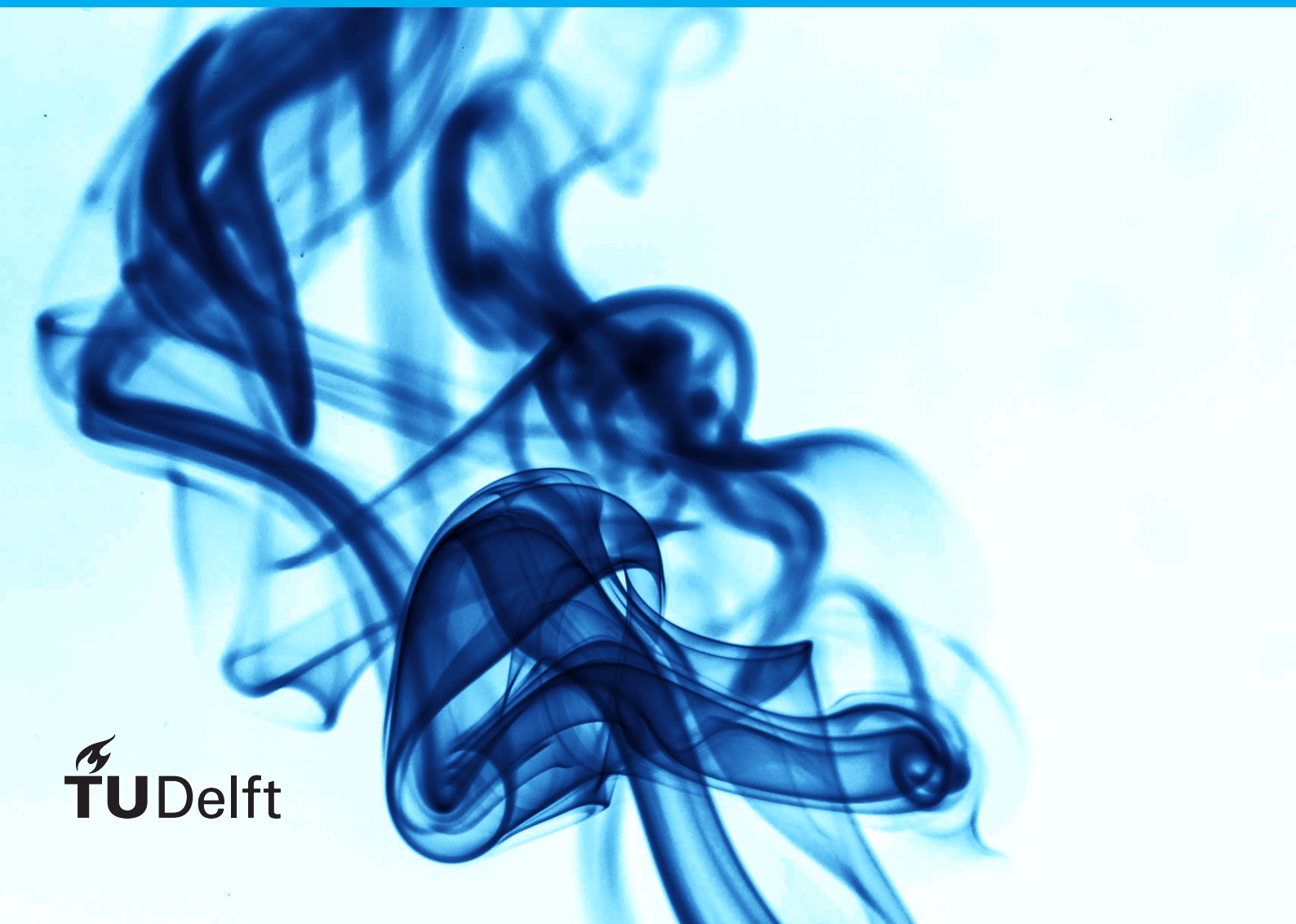
Non-local Frame-independent Neural Network Turbulence Models for RANS Simulation

Ruiying Xu



TU Delft

Non-local Frame-independent Neural Network Turbulence Models for RANS Simulation

by

Ruiying Xu

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Thursday November 25, 2021 at 3:00 PM.

Student number:	5059011	
Project duration:	March 1, 2021 – November 25, 2021	
Thesis committee:	Dr. R. P. Dwight,	TU Delft, supervisor
	Dr. H. Xiao,	Virginia Tech, supervisor
	Dr. N. A. K. Doan,	TU Delft
	Dr. E. van Kampen,	TU Delft

TU Delft

Contents

List of Symbols

Abbreviations

RANS	Reynolds-Averaged Navier-Stokes
DNS	Direct Numerical Simulation
CNN	Convolutional Neural Network
PDE	Partial Differential Equation
CFD	Computational Fluid Mechanics
PIV	Particle Image Velocimetry
ODE	Ordinary Differential Equation
MLP	Multilayer Perceptron
POD	Proper Orthogonal Decomposition
MD	Molecule Dynamics
RSM	Reynolds Stress Models
LES	Large Eddy Simulation
SGS	Sub-grid Scale
ANN	Artificial Neural Network
GA	Genetic Algorithms
S-A	Spalart-Allmaras
ILES	Implicit Large Eddy Simulation
XAI	Explainable Artificial Intelligence
LRP	Layerwise Relevance Propagation
SHAP	Shapley Additive Explanations
KDE	Kernel Density Estimation
SIMPLE	Semi-Implicit Method for Pressure Linked Equations
GAMG	Generalised Geometric/Algebraic Multi-Grid
ReLU	Recified Linear Unit

Greek symbols

ρ	Fluid density
ν	Kinematic viscosity, diffusion coefficient in computing influence lengths
δ_{ij}	Kronecker delta
ν_T	Eddy viscosity
ϕ	Quantity described by conservation / activation function of neural networks
θ	Parameters of neural networks, including weights and biases; cell volume
γ	Learning rate
μ	Mean
σ	Standard deviation
λ	Weight decay factor
α	Slope parameter
ϵ	Error tolerance
ζ	Dissipation coefficient
η	Wall distance function
$\hat{\eta}$	Raw wall distance

Latin symbols

Re	Reynolds number
$\mathbf{q}$	Input feature vector
$\mathbf{x}$	Coordinates vector
$\mathbf{u}$	Velocity vector
$\mathbf{c}$	Scalar feature vector
$\mathscr{Q}$	Input feature matrix
$\mathscr{G}$	Encoding function matrix
$\mathscr{D}$	Invariant input matrix
k	Turbulence kinetic energy
ε	Turbulence kinetic energy dissipation rate
$\boldsymbol{U}$	Velocity vector
t	Time
p	Pressure
u_i, u_j	Components of fluctuating velocity
ρ	Fluid density
$\mathscr{P}$	Production term of turbulence kinetic energy
V	Constant velocity of the reference frame in Galilean transformation
w	Weights of neural networks, width of the periodic hill
b	Biases of neural networks, boundary cell indicator
R	Loss functions
C_L	Lift Coefficient
L	Lift force
q	Dynamic Pressure
S	Surface area
J	Loss functions with regularization terms
N_x	Number of points in x direction
N_y	Number of points in y direction
u	Velocity component in x direction
v	Velocity component in y direction
l_1	Major axis of the influence ellipse
l_2	Minor axis of the influence ellipse
s	Magnitude of strain rate tensor
u	Velocity magnitude
l_δ	Reference boundary layer thickness
r	Proximity to cloud center
r'	Proximity to cloud center in local velocity frame
I	Influence region size

1 Introduction

Turbulence is a physical phenomenon that is ubiquitous in engineering. No matter it is an internal flow such as fluid inside pipes and channels or exterior flows like the air surrounding airplanes and vehicles, turbulence is prevalent. Turbulent flows are characterized by their chaotic nature and the wide range of length and time scales they occupy, which pose a great challenge in understanding and predicting them. Modelling turbulent flows has thus long been an important research topic over the past half-century.

The exact description of fluid flows is given by the Navier-Stokes equations when the continuum assumption applies. However, the computational cost of directly solving for the N-S equation (DNS) in high Re number flows is forbiddingly high, as it grows cubically (Re^3) with regards to the Reynolds number [1]. Simulating turbulent flows with tractable costs requires extra modelling efforts. The most commonly used flow simulation technique is RANS (Reynolds-averaged Navier-Stokes). The idea is to decompose the turbulent flow into the mean flow field described by the RANS equation (primary equation) and the fluctuating velocity field whose influence on the mean flow field is modelled by turbulence closures.

The construction of closure models usually involves a combination of physical understanding and a calibration procedure accommodating simple canonical flow cases. However, the direction towards a better model is not clear due to a large amount of empiricism involved, and the calibration procedure is limited by the capability of incorporating various flow cases. There is no foreseeable breakthrough following the traditional strategies. Researchers have been actively searching for a new line of thought for the turbulence modelling problem. Recent developments in machine learning techniques have attracted people's attention. Interdisciplinary study on machine learning and turbulence modelling provides promising results and opens up novel solutions to this long-standing closure problem [2]. By utilizing the high expressive power of machine learning algorithms such as neural networks, it is possible to make use of a much larger data set and obtain more universal turbulence models.

In the practice of developing machine learning-based turbulence models, it is found that the model performance can be greatly improved by embedding the known physics [3–6], especially when the data available is limited. Most of such works aim at embedding the symmetry of the turbulence flow. This refers to the invariance properties of the flow under transformations of the coordinate system. The transformations refer to translation, rotation and uniform motion(Galilean invariance) of the reference frame. The idea here is the same as constructing a physics-based model. Only when the physic constraints are satisfied, can the model be qualitatively correct.

Another property of turbulence flows that is seldom incorporated in data-driven turbulence models (either physics -based models) is its nonlocality. It is the major paradox that is encountered in turbulence modelling problems[7]. In local data-driven models, the mapping is built between local flow derivatives (velocity gradient and its linear or nonlinear combinations) at that point of interest [5, 8, 9]. Whilst such local models are only applicable under the assumption that there is a local balance between the production, redistribution and the dissipation of the Reynolds stress [10]. The real physics, in contrast, shows that the turbulence quantities at one point are not only determined by the local flow field but they are also greatly influenced by the upstream flow structure or boundary conditions [11].

In this thesis, a novel non-local neural network turbulence model is developed to reproduce the traditional $k-\varepsilon$ turbulence model. The non-local architecture follows the framework proposed in [12]. In the proposed framework, a mapping between mean flow feature vectors $\mathbf{q}$ in a stencil surrounding the point of interest and the turbulence

quantities at that point is built using neural network training. The highlight of the proposed architecture is the integration of all the invariance properties, especially rotation invariance and permutation invariance. For a complete validation of the proposed framework, the well-trained neural network is coupled back with the RANS equation solver forming a closed model. Its performance is proved in the coupled setting for both interpolation and extrapolation flow cases. A simple diagram of the framework is presented in Figure 1.1. The network is

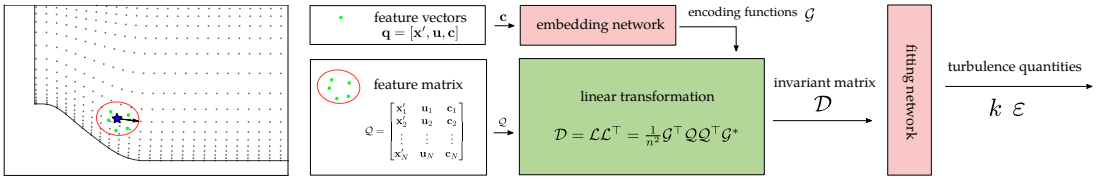


Figure 1.1: Simple diagram of the framework. A stencil near the downhill is visualized as an example. The mesh is sampled every 4 and 10 points for a clearer view.

composed of two parts, the embedding network and the fitting network. In the embedding network, higher-level representations of the set of input mean flow features are extracted through a nonlinear transformation of multilayer perceptron (MLP). The fitting network as indicated by its name fits the neural network to the proper mapping between the inputs (both outputs of embedding network and input feature matrix) and the outputs (turbulence quantities k and ε). The workflow starts with a set of input feature vectors of the stencil enclosing the point of interest. Before entering the network, we distinguish the frame-independent features such as velocity magnitudes from the frame-dependent vector features such as the velocity components. The frame-independent features (referred to as $\mathbf{c}$) of each stencil point are fed individually into the embedding network, in which frame-independent encoding functions for each stencil point are generated. Then the complete set of input features (all feature vector $\mathbf{q}$ in the stencil) are projected to the encoding functions (can also be viewed as averaged over the stencil with the encoding functions as the weight). This step guarantees the permutation invariance of the input point set. To achieve rotation invariance of the frame-dependent features, a pairwise projection $\mathbf{q}_i^\top \mathbf{q}_{i'}$ is also performed before entering the fitting network. The steps for achieving permutation and rotation invariance take the form of a linear transformation. In this way, the input matrix of the fitting network has all the desired invariance properties. Matrix $\mathscr{D}$ is then input to the fitting network, in which the final output turbulence kinetic energy k and turbulence kinetic energy dissipation rate ε are obtained. Details of the methodology and proof of invariance properties are provided in Chapter 4.

Similar to PDE based closure models, the performance of the neural network turbulence model is only verified when it is placed under the coupled setup, in which the turbulence quantities predicted by neural network model are fed back to the RANS equation solver. In this way, the interaction between the primary equation and the neural network turbulence model is inspected. It is a particularly challenging problem for neural network models. Because when coupled with the primary equation and participating in the solver iteration, the model is dealing with unconverged flow fields most of the time. In other words, it is related to highly extrapolated cases, in which the behaviour of the network is usually unpredictable. Studies have shown that the RANS equation can be extremely sensitive with regards to the Reynolds stresses [13]. This poses an extra challenge to the closure model performance. Besides, neural networks are notorious for their opaqueness and lack of interpretability [14], which brings difficulty to tuning the model according to the needs. In our work, the obtained network is tested in numerical experiments in the coupled setup. Its stability is thus proved.

The advantages of our model can be concluded as its nonlocality and strict symmetries as mentioned earlier. Our model takes in information from the neighbouring region and is more likely to capture the real flow physics. Compared to other approaches for attaining nonlocality, such as those based on CNN (Convolutional Neural Network) [15–18], the proposed model strictly ensures symmetries. One of the major drawbacks of CNN is a lack of geometric invariance [19]. General practice for improving invariance is applying augmented dataset [3, 20], in which the input data is transformed by translation and rotation. The enlarged dataset is then fed into the network such that the network acquires certain robustness towards variations of the input data coordinate system. As implied by the procedure, there is no guarantee of the invariance properties of the model. Minor violation of the symmetries may not influence the classification problem of image recognition, for which CNN is initially developed [21]. However, in a physics modelling scenario, it may lead to intrinsically wrong results or even breakdowns of the simulation. Considering the stringent requirement of physics modelling, having solid invariance properties is a huge advantage of our framework over others.

The work presented in this thesis is build up upon the work in [12], in which the network architecture is applied to a passive scalar (can be regarded as concentration) transport equation in laminar flows. Compared to the previous work, the current one is more challenging in several aspects: (1) with higher Reynolds number, the mean flow field has a more complex pattern. The thinner boundary layer indicates more rapidly varying velocities spatially. (2) the transport equations of the turbulent quantities have more complicated form than that of the concentration transport equation, e.g. the concentration transport equation is a single PDE, while the turbulence quantities are described by two coupled PDEs. Specifically, the span of the turbulence quantities are much larger than that of the concentration field, which makes them numerically hard to deal with. (3) the turbulence quantities have strong influence on the mean flow field, whilst the concentration is passively transported by the mean flow field. The neural network turbulence model is trained by and evaluated against $k-\varepsilon$ model [22]. Its accuracy and applicability are thus restricted by the low fidelity training set. The objective here is neither to replace nor to surpass the $k-\varepsilon$ model with neural networks. The work presented serves as a proof of concept regarding the applicability of the non-local framework to turbulence modelling problems. A future plan is to modify the network to accommodate tensor outputs. In this way, it can then be easily transplanted to the problem of predicting Reynolds stress tensor and potentially demonstrate more merits over traditional models.

The following thesis report consists of 5 chapters. In Chapter 2, a thorough literature review on data-driven turbulence modelling is presented. Chapter 3 provides the fundamentals of CFD and machine learning techniques related to the thesis work. Chapter 4 introduced in detail the methodology of the proposed framework, including the data generation, data pre-processing, neural network architecture, network training and the coupling between neural network and numerical solver. Chapter 5 is for major results and discussion. In Chapter 6, the thesis work is concluded briefly and suggestions for future works are given.

2

Literature review

In this chapter, a literature review on machine learning and its application in physics modelling, especially turbulence modelling is presented. Section 2.1 provides a concise description of the development of major machine learning approaches. Attention is paid particularly to random forests and neural networks since they are the most frequently used approach in physics modeling. Section 2.2 presents machine learning-based physics modelling that is relevant to the thesis work. The review is not restricted to turbulence modelling considering that some application scenarios in other physics fields share large similarities to turbulence modelling. Section 2.3 discusses the latest developments in traditional turbulence modelling as the background of the rising data-driven modelling. Finally, Section 2.4 summarizes the recent progress in data-driven turbulence modelling, under which the thesis work is carried out.

2.1. Machine learning

Machine learning is one of today's most flourishing fields of study. It has developed rapidly during the past two decades, thanks to the abundance of data and lower cost in computation [23]. Machine learning can be concluded as an optimization problem, that is, how to improve the performance of a system under certain metrics through the training process. In comparison to traditional optimization problems, machine learning programs are not hard-coded. Instead, they automatically adapt to the needs [24] by learning from the data.

Machine learning methods can be grouped roughly into supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, a dataset composed of pairs of desired outputs (labels) and inputs (features) is required. Based on this, the algorithms infer a function mapping the input to the output by analyzing the training data. Unsupervised learning in comparison requires no labels. Patterns in the data are found by the algorithms on their own. In reinforcement learning, the program interacts with an environment, from which it can receive feedback and adapt itself. There are also other machine learning approaches that don't fit well into the three category framework, such as topic modelling and meta-learning.

Among the three categories of machine learning approaches, supervised learning is probably the most widely used. Although the outcome of supervised learning is a function mapping between inputs and outputs, the form of the mapping is not necessarily a mathematical expression. There is a huge variety of them, the most common ones are decision trees, random forests, neural networks, Bayesian classifiers, etc [23]. Algorithms are developed for the optimization of each form. They are introduced in the following part of the section.

Decision trees are classifiers characterized by a hierarchical structure of questions that are about the features of the items. By answering the questions from the root to the leaf of the tree, the item can be sorted into a class [25]. When decision trees are combined together and form ensembles, they are called random forests. In random forests, each tree is grown on randomly sampled vectors. The random forest's prediction is the class chosen by decision trees the most. The first random forest algorithm is proposed by Ho [26] and later developed by Amit and Geman [27], Dietterich [28], and Breiman [29]. A theoretical analysis of the generalization error of the random forest algorithmss is provided by Breiman [29]. A detailed review of the theoretical study of random forest algorithm is provided in [30].

Neural networks are multilayer networks of neurons, which are simple processors with a certain activation function. Neurons in the input layer receive information outside of the system and get activated. Neurons in the following

layers react to the weighted sum of the previous layer and give an output. Details of the working principle and architecture are presented in Chapter 3. Neural networks have been around for a long time. The initial idea of neural networks dates back to the 1950s or even earlier. The feedforward multilayer perceptron neural network architecture, as well as convolutional neural networks, were proposed [31, 32]. An efficient backpropagation training method that enables deep neural network architecture (networks that have multiple hidden layers) was proposed later in the 1980s by Parker [33] and LeCun et al. [34].

Deep neural networks were proved to possess larger representation power compared to shallow neural networks. Its multilayer architecture allows the abstraction of a hierarchical representation of the input. However, there are still challenges remaining on deep neural network training. One of the biggest problems is the vanishing or exploding gradients which is first identified by Hochreiter. For gradient-based training, the derivatives are calculated based on the chain rules traversing from the final layer to the input layer. As the network gets deeper, derivatives from multiple layers are multiplied together. This leads to exponential growing derivatives when the derivatives are large or vanishing derivatives when they are small. The derivatives positively decide the steps taken in the training iteration. If the steps are too large, the weights will grow unlimited and the network becomes unstable. On the other hand, if the steps are too small, few changes are made to the network and the training process is slow. The gradient exploding and vanishing problem can be partly alleviated by gradient clipping, proper initialization, etc. A review of the studies regarding this fundamental problem is presented in [36].

In recent years, various deep learning models are developed for different purposes. In terms of non-local models, which are closely relevant to the thesis work, one of the important types is the pointNet developed by Qi et al. [37]. The proposed network architecture is specialized in the point cloud geometric data structure, which is simply a set of unstructured points. The core feature of the architecture is that it is invariant under permutation of the inputs. The network is proved to be capable of approximating any continuous set functions by theoretical analysis. The model performance is evaluated in numerical experiments on 3D object classification, part segmentation and semantic segmentation problems. Based on the pioneering work of PointNet, Qi et al. introduced a hierarchical neural network called PointNet++, in which the input point set is partitioned recursively such that the network is able to extract features of a range of scales. A comprehensive review of deep learning for point clouds including variants of PointNet and other methods (convolution and graph-based networks) is presented in [39].

Another type of non-local model that is usually used in image recognition problems is the convolutional neural network (CNN). It is a regularized version of fully connected neural networks. The idea is to look at only part of the input data at a time instead of the entire input. In this way, the network is able to learn the basic elements in the small scales rather than remembering the complex aggregate and thus avoiding the problem of overfitting. According to this idea, the input matrix is segregated into small patches. The patches cover the entire input and have overlaps in between. Lying at the centre of a CNN are the kernel (or filter) and the convolution operation it applies to the input. The kernel is a 2D array of weights that has a smaller size than the input data corresponding to the size of patches. It is like a small window that the network can look into or a pattern that the network can compare the input with. What CNN do is performing a scalar product between the kernel and the patches in the input data array. It is done in a systematic way, following certain orders such as left to right and top to bottom, like a sweep over the input. The scalar products of all the patches form a "filtered" input data array, called the feature map. The feature map can be dealt with regularly, such as being passed to an activation function.

2.2. Machine learning-based physics modelling

The application of machine learning can be found in almost all fields of science and engineering. In this section, some of the machine learning models developed for general physics modelling problems are summarized. These previous works can be instructive in the application of machine learning in the specific field of turbulence modelling. Besides, works done in other fields that share similarity with turbulence modelling is also introduced, from which the inspiration of the proposed framework is drawn.

In terms of physics modelling in general, Raissi et al. [40] introduced the idea of physics informed deep learning. The paper points out that for physics modelling, although the data available is usually limited, there exists prior knowledge that can be utilized. The prior knowledge can be either governing equations or empirical equations from expertise. The paper also introduced the two classes of data-driven modelling problems, namely the data-driven solution and the data-driven discovery. The former refers to using data-driven techniques to find solutions. The latter is the inverse problem of finding the governing equations. In terms of the former problem, Raissi and Karniadakis introduced a paradigm called hidden physics model [41]. Considering that the data acquisition for many of the physics modelling problems is very expansive, they proposed a framework that is capable of leveraging the law of physics and extracting the underlying pattern of high-dimensional data in a

data-efficient way. Such a model is useful when the data is limited and the governing equation is known, such as reconstructing flow field from Particle Image Velocimetry (PIV) data. Zhu et al. proposed a methodology for high-dimensional surrogate modelling with limited data. The neural network is trained by incorporating the governing equations to the loss functions instead of using the labelled data. The model is proved to generalize better than the models trained with data. Berg and Nyström [43] used a deep neural network for solving PDE with complex geometries, in which the classical mesh-based method fails. The benefits of increasing the number of hidden layers (deeper network) are also shown.

On the other hand, attempts are made on approximating governing equations by data using deep neural networks (the second class). The goal is to gain a novel understanding of physics from a large amount of data. Long et al. [44] developed a deep neural network called PDE-Net, in which the underlying PDE physics model can be discovered. All the convolution kernels are constrained in a way that the expressive power is reserved and at the same time, the governing PDE can be identified easily. The proposed network is tested on convection-diffusion equations. Based on PDE-Net, a numeric-symbolic hybrid deep network named PDE-Net 2.0 is developed [45], in which a symbolic network is used to uncover the response function. By introducing the symbolic network, no assumption on the type of the PDE needs to be made. Thus the network is able to approximate governing equations of dynamic systems whose form is unknown. Champion et al. proposed a network that is capable of discovering quantitative descriptions from scientific data. The key part of the network is a deep autoencoder network that can transform the data into a reduced space, in which the representation of the dynamic is sparse. The approach is demonstrated on the high-dimension systems and 2D systems. Qin et al. [47] proposed a residual network (ResNet) that is able to learn the governing equations of dynamic systems based on trajectory data. The properties of the proposed method are shown in numerical experiments on linear ODEs and nonlinear ODEs. Sun et al. [48] proposed a neural network structure that is similar to ResNet but with the layers replaced by MLPs. The new structure is capable of achieving higher accuracy for complex spatio-temporal dynamic systems. Rudy et al.'s work pays special attention to the signal noise of observations. Instead of denoising the data prior to learning or averaging out the error leveraging a huge amount of data, they treat the measurement error also as unknown that needs to be identified. The network's robustness is proved in various canonical cases.

