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, 2023

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, 2023 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 plays a pivotal role in the progress of various diverse fields, including aerospace and automotive engineering, energy system optimization, environmental modeling, biomedical research, and countless other critical domains that shape our technological and scientific landscape. Partial Differential Equations (PDEs), in particular the Navier-Stokes equations, which provide the mathematical framework for analyzing the behaviour of fluid flow for these complex problems are often intractable, i.e; they do not have exact analytical solutions. Thus, they still remain an open problem in the world of mathematics. Hence, the solutions to the Navier-Stokes equations are approximated using numerical methods such as the Finite Element or the more commonly used Finite Volume Method (FVM) which rely on spatial and temporal discretization of PDEs. With the advent of software and technology, emerged the field of Computational Fluid Dynamics (CFD), leveraging computer algorithms for numerical simulations of fluid flow problems. However, these simulations are computationally intensive and may take from several hours to several weeks for complex flows and/or geometries. Although advances in high-performance computing and parallel processing have been a game-changer for CFD, there still remain several complications with respect to grid dependency, convergence issues, model assumptions and simplification of underlying physics, numerical errors as well as turbulence modelling.
The most problematic use-case is turbulence modelling, which is marked by velocity and pressure fluctuations, a wide range of length and time scales - from large vortices down to small eddies and numerical instabilities. In addition, modelling turbulence near walls presents unique problems on its own which requires specialized treatments and techniques in these near-wall regions. Accurately modelling all the intricate phenomena of the turbulent regime, called the Direct Numerical Simulation (DNS), is possible, but at the cost of a huge amount of computational resources, complexity and simulation times. Consequently, in many practical scenarios, simplified turbulence models are employed, even though this comes at the expense of accuracy.
Advancements in the dynamic field of CFD over the past had seen substantial efforts channeled into enhancing turbulence models, improving meshing techniques, efficient Reduced Ordered Modelling (ROM) surrogates and reducing computational complexity. While numerous techniques and models exist to address various turbulence-related challenges, there is no universally applicable model that consistently delivers accurate results across all turbulence settings. As a result, the computational bottleneck often remains an unavoidable obstacle. In response to this challenge, in recent times, there has been a growing interest in implementing Machine Learning (ML) algorithms, specifically to en-

hance the cost-effectiveness of CFD methods. Although training an ML model is often very computationally demanding, a trained model can quickly make predictions on new data, a significant advantage over the commonly used numerical methods. The integration of ML in CFD represents a paradigm shift, enabling simulations to move beyond traditional physics-based models. The synergy between Deep Learning (DL) and CFD offers the potential to uncover novel insights into fluid dynamics, for example, Convolutional Neural Networks (CNNs) can extract relevant features and classify flow regimes whereas Recurrent Neural Networks (RNNs) can predict fluid behaviors based on temporal data for unsteady flows. Incorporating neural networks into fluid dynamics problems not only improves the accuracy and efficiency of simulations, but also opens up new doors for understanding flow behaviours and strategies for flow control and optimization. As neural networks continue to evolve and adapt to the specific challenges of fluid dynamics, they promise to be a pivotal component in the advancement of CFD across various industries and applications.
So far, ML methods have been employed in fluid dynamics research but have not been in engineering practice. Some possible explanations for this might be the scarcity of huge datasets that are open and can be publicly accessed and the poor generalization performance of ML techniques on previously unseen data. This thesis attempts to tackle the generalization problem by implementing a novel approach that combines GNNs with a meta learning principle for accurate and efficient predictions even on unencountered datasets.

1.1. Literature Review

In this section, relevant research and work done with respect to ML in the context of CFD, especially in turbulence modelling is discussed. The evolution of turbulence modeling is a dynamic field with notable milestones and contributions. The pioneering work of Osborne Reynolds in the late 19th century laid the foundation for turbulence research, leading to the eddy viscosity hypothesis and the rise of Reynolds-averaged Navier-Stokes (RANS) modeling in the mid-20th century. The K-epsilon model, introduced in the 1970s, significantly enhanced RANS modeling by striking a balance between accuracy and computational efficiency. Parallel to this development, Large Eddy Simulation (LES) emerged as a complementary approach, aiming to capture large-scale turbulent eddies directly while modeling smaller ones. Advances in computational power enabled the feasibility of LES, particularly for complex geometries and higher Reynolds numbers. Direct Numerical Simulation (DNS), which resolves all turbulent scales, became increasingly attainable due to further advancements in computational resources. Recent research has focused on improving existing models, developing advanced closure schemes, and exploring the use of machine learning techniques. The combination of RANS and LES techniques in hybrid models has also gained traction, offering a promising path towards more efficient and accurate simulations.
More recently, machine learning and deep learning-based turbulence modeling are gain-

