*O*n deep learning f*o*r c*o*mputati*o*nal fluid dynamics

Fernando Adan Gonzalez

Normandie Université

UNIVERSITÉ DE ROUEN NORMANDIE

THÈSE

Pour obtenir le diplôme de doctorat

Spécialité MECANIQUE

Préparée au sein de l'Université de Rouen Normandie

On deep learning for computational fluid dynamics

Présentée et soutenue par
FERNANDO GONZALEZ

Thèse soutenue le 29/11/2023 devant le jury composé de		
M. ERWIN FRANQUET	Professeur des Universités, UNIVERSITE COTE D'AZUR	Rapporteur du jury
M. SIMON BERNARD	Maître de Conférences, Université de Rouen Normandie	Membre du jury
M. SYLVAIN CHEVALIER	Professeur des Universités, Université Paris-Saclay	Membre du jury
M. THOMAS GOMEZ	Professeur des Universités, UNIVERSITE LILLE 1 SCIENCES ET TECHNOLOGIE	Membre du jury
M. FLORIAN YGER	Maître de Conférences, UNIVERSITE PARIS 9 (Dauphine)	Membre du jury
M. PIERRE TRONTIN	Professeur des universités (université Française), Université Claude Bernard Lyon I	Président du jury
M. FRANÇOIS-XAVIER DEMOULIN	Professeur des Universités, Université de Rouen Normandie	Directeur de thèse

Thèse dirigée par FRANÇOIS-XAVIER DEMOULIN (COMPLEXE DE RECHERCHE INTERPROFESSIONNEL EN AEROTHERMOCHIMIE)

ED 591 PSIME

CORIA UMR 6614

Abstract

Computational Fluid Dynamics (CFD) plays a pivotal role in simulating and understanding fluid flow phenomena across various domains, including aerospace, automotive, environmental science, and biomedical engineering. Traditional CFD methods, predominantly based on numerical discretization of the governing equations, often face challenges in achieving high accuracy and computational efficiency, especially for complex and turbulent flows. Deep Learning (DL), a subfield of artificial intelligence, has emerged as a promising approach to address these challenges by integrating data-driven techniques with numerical simulations.

This Ph.D. thesis comprehensively explores integrating Deep Learning techniques into Computational Fluid Dynamics, focusing on enhancing both accuracy and efficiency in flow simulations. This work is divided into two parts; the first one is dedicated to the fundamentals of DL and a literature review of state-of-the-art applications of DL in CFD. The second part explores the questions of how to develop data-driven surrogate models of turbulence flow simulations.

The main takeaways of this thesis are the following:

- Chapter 2 introduces fundamental concepts of Machine Learning that will be used in the work and appear in most of the literature dedicated to ML for CFD.
- Chapter 3 provides an up-to-date survey of the current advancements in the application of ML to CFD. In addition, a critical perspective is provided with the purpose of identifying possible research directions.
- Chapter 4 explores generative modeling with Generative Adversarial Networks (GANs) for synthetic turbulence generation and supervised learning for predicting a turbulent flow. The first section evaluates the main difficulties GANs face in generating turbulent flows by implementing a model for the generation of turbulent signals both in 1D and 2D. The second section implements a framework with an Autoencoder for 3D turbulent flows and a Convolutional LSTM model for temporal prediction using the learned latent representation.
- In chapter 5, we train Neural Operators to learn the solution operator of the Navier-Stokes equation from simulation data. Three models are trained as surrogate models of the simulations. The problem of achieving numerical stability for the DL models is also addressed.

We conclude that Deep Learning provides an interesting set of tools that will be useful to scientists and engineers working with fluid dynamics. There are still many open questions that need to be answered so that Artificial Intelligence becomes part of the standard for the field of CFD. This work provides a ground-level reference for any CFD practitioner wanting to start applying DL for their problems or develop new tools to advance the state-of-the-art for this field.

Keywords— Deep Learning, Computational Fluid Dynamics, Surrogate Models Turbulence, Neural Operators

Résumé

La dynamique des fluides computationnelle (CFD) joue un rôle essentiel dans la simulation et la compréhension des phénomènes d'écoulement des fluides dans divers domaines, notamment l'aérospatiale, l'automobile, les sciences de l'environnement et l'ingénierie biomédicale. Les méthodes traditionnelles de CFD, principalement basées sur la discrétisation numérique des équations gouvernantes, rencontrent souvent des défis pour atteindre une haute précision et une efficacité computationnelle, en particulier dans le cas d'écoulements complexes et turbulents. L'apprentissage profond (DL), un sous-domaine de l'intelligence artificielle, s'est imposé comme une approche prometteuse pour relever ces défis en intégrant des techniques axées sur les données avec des simulations numériques.

Cette thèse de doctorat explore de manière approfondie l'intégration des techniques d'apprentissage profond dans la dynamique des fluides computationnelle, avec un accent sur l'amélioration à la fois de la précision et de l'efficacité dans les simulations d'écoulement. Ce travail est divisé en deux parties : la première est dédiée aux fondamentaux de l'apprentissage profond et à une revue de la littérature sur les applications de pointe de l'apprentissage profond en CFD. La deuxième partie explore les questions liées au développement de modèles de substitution basés sur les données pour les simulations d'écoulement turbulent.

Les idées principales de cette thèse sont les suivantes :

- Le chapitre 2 introduit les concepts fondamentaux de l'apprentissage automatique qui seront utilisés dans ce travail et qui sont présents dans la plupart de la littérature consacrée à l'apprentissage automatique pour la CFD.
- Le chapitre 3 propose une revue actualisée des avancées actuelles dans l'application de l'apprentissage automatique à la CFD. De plus, une perspective critique est fournie dans le but d'identifier des orientations de recherche possibles.
- Le chapitre 4 explore la modélisation générative avec les réseaux antagonistes génératifs (GAN) pour la génération synthétique de turbulence et l'apprentissage supervisé pour la prédiction d'un écoulement turbulent. La première section évalue les principales difficultés auxquelles les GAN sont confrontés lors de la génération d'écoulements turbulents en mettant en place un modèle de génération de signaux turbulents en 1D et 2D. La deuxième section met en œuvre un cadre avec un autoencodeur pour les écoulements turbulents en 3D et un modèle Convolutional LSTM pour la prédiction temporelle en utilisant la représentation latente apprise.
- Dans le chapitre 5, nous formons des opérateurs neuronaux pour apprendre l'opérateur de solution de l'équation de Navier-Stokes à partir de données de simulation. Trois modèles sont formés en tant que modèles de substitution des simulations. Le problème de l'atteinte de la stabilité numérique pour les modèles d'apprentissage profond est également abordé.

Nous concluons que l'apprentissage profond offre un ensemble d'outils intéressants qui seront utiles aux scientifiques et ingénieurs travaillant dans le domaine de la dynamique des fluides. Il reste de nombreuses questions en suspens qui doivent être résolues pour que l'intelligence artificielle devienne une norme dans le domaine de la CFD. Ce travail constitue une référence de base pour tout praticien de la CFD souhaitant commencer à appliquer l'apprentissage profond à ses problèmes ou développer de nouveaux outils pour faire progresser l'état de l'art dans ce domaine.

Mots-Clés— Apprentisage Profond, Mécanique des fluides numérique, modèles de substitution, Turbulence, Operateurs Neuronaux

Acknowledgements

I would like to express my deepest gratitude and appreciation to all those who have supported me throughout my doctoral journey. Completing this thesis would not have been possible without many individuals and organizations' unwavering support, guidance, and encouragement.

I am profoundly thankful to my thesis advisors, François-Xavier Demoulin and Simon Bernard, for their invaluable mentorship and discussions. Their expertise, insightful feedback, and unwavering support played a pivotal role in shaping this research. I am truly fortunate to have had the opportunity to work under their guidance. Besides that, I want to thank them for their trust in giving me this thesis project, ensuring I have the necessary means to work well, and for respecting my decisions and ideas.

I extend my heartfelt gratitude to my family: My parents, grandparents, and brother for their unwavering love, encouragement, and belief in my abilities. Their continuous support has been my constant source of strength throughout this journey. Passing the holidays in the Dominican Republic is a source of energy and inspiration to return to France and give the best of my efforts. I especially want to dedicate this work to my two grandfathers, Narciso and Adan, and grandmother, Morena, who are no longer with us. They are present in some of my fondest memories, and I am sure they will be very happy about this achievement wherever they are.

I am grateful to my friends and colleagues for their support, camaraderie, and the many stimulating discussions that enriched my academic experience. Their friendship has made the challenges of graduate school more bearable and the successes more enjoyable.

Last but not least, I would like to thank the academic and administrative staff of the CORIA lab for their assistance and resources, which were indispensable for successfully completing this thesis, particularly coffee. Thanks to GENCI and the CRIANN for the computational resources used during all these years, hopefully, no GPUs were harmed in the process.

This journey has been long and arduous, but it has been made immeasurably more manageable with the support of all those mentioned above. Thank you for being a part of this endeavor and helping me realize my academic and personal potential.

Acronyms

Re Reynolds Number. 118, 123, 145, 149, 151

AD Automatic Differentiation. 36

AE Autoencoder. x, 26, 27, 55–58, 63, 101, 103, 106, 107, 109, 115, 116

AI Artificial Intelligence. 11, 22

ANN Artificial Neural Network. 11

AR Autoregressive. 129

BN Batch Normalization. 20, 21, 104, 105

CFD Computational Fluid Dynamics. ix, 1–3, 37, 46, 51, 52, 59, 61, 63–65, 67–71, 118, 123, 124, 128, 130, 132, 133, 137

CNN Convolutional Neural Network. 24–26, 35, 42, 51, 53, 56, 58, 59, 61, 62, 65, 79, 102, 103, 120, 121, 147, 151

ConvLSTM Convolutional LSTM. x, 72, 96, 109–116

CR Compression Ratio. 109

DG Discontinuous Galerkin. 53

DGM Deep Generative Model. 73

DL Deep Learning. 3, 23, 24, 36, 51, 54, 66, 120, 137

DMD Dynamic Mode Decomposition. 55–57

DNN Deep Neural Network. 11

DNS Direct Numerical Simulation. ix, 2, 46, 47, 50–54, 61, 62, 64, 68, 86, 97, 100, 112, 114–116

DW Depth-Wise. 103

FAIR Findable, Accessible, Interoperable, Reusable. 63, 64, 68

FFN Feedforward Neural Network. 13, 25, 34, 38

FFT Fast Fourier Transform. 125

FNO Fourier Neural Operator. 65, 125–128, 131, 136, 137, 139–142, 147–151

FVM Finite Volume Method. 61

GAN Generative Adversarial Network. x, 28–30, 51, 63, 67, 71–82, 84, 85, 88, 90, 92–96, 120

GELU Gaussian error linear unit. 15, 128

GN Group Normalization. 21

GNN Graph Neural Network. 60

GP Gradient Penalty. 30

GPU Graphics Processing Unit. 100, 120, 129

GRN Global Response Normalization. 105

HIT Homogeneous Isotropic Turbulence. x, 58, 64, 72, 101, 106–109, 112, 115

HPC High-Performance Computing. 63, 69, 130

INR Implicit Neural Representation. 121

KnW Kolmogorov n-Width. 99, 100

LES Large Eddy Simulation. 2, 46–48, 50, 52–54, 59–61, 68, 77, 118

LN Layer Normalization. 21, 105

LSTM Long Short Term Memory Network. x, 31, 32, 56, 58, 61, 63, 70, 79, 100, 109, 115, 120

MARL Multi-Agent Reinforcement Learning. 53

ML Machine Learning. 1, 2, 7, 8, 10, 11, 15, 23, 26, 33, 36, 39, 43, 45, 46, 49, 54, 55, 58–65, 67–70, 124, 151

MLP Multi-Layer Perceptron. 13, 49, 51

MSE Mean squared error. 7, 92, 94, 109, 132, 141

NN Neural Network. 11, 17, 36, 38, 43, 49, 58, 61, 66, 121, 130, 132, 138

NO Neural Operator. 41

NODE Neural Ordinary Differential Equation. 43

NS Navier-Stokes. 1, 48, 51, 123, 124, 126, 137

ODE Ordinary Differential Equation. 36, 43–45, 55–57, 67

OU Ornstein–Uhlenbeck process. 78

PCA Principal Component Analysis. 57

PDE Partial Differential Equation. 36, 37, 39, 51, 54, 58–60, 66–68, 97, 98, 116, 119, 121, 124, 125, 131, 136, 144, 146

PDF Probability Density Function. 78, 81, 83–85, 93

PINN Physics-Informed Neural Network. 36, 37, 39, 51, 62, 121, 125, 147

POD Proper Orthogonal Decomposition. 42, 55, 57

PPO Proximal Policy Optimization. 53

QKV Query-Key-Value. 33

RAdam Rectified Adam. 19

RANS Reynolds-Averaged Navier Stokes. 2, 14, 37, 46, 48–50, 54, 59, 65, 68, 77, 118

