Incremental machine learning-based accelerator for computational fluid dynamics simulations

by

Sajeda Mokbel

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Chemical Engineering

Waterloo, Ontario, Canada, 2023

Abstract

The simulation of physicochemical processes with computational methods is key for engineering design, with applications in a variety of industries, ranging from pharmaceuticals to aerodynamics. Despite its importance and widespread use, significant challenges related to the accuracy and computational complexity of these simulations remain prominent.

These systems are governed by non-linear transport equations with physical and chemical processes occurring at different spatiotemporal scales in complex geometries. This leads to problems which are computationally expensive and often infeasible to solve. As such, reducing the computational complexity of multiphysics problems without compromising on accuracy is a central goal in the engineering community.

Recently, machine learning has proven to be a promising direction towards this goal. The availability of data from both multiphysics experiments and simulations have led to high-performing neural networks capable of accelerating traditional methods for solving multiphysics problems. Despite these hopeful results, there still exists a gap between machine learning and its optimal application in a realistic engineering design process. This work aims to bridge that gap through two main approaches.

The first approach is by developing a framework which hosts neural network training and existing computational multiphysics software in a unified framework. The second approach is to incrementally determine optimal neural network parameters by running computational multiphysics problems and neural network training in parallel. This has shown to reduce data collection and training time while increasing the speedup of multiphysics simulations over increments.

Acknowledgements

I would first like to thank my supervisor, Dr. Nasser Mohieddin Abukhdeir, for his guidance, mentorship, and supervision throughout the entirety of this work. I would also like to thank Dr. Hector Budman for his guidance and advice throughout this project.

A special thanks to the wonderful COMPHYS group - Alex, Matthew, Arshia, James, Ittisak and Nicholas, thank you for your help, moral support, and friendship during this learning journey.

Thank you to my sisters, Aseel and Abir, and to my dad, for their constant support.

Dedication

To my mom. Thank you for all your love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Motivation

Efficient simulation of physicochemical processes is central to engineering design. These processes typically consist of multiphase flow, heat and mass transfer, reaction and transport of chemical species in multicomponent mixtures, and turbulence. These complex processes span a wide range of space and time scales, making experimentation inefficient and often infeasible for preliminary design. Simulations present a safe and cost-effective alternative to physical prototyping and testing while providing engineers with detailed insights of the system of interest. This is accomplished through high-throughput design screening, which requires the simulation of several designs and operating conditions with relatively small variations. Thus, simulations should be executed quickly so that many configurations can be tested efficiently. This is important for several industrial applications, such as modelling airflow in buildings for efficient energy usage [29], determining the hydrodynamics of fluidized bed systems [44], or designing optimal aerospace vehicles [39].

These processes, and many more in chemical engineering, can be described by complex partial differential equations (PDEs) that represent conservation of mass, momentum, and energy in a continuum. Most industrially-relevant multiphysics occurs in complex geometries such as chemical reactors, separators, and boilers; or porous materials like zeolites. To further complicate matters, many of these processes are convection-dominated, meaning convective transport is significantly larger than other transport processes such as diffusion. The governing equations therefore have high Peclet and Reynolds numbers and are classified mathematically as hyperbolic PDEs.

Conventional numerical schemes encounter multiple computational difficulties when solving hyperbolic PDEs [28]. To effectively model convective flows, upwinding schemes must be used to enforce hyperbolicity numerically. Current numerical methods for convective transport are prone to instabilities such as non-physical oscillations, which occur when a downstream boundary condition forces a rapid change in the solution. High mesh densities are usually required for adequate numerical accuracy in complex geometries, but result in stability issues due to the corresponding requirement of small timescales (Courant-Friedrich-Lewy condition) [16].

Since most engineering applications operate at macroscopic length scales, the processes mentioned above are adequately approximated by continuum models such as the Navier-Stokes equations. The Navier-Stokes equations are derived by substituting intensive properties (i.e., density for mass) into basic laws of physics: the continuity equation and Newton's second law of motion. The force term contains three components: a pressure term, a stress term, and an external force term acting on every fluid particle. This general form, however, is very difficult - if not impossible - to use in practice, as it has a number of unspecified elements. Thus, further assumptions are required to simplify the model.

This work focuses on Newtonian incompressible fluids, though in reality, many common fluids are compressible. The stress term is further simplified by Newton's law of viscosity, which assumes the stress is linearly proportional to the rate of deformation. The Navier-Stokes equations adequately describe many relevant physicochemical processes, with additional equations and variations for reaction, turbulence, heat transfer, and mixtures. Thus, this work focuses on computational multiphysics (CMP) simulations which are modelled by the Navier-Stokes equations. These equations are rarely solved analytically, instead their solutions are approximated numerically with discretization methods.

Solving the Navier–Stokes equations for convection-dominated problems requires numerical methods that may be computationally expensive, and often infeasible to solve [28]. Several approaches have been developed to solve these equations numerically, by discretization using methods of different orders, such as finite difference, finite volume, spectral methods, finite element, and more [56]. However, to ensure convergence, fields must vary smoothly on the mesh, meaning meshes need to resolve very small spatiotemporal scales, which increases the computational cost.

In spite of considerable successes, challenges related to the effective implementation of CMP remain prominent. The effectiveness of simulations is intimately tied to the fidelity of the mathematical models, mesh quality of the geometry, the stability and convergence of the numerical methods used, and computational speed [54]. Efficient simulations are required so that frequent, small changes to a system can be tested in a timely manner. A typical parametric analysis requires many simulations and iterations for a class of systems in a regime, with only small variations to the geometry or fluid parameters. Currently, these past iterations and history of related systems are not used to speed up computations. Thus, numerical solutions should be accelerated by leveraging past iterations of simulations under similar conditions. If we have access to history of related systems, it is important to use it in a way that makes simulations more efficient. This can be accomplished with machine learning techniques.

Machine learning (ML) has shown promising results in many computational multiphysics applications such as turbulence closure modelling [58], operator learning [27], and ML-accelerated physics solvers [38, 2, 26]. This work focuses on ML-accelerated computational multiphysics, which seeks to hybridize classical numerical methods with data-driven deep learning techniques [5]. This idea has already shown promising results in a range of related works [48, 23, 26]. Despite this rapid progress, current work in this field lacks key features required to be used effectively in practice. The models, pipeline, and data are sometimes not publicly available, and if they are, a standalone deep learning model is provided rather than an automated, end-to-end framework coupled with existing multiphysics solvers. Additionally, the choice of neural network architecture is not based on scientific reasoning, models are usually designed similar to popular, high-performing architectures which are used for unrelated tasks like image recognition [45]. Many of these recent publications do not adequately describe their algorithms and sometimes do not show a performance benefit compared to modern traditional methods, particularly when compared to Finite Element software like `OpenCMP` [35] and `NGSolve` [47]. To build accessible ML-accelerated CMP solvers, it is beneficial to consider engineering expertise during the design phase.

1.2 Objectives

The overall objective of this thesis is to develop ML-based acceleration methods for computational multiphysics which are compatible with the engineering design process. Specific objectives include,

Reproduce research on numerical methods combined with deep learning, specifically, CFDNet [38]. By reproducing this work, the performance benefit of ML-coupled physics solvers will be validated. This also provides a foundation for further analysis and improvement of the coupled framework.

Determine optimal neural network architectures and reduce data usage through hyperparameter tuning and incremental training. This reduces data processing and training time while closely following a realistic engineering design cycle.

Apply and validate method to arbitrary 2D and 3D flow problems to ensure the method could be used in practice, and as a basis for further research.

1.3 Thesis Structure

This thesis is grouped into six chapters, including this introduction chapter. Chapter 2 is the background section, describing common fluid transport equations, discretization methods, and a machine learning overview. Chapter 3 consists of an extensive literature review on ML-accelerated multiphysics solvers. Chapter 4 describes the research methods used in this work, detailing the computational methods used for both the multiphysics simulation and the machine learning implementation. Chapter 5 summarizes the results of this work, and finally, Chapter 6 discusses conclusions and future work.

Chapter 2

Background

This chapter introduces the study of multiphysics systems, the most common equations used to describe these systems, and the numerical methods used to solve these equations. Key concepts in machine learning, including detailed descriptions of popular neural network architectures and algorithms, are also discussed.

2.1 Computational Multiphysics in Chemical Engineering

Computational multiphysics (CMP) is the unified study of heat, mass, and momentum transfer in physicochemical processes with computer simulations. These are systems governed by several physical and chemical processes, or multiscale systems, where processes in the same system occur at different time and spatial scales. CMP is widely used in the design and implementation of biological and chemical processes in industry. This coincides with the field of chemical engineering, and so, CMP is an essential skill for a chemical engineers toolkit.

Chemical engineering is a multidisciplinary field which spans a variety of industries, including, but not limited to, pharmaceuticals, cosmetics, energy systems, wastewater treatment, and food production. One example, demonstrating a common industrial application in the field of chemical engineering, is the conversion of raw materials into consumer products through chemical reactions and fluid flow. A common environment where matter undergoes this change is in reactors, which are controlled vessels where a chemical reaction occurs. Different types of reactors are designed to perform this conversion depending on the desired product. In order to develop an efficient reactor at an industrial scale, engineers must understand the physical and chemical processes that occur in the reactor over time. An understanding of the phenomena will help determine the operating conditions and design required for optimal performance. This is accomplished with computational multiphysics.

2.2 Navier-Stokes equations

The processes mentioned above are accurately described at length scales greater than inter-atomic distances, and so, continuum models are used to analyze them. Continuum models consist of equations formulated to describe basic laws of physics representing the conservation of mass, momentum, and energy. These set of equations, referred to as the Navier-Stokes equations, are used to model these processes.

The Navier-Stokes equations arise from three fundamental laws of physics, which state that [54]:

- The mass of a fluid is conserved,
- The rate of change of momentum of is equal to the sum of all forces imposed on a fluid particle, and
- The rate of change of energy is equal to the rate of heat added to the system and the work done on the particle.

The key difference in the equations for fluid mechanics versus the classical physics equations, i.e., Newtons laws, is treating the fluid as a continuum. The laws above are reformulated into equations with macroscopic properties - such as density, velocity, pressure, and temperature; and their derivatives in time and space [54]. A fluid control volume is the smallest element of a fluid where molecular interactions can be neglected. Let Figure 2.1 represent this fluid element of interest.

Equation of continuity

The equation of continuity is developed by writing a mass balance over the fluid control volume, $\Delta x \Delta y \Delta z$, which is fixed in time and space.

$$\Bigg[\text{rate of increase of mass}\Bigg] = \Bigg[\text{rate of mass in}\Bigg] - \Bigg[\text{rate of mass out}\Bigg] \tag{2.1}$$

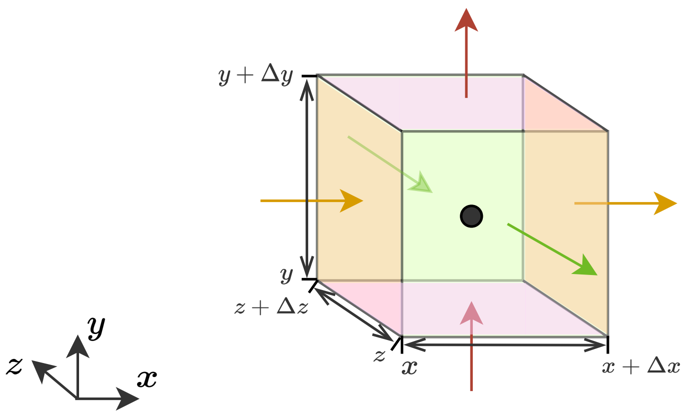


Figure 2.1: *Control volume which a fluid flows through. The arrows indicate the mass flux in and out of the volume at faces normal to all three dimensions of the element*

To begin, consider a fluid passing parallel to the x-axis, through the faces shaded orange. The rate of mass entering the volume from this surface is given by:

$$\Delta y \Delta z[(\rho u)_x - (\rho u)_{x+\Delta x}] \tag{2.2}$$

Similar equations can be written for fluid passing through the volume parallel to the y-axis and the z-axis. Putting these together, the mathematical notation of Equation 2.1 becomes:

$$\begin{aligned} \Delta x \Delta y \Delta z \frac{\partial \rho}{\partial t} =& \Delta y \Delta z[(\rho u)_x - (\rho u)_{x+\Delta x}]+ \\ & \Delta x \Delta z[(\rho v)_y - (\rho v)_{y+\Delta y}]+ \\ & \Delta y \Delta x[(\rho w)_z - (\rho w)_{z+\Delta z}] \end{aligned} \tag{2.3}$$

where u, v, w are the x, y, z components of velocity, respectfully, and ρ is the density of the fluid. The simplified version of this equation is derived by dividing by the differential volume element $\Delta x \Delta y \Delta z$ and taking the limit as $\Delta x, \Delta y,$ andΔz approach zero:

$$\begin{aligned} \frac{\partial \rho}{\partial t} &= -\left(\frac{\partial}{\partial x}\rho u + \frac{\partial}{\partial y}\rho v + \frac{\partial}{\partial z}\rho w\right) \\ &= -(\nabla \cdot \rho \mathbf{v}) \end{aligned} \tag{2.4}$$

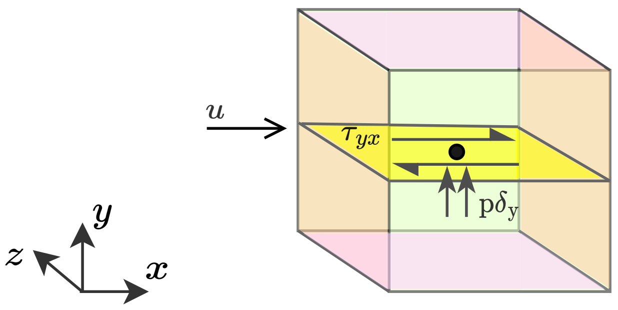


Figure 2.2: *Point of fluid in control volume for steady, unidirectional flow. The molecular stresses acting on the face normal to the y-axis is shown. τ_{yx} is the shear stress from the velocity gradient, and $\mathrm{p}\delta_{\mathrm{y}}$ is the pressure exerted from the surrounding fluid, and will always act perpendicular to the surface.*

For an incompressible flow, where the fluid density is assumed to be constant, this equation is simplified to:

$$(\nabla \cdot \mathbf{v}) = 0 \tag{2.5}$$

Equation of motion

The equation of motion is derived from the momentum balance over the control volume of a fluid element. Consider the fluid element in Figure 2.1. Momentum is transported by two mechanisms: (i) molecular transport and (ii) convective transport. Molecular transport is created by the forces acting on the fluid in the volume element. There will be two contributions to the force in the molecular effects - the first contribution is associated with pressure, and the second contribution is due to the viscous forces acting on the fluid.

A point of a fluid in a control volume will constantly experience a pressure from the weight of surrounding fluid. Consider the surface normal to the y-axis for a fluid point in the centre of the control volume in Figure 2.2. This fluid point will experience a pressure from the weight of the fluid above it. This pressure term is denoted by $\mathrm{p}\delta\mathrm{y}$.

A fluid in a container can be thought of as several stacked thin layers of fluids, subject to effects from boundary layers - more specifically, the no-slip condition. Due to these effects, the fluid layers will flow at different velocities depending on their location, generating a velocity gradient between the layers. The velocity gradient between these layers creates a shear stress parallel to the layers, as pictured as the yellow shaded plane in section 2.2. This shear stress can be described by *Newton's Law of Viscosity*, which relates the velocity gradient to the viscosity of the fluid, μ, with the following formula:

$$\tau_{yx} = -\mu \frac{du}{dy} \tag{2.6}$$

Combining the terms from the viscous and pressure forces, the molecular stresses acting on the face perpendicular to y are defined by:

$$\pi_{xy} = \mathrm{p}\delta_{\mathrm{xy}} + \tau_{xy} \tag{2.7}$$

where δ_{xy} is the Kronecker delta, and is only non-zero when the force is applied normal to the surface of interest. It is important to note that τ_{ij} will not always be as simple as Equation 2.6. In realistic scenarios, a 3D flow can have velocity components in all three dimensions, (u, v, w), and so the viscous stress can be a linear combination of all three velocity gradients. The shear stress can then be defined in a more general form:

$$\tau_{xy} = -\mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) + (\frac{2}{3}\mu - \kappa)\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\delta_{\mathrm{xy}} \tag{2.8}$$

where μ is the viscosity of the fluid and κ is the dilatational viscosity, a term that stems from kinetic theory, and is equal to zero for monatomic gases at low density [42].

Momentum can also be transported by the bulk movement of the fluid. This phenomena is called *convective transport*. Consider again Figure 2.2, except now, imagine the velocity has non-zero components in all three dimensions. Hence, (u, v, w) are all non-zero. The central surface normal to the y-axis shaded in Figure 2.2 would have a flow rate of v (the y-component of velocity). The fluid would also have a momentum of $\rho\mathbf{v}$ per unit volume, where $\mathbf{v}$ is the multidimensional velocity vector. Putting this together, the momentum flux by convection for the central surface shaded yellow in Figure 2.2 would be $v\rho\mathbf{v}$. Similarly, the momentum flux by convection for the faces normal to the x and z axis would be $u\rho\mathbf{v}$ and $w\rho\mathbf{v}$, respectfully. To be concise, this can be written as a second-order tensor:

$$\rho\mathbf{v}\mathbf{v} \tag{2.9}$$

To conclude, the combined momentum-flux tensor $\mathbf{\Phi}$, which combines the momentum flux from both molecular effects and convective effects, is written as,

$$\mathbf{\Phi} = \boldsymbol{\pi} + \rho\mathbf{v}\mathbf{v} \tag{2.10}$$

where $\mathbf{\Phi}$ is a second-order tensor. Given there are three dimensions, the momentum flux has nine independent coefficients in total.