Neural networks are also applied to other classes of problems. In the framework proposed by Hesthaven and Ubbiali [50], neural networks are applied to reduced order modelling problems. The reduced basis is extracted by proper orthogonal decomposition (POD). Whilst, the coefficients of the reduced-order model are approximated by deep neural networks. Karumuri et al. [51] developed a solver-free deep residual network approach for stochastic partial differential equations. The approach is demonstrated on high-dimensional uncertainty propagation problems. By using the proposed method, the forward evaluation in response surface modelling can be done efficiently and thus the curse of dimensionality is alleviated.

In understanding the network architecture used in the thesis work, special attention can be paid to the molecule dynamics (MD) problem in physics and chemistry, for which the network is originally developed. In the MD problem, the potential energy of a many-body system needs to be determined based on the local environment. There are two major difficulties: First, the atomic coordinates cannot be used as the input since they are not invariant to the transformation of the reference frame. Second, the permutation invariance needs to be built into the architecture. To address such state of affairs, Zhang et al. developed a network called Deep Potential Molecular Dynamics (DPMD) that is capable of overcoming the aforementioned limitation of invariance properties. It is the predecessor of the current work.

2.3. Turbulence modelling

In this section, a brief discussion on turbulence modelling (for RANS) is made with special attention to its difficulty and the limitation of current works. The aim is to introduce the background under which machine learning facilitated turbulence modelling comes into play.

Turbulence modelling is a problem that researchers have been endeavouring to solve for over a century. According to Spalart [7], there are two types of philosophies in turbulence modelling, namely the "systematic" philosophy and the "openly empirical" philosophy. The former one refers to the efforts of modelling the higher-order moments as accurate as possible and term by term hoping that the "principle of receding influence" is valid. Examples of the first philosophy are the Reynolds stress models (RSM) and higher moment models, in which all the independent components of the Reynolds stress tensor or higher moments are computed directly. The latter philosophy to the opposite constructs models that satisfy constraints like Galilean invariance but doesn't have a connection with the exact terms in transport equations. The eddy viscosity models follow the second philosophy.

Although both pathways are continuously being explored, the factors that hampered the development of either

of them are strong. The higher moment models despite their success in incorporating curvature and rotation, never become popular in industrial applications due to limited stability and accuracy. There are also doubts on the principle of receding influence and the idea of pursuing the accuracy of each term individually instead of considering the error cancelling between terms. In terms of the eddy viscosity models, the major problem is that too much empiricism is involved. There is no clear direction in the model development and terms can be added freely based on personal intuition. This lead to the situation of excessively growing model terms.
In conclusion, the problem of turbulence modelling has reached "a state of near-desperation" due to the aforementioned problems as described by Spalart, and there is no foreseeable breakthrough from either of the pathway. In this context, researchers are turning to the rising techniques of machine learning and searching for new possibilities. Some of the latest results are discussed in the following sections.

2.4. Data-driven turbulence modelling

In this section, a literature review on the study field of data-driven turbulence modelling is presented. There are various turbulence modelling methods and machine learning methods. The former can be put into three categories, namely Direct Numerical Simulation (DNS), Large Eddy Simulation (LES) and Reynolds-Averaged Navier-Stokes (RANS), roughly in the order of high to low accuracy and low to the high involvement of modelling. DNS simulation is used only for data generation since there is little modelling involved in DNS and its results can be regarded as the "truth" or the target to learn from. The remaining LES and RANS both require closure models: subgrid-scale (SGS) models for LES and Reynolds stress models for RANS. The turbulence modelling technique involved in the thesis work is RANS because of its low computational cost and ease of use. Applying the framework to LES can be a future direction.
The machine learning methods include random forest, artificial neural network (ANN), genetic algorithms (GA), sparse regression and many more. The machine learning techniques have different architectures and are suitable for different kinds of problems. The neural network is characterized by its expressive power and thus applicability to a wide range of problems. Whilst due to its complex and highly entangled structure, it is notorious for being obscure and hard to interpret. It is also prone to overfitting. On the opposite, sparse regression has the advantage of being easily interpretable. But it is normally not applicable to complex problems with a large number of inputs. The number of possible combinations between the two sets of methods from the turbulence modelling field and machine learning field is large. For instance, both artificial network and random forest are applied to SGS modelling problems, its advantage over conventional methods is proved both a priori and a posteriori [53–55]. Sparse regression is used in constructing RANS closure models from high-fidelity data. The improvement in model accuracy is proved in different geometries [56–58]. The ways of combining turbulence modelling and machine learning are also numerous. Machine learning can be directly used for the regression problem of finding the best turbulence model, as it is done in the thesis work. It can also act as a classifier for switching between different schemes [59]. Whether there are optimal matches between techniques of two field and whether there is the best way to combine the two are still questions remain unanswered.
In Section 2.4.1, the application of machine learning techniques at different levels of modelling is introduced. Section 2.4.2 discusses the efforts on making the machine learning models explainable. Section 2.4.3 focuses on embedding flow physics to data-driven models.

2.4.1. Apply machine learning on different levels

When introducing data-driven techniques to the regression problem, a distinction can be made on the level of involvement of machine learning techniques. In CFD, there are roughly two levels of modelling work (or approximation) involved. From top to bottom, they are physical modelling (such as RANS, turbulence model) and discretization (numerical schemes). The modelling we usually referred to is physical modelling. Most of the efforts on data-driven modelling are also put into this part. It will be introduced in detail in the following section. However, it is worth noting that there is a methodology that combines turbulence modelling and discretization called implicit turbulence modelling. It treats the numerical diffusion and the diffusion effect of turbulence as a whole. A data-driven discretization scheme that acts similar to implicit turbulence modelling is also developed [60].
When replacing physical modelling with machine learning, there are different levels of involvement. As the machine learning techniques take up more part in the flow model, extra challenges are posed to data-driven models. Their interaction with the rest of the model and the introduced effect on the total model performance requires careful examination.
In introducing machine learning on the lowest level, only some parameters in the model are estimated by data-driven methods. The main body is still based on observation and physics understanding. This is also called the parameter

estimation problem [61]. For instance, Cheung et al. used Bayesian uncertainty quantification techniques and found the most proper stochastic representation of the inadequacy of the Spalart–Allmaras model. Ray et al. [63] calibrated three parameters in $k-\varepsilon$ model with experimental data using a Bayesian approach. The prediction error was reported to reduce by over 40% compared to nominal model parameters. Pal [64] used a deep neural network to predict the eddy viscosity in LES simulation. The proposed model performed particularly well on coarse grids and the computational time was reduced by 2-8 times compared to the dynamic Smagorinsky model.
On a higher level of involvement, researchers use machine learning methods to reconstruct discrepancy fields and reduce model-data inconsistency. In this case, machine learning methods act as a complement to conventional models and helps improve model accuracy. Wang et al. [8] reconstructed RANS model discrepancies with DNS data and obtained excellent prediction improvement on Reynolds stresses. Wu et al. [65] further improved the prediction of mean flow velocity based on the corrected Reynolds stresses overcoming the potential ill-conditioning of RANS equations. Singh et al. [66] developed a neural network augmented Spalart-Allmaras (S-A) model that was capable of improving the prediction on lift and surface pressure of airfoil under separation.
Taking a step further, machine learning algorithms are proved to be able to replace part of or the entire turbulence model. For instance, Tracey et al. [9] used a neural network to predict terms in the S-A eddy viscosity model and reproduced the S-A model results in multiple flow cases. An example of replacing the entire closure model with a machine learning model was shown in [67], in which the subgrid-scale force in LES simulation was modelled by a spatial artificial neural network. The proposed model outperformed ILES and other traditional LES models both in a priori and a posteriori analysis. The work presented in this report also falls into this category. The closure model of the RANS equation is replaced by a well-trained neural network.
Models fully based on data-driven techniques were also proposed. Jin et al. developed Navier-Stokes flow nets (NSFnets), in which the Navier-Stokes equation is directly built into the deep neural network. This is achieved by incorporating the residuals of the governing equations into the loss function of the network. The model performance is evaluated on turbulence channel flow. It is also tested on flow cases that are challenging for traditional CFD solvers, specifically, problems with noisy boundary conditions and unknown fluid properties.

2.4.2. Explainable data-driven turbulence modelling

Most of the machine learning models are opaque and hard to explain. Explainable AI (XAI) is a major research topic in the field of machine learning. Understanding the models helps to debug and improve them. It is also closely related to other important traits of the model, such as fairness, privacy, robustness, causality, etc [69]. In terms of the application of machine learning in physics modelling problems, the goal is not only better models but also new understandings of physics. Towards achieving such a synergy between machine learning and conventional turbulence study, the barrier of interpretability needs to be broken. However, most of the current researches on data-driven turbulence modelling focused on the accuracy and efficiency of the machine learning model and barely touched upon its interpretation. Only preliminary attempts were made on explainable machine learning models. The interpretation is more like validation of the fact that machine learning models have learnt some known physics rather than a way of obtaining novel physics understanding. Utilizing machine learning techniques to accelerate research on physics is far from practice.
There are two paths for achieving more interpretable data-driven models. The straightforward one is to use machine learning techniques with better interpretability. For instance, Beetham and Capecelatro [70] used sparse regression in which the resultant model was in algebraic form and could be interpreted directly. Another path is to use special techniques in the machine learning field to extract information from complex models. Ling [71] used the rule extraction technique to locate model form error, in which the machine learning models were reduced to understandable rules and then possibly transformed to new domain knowledge. Borde et al. trained a convolutional neural network that predicted anisotropic Reynolds tensor in RANS equation. Various interpretation techniques were used in their work. The occlusion and gradient-based sensitivity analysis showed that the near-wall region has the greatest influence on the model performance, which complied with the underlying physics of the flow problem. Bennett and Nijssen trained a deep neural network for predicting turbulent heat fluxes and used the layerwise relevance propagation (LRP) technique to analyse what relationship is learned by the network. Analysis showed that the learned relationship is physically plausible and new information could be extracted. Tan et al. [74] trained a random forest for improving the accuracy of Sparllart-Allmaras model. The model interpretability is improved by evaluating the contribution of each input feature using Shapley Additive Explanations (SHAP) feature explanation tool. Lellep et al. also used SHAP method and succeeded in finding three modes that had the largest contribution to the relaminarization of the wall-bounded shear flow.

2.4.3. Physics-based data-driven turbulence modelling

In data-driven modelling problems, another important research topic is how to embed prior domain knowledge to data-based machine learning techniques. In the case of turbulence modelling, the most important ones are the invariance properties (discussed in detail in Section 3.1.4). They must be incorporated into the machine learning model, otherwise, the model is intrinsically incorrect. The invariance properties are invariance under translation, fixed rotations, reflections of the coordinate system, and Galilean invariance.

Many works were devoted to developing machine learning models with embedding invariances, the earliest of which was done by Ling et al. In their work, symmetry and invariance properties of the turbulence system were built to both random forest regressor and neural network. Two methodologies were proposed, (1)building a basis of invariant inputs and trained the neural network model on the constructed basis (tensor basis neural network, or TBNN) (2)training on an augmented data set [3, 4]. Through incorporating invariances, both the prediction power and the training efficiency were improved. Kaandorp and Dwight developed a tensor basis random forest (TBRF) algorithm for the modelling of Reynolds stress tensor in RANS equation. The model's prediction on various cases is close to the corresponding DNS and experimental data. Zhou et al. developed a non-local frame-independent neural network architecture that embodies Galilean invariance, rotational invariance and permutational invariance. The network was tested on a scalar transport equation under parameterized periodic hill geometry. The testing results indicate that the proposed framework is promising in turbulence modelling problems. Frezat et al. proposed a framework for modelling subgrid-scale flux that can incorporate invariance properties. The model performance with both the soft and hard constraints are evaluated. The merit of the transformation-invariant neural network against network without embedding invariance and conventional model is proved in numerical experiments.

Researchers also succeeded in embedding other known physics into the machine learning model. For instance, Mohan et al. proposed a Convolutional neural network (CNN) framework with embedding incompressibility. Its prediction shows apparent improvement in terms of local conservation of mass.

2.4.4. Others

Apart from the research topics discussed in the above sections, there are also topics that are less studied currently, but still of importance.

During any kind of modelling procedure, either based on physical intuition or data-driven, inevitably model uncertainty is introduced. It is natural to look also into the uncertainty introduced by data-driven modelling. Wu et al. used two metrics, the Mahalanobis distance and the kernel density estimation (KDE) for measuring the distance between feature space of the training flows and that of the flow to be predicted. It is found that the measured distances were positively correlated to the prediction error of the model. In this way, the prediction confidence of the trained model can be estimated a priori based on the distance between features spaces. Scillitoe et al. incorporated uncertainty quantification into data-driven turbulence model by using Mondrian forests algorithm, a variant of random forest [79] with principled uncertainty estimates. According to the comparison between the quantified uncertainty and the prediction error, the uncertainty estimates of Mondrian forest seems to be a good indication of prediction confidence.

There was also research focus on the generalizability of data-driven turbulence models. [80] looked into the model extrapolation problem in data-driven turbulence modelling and related it to the transfer learning problems. Models were compared regarding their performance in data shift. Special attention was paid to the data processing schemes for the development of a more robust model.

3

Theoretical background

In this section, the theoretical background of the thesis work is presented. The first part contains the turbulence modelling theory for RANS. The derivation of its primary equation and the modelling of its closure term (especially $_{\tilde{}}\varepsilon$ eddy viscosity model) are presented. This part covers the physical modelling step involved in CFD. The second part focuses on the computational methods for solving a turbulent flow, including the discretization scheme, SIMPLE algorithm and solver for a linear system of equations. The third part is devoted to machine learning theory, especially neural network algorithms. The architecture and training of the networks are covered.

3.1. RANS and turbulence modelling

RANS is a commonly used Computational Fluid Dynamic(CFD) technique. The idea of RANS and the modelling of its closure terms are discussed in this section. More detailed discussion on this topic is provided in classic textbooks on turbulence modelling, such as [10] and [81].

3.1.1. RANS equation

For incompressible Newtonian fluid, the following governing equation for momentum applies:

$$\frac{\mathrm{D}\boldsymbol{U}}{\mathrm{D}t} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\boldsymbol{U} \tag{3.1}$$

in which $\boldsymbol{U}$ is the velocity vector, ρ is the fluid density, p is the pressure, $\nu := \mu/\rho$ is the kinematic viscosity with ν the dynamic viscosity. It indicates that the acceleration experienced by fluid particles is a result of pressure gradient and diffusion effects.

Equation (3.1) accurately describes incompressible Newtonian fluid. However, solving (3.1) for high Re numbers numerically is computationally expensive. Because due to the spatial and temporal complexity of turbulence, both smaller timesteps and smaller cell sizes are required for fully resolving the flow field. The computational cost grows cubically with regards to the Reynolds number (Re^3). Thus, it is helpful to think of a simpler model of turbulent flow, following which the cost becomes tractable. RANS is one of such simplified simulation methods, in which only the mean velocity field is resolved. The key idea of RANS is embodied in Reynolds decomposition. It refers to the decomposition of instantaneous velocity into mean velocity and fluctuating velocity. It follows that:

$$\boldsymbol{U}(\boldsymbol{x}, t) = \langle\boldsymbol{U}(\boldsymbol{x}, t)\rangle + \boldsymbol{u}(\boldsymbol{x}, t) \tag{3.2}$$

in which $\langle\cdot\rangle$ refers to the time-averaging operation, $\langle\boldsymbol{U}(\boldsymbol{x}, t)\rangle$ is the mean velocity component and $\boldsymbol{u}(\boldsymbol{x}, t)$ is the fluctuating velocity component.

The governing equation for mean velocity $\langle\boldsymbol{U}(\boldsymbol{x}, t)\rangle$ (the RANS equation) is derived by taking the the time average of (3.1):

$$\frac{\bar{\mathrm{D}}\langle U_j\rangle}{\bar{\mathrm{D}}t} = \nu\nabla^2\langle U_j\rangle - \frac{\partial\langle u_i u_j\rangle}{\partial x_i} - \frac{1}{\rho}\frac{\partial\langle p\rangle}{\partial x_j} \tag{3.3}$$

in which

$$\frac{\bar{\mathrm{D}}}{\bar{\mathrm{D}}t} := \frac{\partial}{\partial t} + \langle\boldsymbol{U}\rangle\cdot\nabla \tag{3.4}$$

is the mean substantial derivative.
By comparing (3.1) and (3.3), it is directly noticed that the only difference is caused by the extra term of the covariance between fluctuating velocity in RANS equation $\langle u_i u_j \rangle$. It is a second-order tensor named Reynolds stress tensor. It represents the effect of the fluctuating velocity component on the mean velocity field. The exact transport equations of the Reynolds stress tensor can be derived from the N-S equation. However, the Reynolds transport equation involves higher-order moments of the fluctuating velocities. It is clear that the model can not be closed by deriving transport equations of higher and higher-order moments. This lead to the so-called closure problem of the RANS models. For solving the RANS equation, a turbulence model must be introduced.
All the RANS turbulence models can be put into two categories, the eddy viscosity models and the Reynolds stress transport models. They represent two strategies for the closure problem. The Reynolds stress transport models aim at approximating the terms in the exact Reynolds stress transport(RST) equations as accurate as possible. It can be seen as a systematic way. On the other hand, the eddy viscosity models are not directly connected to the RST. They are based on the turbulent-viscosity hypothesis (or Boussinesq hypothesis), according to which:

$$\langle u_i u_j \rangle = \frac{2}{3} k \delta_{ij} - \nu_T \left(\frac{\partial \langle U_i \rangle}{\partial x_j} + \frac{\partial \langle U_j \rangle}{\partial x_i} \right) \tag{3.5}$$

in which k is the turbulence kinetic energy ($k \coloneqq \frac{1}{2}\langle u_i u_i \rangle$), ν_T is the eddy viscosity (or turbulence viscosity).
The Boussinesq hypothesis is an analogy to the constitutive model of Newtonian fluids. It assumes that the turbulence behaves similar to the diffusion effect in fluid flows. Under this assumption, the modelling of the second-order Reynolds stress tensor is simplified to the modelling of a single scalar ν_T. The eddy viscosity models are probably the most popular turbulence models thanks to their ease of use and computational efficiency. The turbulence model used in this thesis work, the k-ε turbulence model also belongs to this group.
When comparing two types of turbulence models, the RST models are capable of modelling more complex flows compared to the simpler eddy viscosity models. Because in RST models, the curvature and rotation are accounted for. However, opposed to usually two transport equations required by the eddy viscosity models, RST requires six transport equations for six independent Reynolds stress and another transport equation for dissipative timescale. Consequently, RST is computationally more expensive.

3.1.2. $k-\varepsilon$ eddy viscosity model

As mentioned in Section 3.1.1, based on Boussinesq assumption, the problem lies in determining the eddy viscosity ν_T in (3.5). According to dimension analysis, eddy viscosity can be regarded as the multiplication of a velocity and a length:

$$\nu_T = u^* l^* \tag{3.6}$$

Any two variables that cover the velocity and length scale can form a complete eddy viscosity model. In this thesis work, $k-\varepsilon$ model is used. It is a complete model with two transport equations for k and ε. k is the turbulence kinetic energy and it represents the kinetic energy contained in the fluctuating velocity. ε is the dissipation rate of the turbulence kinetic energy. The transport equation for k in $k-\varepsilon$ model is based on the exact transport equation of k with the flux term modelled by the gradient-diffusion hypothesis. The transport equation for ε in contrast is entirely empirical. The transport equations are as follows:

$$\frac{\bar{\mathrm{D}}k}{\bar{\mathrm{D}}t} = \nabla \cdot \left(\frac{\nu_T}{\sigma_k} \nabla k \right) + \mathscr{P} - \varepsilon \tag{3.7}$$

$$\frac{\bar{\mathrm{D}}\varepsilon}{\bar{\mathrm{D}}t} = \nabla \cdot \left(\frac{\nu_T}{\sigma_\varepsilon} \nabla \varepsilon \right) + C_{\varepsilon 1} \frac{\mathscr{P}\varepsilon}{k} - C_{\varepsilon 2} \frac{\varepsilon^2}{k} \tag{3.8}$$

in which $\mathscr{P}$ is the production term. The turbulence quantities can be described by their convection by mean velocity, diffusion, production and dissipation.
The velocity scale required by (3.6) can be directly derived from the turbulence kinetic energy:

$$u^* = c k^{1/2} \tag{3.9}$$

in which c is a constant.
In high Reynolds number flows, ε scales as u_0^3 / l_0. So the following relation between dissipation rate and length scale can be derived:

$$\varepsilon = C_D k^{3/2} / l^* \tag{3.10}$$

By combining (3.9) and (3.10), the following expression for eddy viscosity is obtained:

$$\nu_T = C_D \frac{k^2}{\varepsilon} \tag{3.11}$$

in which c is absorbed into C_D, and $C_D = 0.09$ is a model constant. Detailed description of the transport equation and the involved model constants can be found in [10]. The exact implementation of $k-\varepsilon$ model with a rapid-distortion theory compression term in OpenFOAM is provided in [82].

3.1.3. Considerations on the choice of model

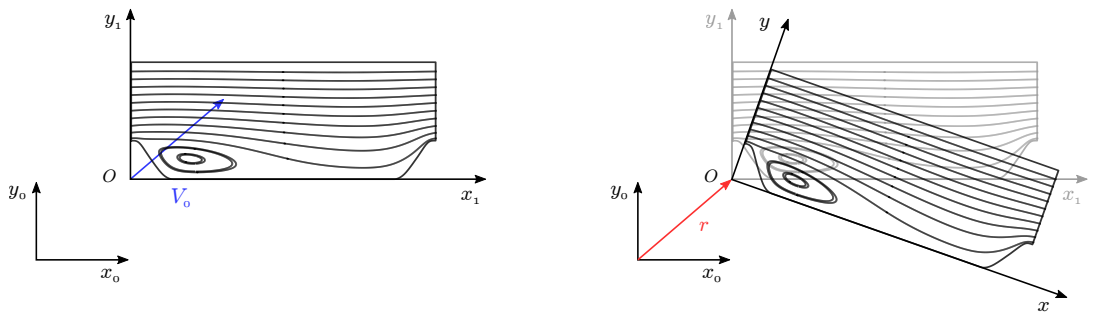
RANS models are usually inferior to LES and DNS in terms of prediction accuracy. Among RANS models, eddy viscosity models are usually less accurate than Reynolds stress models. $k-\varepsilon$ model as an eddy viscosity model despite its limitation in prediction ability, is still selected as the baseline model in the study. The main considerations are as follows:

1. RANS simulation is much computationally cheaper than LES and DNS simulation. Data generation can be done efficiently using RANS models. This is especially important in the model developing stage.
2. In eddy viscosity models, the modelling of Reynolds tensor is transformed into the modelling of a single scalar eddy viscosity. In $k-\varepsilon$ model, only two turbulence quantities are involved. This allows a simpler architecture of the neural network. Besides, the turbulence kinetic energy k has a clear physics interpretation and can be more informative compared to one equation model such as the Spalart–Allmaras model.
3. The accuracy of the trained neural network is closely dependent on the accuracy of the data set. Using low-fidelity data limits the performance of the machine learning model. However, the model developed in this work can be easily transplanted to other data sets. The model accuracy will be improved accordingly.

3.1.4. Symmetries of the governing equation

Invariance properties are crucial in describing flow physics. Three invariance properties of fluid flow are incorporated in the proposed framework, Galilean invariance, translation invariance and rotation invariance. The transformations are visualized in Figure 3.1.

Galilean invariance is a property common to Newtonian mechanics. It states that the motion of objects is the



(a) Galilean invariance

(b) Translation and rotation invariance

Figure 3.1: Invariance properties of the fluid flow

In Figure 3.1a the reference frame is moving in a constant velocity V_0. Figure 3.1b shows the reference frame under a translation $\boldsymbol{r}$ and a rotation.

same regardless of the inertial frame in which the observer resides. In fluid mechanics, it is reflected in the fact that the governing equation refNS remains the same even when the flow velocities have gone through Galilean transformations. The Galilean transformation follows that:

$$\bar{x} = x - Vt, \quad \bar{U} = U - V \tag{3.12}$$

It is assumed that time is completely separable from space, so time in two reference frames is identical:

$$\bar{t} = t \tag{3.13}$$

It can be shown easily that the velocity gradient are Galilean invariant:

$$\frac{\partial U_i}{\partial \bar{x}_j} = \frac{\partial (U_i - V_i)}{\partial x_j} = \frac{\partial U_i}{\partial x_j} \tag{3.14}$$

The material time derivative of velocity is also Galilean invariant:

$$\begin{aligned} \frac{\mathrm{D}\bar{\boldsymbol{U}}}{\mathrm{D}\bar{t}} &= \frac{\partial U_i}{\partial t} + V_j \frac{\partial U_i}{\partial x_j} + \bar{U}_j \frac{\partial U_i}{\partial x_j} \\ &= \frac{\partial U_i}{\partial t} + U_j \frac{\partial U_i}{\partial x_j} \\ &= \frac{\mathrm{D}\boldsymbol{U}}{\mathrm{D}t} \end{aligned} \tag{3.15}$$

So the governing equation of incompressible Newtonian fluid (3.1) is Galilean invariant.
Translation invariance refers to the fact that the flow field is invariant when the reference frame is shifted by a certain distance. Rotation invariance indicates that the flow field is invariant under changes of orientation of the reference frame. It is clear that the flow field has translation and rotation invariance because the frame of observation is irrelevant with the physics law that governs the flow field. It is worth-noting that rotation invariance described here is different from a non-inertial rotating reference frame since, in a rotating frame, there will be the Coriolis force, the centrifugal force and the angular acceleration force. The N-S equation is not invariant to frame rotation.
The symmetries are the essence of fluid physics. The governing equations can be uniquely determined by the symmetry properties. It is essential to have these symmetries also in the neural network models. Details of how this can be achieved are presented in Chapter 4.