ing traction. ML algorithms have been employed to develop surrogate models, which are reduced-order representations of complex turbulence systems. These models are significantly faster to evaluate than full Navier-Stokes simulations, making them suitable for real-time applications and design optimization. Additionally, ML has been used to create data-driven closure terms for RANS and LES models, enhancing the accuracy of turbulence simulations. DL techniques, particularly CNNs and RNNs, have demonstrated remarkable performance in turbulence modeling. These neural networks can effectively learn complex patterns and relationships in turbulent data, enabling the prediction of turbulence quantities from limited input data as well as for non-equilibrium flows and multiphase flows.
It wasn't until the early 2010s that researchers and engineers began exploring the potential of applying machine learning techniques to problems in CFD. One of the earliest works in this field is that of Brunton (cite!!!), who provided a comprehensive review of ML methods applied to fluid mechanics and classifies them into three major categories - supervised, semi-supervised and unsupervised. In supervised learning, the model is trained on labeled data, where the input (features) and the corresponding output (target) are known. Supervised machine learning algorithms can be categorized into classification and regression approaches. Classification algorithms are used to predict categorical outcomes, while regression algorithms are used to predict continuous outcomes. In the context of turbulence modeling, regression algorithms are particularly valuable, as they can be used to predict turbulence quantities such as turbulent viscosity and eddy dissipation rate. Random forests, support vector machines, and neural networks are among the commonly used supervised learning techniques, applicable to both classification and regression tasks. Semi-supervised learning blends aspects of both supervised and unsupervised machine learning. It has been effectively used in tasks involving time-series data and image processing. Unsupervised learning involves training on unlabeled data to discover patterns, groupings, or structures within the data and unlike supervised learning there are no specified prediction targets. Techniques like Reduced Order Modelling (ROM), Proper Orthogonal Decomposition (POD), dimensionality reduction and Clustering fall under unsupervised learning and they mark the precursors of data-driven methods in fluid mechanics.
Another way of classifying ML techniques would be based on their optimization objective. The commonly used Data Driven approaches try to minimize the error between the predictions and target solutions by training available data in a supervised manner without explicitly considering physical laws. Data-driven flow solvers are typically tailored and trained for specific target applications to ensure consistent and high accuracy across cases. While these approaches reduce inference time, the computational burden shifts to a pre-processing phase, involving data generation and model parameter tuning. Physics Informed Neural Networks (PINNs) make use of prior knowledge, for example mathematical models and equations that govern the data and enforce physical intuition for training the model in an unsupervised manner. PINNs can make accurate predictions with limited data, and train models in a faster and computationally efficient way for physics-based problems.

Several intriguing research initiatives in this field will be explored in this section. One of the earliest recognized efforts of using ML methods in turbulence modelling is the work of Zhang and Duraisamy(). In 2015, Duraisamy et al. (cite !!!!) proposed a data-driven method for turbulence closure modelling in which an inverse problem, i.e; Multi-scale Gaussian process regression learns adjustable spatio-temporal term(s) of the RANS equations which are then incorporated into the RANS equations and reconstructed into a functional form. This adjusted turbulence model is then used for predictive purposes. The paper by Ling, Kurzawski and Templeton (cite!!!!) introduces deep neural networks to enhance Reynolds-averaged Navier–Stokes turbulence models by embedding Galilean invariance, leading to improved accuracy in predicting Reynolds stress anisotropy and enhanced performance in velocity field predictions compared to traditional models.
Guastoni et al. (2020) proposed a fully-convolutional neural network model to predict streamwise velocity fields in turbulent open channel flow. The model takes streamwise and spanwise wall-shear stress planes as input and is trained using DNS.Taking into account the successful outcomes of this research, Guastoni et al. (2021) compares two models trained using data from DNS to predict the instantaneous velocity fluctuations in a turbulent open-channel flow. The first model, a fully convolutional neural network (FCN), directly predicts the fluctuations, while the second model, known as FCN-POD, reconstructs the flow fields using a linear combination of orthonormal basis functions obtained through POD. CNNs are particularly useful for flow field reconstruction and denoising noisy data generated from numerical simulations. Zhang et al. () to predict the lift coefficients of airfoils by analyzing images of the airfoils and their surrounding flows. These images encode flow conditions, such as Mach number, as pixel intensities, allowing the CNN to learn the intricate relationship between airfoil geometry and flow behavior. Viquerat and Hachem ()developed a CNN optimized for estimating drag coefficients of various 2D geometries in laminar flow, trained on a comprehensive dataset of random shapes and their corresponding drag forces. This approach significantly improved drag coefficient predictions for real-world geometries like NACA airfoils. Yilmaz and German () applied a CNN to directly predict airfoil performance based solely on airfoil shape, eliminating the need for time-consuming surrogate modeling techniques requiring manual parameter fitting. Guo et al. trained a deep CNN to generate rapid, albeit less accurate, visual approximations of steady-state flow around 2D objects, streamlining the design process.
In CFD simulations, researchers often encounter unstructured mesh data, particularly when dealing with curved or complex geometries. Traditional CNNs, however, are designed for structured grid data, such as images, where the arrangement of data points is regular and grid-like. This limitation prevents CNNs from being directly applied to unstructured mesh flow field data. To address this issue, researchers have explored the utilization of mesh-free techniques and graph-based representations of fluid data. Recent advancements in manipulating unstructured data have led to the development of mesh-free inference methods for point cloud representations and reduced-order models based on graph-based representations of fluid data. Graph theory-based methods have been successfully employed to identify coherent structures within turbulent flow. Hadjighasem et