The equation of motion is derived similarly to the equation of continuity, except with momentum as the conserved quantity. The momentum balance of a fluid element can be described by:

$$\Bigg[\text{rate of increase of mom.}\Bigg] = \Bigg[\text{rate of mom. in}\Bigg] - \Bigg[\text{rate of mom. out}\Bigg] + \Bigg[\text{external forces}\Bigg] \tag{2.11}$$

Consider the same volume element as Figure 2.1. The momentum flux from the orange shaded faces perpendicular to the x-axis can be described as follows:

$$\begin{aligned}\Delta x\Delta y\Delta z\frac{\partial}{\partial t}\rho u = {} & \Delta y\Delta z(\Phi_{xx}|_x - \Phi_{xx}|_{x+\Delta x}) \\ & + \Delta z\Delta x(\Phi_{yx}|_y - \Phi_{yx}|_{y+\Delta y}) \\ & + \Delta x\Delta y(\Phi_{zx}|_z - \Phi_{zx}|_{z+\Delta z}) \\ & + \rho g_x\Delta x\Delta y\Delta z\end{aligned} \tag{2.12}$$

Dividing by the volume element and rewriting in terms of gradients, the general form of the momentum flux equation in tensor notation for all three components is:

$$\frac{\partial}{\partial t}\rho\mathbf{v} = -[\nabla\cdot\rho\mathbf{v}\mathbf{v}] - \nabla p - [\nabla\cdot\tau] + \rho\mathbf{g} \tag{2.13}$$

The Navier Stokes Equations

For a constant ρ and a constant μ, inserting the expanded version of Equation 2.8 into the equation of motion 2.13 we get the *Navier-Stokes* equation:

$$\rho\left(\frac{\partial\mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\mathbf{v}\right) = -\nabla P + \mu\nabla^2\mathbf{v} + \rho\mathbf{g} \tag{2.14}$$

Dimensional analysis shows that the Navier-Stokes equation can be generally described by one number: the *Reynolds* number. By dividing every dimensional term of Equation 2.14 by a suitable reference value, the equation is reformulated to be independent of the physical measurements of the problem. This is important in engineering, where the scale of a problem might be too costly to replicate in the experimental and testing phases of a project. In Equation 2.14, the terms can be non-dimensionalized as follows:

$$\mathbf{v}* = \frac{\mathbf{v}}{v_{ref}}; \qquad \mathbf{p}* = \frac{\mathbf{p}}{\rho v_{ref}^2}; \qquad \mathbf{g}* = \frac{\mathbf{g}L}{v_{ref}^2} \tag{2.15}$$

where L is the characteristic length scale and v_{ref} is the reference velocity. To write Equation 2.14 in its non-dimensional form, a term arises, which is referred to as the Reynolds (Re) number. The Reynolds number describes the ratio of inertial forces to viscous forces, and is defined as:

$$Re = \frac{\rho L v_{ref}}{\mu} \tag{2.16}$$

This number allows Equation 2.14 to be reformulated as:

$$\frac{\partial \mathbf{v}*}{\partial t} + (\mathbf{v}* \cdot \nabla)\mathbf{v}* = -\nabla p * + \frac{1}{\mathrm{Re}}\nabla^2\mathbf{v}* + \mathbf{g}* \tag{2.17}$$

Reynolds-Averaged Navier Stokes for Turbulence Modelling

All flows encountered in nature become chaotic above a certain Reynolds number. At high Reynolds numbers, flows become turbulent and disorderly states of motion develop between fluid layers, making the velocity and pressure fields continuously change over time. This generates rotational structures of varying length scales in the flow called turbulent eddies.

Eddies are responsible for the effective heat and momentum flux across the entire domain [1]. They span a wide range of length scales, and it is typical to have eddies as small as 1×10^{-5} metres for a domain length scale of 1×10^{-1} metres. Thus, to capture the flow dynamics at these length scales, a very fine mesh of roughly $10^9 - 10^{12}$ points is required [54]. Additionally, time would need to be discretized in steps of at least 1×10^{-4} seconds. This is a significant computational burden, and so, additional approximations and simplifications are preferred over direct numerical simulations.

Reynolds-Averaged Navier Stokes (RANS) equations are used to avoid the need to compute the effects of every eddy. The Reynolds-Averaged Navier-Stokes (RANS) equations are time-averaged Navier Stokes equations primarily used to solve turbulent flows. These equations were derived by observing the behaviour of flow velocity in a turbulent regime. Consider the velocity profile of a flow in a tube with a constant imposed pressure gradient in Figure 2.3. It is clear that the velocity fluctuates from a mean value, and thus, can be decomposed into a mean component, denoted $\overline{v_z}$, and a fluctuating component, denoted

v'_z. This is referred to as the Reynolds decomposition. The Reynolds decomposition of the z component of velocity, as shown in Figure 2.3, is therefore:

$$v_z = \overline{v_z} + v'_z \tag{2.18}$$

The mean value, $\overline{v_z}$, is attained by taking the average over a window of time [42]:

$$\overline{v_z} = \frac{1}{t_0} \int_{t-\frac{1}{2}t_0}^{t+\frac{1}{2}t_0} v_z(s)ds \tag{2.19}$$

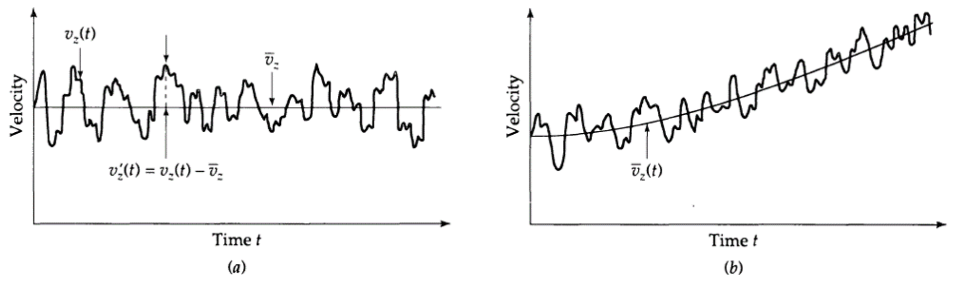


Figure 2.3: *Figure from Chapter 5.2 of [42] showing v_z, its time-averaged value $\overline{v_z}$, and its fluctuating component v'_z in turbulent flow. In (a), $\overline{v_z}$ does not depend on time, while in (b) $\overline{v_z}$ is time dependant.*

Now that the time-averaged quantities have been described, they can be inserted into the continuity equation 2.4 in cartesian notation as:

$$\frac{\partial}{\partial x}(\overline{v_x} + v'_x) + \frac{\partial}{\partial y}(\overline{v_y} + v'_y) + \frac{\partial}{\partial z}(\overline{v_z} + v'_z) = 0 \tag{2.20}$$

The fluctuating component, v'_x, has a time average of zero. Information about the fluctuating component can therefore be obtained by taking the root-mean-square of the velocity fluctuations [54].

The equations of motion can also be re-written in terms of their time-averaged quantities. Before doing so, rules related to time averaged components and their derivatives should be noted. Consider the Reynolds decomposition of two vector fields $\mathbf{v} = \mathbf{V} + \mathbf{v}'$, and $\mathbf{u} = \mathbf{U} + \mathbf{u}'$ where $\mathbf{V}$ and $\mathbf{U}$ are the mean values. The following relationships hold:

$$\begin{aligned}
&\overline{\mathbf{v}'} = 0; \;\; \overline{\mathbf{v}} = \mathbf{V}; \;\; \frac{\partial \overline{\mathbf{v}}}{\partial s} = \frac{\partial \mathbf{V}}{\partial s} \\
&\overline{\mathbf{v} + \mathbf{u}} = \mathbf{V} + \mathbf{U}; \;\; \overline{\mathbf{v}\mathbf{u}} = \mathbf{V}\mathbf{U} + \overline{\mathbf{v}'\mathbf{u}'}; \\
&\overline{\mathbf{v}\mathbf{U}} = \mathbf{V}\mathbf{U}; \;\; \overline{v'U} = 0
\end{aligned} \tag{2.21}$$

Using the Reynolds decomposition in the momentum equations will provide the following formulation for the x-component of velocity, u, where $\mathbf{u}$ is the velocity vector. Similar equations are developed for the y and z components of velocity.

$$\frac{\partial U}{\partial t} + \nabla \cdot (U\mathbf{U}) + \nabla \cdot (\overline{u'\mathbf{u}'}) = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \nu \nabla^2 U \tag{2.22}$$

It is clear from Equation 2.22 that there now exist Reynolds stresses, $u'\mathbf{u}'$, that must be solved for. Turbulence modelling, which will be described further in section 4, focuses on approximating these Reynolds stresses by estimating them in terms of the mean velocity components. This term is required to close the momentum equations.

2.3 Numerical methods

The above equations cannot be solved analytically, they must be approximated numerically. This is accomplished with the application of numerical methods, the most popular ones being the Finite Difference Method (FDM), the Finite Element Method (FEM), and the Finite Volume Method (FVM). This work focuses on both the Finite Element Method (FEM) and the Finite Volume Method (FVM) because of their ability to handle arbitrary geometries. Other methods, such as the FDM, require simple geometries that conform with the coordinate system used. This is not suitable for engineering applications, where most environments involve complex geometries. These complex geometries need an unstructured mesh to resolve regions of high error, the boundary layer of an aircraft wing being one example.

2.3.1 Finite Volume Method

Section 2.1 described the equations governing fluid flow and heat transfer by applying them to a control volume. The Finite Volume Method is a numerical method used to solve these equations by *integrating* over the control volume (CV). To demonstrate this, consider a steady, two-dimensional convection-diffusion equation where convection is only present in the x-dimension:

$$\frac{\partial}{\partial x}\rho u T = \frac{\partial}{\partial x}(\Gamma\frac{\partial T}{\partial x}) + \frac{\partial}{\partial y}(\Gamma\frac{\partial T}{\partial y}) \tag{2.23}$$

where Γ is the diffusion coefficient and T is some quantity being diffused in an arbitrary two-dimensional geometry. The geometry can be discretized into multiple control volumes, as shown in 2.4. Consider a point - or *node* - at the centre of each CV. Focusing on the central CV with node P, we can begin discretizing the diffusion equation in terms of the surrounding nodes and faces by integrating over the CV.

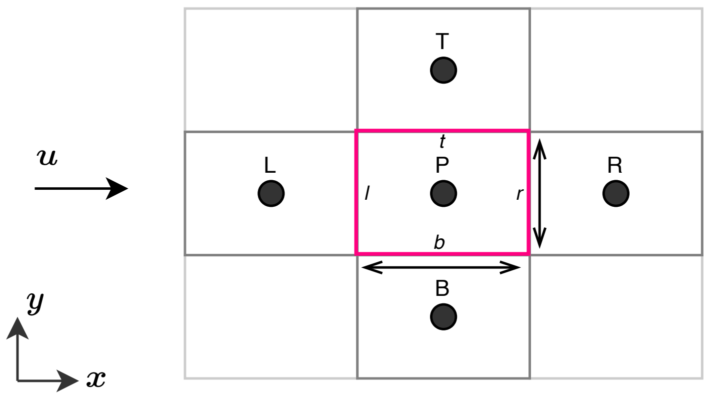


Figure 2.4: *Nodes in a simple 2D geometry. The central node is labelled P, and the surrounding nodes are labelled T (Top), R (Right), L (Left), B (Bottom). The lower case t,r,b,l denote the control volume faces of the point P.*

The equation will be solved over the control volume (CV), hence, it needs to be integrated over the entire CV:

$$\int_{CV}\frac{\partial}{\partial x}\rho u T = \int_{CV}\frac{\partial}{\partial x}(\Gamma\frac{\partial T}{\partial x}) + \frac{\partial}{\partial y}(\Gamma\frac{\partial T}{\partial y})dV \tag{2.24}$$

Gauss' divergence theorem allows a volume integral of field F to be expressed as a surface integral with the following relation:

$$\int_{CV} div(F)dV = \int_{A}(\hat{n}\cdot F)dA \tag{2.25}$$

Using Gauss' divergence theorem, the Equation 2.24 can be rewritten as a surface integral:

$$\int_A \hat{n}\cdot(\rho u T)dA = \int_A (\alpha\frac{\partial T}{\partial x})(\hat{i}\cdot\hat{n})dA + \int_A (\alpha\frac{\partial T}{\partial y})(\hat{j}\cdot\hat{n})dA \tag{2.26}$$

which can then be integrated and discretized.

Discretization of diffusion

To start, the diffusion term will be discretized using the *Central Differencing scheme* (CDS). The CDS allows an approximation $\frac{\partial T}{\partial x}$ for each face by computing the difference of the nodes of each surrounding CV and dividing by the distance between the two nodes. Consider the diffusion term in Equation 2.26. The cross-sectional area normal to the x-direction of the system is integrated from the right face to the left face. Similarly, in the y-direction, the cross-sectional area is integrated from the top face to the bottom face.

$$\Gamma A_r \frac{\partial T}{\partial x}\bigg|_r - \Gamma A_l \frac{\partial T}{\partial x}\bigg|_l + \Gamma A_t \frac{\partial T}{\partial x}\bigg|_t - \Gamma A_b \frac{\partial T}{\partial x}\bigg|_b \tag{2.27}$$

the area on the right and left face on the system is in the direction normal to x. Hence, the area is $\Delta y \Delta z$. Since the effects from the third dimension, z, are to be ignored, it can be assumed that $\Delta z = 1$. Similarly, on the top and bottom faces of the system, the area is $\Delta x \Delta z$. Thus,

$$A_b = \Delta x, A_t = \Delta x, A_r = \Delta y, A_l = \Delta y \tag{2.28}$$

$$\frac{\partial T}{\partial x}\bigg|_r = \frac{T_R - T_P}{\Delta x}, \quad \frac{\partial T}{\partial x}\bigg|_l = \frac{T_P - T_L}{\Delta x}, \quad \frac{\partial T}{\partial x}\bigg|_t = \frac{T_T - T_P}{\Delta y}, \quad \frac{\partial T}{\partial x}\bigg|_b = \frac{T_P - T_B}{\Delta y} \tag{2.29}$$

Plugging the results from 2.28 and 2.29 back into 2.27, the following equation is formulated:

$$\frac{\Gamma\Delta y}{\Delta x}(T_R - T_P) - \frac{\Gamma\Delta y}{\Delta x}(T_P - T_L) + \frac{\Gamma\Delta x}{\Delta y}(T_T - T_P) - \frac{\Gamma\Delta x}{\Delta y}(T_P - T_B) \tag{2.30}$$

For simplicity, diffusion coefficients are defined for each face:

$$D_r = \frac{\Gamma\Delta y}{\Delta x} \quad D_l = \frac{\Gamma\Delta y}{\Delta x} \quad D_t = \frac{\Gamma\Delta x}{\Delta y} \quad D_b = \frac{\Gamma\Delta x}{\Delta y} \tag{2.31}$$

Grouping the T terms together, the diffusion term can be discretized as follows:

$$D_rT_R + D_lT_L + D_tT_T + D_bT_B - (D_r + D_l + D_t + D_b)T_P \tag{2.32}$$

Now, the convection term is left to discretize. For the convection term, another scheme is used to approximate the node values. It is called the *Upwind Differencing Scheme* (UDS). UDS is required for convection-dominated flows, as the CDS fails to converge due to oscillations in the solution. UDS approximates the value of velocity as its upstream value. Since the flow is convecting in the positive x-direction at a constant velocity, $u_r = u_l = u$ and $u > 0$. The coefficients of the discretised equation are therefore always positive, hence satisfying the requirements for boundedness.

Using Gauss' theorem, the convection term is written as the following surface integral:

$$\int_A \hat{n} \cdot (\rho u T) = (\rho u T A)_r - (\rho u T A)_l \tag{2.33}$$

Inserting in the following assumptions to the surface integral,

$$T_r \approx T_P \quad T_l \approx T_L \tag{2.34}$$

the following result is produced:

$$uAT_P - uAT_L \tag{2.35}$$

for simplicity, let

$$F_r = uA|_r \quad F_l = uA|_l \tag{2.36}$$

Now, the diffusion and convection discretizations are equated and grouped by T:

$$F_rT_P - F_lT_L = D_rT_R + D_lT_L + D_tT_T + D_bT_B - (D_r + D_l + D_t + D_b)T_P \tag{2.37}$$

Final discretized form

The final discretized form of the entire equation, without including any boundary conditions or known values, is:

$$[F_r + D_r + D_l + D_t + D_b]T_P = D_rT_R + (D_l + F_l)T_L + D_tT_T + D_bT_B \tag{2.38}$$

Let $a_R = D_r, \quad a_L = (D_l + F_L), \quad a_T = D_t, \quad a_B = D_b$

$$[a_R + a_L + a_T + a_B]T_P = a_RT_R + a_LT_L + a_TT_T + a_BT_B \tag{2.39}$$

Equation 2.39 can now be solved by inserting boundary terms, rewriting in matrix form, and solving the system of equations. It is important to note that a simple problem was chosen only to demonstrate the application of the FVM on a mesh element. In real life applications, the spatial scale is large, and hundreds of thousands of nodes, often millions, must be solved for and hence require advanced computational methods and hardware to pursue.

2.4 Machine Learning

Artificial Intelligence (AI) is a field in computer science and engineering that focuses on the development of machines capable of performing tasks that require human knowledge. Today, AI is a thriving field with many practical applications and active research topics, ranging from cognitive computing to robotics.

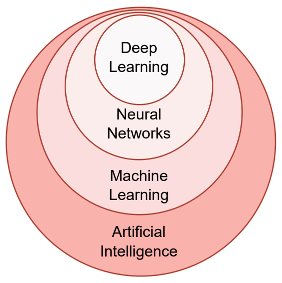


Figure 2.5: *Fields of AI*

While the two terms are sometimes used interchangeably, it is important to note that Machine Learning (ML) is a *subset* of AI that uses mathematical models of data to perform tasks autonomously. In order to achieve desired performance in a model, it must be able to understand the factors of variation of a dataset [17]. Factors of variation refer to different sources of influence coming from the physical world, for instance, language accents in voice recordings or the brightness of the sun in an image. This represents a variability in the dataset, and it must be understood in order to develop an ML model capable of generalizing.

