Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Exploratory Analysis of Turbulent Flow Data using GNN-based Surrogate Model

Vaishali Ravishankar

Computational Science and Engineering (International Master's Program)

Technische Universität München

Master's Thesis

Exploratory Analysis of Turbulent Flow Data using GNN-based Surrogate Model

Author:	Vaishali Ravishankar
1st examiner:	Univ.-Prof. Dr. Hans Joachim Bungartz
2nd examiner:	Prof. Dr. Jochen Garcke
Assistant advisor:	Kislaya Ravi, Christian Gscheidle, Arno Feiden
Submission Date:	April 15th, 2024

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

April 15th, 2024 Vaishali Ravishankar

Acknowledgments

If someone helped you or supported you through your studies, this page is a good place to tell them how thankful you are.

"People sometimes ask me if it is a sin in the Church of Emacs to use vi. Using a free version of vi is not a sin; it is a penance. So happy hacking"

-Richard Stallman

Abstract

Turbulent flows, characterized by their complex and chaotic nature, play a pivotal role in various engineering and natural systems. Understanding and analyzing these phenomena is essential for optimizing design, predicting crucial outcomes and addressing real-world challenges. Therefore, obtaining accurate, efficient and rapid predictions of turbulent behaviors is of utmost importance. Data-driven methods such as deep learning algorithms are being increasingly implemented to speed up flow predictions compared to numerical solvers. However, these models tend to have poor generalization capabilities and are often restricted to simple geometries on structured grids. Hence, a Graph Neural Network (GNN) based surrogate model is proposed to handle unstructured mesh data of turbulent flow simulations. The underlying goal of this research is to leverage the predictions of the surrogate model to perform a comprehensive exploratory analysis of the parameter space that governs the performance of a High-speed Orienting Momentum with Enhanced Reversibility (HOMER) nozzle operating in turbulent flow conditions.

Contents

Part I.

Introduction

1. Introduction

Fluid mechanics, a foundational branch of physics and engineering, underpins technological advancements in numerous critical sectors, including aerospace, automotive engineering, energy systems, environmental science, and biomedical engineering. The predictive modelling of fluid flow is central to the design, optimization, and operational efficiency of countless applications that fuel technological and scientific progress. At the heart of fluid mechanics are the Navier-Stokes Equations (NSE), a set of Partial Differential Equations (PDEs) that describe the motion of fluids in space and time. Despite their foundational importance, these equations pose significant challenges due to their complexity, often defying exact analytical solutions and remaining one of the most perplexing open problems in mathematics [8]. These difficulties ushered in the era of Computational Fluid Dynamics (CFD), a transformative approach employing numerical methods to approximate solutions to the NSE. CFD has become an indispensable tool, leveraging advanced computing algorithms to simulate fluid flow scenarios. Yet, despite its advancements, CFD grapples with computational intensity and a suite of issues ranging from grid dependency to convergence challenges [9], underscoring a persistent need for innovation.

Turbulence modelling, in particular, stands out as a formidable challenge. Turbulence is characterized by velocity and pressure fluctuations across a diverse range of scales, from large vortices to minute eddies, compounded by numerical instabilities, especially in regions close to walls. While Direct Numerical Simulation (DNS) offers an avenue for precise modelling, its demand for extensive computational resources often renders it impractical for routine applications [28]. Consequently, in many practical scenarios, simplified turbulence models are employed, even though this comes at the expense of accuracy.

Advancements in the dynamically evolving field of CFD have seen substantial efforts channeled into enhancing turbulence models [33], improving meshing techniques [41], developing efficient Reduced Order Modelling (ROM) surrogates [2], and reducing computational complexity [39]. The pursuit of efficiency and optimization in CFD, marked by a growing interest in Machine Learning (ML) and Deep Learning (DL), has led to the application of these technologies for fluid dynamics. The integration of DL into CFD represents a promising frontier, offering rapid predictions for new data sets and the potential to generate novel insights into fluid dynamics ([22], [35]). Unlike traditional computational methods, which are tethered to the constraints of processing power and simulation time, DL algorithms, once trained, can offer rapid predictions for new data sets. The potential for DL to enhance CFD extends beyond mere computational savings; it encompasses the ability to generate novel insights into fluid dynamics, streamline flow control and optimization strategies, and ultimately transform the way we approach fluid mechanics re-

search and applications.
Despite the advancements, DL methods have not yet been widely adopted in engineering practice, possibly due to the scarcity of extensive datasets and the poor generalization performance on previously unseen data [5]. This thesis attempts to tackle the generalization problem by implementing a novel approach that leverages Graph Neural Networks (GNNs) to enable precise and efficient predictive models, specifically tailored for the nuanced application of nozzle flow simulation.

1.1. Literature review

In this section, we review significant contributions and advancements in the intersection of ML and CFD, with a particular focus on turbulence modelling. Osborne Reynolds [36] laid the groundwork for turbulence modelling in the late 19th century by establishing the eddy viscosity hypothesis and the Reynolds-averaged Navier-Stokes (RANS). The 1970s saw the introduction of the k-ϵ [23] model, which improved RANS models by balancing accuracy with computational demands. Increased computational capabilities later enabled the adoption of DNS and Large Eddy Simulation (LES) [40] for more precise flow simulations. Recent trends include refining these models through advanced closure schemes and integrating ML, with hybrid RANS-LES models emerging as a promising approach for accurate simulations. ML and DL algorithms have been increasingly deployed to construct surrogate models for complex turbulence systems. These surrogate models, embodying reduced-order representations, offer a streamlined computational alternative to exhaustive simulations. In addition, DL has facilitated the generation of data-driven closure terms for RANS and LES models, significantly elevating the precision of turbulence simulations [25]. Among the spectrum of DL methodologies, Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have emerged as particularly potent in modelling turbulence. These networks adeptly capture and learn from the complex patterns inherent in turbulent flows. This capability extends to challenging scenarios such as non-equilibrium and multiphase flows.
By the early 2010s, the exploration of ML in CFD began, with significant contributions like Brunton's review, [6] categorizing ML into supervised, semi-supervised, and unsupervised learning. Supervised learning involves training models on well-labeled datasets to predict predefined outcomes, with regression algorithms playing a pivotal role in predicting continuous variables, such as turbulence quantities/ Semi-supervised learning, blending elements of supervised and unsupervised learning, has shown effectiveness in analyzing time-series data and images. Conversely, unsupervised learning focuses on unlabeled data and identifies patterns, clusters, or structures. These data-driven methodologies include ROM and Proper Orthogonal Decomposition (POD) [3].
Beyond traditional classification, ML techniques can be discerned by their optimization goals. Data-driven models aim to reduce the prediction error, relying on supervised learning for specific applications to ensure accuracy. These models are meticulously designed

and trained for particular applications to maintain high accuracy. In contrast, Physics Informed Neural Networks (PINNs) [35] integrate physical principles into the training process, harnessing both mathematical models and empirical equations that govern the data. This approach enables PINNs to streamline the model training for physics-centric issues in a computationally efficient manner.
Zhang and Duraisamy's early exploration [46] of data-driven turbulence closure models via multiscale Gaussian process regression paved the way for enhanced RANS equations. Similarly, Ling, Kurzawski, and Templeton [25] significantly advanced RANS models' accuracy by incorporating Galilean invariance through deep neural networks. This period also witnessed the utilization of fully-convolutional neural networks by Guastoni et al. [12] for predicting velocity fields in turbulent flows, demonstrating the applicability of DL in fluid dynamics.
Subsequent research efforts have further broadened the application of ML in CFD. Milano and Koumoutsakos [27] explored the use of neural networks for approximating flow fields around complex geometries, setting a precedent for DL in fluid mechanics. Tracey, Duraisamy, and Alonso ([43]) introduced machine learning techniques to modify turbulence models, illustrating ML's potential to refine simulation models. More recently, Parish and Duraisamy [32] leveraged ML to inform turbulence model discrepancies, enhancing predictive capabilities.
Expanding the scope to unstructured mesh data, recent advancements have explored graph-based and mesh-free techniques for fluid data representation. Trask et al. [45] introduced GMLS-Nets for mesh-free data analysis, demonstrating the versatility of ML approaches in handling complex data structures. Furthermore, Ogoke et al. [31] demonstrated the effectiveness of GCNNs in predicting drag forces around airfoils. Liu et al. [26] showcased the effectiveness of GCNNs and meta-learning in predicting flow dynamics and enhancing turbulence models, highlighting the adaptive capabilities of GNNs in fluid dynamics.
In light of these developments, this research proposes a Graph Neural Network (GNN)-based surrogate model to analyze turbulent flow within a High-speed Orienting Momentum with Enhanced Reversibility (HOMER) nozzle, developed by Michele Trancossi [44].

1.2. Scope and objectives

This thesis focuses on the cutting-edge intersection of Deep Learning (DL), specifically GNNs, and Computational Fluid Dynamics (CFD) to enhance the predictive modelling of fluid dynamics in turbulent nozzle flow simulations. Traditionally, reaching such solutions entails running simulations for extensive time intervals, a process that requires considerable computational resources and time. This thesis introduces an innovative methodology that circumvents the need for prolonged simulation times. By capturing transient solutions at earlier stages—which take significantly less time—and processing them through a surrogate model, we can predict stable, steady-state solutions more efficiently. This approach utilizes Graph Neural Networks (GNNs) as the core technology for the surrogate models,

aiming to significantly reduce the computational burden associated with traditional CFD simulations. The research encapsulated within this thesis covers the development, evaluation, and practical application of a GNN-based surrogate model tailored for nozzle flow simulations. Furthermore, we employ data analysis techniques to categorize simulations based on velocity ratios and Coanda effect occurrences. The objectives set forth outline a clear and structured path towards achieving the goals of this research and its outcomes.

1. **Develop a GNN model for predicting nozzle flow simulation quantities:** The primary objective is to design and train a GNN model, that serves as a surrogate, to accurately predict the steady-state velocity and pressure fields of nozzle flow simulations from early, transient states. This model leverages short-term, less computationally intensive simulation results to accurately forecast stable, steady-state flow conditions.

2. **Investigate the feasibility and efficiency of the surrogate model:** This research seeks to assess the viability of the developed GNN model as a surrogate to traditional CFD simulations for nozzle flow analysis. The surrogate model's efficiency, in terms of computational resources and time, will be evaluated against conventional simulation methods to establish its practicality for real-world applications. This objective focuses on evaluating the computational efficiency and practical feasibility of employing the GNN surrogate model as a viable alternative to conventional, long-duration CFD simulations. It includes an examination of the model's accuracy as well as a detailed comparison between the steady-state solutions predicted by the GNN model and those obtained from traditional CFD simulations, highlighting the computational savings, and potential limitations.

3. **Perform clustering on low-dimensional data to classify simulations:** Another aspect of this thesis is the application of clustering techniques to low-dimensional representations of simulation data. The goal is to categorize simulations based on the velocity ratios between the two inlets of the nozzle to determine occurrences of Coanda adhesion and identify the specific wall (bottom or top) where adhesion takes place. This objective aims to provide deeper insights into the simulation outcomes, facilitating more effective analysis and optimization of nozzle designs and flow conditions.

4. **Investigation of advanced GNN architectures for enhanced model performance**: To explore the potential benefits of incorporating advanced GNN architectures and training strategies, aiming to optimize the model's performance for the complex task of predicting fluid dynamics in nozzle flow scenarios.

Together, these objectives aim to substantially contribute to the field of turbulence modelling using DL techniques, guiding future research and application developments in nozzle flow dynamics and beyond. They offer efficient, and adaptable surrogate modeling approach for simulating the dynamics of nozzle flows, thereby reducing the need for extensive computational resources and time.

Part II.

Background Theory - Fluid Dynamics and Deep Learning

2. Nozzle simulation and fluid mechanics primer

This chapter delves into the nozzle flow dynamics of a High-speed Orienting Momentum with Enhanced Reversibility (HOMER) nozzle, as developed by Trancossi and Dumas [44]. We begin with an understanding of the problem and the underlying principle, which is the Coanda effect. In Section 2.2, we elucidate the governing equations that mathematically define the fluid flow, as well as additional equations for turbulence modelling. Subsequently, we discuss various numerical approaches that discretize the governing equations. Additionally, the chapter provides insights into the simulation setup in Section 2.4, encompassing meshing strategies, boundary condition specifications, and solver settings for the problem.

2.1. Jet deflection in the HOMER nozzle

The HOMER nozzle is designed to produce a controllable and selective deviation of a synthetic jet, generated by mixing two primitive jets, without requiring any mechanical part but solely by taking advantage of the Coanda effect. The general structure of the nozzle is depicted in Figure 2.1. As we can see from the figure, the nozzle has two inlets fed by two impinging jets, followed by a convergence zone, or a septum, where mixing of the flows occurs. The mixing of flows generates a synthetic outflow jet, which can be controlled by modifying the momentums of the primitive jets. Next to the convergence zone is the outflow mouth, with curved walls connected to two convex Coanda surfaces on the top and bottom. The system requires a minimum operating condition of the primitive jets to ensure effective mixing ([44]). The impinging jets must have velocities high enough to generate a synthetic jet of Reynolds number greater than 5000 at the outlet mouth. To guarantee optimum operation, the Reynolds number at the outlet must exceed 10000. In the case of lower Reynolds numbers, the system's behavior is unpredictable.