al. proposed a heuristic based on the generalized eigenvalue problem of the graph Laplacian to determine the locations of coherent structures. This work was extended by Meena et al., who constructed a graph to represent the mutual interaction of individual vortex elements and identified larger vortex communities using network theory-based community detection algorithms. Trask et al. introduced the concept of GMLS-Nets, which parameterize the generalized moving least-squares functional regression technique for application on mesh-free, unstructured data. They demonstrated the effectiveness of GMLS-Nets for uncovering operators governing the dynamics of partial differential equations and predicting body forces associated with flow around a cylinder based on point measurements. Ogoke et al. () implemented a novel approach for using graph convolutional networks (GCNNs) to predict the drag force associated with laminar flow around airfoils from scattered velocity measurements. GNNs can also be integrated with meta-learning techniques to adapt and improve turbulence models based on specific simulation conditions. This enhances model performance and robustness, particularly in cases with limited training data. The work of Liu et al 2023 uses a meta-learning approach for improving the out-of-distribution (OoD) generalization performance of data-driven models in airflow simulations using graph neural networks (GNNs). In the light of promising results obtained from GNN-based architectures for data-driven turbulence modelling, a GNN-based surrogate model is proposed to understand the various parameters affecting the outcome of the turbulent flow in the context of a High-speed Orienting Momentum with Enhanced Reversibility (HOMER) nozzle, developed by Michele Trancossi.(cite!!)

1.2. Scope and Objectives

Why Graph Neural Networks?

1. Poor generalization by other ML methods 2. CNN required problems to be converted to images and the output was also images. GNN uses the underlying structure of CFD meshes 3. Irregular meshing is also possible whereas CNN could only operate on regular grid.

The intersection of machine learning and CFD remains an active area of research, with ongoing work in improving the accuracy and efficiency of CFD simulations, and advancements in physics-informed models for its application to a wider range of real-world problems.

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 developed by Trancossi and Dumas [5]. We begin with an understanding of the problem and the underlying principle, which is the Coanda effect. Then, 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, 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 ([5]). 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, the adhesion effect, and the attraction effect. Newman [3] has demonstrated that Coanda

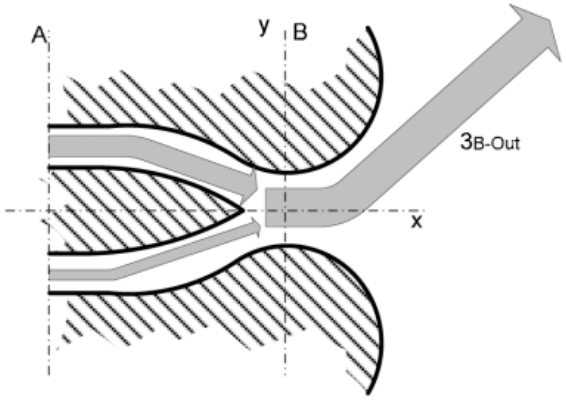

Figure 2.1.: HOMER nozzle

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.

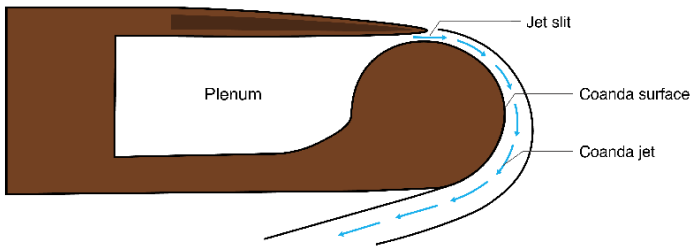

Figure 2.2.: Coanda Effect

2.2. Governing equations

In this section, we talk about the mathematical equations that govern the fluid flow in the nozzle setup. The Navier-Stokes equations (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.