Neural Networks (NNs) are a type of ML model structure designed with a set of operations arranged in a hierarchical fashion, inspired by the structure of a brain. NNs apply a set of computations on an input, transforming the input into a feature representation. Layers in NNs are defined as a step that computes these feature representations. The *depth* of a NN is the number of layers included in the model architecture. Shallow NNs - i.e., NNs with a small number of layers, are difficult to use in practice depending on the task at hand. The lack of feature representations will make it challenging for the model to accurately depict the variability in a dataset.

Deep Learning (DL) is a class of Neural Networks that are structured with several layers - hence the term *deep*. DL was created to tackle the limitations of shallow NNs. Deep learning introduces feature representations that are computed in terms of previous, simpler feature representations. This allows the model to hierarchically break down a problem into simpler concepts, improving performance for complex learning tasks.

2.4.1 Feedforward neural network

To introduce the concept of ML using NNs, the simplest NN architecture - Feedforward Neural Networks - will first be discussed. At a high level, NNs consist of a network of functions with applied weights, referred to as parameters, which the computer determines by analyzing data. A parameter is computed by a series of mathematical operations which produce an output after receiving one or more inputs. In the next layer of a network, the previous output is fed as input to a new layer with a new set of operations that again produce outputs. These outputs are then passed into new sets of operations, and this cycle continues until the original input has gone through all layers. The final output of the last layer in a network therefore produces the NNs output, or, the NNs prediction. This full cycle is referred to as a forward pass.

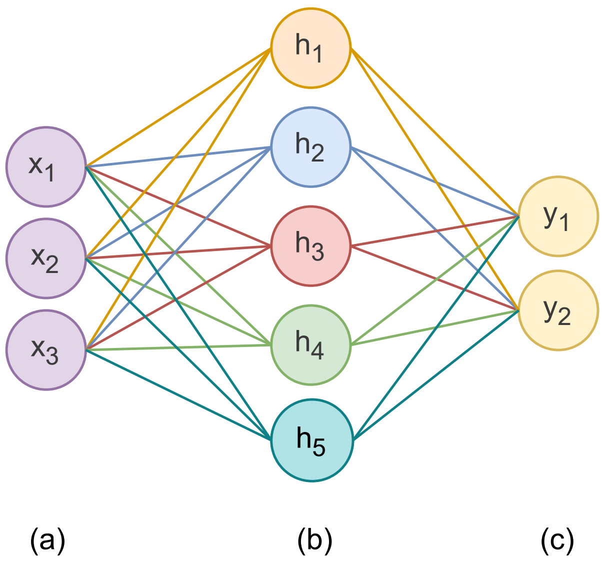


Figure 2.6: *Basic one-layer network. The input data, (a), goes through a layer. The layer is composed of several nodes, (b), with each node consisting of a set of weights and biases applied to each input point, followed by an activation function. Finally, the output (c) is returned.*

A layer consists of several nodes, which are computational units with input connections, a function that applies weights and biases to these inputs, and connections to return an output. In Figure 2.6, (b) represents a layer with 5 nodes. In each node, the following set of operations are applied:

$$\begin{aligned} h(\mathbf{x}) &= f(\mathbf{w}^T\mathbf{x} + \mathbf{b}) \\ &= f(w_1x_1 + w_2x_2 + w_3x_3 + b) \end{aligned} \tag{2.40}$$

where $f(x)$ is an activation function of choice. Activation functions are a key step in the neuron computation - without activations, the learning process would essentially become linear regression. The activation function introduces non-linearity to the model, allowing it to perform more complex tasks. Some common activation functions are visualized below.

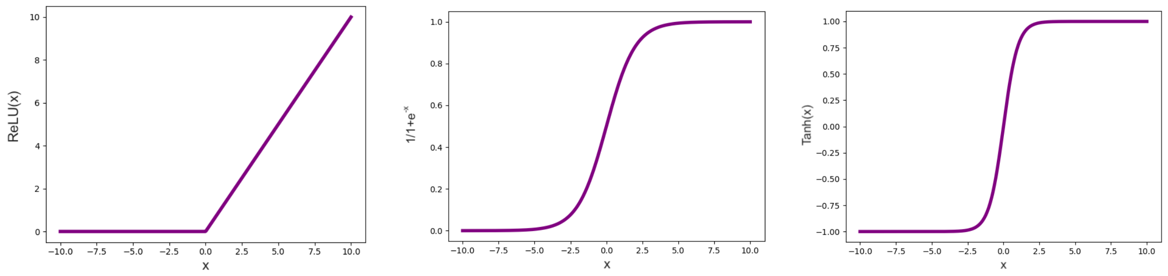


Figure 2.7: *Common activation functions used in ML models*

The choice of an activation function depends on the problem of interest. For instance, if given a problem where negative values are unphysical, the ReLU (rectified linear unit) function would be suitable as it zeros out negative values (see Figure 2.7).

Figure 2.6 is a simple example of a one-layer neural network chosen just to exemplify layer operations. In practice, most neural networks are generated with several layers, and hence, further break down the problem into a hierarchy of representations through deep learning.

Learning parameters in neural networks

The process of learning in a neural network simply refers to the continuous adjustment of the NNs parameters through multiple forward passes until a desired error is reached. This cycle is commonly referred to as an epoch. The error is computed by a loss function, which measures the difference between the NNs prediction and the target output for a given data sample. As such, every input data sample must be provided with a corresponding target value in supervised learning. Unsupervised learning refers to learning without target values, such as in dimensionality reduction or clustering algorithms.

The objective of training is to find the most optimal weights and biases (which correspond to a minimal loss). In order to fully understand how neural networks are trained, it is crucial to understand how the parameters of a neural network are updated during training. This happens through backpropagation.

Backpropagation aims to minimize the loss function by updating the NNs parameters using gradient descent. The power of gradient descent stems from a widely used concept in calculus - gradients. To determine the amount that the model parameters should be adjusted by, the gradient of the error is computed. Backpropagation refers to the backwards propagation of error - where the gradient of the error is traced back to all its dependant variables in the neural network.

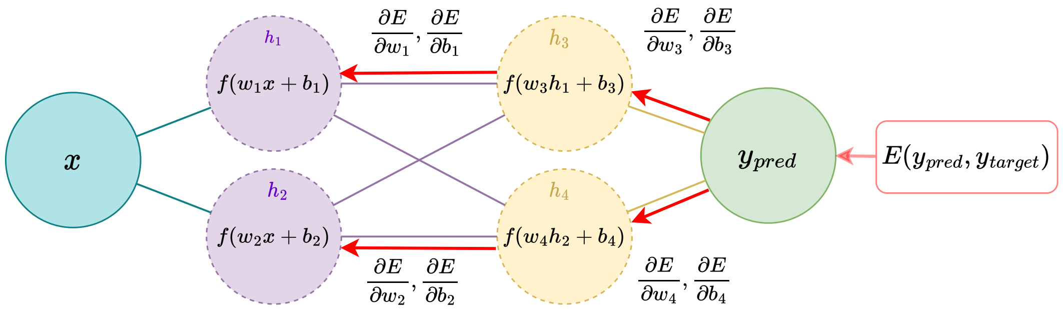

$\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial b_1}$
$\frac{\partial E}{\partial w_3}, \frac{\partial E}{\partial b_3}$
h_1
h_3
$f(w_1x + b_1)$
$f(w_3h_1 + b_3)$
x
y_{pred}
$E(y_{pred}, y_{target})$
h_2
h_4
$f(w_2x + b_2)$
$f(w_4h_2 + b_4)$
$\frac{\partial E}{\partial w_2}, \frac{\partial E}{\partial b_2}$
$\frac{\partial E}{\partial w_4}, \frac{\partial E}{\partial b_4}$

Figure 2.8: *Backpropagation. Once the data, x, goes through a full forward pass, the error is computed and backpropagated through the NN. f(x) is the activation function.*

To illustrate the concept of backpropagation, the gradients of the neural network in Figure 2.8 will be computed in terms of the adjustable neural network parameters. In supervised learning, the error is computed by comparing the NN output with a target value through a loss function. Although the NN parameters are not direct inputs of the error/loss function, the chain rule in calculus allows us to express the gradient of the error in terms of the NN weights and biases. For example, the gradient of parameter w_1 in

Figure 2.8 is computed as follows:

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y_{pred}} \frac{\partial y_{pred}}{\partial h_3} \frac{\partial h_3}{\partial h_1} \frac{\partial h_1}{\partial w_1} \tag{2.41}$$

Optimization

Optimization algorithms use the gradient information to iteratively update NN parameters and hence improve the accuracy of the ML model. To exemplify this, consider a common optimization algorithm used in ML, Stochastic Gradient Descent (SGD). The objective of SGD, like all optimization algorithms, is to minimize the loss function by updating the decision variables, which in this case, are the NN parameters.

The above section described how the gradients of the loss are computed with respect to the NN parameters. The SGD algorithm uses these gradients, along with a learning rate, to update the NN weights and biases so that the next prediction lands as close as possible to the minimal loss. The SGD algorithm follows the direction of the fastest decrease of the loss function, or, towards the negative gradient of the loss function. The parameters are updated at every iteration as follows:

$$w_i \leftarrow w_{i-1} - \eta \frac{\partial E}{\partial w} \tag{2.42}$$

where η is the learning rate and $\frac{\partial}{\partial w} E$ can be obtained from the chain rule for a multi-layer NN, see Equation 2.41.

Figure 2.9 shows gradient descent being applied with a loss function, $E(w)$, on one parameter, w. The figure depicts an ideal learning rate - if the learning rate is too high, it can overshoot the minimum, and if it is too low, reaching convergence will be time consuming.

2.4.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are NNs that contain at least one *convolution* layer. CNNs are a deep learning technique developed mainly to tackle image classification problems, finding a wide range of applications from self-driving cars and facial recognition to image segmentation [10, 49].

The term convolution stems from the mathematical definition of convolution. A convolution is an operation on two functions, f and g, that produces a third function $f * g$

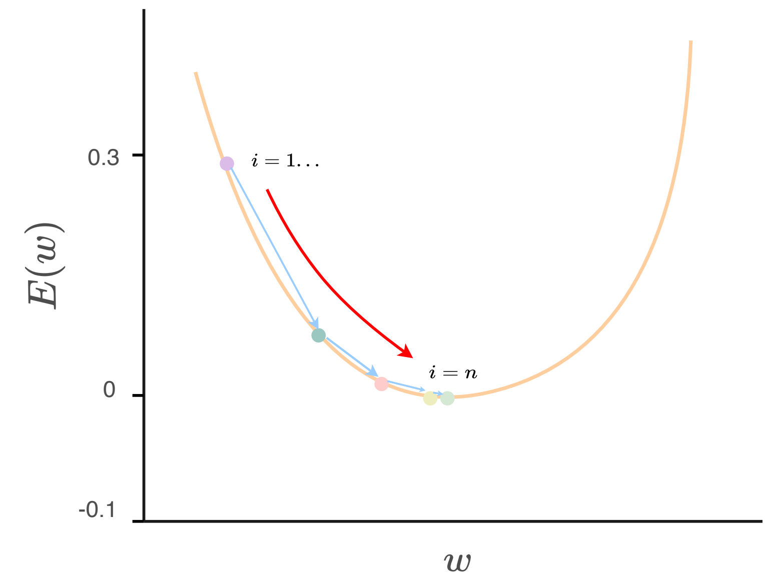


Figure 2.9: *Visual of the loss function, $E(w)$, with respect to the adjustable parameter w. At each iteration, i, w is updated and the loss is reduced.*

which represents the correlation between the two. A convolution between discrete two-dimensional matrices, f and g, can be expressed as follows:

$$y[i,j] = f * g = \sum_{n}\sum_{m} f[n,m]g[i-n,j-m] \tag{2.43}$$

The input, f, can represent an image or uniform spatial grid, and g is the kernel. Here, (i, j) are the indices of the input while (n, m) are the indices of the kernel. The kernel is "flipped" relative to the input, in the sense that as the index to the input increases, the index to the kernel decreases. Due to this, convolution is commutative, meaning that $(f * g) = (g * f)$. Hence, swapping the order of the input and kernel in a convolution operation does not change the output. While the commutative property is nice-to-have for mathematical proofs, it provides no physical significance. As such, in machine learning libraries, convolution is implemented without flipping the kernel, as defined in the equation below. Mathmatically, this is referred to as cross-correlation.

$$y[i,j] = f * g = \sum_n \sum_m f[n,m] g[i+n, j+m] \tag{2.44}$$

In a convolutional layer, each grid point in an image or matrix passes through a convolution kernel. In machine learning context, a convolutional kernel is a smaller matrix comprised of weights. The sum of the dot-products between the kernel weights and input pixels at each unique position is found. This is a scalar, and is assigned to a grid point in the output array. A deconvolution is the transpose of a convolutional layer.

The area the kernel is applied to at a given time is known as the receptive field of the neuron, and is "shone" like a flashlight across the input. For an $n \times n$ input, $k \times k$ kernel, and $s \times s$ stride, the resulting output is of dimension:

$$\left(\frac{n-k}{s} + 1, \frac{n-k}{s} + 1 \right) \tag{2.45}$$

This is depicted in Figure 2.10. Following the convolutional layer, the data is usually passed to an activation function to introduce non-linearities. This ensures that the model has more power than simpler linear regression techniques, and that sequential layers can be stacked effectively.

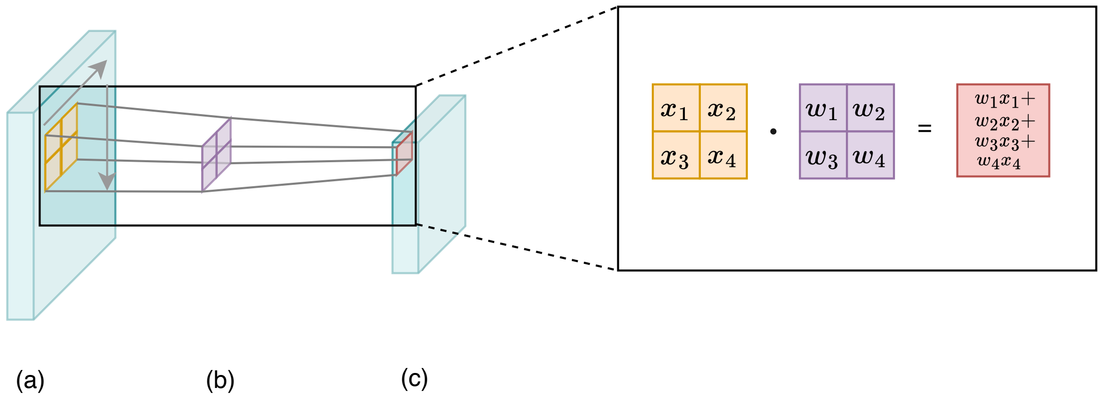


Figure 2.10: *Convolutional layer, where (b) is a 2×2 kernel applied on a patch of the input (a), and (c) is the scalar assigned to a corresponding grid point in the output matrix.*

2.4.3 Graph Neural Networks

A graph is a data structure that describes a set of entities and the connections between them. Many real life data is stored as a graph, such as social networks [32], protein-protein interactions [61], and even cosmological data [13]. *Graph Neural Networks* were developed to effectively apply ML to this data structure. This type of NN employs the same key computations as traditional NNs but in an architecture suitable for graph data structures.

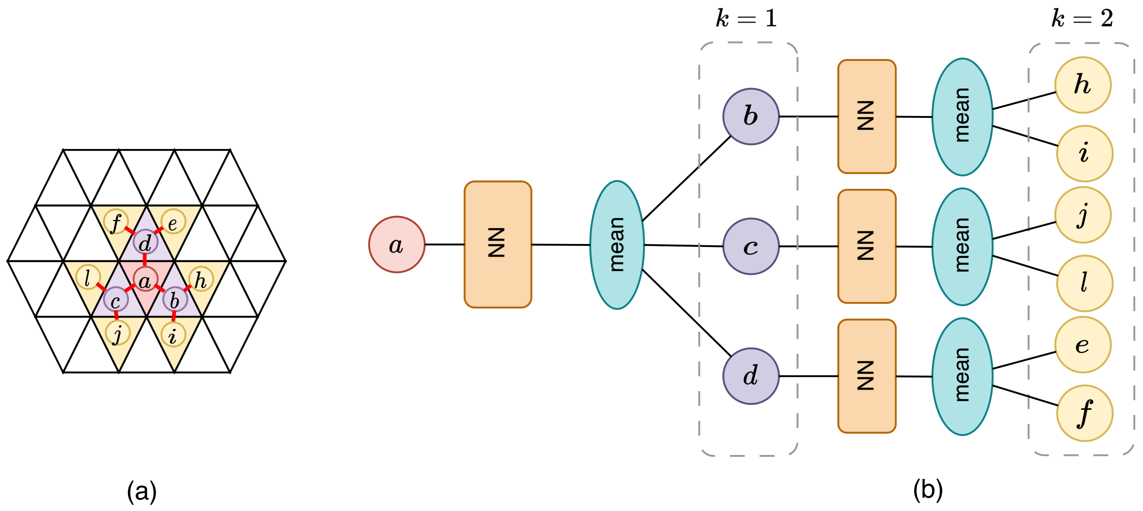


Figure 2.11: *GraphSAGE to predict a target node a. The neighbouring node features are combined using an aggregate function, and fed through a small neural network denoted by the orange NN figure. A search depth equal to 2 is depicted in this figure.*

A graph consists of edges and nodes. Nodes store information while edges describe the connectivity between different nodes. For example, a graph can be a spatial mesh, where the nodes represent a point in space and the edges represent the distance between these points. The edges might be directed based on directional dependencies, but for the scope of this work, an undirected spatial mesh will be used to exemplify the application of graph NNs.