2.1.1. Coanda effect

Coanda effect is the tendency of a stream of fluid emerging from an orifice to follow an adjacent flat or a curved surface and to entrain fluid from the surroundings so that a region of lower pressure develops. In simple terms, it is the tendency of a fluid to adhere to and stay attached to the walls of a convex surface, as demonstrated in Figure 2.2. Different fluid dynamic effects concur to create the Coanda effect, namely the boundary layer effect,

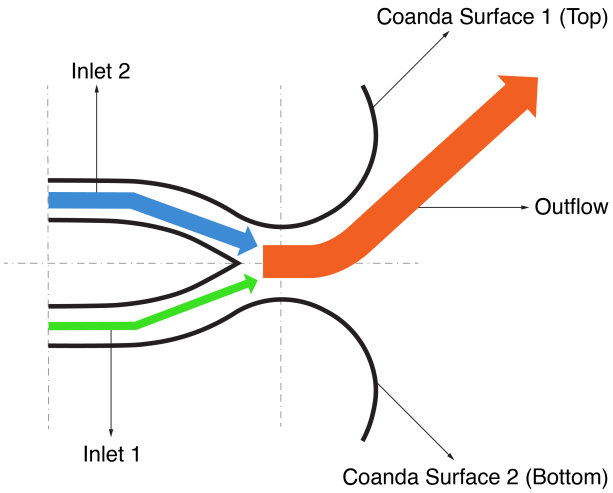

Figure 2.1.: Schematic Overview of the HOMER Nozzle Design: Highlighting dual inlets and Coanda effect surfaces, adapted from Trancossi and Dumas [44]

the adhesion effect, and the attraction effect. Newman [30] has demonstrated that Coanda adhesion to a curved surface is dependent on the equilibrium of forces applied on the fluid. Adhesive motion on a curved surface involves centrifugal force and radial pressure, with contact pressure decreasing due to viscous drag upon jet exit. This pressure differential propels fluid along the curved surface until surface pressure matches ambient pressure, causing detachment between the wall and the fluid jet.

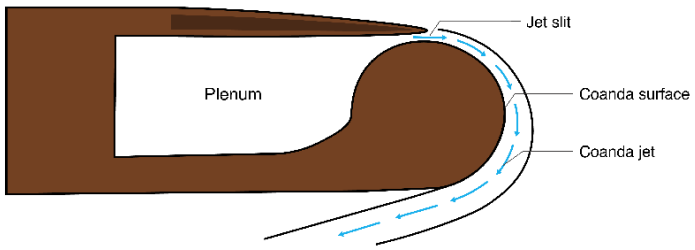

Figure 2.2.: Demonstration of the Coanda Effect: Visualization of a fluid jet adhering to and flowing along a curved surface, illustrating the fundamental principle utilized in the HOMER nozzle design.

2.2. Governing equations

In this section, we talk about the mathematical equations that govern the fluid flow in the nozzle setup. The NSE can be used to mathematically model the flow of an incompressible, Newtonian fluid within the computational domain. Figure 2.3 shows the computational domain for our fluid flow problem. We consider the same homogenous fluid for both primitive jets. This refers to streams with the same chemical and physical properties, i.e; the density of the fluid ρ remains constant. The fluid in consideration is air in ideal gas conditions.

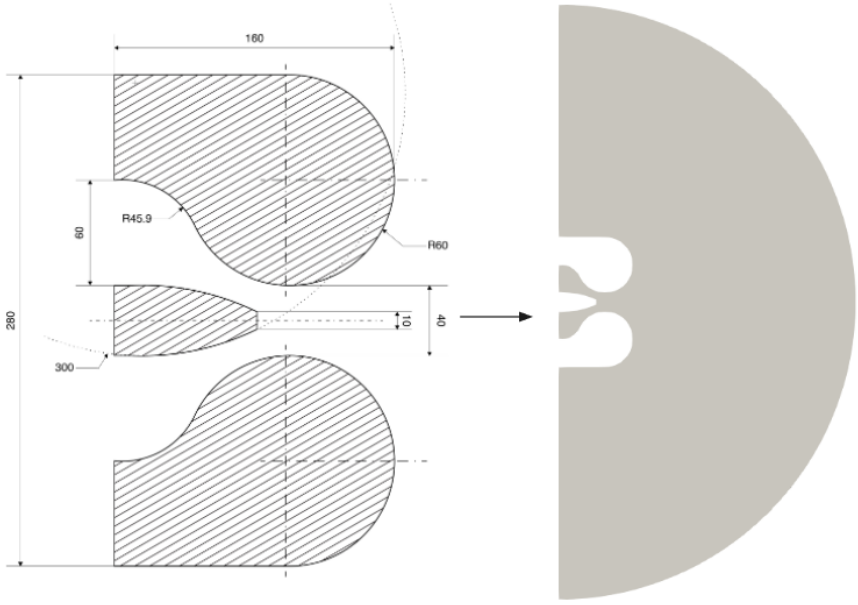

Figure 2.3.: Visualization of the HOMER Nozzle: Left - Geometry of the modified HOMER nozzle; Right - CAD model showcasing the computational simulation domain.

$$\begin{aligned} \frac{\partial u_i}{\partial x_i} &= 0 \\ \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} &= \frac{-1}{\rho} \frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j^2} \end{aligned} \tag{2.1}$$

where, u_i is the flow velocity in the spatial direction x_i, ν is the kinematic viscosity, μ is the dynamic viscosity ($\nu = \mu/\rho$), p is the pressure. A velocity profile from a fully-developed turbulent plane channel flow is prescribed as inlet velocities at $\partial\Omega_{in}$. At the walls $\partial\Omega_{wall}$, we apply no-slip boundary conditions. We impose zero-gradient Neumann boundary conditions on the flow quantities at the outlet.
Turbulence, characterized by its unsteady, highly irregular, rotational and energy dissipa-

tive behaviors at high Reynolds numbers, causes minute fluctuations in velocity, pressure, and temperature across varying scales. While a direct numerical solution (DNS) could theoretically capture these fluctuations by solving the NSE, the immense computational resources required render it impractical for most engineering simulations. Turbulence modelling using RANS (Reynolds-Averaged Navier-Stokes) equations offers a practical compromise by solving time-averaged equations for steady-state or unsteady (URANS) flows. RANS relies on turbulence models to account for unresolved turbulence effects, allowing for efficient simulations of complex engineering systems without resolving every turbulent detail. The underlying principle is to consider the flow as the sum of the mean flow and turbulent/fluctuating components. For a steady-state flow field, Reynolds decomposition is applied to flow quantities. For example, the flow velocity is expressed as $u_i = U_i + u_i'$, where U_i is the mean velocity and u_i' is the fluctuating turbulent component. The Reynolds averaging process introduces an additional term to the NSE known as Reynolds stress. By substituting the averaged quantities in the NSE, we obtain the RANS equations for our steady-state, 2D incompressible flow as,

$$\begin{aligned} \frac{\partial u_i}{\partial x_i} &= 0 \\ u_j \frac{\partial u_i}{\partial x_j} &= \frac{1}{\rho}\frac{\partial}{\partial x_j}\left[-p\delta_{ij} + \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \rho u_i' u_j'\right] \end{aligned} \tag{2.2}$$

where $-\rho u_i' u_j'$ is called the Reynolds stress tensor and represents the effect of the small-scale turbulence on the average flow. Here, u and P are the averaged flow quantities $\langle u \rangle$ and $\langle P \rangle$ but the averaged notation $\langle ... \rangle$ has been omitted for convenience. The RANS equations have no unique solution because they are not in closed form, the unknowns being more than the equations. Thus, additional equations are needed for turbulence closure. The most common strategy used in CFD is to relate the Reynolds stress to the shear rate by the Boussinesq relationship:

$$u_i' u_j' = 2\frac{\mu_t}{\rho} S_{ij} \quad \text{with} \quad S_{ij} = \frac{1}{2}\left(U_{i,j} + U_{j,i}\right) \tag{2.3}$$

where μ_t is the turbulent viscosity, which is usually computed from the turbulence models. Some of the RANS-based turbulence models are outlined below:

1. The Spalart-Allmaras model is a one-equation model that is computationally efficient. It solves for a single variable, the turbulent viscosity, using a transport equation derived from the RANS equations.

2. The k $-$ ϵ model resolves turbulence through two transport equations: one for turbulent kinetic energy (k) and another for the rate of dissipation of turbulent kinetic energy (ϵ).

3. The k $-$ ω model is another two-equation turbulence model that features transport equations for turbulent kinetic energy and a specific rate of turbulence dissipation

(ω). This model is particularly advantageous in accurately predicting near-wall flows and is less susceptible to numerical issues than the k $-$ ϵ model in adverse pressure gradient regions.

4. The k $-$ ω SST (Shear-Stress Transport) model combines aspects of the k $-$ ϵ model near walls and the k $-$ ω model away from walls to provide accurate predictions in both regions. The k $-$ ω SST model is particularly suitable for boundary layer flows, capturing the near-wall behavior accurately while providing robust predictions in the outer flow regions. Its versatility and computational efficiency make it a popular choice for a wide range of engineering applications.

For the purposes of this work, the k $-$ ω SST turbulence model has been adopted.

2.3. Numerical analysis

Numerical analysis on PDEs - in our case, the RANS equations is performed by discretizing the continuous domain into a discrete setup, which results in a system of algebraic equations which are usually linear systems that can be solved by iterative techniques such as Jacobi or Gauss-Seidel. Multigrid methods are another class of iterative numerical techniques used to solve discretized PDEs efficiently for large-scale computational problems. They employ a hierarchy of grids with varying levels of resolution, combining coarse and fine grids to accelerate convergence. By addressing error components on multiple scales, multigrid methods effectively smooth out high-frequency errors while retaining accuracy, making them suitable for problems with smooth solutions or complex geometries.
Some commonly used discretization techniques are the Finite Difference Method (FDM), Finite Element Method (FEM), and Finite Volume Method (FVM). 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.
FVM discretizes PDEs by dividing the computational domain into finite volumes or cells. It conservatively approximates integral forms of conservation laws within each cell. The method calculates fluxes across cell interfaces, preserving conservation principles, making it particularly suited for problems involving fluid flow, heat transfer, and other conservation phenomena. 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 thesis, we are only interested in the discretization of PDEs using FVM, which is the most widely used discretization approach in CFD solvers.

2.4. Simulation setup

Trancossi and Dumas [44] proposed a mathematical model of the HOMER nozzle and carried out 2D, incompressible flow simulations on this geometry. The simplified model predicts the detachment angle of the jet stream over the curved surface. The nozzle chosen for our study is a slightly modified version of a thrust-vectoring propulsive HOMER nozzle. The modified design is inspired by the numerical investigation and experimental validation of Kara and Erpulat [19]. The geometry adopted for the simulations as well as the computational domain are depicted in Figure 2.3. The selected channel length ensures the mean flow quantities are fully-developed, hence establishing steady-state conditions. The meshing and CFD simulation are carried out on `OpenFOAM` which uses finite volume methods to discretize the PDEs.

2.4.1. Mesh generation

The geometry is created using FreeCAD [10] and patch names are assigned based on the type of boundaries. The meshing process on `OpenFOAM` begins with the discretization of the geometry into hexahedral blocks using `blockMesh`. Then, `snappyHexMesh` refines the mesh based on parameters specified in `snappyHexMeshDict`. This includes defining refinement controls, snapping settings, adding boundary layers, and ensuring mesh quality. The process iteratively refines the mesh until the desired quality and resolution are achieved, enabling accurate simulations of the geometry's physical behavior. An unstructured, 3D hybrid mesh with tetrahedral and hexahedral elements mesh has been generated for the computational domain and is enhanced by boundary layer refinement and a refinement box around the nozzle region.

2.4.2. Boundary conditions and solver settings

We carry out an incompressible steady-state CFD simulation on the mesh, setting appropriate boundary and initial conditions. By applying low Reynolds number wall functions for k, ω, and turbulent viscosity ν_t at walls, we account for near-wall turbulence effects, ensuring accurate modeling near boundaries. To maintain a neutral pressure condition, we set the pressure to zero gradient at the walls. We also prescribe adiabatic stationary walls with no slip conditions at the Coanda surfaces and the inner walls of the nozzle. We define the flow entering the domain by prescribing inlets with fixed velocities and set both inlet turbulence intensities to 1% (medium turbulence). Furthermore, we specify the pressure at outlets through a pressure outlet boundary condition, allowing flow to exit the domain without reflecting back. These actions collectively ensure the proper representation of flow behavior within the computational domain. The `simpleFoam` solver is the tool we use for the simulations, employing the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm for effective pressure-velocity coupling. We iteratively resolve the momentum and pressure equations until achieving a predetermined conver-

gence criterion. The $k - \omega$ SST turbulence model serves our purposes, with the fluid's kinematic viscosity set at 1.51×10^{-5}. For velocity's convective term, we use the linear Upwind scheme for discretization, while the divergence term undergoes discretization via a bounded Gauss linear scheme. For the turbulence fields k, ω, ν_t, we employ a bounded Gauss upwind scheme for discretization. We have set the convergence criterion to 10^{-6} for the pressure field and 10^{-8} for all other quantities.