RB Reduced Basis. 98

ReF-ER Remember and Forget Experience Replay. 53

ReLU Rectified linear unit. 14

Rms Root Mean Square. 113

RNN Recurrent Neural Network. x, 27, 30–32, 35, 42, 71, 79–81, 124, 125, 131

ROM Reduced-Order Model. 54–56, 96, 99

RRMSE Relative Root Mean Squared Error. 106, 107, 112, 115, 116

SciML Scientific Machine Learning. 66

SDE Stochastic Differential Equation. 45, 78

SGD Stochastic gradient descent. 15

SGS Sub-Grid Scale. 48, 50–53, 61

SINDY Sparse Identification of Non-Linear Dynamics. 55, 56

SWAG Stochastic Weight Averaging Gaussian. 67

Tanh Hyperbolic Tangent. 14

TBNN Tensor-Basis Neural Network. 49

TKE Turbulent Kinetic Energy. 93, 94, 107, 112, 116, 135, 139, 140, 145–147, 149

U-FNET U - Fourier Network. 128–130, 137–147, 149–151

UDE Universal Differential Equation. 45

UNO U-shaped Neural Operator. 127, 128, 137, 139–142, 150, 151

UQ Uncertainty Quantification. 65–67

VAE Variational Autoencoder. 57, 109

VKP Von-Karman Pao. 90, 92

WENO Weighted Essentially Non-Oscillatory method. 61

WGAN Wasserstein GAN. x, 30, 74–76, 80, 85, 90

Contents

Part I
Fundamentals and Literature Review

1 Introduction

The field of Computational Fluid Dynamics (CFD) is the study and development of numerical methods for solving the Navier-Stokes (NS) equations. The NS equations are non-linear partial differential equations. They can be formulated in several forms depending on the fluid flow problem, leading to a big library of numerical methods that are still growing in the quest to tackle more complex cases, adapt to new computational architectures, and improve the utilization of resources. Recently, machine learning (ML), which can be defined as the collection of methods that leverage data to "teach" an algorithm to perform a specific task, has been disrupting many fields, such as computer vision, Natural Language Processing, recommendation systems, marketing, biology, autonomous driving, medical imaging, etc. outperforming non-data-driven methods in several problems in the aforementioned domains. This bloom in ML has been thanks to the increased data availability and advances in computational hardware that have allowed the training of deep neural networks with billions of parameters.

In fluid mechanics, data-driven methods are well established for analyzing considerable amounts of data from experiments and simulations to understand complex physical behaviors and validate new models. However, even though there is ample availability and intensive use of data in fluid mechanics, the implementation of modern ML algorithms is still lagging behind the results that ML has achieved in other fields.

Machine Learning for Computational Fluid Dynamics (CFD) has gained significant interest in various academic and research communities. This is evident from the numerous articles published in leading fluid mechanics journals, the active involvement of research groups from different universities, workshops conducted at conferences, and the inclusion of courses dedicated to this subject. The intersection of Machine Learning and CFD has generated considerable attention and engagement. According to previously published reviews [28][248][10][54], the attractiveness of ML in CFD is the potential to design algorithms that require less computational resources enabling the use of high fidelity models in scenarios that require many function evaluations as optimization, control, and design; as well as discovering new models for turbulent or multi-physic cases. Simulations nowadays produce big amounts of data that can be leveraged for data-driven modeling and scientific discovery. Although Computational Fluid Mechanics is a scientific discipline that has been practiced

for decades, it still faces some limitations. The first reason is the high cost of computations; high-fidelity numerical simulations require a considerable amount of computational resources running for a significant amount of time, which implies big economic expenses and environmental impact for having the machines running. The complexity of these simulations grows depending on the number of degrees of freedom proportional to the size of the computational domain and the degree of turbulence or other multi-scale behavior, making the DNS of industrial scale flow configurations computationally prohibitive. To overcome this issue, fluid dynamicists rely on modeling complex physical behavior in mathematical forms with lower computational requirements. The main techniques consist in calculating the time-averaged fluid quantities and modeling the fluctuating parts; this is the case for the Reynold-Averaged Navier Stokes (RANS), or by computing the biggest scales and modeling the smaller ones, as in the Large Eddy Simulation (LES) framework. These methods work in many scenarios and make industrial-scale simulations possible. However, there also appears to be a second limitation because modeling the unknown "physics" relies on empiricism and theory. This limits the applicability of some models since processes such as turbulence are not fully understood in fluid mechanics. It is not to be taken lightly what Richard Feynman allegedly said, calling turbulence the most important unsolved problem or classical physics or the reason why the demonstration of existence and smoothness of the Navier-Stokes equations is considered one of the millennium problems by the Clay Mathematics Institute [62][226].

Machine Learning can help in overcoming the challenges that CFD faces by using data from experiments and simulations to extract models to close RANS or LES simulations that could be more precise and generalizable than theoretical models, which provides a framework for the automatic discovery of these models making the process of formulating new models more general and accessible across different scenarios. Another way ML can help CFD is by reducing the computational cost of simulations. With ML, we can reduce the resolution requirements of simulations and let an algorithm correct the error produced by coarsening the computational mesh; the degrees of freedom are reduced, producing a speedup while retaining accuracy. Alternatively, the size of the domain could be reduced by substituting parts of a simulation by Machine-Learned Computations of the flow field like an inflow field generator [278] or by replacing parts of the linear solvers by ML methods that perform these computations faster or even by discovering algorithms that perform faster matrix multiplications [61]. ML can also make CFD more accessible for tasks requiring many function evaluations, such as control, design optimization, and uncertainty quantification, by learning the simulation results and making lightweight surrogate models that retain good accuracy. The advantage of ML for these cases is that once the models are trained, inference time is much faster than running the numerical solver many times.

The objective of this Ph.D. project is to introduce the application of Machine Learning for CFD. Since this thesis is at the intersection of two fields,

we will first introduce in Chapter 2 the general concepts of machine learning and Deep learning, as well as the major applications of Deep Learning, with methods that will appear later in the thesis. In Chapter 3, we will carry out a literature review on the applications of Deep Learning to CFD, with the purpose of assessing and looking for opportunities where DL can help enhance CFD. In Chapter 4, we will implement a Generative Adversarial Network for the generation of synthetic turbulence and a data-driven reduced-order model using supervised learning and neural networks. In Chapter 5, we will implement a Physics-Inspired model to predict turbulent flows and address the problem of achieving numerically stable predictions. Finally, in Chapter 6, we will draw out the general conclusions of the project and future directions.

2 Machine Learning Fundamentals

2.1 Preliminaries

Machine Learning can be defined as any computer program that can learn from experience. Put into a more formal definition [162]:

Definition 2.1 *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.*

Machine learning tasks are how the machine learning system should process an example. An example is a collection of d feature values measured from an object or event. Some of the common tasks solved with machine learning are:

- **Classification:** In this type of task, the computer program is asked to specify which of k categories an example belongs to.
- **Regression:** In this type of task, the computer program is asked to predict a numerical value given an example.
- **Clustering:** Clustering involves categorizing unlabeled data or data points into distinct clusters, where data points with similarities are grouped together, while those that differ from the rest are placed in separate clusters.
- **Anomaly detection:** In this type of task, the computer program sifts through a set of examples and flags some as unusual or atypical.
- **Density estimation:** In the density estimation problem, the machine learning algorithm is asked to learn a function that can be interpreted as a probability density function on the space from which the examples were drawn.

Once the task has been established, a quantitative measure of its performance, denoted as P, is required. The specific performance measure depends on the type of task at hand. In classification problems, accuracy is commonly utilized. Accuracy is the ratio of correctly predicted outputs to the number of examples. In the case of density estimation tasks, it is preferable to employ a metric that assigns a continuous-valued score to each example. For instance, the average log probability assigned by the model to a set of examples can serve as such a metric.

Machine learning algorithms can be broadly classified depending on the type of experience E they have during learning. Two of the most common algorithm types are Unsupervised and Supervised learning. Unsupervised learning experiences a dataset containing many features and then learns useful properties of the structure of this dataset. Usually, an unsupervised learning algorithm learns the probability distribution that generated this dataset. Supervised learning algorithms experience a dataset containing features, but each example is associated

with a label or target. The term supervised learning originates from the view of the target provided by an instructor or teacher who shows the machine learning system what to do. In unsupervised learning, there is no instructor, and the algorithm must learn the underlying patterns in the data.

In the following sections, we will start explaining machine learning more in-depth in the context of supervised learning. We will build from linear model examples and build Deep Learning, which is the main subject of this work. We will cover the main components of a Deep Learning algorithm and how to design and train them.

2.2 Supervised Learning

The goal of supervised learning is to automatically infer a model (hypothesis) from a set of labeled examples that can make predictions given new unlabeled data. Let X be the input space where the data is drawn from, and let Y be the output space that contains the labels or target values. The training set $T = \{(x_i, y_i)\}_{i=1}^{n}$ is a set of n observations independently and identically distributed according to an unknown joint distribution P over the space $Z = X \times Y$. When Y is discrete, we will deal with a classification problem; when Y is continuous, we have a regression problem. Knowing this, we define a supervised learning problem as:

Definition 2.2 ***Supervised Learning*** *is the task of inferring a function, referred to as hypothesis or model, $h : X \to Y$ belonging to some hypothesis class H from a training sample T, that best predicts any y associated with x.*

The hypothesis space we choose should describe, at its very best, the relationship between the spaces X and Y. For example, if we restrict h to be a linear model, the set H comprises the space of all possible linear models. In other words, h should generalize so that for any sample drawn from P, the prediction $h(x)$ should be as close as possible to y. To measure the quality of this prediction, we define a loss function $l : Y \times Y \to \mathbb{R}$ that measures the agreement between $h(x)$ and y. This error between the prediction and labels is associated with the true risk of the model. Thus, the learning process involves finding the hypothesis that minimizes this risk. The true risk is defined as follows:

Definition 2.3 ***True Risk****, also called generalization error $\mathbf{R}_p(h)$, is defined as the mathematical expectation of the loss over the domain P:*

$$\mathbf{R}_p(h) = \mathop{\mathbf{E}}_{(x,y)\sim P} l(h(x), y)$$

The best model h^* is the one that minimizes the true risk. However, we cannot calculate this risk since the probability P is unknown. Only the empirical risk can be calculated on the training samples, which is calculated as follows:

Definition 2.4 *Let T be a training sample. The empirical risk $\mathbf{R}_T(h)$ of a hypothesis h over $T = \{(x_i, y_i)\}_{i=1}^n$ with respect to a loss function l is the average loss suffered by h on the instances in T:*

$$\mathbf{R}_T(h) = \frac{1}{n} \sum_{i=1}^{n} l(h(x_i), y_i)$$

The choice of the hypothesis space H should be wide enough to include hypotheses with small risks but constrained enough to avoid overfitting. Overfitting occurs when a model becomes too complex and starts to fit the noise or random variations in the data rather than the true underlying patterns. Imagine having a large number of data points, each with some level of inherent variability. Suppose we choose a hypothesis space that is overly flexible or complex. In that case, the model may try to account for every variation in the data, including the noise or random fluctuations not representative of the underlying patterns. As a result, the model may perform well on the training data but fail to generalize to new, unseen data. This is why it is crucial to appropriately balance and constrain the hypothesis space.

However, finding the right balance is difficult, especially when dealing with complex data. Complex data exhibits intricate patterns or relationships that are not easily discernible. In such cases, selecting an appropriate hypothesis space becomes more challenging. This is where inductive biases come into play. Inductive biases are assumptions or prior knowledge that guide the learning process and help the model make more informed predictions.

Inductive biases can be based on various factors, such as the properties of the data, symmetries, and invariances. Let's break these down:

1. Data properties: Understanding the properties of the data can provide valuable insights for selecting an appropriate hypothesis space. For example, if the data has a temporal nature, such as time series data, it might be reasonable to assume that past observations are relevant for predicting future ones. This bias towards temporal dependencies can be incorporated into the choice of the hypothesis space, allowing the model to capture the sequential patterns in the data more effectively.

2. Symmetries: Symmetries refer to the inherent structural or geometric properties present in the data. For example, if the data exhibits rotational symmetry, where the patterns remain the same even after rotation, incorporating this bias into the hypothesis space can help the model recognize and leverage such symmetries. This can lead to more efficient and accurate representations of the data.

3. Invariances: Invariances are the properties of the data that remain unchanged under certain transformations. For instance, if the data represents images, there might be invariances to translation, where shifting an object within the image does not change its identity. By incorporating the appropriate inductive biases related to invariances, the hypothesis space

can be tailored to capture these properties, enabling the model to make robust predictions despite such transformations in the data.

By considering inductive biases based on the data properties, symmetries, and invariances, we can improve the selection of the hypothesis space. These biases help guide the model towards more relevant and meaningful hypotheses that are more likely to capture the important aspects of the data. As a result, the model selection process becomes more effective, leading to better performance and generalization capabilities.