A Graph Neural Network architecture developed by Hamilton, W. et. al., named GraphSAGE [20], is popular choice for the application of ML on graphs. This is due to its *inductive* learning approach, which allows generalization of unseen nodes, eliminating the need to provide a global structure of the graph during training and prediction.

GraphSAGE generates node embeddings through sampling and aggregating features from a nodes local neighborhood [20]. Instead of training a unique embedding vector for every single node, a set of aggregator functions are trained to learn feature information from a node's local neighbourhood. The aggregator function combines information from nodes at varying search depths relative to the target node. These search depths are denoted as k in Figure 2.11. The mean operator is the chosen aggregator function in Figure 2.11. During inference, the trained model is able to produce embeddings for new nodes by applying these learned aggregation functions.

Graph Convolutional Networks (GCNs) [25] developed by Kipf & Welling in 2017 also learn feature embeddings by analyzing neighbouring nodes. GCN are inspired by traditional CNNs, except now the convolution is a *graph* convolution which aggregates features from a local neighbourhood.

Chapter 3

Literature Review

The past decade has revealed vast advancements in the field of Machine Learning (ML), particularly in natural language processing and computer vision. The most important applications of ML employ deep learning, a subset of ML that uses neural networks with multiple hidden layers [21, 53]. Deep learning achieves great success by representing a problem in a hierarchical manner, with each concept defined in relation to a simpler concept, and further building on these simple representations to learn more abstract ones [17]. This concept is not new in fluid dynamics - methods such as Reduced Order Modelling focus on representing complex dynamics in terms of a simplified model with dominant terms to reduce the computational cost. Considering the volumes of data available from both experiments and simulations, an interest sparked in the field of deep learning for computational fluid dynamics, and many applications have been developed since [5, 43, 6, 62, 27, 60].

The potential of deep learning to improve fluid dynamics has been demonstrated in a broad range of topics, including measurement techniques in experiments [55] and control strategies for drag reduction [19]. To match the focus of this work, this review will be limited to recent progress in the field which seek to accelerate computational fluid dynamics simulations. Recent work can be grouped into solver accelerators or model replacements, which either combine numerical solvers with ML to provide a computational advantage, or use ML to learn a surrogate model, respectively. Work can be further distinguished by the choice of neural network architecture. Additionally, popular neural network structures with wide reach in fluid dynamics are discussed.

3.1 ML to infer the solution of a model

3.1.1 Convolutional neural networks

Convolutional neural networks (CNNs) have recently been recognized as a powerful model architecture for fluid dynamics. The most prominent features that contribute to their success in the field are (i) parameter sharing, where the same set of weights are used across the spatial domain in a layer, making an efficient and generalizable model; and (ii) local kernels which learn spatial patterns across neighboring inputs. This makes CNNs suitable for large spatial data where neighboring regions are correlated. The shared weights also allow for shorter training times, and require less training data to reach a target loss. As such, CNNs are a popular choice for research in deep learning for fluid dynamics.

Theurey, N. et. al. adapted a special UNet encoder-decoder architecture to analyze the accuracy of deep learning models for Reynolds Averaged Navier-Stokes (RANS) simulations of 2D airfoils [50]. The input data consisted of a mask of the domain, and the output was converged velocity and pressure profiles obtained from running a RANS simulation in `OpenFOAM` with a Spalart-Allmaras turbulence model [59]. Although the simulations were run on an unstructured mesh, the `OpenFOAM` field data was projected to a uniform grid of size (128×128) to make it a suitable input size for the CNN. The performance was measured by the L_1 loss for several test cases of unseen geometries, which was commonly less than 3%. Additional analysis was performed on model sizing. By increasing the model size from 122k parameters to 30.9m parameters, the loss decreased from 6.3×10^{-3} to 3.3×10^{-3}, and the flow patterns are captured more sharply (see Figure 7 in [50]). Similar architectures have been applied to other case studies, such as unsteady RANS in vertical axis turbines [12].

More recently, a different type of encoder-decoder, called Variational Autoencoder (VAE), was used to determine the steady flow fields across supercritical airfoils [57]. A Variational Autoencoder (VAE) consists of an encoder (which reduces the data into a latent space through sequential layers) and decoder (which decodes this latent space back into its original data size through sequential layers). The encoder determines the latent variables **z** from the input data **x** by a learned distribution $P(z|x)$; and the decoder generates similar data **x'** by sampling from this learned distribution $P(z)$. In [57], the latent space has an additional neural network which is trained to learn a mapping between the airfoil shape to the high level features of the steady flow fields. This mapping allows the solution to be predicted given only the airfoil shape, drastically reducing the pre-processing time required before model inference.

Another popular use of CNNs is for super resolution [11]. This is extremely useful in fluid dynamics, where high-resolution (HR) information of flow fields are less accessible due to limited computational or experimental resources [14]. In turbulence modelling, for example, the smallest spatiotemporal scales must be resolved in order to achieve accurate results.

SURFNet [37] is a super-resolution flow network for turbulence modelling that uses incremental transfer learning to resolve flow information at fine scales. A coarse model is trained on a relatively inexpensive set of low-resolution data, with each array of size 64×256, for 10 geometries and 9 variations of each geometry. The learned feature maps from the coarse model are transferred to higher resolution models, incrementally, up to a resolution of size 2048×2048, as depicted in Figure 3.1. With transfer learning, as the model incrementally increased in size, fewer data was required to achieve accurate results. The highest resolution model, 2048×2048, only required one additional geometry for training. This reduced the data collection and training time by $3.6\times$ and $10.2\times$, respectively. As such, transfer learning is a promising approach to train large machine learning models for prediction on fine grids.

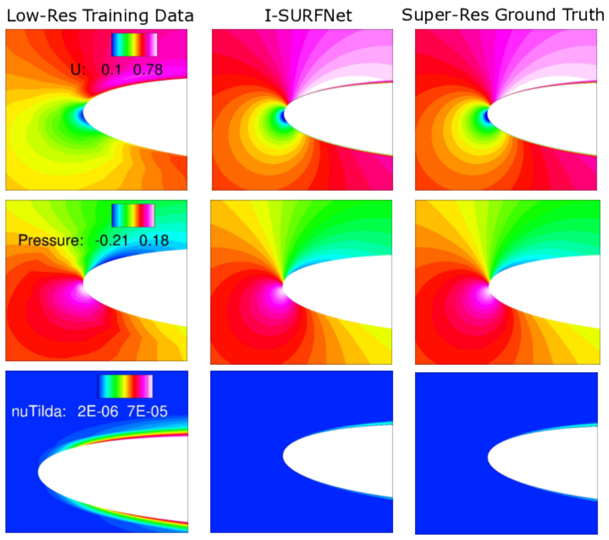


Figure 3.1: *Results from SURFNet [37] © [2021] IEEE. The framework outperforms the sole use of a coarse model, while still having the majority of the training data to be coarse, reducing the dataset collection time.*

3.1.2 Graph neural networks

The main drawback of CNNs is their need for structured data - i.e., regular Euclidean data like pixels in an 2D image. Traditional convolutional kernels operate on structured data, without capturing possible varying spatial information between nodes. In most fluid simulations however, information is stored in an unstructured mesh that represents the discretized physical space of an arbitrary geometry. Therefore, to capture the dynamics of a realistic industrial or environmental setting, the data for the model should be stored in an appropriate data structure. In machine learning, this can be accomplished by using new architectures designed for non-Euclidean data, specifically, graph neural networks.

Extending deep neural networks to non-euclidean data is an emerging research area called *geometric deep learning* [4]. Under this field, graph neural networks (GNNs) have received enormous attention, and their ideas have been applied to several deep learning in CFD applications [9].

In [41], a data-driven framework based on a graph convolutional network (GCN) is developed to predict fluid dynamics on a non-uniform mesh. This method is validated on a few cases of internal flow, achieving an acceptable accuracy with a three-order speedup compared to numerical methods.

3.1.3 Physics-informed neural networks

Physics-informed machine learning integrates domain knowledge directly into machine learning algorithms, ensuring physical consistency in the model output. This idea was first established by M. Raissi et al. [43] in 2019. To demonstrate this concept, they considered the following non-linear partial differential equation (Burgers equation) [30] describing a field u:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \frac{0.01}{\pi}\frac{\partial^2 u}{\partial x^2} = 0 \tag{3.1}$$

The field, $u(x,t)$, is predicted using a neural network, $G(x,t)$, that takes a point in space (x) and time (t) and returns the value of the field, u, at that point. Now consider a corresponding neural network, $f(x,t)$, defined as the left hand side of Equation 3.1. This neural network also takes x and t as input parameters, computes u from $G(x,t)$, and then computes the gradients of u with respect to x and t to formulate equation 3.1. This creates shared parameters between the two networks, $f(x,t)$ and $G(x,t)$, and a coupled loss is computed [43]:

$$Loss(t_f^i, x_f^i, t_u^i, x_u^i, u^i) = \frac{1}{N_f}\sum_{i=1}^{N_f}|f(t_f^i, x_f^i)|^2 + \frac{1}{N_u}\sum_{i=1}^{N_u}|G(t_u^i, x_u^i) - u^i|^2 \tag{3.2}$$

In equation 3.2, $\{t_u^i, x_u^i, u^i\}$ are points sampled from the initial time and boundaries of the domain, which are already known. On the other hand, $\{t_f^i, x_f^i\}$ are sampled collocation points inside the domain, used to train $f(x,t)$. The results of this method can visualized in Figure A.6. of [43], where it is evident that the physics-informed neural network method produces very accurate results.

Physics-Informed Neural Networks (PINNs) received a tremendous reaction from the computational fluid dynamics community due to their computational advantages and physically consistent results. Buaria et. al. developed deep PINNs to resolve small-scale dynamics of fluid turbulence [7], and Nath, K. et. al [36] used PINNs to predict gas flow dynamics in diesel engines. These are just a few examples of the many of applications of PINNs developed in the last few years [24].

3.2 ML for the development of hybrid solvers

Despite the vast amount of research in this area, development is needed for deep learning strategies within multiphysics solvers to prove useful to engineers. Multiphysics solvers like `OpenFOAM` [59] and `NGSolve` [47] have a large user base due to their precise and physically consistent results. As such, efforts have been directed to benefit from the two techniques by combining both the speed of deep learning and the precision of traditional solvers.

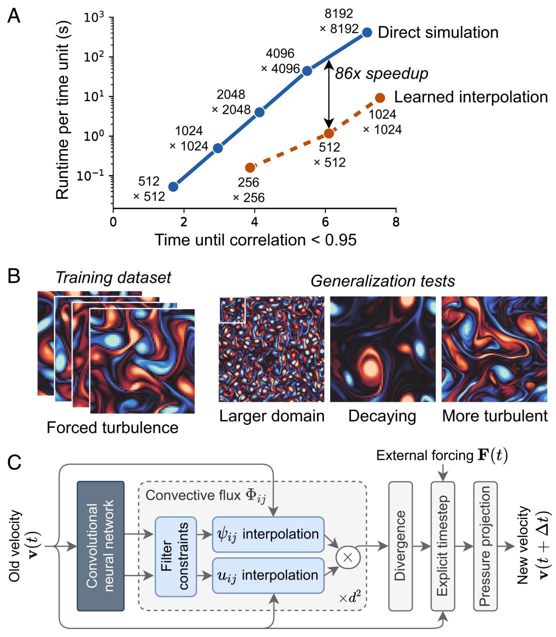


Figure 3.2: *Summary of results from [26]. The framework was tested on Re* = 100000 *Large Eddy Simulations using up to an 8* $\times$ *coarser grid with a 40-fold speedup.*

A recent trend in hybrid techniques is to partially replace the physics solver with a NN to improve performance. The replaced components can either be ones most affected by discretization loss [26] or the most computationally expensive steps in the numerical algorithm [51, 2]. Kochkov, D. et. al [26] aimed to accelerate the computation of Direct Numerical Simulations (DNS) of turbulent flows without compromising on accuracy or stability. In DNS, resolving the smallest spatiotemporal features required for accuracy implies a computational scaling of Re^3. To alleviate this cost, this work developed a hybrid NN-physics solver that can resolve turbulent flows on an order of magnitude *coarser* grid than usually required for the same accuracy and stability. This was accomplished by using a NN to replace components of the algorithm most effected by resolution loss, which, in this case, was the computation of convective flux $\mathbf{\Phi}$. The explicit timestepping and pressure projection were then computed with traditional methods. The NN and numerical method were written in the JAX framework to support reverse-mode automatic differentiation, allowing for the optimization of the entire algorithm. To summarize, this end-to-end process showed computational speedup with accuracy and stability, while also obeying all physical laws by solving the turbulence equations on coarser grids (see Figure 3.2).

Another approach to hybridize NN and physics solvers to improve performance is to replace the most computationally expensive step of the solver by a neural network. One solution method for incompressible Navier-Stokes equations involves computing a pressure projection. This involves resolving the Poisson equation to correct the velocity field. The Poisson equation is written in matrix form as a large sparse linear system $\boldsymbol{Ap_t = b}$, where p_t is the unknown pressure field, $\mathbf{A}$ is the discrete matrix for the Poisson operator, and $\mathbf{b}$ is the source term proportional to the divergence of the velocity field. Due to the large number of free parameters, several iterations are required to obtain a relatively small residual, resulting in a computational burden.

The use of data-driven methods to accelerate the pressure projection step was pioneered by Tompson et. al. [51] in 2016. They solved the Navier-Stokes equations for an inviscid fluid using the euler equations, where advection was computed traditionally and a CNN was used to infer p_t, the unknown pressure field. The training of the CNN was unsupervised, as the divergence of the updated velocity field was directly minimized with the loss function:

$$loss = \sum w_i \{\nabla \cdot (u_{t+1} - \frac{1}{\rho} \nabla p_t)\}^2 \tag{3.3}$$

The performance of the model was tested on plume simulations against a Jacobi iterative method, and similar results were achieved. Although the results were visually impressive, there was a lack of analysis on the accuracy.

Ajuria-Illaramendi et. al. extended Tompsons work with accuracy as the main objective [2]. Using the same CNN architecture, they performed a more rigorous testing phase by increasing the Richardson number: a dimensionless number corresponding to the ratio of buoyant forces to momentum-driven forces. It was found that as the Richardson number increased, the NN based method produced significantly less accurate results than the Jacobi method. As such, they developed a combined CNN-Jacobi method for the pressure projection step, where, if the corrected velocity divergence is higher than a certain threshold, the Jacobi method is used instead. Building on this idea, Weymouth, G.

[60] developed a CNN smoother for the geometric multigrid (GMG) method, minimizing the velocity-divergence over the recursive GMG procedure, achieving 2 to 3-fold speedup compared to standard GMG methods.

Another method for hybrid neural networks was recently introduced by Jeon, J. et al. who developed a framework for ML–CFD cross-coupling computation with `OpenFOAM`, an open-source finite volume method solver for CMP. CFD calculations are performed with `OpenFOAM` on an initial time series, and if a desired tolerance is reached, the next time series is predicted with a neural network [23]. Otherwise, the next time series is calculated with CFD. This process is repeated over a period of time, and the neural network parameters are continuously updated with the latest time-series results. This framework was found to stabilize the residual over extended periods of time, and provided a $1.8\times$ speed-up for a laminar flow case.

CFDNet [38] is a cross-coupled physics-solver-NN framework that accelerates the convergence of RANS simulations. CFDNet is designed to predict three main physical properties of the fluid: the velocity, pressure, and eddy viscosity. This is accomplished by training the CNN on a variety of use cases with immersed geometries in a two-dimensional domain. A key component that distinguishes CFDNet from a simple CNN is that the CNN prediction is fed *back* into `OpenFOAM` to refine the solution and ensure it meets the same residual constraints as a physics solver, as depicted in Figure 3.3. This ensures that the final output of CFDNet is consistent with traditional physics solvers by obeying physical laws, which is an important criteria for scientists and engineers. To test CFDNet, the process is evaluated on cases that were not trained on - specifically, immersed cylinders, airfoils, and ellipses of different aspect ratios; as well as wall-bounded channel flows. It was found that, by using a CNN and refining the prediction with the `OpenFOAM` solver, the same residual tolerances are met in a fraction of the time. CFDNet achieved $1.9 - 7.4\times$ speedups on steady laminar and turbulent RANS flows, as depicted in Figure 3.4.