3. Deep learning primer

This chapter provides a comprehensive explanation of foundational concepts and methodologies in deep learning, with a particular emphasis on Graph Neural Networks (GNNs). It begins by introducing deep learning and elucidating the basics of neural networks, focusing on their architecture and operational principles. Section 3.3 navigates through the intricacies of training neural networks with topics including data partitioning, feature scaling, weights initialization, regularization, batch training, and hyperparameter tuning. Subsequently, it delves into optimization techniques, addressing essential elements such as loss functions, backpropagation, learning rates, and optimizers. Furthermore, this chapter examines model evaluation metrics and explores advanced neural network architectures such as Convolutional Neural Networks (CNNs) in Section 3.5 and Graph Neural Networks (GNNs) in Section 3.6, highlighting their important features and components.

3.1. Introduction to machine learning and deep learning

Machine learning, a dynamic subset of artificial intelligence, is dedicated to developing algorithms that extract insights and patterns from data. This enables systems to enhance their accuracy and decision-making capabilities without being explicitly programmed for each task. The learning process utilizes statistical models and optimization algorithms to iteratively adjust parameters and improve performance. Machine learning approaches are broadly categorized into supervised learning and unsupervised learning based on learning objectives.

Supervised learning involves training a model using labeled data, where each input is paired with a corresponding output. During training, the model learns to map input data to output labels by minimizing the difference between its predictions and the true labels. This approach is commonly used for tasks such as classification and regression.

Unsupervised learning, involves training a model on unlabeled data, where the algorithm aims to find hidden patterns or structures within the data without explicit guidance. Unsupervised learning is often used for tasks like anomaly detection, data exploration, and feature learning, where the data lacks labeled examples or where the underlying structure is unknown.

Deep learning is a subset of machine learning that has gained significant attention due to

its remarkable performance in various tasks, ranging from image and speech recognition to autonomous driving. It utilizes neural networks consisting of multiple layers of interconnected neurons to learn representations of data through iterative processing of input data to make predictions or decisions. Henceforth, when we mention machine learning or deep learning, it refers to supervised learning in this context, unless otherwise specified.

3.2. Fundamentals of neural networks

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of biological neural networks [38]. They consist of interconnected nodes organized into layers, typically including an input layer, one or more hidden layers, and an output layer.

A **perceptron** or an artificial neuron, is the fundamental building block of ANNs. It takes multiple input signals $\mathbf{x} = (x_1, x_2 ... x_n)$, each weighted by a connection weight $w_1, w_2, ... w_n$, sums them up, and applies an activation function σ to produce an output $\mathbf{y} = \sigma\left(\mathbf{w}^T\mathbf{x} + \mathbf{b}\right)$, where $\mathbf{b}$ is the bias. Perceptrons are arranged in layers to build complex neural network architectures.

Activation functions are usually non-linear functions to introduce non-linearity within

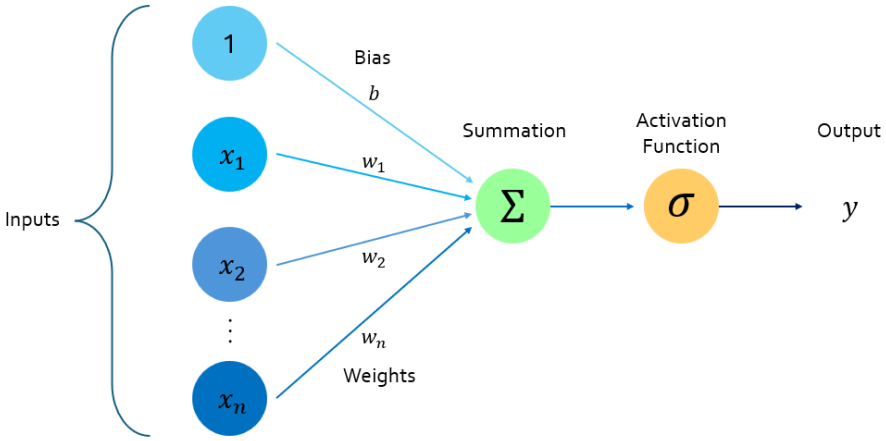

Figure 3.1.: Schematic representation of a perceptron - the basic computational unit of artificial neural networks, illustrating input connections, weights, bias, and an activation function.

the layers of neural networks, allowing them to learn and represent complex relationships in data. Common activation functions include sigmoid, tanh, ReLU (Rectified Linear Unit), and leaky ReLU. Activation functions transform the weighted sum of inputs $\mathbf{z}$ into the output signal $\mathbf{y}$, typically in the range between 0 and 1 (for sigmoid) or -1 and 1 (for tanh).

Without the nonlinear transformation via the activation function, the network would be confined to solving merely linear problems.

Weights in a neural network represent the strength of connections between neurons. They

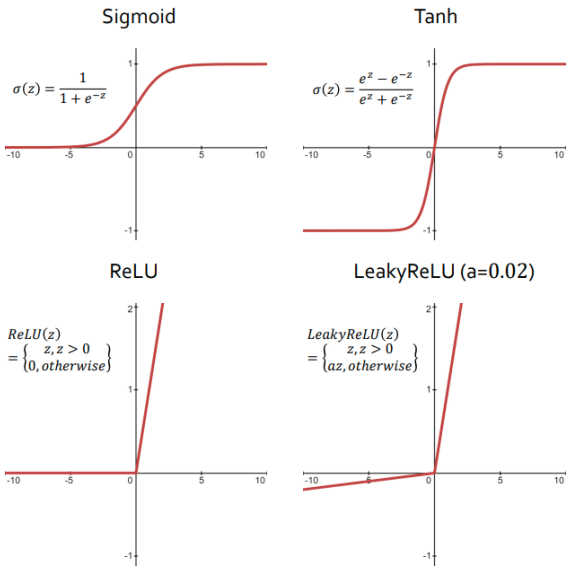

Figure 3.2.: Comparative visualization of common activation functions used in neural networks: (a) Sigmoid, (b) Tanh, (c) ReLU, and (d) Leaky ReLU

are learned parameters to adjust the influence of input signals on the neuron's output. **Biases** allow neural networks to model the offset from zero output, influencing the activation of neurons regardless of the input.

Neural Networks (NNs) consist of interconnected layers of perceptrons that process input data to produce output predictions. NNs can be represented as directed graphs, where nodes correspond to perceptrons, and edges depict connections between them. These connections typically carry weighted signals from one neuron's output to another neuron's input. In particular, we are interested in **feed-forward neural networks**, in which, information flows only in one direction, from the input layer through one or more hidden layers to the output layer. Each layer processes the input data independently, and the output of one layer serves as the input to the next layer. The connections between neurons do not form directed cycles, ensuring that the network architecture is acyclic. Multi Layer Perceptrons (MLPs) are the simplest feed-forward neural networks, and their architecture is described in Figure 3.3.

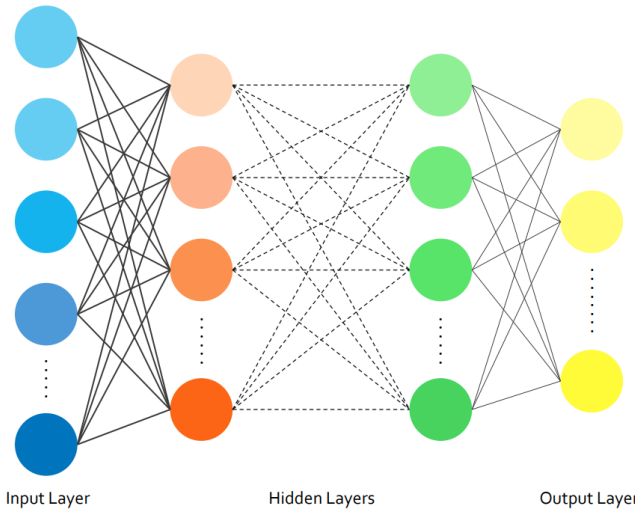

Figure 3.3.: Architecture of a Multi-Layer Perceptron (MLP) showcasing an input layer, multiple hidden layers, and an output layer, demonstrating the flow of information in feed-forward neural networks.

3.3. Training of neural networks

In this section, we delve into the training practices of neural networks, highlighting the optimization of model parameters to enhance task performance. We particularly focus on the methodologies for data handling like data partitioning, feature scaling, and regularization techniques that address model generalization as well as batch training and weight initialization. The process of training neural networks is fundamentally aimed at optimizing a model's parameters to improve its performance on given tasks. Thus, we also talk about optimization strategies, which leverage algorithms like backpropagation to calculate gradients and apply updates via optimizers such as SGD or Adam.

3.3.1. Data partitioning

Data partitioning is a crucial step in deep learning in which the dataset is divided into separate subsets for training, validation, and testing. The data partitioned into training data is used to train the model, and validation data is used to tune hyperparameters and monitor performance. The testing data is used to evaluate the final model's generalization performance. The partitioning process ensures that the model's performance is assessed accurately on unseen data and provides independent datasets for training and evaluation.

3.3.2. Feature scaling

Feature scaling or normalization, is a preprocessing step aimed at bringing all input features to a similar scale. Features with large magnitudes can lead to large gradients during training, which may cause unstable behavior and make it challenging for the optimizer to find the optimum. Feature scaling mitigates this issue by reducing the range of feature values, thus preventing gradient instability and ensuring more reliable optimization. Common feature scaling techniques include,

1. Min-Max normalization: $x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$
2. Z-Score standardization: $x' = \frac{x - \tilde{x}}{\sigma}$
3. Unit length scaling: $\mathrm{x}' = \frac{\mathrm{x}}{\|\mathrm{x}\|}$

3.3.3. Weight initialization

Weight initialization sets the initial values for the model's parameters before training begins. The loss landscape of deep neural networks is complex and non-convex, with multiple local minima. The initial weights dictate the local minimum the weights should converge to; thus, better initialization leads to improved model performance. There are different cases to consider for weight initialization:

1. Initializing all weights to 0 or a constant leads to symmetrical gradients and weight updates across all neurons in the network during backpropagation. This results in the neurons learning identical features, causing the network to lose its representational capacity.

2. Large weights can lead to exploding gradients during training. Large weights can saturate activation functions, pushing them into regions of zero gradients (e.g., in sigmoid or tanh activations), hindering learning and resulting in slow convergence.

3. Small weights help prevent exploding gradients, as activations and gradients remain within a manageable range during training. However, extremely small weights results in small activations, leading to vanishing gradients.

Random initialization with small weights is a common practice in deep learning. Initializing weights to random values drawn from a suitable distribution with a zero mean and small variance breaks symmetry and helps prevent both vanishing and exploding gradients. It encourages each neuron to learn different features from the input data, promoting diverse representations and effective learning. Techniques like Xavier and Kaiming initialization further refine this process by adapting to the specific characteristics of activation functions.

Xavier-Glorot initialization

This technique [11] initializes weights from a normal or uniform distribution with a zero mean. The variance of the distribution is adjusted based on the number of input neurons (fan-in) and output neurons (fan-out) as $\text{Var}(W) = \frac{2}{\text{fan}_{\text{in}}+\text{fan}_{\text{out}}}$. Xavier initialization is commonly used in shallow networks with symmetric activation functions, ensuring balanced weight initialization and stable training dynamics.

Kaiming-He initialization

Kaiming initialization [16] is designed for networks using non-linear activations like ReLU as seen in modern deep learning architectures. It initializes weights from a normal distribution with zero mean, adjusting the variance based on fan-in as $\text{Var}(W) = \frac{2}{\text{fan}_{\text{in}}}$. Kaiming initialization helps mitigate the issue of vanishing gradients associated with ReLU activations, ensuring stable training and faster convergence.