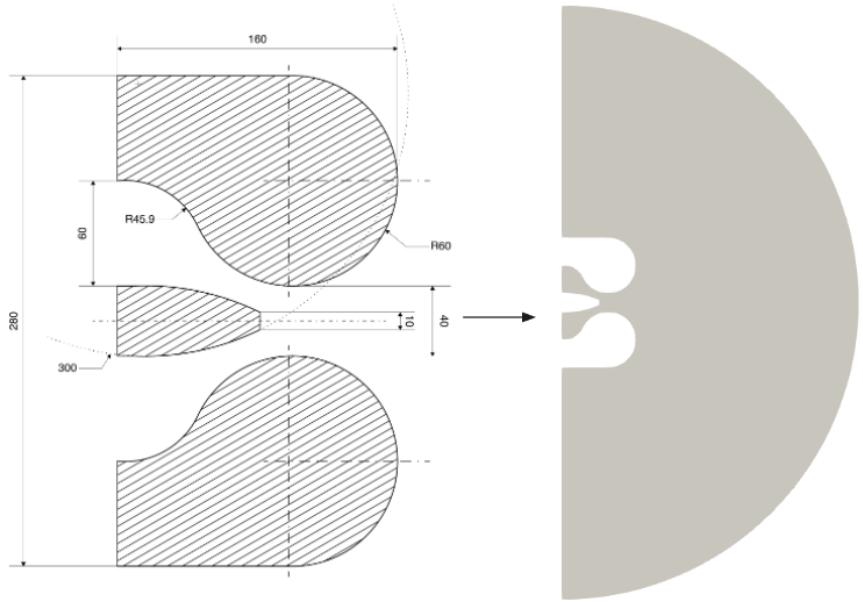

Figure 2.3.: Computational Flow Domain

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

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 dissipative 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 [5] 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 [1]. 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 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. A 3D, unstructured, 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

An incompressible, steady-state CFD simulation is carried out on the mesh by setting appropriate boundary and initial conditions. For k, ν_t, and ω, low Reynolds number wall functions are applied at walls to account for near-wall turbulence effects. This ensures accurate modeling of turbulence near boundaries. Pressure is set to `zeroGradient` at walls to maintain a neutral pressure condition. Adiabatic stationary walls with no slip conditions are prescribed at the Coanda surfaces as well as the inner walls of the nozzle. Inlets are prescribed with fixed values of velocities to define the flow entering the domain. Both the inlet turbulence intensities are set to 1% (medium turbulence). Additionally, a pressure outlet boundary condition is used to specify the pressure at outlets, allowing flow to exit the domain without reflecting back. These boundary conditions collectively ensure proper representation of the flow behavior within the computational domain. The simulations are performed using the `simpleFoam` solver which utilizes the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm for pressure-velocity coupling. This method iteratively resolves the momentum and pressure equations until a predetermined convergence criterion is satisfied. The $k-\omega$ SST turbulence model is utilized, and the kinematic viscosity of the fluid is taken as $1.51e^{-5}$. The convective term of velocity is discretized using the linear UpwindV scheme whereas the divergence term of velocity is discretized using a bounded Gauss linear scheme. For the discretization of the divergence of the turbulence fields k, Ω, ν_T, a bounded Gauss upwind scheme is used. In this case, the convergence criterion was set to $10e^{-6}$ for the pressure field and $10e^{-8}$ 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. Subsequently, it delves into optimization techniques, addressing essential elements such as loss functions, backpropagation, learning rates, and optimizers. Furthermore, it navigates through the intricacies of training neural networks with topics including data partitioning, feature scaling, weights initialization, regularization, batch training, and hyperparameter tuning. Additionally, this chapter examines model evaluation metrics and explores advanced neural network architectures such as Convolutional Neural Networks (CNNs) and Graph Neural Networks (GNNs), highlighting their important features and components.

3.1. Introduction to machine learning and deep learning

Machine learning is a branch of artificial intelligence that focuses on developing algorithms capable of learning from data and improving their performance over time. 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, in contrast, 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. The key distinction between machine learning and deep learning lies in the depth and complexity of the models. We will exclusively delve into the specifics of supervised learning in this work. Henceforth, when we mention machine learning or deep learning, it refers to supervised learning in this context.