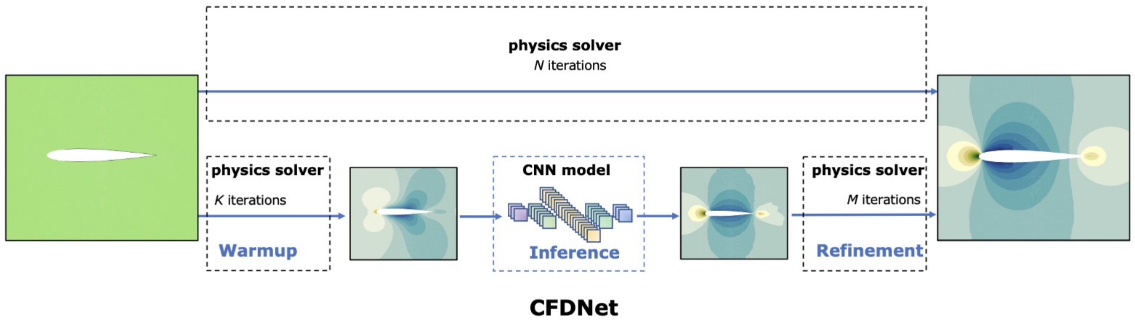


Figure 3.3: *The architecture of the CFDNet framework [38].*

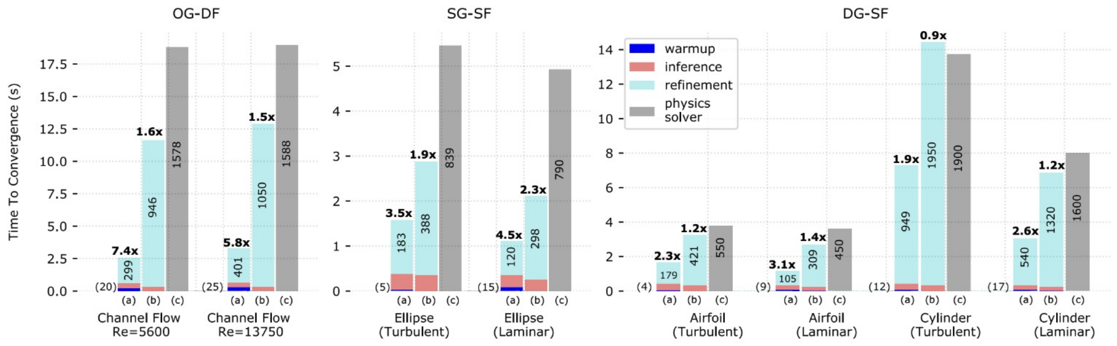


Figure 3.4: *CFDNet speedup results for various benchmark problems in both laminar and turbulent flow regimes [38].*

Chapter 4

Research Methods

This chapter contains the methods used to implement the results of this thesis. First, the computational tools used to solve the incompressible navier stokes (INS) equations on a two-dimensional domain are described. The machine learning implementation is then detailed, and finally, the data generation process is discussed.

4.1 Simulation of Incompressible Navier Stokes

The objective of this work is to accelerate the simulation of INS equations in a two-dimensional domain for both laminar and turbulent flows. As such, solving the governing transport equations with numerical methods was the first step of this project. CMP software contains the numerical methods required to solve fluid flow problems. In this work, the CMP software used is `OpenFOAM` [59], an open source CMP package based on the finite volume method (FVM).

In order to perform calculations to solve the governing equations of a problem, the problem must be adequately described with three main components [18]:

1. A computational mesh representing the domain occupied by the fluid
2. Discretized equations and algorithms to calculate the flow fields of interest
3. Boundary and initial conditions for the flow fields

Geometry creation

The first step was to generate the problem domain. This was accomplished with `GMSH` [15], an open source 3D finite element mesh generator. While geometries can be defined directly in `OpenFOAM` in the appropriate user files, `GMSH` provides a more flexible framework better suited to generate customisable designs.

Figure 4.1 portrays the geometry used for training, a wall-bounded flow with an immersed body. The flow originates from the inlet, depicted by the arrow, and is only present in the x-dimension.

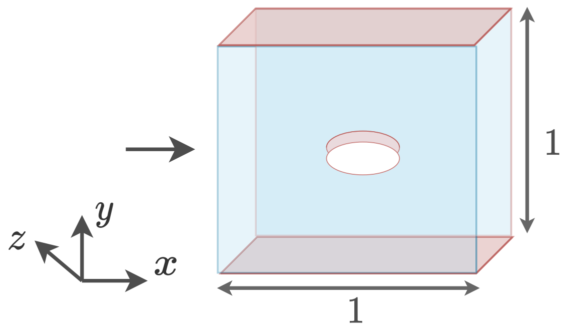


Figure 4.1: *Geometry for training. Walls are depicted by the red and the blue faces are the front and back, set to `empty` in `OpenFOAM` to ensure equations are only solved in the (x,y) dimension rendering the problem as two-dimensional*

Boundary Conditions

Computational Fluid Dynamics problems are defined in terms of initial and boundary conditions. A table summarizing the boundary conditions used to run the INS simulations are presented below.

	p	u	nuTilda	nut
inlet	`freestream` 0	`freestream` $(u_{ref}, 0, 0)$	`freestream` $3\nu_{fluid}$	`freestream` ν_{fluid}
outlet	`freestream` 0	`zeroGradient`	$3\nu_{fluid}$	`freestream` ν_{fluid}
walls	`zeroGradient`	`fixedValue` $(0, 0, 0)$	`fixedValue` 0	`wallFcn` 0
front and back	`empty`	`empty`	`empty`	`empty`

Table 4.1: Boundary conditions in `OpenFOAM`

The `fixedValue` condition is a Dirichlet boundary condition where the values of the faces at the boundary are equal to the value specified, i.e., $\phi_b = \phi_{ref}$. The `zeroGradient` condition sets the normal gradient value to 0, i.e., $\frac{\partial}{\partial n}\phi_b = 0$. The `freestream` condition is a hybrid `fixedValue` and `zeroGradient` condition - for fluid flowing out of a boundary face, `zeroGradient` is applied, otherwise, `fixedValue` is applied [18]. The `wallFcn` condition adds a wall constraint to the eddy viscosity for turbulent flows. In this case, the `nutUSpaldingWallFunction` from `OpenFOAM` was used. The `empty` conditions were applied to enforce a two-dimensional problem. The value ν_{fluid} refers to the viscosity of the fluid, which was set to 1×10^{-5}.

4.1.1 SIMPLE algorithm

The SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm is a widely used iterative procedure for calculating pressure and velocity in steady, incompressible flows. Patankar and Spalding [8] introduced this algorithm to tackle the difficulties associated with handling pressure-velocity coupling in transport equations. The key idea behind the SIMPLE algorithm is to decompose the velocity and pressure solves into two separate steps [54].

Consider the continuity 2.4 and momentum 2.14 equations defined in Chapter 2. Since SIMPLE is most commonly applied to steady problems, the transient term of the momentum equation will be set to zero. The term that accounts for external effects, $\rho\mathbf{g}$, will be neglected [42]. Hence, the equations can be reformulated as:

$$\nabla \cdot \mathbf{v} = 0 \tag{4.1}$$

$$\mathbf{v} \cdot \nabla\mathbf{v} - \nabla \cdot (\nu\nabla\mathbf{v}) = -\nabla p \tag{4.2}$$

In Equation 4.2, p is the kinematic pressure $(\frac{P}{\rho})$ and ν is the kinematic viscosity $(\frac{\mu}{\rho})$. Just as in Equation 2.14, $\mathbf{v}$ is the velocity field in (x, y, z). Hence, when expanded in Cartesian notation, 4.2 becomes three separate equations for three unknowns (v_x, v_y, v_z). The convection term in 4.2 is non-linear, introducing more difficulty when attempting to solve the system of equations. On top of this, the computed velocity field must satisfy the continuity equation.

Given a problem on an arbitrary domain, the above equations can be re-written into matrix form, where M is a matrix storing known coefficients that result from discretization with the FVM:

$$M\mathbf{v} = -\nabla p \tag{4.3}$$

The coefficient matrix is then reformulated into diagonal (A) and off-diagonal (H) terms. Hence, the matrix form of the discretized momentum equation is:

$$A\mathbf{v} - H = -\nabla p \tag{4.4}$$

This can be rearranged for $\mathbf{v}$ and substituted into the continuity equation to obtain a Poisson equation for pressure. Putting this together, the multi-step solution procedure for the SIMPLE algorithm can be visualized in Figure 4.2:

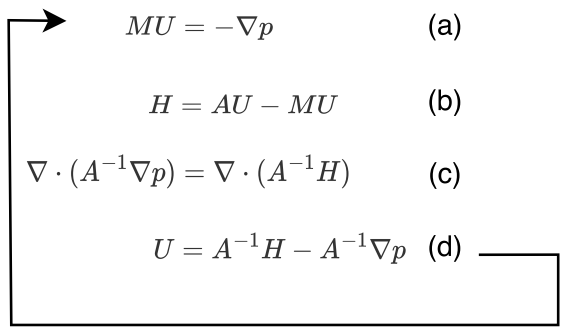

$MU = -\nabla p$ (a)
$H = AU - MU$ (b)
$\nabla \cdot (A^{-1}\nabla p) = \nabla \cdot (A^{-1}H)$ (c)
$U = A^{-1}H - A^{-1}\nabla p$ (d)

Figure 4.2: *Iterative SIMPLE algorithm. Given an initial guess for pressure in (a), a velocity field is calculated. The velocity is split into diagonal and non-diagonal terms in (b), to solve for a corrected pressure in (c) which satasfies the continuity equation. This corrected pressure is applied to (d) to correct the velocity. A residual is computed comparing the two sides of (a), and the procedure is iteratively repeated until convergence is reached.*

The steps of the SIMPLE algorithm are summarized below:

1. Solve the momentum equation (a) with an initial guess for the pressure to obtain a velocity field.
2. Given this velocity field, solve the Poisson equation (c) for a corrected pressure field.
3. Correct the velocity field with this new pressure field through (d), now the velocity satisfies the continuity equation.
4. Compute residual, if greater than desired value, repeat the process to compute the next iteration (see arrow in Figure 4.2).

4.1.2 Spalart-Allmaras Turbulence Model

In Reynolds-Averaged Navier Stokes turbulence models, an additional term is needed to close the momentum equations. Closing the momentum equation refers to approximating the Reynolds stresses (see Equation 2.22) in terms of the mean velocity components. One way that this is accomplished is through eddy viscosity models. This idea was proposed by Boussinesq back in 1877 [46], who stated that the momentum transfer caused by turbulent fluctuations can be modelled with the eddy viscosity, ν_t, through the following relationship (in a two-dimensional domain):

$$\overline{u'v'} = \nu_t \left(\frac{\partial \overline{U}}{\partial y} + \frac{\partial \overline{V}}{\partial x} \right) \tag{4.5}$$

where u and v are the x and y components of velocity, respectively. u' denotes the fluctuating component while $\overline{U}$ denotes the mean component. The Spalart-Allmaras turbulence model contains one additional equation, which is the transport equation of the modified eddy viscosity, $\tilde{\nu_t}$. This is approximated by computing $\tilde{\nu_t}$ as a quartic function in the viscous sublayer. In this work, the Spalart-Allmaras model was used as the turbulence model to remain consistent with the procedure outlined in CFDNet [38].

Consider Figure 4.3. The behaviour of the turbulent viscosity changes drastically in regions close to the wall. In the viscous sublayer ($y^+ < 5$), depicted by the red dashed line, the profile is described by a quartic function. To resolve this region numerically, the mesh has to be extremely fine considering fields must vary smoothly on the mesh.

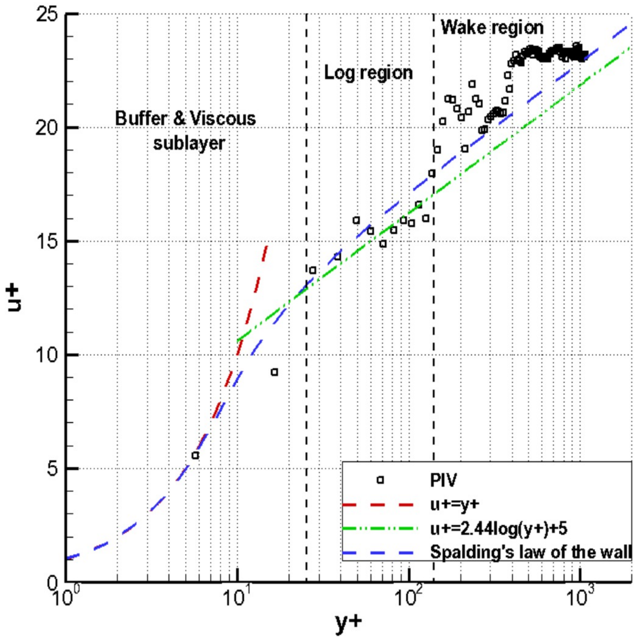


Figure 4.3: *Figure from [31] demonstrating how the turbulent viscosity behaves close to a wall. In the viscous sublayer, and varies with* $(y^+)^4$ *where* y^+ *is a dimensionless wall unit describing the distance from the wall.*

To overcome this, a new variable is introduced, $\tilde{\nu_t}$, which is a function of the turbulent kinematic viscosity ν_t. In the viscous sublayer region, ν_t is formulated as:

$$\nu_t = \tilde{\nu_t} f_{v1} \tag{4.6}$$

where f_{v1} is a cubic function. This accounts for the quartic profile near the wall, and the goal is to determine a $\tilde{\nu_t}$ that is identical to ν_t. As such, in the Spalart-Allmaras model, only one additional equation - the transport equation of $\tilde{\nu_t}$ - is needed to close the RANS turbulence equations. This transport equation is provided below 4.7 [38]. $C_{b1}, C_{b2}, C_{w1}, \sigma$ are constants obtained experimentally. $\tilde{S}$ and f_{t2} are model-specific terms.

$$\bar{v}_i \frac{\partial \tilde{\nu}}{\partial x_i} = C_{b1}(1 - f_{t2})\tilde{S}\tilde{\nu} - \left[C_{w1} f_w - \frac{C_{b1}}{\kappa^2} f_{t2}\right]\left(\frac{\tilde{\nu}^2}{d}\right) + \frac{1}{\sigma}\left[\frac{\partial}{\partial x_i}\left((\nu + \tilde{\nu})\frac{\partial \tilde{\nu}}{\partial x_i}\right) + C_{b2}\frac{\partial \tilde{\nu}}{\partial x_i}\frac{\partial \tilde{\nu}}{\partial x_i}\right] \tag{4.7}$$

4.2 Neural Network Creation

The machine learning algorithms used in this work were developed with Pytorch [40], an open-source machine learning framework based on the Python programming language. Along with Tensorflow and Keras, Pytorch is a popular platform for deep learning research. Its popularity stems from its balance of both speed and usability: it provides structure that makes models, data processors, and optimizers easy to build, while still providing flexibility in its implementation to accelerate performance.

The machine learning implementation was inspired by CFDNet [38], and hence, a similar encoder-decoder neural network architecture was developed in this work. In this context, encoder refers to a series of convolutional operations while a decoder is a series of transposed convolutional operations which occur after the encoder.

A Pytorch convolutional layer is defined in the `Conv2d` method, as depicted in Listing 4.1 below. The `in_channels` are the number of channels coming *into* the layer. For instance, in the first layer of the model, the number of channels can either be 4 for the turbulent network $(u_x, u_y, p, \tilde{\nu})$ or 3 for the laminar network (u_x, u_y, p). The `out_channels` is the number of channels coming *out* of the convolutional layer. To effectively capture feature information, the `out_channels` should always be large compared to the `in_channels`.

```
import torch
# Initialize 1 random 3-channel tensor of size (64, 256)

input = torch.rand(1, 3, 64, 256)
conv_layer = torch.nn.Conv2d(3, 32, (2,8), (2,8))
output_encoder = conv_layer(input)

# View output size after encoder convolution kernel and stride
>>> output_encoder.shape
torch.Size([1, 32, 32, 32])

```

```
# Now, apply transpose convolution to decode the output back into its
   original shape
deconv_layer = torch.nn.ConvTranspose2d(32, 3, (2,8), (2,8))
output_decoder = deconv_layer(output_encoder)

>>> output_decoder.shape
torch.Size([1, 3, 64, 256])
```

Listing 4.1: *Example of single layer encoder-decoder operations in pytorch*

The optimizer used to update the neural network parameters in this work was **RMSProp**. This was initially chosen to stay consistent with the results of CFDNet. However, when other optimizers were explored further on in this work, RMSProp always provided the best performance. As such, the RMSProp optimizer was used in all sections of this work. RMSProp avoids computing very large or very small gradients, ensuring stable results during training.

RMSprop accomplishes this by normalizing the gradient using a moving average of squared gradients. This normalization decreases the step size for large gradients and increases the step size for vanishing gradients. RMSProp essentially treats the learning rate as an adjustable hyperparameter rather than a fixed value, allowing convergence to be reached faster.

RMSProp updates the neural network parameters w with the following set of operations:

$$g_{i+1} \leftarrow \alpha \cdot g_i + (1-\alpha)(\nabla_w E(w))^2 \tag{4.8}$$

$$w_{i+1} \leftarrow w_i - \eta \frac{\nabla_w E}{\sqrt{g_{i+1}} + \epsilon} \tag{4.9}$$

where α is a decay rate, η is the learning rate, and $\nabla_w E(w)$ is the gradient of the loss function with respect to the weight of interest, w. Pytorch sets a default value for α equal to 0.99. In this work, the default value was used for alpha.

4.3 Data generation

In order to train the network, the simulation data had to be structured in a tensor format suitable for a Pytorch model. When handling information stored in meshes, this can be accomplished in two ways:

1. For an unstructured mesh, the data must be projected to a uniform grid. This is accomplished using a built-in `OpenFOAM` interpolation function, `probes`, that returns an interpolated value for the field of interest given any point on the domain.

2. If the grid is structured but not uniform, interpolation methods are not necessarily needed, instead the information can be stored in a spatially consistent manner.

Both of these processes involve common steps for setting up the simulation:

1. Develop mesh in `GMSH` for case study

2. Run the `simpleFoam` solver for the case study, with the following criteria:

   (a) Set a Reynolds number equal to 30 for the laminar problems or 6×10^5 for the turbulent problems by specifying an appropriate u_{ref} value

   (b) Set a low tolerance of 1×10^{-5}

   (c) Obtain (u, v, p, $\tilde{\nu}$) fields

Interpolation

Once all the cases have run, the training data is ready to be processed. The training data is generated as follows:

1. Given the desired number of uniform axial points and uniform vertical points, generate (n, m) points based on the `OpenFOAM` geometry case dimensions

2. Generate numpy arrays for training given these uniformly sampled values

3. Automatically determine u_{ref} from `OpenFOAM` files, and non-dimensionalize the field values with the following equations: $u = \frac{\mathbf{u}}{u_{ref}}, p = \frac{\mathbf{p}}{u_{ref}^2}$

The resulting training data is a numpy tensor of shape $[K, C, N, M]$, where K is the total number of iterations to reach convergence, C is the number of distinct flow parameters, and (N, M) is the size of the two-dimensional domain. See Figure 4.4 for a visualization of the training data tensor. In this work, all cases are structured as a 2D grid of dimension 64×256.