3.2. Computational methods

In the last section, the governing equations for the flow field and the models describing the turbulence are introduced. However, how to solve the equations and obtain the velocity fields numerically remain unanswered. In this section, the computational methods used for solving the flow field are introduced. The section starts with the discretization schemes, through which the differential equations are turned into a linear system of equations. Then follows the methods for solving a linear system of equations. The Gauss-Seidel smoother and the multi-grid solver are introduced. In the end, strategies for approaching the coupled set of PDEs (the continuity equation and the momentum equations) are covered.

3.2.1. Discretization schemes

The finite volume method is currently the most used method in fluid mechanics. It divides the flow domain into a finite number of control volumes. The computational nodes (at which the values are computed) locate at the centre of each volume. A sketch of the finite volume discretization is provided in Figure 3.2.

Different from finite difference method, In finite volume method, the integral form of the conservation laws applies. The conservation law of the quantity ϕ is written as:

$$\int_S \rho\phi \boldsymbol{U} \cdot \boldsymbol{n}\, dS = \int_S \Gamma \nabla\phi \cdot \boldsymbol{n}\, dS + \int_\Omega q_\phi\, d\Omega \tag{3.16}$$

All the terms in the equation appear in the integral form. They are dealt with using the quadrature rules, which ask for value at points located at the surface of the cells. The value at non-nodal points is computed by the interpolation schemes. Quadrature rules and interpolation schemes together form the set of techniques required by the discretization procedure.

Surface and volume integral

In the integral form of the conservation laws, there are two types of integrals, the surface integral and the volume integral. Here we consider the 2D case. for the surface integral, each control volume has four faces. The convective term on the left hand side of (3.16) and the diffusive term on the right hand side can then be expressed as:

$$\int_S f\, dS = \int_{S_e} f\, dS + \int_{S_s} f\, dS + \int_{S_w} f\, dS + \int_{S_n} f\, dS \tag{3.17}$$

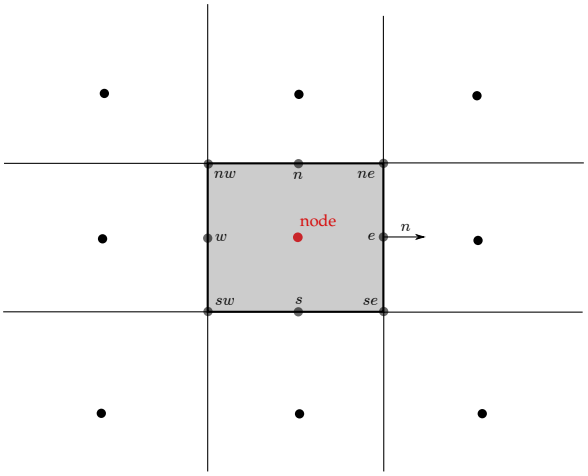


Figure 3.2: Finite volume method

The computational node is marked in red at the centre of the control volume. Four corner points and four midpoints of the surface are labelled according to their relative location to the central node. For instance, the midpoint of the right boundary of the cell is labelled *e*. The surfaces are also referred to by the label of its midpoint.

where f represents the integrand. Since the computational node resides at the centre of the control volume, the integrand on the surface is unknown. Two approximations can be made for the surface integral: (1)The surface integral is approximated by quadrature rules according to values on finite points on the surface. (2)The points on the surface are interpolated by the computational nodes. The interpolation schemes will be discussed later in this section.
The simplest second order quadrature rule is the midpoint rule, according to which the integral on one surface follows:

$$\int_{S_e} f\, dS \approx f_e S_e \tag{3.18}$$

In terms of the volume integral, a similar approximation can be made:

$$\int_{\Omega} q\, d\Omega \approx q_P \Delta\Omega \tag{3.19}$$

in which q_P refers to the value of q at the cell centre. In this case, interpolation is not required.

Interpolation schemes

The most often used interpolation schemes are the first order upwind scheme and the second order linear interpolation scheme. In upwind scheme, the value of ϕ at midpoints is computed as:

$$\phi_e = \begin{cases} \phi_P & \text{if } (\boldsymbol{U}\cdot\boldsymbol{n})_e > 0 \\ \phi_E & \text{if } (\boldsymbol{U}\cdot\boldsymbol{n})_e < 0 \end{cases} \tag{3.20}$$

The upwind scheme will never yield oscillatory results at the cost of being numerically diffusive.
The linear interpolation scheme instead of taking the value of either one of the neighbouring nodes approximates the midpoint by averaging the two neighbouring nodes:

$$\phi_e = \phi_E \lambda_e + \phi_P(1-\lambda_e), \quad \lambda_e = \frac{x_e - x_P}{x_E - x_P} \tag{3.21}$$

Its error reduced faster with decreasing cell size, but the stability is not guaranteed as in the upwind scheme.

3.2.2. Solving for a system of linear equations

The discretized governing equations of all the nodes in the computational domain form a system of linear equations. For simplicity, it is written as:

$$A\boldsymbol{x} = \boldsymbol{b} \tag{3.22}$$

The dimension of the unknown vector $\boldsymbol{x}$ equals the number of cells in the fluid domain (in our case around 40000 cells). This number is so large that proper techniques are required for solving the equations in a numerically efficient way.
For solving the momentum equations, the Gauss-Seidel smoother (sparse linear solvers are also referred to as smoother) is used. It is an iterative method defined as follows:

$$L_* \boldsymbol{x}^{(k+1)} = \boldsymbol{b} - U\boldsymbol{x}^{(k)} \tag{3.23}$$

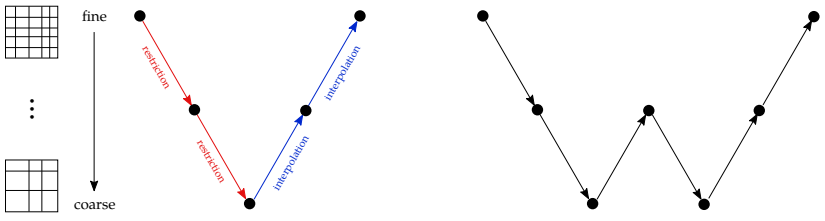
in which the superscript (k) represents the k^{th} iteration of unknown $\boldsymbol{x}$, superscript $(k+1)$ is the next approximation of $\boldsymbol{x}$. L_* is the lower triangular component of A, U is the strictly upper triangular component of A, specifically:

$$A = L_* + U = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} + \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \tag{3.24}$$

The advantage of decomposing A in this way is that the value of elements before $x_j^{(k+1)}$ $(x_1^{(k+1)}, \dots, x_{j-1}^{(k+1)})$ in the current iteration is known. In this way, $\boldsymbol{x}^{(k+1)}$ can be computed sequentially and there is no need to calculate the inverse of matrices. It follows that:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} (b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}) \tag{3.25}$$

Smoothers such as the Gauss-Seidel smoother can reduce the high-frequency (oscillatory) components of the error quite efficiently. However, in terms of the low-frequency (smooth) components, they are not so effective. For a faster convergence, GAMG(Generalised geometric/algebraic multi-grid) solver can be used. The idea of the multi-grid method is to apply smoothers on a hierarchy of grids ranging from fine to coarse. Because the high or low-frequency components of the error are relative to the computational mesh. In other words, the high-frequency error component on a coarse mesh becomes the low-frequency component on a finer mesh. By cycling through meshes from coarse to fine or reversely, components of error with different frequencies can be reduced by a similar rate regardless of the mesh size. The process of switching from coarse fine mesh to coarse mesh is called restriction/injection. Correspondingly, the operation from coarse to fine mesh is referred to as interpolation/prolongation.
There are various strategies for cycling through multiple mesh levels. The one used in OpenFOAM GAMG solver is the V-cycle. The solver starts the smoother iteration on the finest mesh, where the computational mesh is defined. After several (or only one) iteration of the smoother, it continues on to the coarser levels. The process is repeated until it reaches the coarsest level. Then a reverse process follows. The solver returns from the coarsest back to the finest grid. The procedure is sketched in Figure 3.3a. The only problem remains is that how to obtain



(a) V-cycle **(b)** W-cycle

Figure 3.3: Cycling of multi-grid solver

the multi-level grids, also known as the agglomeration algorithm. There are two ways, namely the geometric and the algebraic multi-grid (AMG). In the former way, the geometrical neighbouring cells are merged to generate a coarser grid. In the latter one, the merge of cells happens algebraically on the matrices and the governing equations for the coarse grid are obtained directly. The former is proved to be superior to the latter. Details of the algorithm as implemented in OpenFOAM can be found in the source code [83].

3.2.3. SIMPLE algorithm

The algorithm used for solving the continuity and momentum equations is the Semi-Implicit Method for Pressure Linked Equations [84] (SIMPLE) implemented in OpenFOAM. It is an algorithm developed in the early 1970s and is widely used in CFD.

A major difficulty for solving the N-S equations is that there is no independent equation for the main source term pressure in the momentum equations. In compressible flows, the pressure field is related to the continuity equation through the equation of state. However, such an approach is not applicable in incompressible flows because the continuity equation is a purely kinematic equation for the velocity field instead of a dynamic equation [85]. :

$$\nabla \cdot \boldsymbol{U} = \boldsymbol{0} \tag{3.26}$$

This is the so-called pressure-velocity coupling problem. A technique for solving such difficulties is to construct the pressure field in a way such that the continuity equation is satisfied, as it is done in the SIMPLE algorithm. The sequence of operations [86, 87] are as follows:

1. Make an initial guess of the pressure field $p*$ and solve the momentum equations for the velocities $\boldsymbol{U}*$.
2. Solve the pressure correction equation for the pressure correction p'. The pressure correction equation is derived from the continuity equation. It is susceptible to diverging so under-relaxation is applied when updating the pressure. $p = p^* + \alpha p'$, α is a factor between 0 and 1.
3. Calculate the velocity correction U' based on the pressure correction obtained in the last step. In this step, a major approximation is made. Terms are dropped to simplify the correction equations.
4. Solve for other quantities that influence the flow field such as turbulence quantities.
5. Return to the first step with the corrected pressure p as the new guess.

The whole procedure is repeated until the solution converges.

3.3. Artificial neural networks

In this section, the artificial neural network theory that is related to the thesis work is presented. The section starts with the architecture of basic Multilayer Perceptrons(MLP) networks and their working principle. Then the concepts related to neural network training such as the loss function and gradient descent algorithm are explained. It follows a description of the general workflow of a training process. In the end, the important concept of permutation invariance is explained.

3.3.1. Architecture

There are various neural network architectures suitable for various types of problems. The most basic one is the Multi-Layer Perceptron (MLP), also referred to as the "vanilla" neural network. It is a feed-forward network composed of several layers of neurons (or perceptrons, units) fully connected one to another. A sketch of the MLP neural network is presented in Figure 3.4. Circles in the sketch represent the neurons. A neuron essentially is an activation function that maps the weighted input to an output. The weights reside in the lines connecting the circles. Layers are categorized into the input layer, hidden layer, and output layer, according to the sequence of processes in the network. The network shown in the sketch has two hidden layers.

An MLP works in the following way: First, the inputs are fed into the input layer. Then in the hidden layers, each neuron takes the weighted sum of all the outputs of neurons from the upstream layer as the input, evaluated it according to the activation function and spits out a single output. Taking the first neuron in the first hidden layer $a_0^{(1)}$ as an example (subscript 0 represents the first neuron in the layer, and superscript(1) represent the second layer in the neural network), its output is determined as follows:

$$a_0^{(1)} = \phi(w_{0,0}a_0^{(0)} + w_{0,1}a_1^{(0)} + ... + w_{0,n}a_n^{(0)} + b_0) \tag{3.27}$$

in which ϕ refers to the activation function, $w_{i,j}$ represents the weight corresponding to the connection between the i^{th} neuron in the downstream layer and the j^{th} neuron in the upstream layer, b_0 represents the bias for that layer.

For the rest of the neurons $a_1^{(1)}$, $a_2^{(1)}$, etc., the same procedure applies. It is clear that the terms inside the activation function is a linear transformation of the outputs of the last layer. So (3.27) can be vectorized as follows:

$$\boldsymbol{a}^{(1)} = \phi(\boldsymbol{W}^{(0)}\boldsymbol{a}^{(0)} + \boldsymbol{b}) \tag{3.28}$$

Equation (3.28) applies for all the hidden layers and the output layer. The output of the output layer is then the output of the neural network.

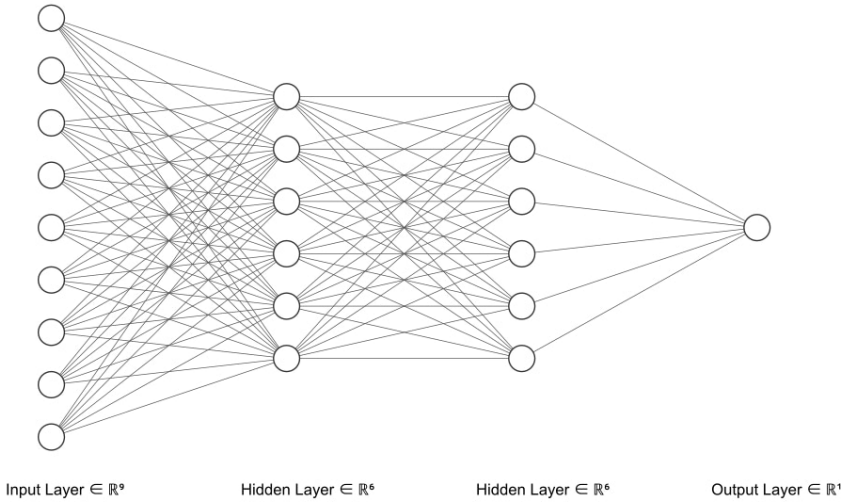
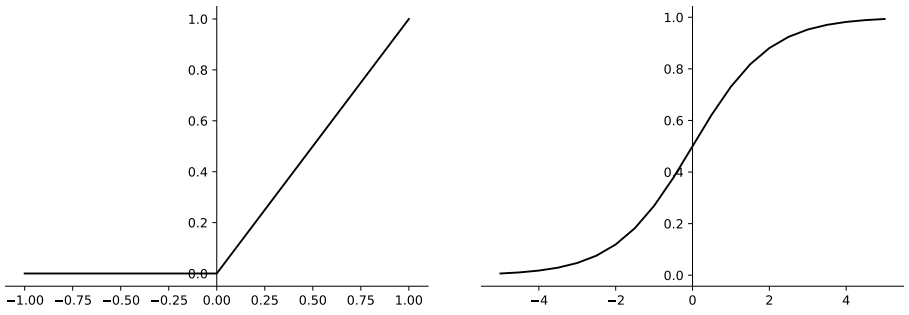


Figure 3.4: Schematics of an MLP neural network architecture[88]

It is clear that apart from the activation function, the other operations performed are all linear transformations. If the activation function is also linear, according to knowledge in linear algebra, the model can be simplified to a two-layer input-output model. Such models lack universality and are not capable of approximate complex functions. So usually, nonlinear activation functions are used. The most commonly used activation function is the rectified linear activation function (ReLU), which is also used in this work. It is a piecewise linear function described by the following expression:

$$f(x) = max(0, x) \tag{3.29}$$

Figure 3.5a shows the output of ReLU with regards to the input. When the input is less than zero, its output



(a) ReLU **(b)** Sigmoid function

Figure 3.5: Activation functions

Figure 3.5b shows the graph of sigmoid function: $S(x) = \frac{1}{1+e^{-x}}$. Mostly used for the prediction of probability since it has an output range of $(0, 1)$. However, it has the issue of gradient saturation and can cause a vanishing gradient during training.

is zero. When the input is larger than zero, its output is the input itself. The ReLU activation function has the advantage of fewer vanishing gradient problems and it is efficient in terms of computation.

3.3.2. Loss function

Regardless of the complex architecture that a neural network may possess, it is nothing more than a function with a set of tunable parameters, its weights and biases. The model performance can simply be evaluated as any other

model by the sum of squared error of the prediction:

$$R(\theta) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{3.30}$$

in which θ refers to the complete set of weights and biases in the model, y_i refers to the prediction of the model and $\hat{y}_i$ refers to the true value.
There are also other types of loss functions, such as the cross-entropy loss function. The choice of the loss function is related to the type of problem. For regression problems, in which the prediction on a value is made, the sum of squared error is the proper choice. For classification problems, cross-entropy is preferred. In our case, the continuum value of the turbulence quantities are predicted, so the sum of squared error is adopted.
For avoiding the overfitting problem, a penalty term on the magnitude of weights is usually added to the loss function. Details for the overfitting issue and solutions to it is introduced later in Section 3.3.6.

3.3.3. Gradient descent

It is proved that multilayer feedforward neural networks are universal approximators[89], which means that a network is capable of approximate any function to any desired level of accuracy as long as there are enough parameters and the parameters are set to proper values. The only problem left is then to decide the value of all the parameters in the model in an efficient way.
The problem of determining the model parameters can be regarded as an optimization problem. It is formulated as finding the proper set of parameters θ such that the loss $R(\theta)$ is as small as possible.
In solving the problem of finding the minimum, the gradient descent algorithm is used. First, the gradient of $R(\theta)$ regarding all the parameters are computed. Then by updating the parameters according to the gradient, ideally the minimum can be reached. The algorithm is as follows:

$$\theta_{n+1} = \theta_n - \gamma \nabla R(\theta_n) \tag{3.31}$$

in which γ is the learning rate. The larger the learning rate, the larger change is made to the model parameters at each step.
The remaining problem is then to find the gradient of the loss function $\nabla R(\theta_n)$. A technique called backpropagation is used. In backpropagation, the gradient is computed starting from the output and then following the chain rule traversing all the parameters upstream.
Taking the simple example of computing the derivative of an output $a_i^{(L)}$ with regard to the weight $w_{j,i}^{(L-1)}$ (the weight connecting the j^{th} node in the $(L-1)^{\text{th}}$ layer with the i^{th} node in the L^{th} layer) as present in (3.27), according to chain rule, it is simply:

$$\frac{\partial a_i^{(L)}}{\partial w_{j,i}^{(L-1)}} = \frac{\partial z_i^{(L)}}{\partial w_{j,i}^{(L-1)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \tag{3.32}$$

$$= a_j^{(L-1)} \cdot \phi'(z_i^{(L)}) \tag{3.33}$$

in which $z_i^{(L)} = w_{i,0} a_0^{(L-1)} + w_{i,1} a_1^{(L-1)} + ... + w_{i,n} a_n^{(L-1)} + b_i$. The directives of biases can be computed in a similar way as the weights. The procedure applies to each of the neurons in the network.
Another thing to notice in practice is that the training is performed in batches. The entire training set is randomly segregated into smaller batches. The loss function as computed in (4.30) is over N samples in a batch instead of the entire training set. Such a strategy approximate the gradient with regard to the entire training set with the gradient with regard to a small batch chosen from it. It can avoid the large amount of computation required for each gradient descent update with the cost of a lower convergence rate.

3.3.4. Initialization

Section 3.3.3 provides the algorithm for updating the model parameters based on gradient descent. In this section, how the model parameters are initialized is discussed. Initialization is the first step in the training process. It can influence greatly the amount of training time needed as well as the final model performance. An extreme example of bad initialization is that if all the model parameters are set to zero at the beginning, the gradient descent algorithm will completely fail, and the model output will remain zero. On the other hand, if the model parameters are initialized as very large values, the training process can barely proceed. The cost will oscillate a lot due to exploding gradient and the network may never reach convergence.

For a proper initialization of the model parameters, the following rules of thumb can be applied: (1)The mean of the parameters is zero. (2)The variance of the parameters of each layer should be at the same level and generally speaking the variance should be small such that the weights are not far from zero.
In the thesis work, the linear layers in the network are initialized by Kaiming He initialization [90], which takes the nonlinear activation function into consideration when deciding the proper distribution of the weights. The weights are initialized by the following normal distribution:

$$W \sim \mathcal{N}(0, \frac{2}{n^l}) \tag{3.34}$$

in which n^l is the number of incoming connections to the l^{th} layer.

3.3.5. Feature scaling

Feature scaling is the procedure of manipulating the data and transform it into certain ranges. It is proved crucial in the context of machine learning, specifically for gradient-based algorithms[91]. Feature scaling is particularly useful in engineering problems where the input features may have different units and spread in various ranges. It is shown that by using proper feature scaling, the training process can be accelerated by an order of magnitude [92]).
As discussed earlier, in gradient descent algorithm, the change made to each parameter is proportional to the gradient of the loss function (see Equation 3.31). Substituting the expression for the loss function, the following can be derived:

$$\theta_{n+1} = \theta_n - \gamma \nabla \sum_{i=1}^{N} (h_\theta(x_i) - \hat{y}_i)^2 \tag{3.35}$$

with h_θ representing the neural network model whose tunable parameters are represented by θ. By applying chain rules, it is evident that the gradient is proportional to the input features:

$$\nabla (h_\theta(x_i) - \hat{y}_i) \propto x_i \tag{3.36}$$

This explains why applying proper feature scaling is important. The absolute value of features will influence the steps taken by the gradient descent algorithm. Having all the features at a similar scale is helpful for faster convergence.
There are two techniques of feature scaling, namely normalization and standardization. Normalization is the procedure of shifting and scaling the features such that they fall between 0 and 1. It follows that:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.37}$$

The second technique standardization is scaling the features in a way that makes the mean value of the features equals zero and the standard deviation of the features equals one:

$$x' = \frac{x - \mu}{\sigma} \tag{3.38}$$

in which μ is the mean and σ is the standard deviation of the features.
Choosing whether normalization or standardization as the feature scaling technique is case-specific and there are no hard rules for it. Furthermore, for physics modelling questions, there is another layer to it. The physical meaning of the features must also be taken into consideration when applying feature scaling. For instance, a common practice in aerodynamics when dealing with lift force is to divide it by the surface area and the dynamic pressure:

$$C_L = \frac{L}{qS} \tag{3.39}$$

By practising nondimensionalization like this, the trivial parameters such as the surface area and the freestream velocity are ruled out. A more general model for lift force can be obtained.
The idea of nondimensionalization also applies to machine learning physics modelling problems. Scaling the data such that it is advantageous for training as well as physically reasonable requires careful study.
The method used in this thesis work has gone through careful experiments and is particularly designed for the turbulence modelling scenario. Although strictly speaking, it can't be classified as either of the above-mentioned techniques, in this thesis report, the scaling method is referred to as normalization. The details of the feature scaling operations applied to the related input feature as well as output labels is discussed in Chapter 4.

3.3.6. Overfitting and measures

Neural networks are characterized by a large number of tunable parameters, and it is prone to overparameterization. In such a case, the model overfits the training data. It doesn't learn the underlying general rules from the data, instead, it remembers the training dataset. The model with an overfitting issue shows bad generalizability to unseen datasets. So overfitting is a problem we definitely want to avoid.

There are many measures to prevent overfitting. The first one is applying an early stopping rule. Since the model is gradually fit to the training data as the training goes, it is possible to stop training before overfitting happens. The strategy is to monitor the validation error throughout the training process. A rise in the validation error can be a sign of overfitting.

The second measure is to perform weight decay, also called regularization. A penalty term for large weights is added to the loss function, such as the L2 norm of the weights. In this way, the loss function will increase if the weights of the network get unreasonably large. The balance between error and regularization term in the loss function can be adjusted through a tuning parameter λ multiplying the regularization term. The loss function with regularization term is then:

$$J(\theta) = R(\theta) + \lambda \sum_{i=1}^{n} w_i^2 \tag{3.40}$$

When computing the gradient of the loss function with regularization, an extra term $2w_i$ will be added. It will push the model weights closer to zero. Sometimes the biases b_j will also be added to the regularization term (also the case in PyTorch). The working principle is the same.

The third measure is to add dropout layers to the neural network. It is a technique proposed by Hinton et al. in [93] and [94]. During training, some randomly chosen units in the network are temporarily 'removed' by the dropout layers. Their connections with the upstream units and downstream units are ignored. By dropping out units in a stochastic way, the co-adaptation between units can be prevented. Co-adaptation means the counteract between units. For instance, the negative effect made by some nodes might be cancelled out by some other nodes. The overall performance on the training set might not be affected but there will be redundancy in the network. Moreover, the redundant part might destabilize the model performance on unseen datasets and cause huge oscillation in the prediction.