3.3.4. Regularization

Regularization broadly refers to techniques used to prevent overfitting by imposing additional constraints on the model's parameters, i.e; by adding penalties to the loss function, thus discouraging complex models. Regularization penalizes large weights in the model, thereby promoting simpler models that generalize better to unseen data. Two common forms of regularization are L_2 regularization (Lasso) and L_1 regularization (Ridge).
L_1 regularization encourages sparsity in the weights, performing feature selection by setting irrelevant weights to zero, making the model simpler and more interpretable.

$$\text{Loss}_{\text{L}_1} = \text{Loss}_{\text{original}} + \lambda \sum_{i=1}^{n} |w_i|$$

L_2 regularization encourages the weights to be spread out more evenly, preventing individual weights from becoming too large.

$$\text{Loss}_{\text{L}_2} = \text{Loss}_{\text{original}} + \lambda \sum_{i=1}^{n} w_i^2$$

Here, λ is the regularization strength and determines the degree of penalty imposed on large weights. A smaller λ value results in weaker regularization, allowing the model to fit the training data more closely but increasing the risk of overfitting. Conversely, a larger λ value increases the regularization effect, resulting in a simpler model that generalizes better but may underfit the training data if set too high.

Dropout

Dropout is a regularization technique used in neural networks during training, where a random fraction of neurons is temporarily dropped out or ignored during forward and backward propagation. This prevents neurons from co-adapting and overfitting to the training data, promoting robustness and generalization.

3.3.5. Batch training and batch normalization

Batch training is a technique in deep learning in which the model updates its parameters based on a subset (or batch) of the training data, rather than the entire dataset. The training data is divided into batches of fixed size, and the model computes the loss gradients for each batch using backpropagation. Since it computes the gradient updates based on an average over the samples in each batch, this reduces the variance in the gradient estimates compared to processing the entire dataset at once. This averaging effect stabilizes the gradients and prevents large fluctuations during training.
Selecting an appropriate batch size is crucial, as too small a batch size leads to frequent updates and noisy gradient updates. On the other hand, too large a batch requires more computational resources and memory despite providing precise gradient estimates, leading to stable optimization and faster convergence. Optimal batch size requires balancing the trade-off between computational efficiency and the quality of gradient estimates.
Another important term in this context is an epoch, which refers to a single pass through the entire training dataset. During one epoch, each batch is processed sequentially through the neural network. Once all batches have been processed, completing a full iteration over the entire dataset, one epoch is considered complete. Typically, training iterates over the entire dataset for multiple times or epochs until the model converges or a predefined stopping criterion is met.
Batch normalization [18] is a technique to improve the stability of neural networks by normalizing the activations of each layer. It operates on mini-batches of data and normalizes the input of each layer to have a mean of zero and a standard deviation of one. This is achieved by computing the mean and standard deviation of the activations across the batch, and then scaling and shifting the activations using learned parameters. The batch normalization transformation for a layer with input $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ is given as,

$$\begin{aligned} \mu_B &= \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \\ \sigma_B^2 &= \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu_B\right)^2 \\ \hat{x}^{(i)} &= \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ y^{(i)} &= \gamma\hat{x}^{(i)} + \beta \end{aligned} \tag{3.1}$$

where μ_B and σ_B^2 are the mean and variance of the mini-batch B of size m, ϵ is a small constant added to avoid division by zero, $\hat{x}^{(i)}$ is the normalized input, γ and β are learnable parameters (scale and shift), and $y^{(i)}$ is the output of the batch normalization layer.

3.3.6. Overfitting and underfitting

Overfitting and underfitting are two common phenomena that affect the performance and generalization ability of the model. Overfitting occurs when a model learns to perform well on the training data but fails to generalize to unseen data. The model becomes overly complex and specific to the training set, leading to poor generalization. Signs of overfitting include high training accuracy but low validation or test accuracy as the model memorizes training examples. Techniques such as regularization, dropout and early stopping can help prevent overfitting by reducing the model's capacity and complexity.
Underfitting occurs when a model is too simple to capture the underlying structure of the data. In this case, the model fails to learn the patterns present in the training data and performs poorly both on the training and unseen data. Underfitting often occurs when the model is too shallow or simple. Signs of underfitting include low training and validation accuracy. Increasing the model's capacity, adding more data, or improving feature engineering can help alleviate underfitting by allowing the model to capture more complex relationships in the data.

3.3.7. Hyperparameters

Hyperparameters in deep learning are fixed parameters set prior to the training process that are not learned from the data. They control various aspects of the learning process, such as the model architecture, optimization settings and the training procedure itself. Some important hyperparameters are the number of neurons per layer, number of layers, activation function, batch size, number of epochs, optimizer, loss function, weight initialization, dropout rate, and regularization strength.
Hyperparameter tuning is the process of selecting the optimal values for these hyperparameters to maximize the performance of the model on unseen data. It involves systematically searching through a predefined space of hyperparameters and evaluating the model's performance using a validation set or cross-validation. The goal is to find the hyperparameters that result in the best generalization performance, balancing between underfitting and overfitting.

k-Fold cross-validation

In k-fold cross-validation [14], the dataset is divided into k subsets or folds, of approximately equal size. The model is trained k times, each time using k-1 folds for training and the remaining fold for validation. This process is repeated k times, with each fold used exactly once as the validation set. In the context of hyperparameter tuning, k-fold

cross-validation helps evaluate the performance of different hyperparameter configurations. Instead of relying on a single validation set, this method averages the performance over multiple folds, providing a more stable estimate of the model's performance.

3.3.8. Optimization

Optimization involves adjusting the parameters of the neural network, such as weights and biases, to minimize a predefined objective function, typically referred to as the loss function. The optimization process iteratively updates the parameters based on the gradients of the loss function with respect to the network's parameters, aiming to converge to a set of optimal parameters that yield the best performance on the given task. Some important aspects of the optimization process are discussed in the following subsections.

Loss function

The loss function quantifies the difference between the model's predictions and the actual target values. It represents the measure of how well the model is performing on the training data. Common loss functions include mean squared error (MSE) for regression tasks and categorical cross-entropy for classification tasks. The loss function $\mathcal{L}(\theta)$ is defined as:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, \hat{y}_i; \theta\right) \tag{3.2}$$

Here, θ represents the parameters of the neural network being optimized, such as weights and biases, y_i is the ground truth and $\hat{y}_i$ is the model prediction. The loss function $\mathcal{L}(\theta)$ depends on these parameters, and it is computed as the average of the individual loss $L\left(y_i, \hat{y}_i; \theta\right)$ over all training examples of size N.

Backpropagation

Backpropagation is a fundamental algorithm used to compute the gradients of the loss function with respect to the parameters (weights and biases) of the neural network. It involves propagating the error backward from the output layer to the input layer, updating the parameters along the way to minimize the loss. The gradients are computed using the chain rule of calculus, enabling efficient optimization of the network's parameters. Mathematically, the gradients $\nabla_\theta \mathcal{L}(\theta)$ of the loss function are computed as,

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta L\left(y_i, \hat{y}_i; \theta\right) \tag{3.3}$$

Learning rate

The learning rate is a hyperparameter that controls the size of the parameter updates, that is, the step-size in the direction of the gradients computed by backpropagation. A higher learning rate may lead to faster convergence but risks overshooting the optimal solution, while a lower learning rate may result in slower convergence but more stable training. The parameter update rule with learning rate η is given by:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta) \tag{3.4}$$

Here, θ_{t+1} and θ_t represent the parameters at time step t and $t+1$ respectively. Learning rate decay is often used to gradually reduce the learning rate during training with the help of learning rate schedulers such as,

- **Step decay** reduces the learning rate by a factor (typically constant) after a fixed number of epochs or iterations.
- **Exponential decay** reduces the learning rate exponentially over time.

Optimizer

The optimizer is responsible for updating the parameters of the neural network based on the gradients computed during backpropagation. It determines the direction and magnitude of parameter updates to minimize the loss function efficiently. Popular optimizers include stochastic gradient descent (SGD) [4], Adam [20], RMSProp [42], and AdaGrad [7]. We use the Adam optimizer in the training and testing phases of our work.
Adam (Adaptive Moment Estimation) is an algorithm for stochastic optimization that combines the ideas of SGD with momentum and RMSProp. It maintains exponentially decaying moving averages of past gradients and past squared gradients for each parameter. These moving averages serve as estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients, respectively. Adam also incorporates bias correction terms to compensate for the initial bias towards zero at the beginning of training. The parameter update rules at time step t are given by,

1. Compute the gradient of the loss function with respect to the parameters, θ_t.
2. Update biased first moment estimate: $m_t = \beta_1 m_{t-1} + (1-\beta_1)\theta_t$.
3. Update biased second raw moment estimate: $v_t = \beta_2 v_{t-1} + (1-\beta_2)\theta_t^2$.
4. Compute bias-corrected first moment estimate: $\hat{m}_t = \frac{m_t}{(1-\beta_1^t)}$.
5. Compute bias-corrected second moment estimate: $\hat{v}_t = \frac{v_t}{(1-\beta_2^t)}$.
6. Update the parameters: $\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{(\sqrt{\hat{v}_t}+\epsilon)}$.

Here, m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered mean and variance) of the gradients, respectively. β_1 and β_2 are exponential decay rates for these moment estimates, typically close to 1. $\hat{m}_t$ and $\hat{v}_t$ refer to the bias-corrected estimates of the first and second moments (uncentered variance) of the gradients, and ϵ is a small constant to prevent division by zero.

3.4. Model evaluation metrics

Model evaluation metrics are essential for assessing the performance of deep learning models. The loss function serves as a crucial metric for evaluating model performance, in addition to optimizing model parameters during training. In regression tasks, where the goal is to predict continuous values, common loss functions include the following:

- **Mean Absolute Error (MAE):** MAE measures the average absolute difference between the predicted values and the actual values:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

  Here, y_i are the ground truth values, $\hat{y}_i$ are the predicted values, and n is the number of samples. MAE is robust to outliers and does not penalize large errors heavily.

- **Mean Squared Error (MSE):** MSE measures the average squared difference between the predicted values and the actual values:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  MSE penalizes larger errors more heavily than MAE since errors are squared. This makes it more sensitive to outliers.

- **Root Mean Squared Error (RMSE):** RMSE is the square root of the average squared difference between the predicted values and the actual values:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

  RMSE is sensitive to outliers, similar to MSE, but is more interpretable as it is in the same units as the target variable.

3.5. Convolutional Neural Networks (CNNs)

CNNs [24] are specialized neural networks designed to process grid-like data, such as images. They utilize convolutional layers and pooling operators to learn spatial hierarchies of features from input data, making them highly effective for tasks like image classification, object detection, and image segmentation.

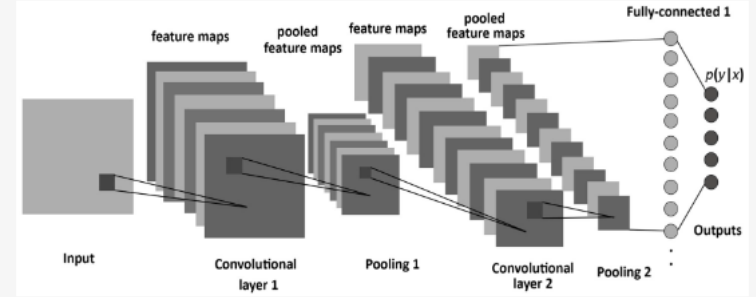

Figure 3.4.: Diagram illustrating the typical structure of a CNN consisting of convolutional, pooling, and fully connected layers. Image taken from [1].

3.5.1. Convolutional layer

The convolutional layer in a Convolutional Neural Network (CNN) consists of a set of learnable filters (kernels) that slide over the input data, performing element-wise multiplication and summing to produce feature maps. The convolution operation preserves the spatial relationship between pixels and learns local patterns like edges, textures, and shapes. Figure 3.5 shows the working of a convolution kernel.

3.5.2. Pooling and unpooling

Pooling and unpooling operators perform effective downsampling and upsampling operations respectively, enabling hierarchical feature extraction while preserving spatial information. Pooling is a down-sampling operation commonly used in CNNs to reduce the spatial dimensions of feature maps. Max pooling and average pooling are popular pooling techniques that select the maximum or average value within each pooling region, respectively. These operations are depicted in Figure 3.6. Conversely, unpooling layers, often used in upsampling, aim to reconstruct the original input resolution from the lower-dimensional representations generated by pooling. These layers typically store the indices of the maximum values during pooling and use them for upsampling. Nearest neighbor interpolation is a simpler upsampling method where each pixel in the input is replicated multiple times to form the output as can be seen in Figure 3.7.

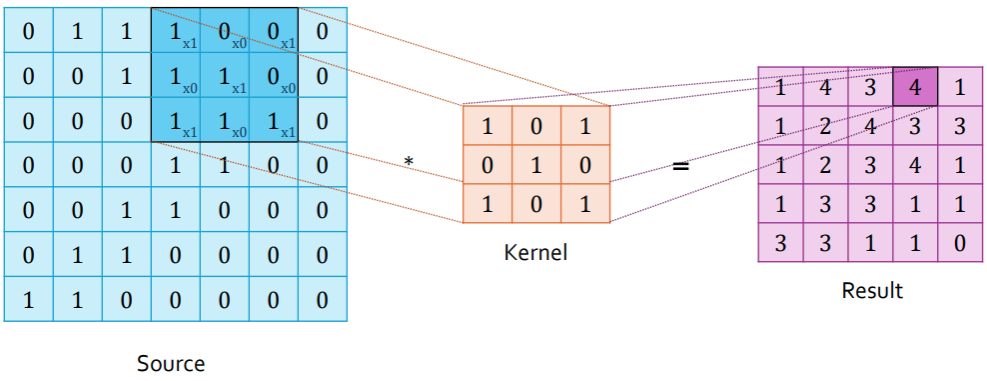

Figure 3.5.: Detailed view of a convolutional layer's operation in a CNN, depicting the convolution process over an input matrix with a specified kernel to produce feature maps.