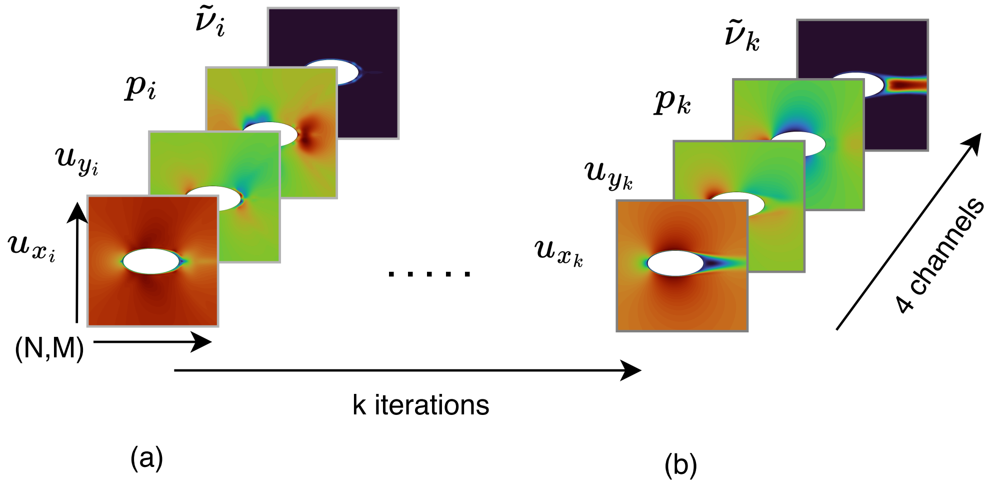


Figure 4.4: *Figure visualizing the training data tensor size. For a simulation that takes k iterations to converge, with each iteration stored in a grid of size (N, M), with C channels (4 in this case) the overall tensor size for pytorch would be $[K, C, N, M]$. (a) is an input tensor while (b) is the target tensor.*

Chapter 5

Results

5.1 Validation of CFDNet

The motivation of this project is to accelerate current computational fluid dynamics methods with the help of machine learning. After exploring literature on the topic, the validation of a CFDNet [38] - mentioned in Chapter 3 and again summarized below - was chosen as a starting point.

In engineering design, validation plays a key role in the development and evaluation of new methods [3]. As such, the first phase of this project consisted of the validation of results from CFDNet.

Summary of CFDNet

The goal of CFDNet is to overcome key limitations of other work in the field of ML in Computational Fluid Dynamics. Firstly, DL is commonly used to predict converged flow fields as an end-to-end surrogate without explicitly adding conservation laws [57, 50]. This means that the output does not necessarily meet the constraints of traditional physics-based solving algorithms, which is an important criteria for scientists and engineers. Secondly, most work in the field only provide information on a subset of flow variables rather than all that are present in the problem, which paints an incomplete solution [50, 48]. Lastly, a majority of approaches lack generalization abilities, usually focusing on one type of geometry or flow case. CFDNet tackles these challenges by developing an end-to-end framework that leverages both DL and traditional physics solvers to accelerate Reynolds-Averaged Navier Stokes problems on different domains.

Case studies

The following case studies were chosen to train the CFDNet framework and to validate their findings. These are common benchmark problems in engineering, with relevant applications in industry and research. As such, the datasets and models used for this work - both in the validation section and beyond - were generated with these case studies as an inspiration.

Wall-bounded flows

The first case study in CFDNet is wall-bounded, two-dimensional turbulent flows. Many efforts have been dedicated to this case, as confined geometries like pipes and channels are common in industry and thoroughly studied in the design phase. Simulations are particularily important when studying turbulence in these geometries, as obtaining experimental measurements can be challenging in wall-bounded designs. The physical effects of the wall and the limitations it presents on the implementation of sensing technologies causes some of these challenges.

As such, one application of CFDNet is to accelerate the exploration of the design space for different Reynolds (Re) numbers in wall-bounded flows. For the scope of this study, models were only developed for wall-bounded flow around immersed bodies in both laminar and turbulent regimes. This is because flow around immersed bodies captures the challenges of wall-bounded flows described above, while adding a layer of complexity by introducing an immersed body.

Flow around immersed geometry

The study of flow around solid bodies is an extensive field of research due to its broad uses in engineering; such as for determining the best structure for a bridge [33], or designing high-performance vehicles [39]. In aerodynamics, for instance, knowledge on the flow behaviour such as drag and lift around wings helps engineers optimize the shape and surface contours of vehicles, improving fuel efficiency, maneuverability, and enhancing overall performance. This makes the study of flow around geometries a good candidate for the application of ML.

Since the flow around solid bodies is an important research problem, CFDNet is applied to flows around geometries in both laminar and turbulent flow regimes. Two training sets were generated for this case study:

1. Laminar flow around solid bodies with constant $Re = 30$
2. Turbulent flow around solid bodies with constant $Re = 6 \times 10^5$

The training sets consist of six different ellipses as shown in Figure 5.1. The different ellipses are obtained by changing the aspect ratio (the ratio of the vertical to the horizontal semi-axis length) from 0.1 to 0.7. The geometries were generated and meshed using GMSH. See Chapter 4 for further explanation on the data generation process.

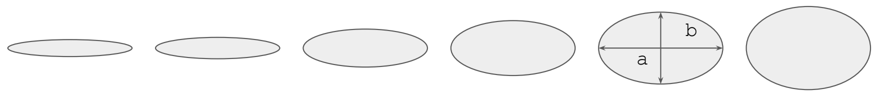


Figure 5.1: *Use cases for training, as inspired by [38]. The aspect ratio, $\frac{a}{b}$, varies from 0.1 to 0.7.*

Simulation details

The training dataset is generated by solving the Reynolds-Averaged Navier Stokes equations until steady state with the SIMPLE algorithm in `OpenFOAM` [59]. For the turbulent case studies, the Spalart-Allmaras one equation turbulence model was used. Since the Spallart-Allmaras model is a RANS model, a steady state is reached by computing the time-averaged flow components. As such, the converged solution in the turbulent flow cases is the steady solution.

Residual details The simulations are run until steady state, which, in this case, means reaching a residual value of 1×10^{-6} and 1×10^{-4} for the velocity and pressure, respectively, and 1×10^{-5} for the modified eddy viscosity.

Turbulence Properties The discrete form of the transport equations are numerically solved on a structured grid with appropriate boundary conditions. The discretization scheme for the gradient terms of the equations are computed using a second-order, least-squares interpolation method using neighbouring cells. The gradient schemes specify the handling of the gradient terms of the equation. For the convection and the modified turbulent viscosity terms, an unbounded, second-order, upwind-biased scheme is used [38]. The diffusion terms of the transport equations are evaluated with Gaussian integration, along with a linear interpolation method for calculating viscosity [38].

In order to implement this in an `OpenFOAM` environment, the `fvSchemes` file in the `system` directory must be modified to reflect the desired residuals and schemes specified above. Please see Appendix 1 for access to the `fvSolution` and `fvSchemes` files used in this work.

Machine Learning Model

The ML model used in this section of the results - the validation of CFDNet - uses the same network architecture as CFDNet [38]. This architecture is shown in the figure below.

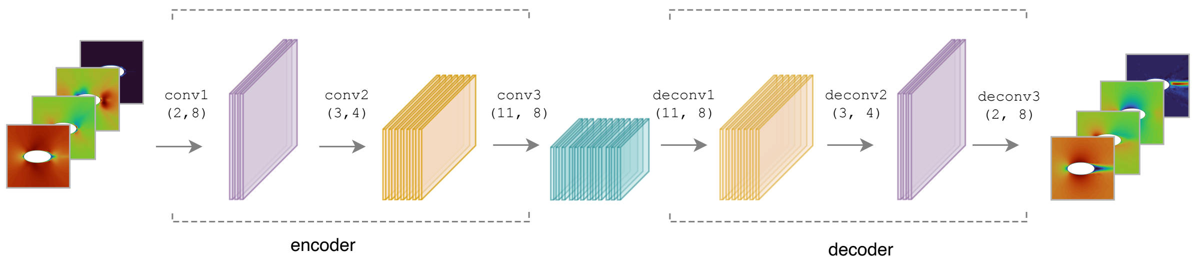


Figure 5.2: *Machine Learning model used in this work, inspired by [38]. The tuples under each convolution in the figure state the kernel and stride size at that layer.*

The encoder-decoder architecture of the network is suitable for this study, as it allows the input data to be efficiently represented in a latent space, significantly reducing the computation time for training and validation. Additionally, a CNN is well-suited for data on a grid as it captures both local and global spatial patterns. Refer to Section 2.4 for further details on the ML architecture.

The convolution kernel size for every layer is depicted in Figure 5.2. The stride size is equal to the kernel size. After the first and last layer of the neural network, the `PReLU` [22] activation function is applied. This activation function is chosen over `ReLU` because it is expected to have data values that are not exactly between $(0, 1)$ after non-dimensionalization. Some of the values are below 0 and other values are greater than 1. If `ReLU` was used, it would zero out the negative values generating inconsistent output results. The internal layers apply `tanh` activation functions after every output [38].

Results

In order to validate CFDNet, reproducing their key results was first attempted. The major claim that CFDNet makes is that their coupled NN-physics-solver framework reduces the computational time required to simulate 2-dimensional INS problems without compromising on accuracy. Additionally, they claim that the warm-up phase significantly increases the overall speed-up to convergence. In this context, warm-up refers to running a few iterations with the physics solver prior to neural network usage. The input to the neural network is therefore an intermediate iteration, and not an initialization. In order to ensure physically consistent results, the refinement stage is needed. Refinement consists of running the physics solver until the desired residual is reached with the neural network prediction as the initialization. Although the NN produces outputs with relatively low residuals (1×10^{-3} - 1×10^{-2}), it takes several `OpenFOAM` iterations to drop a residual an order of magnitude. The majority of the compute time is therefore dedicated to the refinement stage.

Figure 5.3 is grouped by interpolative (subset geometry) and extrapolative (different geometry) test cases. The left and right groups in these test cases are laminar and turbulent flow regimes, respectively.

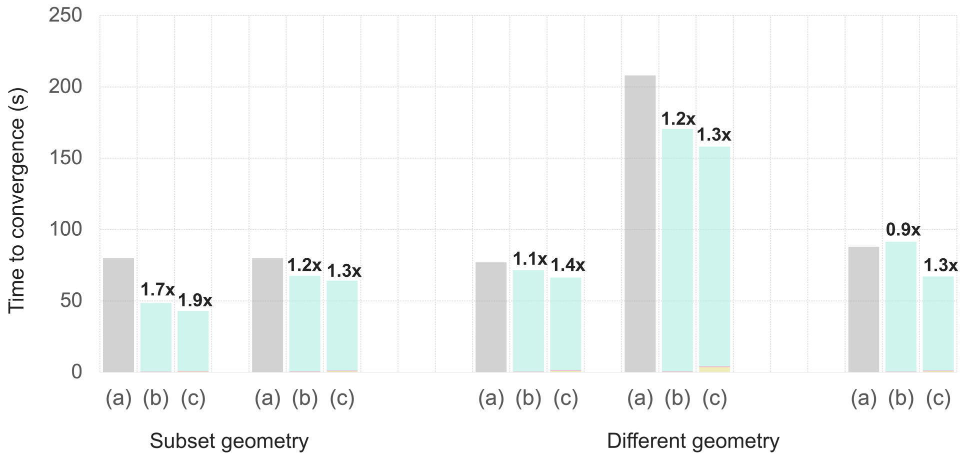


Figure 5.3: *Validation of CFDNets speedup results. (a) is the physics solver only, (b) is the NN + refinement without warmup, and (c) is the warmup + NN + refinement. The cyan, pink, and yellow bars are the times consumed by the physics solver, NN, and warmup, respectively.*

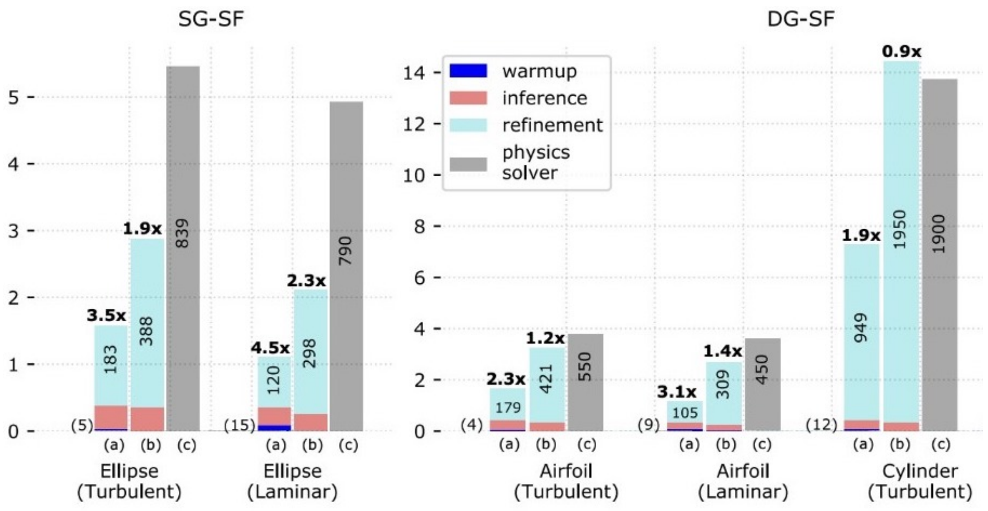


Figure 5.4: *CFDNet [38] speedup results. (a) is the warmup + NN + refinement, (b) is the NN + refinement without warmup, and (c) is the physics solver only.*

While the exact speedups were not obtained, it can be concluded that the method developed follows the same general trends as the results shown by CFDNet. It is clear that the majority of compute time is dedicated to the refinement stages, and that warmup significantly improves the NN output. Additionally, the NN output produces poorer results for turbulent extrapolative test cases of different geometries when warmup is not introduced, which is consistent with the findings of CFDNet. Another important aspect to note is that the initialization process was not described in CFDNet, so it is assumed that they initialized with zero. However, it is common to initialize the INS equations by solving their linearized version: the *Stokes* equations. This provides a suitable initial guess to the iterative solver, and usually reduces the time to convergence. Hence, it is unknown whether the same magnitude of speedup in CFDNet would be achieved if the solution to the Stokes equation was used to initialize instead.

It is clear from Figure 5.3 that inference, which includes data processing times, takes significantly less computational time than the physics solver. This inspired the idea of re-using the neural network when possible to further reduce compute time, leading to the next section of results.

5.2 Recursion

Once a trained model was obtained during the validation phase, the benefit of warm-up and whether it is necessary was explored. Warm-up provides the neural network with an intermediate SIMPLE iteration, which is closer to the solution than an initialization, and this was found to significantly reduce time to convergence [38]. If warm-up is needed to achieve better neural network output results, then a closer estimate to the converged result - which is exactly what the neural network is trained to do - should be leveraged. As such, the warm-up process can be replaced with a recursive step, further reducing compute time and data required for inference.

The recursive step works as follows: first, use an initialization as input to the neural network. In this case, the initialization was zero. Use the prediction from this initialization again as input to the neural network. This new prediction is the recursive result. While this process can be repeated numerous times, there was no significant benefit of using recursion more than once. While testing the performance of this method, it was found that the prediction obtained from recursion is closer to the converged solution than warm-up. When evaluating a new geometry (cylinder), the recursive output produced a **1.8x** speed up compared to without recursion, which only provided a **1.3x** speed up, as seen in Figure 5.5.

The results from Figure 5.6 might justify why recursion provides a significant advantage. It is clear that the velocity magnitude prediction is more accurate around the boundary

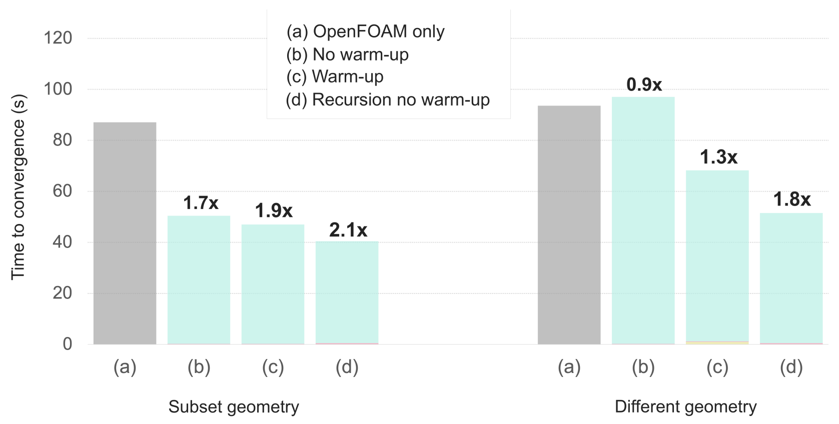
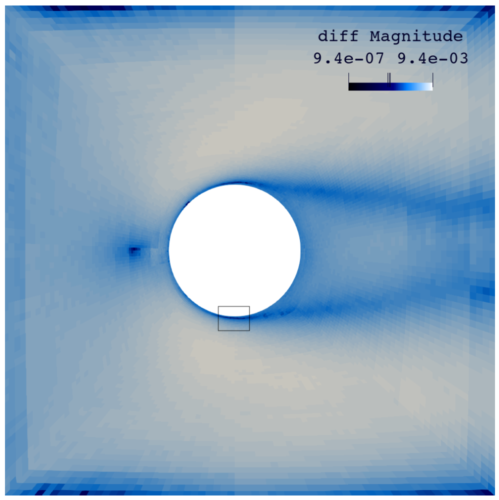
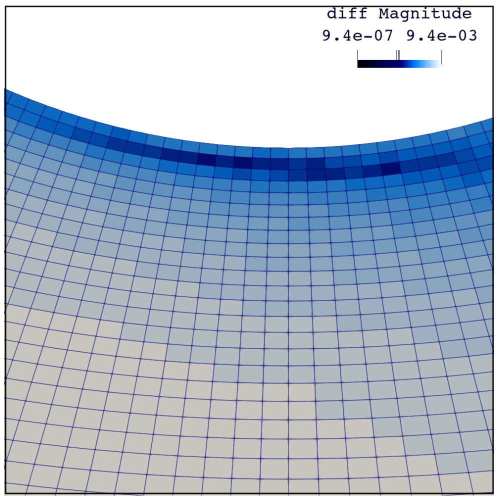


Figure 5.5: *Recursion to replace warmup. The figure represents the time to convergence for (a) the physics solver alone, (b) the CFDNet framework without warmup, (c) the CFDNet framework with wamrup, and (d) recursion instead of warmup.*

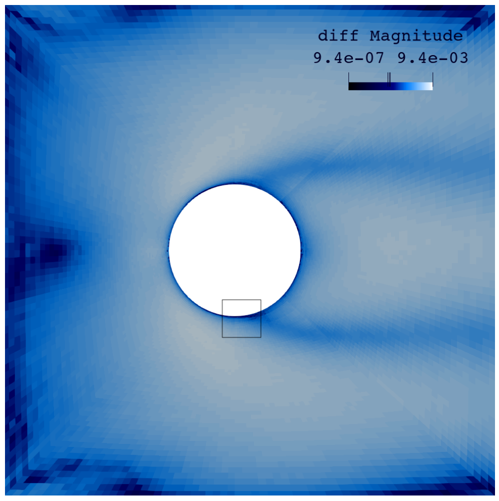
layer for the recursive output compared to the output without recursion. Due to the no-slip condition and the velocity gradients that arise from it, resolving the flow dynamics around the boundary layer is computationally challenging. As such, achieving accurate results in these regions is crucial for reaching convergence.