3.3.7. Training of neural networks

With all the necessary components on hand, the entire workflow of training a neural network is presented below:

1. Design of the network architecture. The network architecture is decided according to the specific problem, including the type of network, hyperparameters, activation functions, etc. It is decisive in ensuring the desired properties of the network (symmetries for instance).
2. Data generation. Depending on the data required, this step can be very time-consuming or the data can be hard to access.
3. Data preprocessing. Proper scaling of the input data is critical for efficient training.
4. Initialization of the network. The weights of the network need to be initialized properly. Otherwise, the training may diverge.
5. Training. The network weights are updated according to the gradient and the learning rate. Validation errors should be monitored to prevent the overfitting issue.
6. Validation. The network performance should be evaluated on unseen cases.

3.3.8. Permutation invariance

In Section 3.1.4, three invariance properties of flow physics are introduced. The remaining invariance property related to the neural network architecture is explained here. Permutation invariance is a property that is common in all kinds of multivariate functions. If a function is permutation invariant, then its outputs are irrelevant to the ordering of its inputs. A simple example of a permutation invariant function is as follows:

$$f(x_1, x_2, x_3) = x_1 + x_2 + x_3 + x_1x_2 + x_1x_3 + x_2x_3 + x_1x_2x_3 \tag{3.41}$$

It is clear that switching x_1, x_2 and x_3 arbitrarily doesn't influence the value of $f(x_1, x_2, x_3)$. Because all the inputs have equal status in the function. Another typical permutation invariant function is taking the maximum:

$$f(x_1, x_2, x_3) = max(x_1, x_2, x_3) \tag{3.42}$$

In the context of neural networks, permutation invariance specifically targets those with sets as inputs. The idea is the same as for functions. Specifically, the ordering of elements in the input set should not influence the network

output. For instance, if we have a set of points in a 2D plane that follows the silhouette of an object, no matter in which order we input those points to the networks, the prediction on the type of the object should not change.

4

Methodology

In this chapter, the methodology of the thesis work is introduced thoroughly. Section 4.1 presents the setup for the CFD simulation and the obtained characteristic flow cases. Section 4.2 explains how the non-locality of the neural network model is achieved. In Section 4.3, the selection of input features and the operations applied for normalization are described.
In Section 4.4 and Section 4.5, two neural network architectures used in this thesis are explained. The pointNet architecture is a variation of the one proposed in [37]. It is rather simple and straightforward and gives a good example of how the permutation invariance can be accomplished through the summation operation. For the pointNet architecture adopted in this work, as a trade-off for its simplicity, rotation invariance is guaranteed by using only the rotation-invariant features at the beginning rather than embedding it into the network. The VCNN network is developed based on the network in [12]. It has a more sophisticated structure and is capable of dealing with rotation variant input features and transform them into rotation invariant form. In this sense, the VCNN architecture is more powerful and has higher adaptability to various application scenarios.
Section 4.6 describes the topics related to neural network training, including the dataset, and regularization. This section applies to both pointNet and VCNN architectures. Section 4.7 presents the coupling between neural network and flow solver simpleFOAM. The data exchange between the two components and the sequence of operations are explained. At the end of the chapter, the software and libraries used are listed.

4.1. CFD simulation

4.1.1. Flow case setups

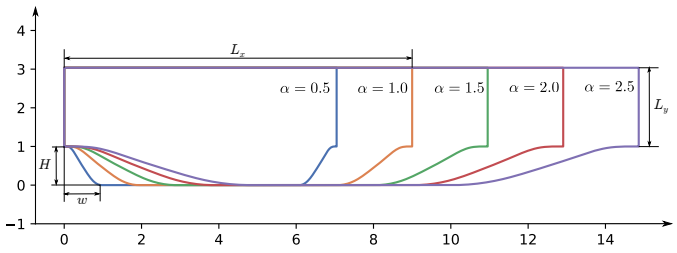


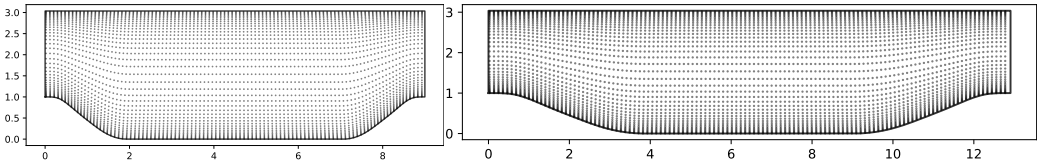
Figure 4.1: Periodic hill geometries with varying slope parameters α.

The geometry used in this work is the 2D parameterized periodic hill. The height of the hill (H) is fixed to 1m and the geometry is uniquely determined by the slope parameter α, which is defined as:

$$\alpha = \frac{w}{1.93} \tag{4.1}$$

where w is the width of the slope. Figure 4.1 shows the shape of the flow domain with varying slope parameters. Hills with larger slope parameters are gentler, whilst hills with smaller slope parameters are steeper.

The mesh for CFD computation is generated by blockMesh utility provided in OpenFOAM. Due to the simplicity of the geometry, the whole flow field is treated as a single block. The number of grid points in y direction (N_y) is fixed to 200 and it is denser towards to upper and lower tunnel walls. The number of grid points in x direction (N_x) varies according to the slope parameter ($N_x = 200 + 200 \times (\alpha - 1) \times 0.2$) such that all flow cases has roughly the same mesh density. With the factor 0.2 in the equation, the number of cells increases by 800 as the slope parameter increases by 0.1. The mesh distribution is uniform in x direction and is concentrated to the boundary in y direction. Examples of the generated mesh with $\alpha = 1.0$ and $\alpha = 2.0$ are presented in Figure 4.2.
The boundary conditions for the periodic hill case are straightforward. The upper and lower boundaries are fixed



(a) Mesh for $\alpha = 1.0$ **(b)** Mesh for $\alpha = 2.0$

Figure 4.2: Example meshes ($\alpha = 1.0$ and $\alpha = 2.0$)

Each black dot represents center of a cell. For clarity, the mesh is sampled every 2 cells in x direction and every 5 cells in y direction.

wall. The inlet and outlet are periodic boundary conditions. A flow velocity of $1\,\mathrm{m\,s^{-1}}$ is specified at the inlet. In terms of the transport properties, the flow Reynolds number is 10595, at which the flow is strongly turbulent. Since the reference velocity (velocity at the inlet) is $1\,\mathrm{m\,s^{-1}}$ and the reference length is 1m (height of the hill), the kinematic viscosity is calculated to be 9.4652e-5$\mathrm{m^2\,s^{-1}}$.
The turbulence model used for data generation is the $k-\varepsilon$ model. The algorithm for solving the governing equation is the SIMPLE algorithm. The theory of the turbulence model and solver algorithm is described previously in Chapter 3.

4.1.2. Characteristic flow fields

A typical flow case with $\alpha = 1.5$ is visualized in Figure 4.3. The figures present contours of u, k, ε and ν_T. The

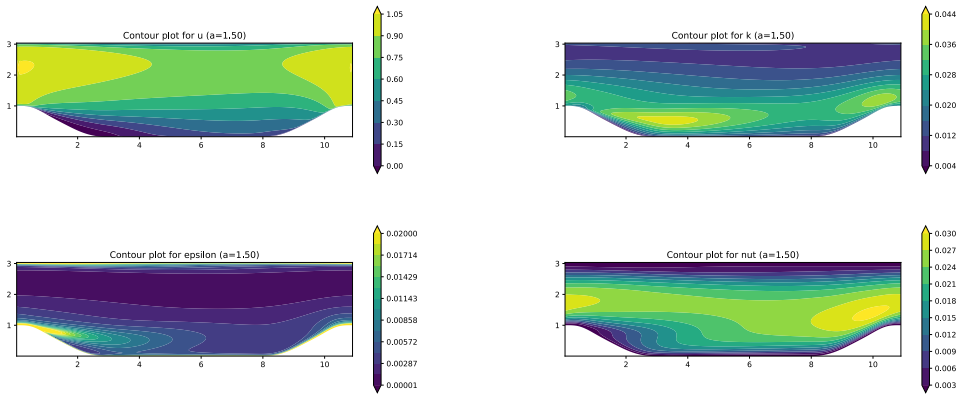


Figure 4.3: Flow case $\alpha = 1.5$

periodic hill case is characterized by the recirculation region at the back of the hill, where the velocity component in x direction is small and the turbulence kinetic energy is high.
It is also noticed that the turbulence quantities occupy a broad range. For instance, ε ranges from near zero to around 0.02 and it changes rapidly near the wall as expected in high Re flows. The distribution of turbulence quantities poses an extra challenge to neural network training as will be discussed later in Section 4.3.

4.2. A non-local constitutive model

The central idea of the developed non-local neural network is to predict the turbulence quantities of a cell according to the mean flow properties of its neighbouring cells. For this purpose, an influence region need to

be defined. As shown in the transport equations for turbulence quantities Equation 3.7, the non-local behaviour includes convection and diffusion. The two effects are considered separately when computing the size of the stencil. Convection is performed by the mean velocity. So the size and shape of the region that influence the point of interest should be related to the local mean flow velocity. The convection effect in a certain direction is proportional to the velocity component in that direction. The larger the velocity component, the larger the length of influence. The diffusion effect on the other hand is isotropic. Even in the direction perpendicular to the local velocity, the length of influence is non-zero and related to the diffusion coefficient.
Based on the above observation on turbulent physics, the influence region is approximated by an ellipse with its major axis aligned with local velocity. The length of the major axis is correlated with local mean flow velocity and the minor axis is decided according to diffusive properties of the flow. Ideally, the size of the influence region should be decided quantitatively based on the transport equations. However, due to the complexity of the transport equations, an exact analytic solution for the influence region is not available. Instead, we used the analytic solution of a 1D convection-diffusion equation with related coefficients chosen empirically. It is assumed that the simplified calculation of the influence length is sufficient for our purpose. The detailed derivation of the influence region of a 1D convection-diffusion equation can be found in [17]. The length of the major axis l_1 and the minor axis l_2 follows:

$$l_1 = \left| \frac{2\nu \log\epsilon}{\sqrt{|\boldsymbol{u}|^2 + 4\nu\zeta} - |\boldsymbol{u}|} \right|, \quad l_2 = \left| \sqrt{\frac{\nu}{\zeta}} \log\epsilon \right| \tag{4.2}$$

in which $\nu = 0.1$ is the diffusion coefficient, $\zeta = 3$ is the dissipation coefficient and $\epsilon = 0.2$ is the error tolerance. The error tolerance is a value between 0 and 1. The smaller the error tolerance, the more accurate, and the larger the influence region.
Following Equation 4.2, the major axis increases approximately linearly against the local mean flow velocity. The minor axis is a constant and is irrelevant to the mean flow velocity. When the mean flow velocity is zero, the major axis is equal to the minor axis. A sketch for the influence region of several sample points in the flow case is provided in Figure 4.4. Due to the varying mesh density, the number of cell points that fall into the ellipse can

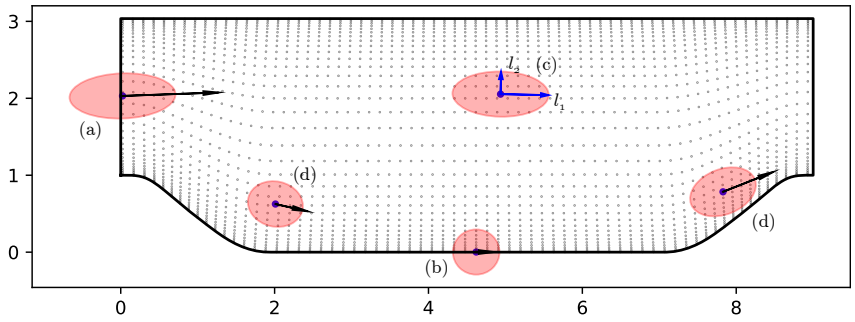


Figure 4.4: Shape of the stencils at various locations in the flow domain.

For points such as (a), periodic boundary condition applies. Flow data at the inlet is concatenated to the outlet and vice versa. For points whose stencil involves the wall boundaries at the bottom and the top of the channel (point (b) in the figure), samples are taken from the fluid domain. Points (d) indicates how the stencils are aligned with the local mean velocity field. The major axis of point (c) in the main flow is largely due to its large local mean velocity. The mesh points are sampled for clearer visualization and are presented in grey dots. The major axis l_1 and the minor axis l_2 as computed by Equation 4.2 are indicated at point (c) in blue.

vary a lot from location to location. For the convenience of training, a fixed number of points are usually sampled inside the stencil. If the number of samples in the stencil is set to 200, then in those stencils with more than 200 points, random points are removed. This practice of sampling will introduce randomness to neural network prediction. So during validation, full stencils are used for better consistency at all points.

4.3. Input features, outputs and normalization

4.3.1. Inputs and outputs

The selection of input features for turbulence modelling is constraint by the flow physics in two aspects:

1. The invariance properties. If the input features have certain invariance properties, the model will directly inherit those properties.

2. Relation to the output. If the inputs and outputs are irrelevant, the model obtained is incorrect, even if it performs well.

Besides, the inputs should supply sufficient information. This requirement sometimes conflicts with the first one. For instance, if the velocity magnitude is used instead of the velocity vector, although rotation invariance is guaranteed, there will be information loss.
Following the above guidelines, the features are selected and presented in Table 4.1. Feature 1-4, the relative

index	features	definition	description
1	x'	$\frac{x-x_0}{\|\boldsymbol{x}-\boldsymbol{x}_0\|+\epsilon_0}$	relative x-coordinate to cloud center
2	y'	$\frac{y-y_0}{\|\boldsymbol{x}-\boldsymbol{x}_0\|+\epsilon_0}$	relative y-coordinate to cloud center
3	u	$u-u_0$	velocity component (relative to center point) in x direction
4	v	$v-v_0$	velocity component (relative to center point) in y direction
5	s	$\|\|\boldsymbol{s}\|\|$	magnitude of the strain rate tensor
6	b	1(yes)/0(no)	boundary cell indicator
7	θ	-	cell volume
8	u	$\|\boldsymbol{u}\|$	velocity magnitude
9	η	$\min(\hat{\eta}/l_\delta, 1)$	wall distance function
10	r	$\frac{\epsilon_r}{\sqrt{x'^2+y'^2}+\epsilon_r}$	proximity to cloud center
11	r'	$\epsilon_{r'}-\frac{\boldsymbol{u}^\top \boldsymbol{x}'}{\|\boldsymbol{x}'\|^2}$	proximity in local velocity frame

Table 4.1: Input features.

x_0, y_0, $\boldsymbol{x}_0$ are the coordinates of cloud center. $\epsilon_0 (= 10^{-5})$ is to avoid divided by zero. $\hat{\eta}$ is the raw wall distance. $\epsilon_r (= 0.01)$ and $\epsilon_{r'} = |\boldsymbol{u}|/|\boldsymbol{x}'|$ are used to transform the function into desired range. l_δ is a characteristic length of the boundary layer.

coordinates and the relative velocity components are vectors. They are translation and Galilean invariance being relative to the cloud centre. Features 5-11 are scalar features. Except for the two invariance properties possessed by the vector features, they also have rotation invariance.
In VCNN, all 11 features are taken as the inputs. Rotation invariance is achieved through pairwise projection as will be discussed in Section 4.5. Whilst in pointNet, only the 7 scalar features are used. PointNet is rotation invariant from the inputs.
The neural network outputs are identical to that of $k-\varepsilon$ turbulence model. There are two outputs, turbulence kinetic energy k and its dissipation rate ε. With these two quantities at hand, the turbulence viscosity ν_T can be computed through Equation 3.11.

4.3.2. Normalization

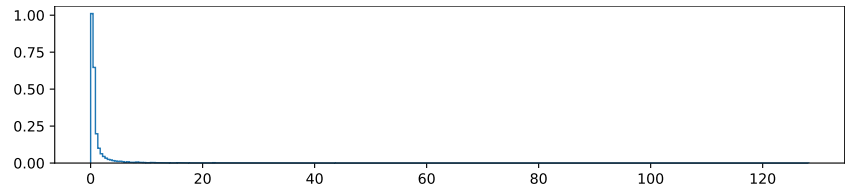
By inspecting the distribution of input features and outputs in our periodic hill case, it is found that some of them are in a range that is not suitable for neural network training. They are the strain rate magnitude s, the cell volume θ, the turbulence kinetic energy k and the turbulence kinetic energy dissipation rate ε. Their distribution in flow case $\alpha = 1.5$ are shown in Figure 4.5.
The distribution of strain rate magnitude is concentrated to the lower end. s of most of the cell points are below 3. Based on this character, the strain rate magnitude s is capped to 3:

$$s = \max(\hat{s}, 3) \tag{4.3}$$

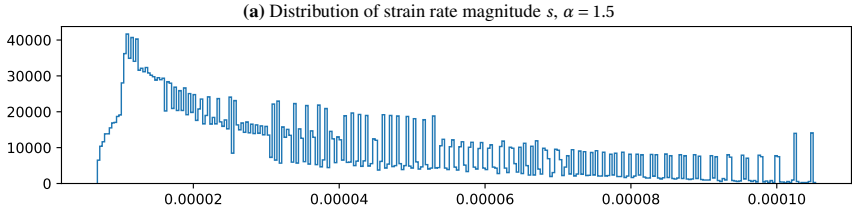
The hat here represents the original data before scaling. The information loss due to the simple capping is considered acceptable, considering that the points with large strain rate magnitude mainly locate near the wall and those regions are less important in our case.
The distribution of cell volume is relatively even. A simple scaling among all the samples in a stencil is applied such that its range is adjusted to $(0, 1)$:

$$\theta = \frac{\hat{\theta}}{\bar{\theta}} \tag{4.4}$$

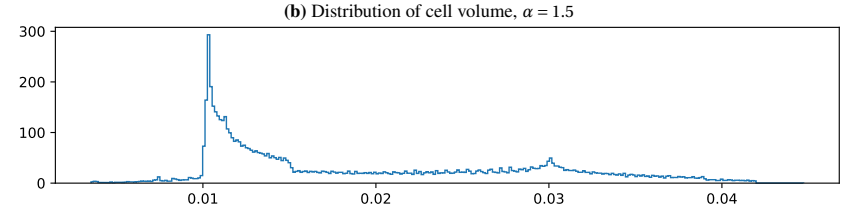
in which $\bar{\theta}$ is the mean cell volume in the stencil.
The distribution of k is roughly between 0 and 0.05. Simple scaling is performed by dividing it by reference



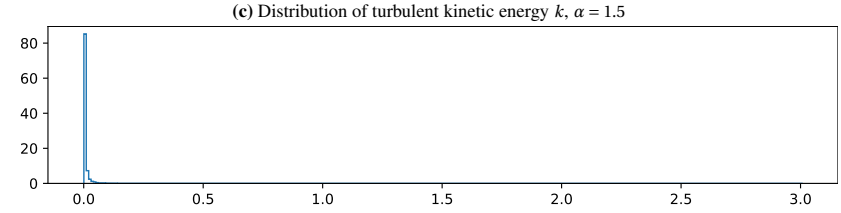
(a) Distribution of strain rate magnitude s, $\alpha = 1.5$



(b) Distribution of cell volume, $\alpha = 1.5$



(c) Distribution of turbulent kinetic energy k, $\alpha = 1.5$



(d) Distribution of dissipation rate ε, $\alpha = 1.5$

Figure 4.5: Distribution of the original input features, $\alpha = 1.5$

turbulence kinetic energy:

$$k = \frac{\hat{k}}{3/2(U_{\text{ref}} I)^2} \tag{4.5}$$

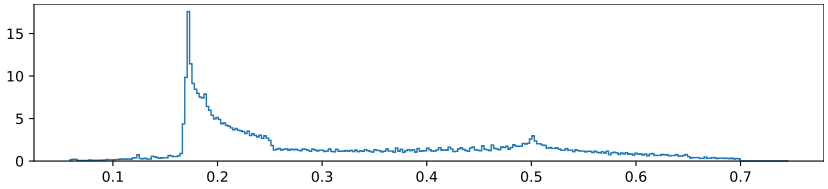
in which $\hat{k}$ is the raw data of k, $U_{\text{ref}} = 1$ is the reference velocity, $I = u/\langle U \rangle = 0.2$ is the estimated turbulence intensity.
The distribution of ε is similar to that of the strain rate magnitude. However, the large dissipation rate region is important. A simple capping doesn't suit our needs. The normalization of ε is done by taking the logarithm, considering that most of the samples are clustered below 0.1 and there are some extreme samples above 3.

$$\varepsilon = \frac{1}{2} \log_{10} \hat{\varepsilon} \tag{4.6}$$

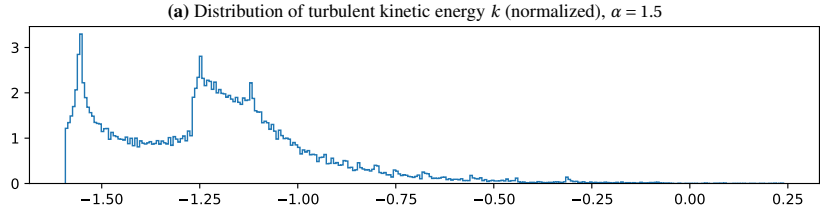
in which the coefficient $1/2$ is for scaling the value to the proper range. The transformation from linear to log scale will lead to a higher resolution for samples whose values are below 1 and a lower resolution for samples whose values are above 1.
The distribution of turbulence kinetic energy and dissipation rate after normalization is presented in Figure 4.6. After normalization, they distributed roughly in range $[0, 1]$ and $[-2, 0]$.

4.4. PointNet neural network

The proper network architecture for non-local turbulence modelling problems should be able to deal with a set of vectors. For networks with sets as inputs, the primary challenge is to achieve permutation invariance, that is, the model output should be invariant to the ordering of the elements in the input set. For this purpose, the simple pointNet architecture is introduced. It is not the main architecture used for the thesis. However, it can act as a comparison to the main architecture VCNN (later introduced in Section 4.5).
The idea of pointNet architecture is to first transform the set of input features into a single feature that is representative



(a) Distribution of turbulent kinetic energy k (normalized), $\alpha = 1.5$



(b) Distribution of dissipation rate ε (normalized), $\alpha = 1.5$

Figure 4.6: Distribution of the normalized input features

of the entire set. The transformation used must be permutation invariant. Simple examples of such operations are summation and taking the maximum value. With the permutation invariant features at hand, the rest is the same as normal neural networks. In the network in this thesis, the permutation invariant operation used is taking the average over the stencil. The consideration is that averaging might be more suitable for regression problems compared to taking the maximum.

A sketch of the network architecture is presented in Figure 4.7. The inputs of pointNet are the scalar features in $\mathscr{Q}$ (feature 5-11) rather than the full feature $\mathscr{Q}$ matrix. In this way, the neural network model is rotation invariant from the beginning.

The network consists of two parts. In the first part, the embedding network, each row of the input feature matrix $\mathscr{Q}$ is fed into the shared embedding MLP network, in which the 7 features go through a non-linear transformation and m encoding functions G (usually more than 7) are derived. Since the embedding network is shared by all the stencil points, each point in the stencil is dealing with individually and identically.

Before entering the fitting part, a crucial operation is performed. An average pooling is applied to the encoding functions among the stencil. Concretely, $D_i = \sum_{k=1}^{n} G_{ki}$. This step is where the information contained in each stencil point is integrated into a single feature of the stencil D_i. It bridges the local information of each sample and a non-local model that treats the stencil as a whole.

After average pooling, the matrix $\mathscr{D}$ has all the desired invariance properties. In the following fitting network, no special operation is needed. The fitting network is also an MLP. It takes in the stencil features $\mathscr{D}$ as inputs and provides predictions on the turbulence quantities k and ε.

The pointNet architecture used here is a simplified version of the original build as proposed in [37]. The simplification is made to the transformation carried out between networks for attaining a certain degree of rotating invariance properties. It is omitted because rotation invariance as defined in the object classification problem setup is different from that defined in turbulence modelling. The drawback of the simplified network architecture is that rotation invariance is not embedded into the architecture itself. The input features themselves need to be rotation invariant before entering the network.

The exact hyperparameters used are presented in Figure 4.8. The embedding network is composed of three fully connected layers. The number of nodes in each layer increases by a factor of two. Through the embedding network, features are expanded from 7 to 256. After the average pooling, the 256 non-local feature is fed into a two-layer MLP with 64 and 2 nodes respectively. The number of total learnable parameters in the network is 70914.

It is reported that the model performance is greatly influenced by the number of encoding functions m [37]. 256 encoding functions are sufficient for our purpose without causing overfitting issues.

4.5. Vector cloud neural network (VCNN)

The second neural network architecture applied is the VCNN. It is adapted from the network proposed in [12]. The methodology used for achieving permutation invariance is the same as the one used in pointNet. An extra challenge is posed on embedding rotation invariance on top of permutation invariance.