3.5.3. The U-Net architecture

U-Net [37] is a CNN consisting of a U-shaped network structure with a contracting path (encoder) followed by an expanding path (decoder), which enables precise segmentation of structures in medical images, such as cells, organs, or tumors. Some important components of the U-Net architecture are discussed below.

1. The **encoder** comprises a series of down-convolutional and max pooling layers that gradually reduce the spatial dimensions of the input image while increasing the number of feature channels. This path extracts high-level features from the input image while preserving spatial context.

2. The **decoder** consists of up-convolutional (transposed convolution) and concatenation layers that gradually increase the spatial dimensions of the feature maps while reducing the number of feature channels. This path generates segmentation masks by upsampling the low-resolution feature maps obtained from the encoder and combining them with high-resolution feature maps using skip connections.

3. **Skip connections** or residual connections [17], are direct connections between layers at the same hierarchical level in the network. In the U-Net architecture, skip connections connect the encoder to the corresponding layers in the decoder. This enables the network to retain fine-grained spatial information from the encoder while recovering spatial details lost during downsampling. By directly linking the encoder and decoder layers, skip connections facilitate the flow of information across different scales, improving the model's ability to capture both local and global context.

A notable observation pertinent to our current endeavor is the striking resemblance between the U-Net architecture and the V-cycle multi-grid method, as noted by He and Xu

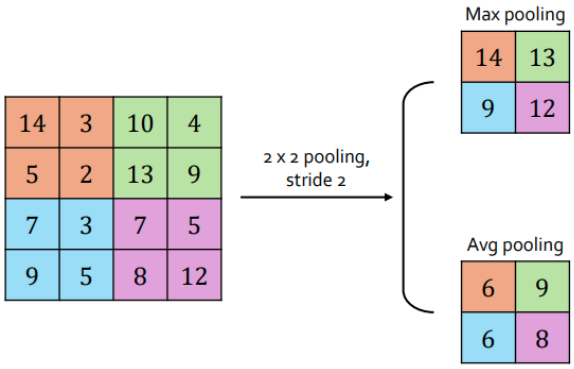

Figure 3.6.: Visualization of pooling operations for downsampling spatial dimensions in CNNs, with examples of (a) max pooling, and (b) average pooling

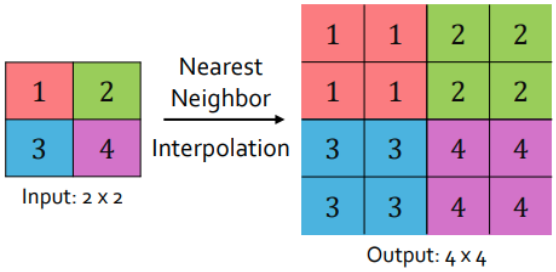

Figure 3.7.: Illustration of the unpooling process of nearest neighbor interpolation, used for upsampling in CNNs

[15]. Both employ a hierarchical structure wherein information is exchanged across varying resolutions.

3.6. Graph Neural Networks (GNNs)

A major limitation of CNNs is their inability to directly operate on irregular data formats, such as social networks, recommender systems, molecular structures, or citation networks. In 2017, Kipf and Welling [21] introduced the Graph Convolutional Network (GCN), a foundational architecture that laid the groundwork for modern GNNs. Since then, numerous advancements and variants of GNNs have been proposed. In contrast to CNNs which are well-suited for grid-like structured data such as images, GNNs are tailored for data represented as graphs, which are characterized by non-Euclidean and irregular structures, where entities (nodes) and their relationships (edges) vary in connectivity and structure.

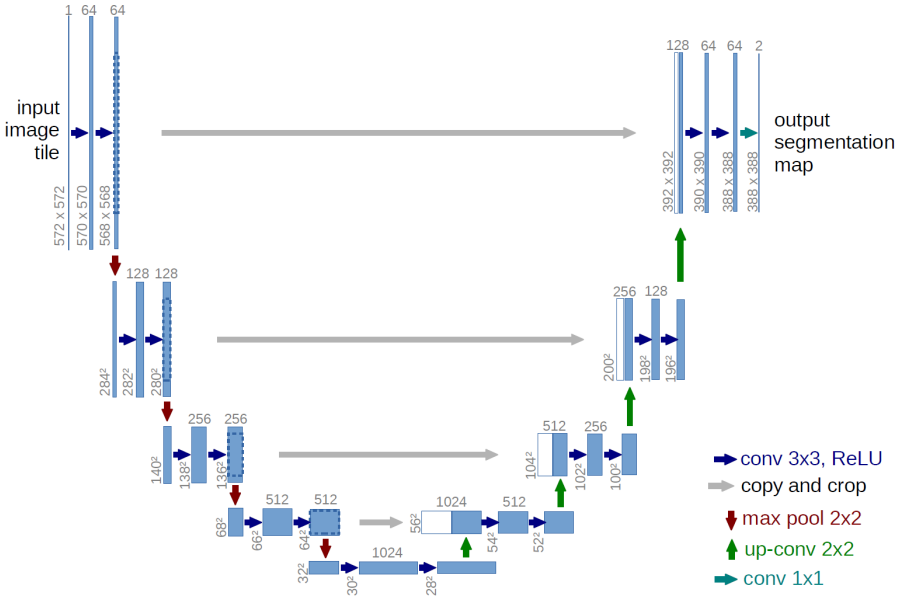

Figure 3.8.: Structure of the U-Net architecture, demonstrating its U-shaped design with contracting and expanding paths [37]

.

GNNs excel in processing unstructured data by leveraging the inherent graph structure by dynamically aggregating information from neighboring nodes based on their connectivity. Graphs are set up using nodes, edges, adjacency matrices, node attributes, and edge attributes.

1. **Nodes (V)**: Nodes represent entities in a graph, such as users in a social network, atoms in a molecule, or words in a document. Formally, a graph can be denoted as $G = (V, E)$, where V is the set of nodes.

2. **Edges (E)**: Edges define relationships or connections between nodes in a graph. Each edge e_{ij} connects node v_i to node v_j, where $v_i, v_j \in V$. The edge set E can be represented as a collection of tuples (v_i, v_j) indicating the connections between nodes.

3. **Adjacency matrix (A)**: An adjacency matrix is a binary $n \times n$ matrix representing the connections between nodes in a graph. For an undirected graph, A_{ij} is 1 if there exists an edge between nodes v_i and v_j, and 0 otherwise. For directed graphs, the adjacency matrix may be asymmetric to represent the directionality of edges. The adjacency matrix A of a graph $G = (V, E)$ can be defined as,

$$A_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

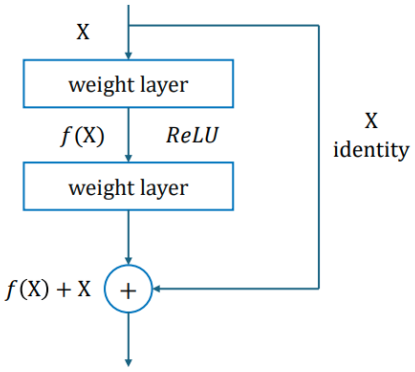

Figure 3.9.: Diagram demonstrating the concept of skip connections within neural network architectures. Skip connections bypass one or more layers by directly feeding the output from an earlier layer to a later layer.

4. **Node attributes and feature matrix(X)**: Node attributes or features represent information associated with each node in the graph. These features can encode characteristics such as velocity, pressure, and temperature as node embeddings. The node feature matrix X for a graph with N nodes and D features is a $N \times D$ matrix where each row corresponds to a node and each column represents a feature dimension, given by,

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_N^T \end{bmatrix} \quad \text{where,} \quad x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{iD} \end{bmatrix}$$

where x_i represents the feature vector associated with node v_i.

5. **Edge weights or attributes (W)**: Edge weights quantify the strength or intensity of relationships between nodes connected by edges. These weights can represent similarity measures, distances, or any other relevant information associated with edge connections. Similar to node weights, edge weights can be learned or predefined.

GNNs leverage these components to perform message passing and aggregation operations across the graph structure, which are discussed below.

3.6.1. Graph convolutions

Graph convolutions update the feature representations of nodes in a graph by aggregating information from their neighboring nodes. There are two ways to perform graph convolution operations. In spectral convolution, graph signals are transformed into the spectral

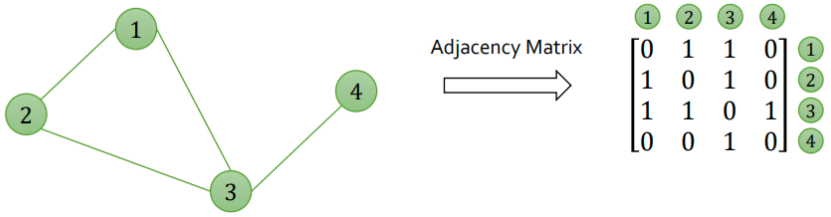

Figure 3.10.: Demonstration of an adjacency matrix representing graph connectivity and the binary representation of node relationships in graph-structured data.

domain using techniques like graph Fourier transforms. Spatial convolution directly operates on the graph's topology and neighborhood structure, aggregating information from neighboring nodes without explicitly transforming the graph. Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and GraphSAGE are common types of GNNs that utilize spatial convolutions. In this work, we only deal with spatial convolutions and refer to them as graph convolutions henceforth. The main steps involved in graph convolutions are as follows:

1. **Message passing**: Nodes exchange messages with their neighbors, aggregating information from neighboring nodes. The message passed from node v_j to node v_i can be represented as:
$$m_{ij} = \frac{1}{c_{ij}} W h_j$$
where m_{ij} is the message from node j to node i, W is the learnable weight matrix, and c_{ij} is a normalization factor.

2. **Aggregation**:Nodes aggregate the messages received from their neighbors to update their own feature representations. The aggregated message a_i for node i can be computed as the sum or average of the incoming messages.
$$a_i = \sum_{j \in \mathcal{N}(i)} m_{ij}$$
where $\mathcal{N}(i)$ denotes the set of neighboring nodes of v_i.

3. **Update**: Nodes update their feature representations using the aggregated messages and their own features. The updated feature representation $h_i^{(l+1)}$ for node i at layer $l+1$ can be computed as:
$$h_i^{(l+1)} = \sigma(a_i)$$

These steps are performed iteratively across multiple layers of the GNN. At each layer, nodes update their feature representations based on the aggregated messages. The iterative propagation of messages allows nodes to incorporate information from distant parts

of the graph and refine their representations over multiple layers.
Graph convolution operators typically exhibit local connectivity, where each node's representation is updated based on the information from its neighboring nodes. This local connectivity property allows the model to capture localized patterns and dependencies within the graph structure. Weight sharing is a key aspect of graph convolutions, where the same set of learnable parameters (weights) is shared across different nodes in the graph. This allows for parameter efficiency and enables the model to generalize well to unseen nodes and graphs.

3.6.2. Graph pooling

Graph pooling aggregates node representations across the entire graph to compute global graph-level features and create a coarser graph representation. It reduces the size of the graph representation while preserving important structural and relational information. By selecting representative nodes or aggregating node features, graph pooling enables GNNs to focus on relevant information while reducing computational complexity. Graph pooling also facilitates hierarchical feature learning by allowing GNNs to operate at multiple levels of granularity, enabling the model to capture both local and global patterns in the graph. The different types of graph pooling are:

1. **Top-k pooling** selects the top k nodes based on criteria such as node importance or feature values, and aggregates their information to create a coarser graph representation. This method retains the most informative nodes while reducing the size of the graph, making it suitable for tasks requiring node selection or summarization.

2. **Max pooling** selects the node with the maximum feature value from each neighborhood and aggregates their information to create a coarser representation of the graph. It emphasizes the most salient nodes in each neighborhood, capturing important features while reducing redundancy.

3. **Self-attention graph pooling** leverages attention mechanisms to dynamically weight the contributions of neighboring nodes based on their importance and similarity. It allows nodes to attend to relevant information in their neighborhoods, facilitating adaptive aggregation and effective summarization of the graph. This method is useful for capturing long-range dependencies and global patterns in the graph.

Graph unpooling is a complementary operation to graph pooling, aimed at upsampling or reconstructing the original graph representation after downsampling. While graph pooling creates a coarser representation, graph unpooling aims to recover the finer details and restore the original graph structure. Some common types of graph unpooling include,

1. **Max unpooling** is an unpooling strategy used in conjunction with max pooling. During max pooling, the locations of the maximum activations are stored. In max unpooling, these locations serve as masks to place the pooled values back into their original positions in the unpooled feature map.