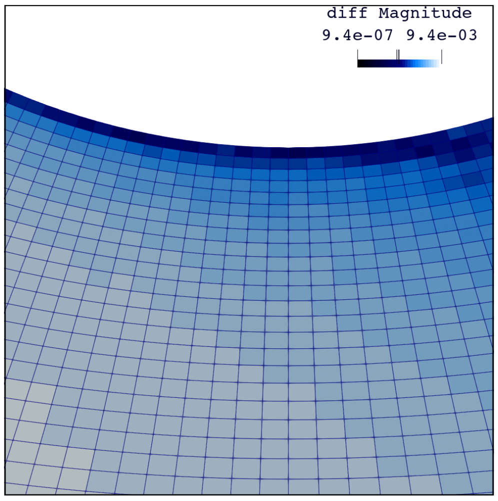
(a)



(b)



(c)



(d)

Figure 5.6: *Element-wise difference of $U_{magnitude}$ for recursion. The top row compares the prediction, without recursion, against the converged result. The bottom row compares the prediction and converged result after one recursive iteration. The boundary layer regions are closer to the converged result, as depicted by the low diff values, for the recursive case.*

5.3 Hyperparameter Optimization

Simulation-based design plays an important role in engineering, where simulations are a key step in verification and evaluation. Simulation-based design aims to eliminate unsuitable iterations as early as possible in the prototyping phase, before resources are allocated to build and test potentially poorly performing models [34].

Similarly in machine learning, it is important to optimize model performance and eliminate poor model structures early on. This is accomplished by modifying configuration values referred to as *hyperparameters* until the model produces satisfactory results. In this work, the hyperparameters of interest are the layer sizes and learning rate. The goal of this section is to couple the neural network hyperparameter optimization process with the simulation-based design process to achieve an optimal model early in the design phase.

This study aims to generate an efficient training process that properly reflects the engineering design cycle through incremental training [52]. Incremental training continuously updates a trained neural network as new data is discovered over time. This ensures that the neural network still predicts well for new problems that differ from the problems originally trained on. As such, rather than curating a dataset *before* training, this work simultaneously runs simulations and neural network training. The simulation process in this work refers to 2-dimensional INS problems with an immersed geometry, with each design iteration differing by changing the size of the immersed geometry. For instance, design iteration 1 would be an immersed geometry of ellipse aspect ratio of 0.1, design iteration 2 would have an ellipse aspect ratio of 0.25, and so on. The hyperparameters of the model are incrementally optimized to better suit the dataset over time. This means that at each design iteration, more layers are added to the neural network to better capture feature information. By doing this, the model incrementally improves and cuts down the time required to achieve minimal loss.

The general process is as follows:

1. A dataset is generated for one design iteration with `OpenFOAM`. Given initial hyperparameters, a neural network is trained to start with.

2. The next design iteration uses this optimal neural network through a CFDNet process: `OpenFOAM` for warm-up $\rightarrow$ NN $\rightarrow$ `OpenFOAM` for refinement. The next dataset is generated from this process.

3. The simulation results from this design iteration are added to the existing dataset.

4. The neural network parameters are initialized with the weights from the previously obtained optimal neural network, and new layers are added on top of this while adjusting the learning rate. The neural network is trained with these new hyperparameters and the extended dataset.

5. Steps 2-4 are repeated for the remaining design iterations.

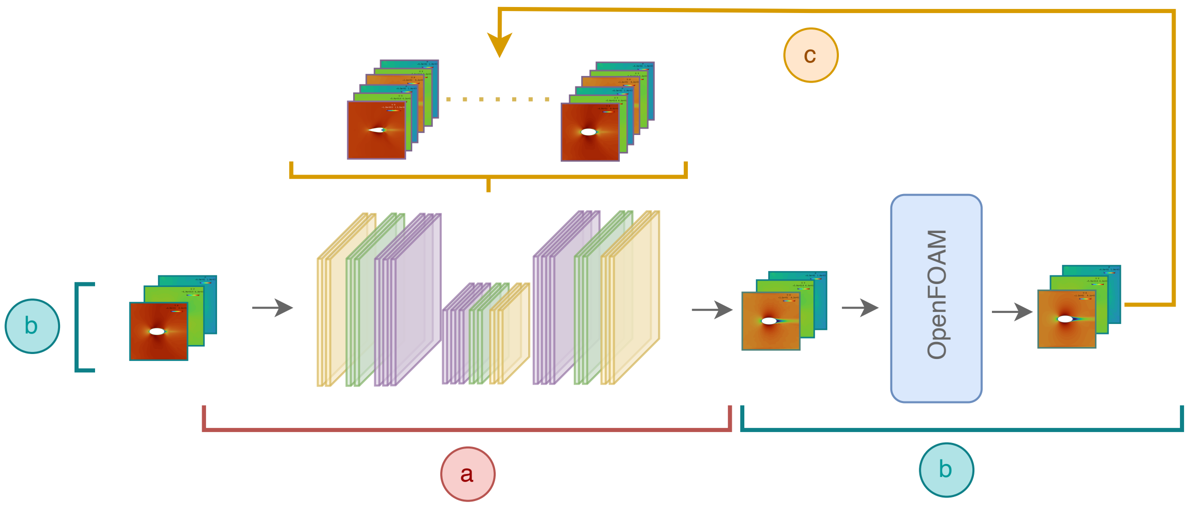


Figure 5.7: *Hyperparameter Optimization setup. Step (a) is training the model with the current dataset. In step (b), new design iterations use the state of the trained model, and the output is fed into OpenFOAM for refinement. In step (c), the input and* `OpenFOAM` *refinement output is used to generate a dataset, which is added to the existing dataset. Additionally, the hyperparameters are optimized and step (a) is run again.*

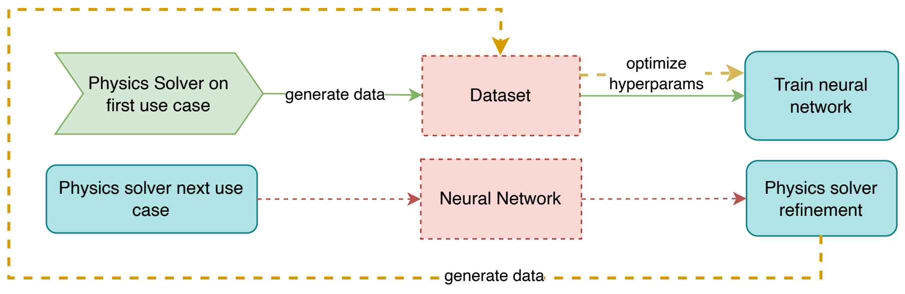


Figure 5.8: *Hyperparameter Optimization flow chart description of each step. The dotted lines represent the processes which are continuously updating.*

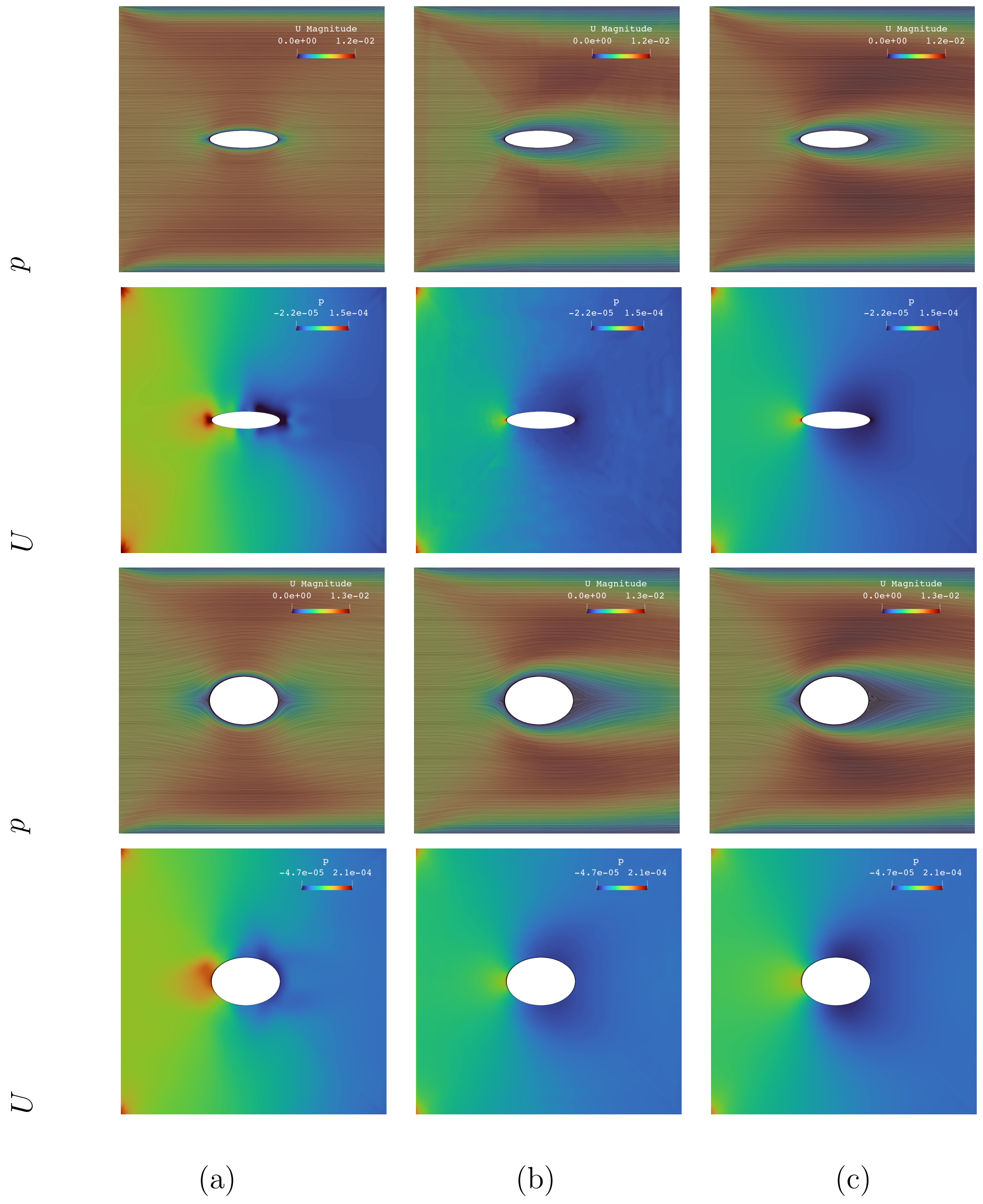


Figure 5.9: *Incremental training results. The first two rows show the NN output of an early design iteration, and the last two rows show the last design training result. (a) is the input to the neural network, (b) is the neural network output, and (c) is the output after refinement.*

The incremental training with hyperparameter optimization process proved successful when evaluating on nine design cases of immersed ellipses with aspect ratios ranging from 0.1 to 0.8, as depicted in Figure 5.9. In the initial training cases, the speed-up relative to solely running `OpenFOAM` was around **1.3x**. By the time the model has reached the last design iteration, the speed-up reached **2.6x**, as depicted in Figure 5.10. The speed-ups steadily increased as the model was provided more data from the different design cases.

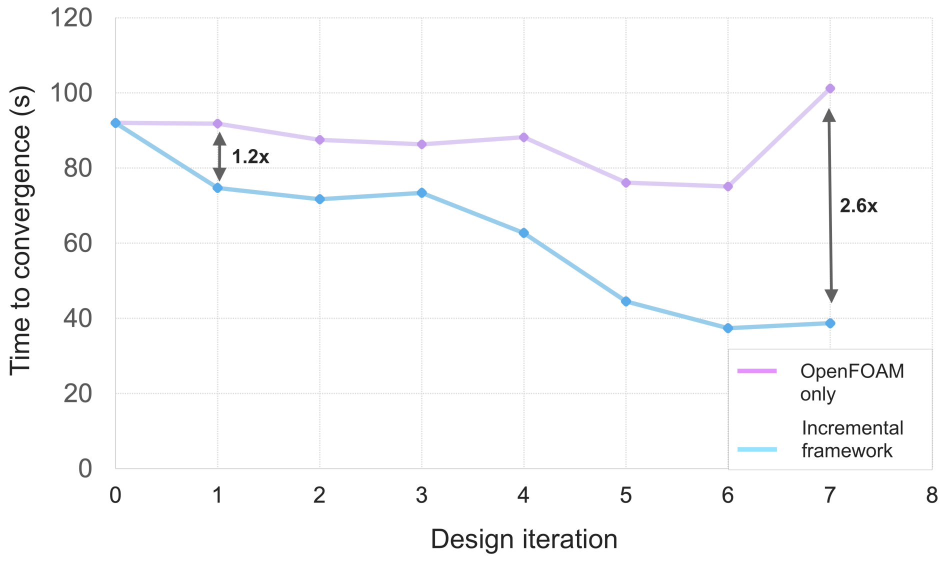


Figure 5.10: *Incremental training results. The x-axis represents increasing design iterations, and the y-axis represents the compute time of obtaining the converged results.*

5.4 Implementation of an end-to-end framework

In addition to exploring methods to enhance the efficiency of CFDNets process, a primary goal of this work was to establish a practical framework that can be readily used by researchers and engineers.

This work provides a user-centered framework which merges ML models and their training with physics solvers. Currently, this framework is usable with the opensource FVM solver, `OpenFOAM` [59], and also has some functionality implemented for use with opensource Finite Element Method solver, `OpenCMP` [35].

The use of this framework is housed in the user configuration files. Two main user configuration files are specified, `config_train` and `config`, the first to specify training configurations, and the latter to run the framework once the user is satisfied with the trained model. Example files are provided in Appendix 2.

The framework contains a few key features:

- **Data processing:** The framework automatically generates tensors in an appropriate structure required for the neural network training. The user specifies the desired data size and whether interpolation is to be used to generate this data.
- **Training:** The user can train regularly or with hyperparameter optimization. With the user configuration files, a range to sample hyperparameter values from can be specified. For instance, if optimizing by increasing the layer sizes after every iteration, the user can specify the desired layer size to sample from, i.e., `layer_1 = [128,256]`.
- **Recursion:** Once a trained model is obtained, the user can use this model recursively by specifying the number of desired recursions.
- **Warmup:** The user can use the physics solver before input to the neural network model by specifying a high residual to run the simulation up until.
- **Refinement:** Functionality is provided to go back into `OpenFOAM` for refinement up to a low residual as specified by the user.

Please refer to Appendix 1 for details on the user configuration files and on how to run the framework. These configuration files are subject to change if additional functionality is implemented over time.

Figure 5.11 shows the results of running the framework with warm-up for an unseen geometry.

