao Huang et al. "Convolutional networks with dense connectivity". In: *IEEE transactions on pattern analysis and machine intelligence* (2019).

[66] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics.* JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[67] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision.* 2015, pp. 1026–1034.

[68] Organització Internacional per a la Normalització. *Accuracy (trueness and Precision) of Measurement Methods and Results.* International Organization for Standardization, 1994.

[69] David R Cox. "The regression analysis of binary sequences". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 20.2 (1958), pp. 215–232.

[70] "Mean Squared Error". In: *Encyclopedia of Machine Learning.* Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 653–653. URL: https://doi.org/10.1007/978-0-387-30164-8_528.

[71] "Mean Absolute Error". In: *Encyclopedia of Machine Learning.* Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 653–653. URL: https://doi.org/10.1007/978-0-387-30164-8_528.

[72] Sewall Wright. "Correlation and causation". In: (1921).

[73] Eric W Weisstein. *"Relative Error" From MathWorld–A Wolfram Web Resource.* URL: https://mathworld.wolfram.com/RelativeError.html.

[74] Tomáš Mikolov et al. "Statistical language models based on neural networks". In: *Presentation at Google, Mountain View, 2nd April* 80.26 (2012).

[75] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[76] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[77] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[78] Ehsan Fathi and Babak Maleki Shoja. “Deep neural networks for natural language processing”. In: *Handbook of statistics*. Vol. 38. Elsevier, 2018, pp. 229–316.

[79] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[80] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.

[81] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. “Learning deconvolution network for semantic segmentation”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1520–1528.

[82] Joan Bruna et al. *Spectral Networks and Locally Connected Networks on Graphs*. 2014. arXiv: 1312.6203 [cs.LG].

[83] Eric W Weisstein. “Convolution theorem”. In: *From MathWorld-A Wolfram Web Resource, 2006a. URL http://mathworld. wolfram. com/ConvolutionTheorem. html* (2014).

[84] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. “CNNs on Graphs with Fast Localized Spectral Filtering”. In: *NeurIPS*. 2017. arXiv: 1606.09375 [cs.LG].

[85] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *5th ICLR*. 2017.

[86] James Atwood and Don Towsley. *Diffusion-Convolutional Neural Networks*. 2016. arXiv: 1511.02136 [cs.LG].

[87] Petar Veličković et al. “Graph Attention Networks”. In: *ICLR*. 2018. arXiv: 1710.10903 [stat.ML].

[88] Federico Monti, Davide Boscaini, et al. “Geometric deep learning on graphs and manifolds using mixture model CNNs”. In: *CVPR*. 2016. arXiv: 1611.08402 [cs.CV].

[89] Charles R. Qi et al. *PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space*. 2017. arXiv: 1706.02413 [cs.CV].

[90] Zonghan Wu et al. "A comprehensive survey on graph neural networks". In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[91] Justin Gilmer et al. *Neural Message Passing for Quantum Chemistry*. 2017. URL: https://arxiv.org/abs/1704.01212.

[92] Matthias Fey et al. "Splinecnn: Fast geometric deep learning with continuous b-spline kernels". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 869–877.

[93] Stephen T Barnard and Horst D Simon. "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems". In: *Concurrency: Practice and experience* 6.2 (1994), pp. 101–117.

[94] Bruce Hendrickson, Robert W Leland, et al. "A Multi-Level Algorithm For Partitioning Graphs." In: *SC* 95.28 (1995), pp. 1–14.

[95] Hongyang Gao and Shuiwang Ji. *Graph U-Nets*. 2019. URL: https://arxiv.org/abs/1905.05178.

[96] Boris Knyazev, Graham W. Taylor, and Mohamed R. Amer. *Understanding Attention and Generalization in Graph Neural Networks*. 2019. URL: https://arxiv.org/abs/1905.02850.

[97] Rex Ying et al. "Hierarchical Graph Representation Learning with Differentiable Pooling". In: *CoRR* abs/1806.08804 (2018). arXiv: 1806.08804. URL: http://arxiv.org/abs/1806.08804.

[98] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. "Self-Attention Graph Pooling". In: *CoRR* abs/1904.08082 (2019). arXiv: 1904.08082. URL: http://arxiv.org/abs/1904.08082.