In this section, two invariance properties are first discussed respectively in Section 4.5.1 and Section 4.5.2. After

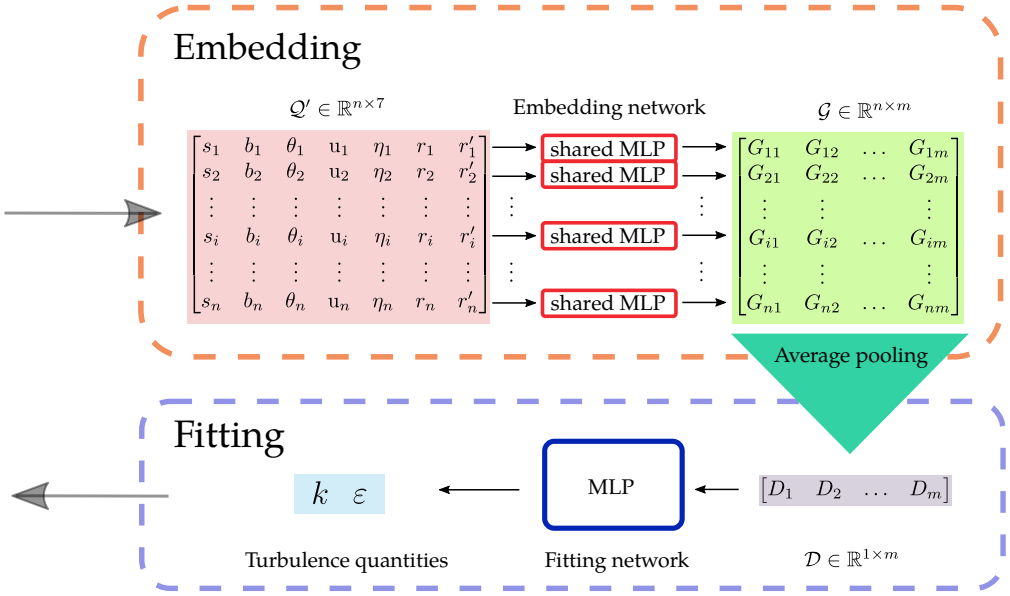


Figure 4.7: Schematics for the pointNet architecture.

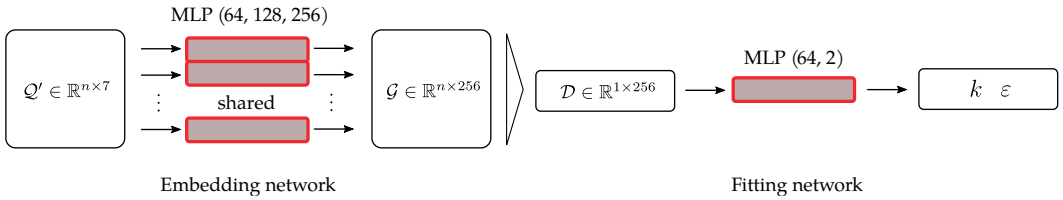


Figure 4.8: Hyperparameters for the pointNet neural network

The network parameters are shown in the format of a tuple. Each element in the tuple represents a neural network layer. The number represents the number of outputs of that layer. For instance, (64,2) for the fitting network indicates that there are two layers in the network and the first layer has 64 outputs, the second layer has 2 outputs.

that, the rationale based on which they are integrated into a complete architecture is presented in Section 4.5.3.

4.5.1. Rotation invariance

In the context of neural network models, rotation invariance means that the model output is invariant when the input features are expressed in a reference frame with arbitrary rotation:

$$\mathcal{M}(\mathcal{R}\mathcal{A}\mathcal{R}^{\top}) = \mathcal{M}(\mathcal{A}) \tag{4.7}$$

in which $\mathcal{M}$ is the neural network model, $\mathcal{A}$ is the input matrix (second order tensor), $\mathcal{R}$ is an arbitrary rotation matrix.

In our case it can be specified as: Rotation of the stencil reference frame will not influence the prediction of k and ε at the stencil center point. This property is required because scalar outputs such as k and ε themselves are invariant under change of orientation of the reference frame.

A simple way of achieving rotation invariance is to perform pairwise projection ($\mathcal{Q}\mathcal{Q}^{\top}$), in which the feature vector of every point in the stencil is projected onto all the other points. In other words, every point is represented by its correlation with others. The set of points in the stencil forms a basis on which they are expressed. The

coordinate of each point is then irrelevant to the rotation of the coordinate system.
The above procedure of pairwise projection can be expressed as the following matrix multiplication:

$$\mathscr{D} \in \mathbb{R}^{n\times n} = \mathscr{Q}\mathscr{Q}^{\top} \tag{4.8}$$

in which $\mathscr{Q} = \mathbb{R}^{n\times l}$ is the input feature matrix and n represents the size of the stencil, l represents the number of features. The proof of rotation invariance is straightforward. The input feature matrix under a rotation $\mathscr{P}$ follows:

$$\mathscr{Q}_r = \mathscr{Q}\mathscr{P} \tag{4.9}$$

In two dimensional cases, $\mathscr{P} \in \mathbb{R}^{2\times 2}$. The pairwise projection of the matrix after rotation is then:

$$\mathscr{D}_r = \mathscr{Q}\mathscr{P}\mathscr{P}^{\top}\mathscr{Q}^{\top} \tag{4.10}$$

For rotation matrices, $\mathscr{P}^{\top} = \mathscr{P}^{-1}$. It follows directly that:

$$\mathscr{D}_r = \mathscr{Q}\mathscr{Q}^{\top} = \mathscr{D} \tag{4.11}$$

It is worth noting that the pairwise projection is not valid if the original data set has experienced any non-linear transformation. Since in that case, the associative law of matrix multiplication is not necessarily valid. Usually,

$$g(\mathscr{Q}\mathscr{P}) \neq g(\mathscr{Q})\mathscr{P} \tag{4.12}$$

in which g is a general non-linear function. This is important in the combination of rotation invariance and permutation invariance.

4.5.2. Permutational invariance

The idea for achieving permutation invariance in VCNN is the same as in pointNet. The information contained in each data point separately needs to be transformed to some kind of collective feature of the entire set. Furthermore, the collective features have no relevance to the ordering of the elements. In this way, they can be processed normally as in any other point-based neural network, for instance, by Multilayer Perceptron (MLP) network.
Concretely, according to Kolmogorov-Arnold representation theorem [95], any function f that is permutational invariant with respect to the element in its input sets $\{\boldsymbol{z_i}\}_{i=1}^{n}$, can be represented in the following form:

$$f(\{\boldsymbol{z_i}\}_{i=1}^{n}) = \Phi(\frac{1}{n}\sum_{n=1}^{n}\phi(\boldsymbol{z}_i)) \tag{4.13}$$

in which ϕ is an arbitrary multidimensional function and Φ is an arbitrary function (the number of input variables is dependent on the output of function ϕ). A key point is that, each element must be treated identically (by function ϕ) before the summation. After summation the operations are applied to the collective feature $\frac{1}{n}\sum_{n=1}^{n}\phi(\boldsymbol{z}_i)$.

4.5.3. Integration of the invariance properties

As explained in Section 4.3, all the input features are translation and Galilean invariant. It is automatically guaranteed that the model is also translation and Galilean invariance without extra efforts on the architecture.
For achieving rotation and permutation invariance, one can find a possible conflict between the two. Using pairwise projection for rotation invariance requires that the original data set can not be transformed non-linearly beforehand. The framework of achieving permutational invariance on the other hand relies on the expressive power of nonlinear transformation in the embedding network. In other words, the non-linear transformation must be done before the summation.
One way to resolve the conflict is to deliver the information obtained in the embedding network through weights in the summation. Specifically, function ϕ in (4.13) takes the following form:

$$\phi(\boldsymbol{z}_i) = G_{ij}\boldsymbol{z}_i \tag{4.14}$$

in which the weights G_{ij} (also referred to as the encoding functions) are the outputs of the embedding network. They are generated through non-local transformation (as described by the embedding network) of the input data, whilst the data itself remains unchanged. Moreover, multiple sets of weights can be used to better preserve the information contained in the input data (referred to by index j).

The remaining issue is then if the weights stem from the original input data without pairwise projection, then they are not rotation invariant. This problem can be resolved by using only the rotation invariant scalar features as the input of the embedding network. The first four coordinate relevant features corresponding to the velocity and coordinate vectors are thus discarded.

The schematics of the framework based on the above analysis is depicted in Figure 4.9. The neural network takes the input feature matrix $\mathcal{Q}$ which contains all the feature vectors in a stencil as the input and gives the prediction on turbulence quantities k and ε. In the first embedding stage, the scalar features from each row of the input feature matrix are fed into the embedding network, in which they are transformed by the embedding network into rotation invariant encoding functions in $\mathcal{G}$ and its submatrix $\mathcal{G}^*$. In this stage, all the elements in the stencil are treated identically and individually. Before entering the second fitting stage, a pairwise projection and a weighted summation are performed simultaneously by operation $\mathcal{D} = \frac{1}{n^2}\mathcal{G}^\top \mathcal{Q}\mathcal{Q}^\top \mathcal{G}^*$. In this way, information from each stencil point is integrated permutation invariantly into nonlocal features in $\mathcal{L}$ and $\mathcal{L}^*$. In the second fitting stage, $\mathcal{D}$ is flattened into a vector and fed into the fitting network. The output of the fitting network will be the prediction on turbulence quantities k and ε.

The hyperparameters of the network are presented in Figure 4.10. There are three layers in both the embedding network and the fitting network. The total number of tunable parameters is 47810.

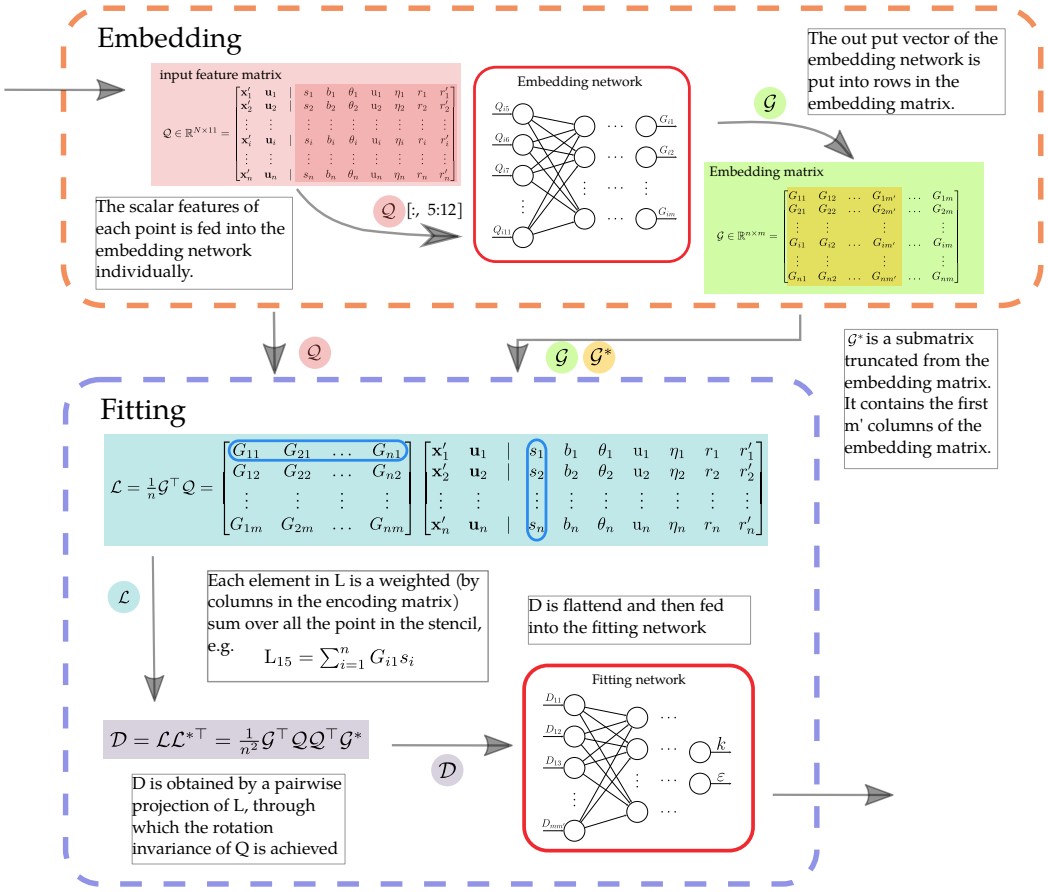


Figure 4.9: Schematics for the vector cloud neural network

4.5.4. Proof of invariance properties with minimal example

In this section, the invariance properties of the neural network is proved by a minimal example. Considering the simple case with $n = 4$ (four points in the cloud), $l = 3$ (three input features, $\mathbf{q} = [x, y, r]^\top$) and $m = m' = 2$ (two

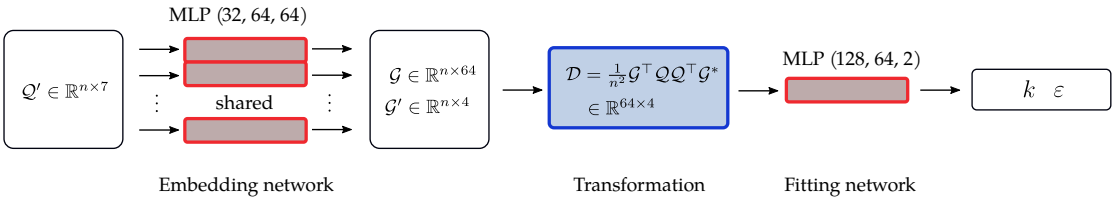


Figure 4.10: Hyperparameters for the VCNN neural network

encoding functions from embedding layers). The input matrix is as follows:

$$\mathcal{Q} \in \mathbb{R}^{4\times 3} = \begin{bmatrix} x_1 & y_1 & r_1 \\ x_2 & y_2 & r_2 \\ x_3 & y_3 & r_3 \\ x_4 & y_4 & r_4 \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1^\top \\ \mathbf{q}_2^\top \\ \mathbf{q}_3^\top \\ \mathbf{q}_4^\top \end{bmatrix} \tag{4.15}$$

The scalar feature r of each point in the cloud is fed into the embedding network. They are processed individually and identically. The outputs of embedding network are organized in an matrix (the embedding matrix) such that each row corresponds to a point in the cloud, and each column corresponds to a encoding function. The shape of the embedding matrix is $n \times m$. With $n = 4$ and $m = 2$, the outputs of the embedding network is then:

$$\mathcal{G} \in \mathbb{R}^{4\times 2} = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \\ G_{31} & G_{32} \\ G_{41} & G_{42} \end{bmatrix} = \begin{bmatrix} \mathbf{g}_1^\top \\ \mathbf{g}_2^\top \\ \mathbf{g}_3^\top \\ \mathbf{g}_4^\top \end{bmatrix} \tag{4.16}$$

in which $\mathbf{g}_i = f(r_i)$
The input matrix of the fitting layers is computed as:

$$\mathcal{D} \in \mathbb{R}^{2\times 2} = \mathcal{G}^\top \mathcal{Q}\mathcal{Q}^\top \mathcal{G} \tag{4.17}$$

Here we introduce notation $\mathcal{L} = \mathcal{G}^\top \mathcal{Q} \in \mathbb{R}^{2\times 3}$ for convenience, then:

$$\mathcal{D} = \mathcal{G}^\top \mathcal{Q}\mathcal{Q}^\top \mathcal{G} = \mathcal{L}\mathcal{L}^\top \tag{4.18}$$

Substituting Equation 4.15 and Equation 4.16 into Equation 4.17:

$$\mathcal{D} \in \mathbb{R}^{2\times 2} = \mathcal{G}^\top \mathcal{Q}\mathcal{Q}^\top \mathcal{G} \tag{4.19}$$

$$= \begin{bmatrix} \mathbf{g}_1 & \mathbf{g}_2 & \mathbf{g}_3 & \mathbf{g}_4 \end{bmatrix} \begin{bmatrix} \mathbf{q}_1^\top \\ \mathbf{q}_2^\top \\ \mathbf{q}_3^\top \\ \mathbf{q}_4^\top \end{bmatrix} \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 & \mathbf{q}_4 \end{bmatrix} \begin{bmatrix} \mathbf{g}_1^\top \\ \mathbf{g}_2^\top \\ \mathbf{g}_3^\top \\ \mathbf{g}_4^\top \end{bmatrix} \tag{4.20}$$

$$= (\mathbf{g}_1\mathbf{q}_1^\top + \mathbf{g}_2\mathbf{q}_2^\top + \mathbf{g}_3\mathbf{q}_3^\top + \mathbf{g}_4\mathbf{q}_4^\top)(\mathbf{q}_1\mathbf{g}_1^\top + \mathbf{q}_2\mathbf{g}_2^\top + \mathbf{q}_3\mathbf{g}_3^\top + \mathbf{q}_4\mathbf{g}_4^\top) \tag{4.21}$$

$$= \left(\sum_{i=1}^{4} \mathbf{g}_i\mathbf{q}_i^\top\right)\left(\sum_{i'=1}^{4} (\mathbf{g}_{i'}\mathbf{q}_{i'}^\top)^\top\right) \tag{4.22}$$

Let $\mathbf{g}_i\mathbf{q}_i^\top = L_i$ and $(\mathbf{g}_i\mathbf{q}_i^\top)^\top = L_{i'}$, it follows that:

$$\mathcal{L} = \sum_{i=1}^{4} L_i = \sum_{i=1}^{4} \mathbf{g}_i\mathbf{q}_i^\top \tag{4.23}$$

$$\mathcal{L}^\top = \sum_{i'=1}^{4} L_{i'} = \sum_{i'=1}^{4} (\mathbf{g}_i\mathbf{q}_i^\top)^\top \tag{4.24}$$

Expanding Equation 4.22 using notation L_i and $L_{i'}$:

$$\mathscr{D} = \sum_{i=1}^{4} \sum_{i'=1}^{4} L_i L_{i'} \tag{4.25}$$

$\mathscr{D}$ is obtained through a double summation over all the points in the cloud, thus it is permutation invariant with regard to input data.
The result can be expanded directly, for instance:

$$\begin{aligned} L_i = \mathbf{g}_1 \mathbf{q}_1^\top &= \begin{bmatrix} G_{11} \\ G_{12} \end{bmatrix} \begin{bmatrix} x_1 & y_1 & r_1 \end{bmatrix} \\ &= \begin{bmatrix} G_{11}x_1 & G_{11}y_1 & G_{11}r_1 \\ G_{12}x_1 & G_{12}y_1 & G_{12}r_1 \end{bmatrix} \end{aligned}$$

As discussed before, the translational and Galilean invariance are straightforward since only relative coordinates and relative velocity are selected as input features.
The rotational invariance is achieved through pairwise projection $\mathscr{Q}\mathscr{Q}^\top$. Presume that the input features are transformed through a rotation $\mathscr{R} \in \mathbb{R}^{2\times 2}$. For convenience, the input feature matrix can be rewritten as follows:

$$\mathscr{Q} \in \mathbb{R}^{4\times 3} = \begin{bmatrix} x_1 & y_1 & r_1 \\ x_2 & y_2 & r_2 \\ x_3 & y_3 & r_3 \\ x_4 & y_4 & r_4 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\top & r_1 \\ \mathbf{x}_2^\top & r_2 \\ \mathbf{x}_3^\top & r_3 \\ \mathbf{x}_4^\top & r_4 \end{bmatrix}$$

Notice that vector $\mathbf{x}_i^\top$ are row vectors, so the rotation matrix then should be applied by post-multiplication. The transformed input matrix ($\mathscr{Q}_r$) is computed as:

$$\mathscr{Q}_r \in \mathbb{R}^{4\times 3} = \begin{bmatrix} \mathbf{x}_1^\top \mathscr{R} & r_1 \\ \mathbf{x}_2^\top \mathscr{R} & r_2 \\ \mathbf{x}_3^\top \mathscr{R} & r_3 \\ \mathbf{x}_4^\top \mathscr{R} & r_4 \end{bmatrix}$$

Since $\mathscr{G}$ is a nonlinear transformation of only the scalar features, it remains the same after rotation.

$$\mathscr{D}_r \in \mathbb{R}^{2\times 2} = \mathscr{G}^\top \mathscr{Q}_r \mathscr{Q}_r^\top \mathscr{G}$$

The proof of rotational invariance is equivalent to proving $\mathscr{Q}_r \mathscr{Q}_r^\top = \mathscr{Q}\mathscr{Q}^\top$. The derivation is presented below:

$$\mathscr{Q}_r \mathscr{Q}_r^\top = \begin{bmatrix} \mathbf{x}_1^\top \mathscr{R} & r_1 \\ \mathbf{x}_2^\top \mathscr{R} & r_2 \\ \mathbf{x}_3^\top \mathscr{R} & r_3 \\ \mathbf{x}_4^\top \mathscr{R} & r_4 \end{bmatrix} \begin{bmatrix} \mathscr{R}^\top \mathbf{x}_1 & \mathscr{R}^\top \mathbf{x}_2 & \mathscr{R}^\top \mathbf{x}_3 & \mathscr{R}^\top \mathbf{x}_4 \\ r_1 & r_2 & r_3 & r_4 \end{bmatrix} \tag{4.26}$$

$$= \begin{bmatrix} \mathbf{x}_1^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_1 + r_1^2 & \mathbf{x}_1^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_2 + r_1 r_2 & \mathbf{x}_1^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_3 + r_1 r_3 & \mathbf{x}_1^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_4 + r_1 r_4 \\ \mathbf{x}_2^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_1 + r_2 r_1 & \mathbf{x}_2^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_2 + r_2^2 & \mathbf{x}_2^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_3 + r_2 r_3 & \mathbf{x}_2^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_4 + r_2 r_4 \\ \mathbf{x}_3^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_1 + r_3 r_1 & \mathbf{x}_3^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_2 + r_3 r_2 & \mathbf{x}_3^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_3 + r_3^2 & \mathbf{x}_3^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_4 + r_3 r_4 \\ \mathbf{x}_4^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_1 + r_1 r_4 & \mathbf{x}_4^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_2 + r_2 r_4 & \mathbf{x}_4^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_3 + r_4 r_3 & \mathbf{x}_4^\top \mathscr{R}\mathscr{R}^\top \mathbf{x}_4 + r_4^2 \end{bmatrix} \tag{4.27}$$

$$= \begin{bmatrix} \mathbf{x}_1^\top \mathbf{x}_1 + r_1^2 & \mathbf{x}_1^\top \mathbf{x}_2 + r_1 r_2 & \mathbf{x}_1^\top \mathbf{x}_3 + r_1 r_3 & \mathbf{x}_1^\top \mathbf{x}_4 + r_1 r_4 \\ \mathbf{x}_2^\top \mathbf{x}_1 + r_2 r_1 & \mathbf{x}_2^\top \mathbf{x}_2 + r_2^2 & \mathbf{x}_2^\top \mathbf{x}_3 + r_2 r_3 & \mathbf{x}_2^\top \mathbf{x}_4 + r_2 r_4 \\ \mathbf{x}_3^\top \mathbf{x}_1 + r_3 r_1 & \mathbf{x}_3^\top \mathbf{x}_2 + r_3 r_2 & \mathbf{x}_3^\top \mathbf{x}_3 + r_3^2 & \mathbf{x}_3^\top \mathbf{x}_4 + r_3 r_4 \\ \mathbf{x}_4^\top \mathbf{x}_1 + r_4 r_1 & \mathbf{x}_4^\top \mathbf{x}_2 + r_4 r_2 & \mathbf{x}_4^\top \mathbf{x}_3 + r_4 r_3 & \mathbf{x}_4^\top \mathbf{x}_4 + r_4^2 \end{bmatrix} \tag{4.28}$$

$$= \mathscr{Q}\mathscr{Q}^\top \tag{4.29}$$

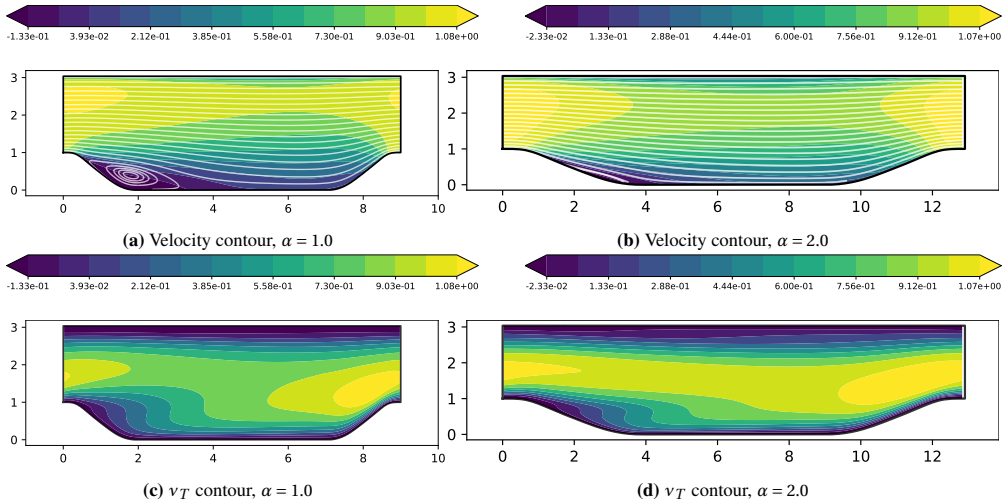
4.6. Neural network training

In this section, the methodology related to neural network training is presented. This section is applicable for both pointNet and VCNN architecture. In Section 4.6.1, how the raw data from the simulation is preprocessed is introduced. The training and testing data set used are described. Section 4.6.2 shows the loss function for training and the regularization of the network. Adam algorithm [96] is used as the optimizer with a learning rate of 10^{-3}.