Other types of risk minimization that are commonly used are:

Definition 2.5 ***Structural Risk Minimization.*** *In Structural Risk Minimization, we can use an infinite sequence of hypotheses classes $H_1 \subset H_2 \subset ...$ of increasing size and select the hypothesis that minimizes a penalized version of the empirical risk that favors "simple" classes:*

$$h_T = \underset{h \in H_c, c \in \mathbb{N}}{\arg\min} R_T(h) + \text{pen}\,(H_c)$$

Definition 2.6 ***Regularized Risk Minimization.*** *Regularized risk minimization is easier to implement than SRM. Here, one picks a single, large hypothesis space H and a regularizer (usually some norm $||h||$ and selects a hypothesis that achieves the best trade-off between empirical risk minimization and regularization:*

$$h_T = \underset{h \in H}{\arg\min} R_T(h) + \lambda \|h\|$$

Where λ is the trade-off parameter, the role of regularization is to penalize "complex" hypotheses. In a broader sense, regularization is any modification to a learning algorithm intended to reduce its generalization error but not its training error. The best model resulting from risk minimization is the one that can achieve optimal capacity. This is the point where the difference between test and training error is the lowest (as seen in fig. 1) [74].

To train an ML model in a supervised learning setting, we need a performance measure or associated cost we want to minimize. This loss function depends on the type of task we want to train our model on. Some of the most common loss functions are:

- **Mean squared error (MSE)**: The mean squared error is often used for regression problems using neural networks. Using mean-squared error can be justified as estimating our model's maximum likelihood. We want to find the model parameterized by θ that maximizes the probability of finding the target $y \in Y$ given $x \in X$. This can be cast as:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P\left(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right). \tag{1}$$

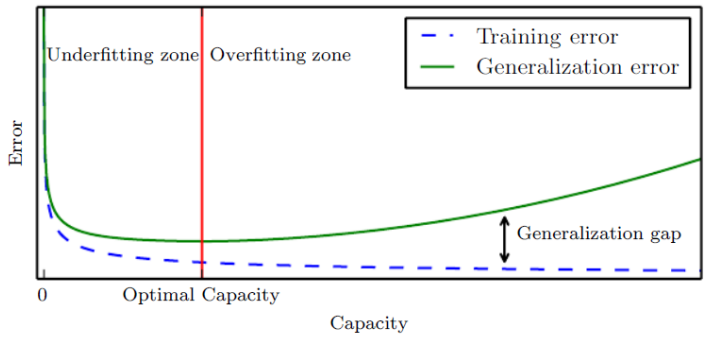


Figure 1: Relationship between capacity and error. In the left section of the graph, training error and generalization error are high; this is the underfitting regime. As capacity increases, the gap between training and generalization errors increases, resulting in overfitting. Figure from [74].

For regression we can define $p(y|x) = \mathcal{N}(y; f(x;\theta), \Sigma^2)$, then maximization of the conditional log-likelihood becomes:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P\left(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) = -\frac{1}{m}\sum_{i=1}^{m} \left\| f^{(i)}(x;\theta) - y^{(i)} \right\|^2 + const. \tag{2}$$

Which is equivalent to minimizing the negative of this function:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{m}\sum_{i=1}^{m} \left\| f^{(i)}(x;\theta) - y^{(i)} \right\|^2 = MSE \tag{3}$$

The constant that appears is discarded since it depends on the variance of the Gaussian distribution and not the model parameters. We can observe that regression with the mean squared error is equivalent to finding the model that best predicts the mean of the multi-variate Gaussian over the training data if this one is i.i.d.

- **Cross-Entropy**: Minimizing the cross-entropy is equivalent to minimizing the negative log-likelihood:

$$l(x, y; \boldsymbol{\theta}) = -\mathop{\mathbb{E}}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\mathrm{data}}} \left[\log p_{\mathrm{model}}\left(\boldsymbol{y} \mid \boldsymbol{x}\right)\right] \tag{4}$$

We can minimize this cross-entropy without assuming the shape of the p_model. This is preferred on classification problems that contain two or more classes. For problems with C classes where the output of the model $f(x;\theta)$ represent a label prediction:

$$l_n = -\sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} w_{y_n} \log \frac{\exp\left(f(x;\theta), y_n\right)}{\sum_{c=1}^{C} \exp\left(f(x;\theta), c\right)} \tag{5}$$

Where w_{y_n} is a weight assigned to each class, This function is equivalent to applying the softmax function to the model's output and calculating the negative log-likelihood. For problems with Bernoulli-type output distributions, the loss is called **Binary Cross-Entropy**:

$$l_n = -\frac{1}{N} \sum_{n=1}^{N} \left[y_n \cdot \log f(x;\theta) + (1 - y_n) \cdot \log\left(1 - f(x;\theta)\right)\right] \tag{6}$$

2.3 Deep Learning

Deep learning is a subfield of machine learning that focuses on training and building artificial neural networks to learn and make intelligent decisions from large amounts of data. It is inspired by the structure and functioning of the human brain, particularly the interconnections of neurons.

Recently, Deep learning has achieved remarkable success in various fields, including computer vision, natural language processing, speech recognition, and reinforcement learning. It has enabled breakthroughs in image classification, object detection, machine translation, sentiment analysis, etc. The ability of deep learning models to automatically learn hierarchical representations from raw data has greatly contributed to their effectiveness in solving complex problems and surpassing the performance of traditional machine learning approaches.

Being a sub-field of Machine Learning, the aim of Deep Learning remains still making predictions or taking actions based on data. Still, some fundamental characteristics make Deep Learning different than other methods:

- Representation of Features: In traditional machine learning, feature engineering plays a crucial role. Human experts manually extract and select relevant features from the input data, which are then used to train the machine-learning model. These features are typically handcrafted and require domain knowledge. On the other hand, deep learning algorithms automatically learn hierarchical representations of the data through multiple layers of artificial neurons. Deep learning models have the ability to learn and extract relevant features from raw or high-dimensional data, eliminating the need for extensive feature engineering.
- Complexity and Scalability: Deep learning models are typically deeper and more complex than traditional machine learning models. Deep neural networks can have numerous layers, each containing many neurons. This increased complexity allows deep learning models to capture intricate patterns and relationships in the data. However, this complexity also requires

more computational resources and data to train effectively. Traditional machine learning models, in comparison, are often more straightforward and require less computational power.

- Data Requirements: Deep learning models often require large amounts of labeled data for training. Since deep learning models learn representations directly from the data, having sufficient labeled examples is essential to perform well. In contrast, traditional machine learning models can perform well with smaller datasets and may rely more on the quality of the features provided.

- Interpretability: Traditional machine learning models provide more interpretability and explainability than deep learning models. This is because the features used in traditional machine learning are often explicitly defined and understood by humans. In deep learning, the learned representations are more abstract and may be challenging to interpret. However, efforts are being made to develop techniques for interpreting and understanding deep learning models.

- Performance: Deep learning models have demonstrated remarkable performance in various domains, such as image and speech recognition, natural language processing, and playing complex games. They have achieved state-of-the-art results in many tasks. Traditional machine learning models are still effective in many scenarios, especially when dealing with smaller datasets or problems with well-defined features.

2.3.1 Motivation: Deep Representation Learning

In ML, representation learning is the set of techniques that allow the techniques for learning representations of the data that make it easier to extract useful information when building classifiers or other predictors from raw data. This replaces the need for feature engineering for ML algorithms to perform a determined task.

Deep learning and representation learning are closely interconnected. Deep learning can be seen as a specific approach to representation learning. Deep learning, specifically through deep neural networks, can automatically learn hierarchical representations of data by stacking multiple layers of nonlinear transformations. These deep representations capture complex and abstract features at different levels of abstraction, enabling the model to learn more intricate patterns and structures within the data. Therefore, deep learning can be viewed as a powerful technique within the broader field of representation learning, as it enables the automatic discovery and extraction of hierarchical representations from raw input data.

Several factors characterize a good representation. Good representations are expressive, meaning a learned representation of reasonable size can capture

many possible input configurations. Another desired property is feature reuse; deep architectures lead to abstract representations that are invariant of local features of the inputs allowing the re-use of different parts of the architecture. Finally, the disentanglement of a factor of variation allows us to distinguish the different elements in the data that interact with each other. For example, in an image of a face, a disentangled representation would ideally have separate dimensions or components that encode the person's identity, pose, facial expression, lighting conditions, and other relevant factors. Each component would capture a single aspect of the input data, enabling more flexible and controllable manipulation of the learned representation [13].

Deep Learning has achieved greater state-of-the-art performance in various ML tasks. One of the factors of this success is the ability of DNN to learn semantically rich latent representations for different types of data, which allows for capturing very complex concepts, which cannot be done through handcrafted features. The following subsections will show the most relevant algorithms for diverse data modalities and tasks.

2.4 Neural Networks

At the core of Deep Learning, the hypothesis is represented with a neural network. Neural networks provide a general practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Neural Networks nowadays are the center of many AI breakthroughs because of their generalization capabilities and the power to adapt to many tasks and problems. Training NNs on large-scale datasets has also become very efficient on modern computational hardware, providing fast inference time and easy parallel training on multiple devices [264]. The following sections will explain neural networks and how to build and train them.

2.4.1 Perceptrons

The term neural network originates in the attempts to find mathematical representations of information processing in biological systems [212]. The study of NNs has been partly inspired by the observation that biological learning systems are built of very complex webs of interconnected neurons. Artificial Neural Networks (ANN) are built out of a densely interconnected set of simple units, where each unit takes several real-valued inputs and produces a single real-valued output. While artificial neural networks are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs are inconsistent with biological systems [162].

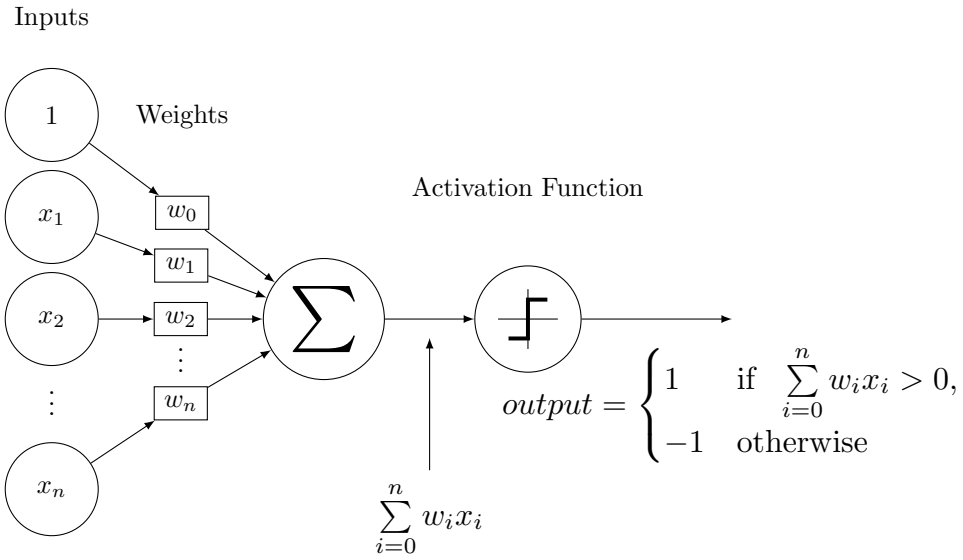


Figure 2: A perceptron.

The first NNs were built around units called Perceptrons. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, and then outputs one if the result is greater than some threshold and -1 otherwise. More precisely, given a set of inputs $\mathbf{x} = \{x_1, ..., x_n\}$ the output of the perceptron is:

$$f(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases} \tag{7}$$

A graphical description of the perceptron can be seen in Fig. 2. Each input vector element is associated with a weight w_n aggregated by a linear operator, in this case, a sum. The output then passes through an activation function.

A single perceptron can be used to represent many boolean functions. For example, if we assume the values to be 1 (True) and -1 (False), and using a two input value perceptron, if the weights are $w_= -0.8$ and $w_1 = W_2 = 0.5$ we have the AND function, if we set the value of $w_0 = 0.3$ we have the OR function. Perceptrons can represent all primitive boolean functions, but some require more than one perceptron. In fact, The ability of perceptrons to represent primitive boolean functions means that an interconnected network of these units can represent every boolean function. This motivates the development of multi-layer perceptrons or feedforward neural networks, where each network weight belongs to the set of parameters to be learned during training.

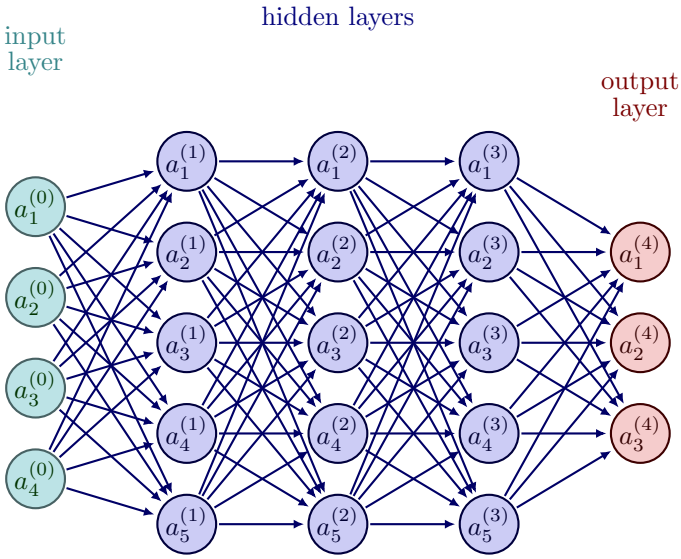


Figure 3: Feedforward Neural Network diagram.