2. **Nearest neighbor interpolation** aims to recover the original graph topology by identifying the nearest neighbors of pooled nodes in the coarse representation and reinstating unpooled nodes based on their proximity. It reconstructs edges between unpooled nodes and their nearest neighbors, restoring connectivity and preserving local structure.

3.6.3. Hierarchical multi-resolution approach

Multi-resolution approaches in the context of GNNs involve operating on graphs at multiple levels of granularity, similar to the multigrid method in numerical analysis. Unlike CNNs, where downsampling operators automatically coarsen the mesh, in GNNs, we create a hierarchy of meshes with increasing complexity over the domain of interest. Hence, traditional pooling operators may not be suitable for GNNs, as they focus on selecting nodes to construct a coarse graph, which is unnecessary for mesh data. Instead, we can easily define operators that transform features from one mesh to the next, by generating a set of meshes with varying coarseness.
Creating a mesh hierarchy of different levels of coarseness can be performed by well-established techniques in numerical analysis. One commonly used algorithm for mesh construction is Delaunay triangulation, which maximizes the minimum angle of all triangles to avoid sliver triangles. This algorithm gradually inserts new nodes into the triangulation and connects them with their neighbors under specific rules. Incremental decimation is another mesh coarsening method that aims to reduce the number of points while preserving specific properties of the original mesh. It iteratively removes one vertex or edge with minimal changes until certain criteria are met. These techniques offer flexibility in creating mesh hierarchies, making them suitable for GNNs applied to mesh data.

Sampling operator

We introduce a sampling operator for converting data between two meshes, denoted as M_1 and M_2, inspired by the k-nearest interpolation proposed in PointNet++ [34]. Let z be a node from M_1, and assume its k nearest neighbors on M_2 are denoted as $x_1, \ldots, x_k$. For a node feature f, the interpolated feature $f(z)$ is defined based on the features of x_i as,

$$\mathbf{f}(\mathbf{z}) = \frac{\sum_{i=1}^{k} w\left(x_i\right) \mathbf{f}\left(x_i\right)}{\sum_{i=1}^{k} w\left(x_i\right)}, \text{ where } w\left(x_i\right) = \frac{1}{\|z - x_i\|_2} \tag{3.5}$$

With these operators, both upsampling and downsampling operators can be defined straightforwardly for developing multi-resolution architectures for mesh-based problems.

Part III.

Implementation, results and discussion

4. Implementation, results and discussion

4.1. Data pre-processing/preparation

4.1.1. Dataset generation

Nozzle simulations are carried out for 120 cases, each with a different set of Inlet 1 and Inlet 2 velocities. The velocity ratio between the two velocities lies in the range of [1,10]. Velocity ratio in our case refers to the ratio of higher velocity to that of lower velocity. Simulation data are obtained at two intervals: 1,000 and 30,000 time-steps. The results at 1,000 time-steps, which are still developing and unstable, serve as inputs to the surrogate model. In contrast, the results at 30,000 time-steps, which represent stable solutions, are considered the ground truth or target data for model training. These datasets are then transformed into graph data, which encapsulate the spatial relationships and properties of flow fields, and normalized to facilitate efficient learning. The GNN architecture is designed to learn these spatial relationships and predict the stable, steady-state fields from the early, unstable simulation results.

4.1.2. Transformation of mesh data to graph data

Conventional Reynolds Averaged Navier-Stokes (RANS) solvers require substantial distances from domain boundaries to mitigate adverse effects on solutions around the region of interest. However, this is not required for the deep learning task. Hence, we narrow our attention to a small region just enclosing the nozzle as seen in Figure 4.1. We clip the CFD mesh appropriately and resample the velocity and pressure fields to this mesh with reduced spatial extent. We define the cell-centers on the clipped mesh and assign them as the nodes of the graph. Two adjacent cells in the mesh (cells that share an edge) corresponds to their respective nodes (v_i and v_j) on the graph being connected by an edge e_{ij}. The graph connectivity is then given by the edge index data structure which comprises two lists - one stores the source node indices and the other has the destination node indices. CFD solvers typically assign pressure, velocity and other fields to each cell of the mesh whereas graphs require node features, i.e; fields defined on each node. Therefore, the cell data (fields) are converted to point data at the cell centers making it suitable for graph representation. The simulation data is then saved in a `hdf5` format. This is then directly used to read u_x, u_y, p, c_x, c_y, and γ_{tag}. The edge index data structure required for the GNN model is generated by computing adjacent cells and storing their indices in a co-ordinate list (COO) format.

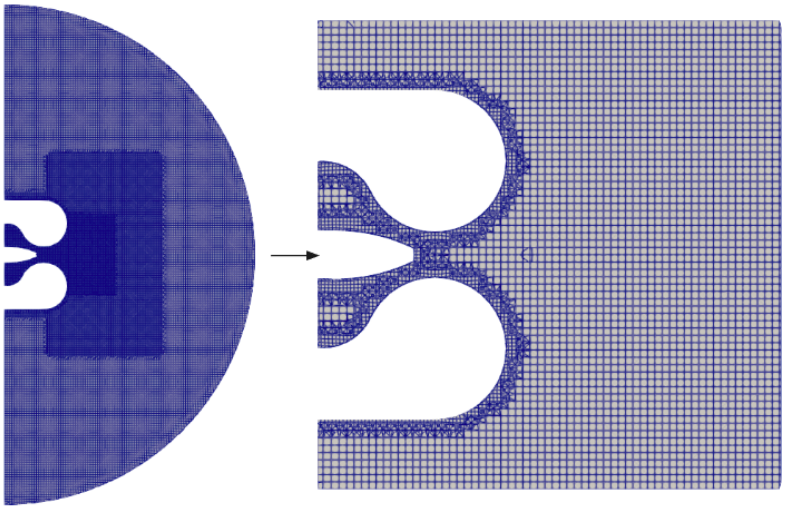

Figure 4.1.: Depiction of the area of focus for deep learning - the original CFD mesh (left) is clipped and transformed into the region of interest (right)

4.1.3. Model inputs and outputs

After data pre-processing, the simulation mesh is considered as a bidirectional graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ where the set of N nodes denoted as $\mathbf{V}$ are linked by the set of edges $\mathbf{E}$ of the mesh. To construct a graph, we need,

- A feature description, consolidated into an $N \times D$ feature matrix X (where N represents the number of nodes, and D denotes the number of input features).
- The graph connectivity or relationships within nodes is represented in matrix form as an adjacency matrix, A or as an edge set E of the shape $2 \times P$, where P is the number of pairs of connected nodes in E.

Let each node have F_X features, and F_Y predictions. The GNN maps the set of node features and edge index matrices to predictions as,

$$\text{GNN} : \mathbb{R}^{N \times F_\text{X}}, \mathbb{W}^{2 \times P} \rightarrow \mathbb{R}^{N \times F_\text{Y}} \tag{4.1}$$

We then get a graph-level output Z of the shape $N \times F_\text{Y}$.
The node feature vector $\mathbf{x}_i$ and prediction vector $\mathbf{y}_i$ of interest at each node v_i is given as,

$$\begin{aligned} \mathbf{x}_i &= [u_{x,i}, u_{y,i}, c_{x,i}, c_{y,i}, \gamma_{\text{tag},i}] \\ \mathbf{y}_i &= [u_{x,i}, u_{y,i}, p_i] \end{aligned} \tag{4.2}$$

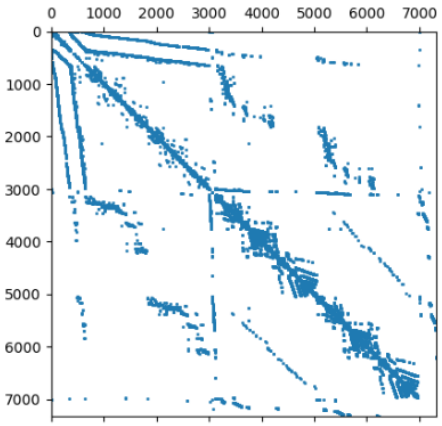

Figure 4.2.: Visualization of the adjacency matrix representing graph connectivity of the clipped CFD mesh with 7329 nodes.

where $u_{x,i}$ and $u_{y,i}$ are the node velocities in X and Y directions, $c_{x,i}$ and $c_{y,i}$ are the spatial co-ordinates of the nodes and p_i is the node pressure. $\gamma_{\text{tag},i}$ is the node tag that defines which cell the node belongs to: inlet, walls or internal mesh. To summarize, our model has 5 input channels (representing node features) and 3 output channels (denoting node predictions). In addition to these channels, the GNN model also requires an edge index matrix to internally compute the adjacency matrix for the graph.

4.1.4. Data normalization

Data normalization is performed on both input channels (node features) and output channels (target vectors), carried out in three steps outlined below.

1. Following common practice, we normalize all the fields of interest with respect to the magnitude of free-stream or reference velocity u_0 to make them dimensionless.

$$\tilde{u} = u/\left\|u_0\right\|, \quad \tilde{p} = p/\left\|u_0\right\|^2 \tag{4.3}$$

   The latter plays a crucial role as it eliminates the quadratic scaling effect present in the pressure values of the target data, effectively flattening the solution space, thereby simplifying the task for the neural network in subsequent stages.

2. Next, we subtract the mean pressure from the dimensionless pressure values.

$$\hat{p} = \tilde{p} - p_{mean}, \quad \text{where } p_{mean} = \sum_i p_i/n \tag{4.4}$$

   n is the number of training samples and p_i denotes individual pressure values. Without this step, the pressure targets depict an ill-posed learning objective since the random pressure offsets in the solutions lack correlation with the inputs.

3. As a final step, every channel undergoes normalization to the range of [-1, 1] (or [0,1]). This standardization aims to mitigate errors stemming from finite numerical precision during the training period. We opt for the maximum absolute value of each quantity across the entire training dataset to normalize the data.

The dataset is split into 3 parts and distributed as Training Data : Validation Data : Test Data in the ratio 80:10:10.

4.2. Graph U-Net

Here, we introduce the Graph U-Net architecture, a foundational framework for the surrogate models used in this work. We analyze the benefits and shortcomings of this model as well as explain the motivation behind developing a modified GNN.

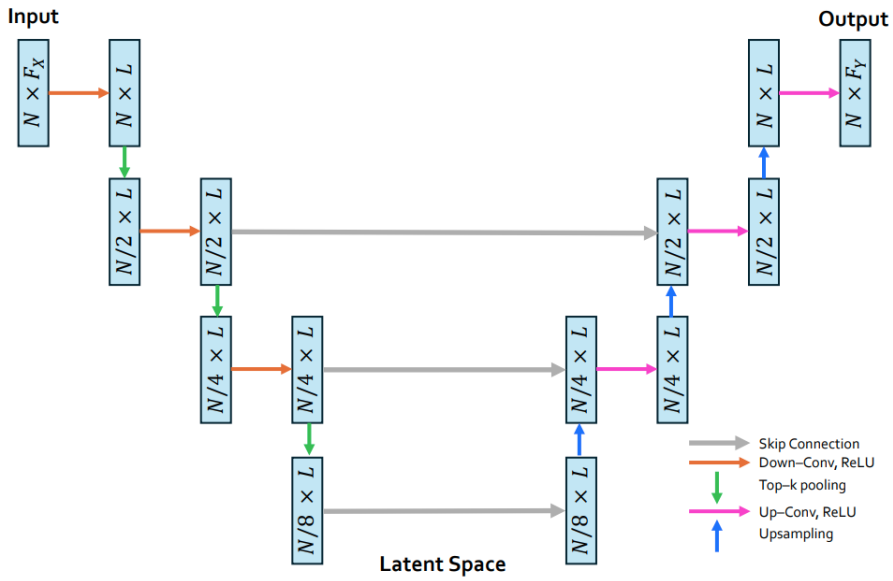

Figure 4.3.: Detailed view of the Graph U-Net architecture applied to fluid dynamics, showcasing an encoder-decoder structure with skip connections. Each layer's application is annotated with the resultant dimensions, illustrating the feature reduction or expansion throughout the network. Here, L is the number of hidden layers, N is the number of nodes in the graph, and F_X and F_Y denote the number of input and output features respectively.

4.2.1. GCNConv layer

GCNs are NNs operating on graph-structured data that extend the concept of convolutional operations from regular grid-like data, such as images, to irregular and non-

Euclidean graph structures, using the spectral graph convolution operation GCNConv. The GCNConv layer aggregates information from neighboring nodes and updates the representations of each node based on this aggregated information, expressed as,

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)} + B^{(l)} h_i^{(l)}\right) \tag{4.5}$$

where $W^{(l)}$ and $B^{(l)}$ are the learnable weight and bias matrices for layer l, σ is the activation function and c_{ij} is a normalization factor. In contrast to filters in CNNs, the weight matrix is consistent and shared across all nodes. However, unlike pixels, nodes do not have a fixed number of neighbors. To maintain uniform value ranges across all nodes and facilitate comparability between them, we normalize the outcomes based on the degree of the nodes, i.e; the number of connections of each node. Hence, $c_{ij} = \sqrt{\text{deg(i)}}\sqrt{\text{deg(j)}}$ where, deg(i) and deg(j) denotes the degree of the nodes v_i and v_j respectively.