3.2. Fundamentals of neural networks

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of biological neural networks. 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, inspired by biological neurons in the brain, 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

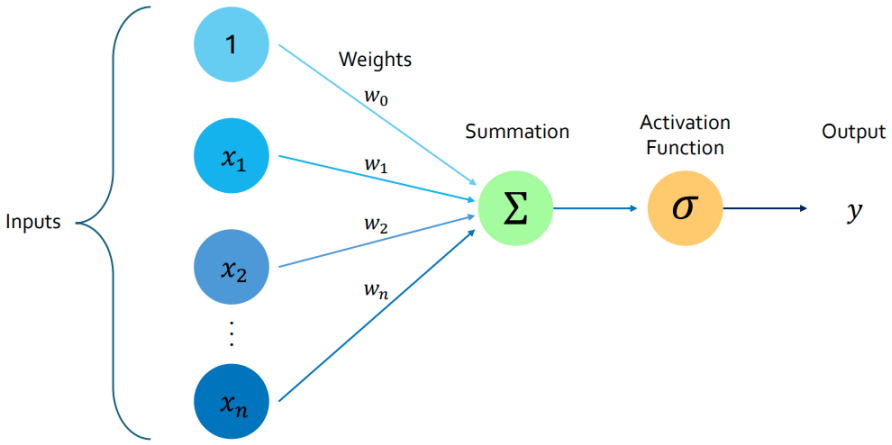

Figure 3.1.: Perceptron

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

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. Based on the structure of this connection graph, neural networks can be broadly classified into feed-forward neural networks and recurrent neural networks.
In **feed-forward neural networks**, 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 3.2. **Recurrent neural networks**,

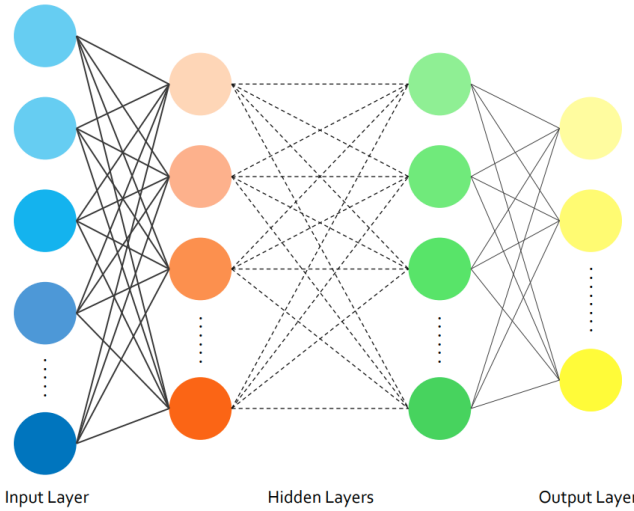

Figure 3.2.: Architecture of a Multi Layer Perceptron

on the other hand, are designed to handle sequential data by allowing connections between neurons to form directed cycles. The feedback loops allow the network to maintain a memory of past inputs, making them suitable for tasks involving sequential data.

3.3. 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.

3.3.1. 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:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, \hat{y}_i; \theta\right) \tag{3.1}$$

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.

3.3.2. 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,

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

3.3.3. 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:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta) \tag{3.3}$$

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.

3.3.4. 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), Adam, RMSProp, and AdaGrad.

1. **Stochastic Gradient Descent (SGD)** performs parameter updates using a fixed learning rate, which can lead to oscillations or slow convergence in the presence of sparse or noisy gradients.
2. **RMSProp (Root Mean Square Propagation)** adapts the learning rate for each parameter based on the magnitudes of recent gradients. It maintains a moving average of squared gradients for each parameter and divides the learning rate by the square root of this average to scale the parameter updates. RMSProp gets rid of the oscillations and slow convergence associated with fixed learning rates in SGD.
3. **AdaGrad (Adaptive Gradient)** algorithm adapts the learning rate for each parameter based on the history of gradients accumulated during training. It gives larger updates to parameters with smaller gradients, and vice versa. AdaGrad is useful for sparse data or problems with varying feature scales, as it automatically adjusts the learning rates for different parameters.
4. **Adam (Adaptive Moment Estimation)** maintains exponentially decaying moving averages of past gradients and past squared gradients for each parameter. Adam also incorporates bias correction terms to compensate for the initial bias towards zero at the beginning of training. The parameter update rule for Adam is given by

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t \tag{3.4}$$

   where, $\hat{m}_t$ is the bias-corrected estimate of the first moment (mean) of the gradients, $\hat{v}_t$ is the bias-corrected estimate of the second moment (uncentered variance) of the gradients, and ϵ is a small constant to prevent division by zero.

3.4. Training of neural networks

3.4.1. 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.

3.4.2. 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: $\mathrm{x}' = \frac{\mathrm{x}}{\|\mathrm{x}\|}$

3.4.3. 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 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 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.

3.4.4. 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 L1 regularization (Lasso) and L2 regularization (Ridge).
L1 regularization encourages sparsity in the weights, performing feature selection by setting irrelevant weights to zero, making the model simpler and more interpretable.

$$\text{Loss}_{\text{L1}} = \text{Loss}_{\text{original}} + \lambda \sum_{i=1}^{n} |w_i|$$

L2 regularization encourages the weights to be spread out more evenly, preventing individual weights from becoming too large.

$$\text{Loss}_{\text{L2}} = \text{Loss}_{\text{original}} + \lambda \sum_{i=1}^{n} w_i^2$$

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.