2.4.2 Feedforward Neural Networks

Feedforward neural networks (FFN) or multilayer perceptrons (MLP) are among the most common models or architectures used in Deep Learning. These models are called feedforward because information flows through the function evaluated from x, through the intermediate computations used to define f, and finally to the output y. Feedforward neural networks are called multi-layer perceptrons because they can be seen as many perceptrons stacked together in a series of layers. In fact, FFN is a composition of several intermediate functions, and the length or amount of layers gives the notion of depth of the network. In contrast, the dimensionality of each layer (how many weights it has) is the width of the network. The first layer has the dimensionality of the input, and the last layer (output layer) is the one of the output. The intermediate layers are of higher width than the input and output layers. They are called hidden layers because their individual output is not seen or is of particular interest for practical applications. These parts of a FFN are illustrated in Fig. 3.

Neural networks are non-linear by construction. Neural networks are non-linear models because of the activation functions used in their neurons. Activation functions introduce non-linearity into the computations performed by the neural network. In a neural network, each neuron receives inputs from the previous layer, applies a weighted sum to those inputs, and then passes the result through an activation function. This activation function introduces non-linearity by mapping the weighted sum to a different range, transforming the output non-linearly. If neural networks used only linear activation functions,

such as the identity function, the network would collapse into a single linear layer and lose expressiveness. Some of the most common activation functions used in practice are:

- **Rectified linear units (ReLU)**: rectified linear units use the activation

$$g(z) = \max\{0, z\}$$

. The difference between a linear unit and a RANS is that the latter outputs zero over half its domain. The RANS and its variants are normally the default choice for hidden layers because they are easier to optimize using gradient-based optimization methods since its derivative is always one on the regions the unit is active and the second derivative is 0. Another type used often is leaky ReLU

$$g(z) = \max\{\alpha z, z\}$$

Where α is a small slope on the negative parts.

- **Hyperbolic tangent (Tanh)**: this activation function ($g(z) = \tanh(z)$) was used often before the adoption of ReLU, this function saturates to a high value when z is very positive, saturate to a low value when z is very low and is very sensitive when the input is around 0. Although ReLUs are preferred in most cases because of the lack of saturation, tanh is suitable when a probabilistic output is expected or for gating operations in recurrent neural networks (which we will see later).

- **Sigmoid Units**: This function

$$f(x) = \frac{1}{1 + e^{-x}}$$

is used for output layers when the problem is a binary classification one when the neural network needs to predict $P(y = 1 \mid \boldsymbol{x})$.

- **Softmax Units**: Softmax units represent a probability distribution over a discrete variable with n possible values. These are often used as the output of a classifier to represent the probability distribution over n different classes. The Softmax function is given by:

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

. Softmax is used almost always as an output activation unit. Still, it has appeared as the activation on recent neural networks called transformers to help capture long-range dependencies in data.

- **Swish**: The swish activation

$$f(x) = x \cdot \text{sigmoid}(\beta x).$$

was found via an empirical study that consisted on finding better alternatives to ReLUs [200], motivated on improving the training dynamics and accuracy of deep models.

- **Gaussian error linear unit (GELU)**: The GELU activation is represented as

$$\text{GELU}(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}[1 + \text{erf}(x/\sqrt{2})]$$

Where $\phi(x)$ is a cumulative gaussian distribution. This nonlinearity is an alternative to ReLUs and is a substitute for them when combined with other methods for stochastic regularization [85]. This improves training convergence and may improve accuracy in some cases.

2.5 How to train Neural Networks

In this section, we explain how Deep Learning algorithms are trained. The main goal of training a neural network is to find the optimal set of parameters that minimize the cost associated with a specific task. Much of the general procedure is the same as other ML algorithms, having a training dataset for our desired task, a cost function we want to minimize, and our model, which is the neural network. We will explain these components, what happens under the hood of this process, what tools we use, and some considerations that need to be considered.

2.5.1 Learning with Gradient Descent

Stochastic gradient descent (SGD) and its variants are the prevalent optimization algorithms for training most deep learning models. Stochastic gradient descent is an extension of the gradient descent algorithm. A challenge for deep learning is that large training datasets are needed for good generalization, but large training sets are also more computationally expensive. The loss function used by a machine learning algorithm is often decomposed as a sum over training examples of a per-example loss function:

$$J(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\text{data}}} l(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} l\left(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}\right) \tag{8}$$

Where $L(\boldsymbol{x}, y, \boldsymbol{\theta})$ is the per example loss. The problem of finding the best parameters that define the best ML model corresponding to a dataset is a minimization problem. A classical method to solve this type of problem is Gradient descent, which requires computing the gradient of the objective function w.r.t. the parameters:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} l\left(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}\right) \tag{9}$$

The computational cost of this operation is $O(m)$. As the size of the training set grows to billions of examples, the time to perform a single gradient step becomes prohibitively long. The insight of SGD is that the gradient is an expectation. The expectation may be appropriately estimated using a small set of examples. On each step of the algorithm, we can sample a minibatch of examples $\mathbb{B} = \left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m')}\right\}$ drawn uniformly from the training set. The minibatch m' is chosen to be relatively small, ranging from 1 to a few hundred examples. The estimate of the gradient is:

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L\left(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}\right) \tag{10}$$

Using examples from the minibatch $\mathbb{B}$, the stochastic gradient descent algorithm then follows the estimated gradient downhill:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}, \tag{11}$$

With ϵ being the learning rate [74].

2.5.2 The engine: Automatic Differentiation

When a feedforward neural network is used, information propagates from the input x through each layer up to the output $\hat{y}$, called forward propagation. This forward propagation continues over training until it produces a scalar cost $J(\theta)$. The back-propagation algorithm [212] allows information from the cost to flow backward through the network to compute the gradient and to update the parameters accordingly.

The backpropagation algorithm corresponds to the reverse mode of automatic differentiation. Automatic differentiation is a set of techniques to compute partial derivatives of a function specified by a computer program. Let $f_1, ..., f_n$ be for which we know how to calculate the derivatives. For any composition of these functions $f(x) = f_{i_m}(\cdots(f_{i_1}(x))\cdots)$, with $i_1, \ldots, i_m \in \{1, \ldots, n\}$, we also know how to compute the derivative of f via the chain rule:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f_{i_m}}{\mathrm{d}f_{i_{m-1}}} \cdots \frac{\mathrm{d}f_{i_2}}{\mathrm{d}f_{i_1}} \frac{\mathrm{d}f_{i_1}}{\mathrm{d}x} \tag{12}$$

There are mainly two approaches to computing these derivatives using automatic differentiation. Forward-mode auto differentiation, also known as forward sensitivity, proceeds by recursive computing :

$$\frac{\mathrm{d}f_{i_q}}{\mathrm{d}x} = \frac{\mathrm{d}f_{i_q}}{\mathrm{d}f_{i_{q-1}}} \frac{\mathrm{d}f_{i_{q-1}}}{\mathrm{d}x} \tag{13}$$

For $q = 2, ..., m$. Reverse mode auto differentiation or backpropagation proceeds by recursive computing:

$$\frac{\mathrm{d}f_{i_m}}{\mathrm{d}f_{q-1}} = \frac{\mathrm{d}f_{i_m}}{\mathrm{d}f_{i_q}} \frac{\mathrm{d}f_{i_q}}{\mathrm{d}f_{q-1}} \tag{14}$$

For $q = m-1, ..., 1$ and supposing that x is a vector and f_{i_m} outputs a scalar. As with all hidden layers of an NN outputs vectors, each evaluation of the forward mode auto differentiation is a matrix-matrix product. At the same time, the evaluation in reverse-mode auto differentiation is a vector-matrix product that is computationally cheaper. For this reason, backpropagation is used to compute the gradients of neural networks [106].

2.5.3 Improvements to Stochastic Gradient Descent

Stochastic gradient descent can sometimes be slow; the momentum method is designed to accelerate learning in the face of high curvature, small but consistent gradients, or noisy gradients [189]. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. According to Newton's laws of motion, momentum derives from a physical analogy in which the negative gradient is a force moving a particle through parameter space. Assuming unit mass, the velocity vector is the particle's velocity in the momentum algorithm. A hyper-parameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay. The update rule then is given by [74]:

$$\begin{aligned} \boldsymbol{v}_n &\leftarrow \alpha \boldsymbol{v}_{n-1} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L \left(\boldsymbol{f} \left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} \right), \boldsymbol{y}^{(i)} \right) \right) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{v}_n \end{aligned} \tag{15}$$

A variant of momentum is also used called Nesterov's momentum [236][175]. The update rule is given in this case by:

$$\begin{aligned} \boldsymbol{v} &\leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L \left(\boldsymbol{f} \left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v} \right), \boldsymbol{y}^{(i)} \right) \right], \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{v}, \end{aligned} \tag{16}$$

The difference between Nesterov and standard momentum is where the gradient is evaluated. Nesterov momentum evaluates the gradient after the current velocity is applied. Nesterov momentum brings the rate of convergence from $O(1/k)$ to $O(1/k^2)$.

The learning rate is one of the most difficult hyperparameters to set because it significantly impacts model performance. The loss function is often highly sensitive to some directions in parameter space and insensitive to others. The momentum algorithm can mitigate these issues but introduces another hyperparameter to set. Optimization algorithms with adaptive learning rates are often used in modern machine learning.

One of the most popular methods used in Deep Learning is the Adam (adaptive moment estimation) algorithm [113]. The Adam algorithm updates exponential moving averages of the gradient (m_t) and the raw second-order moment (v_t), which is the gradient squared. The hyperparameters $\beta, \beta \in (0, 1]$ control the exponential decay rates of these moving averages. Let ϵ be a small factor to prevent zero-division, t the time-step, and α the learning rate. The Adam's update rule is given by:

$$\begin{aligned} g_t &= \nabla_\theta f_t(\theta_{t-1}) \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= m_t / (1-\beta_1^t) \\ \hat{v}_t &= v_t / (1-\beta_2^t) \\ \theta_t &= \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) \end{aligned} \tag{17}$$

Another adaptive learning rate method is Adadelta [279]. Adadelta builds over two ideas; the first is accumulating the sum of squared gradients over a fixed window of previous iterations, and the accumulated gradient is used as a denominator to scale the learning rate. Assuming that at time t the running average is $E\left[g^2\right]_t$ then it is computed:

$$E\left[g^2\right]_t = \rho E\left[g^2\right]_{t-1} + (1-\rho) g_t^2 \tag{18}$$

Where ρ is a decay constant similar to momentum. The square root of this quantity is used in order to keep the update rule consistent dimensionally, being ϵ a constant for better condition and η the learning rate we have the parameter update as:

$$\begin{aligned} \text{RMS}[g]_t &= \sqrt{E\left[g^2\right]_t + \epsilon} \\ \Delta\theta_t &= -\frac{\eta}{\text{RMS}[g]_t} g_t \end{aligned} \tag{19}$$

The second idea is to correct the units of the gradient update. When we update the parameters using first-order gradients, the dimensions of these are different than those of the parameters themselves:

$$\text{units of } \Delta\theta \propto \text{ units of } g \propto \frac{\partial f}{\partial \theta} \propto \frac{1}{\text{units of } \theta} \tag{20}$$

Second-order methods that use Hessian information or an approximation have correct units:

$$\Delta\theta \propto H^{-1} g \propto \frac{\frac{\partial f}{\partial \theta}}{\frac{\partial^2 f}{\partial \theta^2}} \propto \text{ units of } \theta \tag{21}$$

This concept is applied in Adadelta by assuming a diagonal Hessian so it can be approximated as:

$$\Delta\theta = \frac{\frac{\partial f}{\partial \theta}}{\frac{\partial^2 f}{\partial \theta^2}} \Rightarrow \frac{1}{\frac{\partial^2 f}{\partial \theta^2}} = \frac{\Delta\theta}{\frac{\partial f}{\partial \theta}} \tag{22}$$

Substituting this in the update rule of eq. 19:

$$\begin{aligned} E\left[\Delta\theta^2\right]_t &= \rho E\left[\Delta\theta^2\right]_{t-1} + (1-\rho)\Delta\theta_t^2 \\ \text{RMS}[\Delta\theta]_t &= \sqrt{E\left[\Delta\theta^2\right]_{t-1} + \epsilon} \\ \Delta\theta_t &= -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g]_t} g_t \end{aligned} \tag{23}$$

Other modern variants of adaptive learning rate methods exist, such as AdaBelief [285], which adapts the learning rate on the "belief" of the current gradient direction, resulting in faster convergence and improved stability. Another example is the rectified Adam (RAdam) [150] that rectifies the variance of Adam algorithm.

Deep Learning optimizers continue to evolve as neural networks get larger and more complicated to train. Nevertheless, Adam and its variants remain the default option, and sometimes very little performance improvement can be seen when using different optimizers. These optimizers are of the type of first-order optimization techniques since the computation of the Hessian is avoided. Second-order optimization methods are not used in Deep Learning because they have high computational and memory costs and are more difficult to implement for distributed training.