4.6.1. Dataset

The raw data for neural network training is generated in OpenFOAM RANS simulation following the setup discussed in Section 4.1. The training set consists of 11 flow cases, $\alpha = 1.0, 1.1, 1.2, ..., 2.0$. Two representative flow cases are shown in Figure 4.11.

The upper two figures Figure 4.11a and Figure 4.11b present the mean velocity contour of flow case $\alpha = 1.0$ and



(a) Velocity contour, $\alpha = 1.0$ **(b)** Velocity contour, $\alpha = 2.0$

(c) ν_T contour, $\alpha = 1.0$ **(d)** ν_T contour, $\alpha = 2.0$

Figure 4.11: Representative simulation results.

$\alpha = 2.0$. The recirculation zone at the back of the hills is indicated by the streamline. In flow case $\alpha = 1.0$, the recirculation zone can be identified easily. Whilst, it is barely seen in flow case $\alpha = 2.0$. The lower two figures Figure 4.11c and Figure 4.11d show the eddy viscosity contour of the respective flow cases. Flow case $\alpha = 1.0$ is characterized by a higher level of eddy viscosity. The distribution pattern of both cases is quite similar.

For obtaining the input feature matrix, preprocessing the raw data are required. The steps are listed below following the sequence of operation:

1. Flow field data (mean velocity and strain rate magnitude) and geometry data (cell centres, cell volumes, boundary indicators and wall distance) are read from the OpenFOAM time directory files.
2. Periodic boundary condition is applied to the extracted data such that the influence regions of cells near the inlet and outlet are complete.
3. Wall distance function is computed based on the wall distance of each cell according to the expression given in Table 4.1. s, k and ε are normalized as indicated in Section 4.3.2.
4. The cell points in a single flow case are iterated over. The influence region is computed as in Section 4.2. The cell points inside the influence region are gathered to form the stencil of the centre point. If the stencil size (200 for training) is larger than the number of points in the influence region, random points are repeated. If the stencil size is smaller than the points inside, random points are removed. In this way, the input matrix for each point has the same dimensions.
5. Relative coordinates, relative velocities, r, r', velocity magnitude are computed for each centre point. Cell volume is normalized by the mean cell volume in the stencil.
6. Last two steps are repeated for each flow case. After the training data from each flow case is generated, they are stacked together to get the complete training set.

Feature scaling is performed in two steps. The first time is before the stencils are formed. The targets are s, k and ε. The second time is after the stencils are formed and it is carried out in stencils. It is performed on the cell volume.

The training dataset contains every cell point in 11 flow (in total $(40000 + 48000) \times 11/2 = 484000$ points). The number of samples taken in stencils for training is 200. The shape of the training dataset is then $484000 \times 200 \times 11$.

For testing the generalization ability of the network, flow cases $a = 0.9, 0.8, \ldots, 0.5, 2.1, 2.2, \ldots, 2.5$ are chosen as the extrapolation test set; flow case $a = 1.05, 1.15, \ldots, 1.85, 1.95$ are chosen as the interpolation test set. The processing of testing data is almost identical to that of the training data. The only difference is that there is no random sampling in the stencils. All the points are used as input for better performance.

The flow cases used for training and testing are summarized in Table 4.2 and visualized in Figure 4.12 below.

Dataset	slope parameters α	number of cases
Training set	1.0, 1.1, 1.2, 1.3, ..., 1.8, 1.9, 2.0	11
Interpolation testing set	1.05, 1.15, . . . , 1.85, 1.95	11
Extrapolation testing set	0.9, 0.7, 0.5, 2.1, 2.3, 2.5	10

Table 4.2: Datasets for training and testing

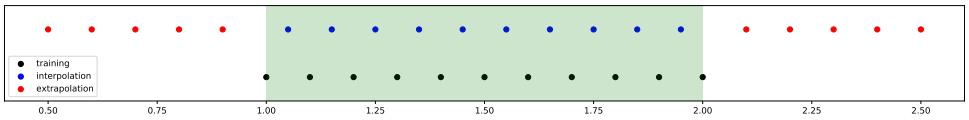


Figure 4.12: Datasets for training and testing

4.6.2. Loss function and regularization

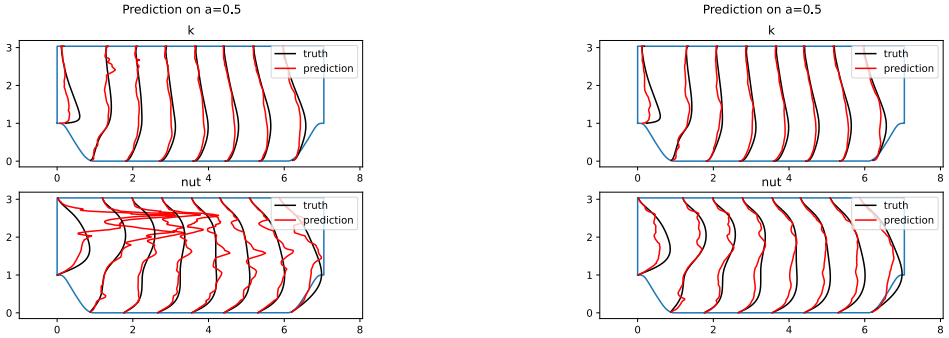
The sum of squared error is used as the loss function for the neural network training as shown below:

$$R(\theta) = \sum_{i=1}^{N} (\tilde{k}_i^{(n)} - k_i^{(n)})^2 + \sum_{j=1}^{N} (\tilde{\varepsilon}_j^{(n)} - \varepsilon_j^{(n)})^2 + \lambda \sum_{k=1}^{N} (\tilde{\nu}_{T_k} - \hat{\nu}_{T_k})^2 \tag{4.30}$$

in which $\tilde{}$ represents the predicted value, superscript (n) indicates the normalized value. λ takes the value of 1.
As discussed in Section 3.3.6, regularization is performed to avoid overfitting problem, in which an extra L2 penalty term is added to the loss function as follows:

$$R(\theta) = \sum_{i=1}^{N} (\tilde{k}_i^{(n)} - k_i^{(n)})^2 + \sum_{j=1}^{N} (\tilde{\varepsilon}_j^{(n)} - \varepsilon_j^{(n)})^2 + \lambda \sum_{k=1}^{N} (\tilde{\nu}_{T_k} - \hat{\nu}_{T_k})^2 + \lambda_w \sum_{j=1} w_j^2 \tag{4.31}$$

By using the regularized loss function Equation 4.31, the magnitude of the weights and biases is kept to a lower level throughout the training process. A comparison between neural networks performance with and without regularization in an uncoupled setting is presented in Figure 4.13.
The $L1$ norm of the unregularized model is 12012 and that of the regularized model is 1444. It is evident that the



(a) k and ν_T profiles (unregularized)

(b) k and ν_T profiles (regularized)

Figure 4.13: Comparison between regularized and unregularized neural networks on flow case $a = 0.5$

regularized network is more robust to model extrapolation and produce less oscillation in the prediction. Whilst the unregularized model is very sensitive to the change of input distribution.
A comparison of the two models' response surface to the baseline $k-\varepsilon$ model is shown in Figure 4.14. The final output of the neural network ν_T is plotted against the stencil averaged velocity in x direction. It is a 2D projection of the multi-variant function of the neural network.

It can be observed that the output of the unregularized model for high velocity is oscillating rapidly, which corresponds to the oscillatory ν_T profile in Figure 4.13a. The regularized model also deviate from the baseline model, but the oscillation is much more moderate, which means better robustness. The improvement in robustness

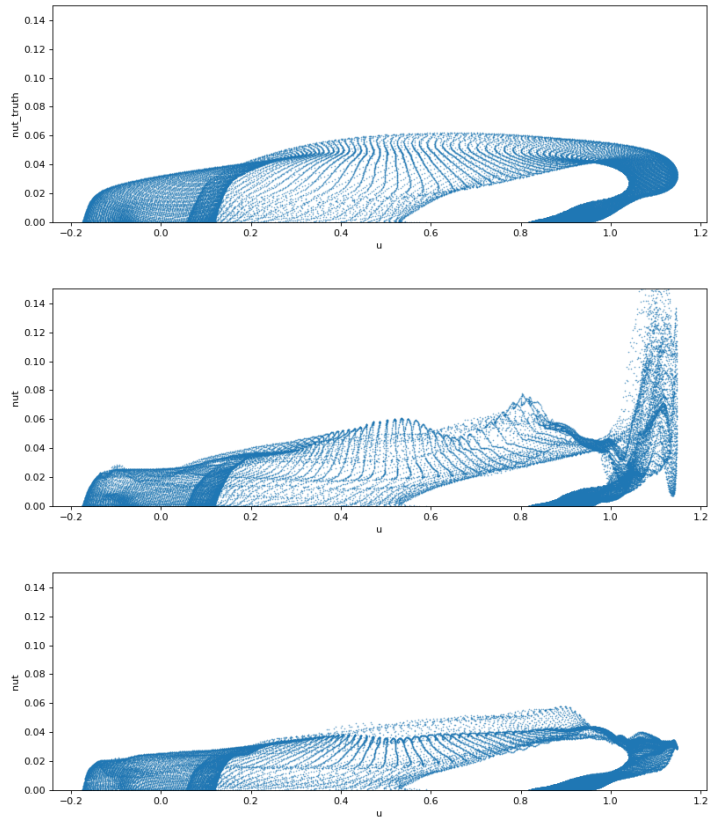


Figure 4.14: Comparison between regularized and unregularized neural networks on flow case $a = 0.5$

of the neural network model is also presented in a coupled setup. When using the unregularized neural network, the coupled solver is hard to converge. A detailed discussion of the model performance in both uncoupled and coupled setup is presented in Chapter 5.

4.7. Coupling of neural network with simpleFoam solver

In the thesis work, the data-driven model is tested in a coupled setting. The trained neural network functions as a normal turbulence model. It computes and transfers the turbulent quantities to a RANS solver.
The data transfer between two components and the entire workflow is depicted in Figure 4.15.
The steps of the coupled solver are described below following the order of operation:

1. The mean flow field and the turbulence quantities are initialized, same as in a normal numerical simulation.
2. SimpleFOAM solver reads the output from the last time step and solves for the mean flow quantities, e.g. pressure and flow field. Quantities required by the neural network such as the strain rate magnitude is also calculated. All the simulation data is written to files in OpenFOAM format.
3. The simulation data is read and processed as described in Table 4.2. The features are computed and organized into an input matrix.
4. The input matrix is fed into the well-trained neural network turbulence model. The turbulence quantities are obtained and written into OpenFOAM files.
5. Steps 2-4 are repeated until the simulation reaches the specified number of iteration steps.

4.8. Software and Libraries

The software package used for all the CFD simulations in this work is OpenFOAM v2012.
Loading data of OpenFOAM into python NumPy array uses package foam utility (see repository: `https://github.com/cmichelenstrofer/DAFI.git`). The preprocessing of data utilized pandas and NumPy library. The construction and training of the neural network used the python machine learning library PyTorch [97].

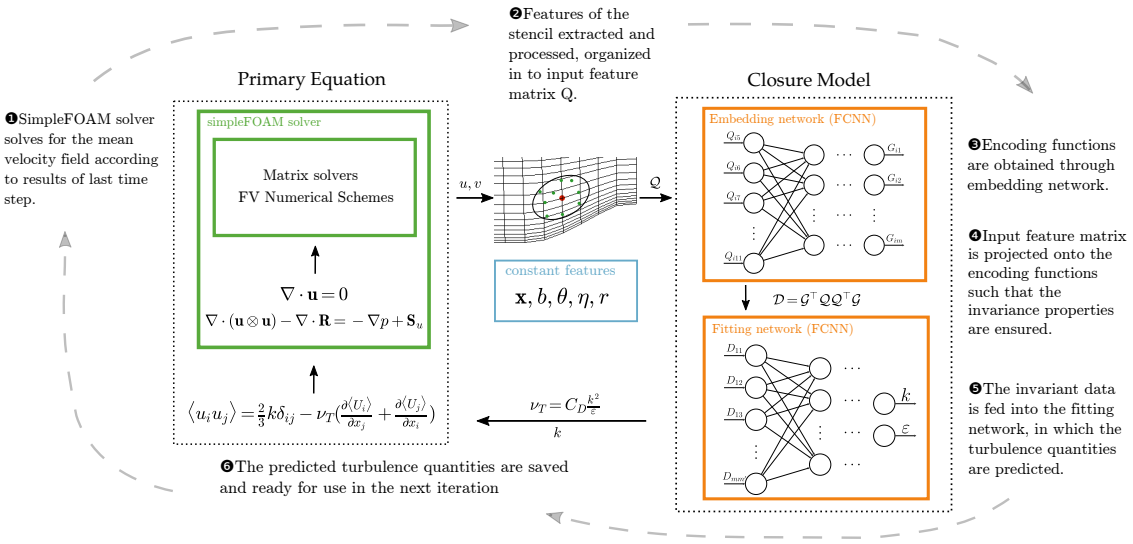


Figure 4.15: Schematics of the coupled solver.

OpenFOAM solver written in C++ and neural network written in python are coupled by Python/C API [98]. The main code of the work can be found in GitHub repository `https://github.com/brossard1931/FIVCNN_coupling.git`.

5

Results and Discussion

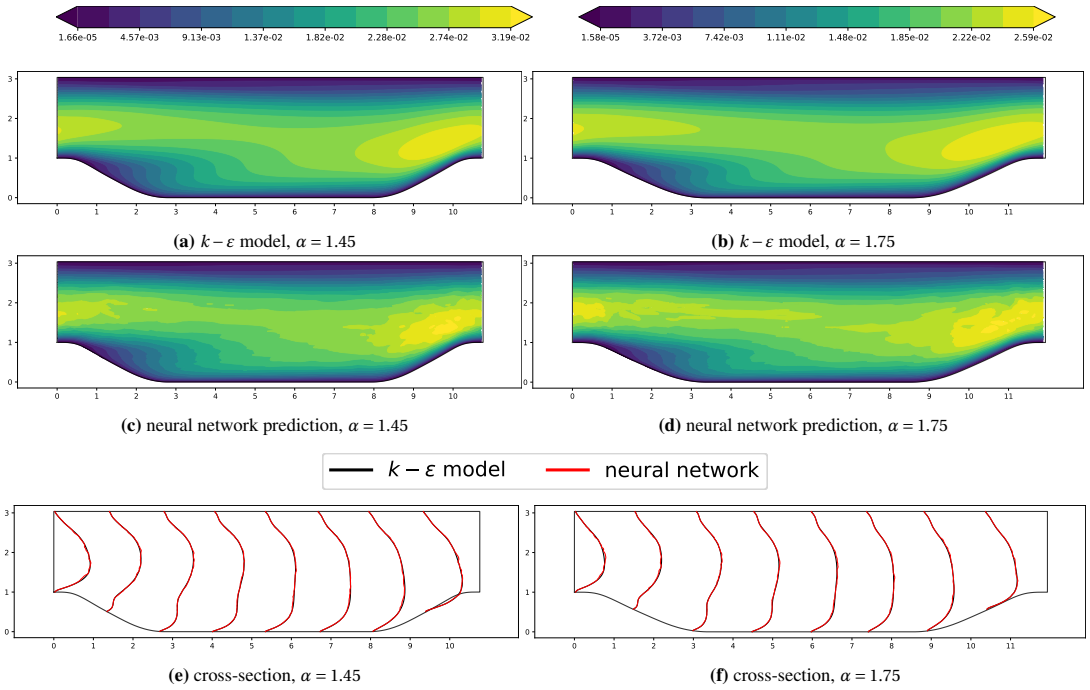
5.1. Performance of VCNN in uncoupled and coupled setups

The performance of the neural network-based turbulence model is evaluated in both uncoupled and coupled setups. In the uncoupled setup, the converged mean flow field of RANS with $k-\varepsilon$ turbulence model is used as the neural network input. The neural network prediction is compared with that of the baseline standard $k-\varepsilon$ turbulence model. In the coupled setup, simpleFOAM solver is coupled with the neural network turbulence model. The coupled solver starts iteration from a given initial condition. During the simulation, the coupled solver is continuously confronted with unconverged mean flow fields. Thus, compared to the coupled setup, the coupled setup poses a larger challenge to the neural network-based turbulence model.
The same neural network model is used for testing in coupled and uncoupled setups. The training dataset is presented in Table 4.2 and Figure 4.12.

5.1.1. Uncoupled setup

A thorough investigation into the interpolation and extrapolation capacity of the obtained network is first performed in the uncoupled setup. The neural network prediction of ν_T on two interpolation cases with $a = 1.45$ and $a = 1.75$ are visualized in Figure 5.1. In an uncoupled setup, the information is one-way transformed from the mean velocity field to the neural network turbulence model. Thus there is no influence of the neural network prediction on the mean velocity field and the contours and profiles are only for the prediction on eddy viscosity ν_T.
It is shown that the prediction of the neural network is highly consistent with the results of $k-\varepsilon$ model. The contour plots of the predictions Figure 5.8c and Figure 5.8d have very similar patterns to that of the baseline $k-\varepsilon$ model. There are only small oscillations observed in neural network prediction that differ from the $k-\varepsilon$ model prediction. The neural network prediction is inferior to the $k-\varepsilon$ model in its smoothness. Considering the complexity of the neural network and a lack of hard-coded diffusion terms in it, such a level of noise is acceptable. From the profile plots, the difference between the neural network prediction and the $k-\varepsilon$ model simulation results is barely discernible.

For machine learning models, extrapolation is usually more challenging than interpolation. A first test is performed on flow case $\alpha = 0.9$ and $\alpha = 2.1$. A comparison between ν_T predictions of the neural network model and baseline $k-\varepsilon$ model on these two flow cases are presented in Figure 5.2. When comparing Figure 5.2c, Figure 5.2d and Figure 5.2a, Figure 5.2b, it is found that although the neural network prediction looks more chaotic, the overall pattern is close. The regions of high and low eddy viscosity are well presented in the network prediction contour. In the cross-section plot, the network prediction overlaps the baseline model prediction. The noise is the largest near the inlet and is small in the rest of the flow domain. The interpolation and extrapolation tests discussed above prove the generalization ability of the neural network model. In order to test the generalization limit of the network, the neural network model is further challenged by the highly extrapolated flow cases $a = 0.7$ and $a = 2.3$. The predictions of the neural network model are compared with the baseline case in Figure 5.3. It can be observed that to such extend of extrapolation, although predictions of the neural network share certain similarities with the baseline, some unphysical patterns emerge. For instance, in Figure 5.9c, a region near the top of the inlet has a very large eddy viscosity. Whilst in the baseline case in Figure 5.9a, such a pattern doesn't exist.



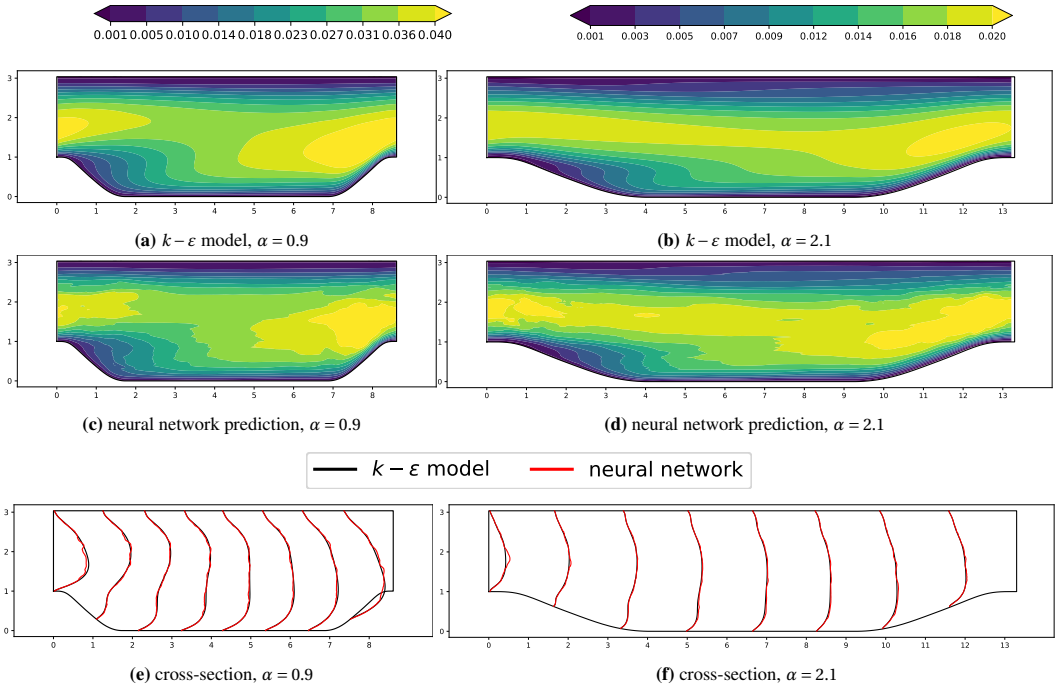
(a) $k-\varepsilon$ model, $\alpha = 1.45$ **(b)** $k-\varepsilon$ model, $\alpha = 1.75$

(c) neural network prediction, $\alpha = 1.45$ **(d)** neural network prediction, $\alpha = 1.75$

(e) cross-section, $\alpha = 1.45$ **(f)** cross-section, $\alpha = 1.75$

Figure 5.1: Comparison of the contour and the cross-section profiles of turbulence viscosity ν_T for interpolation flow cases $\alpha = 1.45$ and $\alpha = 1.75$ (VCNN)

Since the scale of eddy viscosity is small compared to the length scale, in the cross-section plots, eddy viscosity is scaled by a factor of 30 such that the curve can be seen more clearly.



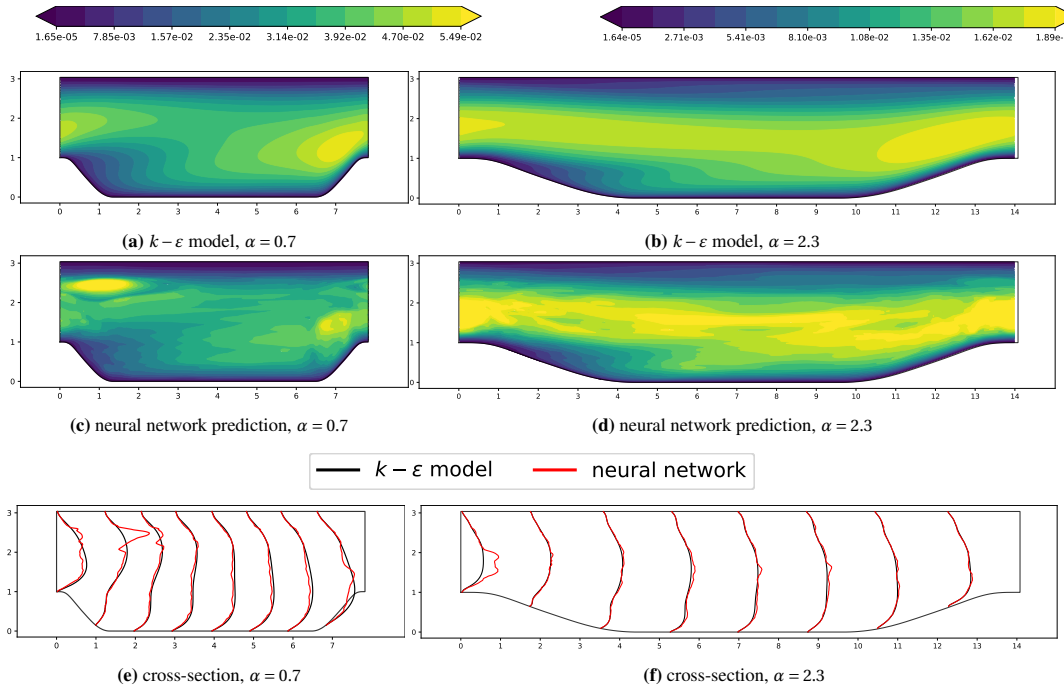
(a) $k-\varepsilon$ model, $\alpha = 0.9$ **(b)** $k-\varepsilon$ model, $\alpha = 2.1$

(c) neural network prediction, $\alpha = 0.9$ **(d)** neural network prediction, $\alpha = 2.1$

(e) cross-section, $\alpha = 0.9$ **(f)** cross-section, $\alpha = 2.1$

Figure 5.2: Comparison of the contour and the cross-section profiles of turbulence viscosity ν_T for flow cases $\alpha = 0.9$ and $\alpha = 2.1$ (VCNN)

The overall level of ν_T is also lower than that of the baseline prediction. In flow case $\alpha = 2.3$, the region near the bottom of the inlet has a higher ν_T prediction, and the predicted ν_T is generally higher than that of the baseline. By inspecting the contour plots and the cross-section profiles, a quantitative evaluation of the model performance



(a) $k-\varepsilon$ model, $\alpha = 0.7$ **(b)** $k-\varepsilon$ model, $\alpha = 2.3$

(c) neural network prediction, $\alpha = 0.7$ **(d)** neural network prediction, $\alpha = 2.3$

(e) cross-section, $\alpha = 0.7$ **(f)** cross-section, $\alpha = 2.3$

Figure 5.3: Comparison of the contour and the cross-section profiles of turbulence viscosity ν_T for $\alpha = 0.7$ and $\alpha = 2.3$ (VCNN)

is obtained. For a quantitative evaluation of the model performance, the normalized prediction error is introduced:

$$\text{error} = \frac{\sqrt{\sum_{i=1}^{N} |\nu_{T_i} - \hat{\nu}_{T_i}|^2}}{\sqrt{\sum_{i=1}^{N} |\hat{\nu}_{T_i}|^2}} \tag{5.1}$$

in which ν_T is the neural network prediction, and $\hat{\nu}_T$ refers to the results of $k-\varepsilon$ model (the baseline case). The total error of all mesh points is normalized by the sum of squared baseline results, avoiding possible small values in the denominator.