4.2.2. Top-k pooling layer

Top k-Pooling selects the k nodes with the highest scores based on a specified criterion. This operation retains the most important nodes in the graph while discarding less relevant nodes, effectively downsampling the graph. It uses a pooling ratio approach, where $k \in (0, 1]$ such that the graph has $\lfloor kN \rfloor$ nodes after the pooling operation. The decision of which nodes to discard is based on a projection score computed against a learnable vector, $\overrightarrow{p}$. Fully expressed, computing a pooled graph, (X^l, A^l), from an input graph, (X, A) can be described using the following algorithm:

1. Compute a score for each node in the graph based on node importance or feature relevance as,
$$\tilde{\mathbf{y}} = \frac{X\overrightarrow{p}}{k||\overrightarrow{p}||_2}$$
2. Select the top k nodes with the highest scores,
$$\overrightarrow{i} = \text{top-k}(\tilde{\mathbf{y}}, k)$$
3. Discard the remaining nodes and their associated edges, resulting in a downsampled graph $G' = (V', E')$ where V' contains only the selected nodes.
4. Retain the features corresponding to the selected nodes to form the downsampled feature and adjacency matrices,
$$X' = (X \odot \tanh(\tilde{\mathbf{y}}))_{\overrightarrow{i}}, \quad A' = A_{\overrightarrow{i},\overrightarrow{i}}$$

Here, $||.||_2$ represents the L_2 norm, top-k selects the top-k indices, $\odot$ denotes element-wise multiplication, and $\cdot_{\overrightarrow{i}}$ represents an indexing operation that extracts slices at indices specified by $\overrightarrow{i}$.

4.2.3. Upsampling

In the Graph U-Net architecture, unpooling is not a distinct operation like in traditional U-Nets. Instead, skip connections are used to implicitly perform unpooling. During decoding, downsampled features from the encoder are combined with zeros or empty features in the decoder using skip connections. This integration effectively restores spatial details and contextual information from the original input graph, ensuring that important features are retained and allowing for the recovery of detailed graph structures. Therefore, unpooling in Graph U-Net is seamlessly integrated into the skip connection mechanism, facilitating the reconstruction of the original graph resolution during decoding.

4.2.4. Results and Discussions

We usually generate two separate CFD datasets, one for the network input and another for the target dataset. However, in this case where the GNN uses the exact architecture of Graph U-Net, we first investigate the ability of the model to reconstruct the target data when the same is given as input. That is, we are interested in the reconstruction of the target dataset without performing a prediction task. We also perform the prediction task on this Graph U-net architecture, which forms the baseline for all the other architectures proposed in this work. The following settings are applied to both the reconstruction and prediction tasks. We implement a 9-fold cross-validation for the training process. The initial learning rate is set to 0.0005 and a Step LR scheduler is used to decay the learning rate by a factor of 0.75 after every 100 epochs. We use the Adam optimizer and train on the Root Mean Squared Error (RMSE) loss for 500 epochs. The model's hyperparameters are selected by a hyperparameter tuning process. Table 4.1 presents the model complexity in terms of total number of trainable parameters. Among the models with similar performance, we choose the one that is the simplest, i.e; has relatively lesser trainable parameters, without compromising on the accuracy or facing the risk of overfitting. Hence, we execute the training process using a GNN architecture with 48 hidden layers and perform the downsampling and upsampling operations 5 times as the depth equals 5. The key parameters of the model as well as the hyperparameters are listed in Table 4.2. The training, validation loss are quantified with the RMSE heuristic as the average over 9 folds after 500 epochs.

Reconstruction of dataset

As mentioned previously, we prescribe the same dataset (simulation results at 30000 timesteps) as the input and target data to evaluate the effectiveness of reconstruction of the Graph U-Net model. The CFD results, predictions of the GNN model and the absolute difference between these data for a simulation case from the test dataset with 18 m/s and 27 m/s as the inlet 1 and inlet 2 velocities respectively are shown in Figure 4.4. We observe that although the GNN predicts the trends of the outflow jet similar to the target data, the

Table 4.1.: Hyperparameter tuning - Table depicting the number of hidden layers, depth, total number of trainable parameters and the training loss measured with the RMSE criterion for the Baseline architecture.

No. of hidden layers	Depth	Total trainable parameters	Training loss (RMSE)
48	2	7k	
	3	12k	
	4	17k	
	5	22k	0.04272
64	2	13k	
	3	21k	
	4	30k	
	5	31k	
128	2	51k	
	3	84k	
	4	117k	
	5	150k	

Table 4.2.: Model hyperparameters

Hyperparameter	Value/Description
Number of hidden layers	48
Network depth	5
Pooling ratios	[0.5,0.5,0.5,0.5,0.5]
Batch size	4
Activation function	ReLU
Weight initialization	Kaiming
Initial learning rate	0.0005

scale or the range of velocity magnitude are slightly different for the two cases. Thus, the baseline model is unable to accurately capture the velocity fields for reconstruction. The poor reconstruction can be attributed to the lack of explicit unpooling layers in Graph U-Net, which can lead to inadequate reconstruction of the original graph structure during decoding.

Prediction of steady-state solutions

In this case, we predict the steady-state solutions from the earlier time step. The CFD results, predictions of the GNN model and the absolute difference between these data for four simulation cases from the test dataset are shown in Figure 4.5. As we can understand from the figure, the predicted direction of the outflow jet as well as its magnitude vary

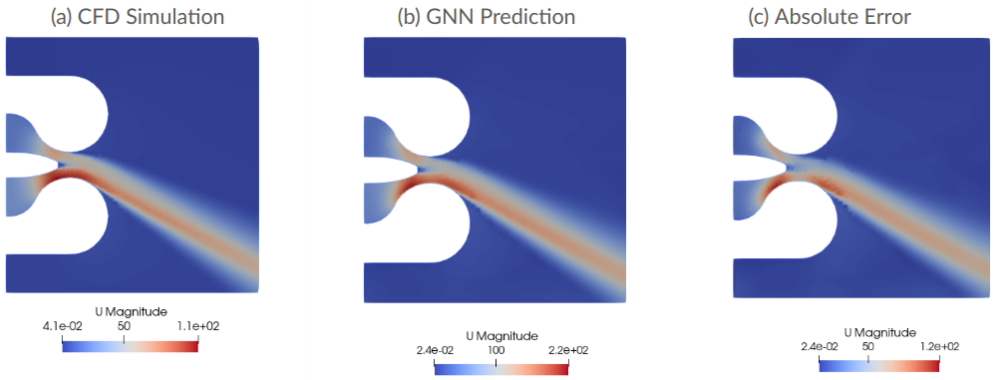

Figure 4.4.: Visualization of velocity fields for a simulation case with Inlet 1 velocity = 18 m/s and Inlet 2 velocity = 27 m/s. Here, (a) represents results from CFD, which is the target data, (b) represents the GNN predictions for the velocity field, and (c) is the absolute difference between the target data and GNN predictions

significantly from the expected CFD target data. Thus, the baseline model is not feasible for the prediction task of nozzle flow dynamics. To better comprehend and evaluate the model performance, we estimate the training, validation and test loss of the baseline model for both the tasks in Table 4.3. Furthermore, we note down the absolute difference between the input and target data for the test dataset of the prediction task.

Table 4.3.: Model evaluation metrics of the Baseline model for the reconstruction and prediction tasks.

Model	**Training Loss**	**Validation Loss**	**Test Loss**	**Abs Difference**
Baseline - Reconstruction	0.04272	0.04340	0.04562	-
Baseline - Prediction	0.1459	0.1503	0.1635	0.2626

Some other limitations to this architecture are,

1. **Limited Scalability:** Originally designed for small graphs with around 100 nodes, Graph U-Net relies on dense matrix multiplications, which are memory-intensive and not scalable. This leads to memory constraints and slower training times, thus making it impractical for complex, large-scale graph data.

2. **Computational Overhead:** Graph U-Net conducts dense adjacency matrix multiplication in the forward pass, resulting in longer training and inference times.

3. **Limited Expressiveness:** GCNConv layers may have limited expressiveness in capturing higher-order graph structures and capturing long-range dependencies in the

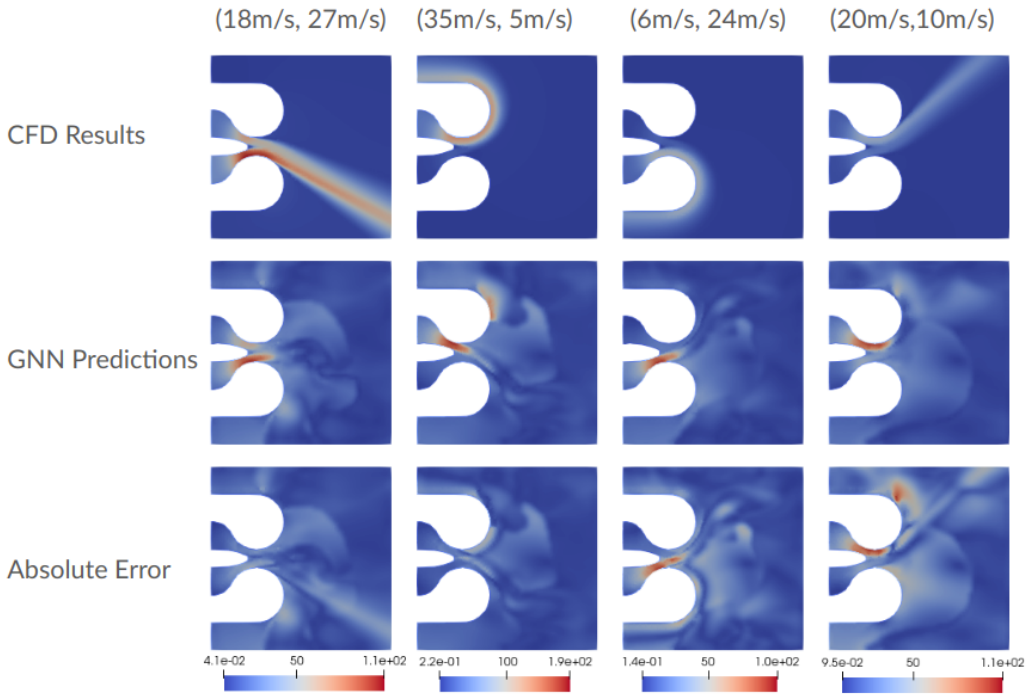

Figure 4.5.: Visualization of velocity fields for four simulation case with inlet velocity values prescribed as (Inlet 1, Inlet 2) presented in four columns. Here, the first row represents results the target data, the second row corresponds to the GNN predictions for the velocity field, and the last row is the absolute difference between the target data and GNN predictions

graph. Its optimal depth is found to be 2 or 3 layers [21]. Deeper models beyond 7 layers can encounter training difficulties due to increased context size per node and heightened risk of overfitting with a larger number of parameters.

Due to these disadvantages, Graph U-Net may exhibit poor performance in terms of both accuracy and efficiency, particularly for complex geometries or large-scale datasets. Hence, there is a paramount necessity to rely on a modified GNN architectures for our work.

4.3. Proposed architecture

In this section, we discuss the architecture of the proposed GNN surrogate model in detail. We elucidate the novel components in the architecture and the adjustments made on the original Graph U-Net framework. Then, we go on to provide comprehensive details on the hyperparameters and other implementation specifics of the proposed GNN model.

Finally, we demonstrate the training process and share the predictions, training and test results obtained for the CFD application. The GNN is developed on the Pytorch deep learning framework using the Pytorch Geometric (PyG) library. Training and testing are performed on V100 GPU compute node using a single GPU (nVidia Tesla V100).
There are 3 different surrogate models proposed, each with a slight variation of the Graph U-Net architecture, namely,

1. *Graphknn:* This model uses the k-NN interpolation technique used in PointNet++ [34] for unpooling or upsampling of features instead of relying on skip connections. The downsampled features at different depths (levels of coarsening) are stored so that the upsampled node co-ordinates required for k-NN interpolation can be obtained from $[c_x, c_y]$ at the downsampled feature of the same depth.

2. *SAGEknn:* In this surrogate, we replace the GCNConv layers of *BL + knn* with GraphSAGE convolutional layers from [13].

3. *MoNetknn:* In this surrogate, we replace the GCNConv layers of *BL + knn* with GMMConv convolutional layers from Mixture Model Network (MoNet) [29].