2.5.4 Regularization for Deep Neural Networks

Regularization is the collection of strategies used to reduce the overfitting of machine learning models. In other words, these strategies aim to reduce the test error, making the model perform better on new inputs different than the training data.

Many regularization strategies are based on limiting the model complexity of models by adding a term $\Omega(\theta)$ to the objective function J to penalize the most complex models. The regularized objective function becomes $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda\Omega(\boldsymbol{\theta}) \tag{24}$$

Where λ is a hyperparameter that weights the relative contribution of the norm penalty term Ω relative to the standard objective function J. One of the simplest kinds of parameter norm penalty is the L^2 regularization, commonly known as weight decay. This makes the regularized loss function using the L^2 norm look like this:

$$\tilde{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda\|\theta\|^2 \tag{25}$$

Another option is to use the L^1 norm:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \lambda \|\theta\| \tag{26}$$

The effect of weight decay is that it makes the neural network's weights smaller, preventing overfitting since the result will be a function with less active neurons where only the more important weights are active. The difference between L^1 and L^2 regularization is sparsity. L^1 regularized models will have weights that are equal to zero. For this reason, L^1 regularization is preferred if we have many features we want to reduce because unimportant features will be zeroed out. In contrast, L^2 regularization will make the contribution of less important features smaller but not zero, and this is preferred if interdependence exists between the inputs or our model.

2.5.5 Normalization and Data-Augmentation

Other techniques to improve convergence and generalization are independent of the optimization algorithm and objective function used. In this section, we will discuss two of the most used in Deep Networks to mitigate overfitting: Normalization and Data Augmentation.

- **Normalization**:
  The first method that introduced normalization in deep neural network layers was Batch Normalization (BN) [93]. Deep models involve the composition of several functions or layers. One of the difficulties in training deep neural networks is that the inputs to each layer are affected by the parameters of all the preceding layers. This means that (small) changes in parameter values tend to be amplified as the network becomes deeper. The change in the distribution of the inputs as they pass through each layer is a problem because the layers need to adapt to the new distribution continuously. When the distribution of a learning system changes, it is said to experience a covariate shift. Batch Normalization reduces a neural network's internal covariate shift by fixing the layer inputs' distribution x as the training progresses. Batch normalization normalizes each input feature independently with a mean of zero and variance of 1. Normalization changes what the input may represent, so two learnable parameters γ, β are introduced to scale and shift the normalized value. The Batch Normalization transform is implemented as follows:

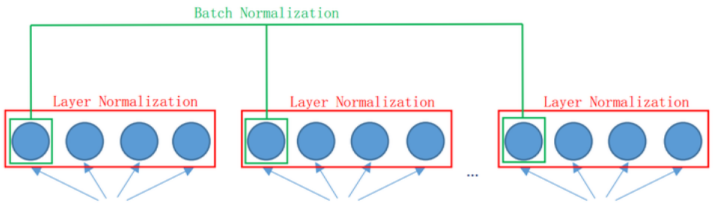


Figure 4: Difference between BatchNorm and LayerNorm: While batch Norm computes the statistics of each activation over the batch size, LayerNorm statistics are calculated over all activations for each batch element. Source [9]

$$
\begin{aligned}
\mu_{\mathcal{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \\
\sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m} \left(x_i - \mu_{\mathcal{B}}\right)^2 \\
\widehat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\
y_i &\leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}\left(x_i\right)
\end{aligned}
\tag{27}
$$

Batch Normalization accelerates the training of many neural network models. However, it doesn't mean that it will improve every deep learning architecture. BN depends on the mini-batch size, and in other models like recurrent models, the statistics also depend on the sequence length and time-step. A technique to mitigate this issue is Layer Normalization (LN) [4], which modifies BN by estimating the normalization statistics from the summed inputs to the neurons within a hidden layer. The main difference between BN and LN is how the statistics are calculated. BN computes the mathematical expectations of each feature over the minibatch, and LN computes the expectations of the hidden features of each layer independently of mini-batch size; see Fig. 4 for a graphical explanation.

Another technique inspired by LN is Group Normalization GN [268]. This method aims to generalize to smaller batch sizes where BN had little effect by avoiding batch computation. However, unlike LN aggregating all input features, GN divides the features into groups. This is useful for computer vision models where groups of features may have similar statistics. The difference between different normalization methods can be seen in Fig. 5.

- **Data-Augmentation:** Data-augmentation is a method to approach overfitting from the training dataset. This is done under the hypothesis that augmentations can extract more information from the original dataset. The purpose of these augmentations is to make the model more robust to possible transformations to the inputs that they may encounter during

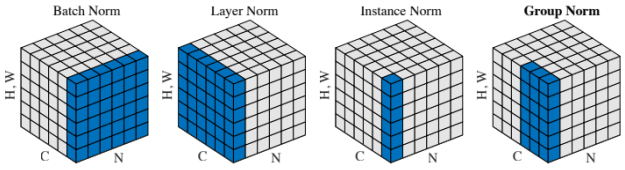


Figure 5: Normalization methods. Each cube shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels [268].

operating conditions. Data augmentation can be categorized as warping augmentations, which consists of applying transformations to the data, like rotations, scaling, shifting, etc. And oversampling, which consists of generating additional synthetic data that is similar to the dataset.

2.6 Limitations of Deep Learning

Deep Learning has proven to be a powerful tool in AI, but it still has some limitations:

1. Data requirements: Deep learning algorithms require large amounts of data to train, and collecting and annotating this data can be time-consuming and costly. Since the lack of theoretical knowledge of deep neural networks, it is difficult to predict how much data is needed to train a model to perform a certain task. Theoretical work demonstrated the difficulty of constructing classifiers in problems with small sample sizes [202][82]. This can be a limitation for applications where data is scarce or difficult to obtain.

2. Overfitting: Deep learning models tend to overfit the training data, which means they may perform well on the training data but not on new, unseen data. The recent trend in Deep Learning is to use over-parameterized models. This can be seen in recent large language models that showcase state-of-the-art performance but contain billions of parameters. Experience and theory have demonstrated that over-parameterization can improve the model's performance [60][31]. However, increasing the model complexity increases the need for data, making the model overfit more easily when provided with insufficient data samples. This is particularly a problem when the training data is limited.

3. Lack of interpretability: Interpretability is a concept that is defined differently by different authors [148][170][201]. Interpretability can be defined as the property a Machine Learning model has to be explained. Interpretable

properties can be classified in terms of transparency and post-hoc explanations [148]. Transparency is the understanding of the mechanisms by which the model works. Transparency can be considered at different levels: At the level of the entire model (Simulability), at the level of its parts (decomposability), and at the level of the training algorithm (algorithmic transparency). Post-hoc explanations refer to explanations provided after the model is trained. Post-hoc explanations include language explanations, visualization of representations, and explanations by example that consider examples the model considers to be similar. Interpretability, in general, is important because it aids in trust, safety, and contestability of ML models. With an interpretable model, we can assess in which conditions the model will work and when not, also providing the ability to reject certain decisions it makes.

4. Computational resources: Training and running deep learning models require significant computational resources, which can be a limitation for certain applications or organizations with limited budgets. Recent advances in Deep Learning have been made possible by powerful hardware configurations, which implies significant economic costs [240]. In fig. 6, a comparison of the growth of DL model size versus the growth in hardware performance can be seen.

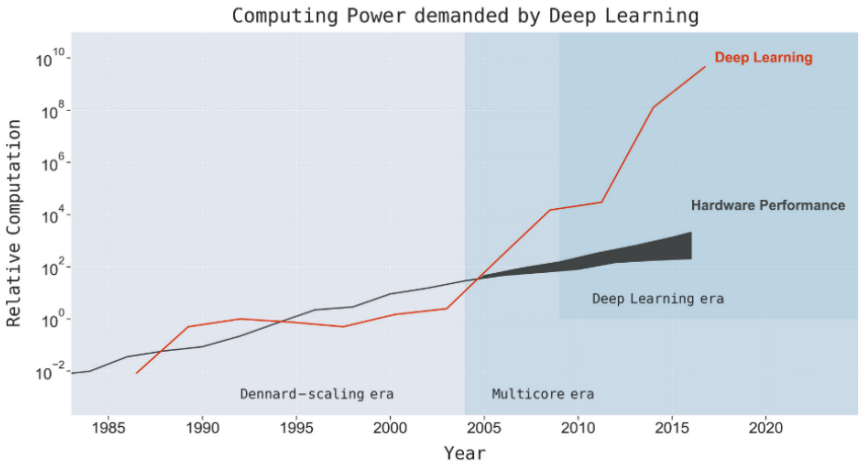


Figure 6: Computing power used in the largest DL model of the correspondent year vs. the growth in hardware performance. Picture from [240].

Considering these limitations when applying any ML technique to a problem of our interest is important. Tackling these issues leads to advancements and new findings in the field.

2.7 Deep Learning Algorithms

2.7.1 Deep Learning for Computer Vision

Convolutional Neural Networks

Convolutional networks, also known as convolutional neural networks or CNNs, are specialized neural networks for processing data with a known, grid-like topology. A typical example is image data, which can be considered as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name convolutional neural network indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are neural networks that use convolution instead of general matrix multiplication in at least one of their layers.

In its most general form, convolution is a mathematical operation of two functions, f and g, that produces a third function $(f * g)$ that expresses how the shape of one is affected by the other. The convolution is defined as the integral of the product of the two functions after one is reflected about the y-axis and shifted:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau \tag{28}$$

When we work with images in DL, this data is represented as a discrete array of values distributed in a uniform grid. In other words, the convolution for discrete functions of n dimensions is given by:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n - m] \tag{29}$$

The convolution of two finite sequences is defined by extending the sequences to finitely supported functions on the set of integers. In the case of image processing, the functions f and g are discrete functions of 2 dimensions, so their convolution would be:

$$(f * g)[n_1, n_2] = \sum_{m_1=-\infty}^{\infty} \sum_{m_2=-\infty}^{\infty} f[m_1, m_2]g[n_1 - m_1, n_2 - m_2] \tag{30}$$

Figure 8 shows the convolution operation performed over a discrete-valued grid.

For processing images, we take f, a matrix with pixel values that represent an image, and g would be a smaller matrix called kernel that can be used for blurring, edge detection, or other types of filtering operation. Figure 7 illustrates how a convolution operation can result in specific effects on images with

a well-chosen kernel (filters).

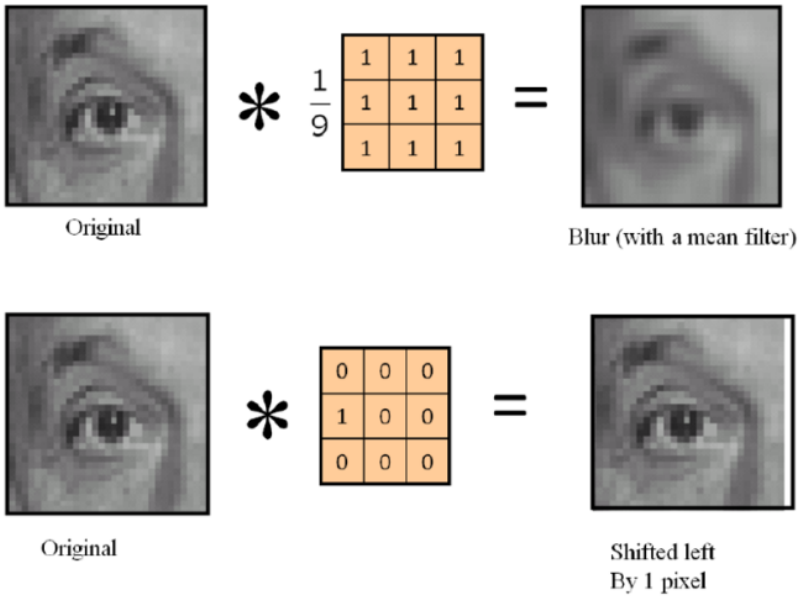


Figure 7: Effects of kernels on an image. From CS1114: Introduction to computing using MatLab and robotics [47].

The underlying principle of convolutional neural networks (CNN) lies in their utilization of trainable kernels to detect significant data representations pertinent to a specific task. A typical convolutional layer has three stages (see fig.8). In the first stage, the layer performs several convolutions in parallel to produce a set of feature maps. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage. In the third stage, a downsampling or upsampling function is used to further modify the layer's dimensionality. This down/up-sampling can be done implicitly with the convolution or pooling function. A pooling function replaces the layer's output at a certain location with a summary statistic of the nearby outputs. Pooling helps make the representation approximately invariant to small translations of the input. Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.