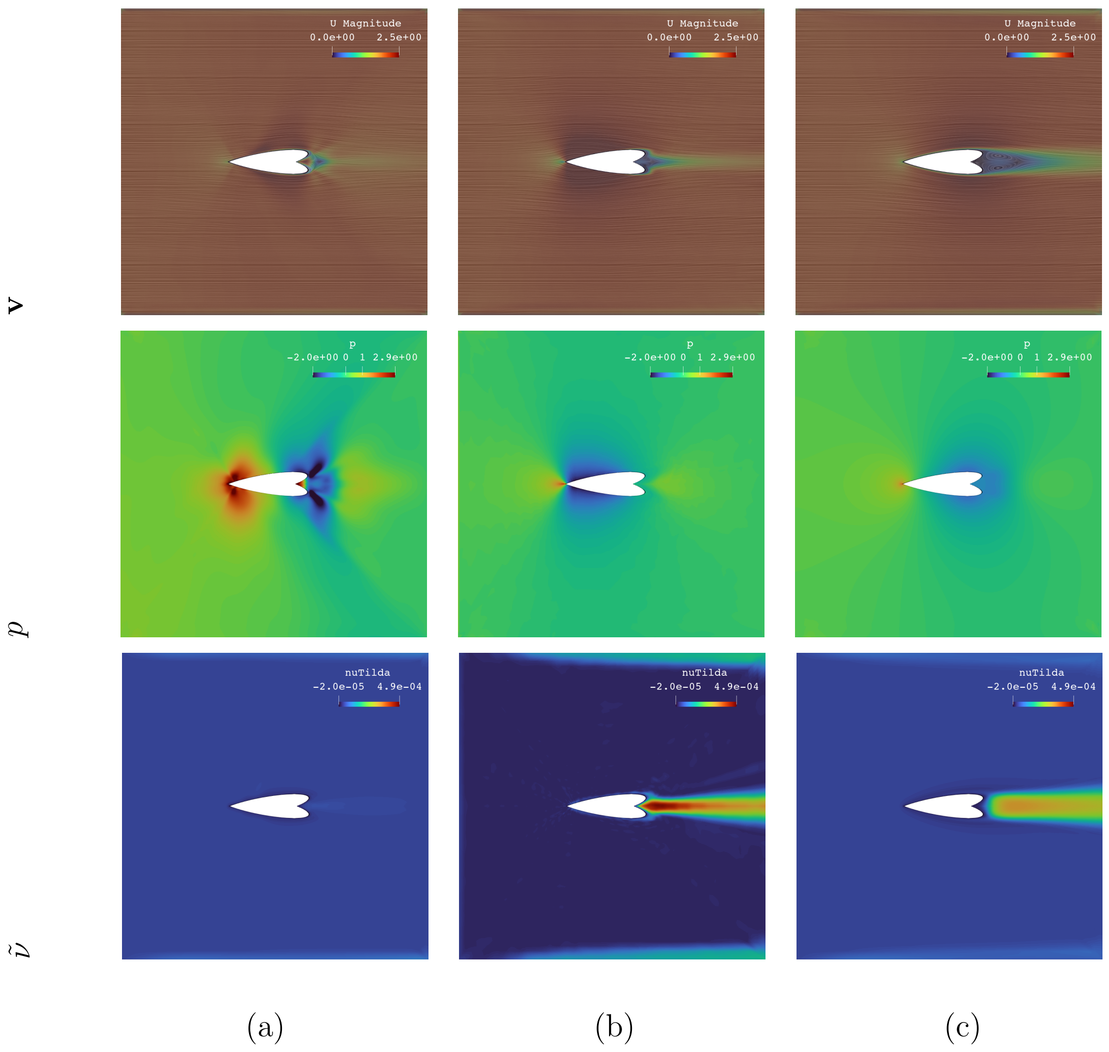


Figure 5.11: *Results for an unseen geometry in the coupled physics solver and NN framework. (a) is the input to the NN, (b) is the NN output, and (c) is the output after refinement. For a completely unseen geometry, a speedup of* ***1.3x*** *was obtained.*

Chapter 6

Conclusions & Future Work

This work focused on the development of a coupled machine learning (ML) and physics solver framework to reduce the time to convergence of computational fluid dynamics (CFD) simulations governed by the incompressible Navier Stokes equations. Despite considerable progress in the field of ML-accelerated CFD, there remains work to be done for its practical application in engineering design. As such, the main objective of this work was the development of ML-accelerated CFD methods that are compatible with an engineering design process.

The first section of this work was the validation of similar research in the field - CFDNet [38]. It was found that the simulation speed-ups followed the same relative pattern as the results presented in CFDNet. After obtaining a suitable model from the validation of CFD-Net, more extensions were implemented. Firstly, the neural network was used recursively to further reduce time to convergence. Additionally, an incremental training method that is compatible with the engineering design process was developed. The adaptation of CFDNet along with these new methods were implemented in a user-friendly framework which joins the machine learning implementation with the physics solver through a common interface.

The results of this thesis can be summarized into the main points below:

- **Model validation of similar research in the field.** This was accomplished by the development and evaluation of a framework inspired by CFDNet. The results obtained followed a pattern relative to those presented in CFDNet. Namely, warmup further reduced the time to convergence, and the coupled framework almost always provided a speedup relative to running the physics solver alone. It was expected to have some level of discrepancy between the results of this work and the results of

CFDNet. Considering the code and data from CFDNet were not provided, exact replication was not possible.

- **Recursion can be used to replace warm-up.** Considering that the neural network was trained to produce a solution close to the converged solution, warm-up should not be necessary. The neural network was used recursively instead of warm-up, which was shown to further reduce time to convergence for multiple test cases.

- **Incremental training while tuning hyperparameters can provide an optimal machine learning model earlier in the design phase.** Simultaneously training the neural network while running `OpenFOAM` simulations provided a performance benefit that increased incrementally through every case study. This is advantageous in a typical parametric study since the neural network can be used to accelerate simulations while training. This reduces the overall data processing time while decreasing the time to convergence for these case studies.

- **Extension to arbitrary domains.** This was accomplished by projecting an unstructured mesh onto a uniform grid, which created a consistent input domain for the neural network. Rather than depending on a structured grid which limits the possible design space, this projection allows for the implementation of arbitrary domains that can still benefit from the acceleration framework.

This work can be extended in a few directions. One would be to apply this framework to physics solvers beyond `OpenFOAM`, particularly to those based on the finite element method (FEM) such as `NGSolve` or `OpenCMP`. It is important to determine if a performance benefit still exists for these solvers, and, if not, how the process needs to be changed to create one. A reoccurring problem with this work was the interpolation error that arises when projecting between uniform and non-uniform grids. Considering the approximation order within elements are higher in the FEM, interpolation error will be heightened. As such, new methods that can alleviate this issue while still being generalizable (to a certain extent) should be developed.

Another important inclusion is the extension to three-dimensional domains. Two-dimensional domains are most often found in literature on ML-accelerated CFD due and their sufficient results for symmetric domains. However, more often than not, flows in complex geometries need to be modelled, requiring transport equations to be solved in three dimensions. The framework developed in this thesis provides functionality for data processing in three-dimensions, however, further analysis should be conducted to determine acceleration details in three-dimensional domains.

References

[1] In Pauline M. Doran, editor, *Bioprocess Engineering Principles (Second Edition)*, pages 899–919. Academic Press, London, second edition edition, 2013.

[2] Ekhi Ajuria-Illaramendi, Antonio Alguacil, Michaël Bauerheim, Antony Misdariis, Bénédicte Cuenot, and Emmanuel Benazera. Towards a hybrid computational strategy based on Deep Learning for incompressible flows. In *AIAA AVIATION FORUM*, pages 1–17, Virtual Event, United States, June 2020.

[3] A. Barth, Emmanuel Caillaud, and Bertrand Rose. How to validate research in engineering design ? *International Conference on Engineering Design ICED11, Denmark*, 2:1–11, 01 2011.

[4] M. M. Bronstein, J. Bruna, Y. Lecun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017. Cited By :1680.

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

[6] Jie Bu and Anuj Karpatne. *Quadratic Residual Networks: A New Class of Neural Networks for Solving Forward and Inverse Problems in Physics Involving PDEs*, pages 675–683.

[7] Dhawal Buaria and Katepalli R. Sreenivasan. Forecasting small-scale dynamics of fluid turbulence using deep neural networks. *Proceedings of the National Academy of Sciences*, 120(30), jul 2023.

[8] LS Caretto, AD Gosman, SV Patankar, and DB Spalding. Two calculation procedures for steady, three-dimensional flows with recirculation. In *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics: Vol. II Problems of Fluid Mechanics*, pages 60–68. Springer, 1973.

[9] Filipe de Avila Belbute-Peres, Thomas D. Economon, and J. Zico Kolter. Combining differentiable pde solvers and graph neural networks for fluid flow prediction, 2020.

[10] Jun Deng, Xiaojing Xuan, Weifeng Wang, Zhao Li, Hanwen Yao, and Zhiqiang Wang. A review of research on object detection based on deep learning. *Journal of Physics: Conference Series*, 1684(1):012028, nov 2020.

[11] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image super-resolution using deep convolutional networks, 2015.

[12] Chloe Dorge and Eric Louis Bibeau. Deep learning based prediction of unsteady reynolds averaged navier stokes solutions for vertical axis turbines. *Energies*, 16(3), 2023.

[13] Farida Farsian, Federico Marulli, Lauro Moscardini, and Carlo Giocoli. New applications of graph neural networks in cosmology, 2022.

[14] Han Gao, Luning Sun, and Jian-Xun Wang. Super-resolution and denoising of fluid flow using physics-informed convolutional neural networks without high-resolution labels. *Physics of Fluids*, 33(7):073603, jul 2021.

[15] Geuzaine, Christophe and Remacle, Jean-Francois. Gmsh.

[16] Nickolay Y. Gnedin, Vadim A. Semenov, and Andrey V. Kravtsov. Enforcing the courant–friedrichs–lewy condition in explicitly conservative local time stepping schemes. *Journal of Computational Physics*, 359:93–105, 2018.

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

[18] Christopher Greenshields. *OpenFOAM v11 User Guide*. The OpenFOAM Foundation, London, UK, 2023.

[19] L. Guastoni, J. Rabault, P. Schlatter, H. Azizpour, and R. Vinuesa. Deep reinforcement learning for turbulent drag reduction in channel flows, 2023.

[20] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

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

[23] Joongoo Jeon, Juhyeong Lee, Hamidreza Eivazi, Ricardo Vinuesa, and Sung Joong Kim. Physics-informed transfer learning strategy to accelerate unsteady fluid flow simulations, 2022.

[24] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, Jun 2021.

[25] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[26] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. Machine learning accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21):e2101784118, 2021.

[27] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, and Andrew Stuart. Neural operator: Learning Maps Between Function Spaces With Applications to PDEs.

[28] H.-O. Kreiss. Numerical methods for hyperbolic partial differential equations. In SEYMOUR V. PARTER, editor, *Numerical Methods for Partial Differential Equations*, pages 213–254. Academic Press, 1979.

[29] Obula Reddy Kummitha, R Vijay Kumar, and V Murali Krishna. CFD analysis for airflow distribution of a conventional building plan for different wind directions. *Journal of Computational Design and Engineering*, 8(2):559–569, 01 2021.

[30] S. Kutluay, A.R. Bahadir, and A. Özdeş. Numerical solution of one-dimensional burgers equation: explicit and exact-explicit finite difference methods. *Journal of Computational and Applied Mathematics*, 103(2):251–261, 1999.

[31] Hyuk Lee, Young Kim, Yung Byun, and Soo Hyung Park. Mach 3 boundary layer measurement over a flat plate using the piv and ir thermography techniques. 01 2017.

[32] Xiao Li, Li Sun, Mengjie Ling, and Yan Peng. A survey of graph neural network based recommendation in social networks. *Neurocomputing*, 549:126441, 2023.

[33] Kai Liaw. Simulation of flow around bluff bodies and bridge deck sections using cfd. 01 2005.

[34] Wafa Mefteh. Simulation-based design: Overview about related works. *Mathematics and Computers in Simulation*, 152:81–97, 2018.

[35] Elizabeth Monte. Opencmp: An open-source computational multiphysics package. Master's thesis, 2021.

[36] Kamaljyoti Nath, Xuhui Meng, Daniel J. Smith, and George Em Karniadakis. Physics-informed neural networks for predicting gas flow dynamics and unknown parameters in diesel engines. *Scientific Reports*, 13(1), aug 2023.

[37] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas Malaya, and Aparna Chandramowlishwaran. Surfnet: Super-resolution of turbulent flows with transfer learning using small datasets, 2021.

[38] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas Malaya, and Aparna Chandramowliswharan. Cfdnet: A deep learning-based accelerator for fluid simulations. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[39] Thomas P. O'Driscoll and Andrew R. Barron. Cfd analysis of the location of a rear wing on an aston martin db7 in order to optimize aerodynamics for motorsports. *Vehicles*, 4(2):608–620, 2022.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[41] Jiang-Zhou Peng, Wang Yizhe, Siheng Chen, Zhihua Chen, Wei-Tao Wu, and Nadine Aubry. Grid adaptive reduced-order model of fluid flow based on graph convolutional neural network. *Physics of Fluids*, 34:087121, 08 2022.

[42] Edwin N. Lightfoot R. Byron Bird, Warren E. Stewart. *Transport Phenomena.* John Wiley and Sons, 2006.

[43] M. Raissi, P. Perdikaris, and G.E. 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.

[44] Andrés Reyes-Urrutia, Cesar Venier, Néstor Javier Mariani, Norberto Nigro, Rosa Rodriguez, and Germán Mazza. A cfd comparative study of bubbling fluidized bed behavior with thermal effects using the open-source platforms mfix and openfoam. *Fluids*, 7(1), 2022.

[45] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. cite arxiv:1505.04597Comment: conditionally accepted at MICCAI 2015.

[46] François G Schmitt. About Boussinesq's turbulent viscosity hypothesis: historical remarks and a direct evaluation of its validity. *Comptes Rendus Mécanique*, 335(9-10):617–627, October 2007.

[47] Joachim Schoeberl. C++11 implementation of finite elements in ngsolve. 09 2014.

[48] Pushan Sharma, Wai Tong Chung, Bassem Akoush, and Matthias Ihme. A review of physics-informed machine learning in fluid mechanics. *Energies*, 16(5), 2023.

[49] I Sonata, Y Heryadi, L Lukas, and A Wibowo. Autonomous car using cnn deep learning algorithm. *Journal of Physics: Conference Series*, 1869(1):012071, apr 2021.

[50] Nils Thuerey, Konstantin Weißenow, Lukas Prantl, and Xiangyu Hu. Deep learning methods for reynolds-averaged navier–stokes simulations of airfoil flows. *AIAA Journal*, 58(1):25–36, jan 2020.

[51] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks, 2022.

[52] Gido M. van de Ven, Tinne Tuytelaars, and Andreas S. Tolias. Three types of incremental learning. *Nature Machine Intelligence*, 4(12):1185–1197, Dec 2022.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[54] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics - the finite volume method.* Addison-Wesley-Longman, 1995.

[55] Ricardo Vinuesa, Steven Brunton, and Beverley McKeon. The transformative potential of machine learning for experiments in fluid mechanics, 03 2023.

[56] Ricardo Vinuesa and Steven L. Brunton. Enhancing computational fluid dynamics with machine learning. *Nature Computational Science*, 2(6):358–366, Jun 2022.

[57] Jing Wang, Cheng He, Runze Li, Haixin Chen, Chen Zhai, and Miao Zhang. Flow field prediction of supercritical airfoils via variational autoencoder based deep learning framework. *Physics of Fluids*, 33(8), 08 2021. 086108.

[58] Rui Wang, Karthik Kashinath, Mustafa Mustafa, Adrian Albert, and Rose Yu. Towards physics-informed deep learning for turbulent flow prediction, 2020.

[59] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Computer in Physics*, 12(6):620–631, 11 1998.

[60] Gabriel D. Weymouth. Data-driven Multi-Grid solver for accelerated pressure projection. *Computers & Fluids*, 246:105620, 2022.

[61] Fang Yang, Kunjie Fan, Dandan Song, and Huakang Lin. Graph-based prediction of protein-protein interactions with attributed signed graph embedding. *BMC Bioinformatics*, 21(1):323, Jul 2020.

[62] Ali Girayhan Özbay and Sylvain Laizet. Deep learning fluid flow reconstruction around arbitrary two-dimensional objects from sparse sensors using conformal mappings. *AIP Advances*, 12(4), 04 2022. 045126.

APPENDICES

Appendix 1

```
solvers
{
    p
    {
        solver          GAMG;
        tolerance       1e-08;
        relTol          0.1;
        smoother        GaussSeidel;
    }

    U
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        nSweeps         2;
        tolerance       1e-08;
        relTol          0.1;
    }

    nuTilda
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        nSweeps         2;
        tolerance       1e-08;
        relTol          0.1;
    }
}

SIMPLE
```

```
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        p               1e-4;
        U               5e-6;
        nuTilda         1e-4;
    }
}

relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
        nuTilda         0.7;
    }
}
```

Listing 1: Residual details implemented in OpenFOAM

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         leastSquares;
}

divSchemes
{
    default         none;
    div(phi,U)      Gauss linearUpwind grad(U);
    div(phi,nuTilda) Gauss linearUpwind grad(nuTilda);
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default          Gauss linear corrected;
}

interpolationSchemes
{
    default          linear;
}

snGradSchemes
{
    default          corrected;
}

wallDist {
    method      meshWave;
}
```

Listing 2: Numerical scheme details implemented in OpenFOAM

Appendix 2

```
[SIMULATION DETAILS]
software = OpenFOAM
fields = u, p
tolerance = 0.0001
train_dir = OpenFOAM/sajeda-9/train
test_dir = OpenFOAM/sajeda-9/test
generate_training_data = True
data_dir_str = train_data
use_probes = False
turbulence = False

[CNN DETAILS]
x_uniform_dim = 64
y_uniform_dim = 256
num_dimensions = 2

[TRAIN DETAILS]
train_model = False
optimize_hyperparams = False
model_save = model_laminar
incremental_train = False
regular_train = False
```

```

[HYPERPARAMETERS]
learning_rate = 6e-4, 6e-6
num_epochs = 40
batch_size = 2, 6, 10
batch_layers = True
display_epochs = 1
loss_function = MSELoss
layer_1 = 3, 32
layer_2 = 32, 64, 128
layer_3 = 128, 256
layer_4 = 128, 256
kernel_conv1 = 2, 2, 2, 8
dropout = 0.0, 0.1
optimizer = RMSprop
momentum = 0.8, 0.9
skip_connections = 0
```

Listing 3: Example user configuration file for data generation and training. User can specify the type of training and type of data generation

```

[SIMULATION DETAILS]
software = OpenFOAM
working_dir = OpenFOAM/sajeda-9/run2/
task = inference, refinement
fields = u, p

[ERROR ANALYSIS]
tolerance_pre = 0.01
tolerance_post = 0.0001
save_to_file = True

[CNN MODEL]
x_uniform_dim = 256
y_uniform_dim = 64
recursive = 1
model_name = model.pt
```

Listing 4: Example user configuration file for a full framework run