We perform hyperparameter tuning for each of the architectures to arrive at the ideal choice of design parameters. The key aspects of the three architectures along with their hyperparameters are tabulated in Table 4.4.

Table 4.4.: Features and hyperparameters comparision of 1. *Graphknn*, 2. *SAGEknn*, and 3. *MoNetknn* architectures.

Feature/Hyperparameter	*Graphknn*	*SAGEknn*	*MoNetknn*
Number of hidden layers	48	48	48
Network depth	3	3	3
Convolution layer	GCNConv	SAGEConv	GMMConv
Pooling operation	Top-k	Top-k	Top-k
Unpooling operation	k-NN interpolation	k-NN interpolation	k-NN interpolation
Batch size	4	4	4
Activation function	ReLU	ReLU	ReLU
Weight initialization	Kaiming	Kaiming	Kaiming
Initial learning rate	0.0005	0.0005	0.0005

Similar to the baseline surrogates, we perform 9-fold cross-validation on the training and validation datasets. The optimizer, scheduler, and the loss metric remain the same as the baseline experiments. Figures **??**, **??**, **??** and **??** depict the prediction results and the absolute errors obtained from the three surrogates for four different flow conditions.

To discern the performance of these models, we compute the RMSE loss over the training, validation and test datasets, denoted by Table 4.5.

Table 4.5.: Model evaluation metrics of 1. *Graphknn*, 2. *SAGEknn*, and 3. *MoNetknn* architectures.

Model	**Training Loss**	**Validation Loss**	**Test Loss**	**Abs Difference**
Graphknn	0.134412	0.1431491	0.143952	
SAGEknn	0.123704	0.138665	0.14940	
MoNetknn				

The training and inference times of the three models along with the time taken for CFD simulations are denoted by Table 4.6.

Table 4.6.: Time taken for training and testing of 1. *Graphknn*, 2. *SAGEknn*, 3. *MoNetknn* models and 4. Simulation runtime.

Model	*Graphknn*	*SAGEknn*	*MoNetknn*	Simulation
Training time	0.134412	0.1431491	0.143952	
Test time	0.123704	0.138665	0.14940	

5. Clustering

Part IV.

Conclusion

6. Conclusion

This thesis has effectively demonstrated the viability and efficiency of employing Graph Neural Networks (GNNs) as surrogate models for the prediction of stable, steady-state nozzle flow dynamics, marking a significant advancement in the field of Computational Fluid Dynamics (CFD). Through a methodical approach that leverages transient simulation data, this research has offered a groundbreaking alternative to traditional CFD simulations, which are characterized by their computational intensity and extensive time requirements. The findings from this study provide a concrete foundation for the assertion that GNNs can serve as powerful tools in the simulation and analysis of fluid dynamics, specifically in applications related to nozzle flows.

6.1. Contributions

1. Development of a GNN-Based Surrogate Model: The research successfully developed and optimized a GNN model capable of predicting the velocity and pressure fields in nozzle flow simulations from early, transient states. This surrogate model has proven to effectively use short-term, less computationally demanding simulation results to forecast stable, steady-state flow conditions with remarkable accuracy. This achievement not only demonstrates the technical feasibility of GNNs in fluid dynamics but also positions them as a cornerstone for future CFD simulation methodologies.

2. Comparative Analysis of Model Efficiency: A thorough investigation into the efficiency and practicality of the surrogate model was conducted, comparing its performance against traditional, long-duration CFD simulations. The GNN model exhibited substantial computational savings while maintaining a high level of prediction accuracy. This comparative analysis provided empirical evidence supporting the surrogate model's capability to serve as a viable and efficient alternative to conventional CFD methods.

3. Innovative Application of Clustering Techniques: The thesis explored the use of clustering on low-dimensional data to classify nozzle flow simulations effectively. This approach allowed for an enhanced understanding of the simulation data, leading to more precise predictions of Coanda effect occurrences and facilitating better design and optimization of nozzle configurations. This contribution highlights the potential of integrating advanced data analysis techniques with DL models to improve

simulation outcomes.

4. Exploration of Advanced GNN Architectures: The investigation into advanced GNN architectures and training methodologies underscored the potential for further enhancements in model performance. This exploration revealed that incorporating different graph convolutional layers, attention mechanisms, and training strategies could significantly refine the surrogate model's predictive accuracy and efficiency, paving the way for future advancements in the field.

6.2. Future directions

The research presented in this thesis unequivocally asserts the transformative potential of GNNs as surrogate models in the domain of CFD. By achieving significant computational efficiency and maintaining high levels of accuracy in predicting nozzle flow dynamics, this study sets a new benchmark for future research in fluid dynamics simulations. Looking forward, it is imperative to expand upon this work by exploring more complex nozzle configurations, integrating multiscale modeling approaches, and further refining GNN architectures to enhance their generalizability and predictive capabilities across a broader spectrum of fluid dynamics applications. Some possible directions of future research could be,

1. **Enhancement of GNN architecture for improved accuracy:** One of the primary objectives moving forward is the development of an improved GNN architecture that enhances prediction accuracy. This involves exploring novel graph convolutional layers, attention mechanisms, and network structures that can more effectively capture the complexities of fluid dynamics. Advanced neural network designs that specifically address the challenges of turbulent flow simulations will be critical in achieving higher levels of accuracy in predicting nozzle flow behavior.

2. **Prediction of extended flow quantities of interest:** An additional critical objective for future work is the extension of the GNN model's capabilities to predict a broader array of flow quantities that are crucial for a comprehensive understanding of fluid dynamics. Specifically, the model will be tailored to accurately forecast values such as turbulent viscosity ν_t,turbulent kinetic energy k, and the specific rate of dissipation . These quantities are fundamental in the analysis and modeling of turbulent flows, providing deeper insights into the behavior of fluids in various engineering applications.

3. **Optimization of model training times:** Another crucial area of research is the optimization of the GNN model to reduce training times without compromising accuracy. Techniques such as pruning, quantization, and knowledge distillation could be employed to streamline the model, making it more computationally efficient. Additionally, leveraging parallel computing and advanced hardware accelerators will

be explored to further enhance training efficiency, enabling the model to learn from larger datasets in shorter time frames.

4. **Development of Physics-Informed GNNs:** Incorporating governing equations of fluid dynamics into the model's loss function presents a promising approach to enhance prediction accuracy. By developing a Physics-Informed Graph Neural Network (PI-GNN), the model can leverage both data-driven learning and the inherent physics of fluid flow, ensuring more accurate and physically plausible predictions. This approach not only aids in better capturing the complex phenomena associated with nozzle flows but also in improving the model's interpretability and reliability.

5. **Transfer learning and model generalization:** Future research will also focus on enhancing the model's generalization capabilities through transfer learning. This involves training the GNN model on a diverse set of nozzle flow scenarios and then fine-tuning it for specific applications, enabling the model to adapt to different nozzle geometries and flow boundary conditions effectively. The goal is to validate the model across a wide range of nozzle configurations and operating conditions, ensuring its applicability and robustness in varied fluid dynamics simulations.

In sum, this thesis not only fulfills its objectives but also opens new avenues for research, underscoring the pivotal role of DL in advancing CFD towards more efficient and accurate simulations. The technical achievements documented herein confirm the feasibility of GNNs as powerful surrogate models, signifying a major step forward in our ability to simulate and understand complex fluid dynamics phenomena with unprecedented efficiency and precision.

Appendix

List of Acronyms

CFD Computational Fluid Dynamics. 5

CNN Convolutional Neural Network. 28

DL Deep Learning. 5

GNN Graph Neural Network. 5

MoNet Mixture Model Network. 48

NSE Navier-Stokes Equations. 3, 11, 12

RANS Reynolds Averaged Navier-Stokes. 39

RMSE Root Mean Squared Error. 44

List of Symbols

ν_t turbulent viscosity. 14, 54

Bibliography

[1] Saleh Albelwi and Ausif Mahmood. A framework for designing the architectures of deep convolutional neural networks. *Entropy*, 19(6), 2017.

[2] Peter Benner, Serkan Gugercin, and Karen Willcox. A survey of projection-based model reduction methods for parametric dynamical systems. *SIAM Review*, 57(4):483–531, 2015.

[3] Gal Berkooz, Philip Holmes, and John L. Lumley. The proper orthogonal decomposition in the analysis of turbulent flows. *Annual Review of Fluid Mechanics*, 25(1):539–575, 1993.

[4] Léon Bottou. Large scale online learning. *Advances in neural information processing systems*, 22:217–224, 2010.

[5] Michael P. Brenner, Jeff D. Eldredge, and Jonathan B. Freund. Perspective on machine learning for advancing fluid mechanics. *Physical Review Fluids*, 4(10):100501, 2019.

[6] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. Machine learning for fluid mechanics. *Annual Review of Fluid Mechanics*, 52(1):477–508, 2020.

[7] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *Journal of Machine Learning Research*, volume 12, pages 2121–2159, 2011.

[8] Charles L. Fefferman. Existence and smoothness of the navier-stokes equation. *The Clay Mathematics Institute*, 2006. Millennium Prize Problems.

[9] Joel H. Ferziger and Milovan Perić. *Computational Methods for Fluid Dynamics*. Springer, 2002.

[10] FreeCAD Community. Freecad 0.19: An open source parametric 3d cad modeler, 2021. Accessed: 22.12.2023.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[12] Luca Guastoni, Alejandro Güemes, Andrea Ianiro, Stefano Discetti, Philipp Schlatter, Hossein Azizpour, and Ricardo Vinuesa. Convolutional-network models to predict wall-bounded turbulence from wall quantities. *Journal of Fluid Mechanics*, 928, 2021.

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[14] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2 edition, 2009. Discussion on k-fold cross-validation.

[15] Juncai He and Jinchao Xu. Mgnet: A unified framework of multigrid and convolutional neural network. *Science China Mathematics*, 62(7):1331–1354, May 2019.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. 2015.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[18] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[19] Emre Kara and Hüdai Erpulat. Experimental investigation and numerical verification of coanda effect on curved surfaces using co-flow thrust vectoring. *International Advanced Researches and Engineering Journal*, 5:72–78, 2021.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[22] J. Nathan Kutz. Deep learning in fluid dynamics. *Journal of Fluid Mechanics*, 814:1–4, 2017.

[23] B. E. Launder and D. B. Spalding. The numerical computation of turbulent flows. *Computer Methods in Applied Mechanics and Engineering*, 3(2):269–289, 1974.

[24] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[25] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807:155–166, 2016.

[26] Wenzhuo Liu, Mouadh Yagoubi, and Marc Schoenauer. Meta-learning for airflow simulations with graph neural networks. 2023.

[27] Michele Milano and Petros Koumoutsakos. Neural network modeling for near wall turbulent flow. *Journal of Computational Physics*, 182(1):1–26, 2002.

[28] Parviz Moin and Krishnan Mahesh. *Direct Numerical Simulation: A Tool in Turbulence Research*. Annual Review of Fluid Mechanics, Annual Reviews, 1998.

[29] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. *CoRR*, abs/1611.08402, 2016.

[30] BG Newman. The deflection of plane jets by adjacent boundaries-coanda effect. *Boundary layer and flow control*, 1961.

[31] Francis Ogoke, Kazem Meidani, Amirreza Hashemi, and Amir Barati Farimani. Graph convolutional networks applied to unstructured flow field data. *Machine Learning: Science and Technology*, 2(4):045020, sep 2021.

[32] Eric J. Parish and Karthik Duraisamy. A paradigm for data-driven predictive modeling using field inversion and machine learning. *Journal of Computational Physics*, 305:758–774, 2016.

[33] Stephen B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[34] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. 2017.

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

[36] O. Reynolds. On the dynamical theory of incompressible viscous fluids and the determination of the criterion. *Philosophical Transactions of the Royal Society of London. A*, 186:123–164, 1895.

[37] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. 2015.

[38] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, 1986.

[39] Peter J. Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28, 2010.

[40] Joseph Smagorinsky. General circulation experiments with the primitive equations: I. the basic experiment. *Monthly Weather Review*, 91(3):99–164, 1963.

[41] Joe F. Thompson, Bharat K. Soni, and Nigel P. Weatherill. Automatic and adaptive mesh generation for curved domains. *Applied Mechanics Reviews*, 51(11):669–704, 1998.

[42] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 2012. Available online: https://www.coursera.org/course/neuralnets.

[43] Brendan D. Tracey, Karthikeyan Duraisamy, and Juan J. Alonso. *A Machine Learning Strategy to Assist Turbulence Model Development*.

[44] Michele Trancossi and Antonio Dumas. A.c.h.e.o.n.: Aerial coanda high efficiency orienting-jet nozzle. *SAE Technical Papers*, 10 2011.

[45] Nathaniel Trask, Ravi G. Patel, Ben J. Gross, and Paul J. Atzberger. Gmls-nets: A framework for learning from unstructured data. 2019.

[46] Ze Jia Zhang and Karthikeyan Duraisamy. Machine learning methods for data-driven turbulence modeling.