A typical CNN for image classification would have a feature detection stage that contains a hierarchy of convolutional layers that gradually learn features at different scales and a classification stage where a FFN produces the desired

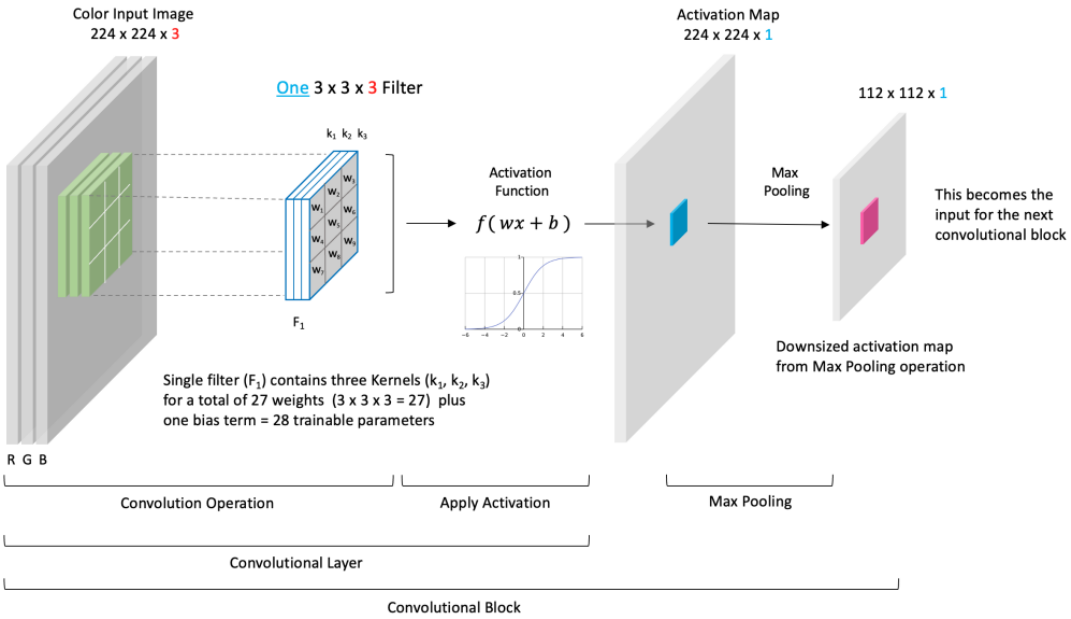


Figure 8: The components of typical convolutional neural network layer [245]

output based on the learned representations, a classic model that uses this framework is VGG-16 [229] (see fig. 9). Modern implementations of CNNs have had great success in solving computer vision tasks such as classification and object recognition of large image datasets like ResNets [83][238] and image segmentation like U-Net [208].

Autoencoders

Autoencoders are a ML method belonging to the unsupervised learning algorithms family. In unsupervised learning, we do not need labeled data. The main goal is to learn a semantic representation. In autoencoders, the objective is to learn a latent representation of the data, a parsimonious set of variables that relate to the features present in the image. This latent variable is normally of less dimensionality than the data. For this reason, autoencoders themselves are viewed commonly as data compression algorithms. Autoencoders also learn a latent representation of the governing features of the dataset, meaning that they can also be used as a denoising method. Last but not least, the latent representation learned by autoencoders can be used as a component on much larger models since the latent variables facilitate learning as they represent the feature density function of the data.

The architecture of AEs comprises two subnetworks: the encoder, which

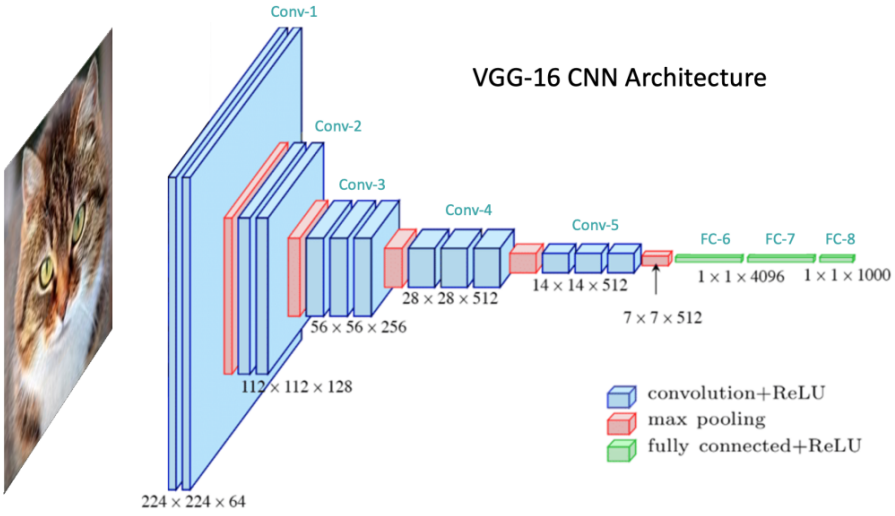


Figure 9: VGG-16 architecture for image recognition [245][229].

encodes the input to the latent variables, and the decoder, which reconstructs the input data from the latent variables. Being $x \subset X \in \mathbb{R}^{d_x}$, a sample from the dataset, $\mathcal{E}(\cdot;\theta)$ the encoder network with parameters θ and $\mathcal{D}(\cdot;\psi)$ the decoder network with parameters ψ, the latent variable is the output of the encoder which is:

$$z = \mathcal{E}(x;\theta) \tag{31}$$

Where $z \subset Z \in R^{d_z}$. The AEs are trained by minimizing the reconstruction loss of the Encoder-Decoder pair:

$$(\theta,\psi) = \arg\min_{\theta,\psi} ||x - \hat{x}||_2^2 \tag{32}$$

$$\hat{x} = (\mathcal{D}(\cdot;\psi) \circ \mathcal{E}(\cdot;\theta))(x) \tag{33}$$

The most basic autoencoder is built using feed-forward neural networks. As seen in Fig. 10, the encoder and decoder networks are neural networks with layers that change the representation size sequentially. The autoencoder's goal is generally that the reconstruction is the same as the input and can generalize to unseen samples of the same type as the training data. Autoencoders are very flexible. Modifying the loss function can train them for segmentation, super-resolution, or denoising. Also, the neural network used can be changed to suit the inductive biases of the data. For example, one can use convolutional layers to form convolutional autoencoders (see fig. 11 better suited for image-related tasks, or we can use RNNs for sequences [232].

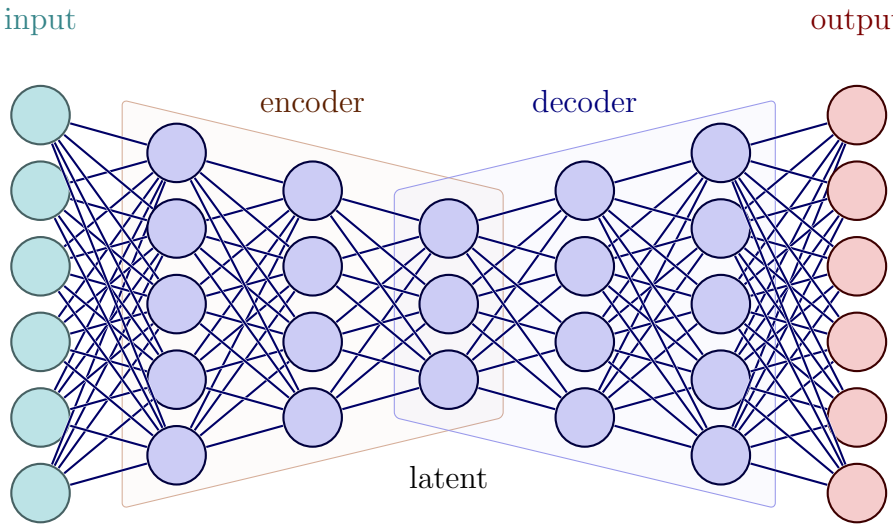


Figure 10: Schematic of an autoencoder. The input reduces its size at each layer until it arrives at a reduced latent representation. From there, the decoder maps the latent variable to the input space to form a reconstructed output. If training for reconstruction ideally $x \approx \hat{x}$.

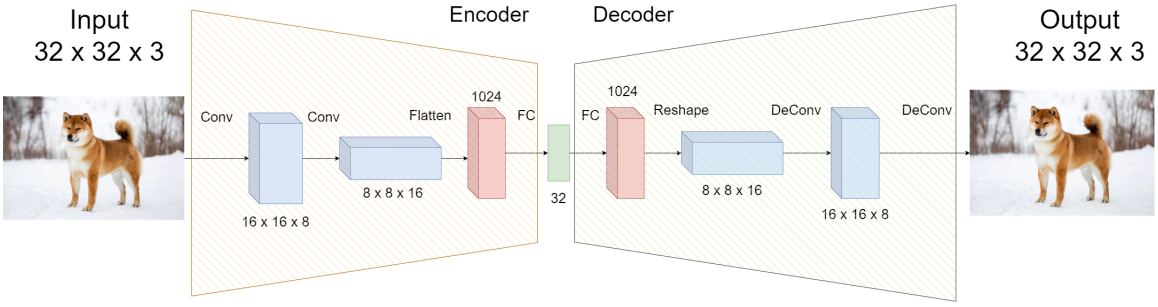


Figure 11: Diagram of a simple Convolutional Autoencoder. The encoder uses convolutions to downsample the image and a final fully connected layer to get the latent vector. The decoder is a mirrored architecture using deconvolutions to upsample the latent representation to the original size.

Generative Adversarial Networks

Generative Adversarial Networks (GAN)[75] are a generative model consisting of two neural networks: a generator and a discriminator. The generator learns to create new data, while the discriminator learns to distinguish the generated data from real data.

The generator network inputs a random noise vector and maps it to a data space, such as an image, audio, or text. The generator's goal is to create data similar to the real data, and it is trained to do so by trying to fool the discriminator network. Conversely, the discriminator is trained to identify whether the data it receives is real or generated. It takes in real and generated data and outputs a probability of the input being real. This process is illustrated in Fig. 12.

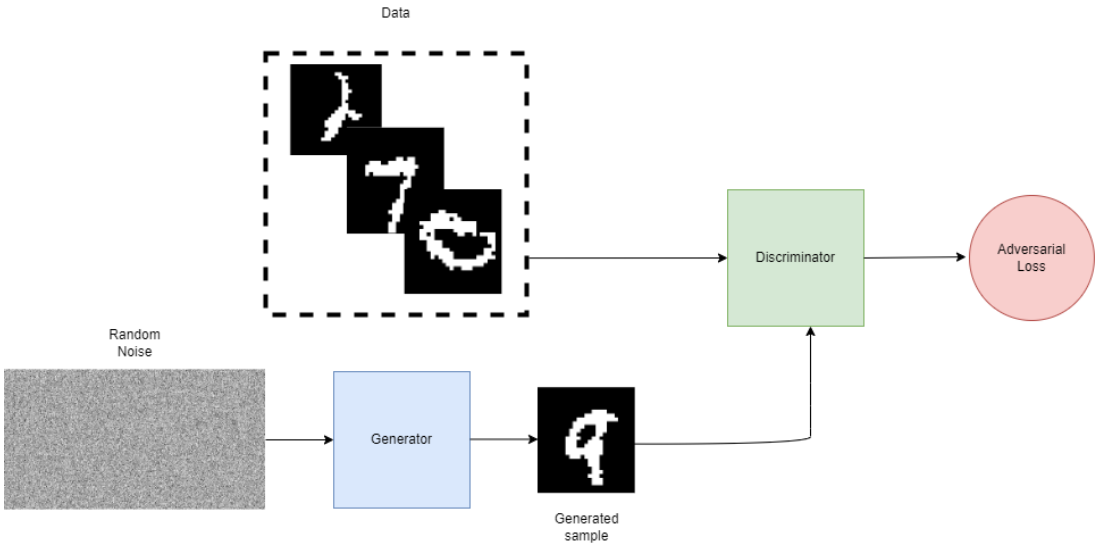


Figure 12: Diagram of a GAN: The generator learns to generate a sample that resembles the training data from random noise. The discriminator is trained to distinguish real from fake samples. The generator is trained to fool the discriminator. This process is called "adversarial" because it is a competition between the two networks.

During training, the generator and the discriminator are trained simultaneously in an adversarial manner. The generator tries to create data that can fool the discriminator, while the discriminator tries to identify the fake data correctly. This process continues until the generator creates data indistinguishable from real data, at which point the GAN is considered to have converged.

The training objective of the GAN is to find the equilibrium between the generator and the discriminator, which is a min-max training process that tries to minimize the times the samples from the generator are rejected and maximize the times the discriminator succeeds in assigning the correct label to real and fake data. Expressed mathematically, this cost function is:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}\,(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \quad (34)$$

In this function:

- $D(x)$ is the discriminator's estimate of the probability that real data instance x is real.
- $\mathbb{E}_{\boldsymbol{x}}$ is the expected value over all real data instances.
- $G(z)$ is the generator's output when given noise z.
- $D(G(z))$ is the discriminator's estimate of the probability that a fake instance is real.
- $\mathbb{E}_{\boldsymbol{z}}$ is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances $G(z)$).

The formula derives from the cross-entropy between the real and generated distributions. The generator can't directly affect the $log(D(x))$ term in the function, so, for the generator, minimizing the loss is equivalent to minimizing $log(1 - D(G(z)))$.

GANs can generate high-quality samples, but one of the main limitations of GANs is their training instability and tendency to overfit. Different techniques exist to mitigate these issues: for example, there exists the possibility to use different loss functions such as with WGAN[3], WGAN-GP[79] and MMD-GAN [134]; also to improve training stability techniques such as the progressive growing of training complexity [103], where the GAN is trained first on lower resolution images then the resolution is increased gradually, or architectural changes like BigGAN [24]. GANs have had extensive applications in computer vision [258], time series [25], and other tasks where it is desirable to learn data samples following a complex distribution with good quality, while there is no need to have a tractable representation of this distribution.