3.4.5. 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 (BatchNorm) is a technique to improve the stability and performance 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,

$$\begin{aligned} \mu_B &= \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \\ \sigma_B^2 &= \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu_B\right)^2 \\ \hat{x}^{(i)} &= \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ y^{(i)} &= \gamma\hat{x}^{(i)} + \beta \end{aligned} \tag{3.5}$$

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.

3.4.6. 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.

3.4.7. 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, 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.

3.5. 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.

3.6. Convolutional Neural Networks (CNNs)

CNNs 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.

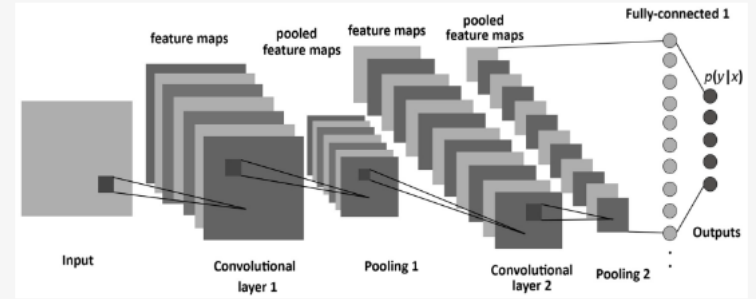

Figure 3.3.: Convolutional Neural Network

3.6.1. Convolutional layer

The convolutional layer in a 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 3.4 shows the working of a convolution kernel.

3.6.2. 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

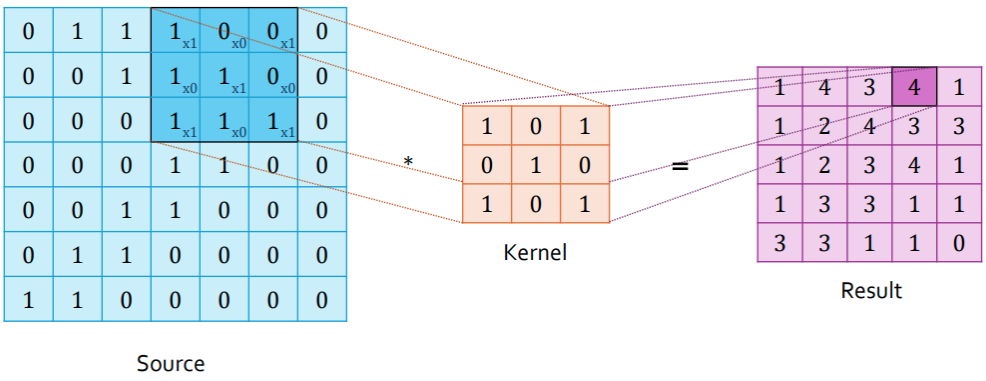

Figure 3.4.: Convolutional Layer

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 3.5. 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 3.6.

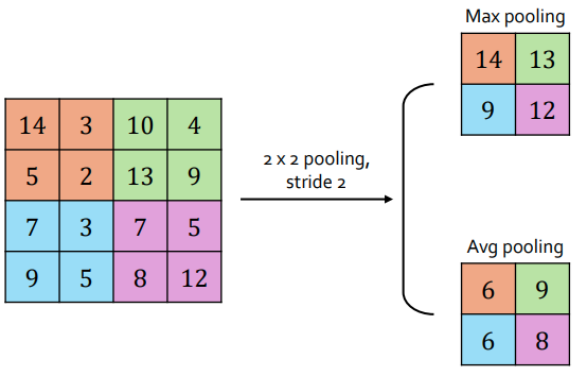

Figure 3.5.: Pooling

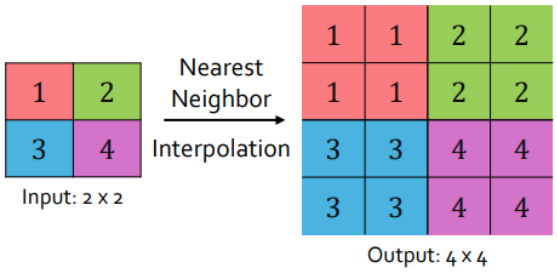

Figure 3.6.: Unpooling

3.6.3. The U-Net architecture

U-Net 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.