[99] Frederik Diehl. "Edge contraction pooling for graph neural networks". In: *arXiv preprint arXiv:1905.10990* (2019).

[100] Fuzhen Zhuang et al. "A comprehensive survey on transfer learning". In: *Proceedings of the IEEE* 109.1 (2020), pp. 43–76.

[101] Jeremy Howard and Sebastian Ruder. *Universal Language Model Fine-tuning for Text Classification*. 2018. URL: https://arxiv.org/abs/1801.06146.

[102] Jui-Ting Huang et al. "Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, pp. 7304–7308.

[103] Maxime Oquab et al. "Learning and transferring mid-level image representations using convolutional neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 1717–1724.

[104] Sinno Jialin Pan and Qiang Yang. "A survey on transfer learning". In: *IEEE Transactions on knowledge and data engineering* 22.10 (2009), pp. 1345–1359.

[105] Chuanqi Tan et al. "A survey on deep transfer learning". In: *International conference on artificial neural networks*. Springer. 2018, pp. 270–279.

[106] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. "A survey of transfer learning". In: *Journal of Big data* 3.1 (2016), pp. 1–40.

[107] Wenyuan Dai et al. "Transferring naive bayes classifiers for text classification". In: *AAAI*. Vol. 7. 2007, pp. 540–545.

[108] Joaquin Quinonero-Candela et al. *Dataset shift in machine learning*. Mit Press, 2008.

[109] Steffen Bickel, Michael Brückner, and Tobias Scheffer. "Discriminative learning for differing training and test distributions". In: *Proceedings of the 24th international conference on Machine learning*. 2007, pp. 81–88.

[110] Jing Jiang and ChengXiang Zhai. "Instance weighting for domain adaptation in NLP". In: ACL. 2007.

[111] Wenyuan Dai et al. "Co-clustering based classification for out-of-domain documents". In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007, pp. 210–219.

[112] Hal Daumé III. "Frustratingly easy domain adaptation". In: *arXiv preprint arXiv:0907.1815* (2009).

[113] Jindong Wang et al. "Balanced distribution adaptation for transfer learning". In: *2017 IEEE international conference on data mining (ICDM)*. IEEE. 2017, pp. 1129–1134.

[114] Lilyana Mihalkova, Tuyen Huynh, and Raymond J Mooney. "Mapping and revising markov logic networks for transfer learning". In: *Aaai*. Vol. 7. 2007, pp. 608–614.

[115] Lilyana Mihalkova and Raymond J Mooney. "Transfer learning by mapping with minimal target data". In: *Proceedings of the AAAI-08 workshop on transfer learning for complex tasks*. 2008.

[116] Lilyana Mihalkova and Raymond J Mooney. "Transfer learning by mapping with minimal target data". In: *Proceedings of the AAAI-08 workshop on transfer learning for complex tasks*. 2008.

[117] Yoshua Bengio, Aaron Courville, and Pascal Vincent. "Representation learning: A review and new perspectives". In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1798–1828.

[118] Timothy Hospedales et al. "Meta-learning in neural networks: A survey". In: *IEEE transactions on pattern analysis and machine intelligence* 44.9 (2021), pp. 5149–5169.

[119] Chelsea Finn. *Stanford's CS 330 class on Deep Multi-Task and Meta-Learning*. Sept. 2021.

[120] Adam Santoro et al. "Meta-learning with memory-augmented neural networks". In: *International conference on machine learning*. PMLR. 2016, pp. 1842–1850.

[121] Tsendsuren Munkhdalai and Hong Yu. *Meta Networks*. 2017. URL: https://arxiv.org/abs/1703.00837.

[122] Nikhil Mishra et al. *A Simple Neural Attentive Meta-Learner*. 2017. URL: https://arxiv.org/abs/1707.03141.

[123] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. "Siamese neural networks for one-shot image recognition". In: *ICML deep learning workshop*. Vol. 2. Lille. 2015, p. 0.

[124] Oriol Vinyals et al. "Matching networks for one shot learning". In: *Advances in neural information processing systems* 29 (2016).

[125] Flood Sung et al. "Learning to compare: Relation network for few-shot learning". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 1199–1208.