2.7.2 Deep Learning for Sequential Data

Recurrent Neural Networks

Recurrent Neural Networks (RNN) are neural networks designed for processing sequential data, i.e., a sequence of values $x_{(1)}, x_{(2)}, ..., x_{(n)}$. The main difference between RNNs and multi-layer neural networks is that the parameters are shared for multiple time steps. Let $f : \mathcal{X} \times \theta \to \mathcal{H}$ be a function representing an RNN cell (neuron) that takes the inputs $x_{(t)}$ and calculates a hidden state

$h_{(t)}$ that is passed for the calculation of the next hidden state. Many RNN models follow an equation similar to eq. 35. Fig. 13 gives a graphical representation of a basic RNN unit (compressed and unfolded). In this process, each sequence step is used as input to calculate the hidden state at that step, along with the previous hidden state. The hidden state at each step then generates an output, which can be further utilized as a variable to compute the desired output. The general form of the output (hidden state) of an RNN cell is given in eq. 35.

$$h_{(t)} = f\left(h_{(t-1)}, x_{(t)}; \theta\right) \tag{35}$$

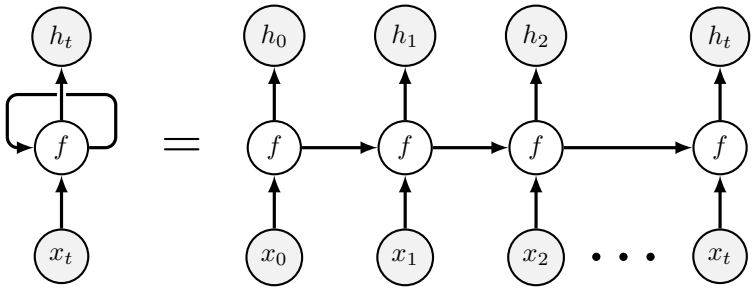


Figure 13: Representation of a RNN model

One of the problems of RNNs is that they have difficulties learning long-term dependencies. Suppose the current output depends on an input located many steps back, for example, when predicting the next word in a long sentence. In that case, the magnitude of the contextual information may exceed the Recurrent Neural Network (RNN) capacity due to long-range correlations, rendering it challenging to process and make accurate predictions effectively. The basic problem with these dependencies is that gradients propagated over many stages tend to either vanish or explode. Even if the recurrent network is stable during training, the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions compared to short-term ones [74].

The first successful solution to this problem has been given with the Long Short Term Memory Networks (LSTM) [91]. LSTMs are composed of a complex cell unit that embeds gates (input, output, and forget gates) to regulate the flow of information into and out of the cell (see fig. 14). This mechanism allows the cell to remember values over arbitrary time intervals. Gates control the amount of information that is forgotten in each state. An LSTM cell can be defined with the following equations:

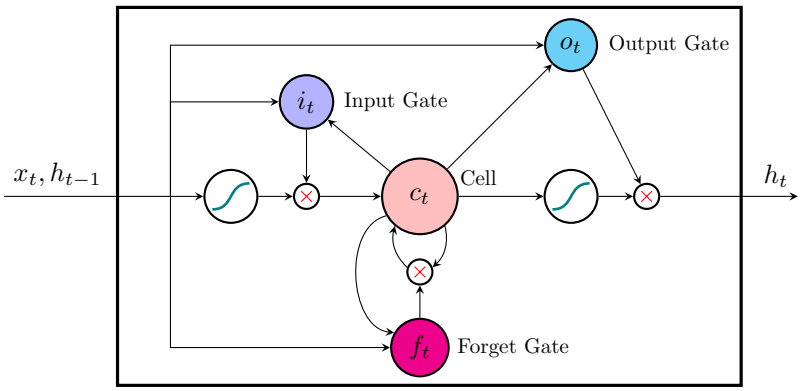


Figure 14: LSTM cell diagram. The input and previous hidden state pass through each of the gates in order to calculate the new cell state, which is then used to produce the next hidden state.

$$
\begin{aligned}
i_t &= \sigma\left(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}\right) \\
f_t &= \sigma\left(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}\right) \\
g_t &= \tanh\left(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}\right) \\
o_t &= \sigma\left(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}\right) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh\left(c_t\right)
\end{aligned}
\tag{36}
$$

Where h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, h_{t-1} is the hidden state of the unit at the step $t-1$ or the initial hidden state at the step 0, and i_t, f_t, g_t, O_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

Recurrent Neural Networks and their variants have played a role in developing technologies such as speech recognition, sentiment analysis, machine translation, and weather forecasting, to name a few. Nowadays, they have been substituted with larger models called transformers (which we present in the next subsection) based on an attention mechanism, which has shown a much higher potential for the semantic representation of sequences, especially for texts. Nevertheless, RNNs remain a good architectural choice when a lightweight model is needed to learn sequential data.

Transformers

The transformer architecture [246] is a more recent deep learning model that has produced revolutionary results mainly in NLP but has recently been used in other modalities such as computer vision and speech processing. These models are a type of neural network that relies on the attention mechanism

[6][188]. Before explaining the transformer architecture's construction, we will briefly explain what attention is in ML. Taking a NLP task as an example, the attention mechanism relates a word in a sentence to the other words in the input. Transformers commonly use self-attention, which means that they use the inputs to define the queries, keys, and values. This attention model is called the query-key-value attention model(QKV). Given the matrix representation of queries $\mathbf{Q} \in \mathbb{R}^{N \times D_k}$, keys $\mathbf{K} \in \mathbb{R}^{M \times D_k}$ and values $\mathbf{V} \in \mathbb{R}^{M \times D_v}$, the scaled-dot product attention used by transformers is:

$$\text{Attention}(\mathrm{Q}, \mathrm{K}, \mathrm{V}) = \text{softmax}\left(\frac{\mathrm{Q}\mathrm{K}^\top}{\sqrt{D_k}}\right)\mathrm{V} = \mathrm{AV} \tag{37}$$

Where D_k, D_v denotes the dimension of the keys and values matrix respectively, A is the attention matrix, and softmax is applied row-wise. The dot products of queries and keys are scaled by $\sqrt{D_k}$ to alleviate the softmax function's vanishing gradients. Transformers use multi-head attention to project Q, V, and K into H sets of learned projections. For each "head" H, the attention is calculated using equation 37 and concatenates each of the outputs:

$$\begin{aligned} &\text{MultiHeadAttn}\ (\mathrm{Q}, \mathrm{K}, \mathrm{V}) = \text{Concat}\left(\text{head}_1, \cdots, \text{head}_H\right)\mathbf{W}^O \\ &\text{where head}_i = \text{Attention}\left(\mathrm{Q}\mathrm{W}_i^Q, \mathrm{K}\mathrm{W}_i^K, \mathrm{V}\mathrm{W}_i^V\right) \end{aligned} \tag{38}$$

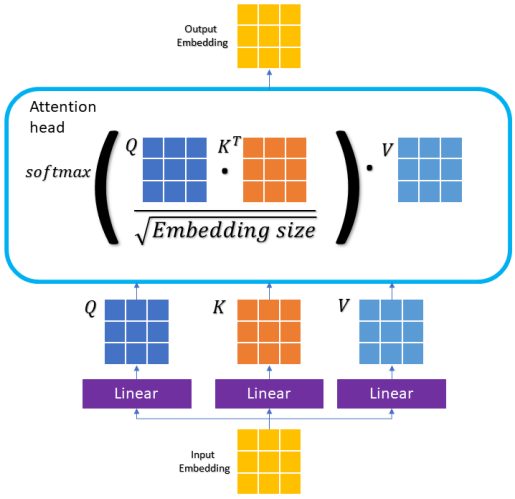


Figure 15: Diagram of Self-Attention head.

Being $\mathbf{W}$ a learnable parameter matrix. In Fig. 15, a graphical representation of that attention head can be seen. The original transformer architecture [246] is an Encoder-Decoder model in which each is a stack of L identical blocks.

Each encoder block comprises a multi-head attention module and a position-wise FFN. For adding depth to the models, a residual connection is used around each module, followed by layer normalization[4]. In contrast, decoder blocks insert cross-attention modules between the multi-head self-attention modules and the position-wise FFNs. The self-attention modules in the decoder are adapted to prevent each position from attending to subsequent positions [146]. The transformed architecture can be seen in Fig. 16.

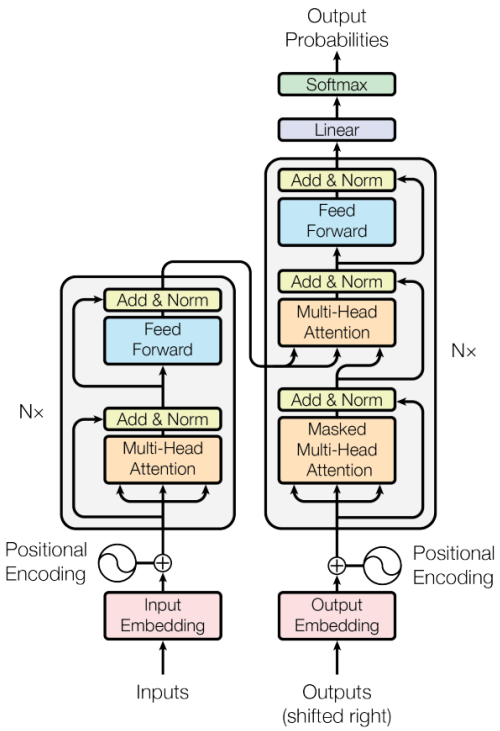


Figure 16: Vanilla Transformer architecture from [246].

There are three types of attention depending on the source of queries and key-value pairs:

- Self-attention. In the transformer encoder, $Q = K = V = X$, X is the previous layer's output.
- Masked Self-attention. In the decoder of the transformer, the self-attention is restricted such that queries at each position can only attend to all key-value pairs up to and including that position. In other words, that means that values beyond that position are hidden or "masked." This is done by applying a mask function to the unnormalized attention matrix $\hat{\mathbf{A}} = \exp\left(\frac{Q^\top}{\sqrt{D_k}}\right)$, where the illegal positions are masked out by setting

$\hat{A}_{ij} = -\infty$ if $i < j$. This attention type is often called autoregressive or causal attention.

- Cross-attention. The queries are projected from the outputs of the previous decoder layer, where the keys and values are projected using the outputs of the encoder.

Transformers use positional embedding because, unlike RNNs or CNNs, they do not include a notion of locality, so positional embedding is used to model position representation. Transformers are more general-purpose architecture, and even though they were initially designed for NLP tasks, they do not specifically address certain inductive biases. It can be counter-intuitive to think that a model with no inductive biases has replaced other methods that were previously state-of-the-art tasks, for they were specifically designed. The advantage of transformers in this sense is that by not having strong inductive biases like RNNs and CNNs, they can learn long-range dependencies more effectively from given enough data. Another property of transformers is that they get better the bigger they get, meaning that their performance scales up depending on their number of parameters, at the expense of being more computationally expensive.

There are many types of transformers, with architectural changes or more efficient computations. Two important variations are the encoder-only transformers with models such as BERT [50], which is used to learn text representations that then can be used for a variety of downstream tasks and the decoder-only transformers like GPT [26] that are trained for text generation, given an input sequence, they try to predict the next ones. Beyond language, transformers have been used for different tasks such as computer vision [51], high-resolution image generation [280], learning on graphs [110], multi-modal learning [95], for predicting chemical reactions [222], predicting the structure of proteins [204], climate modeling [184][176] and many more.

The main limitation of using transformers is the amount of data and computation they require. This is especially challenging for science since data is decentralized, and because of the computational expense, training these models is prohibitive to labs with modest resources. With computational hardware and algorithms getting more powerful and efficient, and with the increase of collaborative efforts, the adoption of these models for scientific problems will grow and bring great capabilities in terms of generalization to complex problems.

2.7.3 Deep Learning And Differential Equations

Differential equations constitute the mathematical modeling tool used in the natural sciences. Through them, we can encode physical processes, chemical reactions, stochastic processes, etc. Differential equations serve as an explainable modeling tool to understand what is happening behind certain phenomena

and predictive models through their solution with numerical methods. The solution of differential equations gets difficult when we encounter problems with multi-physics/ multi-scale behavior because of the cost of computation, because of ill-posed problems where some conditions are missing, or when the model is not accurate enough. In the case of the use of numerical methods for inverse problems, where we want to infer properties of the system via simulations, or when want to use them for optimization and control, requires reduced-order or surrogate models that mimic the system response and are fast to compute because the direct use of the full-order model is computationally prohibitive.

To improve current modeling approaches in science, integrating data in models is of particular interest since the availability of data from multiple experiments and simulations is growing every year. Machine learning, particularly DL, is a promising methodology because of its capability of universal approximation on the big data regime and automatic feature detection. However, unlike computer vision and NLP, where data is massively available, scientific data presents other challenges as it is more expensive, presents different types of modalities, is harder to reproduce, and is dispersed among different research centers, in many cases restricted to internal use due to confidentiality issues. In addition to that, the requirements of data-driven methods in science are more constrained because several properties inherent to the system must be learned as well, making complex optimization objectives.