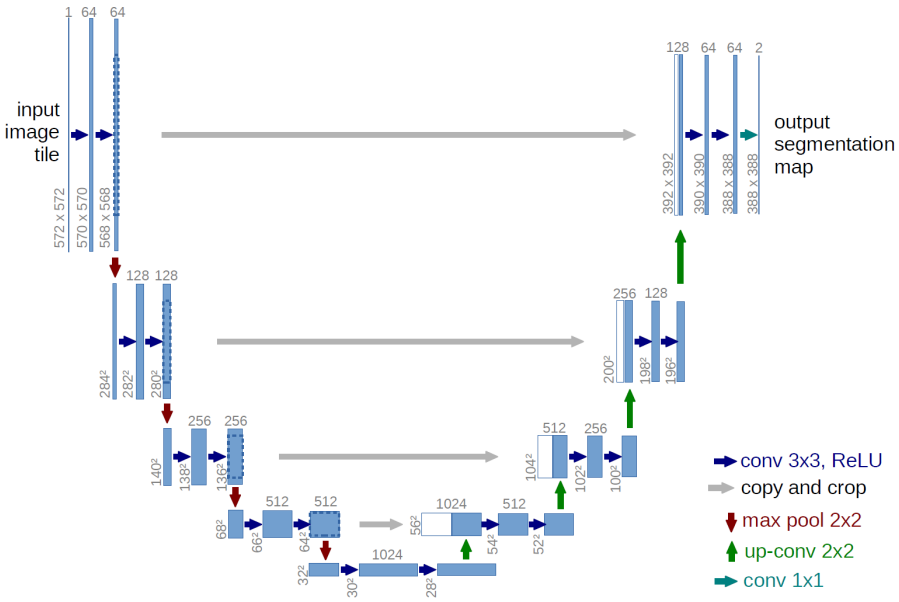

Figure 3.7.: U-Net Architecture

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, 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.

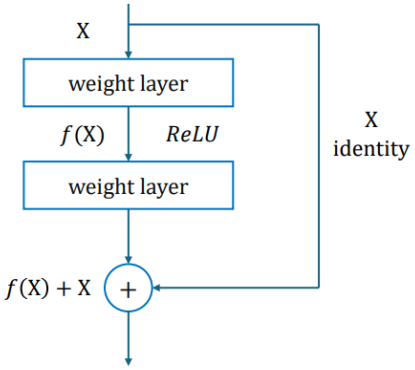

Figure 3.8.: Skip Connection

A notable observation pertinent to our current endeavor is the striking resemblance between the U-Net architecture and the V-cycle multi-grid method. Both employ a hierarchical structure wherein information is exchanged across varying resolutions.

3.7. 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 [2] 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. 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,

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

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.

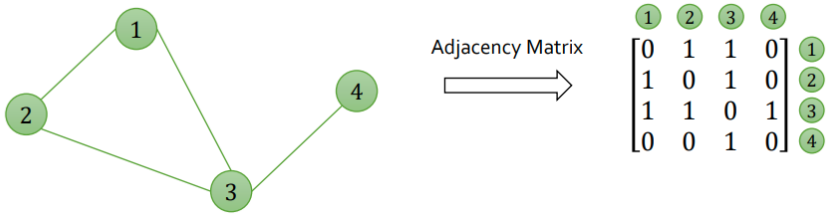

Figure 3.9.: Adjacency Matrix

3.7.1. 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 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.

3.7.2. 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.

3.7.3. 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++ [4]. 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,

$$\mathbf{f}(\mathbf{z}) = \frac{\sum_{i=1}^{k} w\left(x_i\right) \mathbf{f}\left(x_i\right)}{\sum_{i=1}^{k} w\left(x_i\right)}, \text{ where } w\left(x_i\right) = \frac{1}{\left\|z - x_i\right\|_2} \tag{3.6}$$

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

Part III.

Methodology and Implementation

4. Implementation

4.1. Data pre-processing

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 [1,9]. Velocity ratio in our case refers to the ratio of higher velocity to that of lower velocity. The simulation results, i.e; the steady-state fields are logged after 1000 and 30000 timesteps. The simulation results after 30000 steps are taken as the ground truth values or simply target data, whereas the input to the surrogate model are the results after 1000 steps.

4.1.2. Transformation of mesh data to graph data

Conventional 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 of 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 coordinate list (COO) format.