[126] Jake Snell, Kevin Swersky, and Richard Zemel. "Prototypical networks for few-shot learning". In: *Advances in neural information processing systems* 30 (2017).

[127] Sachin Ravi and Hugo Larochelle. "Optimization as a model for few-shot learning". In: (2016).

[128] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-agnostic meta-learning for fast adaptation of deep networks". In: *International conference on machine learning*. PMLR. 2017, pp. 1126–1135.

[129] Alex Nichol, Joshua Achiam, and John Schulman. *On First-Order Meta-Learning Algorithms*. 2018. URL: https://arxiv.org/abs/1803.02999.

[130] Zhenguo Li et al. "Meta-sgd: Learning to learn quickly for few-shot learning". In: *arXiv preprint arXiv:1707.09835* (2017).

[131] Harkirat Singh Behl, Atılım Güneş Baydin, and Philip HS Torr. "Alpha maml: Adaptive model-agnostic meta-learning". In: *arXiv preprint arXiv:1905.07435* (2019).

[132] Fengwei Zhou, Bin Wu, and Zhenguo Li. "Deep meta-learning: Learning to learn in the concept space". In: *arXiv preprint arXiv:1802.03596* (2018).

[133] Luisa Zintgraf et al. "Fast context adaptation via meta-learning". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 7693–7702.

[134] Antreas Antoniou, Harrison Edwards, and Amos Storkey. *How to train your MAML*. 2018. URL: https://arxiv.org/abs/1810.09502.

[135] Chelsea Finn and Sergey Levine. "Meta-Learning and Universality: Deep Representations and Gradient Descent can Approximate any Learning Algorithm". In: *CoRR* abs/1710.11622 (2017). arXiv: 1710.11622. URL: http://arxiv.org/abs/1710.11622.

[136] Tobias Pfaff et al. "Learning mesh-based simulation with graph networks". In: *arXiv preprint arXiv:2010.03409* (2020).

[137] Ali Kashefi, Davis Rempe, and Leonidas J. Guibas. "A Point-Cloud Deep Learning Framework for Prediction of Fluid Flow Fields on Irregular Geometries". In: *arXiv:2010.09469 [physics]* (Oct. 15, 2020). arXiv: 2010.09469. URL: http://arxiv.org/abs/2010.09469 (visited on 11/25/2020).

[138] Jonathan Richard Shewchuk. "Delaunay refinement algorithms for triangular mesh generation". In: *Computational geometry* 22.1-3 (2002), pp. 21–74.

[139] The CGAL Project. *CGAL User and Reference Manual*. 5.1. CGAL Editorial Board, 2020. URL: https://doc.cgal.org/5.1/Manual/packages.html.

[140] Osborne Reynolds. "IV. On the dynamical theory of incompressible viscous fluids and the determination of the criterion". In: *Philosophical transactions of the royal society of london.(a.)* 186 (1895), pp. 123–164.

[141] Philippe Spalart and Steven Allmaras. "A one-equation turbulence model for aerodynamic flows". In: *30th aerospace sciences meeting and exhibit*. 1992, p. 439.

[142] *Automatic Airfoil C-Grid Generation*. URL: https://curiosityfluids.com/2019/04/22/automatic-airfoil-cmesh-generation-for-openfoam-rev-1/.

[143] Bob Allen. "NACA Airfoils". In: *NASA. January* 31 (2017).

[144] Charles L Ladson. *Effects of independent variation of Mach and Reynolds numbers on the low-speed aerodynamic characteristics of the NACA 0012 airfoil section*. Vol. 4074. National Aeronautics, Space Administration, Scientific, and Technical . . ., 1988.

[145] NASA. *2D NACA 0012 Airfoil Validation Case*. URL: https://turbmodels.larc.nasa.gov/naca0012_val.html.

[146] Dmitrii Kochkov et al. "Machine learning–accelerated computational fluid dynamics". In: *Proceedings of the National Academy of Sciences* 118.21 (2021), e2101784118.

[147] Konstantinos Poulios et al. "GetFEM: Automated FE modeling of multiphysics problems based on a generic weak form language". In: (2020).

[148] Marsha J Berger and Joseph Oliger. "Adaptive mesh refinement for hyperbolic partial differential equations". In: *Journal of computational Physics* 53.3 (1984), pp. 484–512.