The prediction error of all the interpolation and extrapolation testing sets (as listed in Table 4.2) is summarized in Figure 5.4.

When the slope parameter is covered by the training set (interpolation), the validation error remains at a low

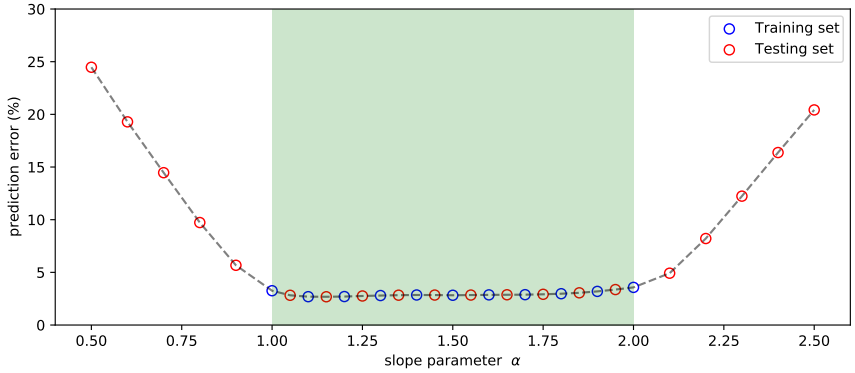


Figure 5.4: Prediction error computed by Equation 5.1 over different slope parameters.

Light green background indicates the range of slope parameter covered by the testing set (marked with blue circles). Red circles represent the testing flow cases, including interpolation and extrapolation.

level (around 2%), which is close to the training error. When the model is extrapolated, the error rate increases drastically. When it is extrapolated by 0.1 slope parameter to 0.9 and 2.1, the validation error is around 5%. Whilst for flow cases $\alpha = 0.5$ and $\alpha = 2.5$, the validation error is over 20%. The error increases by roughly 5% for

every increasing or increasing 0.1 of the slope parameter. Besides, it is noticed that the model performs slightly better when extrapolation to larger slope parameters compared to lower slope parameters. This indicates a larger difficulty in predicting flow with higher turbulence intensity.
In terms of the notoriously bad extrapolation ability of neural network models, an often-used strategy for enhancing model performance is augmenting the dataset with more training cases. For instance, if a higher prediction capacity on lower slope parameters is desired, an extra data point at $\alpha = 0.7$ can be added to the current dataset. In this way, the neural network can adapt to cases with low slope parameters during training. A test on the effectiveness of extending the dataset is performed, in which the training set is composed of flow cases $\alpha = 0.5, 0.7, \ldots, 2.3, 2.5$ (in total 11 cases). The validation error of the model in the range $[0.5, 2.5]$ is shown in Figure 5.5. The error rate

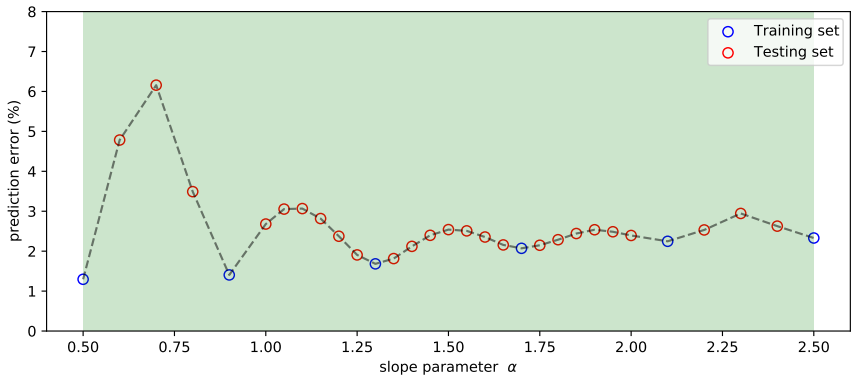


Figure 5.5: Prediction error as computed by Equation 5.1 over different slope parameters (augmented dataset).

of the model trained on the new dataset shows a wavy pattern. For cases included in the training set, the error rate is small. When getting far from the training cases, the error increases until approaching another training case. It reaches the largest right in the middle between two training cases. The validation error plot agrees with our expectation of higher error for cases that deviate from training cases. Nonetheless, the prediction capacity of the network on the cases of interest can be improved evidently by adding proper data points to the training set. The maximum validation error appears at flow case $\alpha = 0.7$ with an error rate of around 6.5%. It is much less than the largest error rate 25% of the last model.
It is worth noting that similar to previous tests, the largest error appears at the low slope parameters region. This might be because the flow field changes a lot in this range and knowing flow case 0.5 and 0.9 helps less in predicting flow case 0.7.

5.1.2. Coupled setup

In the coupled setup, the neural network model performance is surprisingly good and comparable to that in the uncoupled setup. Tests are performed on an initial condition taken from the $k-\varepsilon$ simulation at time step 500. Its location in the entire convergence history of a simulation is shown in Figure 5.6. At iteration 500, the flow field is far from convergence. Whilst it provides a reasonable initial condition for the neural network prediction.
The comparison between results of the neural network coupled solver and PDE solver on one interpolation and

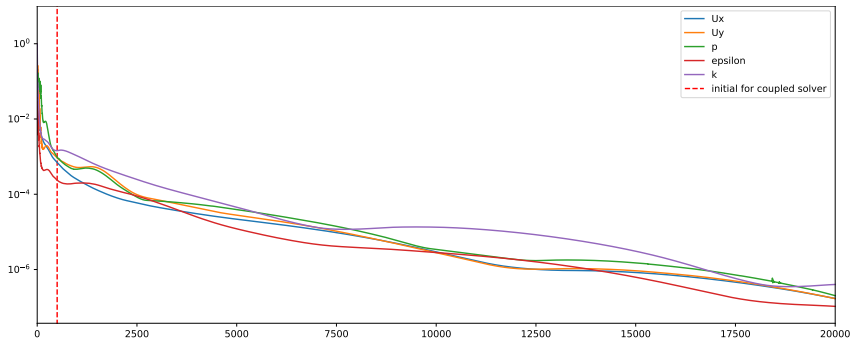
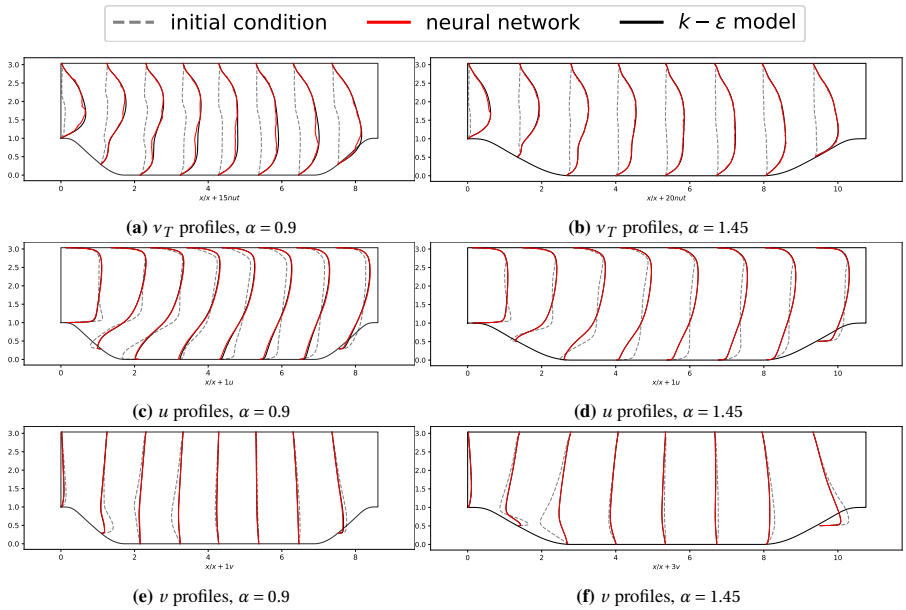


Figure 5.6: Convergence history of a simulation case (a=1.45)

one extrapolation case is shown in Figure 5.7. The dashed grey lines show the initial condition and the solid

red lines and black lines show the neural network and $k-\varepsilon$ results respectively. By comparing the grey dashed lines and the solid lines, it can be seen that there is a large discrepancy between the initial condition and the final converged results, especially in terms of the eddy viscosity. Although faced with unseen flow fields, the neural network turbulence model is capable of providing reasonable predictions without causing divergence of the primary RANS equation.
In the extrapolation flow case $\alpha = 0.9$, the nn coupled solver underpredicted the eddy viscosity in the bottom region of the flow. It resulted in a slightly smaller velocity in x direction in the corresponding region. The difference in ν can hardly be noticed. In the interpolation flow case $\alpha = 1.45$, the converged results of nn coupled solver differ from the baseline solver with a small high-frequency noise. Its influence on the mean velocity is negligible.
In conclusion, the neural network coupled flow solver is capable of providing results that are comparable to that of $k-\varepsilon$ solver on interpolation and moderate extrapolation flow cases.



(a) ν_T profiles, $\alpha = 0.9$ **(b)** ν_T profiles, $\alpha = 1.45$

(c) u profiles, $\alpha = 0.9$ **(d)** u profiles, $\alpha = 1.45$

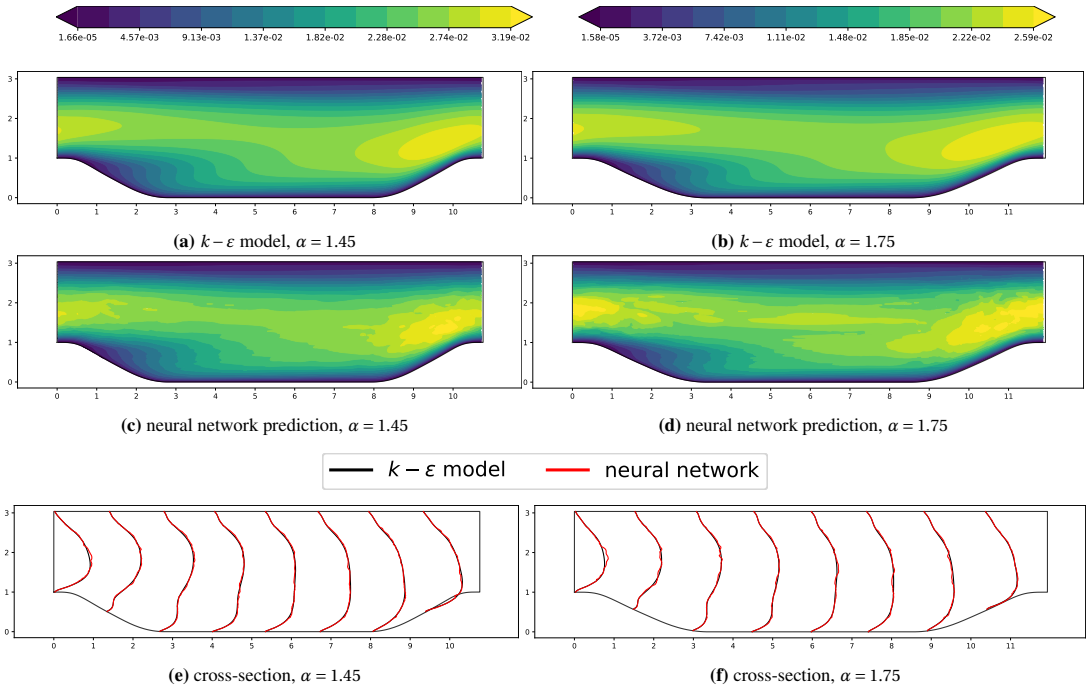
(e) ν profiles, $\alpha = 0.9$ **(f)** ν profiles, $\alpha = 1.45$

Figure 5.7: Comparison of the velocity and eddy viscosity profiles.

The initial condition marked in the grey dashed line is the corresponding baseline $k-\varepsilon$ model simulation at iteration 500 (starting from uniform flow field).

5.2. Performance of pointNet

After a thorough evaluation of the VCNN model performance, in this section, a brief report on the performance of pointNet architecture is presented as a comparison. The training dataset for pointNet architecture is the same as that of VCNN (see Table 4.2 and Figure 4.12).
Before discussing the validation error, it is worth noting that the training error of pointNet is higher than that of VCNN. For VCNN, the training error is around 2%, whilst for pointNet, it hardly dropped below 4%. A relatively high training error can be attributed to an insufficient linkage between the inputs and the outputs. In FVCC, all the input features listed in Section 4.3 are fed into the network. However in pointNet, out of the consideration of rotation invariance, the vector features are removed from the input dataset. In other words, there is possible information loss in the training of pointNet.
The pointNet model prediction on interpolation flow cases $\alpha = 1.45, 1.75$ and extrapolation flow cases $\alpha = 0.9, 2.1$ is shown in Figure 5.8 and Figure 5.9. According to the contour and the cross-section profile plots, the performance of pointNet architecture is a bit inferior compared to VCNN. Although the overall prediction of pointNet is also close to the baseline model, the pointNet model appears to have larger high-frequency oscillatory noise components. Even in interpolation cases on which VCNN shows excellent performance, the pointNet prediction has oscillatory error centring the baseline results. The discrepancy in extrapolation cases is also larger than that of VCNN.
Based on the definition of error rate Equation 5.1, the model performance of pointNet on flow cases with varying

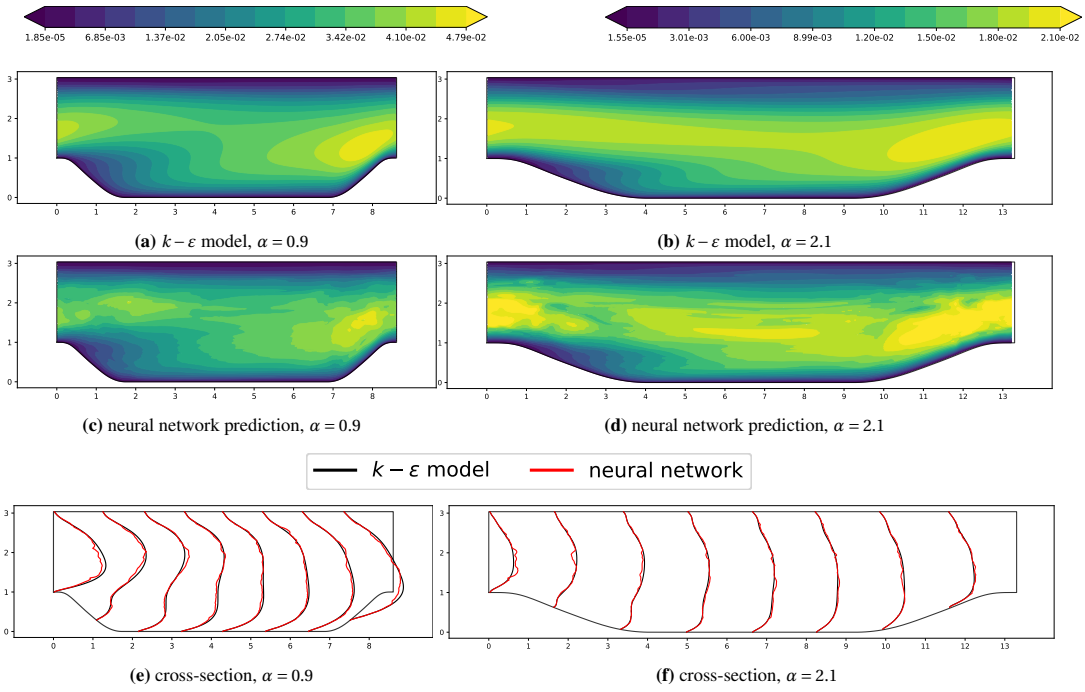


(a) $k-\varepsilon$ model, $\alpha = 1.45$ **(b)** $k-\varepsilon$ model, $\alpha = 1.75$

(c) neural network prediction, $\alpha = 1.45$ **(d)** neural network prediction, $\alpha = 1.75$

(e) cross-section, $\alpha = 1.45$ **(f)** cross-section, $\alpha = 1.75$

Figure 5.8: Comparison of the contour and the cross-section profiles of turbulence viscosity ν_T for interpolation flow cases $\alpha = 1.45$ and $\alpha = 1.75$ (pointNet)



(a) $k-\varepsilon$ model, $\alpha = 0.9$ **(b)** $k-\varepsilon$ model, $\alpha = 2.1$

(c) neural network prediction, $\alpha = 0.9$ **(d)** neural network prediction, $\alpha = 2.1$

(e) cross-section, $\alpha = 0.9$ **(f)** cross-section, $\alpha = 2.1$

Figure 5.9: Comparison of the contour and the cross-section profiles of turbulence viscosity ν_T for $\alpha = 0.9$ and $\alpha = 2.1$ (pointNet)

slope parameters is shown in Figure 5.10. The error rate of pointNet is generally larger than that of VCNN, which is in accordance with the larger training error. With training error as a reference, the performance for interpolation is comparable to that of VCNN. The extrapolation to smaller slope parameters in comparison is much worse. The largest validation error is at $\alpha = 0.5$ with a value of over 45%.

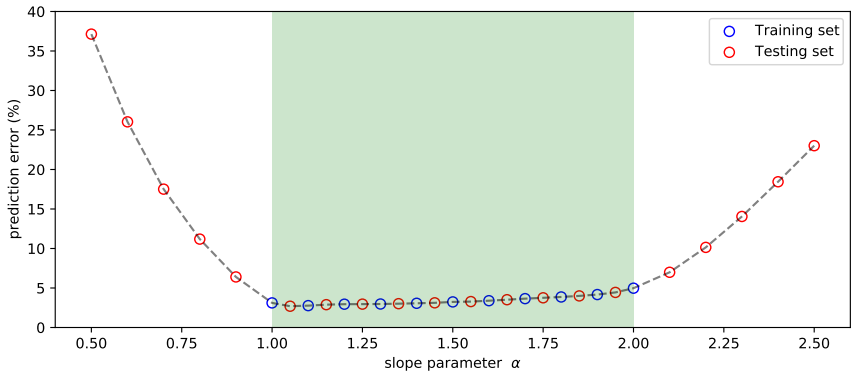


Figure 5.10: Prediction error over different slope parameters (pointNet).

5.3. Scale of the influence region

Since one of the major model properties is its nonlocality, it is helpful to understand the effect of influence region size on model performance. A parametric study on the scale of the influence region was performed in the uncoupled setup. The results is shown in Figure 5.11. x-axis is the factor with regards to the baseline choice of the influence region as provided in (4.2). The experiment consist of set $[0, 0.05, 0.1, 0.2, 0.5, 1, 1.5, 2.0, 3.0]$. A factor of 0 refers to a local model in which only the mean flow properties of the point of interest itself is regarded as the input of the model. The performance of models in the entire flow domain as well as in the recirculation region with various scales of the influence region is evaluated on two extrapolation flow cases $a = 0.5$ and $a = 0.8$. The recirculation region is marked by a red block in Figure 5.12.

It can be seen that the validation error showed a declining tendency with a larger influence region. Along the curves, two stages can be identified according to their slope. The first stage is from $I = 0$ to $I = 0.1$, in which the improvement of performance is the most evident. Figure 5.12 shows the size of influence regions at four points located at the inlet, in the recirculation region and mainstream, and near the bottom. As indicated in Figure 5.12, when the influence region reaches $0.1I$, stencils in most parts of the flow domain contain neighbouring points both along and orthogonal to the streamwise direction. Stencils like this can be considered as containing a rather "complete" set of information (gradient in all the directions) for the prediction of turbulence quantities. It is noticed that in the recirculation region, $I = 0.2$ also belongs to the first stage. Because in the near-wall region, only until $0.2I$ are the neighbouring points along the main flow direction included.

In the second stage ($I > 0.1$), the performance shows a little improvement. It indicates that by including points in a larger neighbourhood, the prediction capacity of the network may increase. There is also an increase of error when the influence region reaches the size of 3.0. This might be an indication of the negative effects of using a too large influence region. However, the small fluctuation of validation error is not persuasive enough for making a solid conclusion for the second stage considering the complexity of the model and the randomness introduced during the training process.

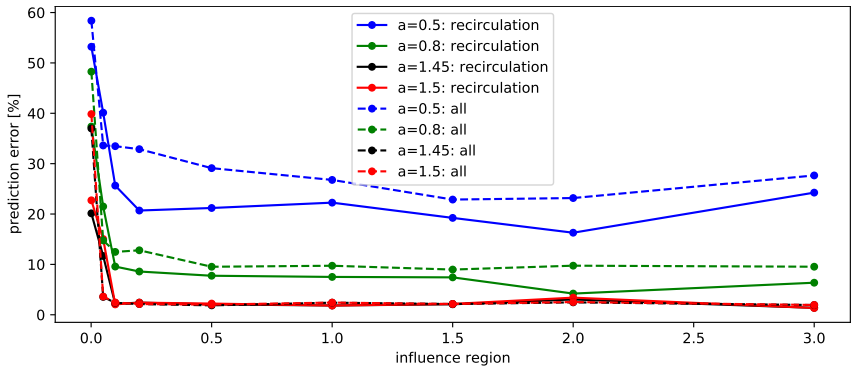


Figure 5.11: Parametric study on the influence region.

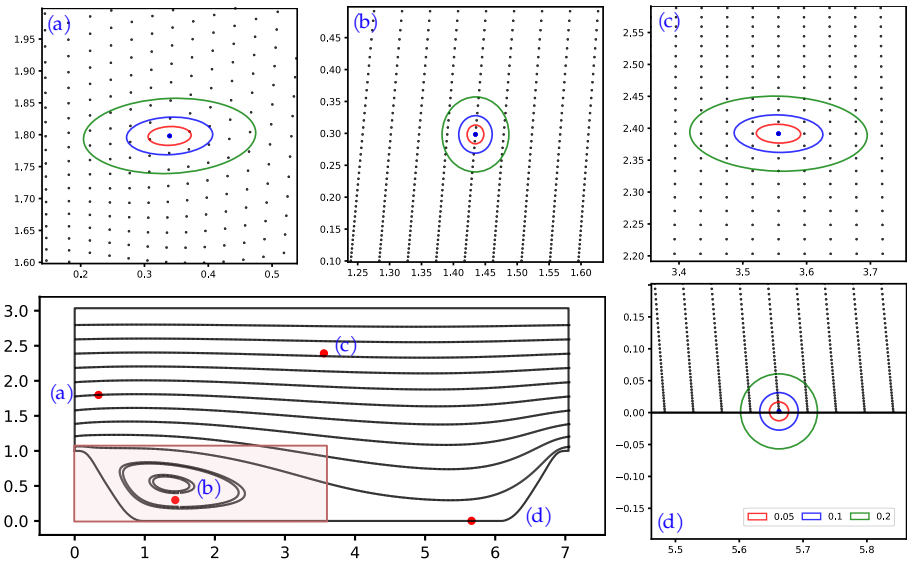


Figure 5.12: Visualization of stencils under various sizes of influence region for flow case $\alpha = 0.5$.

5.4. Flow physics learned by the neural network

Neural networks are notorious for being abstruse and hard to interpret. In this section, some tentative understanding of the flow physics learned by the neural network is provided. The interpretation is still vague and incomplete. Hopefully, it can reveal part of the mechanism based on which the network works.

As discussed in Section 4.5, there are two subnetworks in VCNN architecture, the embedding network and the fitting network. The outputs of the fitting network are the final outputs and are not helpful for understanding the network. Whilst the encoding functions given by the embedding network are intermediate results and have clear numerical meanings.

One of the encoding functions G_1 is visualized in Figure 5.13. In the training of this network, m' is set to 1 such that the information extracted in the embedding network is concentrated in the first encoding function G_1. Across stencils at different locations, the contours of G_1 showed a consistent pattern. The gradient of the encoding function G_1 is well aligned with the wall-normal direction, especially for stencils near the periodic hills. There is good accordance between the pattern of G_1 and the physics that flow inside a periodic hill tunnel is dominated by walls.