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,

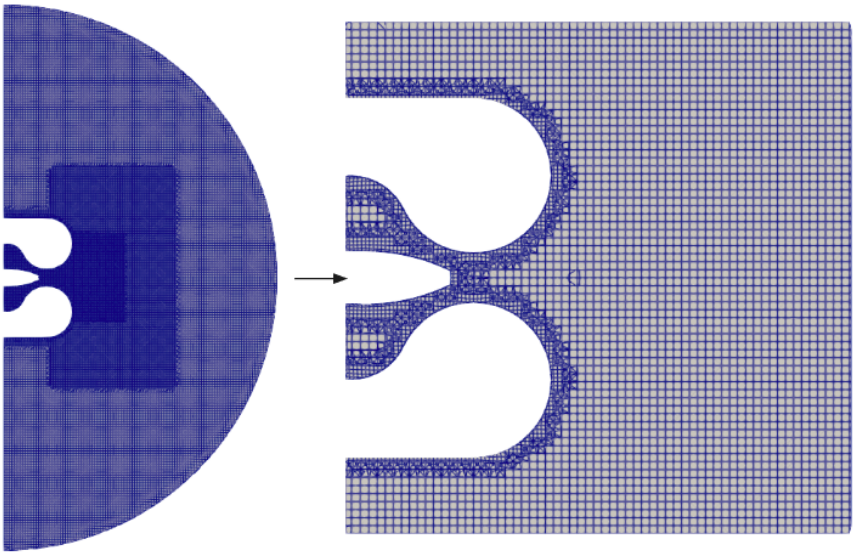

Figure 4.1.: Region of interest

- 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,

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

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 coordinates 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 freestream 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.

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.

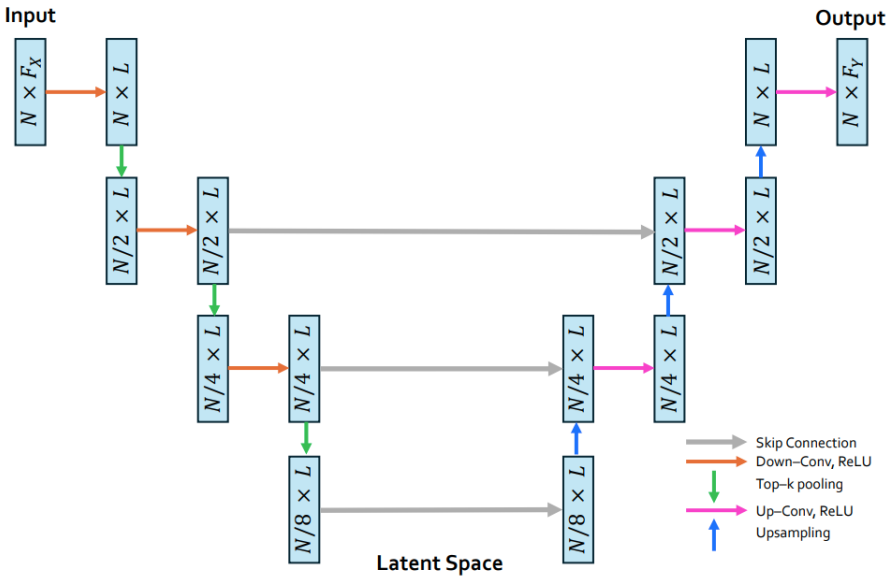

Figure 4.2.: Graph U-Net architecture

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 L2 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

4.2.5. Limitations of Graph U-Net

There are several limitations to this architecture and each case is outlined below:

1. **Poor Reconstruction:** Graph U-Net lacks explicit unpooling layers, which can lead to inadequate reconstruction of the original graph structure during decoding. This omission may result in loss of detailed information and suboptimal performance in tasks requiring precise graph reconstruction.

2. **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.

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

4. **Limited Expressiveness:** GCNConv layers may have limited expressiveness in capturing higher-order graph structures and capturing long-range dependencies in the graph. Its optimal depth is found to be 2 or 3 layers. (CITE) 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 4 different surrogate models proposed, each with a slight variation of the Graph U-Net architecture which is considered as the baseline model and referred to as *BL*. They are,

1. *BL + knn:* This model uses the k-NN interpolation technique used in PointNet++ [4] 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. *BL + SAGE + knn:* In this surrogate, we replace the GCNConv layers of *BL + knn* with GraphSAGE.

3. *BL + crsn + knn:* In this surrogate, we first obtain the set of coarsened meshes through incremental decimation of the CFD mesh by running a Python script on Paraview.

With the mesh indices and co-ordinates of the low-resolution mesh nodes, we compute the downsampled features for the graph using the sampling operator described in the subsection 3.7.3. Thus, we implement the pooling operation with the help of sampling operator and perform unpooling operation using k-NN interpolation.

4. *BL + SAGE + crsn + knn:* Similar to the previous model in every detail, the only difference in this model is that it uses GraphSAGE layers instead of the GCNConv layers as convolutions.

5. *BL + MoNet + sampl. + knn:* This model is also similar to *BL + crsn + knn* but it applies MoNet blocks for up convolutions and down convolutions.

4.4. Model hyperparameters and training parameters

4.4.1. Parameter study of graph layers

The design choices in the model architecture can impact the overall performance of the model.

Part IV.

Model Evaluation, Results and Discussions

5. Model Evaluation, Results and Discussions

Part V.

Conclusion

6. Conclusion

Appendix

Bibliography

[1] 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.

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

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

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

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