Since the encoding function is local in its nature, it can also be plotted over the entire flow domain as shown

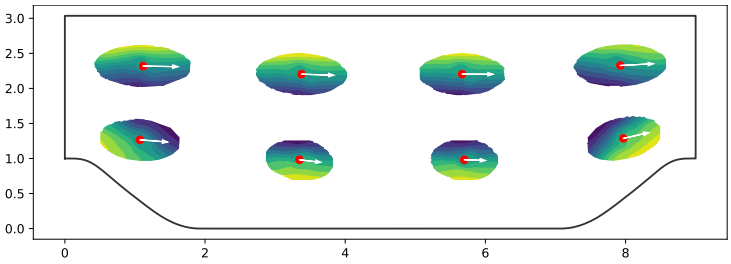
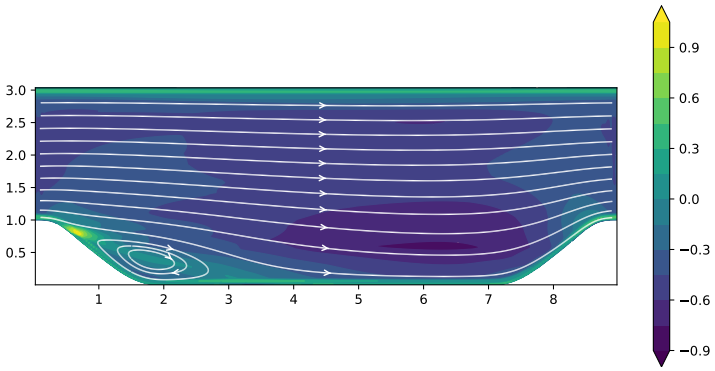
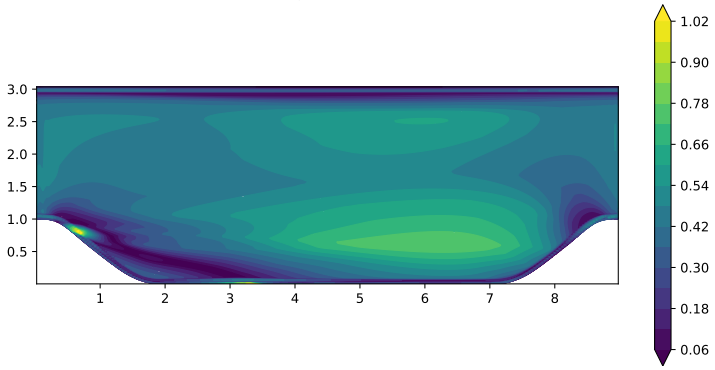


Figure 5.13: Contour plot of encoding function G_1 inside stencils at various location.

in Figure 5.14. If we construe the encoding function as a weight given to each sample in the stencil when doing summation, then the larger the absolute value of the point, the more important the network regards it as. It is observed in Figure 5.14b that the encoding functions have a larger magnitude when the strain rate is large, especially at the points of separation reattachment.



(a) Contour of encoding function G_1 (with streamline), $\alpha = 1.0$



(b) Contour of encoding function(absolute value) G_1, $\alpha = 1.0$

Figure 5.14: Visualization of encoding function G_1 over the whole flow domain

6

Conclusion and Recommendation

6.1. Conclusion

Turbulence modelling is one of the most challenging modelling problems in the field of engineering. Although there have been various models developed in the past 50 years, and some of them such as the $k-\varepsilon$ model have become the workhorse in fluid simulation, the current models are still limited in their accuracy and applicability. For instance, all of the eddy viscosity models are under the restriction of the Boussinesq assumption. It assumes that the Reynolds stress anisotropy is determined by the mean velocity gradient. However, this has been proved wrong in simple flow cases such as axisymmetric contraction. According to the rapid-distortion theory (RDT), the Reynolds stress is determined by the total mean strain instead of the mean velocity gradient. In such a case, the intrinsic assumption of eddy viscosity models breaks down.

In the thesis work, the rising machine learning techniques are utilized and applied to the long-standing turbulence modelling problem. A framework for building neural networks based turbulence models is proposed, including the data-preprocessing procedure and the construction of neural network architectures. Turbulence models based on this framework possess two major attributes, non-locality and invariance properties. Non-locality of the models means that the local quantities of interests at a certain point are determined not only by local mean flow fields but also by the mean flow quantities in their neighbouring region. A mapping between the mean flow features in the neighbouring region to the turbulence quantities at the central point is built. This attribute matches the non-local nature of turbulence.

In terms of the invariance properties, the model is invariant in two senses. On one hand, it is frame-independent based on flow physics. The model outputs are invariant with regards to translation, rotation of the reference frame. It also possesses Galilean invariance, according to which the prediction on the turbulence quantities is identical in all inertial frames. On the other hand, it is permutation invariant with regards to the input feature vector set. It is a requirement posed by the need for computation, according to which the model prediction is irrelevant to the ordering of the sample points in the stencil.

Regarding the specific neural network architectures, there are two architectures introduced in this thesis work. They both possess the attributes stated above. The major difference is how they achieve rotation invariance. The first Vector Cloud Neural Network (VCNN) achieves rotation invariance through a pairwise projection implemented in its architecture. In other words, the network has embedded rotation invariance and the input feature matrix can be rotation variant. In contrast, rotation invariance of the second pointNet architecture is achieved by selecting only the rotation invariance features as the network input. Rotation invariance is not built into the architecture itself. This gives pointNet the advantage of having a simpler structure. The remaining part of the networks is very much alike.

The training of the networks uses simulation data obtained by k-ε turbulence model on parametric periodic hills. The training set is composed of 11 flow cases with their slope parameters ranging from 1.0 to 2.0. It covers the low slope parameter cases characterized by evident recirculation region and high slope parameter cases with barely any separation. During training, The ADAM optimizer runs until the training error is brought to around 2% and the validation error stays at a low level.

The trained networks are evaluated in both coupled and uncoupled settings with unseen flow cases(interpolation and extrapolation). In the uncoupled setup, the VCNN network shows low training error in interpolation cases.

The error rate is comparable to that on the training set (around 2%). In terms of extrapolation, the training error grows relatively rapid as the slope parameter deviates further from the training set. For flow case $\alpha = 0.9$, the validation error is around 5%. As the slope parameter goes lower to $\alpha = 0.5$ (the lowest in the extrapolation set), the error reaches 25%. Towards the higher end, the flow case with the highest slope parameter $\alpha = 2.5$ has a validation error of around 20%. The high performance for interpolation and bad performance for extrapolation indicated that the application scenario should be covered well by the training data or at least close enough to the training data. An experiment on a broader but sparser training dataset is performed. The results show that with 6 flow cases linearly spaced between 0.5 and 2.5 (every 0.4), the prediction error is well controlled. The largest error appears at $\alpha = 0.7$ with a value of around 6%.
The performance of pointNet is comparable to that of VCNN in terms of interpolation and extrapolation to larger slope parameters. However, when extrapolated to low slope parameter cases, the performance is much worse than that of VCNN. The largest error is nearly 40% at $\alpha = 0.5$. This can be due to the information loss of discarding the vector features in the inputs.
In the coupled setup, the neural network is coupled with simpleFOAM solver of OpenFOAM and provides the turbulence quantities in the solver iteration. The experiments are performed on a interpolation flow case $\alpha = 1.45$ and a extrapolation flow case $\alpha = 0.9$. The coupled solver produced a prediction similar to that of the baseline $k-\varepsilon$ turbulence model. The tests performed in the coupled setting shows good numerical behaviour of the neural network turbulence model. It is capable of stabilizing the simulation from early-stage simulation results.
For a better understanding of the non-locality property of the network, a parametric study of the stencil scale was performed. Models were trained with a stencil scale ranging from 0 (local) to 3 times the baseline scale. The prediction error of the strictly local model (only the center point is taken as input) was larger than 20% for both training and testing cases. The model prediction error reduced rapidly until the closest mesh points in roughly all directions were taken into consideration. This can be regarded as having similar nonlocality as the classic $k-\varepsilon$ model, in which the velocity gradient is used. For larger influence region, no evident improvement was found. In the end, an attempt was made on interpreting the flow physics learned by the network with the encoding functions visualized. It was shown that the encoding function had a strong relationship with wall distance and strain rate magnitude.

6.2. Recommendation for future work

Current works indicated that non-local neural networks are very promising in terms of turbulence modelling. By including information in the neighbouring flow field, the prediction performance of the model can be largely improved. Nevertheless, there are many improvements that can be made to the thesis work.
First of all, the only geometry used now is the parametric periodic hill geometry. For a more comprehensive evaluation of the framework, a first step is to perform experiments on various fluid domain geometries, such as the internal flow through a backwards-facing step and the external flow around an airfoil. These flow cases can be helpful in showing the merits of all the invariance properties embedded in the network architecture. Besides, the framework can be easily transplanted to high-fidelity databases, such as DNS simulation data. In this way, the model accuracy can be improved greatly.
Except for the limitations mentioned above, a major shortcoming of the network architecture is that it is only capable of predicting scalar quantities. For instance, in our case, the outputs of the network are scalar quantities k and ε. The following direction is to modify the architecture such that it can predict tensors such as the Reynolds stress tensor. In this way, neural network turbulence models that are comparable to Reynolds stress transport models can be developed. The model applicability can thus be promoted fundamentally. The biggest challenge for extending the network output from scalar to tensor is to reserve the invariance properties of the current architecture. This may involve the use of an invariant tensor basis.
Another improvement regarding the framework is the implementation of coupling between turbulence models and the simpleFOAM solver. Currently, they are coupled through files, that is, the network reads the mean flow information file written by simpleFOAM solver and writes the prediction into another file. SimpleFOAM solver, in turn, reads the network outputs and writes the mean flow field. A possible improvement is to "fully" couple the two components. Corrections provided by the network can be directly made to the solver. In this way, the solver efficiency is possibly promoted. However, such modification will pose a great challenge to the stability of the system. Its applicability requires careful investigation.

Bibliography

[1] Parviz Moin and Krishnan Mahesh. "Direct numerical simulation: a tool in turbulence research". In: *Annual review of fluid mechanics* 30.1 (1998), pp. 539–578.

[2] Karthik Duraisamy, Gianluca Iaccarino, and Heng Xiao. "Turbulence modeling in the age of data". In: *Annual Review of Fluid Mechanics* 51 (2019), pp. 357–377.

[3] Julia Ling, Reese Jones, and Jeremy Templeton. "Machine learning strategies for systems with invariance properties". In: *Journal of Computational Physics* 318 (2016), pp. 22–35.

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

[5] Mikael LA Kaandorp and Richard P Dwight. "Data-driven modelling of the Reynolds stress tensor using random forests with invariance". In: *Computers & Fluids* 202 (2020), p. 104497.

[6] Hugo Frezat et al. "Physical invariance in neural networks for subgrid-scale scalar flux modeling". In: *Physical Review Fluids* 6.2 (2021), p. 024607.

[7] Philippe R Spalart. "Philosophies and fallacies in turbulence modeling". In: *Progress in Aerospace Sciences* 74 (2015), pp. 1–15.

[8] Jian-Xun Wang, Jin-Long Wu, and Heng Xiao. "Physics-informed machine learning approach for reconstructing Reynolds stress modeling discrepancies based on DNS data". In: *Physical Review Fluids* 2.3 (2017), p. 034603.

[9] Brendan D Tracey, Karthikeyan Duraisamy, and Juan J Alonso. "A machine learning strategy to assist turbulence model development". In: *53rd AIAA aerospace sciences meeting*. 2015, p. 1287.

[10] Stephen B Pope. *Turbulent flows*. 2001.

[11] Thomas B Gatski, M Yousuff Hussaini, and John L Lumley. *Simulation and modeling of turbulent flows*. Oxford University Press, 1996.

[12] Xu-Hui Zhou, Jiequn Han, and Heng Xiao. "Frame-independent vector-cloud neural network for nonlocal constitutive modelling on arbitrary grids". In: *arXiv preprint arXiv:2103.06685* (2021).

[13] Jinlong Wu et al. "Reynolds-averaged Navier–Stokes equations with explicit data-driven Reynolds stress closure can be ill-conditioned". In: (2019).

[14] Quanshi Zhang and Song-Chun Zhu. "Visual interpretability for deep learning: a survey". In: *arXiv preprint arXiv:1802.00614* (2018).

[15] Luca Guastoni et al. "Prediction of wall-bounded turbulence from wall quantities using convolutional neural networks". In: *Journal of Physics: Conference Series*. Vol. 1522. 1. IOP Publishing. 2020, p. 012022.

[16] Corentin J Lapeyre et al. "Training convolutional neural networks to estimate turbulent sub-grid scale reaction rates". In: *Combustion and Flame* 203 (2019), pp. 255–264.

[17] Xu-Hui Zhou, Jiequn Han, and Heng Xiao. "Learning nonlocal constitutive models with neural networks". In: *Computer Methods in Applied Mechanics and Engineering* 384 (2021), p. 113927.

[18] Craig R Gin et al. "DeepGreen: Deep Learning of Green's Functions for Nonlinear Boundary Value Problems". In: *arXiv preprint arXiv:2101.07206* (2020).

[19] Aharon Azulay and Yair Weiss. "Why do deep convolutional networks generalize so poorly to small image transformations?" In: *arXiv preprint arXiv:1805.12177* (2018).

[20] Connor Shorten and Taghi M Khoshgoftaar. "A survey on image data augmentation for deep learning". In: *Journal of Big Data* 6.1 (2019), pp. 1–48.

[21] Kunihiko Fukushima. "Neocognitron: A hierarchical neural network capable of visual pattern recognition". In: *Neural networks* 1.2 (1988), pp. 119–130.

[22] Brian Edward Launder and Dudley Brian Spalding. "The numerical computation of turbulent flows". In: *Numerical prediction of flow, heat transfer, turbulence and combustion*. Elsevier, 1983, pp. 96–116.

[23] Michael I Jordan and Tom M Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245 (2015), pp. 255–260.

[24] Issam El Naqa and Martin J Murphy. "What is machine learning?" In: *machine learning in radiation oncology*. Springer, 2015, pp. 3–11.

[25] Carl Kingsford and Steven L Salzberg. “What are decision trees?” In: *Nature biotechnology* 26.9 (2008), pp. 1011–1013.
[26] Tin Kam Ho. “Random decision forests”. In: *Proceedings of 3rd international conference on document analysis and recognition*. Vol. 1. IEEE. 1995, pp. 278–282.
[27] Yali Amit and Donald Geman. “Shape quantization and recognition with randomized trees”. In: *Neural computation* 9.7 (1997), pp. 1545–1588.
[28] Thomas G Dietterich. “Ensemble methods in machine learning”. In: *International workshop on multiple classifier systems*. Springer. 2000, pp. 1–15.
[29] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32.
[30] Gérard Biau and Erwan Scornet. “A random forest guided tour”. In: *Test* 25.2 (2016), pp. 197–227.
[31] Aleksei Grigorevich Ivakhnenko and Valentin Grigor’evich Lapa. *Cybernetic predicting devices*. Tech. rep. PURDUE UNIV LAFAYETTE IND SCHOOL OF ELECTRICAL ENGINEERING, 1966.
[32] Kunihiko Fukushima. “Neural network model for a mechanism of pattern recognition unaffected by shift in position-Neocognitron”. In: *IEICE Technical Report, A* 62.10 (1979), pp. 658–665.
[33] DB Parker. “Learning-logic: Casting the cortex of the human brain in silicon”. In: *Technical report TR-47* (1985).
[34] Yann LeCun et al. “A theoretical framework for back-propagation”. In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1. 1988, pp. 21–28.
[35] Sepp Hochreiter. “The vanishing gradient problem during learning recurrent neural nets and problem solutions”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.
[36] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.
[37] Charles R Qi et al. “Pointnet: Deep learning on point sets for 3d classification and segmentation”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 652–660.
[38] Charles R Qi et al. “Pointnet++: Deep hierarchical feature learning on point sets in a metric space”. In: *arXiv preprint arXiv:1706.02413* (2017).
[39] Yulan Guo et al. “Deep learning for 3d point clouds: A survey”. In: *IEEE transactions on pattern analysis and machine intelligence* (2020).
[40] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. “Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations”. In: *arXiv preprint arXiv:1711.10561* (2017).
[41] Maziar Raissi and George Em Karniadakis. “Hidden physics models: Machine learning of nonlinear partial differential equations”. In: *Journal of Computational Physics* 357 (2018), pp. 125–141.
[42] Yinhao Zhu et al. “Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data”. In: *Journal of Computational Physics* 394 (2019), pp. 56–81.
[43] Jens Berg and Kaj Nyström. “A unified deep artificial neural network approach to partial differential equations in complex geometries”. In: *Neurocomputing* 317 (2018), pp. 28–41.
[44] Zichao Long et al. “Pde-net: Learning pdes from data”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 3208–3216.
[45] Zichao Long, Yiping Lu, and Bin Dong. “PDE-Net 2.0: Learning PDEs from data with a numeric-symbolic hybrid deep network”. In: *Journal of Computational Physics* 399 (2019), p. 108925.
[46] Kathleen Champion et al. “Data-driven discovery of coordinates and governing equations”. In: *Proceedings of the National Academy of Sciences* 116.45 (2019), pp. 22445–22451.
[47] Tong Qin, Kailiang Wu, and Dongbin Xiu. “Data driven governing equations approximation using deep neural networks”. In: *Journal of Computational Physics* 395 (2019), pp. 620–635.
[48] Yifan Sun, Linan Zhang, and Hayden Schaeffer. “Neupde: Neural network based ordinary and partial differential equations for modeling time-dependent data”. In: *Mathematical and Scientific Machine Learning*. PMLR. 2020, pp. 352–372.
[49] Samuel H Rudy, J Nathan Kutz, and Steven L Brunton. “Deep learning of dynamics and signal-noise decomposition with time-stepping constraints”. In: *Journal of Computational Physics* 396 (2019), pp. 483–506.
[50] Jan S Hesthaven and Stefano Ubbiali. “Non-intrusive reduced order modeling of nonlinear problems using neural networks”. In: *Journal of Computational Physics* 363 (2018), pp. 55–78.
[51] Sharmila Karumuri et al. “Simulator-free solution of high-dimensional stochastic elliptic partial differential equations using deep neural networks”. In: *Journal of Computational Physics* 404 (2020), p. 109120.
[52] Linfeng Zhang et al. “Deep potential molecular dynamics: a scalable model with the accuracy of quantum mechanics”. In: *Physical review letters* 120.14 (2018), p. 143001.

[53] Masataka Gamahara and Yuji Hattori. “Searching for turbulence models by artificial neural network”. In: *Physical Review Fluids* 2.5 (2017), p. 054604.
[54] Zelong Yuan, Chenyue Xie, and Jianchun Wang. “Deconvolutional artificial neural network models for large eddy simulation of turbulence”. In: *Physics of Fluids* 32.11 (2020), p. 115106.
[55] Zhuo Wang et al. “Investigations of data-driven closure for subgrid-scale stress in large-eddy simulation”. In: *Physics of Fluids* 30.12 (2018), p. 125101.
[56] Martin Schmelzer, Richard P Dwight, and Paola Cinnella. “Discovery of algebraic Reynolds-stress models using sparse symbolic regression”. In: *Flow, Turbulence and Combustion* 104.2 (2020), pp. 579–603.
[57] Yu Zhang et al. “Customized data-driven RANS closures for bi-fidelity LES–RANS optimization”. In: *Journal of Computational Physics* 432 (2021), p. 110153.
[58] Jasper P Huijing, Richard P Dwight, and Martin Schmelzer. “Data-driven RANS closures for three-dimensional flows around bluff bodies”. In: *Computers & Fluids* 225 (2021), p. 104997.
[59] Romit Maulik, Omer San, and Jamey D Jacob. “Spatiotemporally dynamic implicit large eddy simulation using machine learning classifiers”. In: *Physica D: Nonlinear Phenomena* 406 (2020), p. 132409.
[60] Yohai Bar-Sinai et al. “Learning data-driven discretizations for partial differential equations”. In: *Proceedings of the National Academy of Sciences* 116.31 (2019), pp. 15344–15349.
[61] Andrea Beck and Marius Kurz. “A perspective on machine learning methods in turbulence modeling”. In: *GAMM-Mitteilungen* 44.1 (2021), e202100002.
[62] Sai Hung Cheung et al. “Bayesian uncertainty analysis with applications to turbulence modeling”. In: *Reliability Engineering & System Safety* 96.9 (2011), pp. 1137–1149.
[63] Jaideep Ray et al. “Bayesian calibration of a k-ε turbulence model for predictive jet-in-crossflow simulations”. In: *44th AIAA Fluid Dynamics Conference*. 2014, p. 2085.
[64] Anikesh Pal. “Deep learning parameterization of subgrid scales in wall-bounded turbulent flows”. In: *arXiv preprint arXiv:1905.12765* (2019).
[65] Jin-Long Wu, Heng Xiao, and Eric Paterson. “Physics-informed machine learning approach for augmenting turbulence models: A comprehensive framework”. In: *Physical Review Fluids* 3.7 (2018), p. 074602.
[66] Anand Pratap Singh, Shivaji Medida, and Karthik Duraisamy. “Machine-learning-augmented predictive modeling of turbulent separated flows over airfoils”. In: *AIAA journal* 55.7 (2017), pp. 2215–2227.
[67] Chenyue Xie, Jianchun Wang, and E Weinan. “Modeling subgrid-scale forces by spatial artificial neural networks in large eddy simulation of turbulence”. In: *Physical Review Fluids* 5.5 (2020), p. 054606.
[68] Xiaowei Jin et al. “NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations”. In: *Journal of Computational Physics* 426 (2021), p. 109951.
[69] Finale Doshi-Velez and Been Kim. “Towards a rigorous science of interpretable machine learning”. In: *arXiv preprint arXiv:1702.08608* (2017).
[70] S Beetham and J Capecelatro. “Formulating turbulence closures using sparse regression with embedded form invariance”. In: *Physical Review Fluids* 5.8 (2020), p. 084611.
[71] Julia Ling. “Using machine learning to understand and mitigate model form uncertainty in turbulence models”. In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2015, pp. 813–818.
[72] Haitz Sáez de Ocáriz Borde, David Sondak, and Pavlos Protopapas. “Convolutional Neural Network Models and Interpretability for the Anisotropic Reynolds Stress Tensor in Turbulent One-dimensional Flows”. In: *arXiv preprint arXiv:2106.15757* (2021).
[73] Andrew Bennett and Bart Nijssen. “Explainable AI uncovers how neural networks learn to regionalize in simulations of turbulent heat fluxes at FluxNet sites”. In: *Earth and Space Science Open Archive ESSOAr* (2021).
[74] Jianheng Tan et al. *Towards Explainable Machine Learning Assisted Turbulence Modelling for Transonic Flows*. Oct. 2020.
[75] Martin Lellep et al. “Interpreted machine learning in fluid dynamics: Explaining relaminarisation events in wall-bounded shear flows”. In: *arXiv preprint arXiv:2102.05541* (2021).
[76] Arvind T Mohan et al. “Embedding hard physical constraints in neural network coarse-graining of 3d turbulence”. In: *arXiv preprint arXiv:2002.00021* (2020).
[77] Jin-Long Wu et al. “A priori assessment of prediction confidence for data-driven turbulence modeling”. In: *Flow, Turbulence and Combustion* 99.1 (2017), pp. 25–46.
[78] Ashley Scillitoe, Pranay Seshadri, and Mark Girolami. “Uncertainty quantification for data-driven turbulence modelling with Mondrian forests”. In: *Journal of Computational Physics* 430 (2021), p. 110116.

[79] Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. "Mondrian forests: Efficient online random forests". In: *Advances in neural information processing systems* 27 (2014), pp. 3140–3148.
[80] Shirui Luo et al. "Review and examination of input feature preparation methods and machine learning models for turbulence modeling". In: *arXiv preprint arXiv:2001.05485* (2020).
[81] David C Wilcox et al. *Turbulence modeling for CFD*. Vol. 2. DCW industries La Canada, CA, 1998.
[82] *OpenFOAM: User Guide v2012*. URL: https://www.openfoam.com/documentation/guides/latest/doc/guide-turbulence-ras-k-epsilon.html.
[83] *GAMGSolverAgglomerateMatrix.C*. https://develop.openfoam.com/Development/openfoam/blob/OpenFOAM-v2012/src/OpenFOAM/matrices/lduMatrix/solvers/GAMG/. Accessed: 2021-09-19.
[84] LS Caretto et al. "Two calculation procedures for steady, three-dimensional flows with recirculation". In: *Proceedings of the third international conference on numerical methods in fluid mechanics*. Springer. 1973, pp. 60–68.
[85] Joel H Ferziger, Milovan Perić, and Robert L Street. *Computational methods for fluid dynamics*. Vol. 3. Springer, 2002.
[86] Suhas V Patankar. *Numerical heat transfer and fluid flow*. CRC press, 2018.
[87] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method*. Pearson education, 2007.
[88] *NN-SVG (Publication-ready NN-architecture schematics)*. URL: http://alexlenail.me/NN-SVG/index.html.
[89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.
[90] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.
[91] Xing Wan. "Influence of feature scaling on convergence of gradient iterative algorithm". In: *Journal of Physics: Conference Series*. Vol. 1213. 3. IOP Publishing. 2019, p. 032021.
[92] Jorge Sola and Joaquin Sevilla. "Importance of input data normalization for the application of neural networks to complex industrial problems". In: *IEEE Transactions on nuclear science* 44.3 (1997), pp. 1464–1468.
[93] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).
[94] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.
[95] Andrei Nikolaevich Kolmogorov. "On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition". In: *Doklady Akademii Nauk*. Vol. 114. 5. Russian Academy of Sciences. 1957, pp. 953–956.
[96] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).
[97] *PyTorch*. URL: https://pytorch.org/.
[98] *Embedding Python in Another Application*. URL: https://docs.python.org/3/extending/embedding.html#.