For the reasons mentioned above, new machine learning methods have been developed that exploit the structure of physical systems to make models more appropriate for scientific applications. These methods are based on the numerical methods used to obtain the solution of partial differential equations (PDE) and ordinary differential equations (ODE). The first method introduced is the physics-informed neural network (PINN) [198] that uses the PDE residual as optimization objective/constraint, Neural Operators [117] that attempts to learn operators on functional spaces and neural differential equations [106] that build neural networks as infinite depth dynamical systems that are solved with an ODE integrator.

Physics-Informed Neural Networks

Physics-Informed Neural Networks (PINNs) exploit neural networks and automatic differentiation (AD) to calculate the derivatives relevant to the system modeled by a PDE and use this PDE as an optimization objective. On the ML side, PINNs can be seen as a special form of Neural Implicit Representation [161], which uses coordinate-based neural networks (commonly MLPs) to represent a signal. In the case of PINNs, the signal the NN represents a function $u(\mathbf{x}, t)$ that satisfies a PDE $\frac{du}{dt} = \mathcal{N}(x, t, u)$ for given initial and boundary conditions.

To illustrate better how PINNs work, let's take, for example, the viscous

burgers' equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = v\frac{\partial^2 u}{\partial x^2} \tag{39}$$

We want to solve this equation for a specific initial condition $u(x,0) = u_0$ and Dirichlet boundary conditions $u(x,t) = f(x), \quad \forall x \in \partial\Omega$, where Ω is the physical domain of interest, and $\partial\omega$ represent its boundaries. The general approach to solving the PDE is to discretize it to form a system of differential methods and then solve it with an appropriate numerical method. With the PINN, the neural network is a parametric function whose weights are determined by optimization. Assuming we have data of the solution of this equation $u(x,t)$ for a given time interval $t \in [0,T]$ and $x \in \Omega \subset \mathrm{R}$ we get the approximated the solution $u_\theta(x,t)$ of eq.39 by minimizing:

$$L = \lambda_1 L_{data} + \lambda_2 L_{pde} \tag{40}$$

$$L_{data} = \frac{1}{N}\sum_{i=0}^{N} ||u^{(i)}(x,t) - u_\theta(x,t)||^2 \tag{41}$$

$$L_{pde} = \frac{1}{N_{pde}}\sum_{i=0}^{N_{pde}} \left\| \frac{\partial u_\theta}{\partial t} + u_\theta\frac{\partial u_\theta}{\partial x} - v\frac{\partial^2 u_\theta}{\partial x^2} \right\|^2_{(x_i,t_i)} \tag{42}$$

Where λ_1 and λ_2 are multipliers that regulate the importance of the terms of the loss function. In PINNs, these hyperparameters are important as they can improve the accuracy and convergence of the learning process, which determines if the PINN will fail or not [253][256]. A diagram of this kind of PINN is shown in fig.17.

The advantage of PINNs over numerical methods is that they do not rely on a mesh-based discretization since we use automatic differentiation to compute partial derivatives. With the possibility of using data to optimize the model, inverse problems, or work on cases with boundary/initial conditions extracted from observational data. PINNs can work on different types of PDEs like Navier-Stokes equations [199] and RANS equations [56] used in CFD, stochastic differential equations [272][43], Poisson equation [156], among several others.

Nevertheless, one of the main limitations of PINNs is their inability to tackle multi-scale problems like turbulent flows. This is mainly due to a problem known as spectral bias [196]. This error means neural networks better fit lower-frequency functions, and the higher frequencies can be neglected or difficult to learn. This can be mitigated by designing special architectures or methods like Fourier feature embeddings [239]. This consists of adding an input embedding based on a sinusoidal representation of the coordinates. Fourier Features help to learn higher-frequency details, as seen in Fig. 18. Fourier Features are calculated as follows:

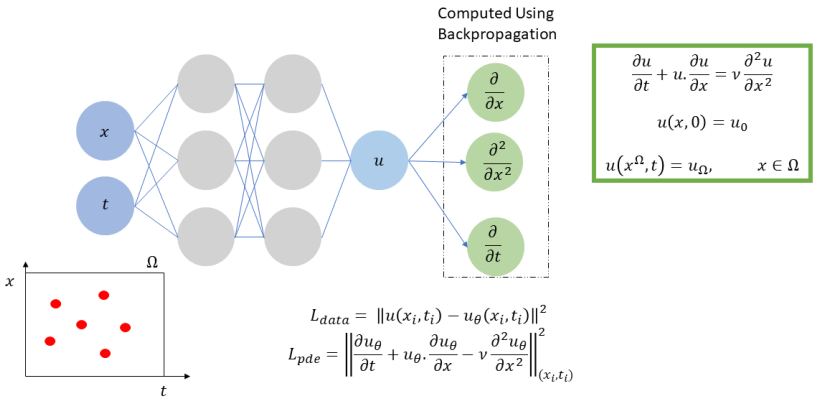


Figure 17: Diagram of physics-informed neural network for approximating the solution to Burgers' equation. A coordinate-based NN represents the solution function that satisfies certain IC and BC. The derivatives are calculated through automatic differentiation, so a computational mesh is not required.

$$\gamma(\mathbf{v}) = [\cos(2\pi \mathbf{B}\mathbf{v}), \sin(2\pi \mathbf{B}\mathbf{v})]^{\mathrm{T}} \tag{43}$$

Where $\mathbf{B}$ is a Gaussian random matrix and $\mathbf{v}$ is the vector containing the input coordinates (x, y, z) for 3D or (x, y) for 2D data.

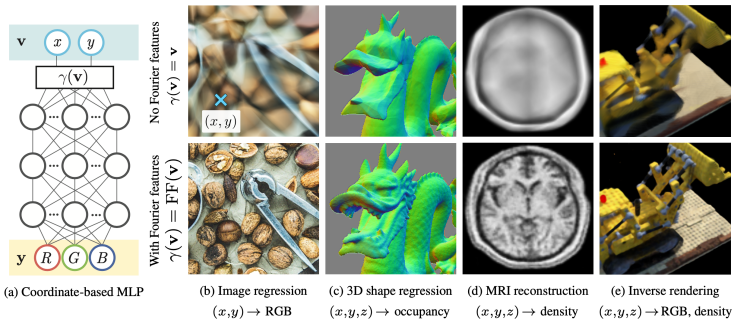


Figure 18: Passing input points through a Fourier Feature mapping enables an FFN a) to learn high-frequency functions for different tasks. The network takes as input the coordinates and outputs the RGB colors or other properties [239].

However, still, for PINNs, this issue is not fully resolved for multi-scale problems [254], as PINNs require careful selection of Fourier feature embeddings; this

is dependent on the nature of the problem and requires domain knowledge. This is prohibitive to do in cases where some parts of the physics are unknown, and it hinders generalization for more complex dynamics like turbulence, where these embeddings have to be re-adjusted when the parameters of the model change.

Another problem of PINNs is their trainability. The loss term comprises data and PDE residuals, which is a non-convex optimization problem. Sometimes, the data can be noisy or have some errors if it comes from under-resolved numerical simulations; this may cause competing behavior between loss terms [253]. Techniques to remedy these issues include special neural network architectures with adaptive activation [96][97], learning rate annealing methods [253], and adaptive sampling of the data points to ameliorate the convergence [86]. This chooses PINN very hard, depending on the system to be learned. On the other hand, the computation of derivatives can be a drawback, especially when high-order derivatives appear, making the optimization process very slow.

Although limitations exist, PINNs continue to develop in theory and applications. For example, PINNs are being used to develop simulation software that uses ML to integrate data that can be used for digital twins and inverse problems, as is the case of NVIDIA modulus [86]. PINNs have been used for applications such as finding blow-up of solutions to the Euler equation [257] and for optimal control of PDEs [169]. Another outlook is the combination of PINNs with other ML methods to make more general models with convergence guarantees [286][252].

Neural Operators

Neural networks are mappings between finite-dimensional spaces, for example, a finite set of images to a finite set of classes. In the real world, many problems can be modeled as infinite dimensional spaces of functions, for example, the prediction of climate events as a function of variables like Temperature or wind velocity, which are themselves functions that depend on location and time. Learning the mapping between function spaces has led to the design of novel deep learning architectures that can be classified as neural operators [117], which are independent of the discretization of the function space. One example of the application of a learned operator can be the learning of the solution operator of a PDE, where the input is a function of the initial condition, boundary conditions, or coefficients, and the output is the solution function of the PDE. What is advantageous about this framework is that, in theory, subsampled sets of these function spaces could be learned and generalized to points not seen during training. Another advantage is that the operator could be learned without knowing the underlying model or parts of it, which allows one to apply them to data where some parts of the model are unknown or missing.

The goal of neural operators is to learn the mapping between two infinite-dimensional spaces using a finite set of observations of input-output pairs. Let's

take $\mathcal{A}$ and $\mathcal{U}$, Banach spaces of functions defined on bounded domains $D \subset \mathbb{R}^d$, $D' \subset \mathbb{R}^{d'}$ respectively and $\mathcal{G}^\dagger : \mathcal{A} \to \mathcal{U}$ be a non-linear mapping. Supposing we have observations $\{a_j, u_j\}_{j=1}^N$ where a_j μ are independent and identically distributed samples drawn from some probability measure μ supported on $\mathcal{A}$ and $u_j = \mathcal{G}^\dagger(a_j)$ is possibly corrupted with noise. The goal is to build an approximation of $\mathcal{G}^\dagger$ by constructing a parametric mapping:

$$\mathcal{G}_\theta : \mathcal{A} \to \mathcal{U}, \theta \in \mathbb{R}^p \tag{44}$$

With parameters from the infinite dimensional space $\mathbb{R}^p$ and then choosing $\theta^\dagger \in \mathbb{R}^p$ so that $\mathcal{G}_{\theta^\dagger} \approx \mathcal{G}^\dagger$. We find the parameters by solving the empirical-risk minimization problem [117]:

$$\min_{\theta \in \mathbb{R}^p} \mathbb{E}_{a \sim \mu} \left\| \mathcal{G}^\dagger(a) - \mathcal{G}_\theta(a) \right\|_{\mathcal{U}}^2 \approx \min_{\theta \in \mathbb{R}^p} \frac{1}{N} \sum_{j=1}^{N} \left\| u_j - \mathcal{G}_\theta\left(a_j\right) \right\|_{\mathcal{U}}^2 \tag{45}$$

The general framework of a Neural Operator follows these principles: following the definition by [117] let us assume that the input functions $a \in \mathcal{A}$ are $\mathbb{R}^{d_a}$ valued and defined in the bounded domain $D \subset \mathbb{R}^d$. In contrast, the output functions $u \in \mathcal{U}$ are $\mathbb{R}^{d_u}$ valued and defined on the bounded domain $D' \subset \mathcal{R}^{d'}$. The proposed architecture $\mathcal{G}_\theta : \mathcal{A} \to \mathcal{U}$ has the following overall structure:

1. **Lifting:** the input is mapped to the first hidden representation in higher dimensional feature space: $\{a : D \to \mathbb{R}^{d_a}\} \to \{v_0 : D \to \mathbb{R}^{d_{v_0}}\}$, where $d_{v_0} > d_a$.
2. **Iterative Kernel Integration:** for $t = 0, ..., T-1$ map each hidden representation to the next $\{v_t : D_t \to \mathbb{R}^{d_{v_t}}\} \to \{v_{t+1} : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}\}$ using the sum of a local linear operator, the integral kernel operator and a bias function composing each layer with non-linearity.
3. **Projection:** Using a pointwise function, as the lifting step, we project the hidden representation to the function space of the solution function: $\{v_T : D_T \to \mathbb{R}^{d_{v_T}}\} \to \{u : D' \to \mathbb{R}^{d_u}\}$.

This structure can be represented mathematically as shown in the equation below:

$$\mathcal{G}_\theta := \mathcal{Q} \circ \sigma_T \left(W_{T-1} + \mathcal{K}_{T-1} + b_{T-1}\right) \circ \cdots \circ \sigma_1 \left(W_0 + \mathcal{K}_0 + b_0\right) \circ \mathcal{P} \tag{46}$$

Where $\mathcal{P} : \mathbb{R}^{d_a} \to \mathbb{R}^{d_{v_0}}, \mathcal{Q} : \mathbb{R}^{d_{v_T}} \to \mathbb{R}^{d_u}$ are the local lifting and projection maps respectively, $W_t \in \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}}$ are local linear operators, $\mathcal{K}_t : \left\{v_t : D_t \to \mathbb{R}^{d_{v_t}}\right\} \to \left\{v_{t+1} : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}\right\}$ are integral kernel operators, $b_t : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}$ are bias functions, and σ_t are fixed activation functions acting locally as maps $\mathbb{R}^{v_{t+1}} \to \mathbb{R}^{v_{t+1}}$ in each layer. In this framework, $\mathcal{Q}, \mathcal{P}$ are pointwise local operations that do not depend on the discretization of the data. This

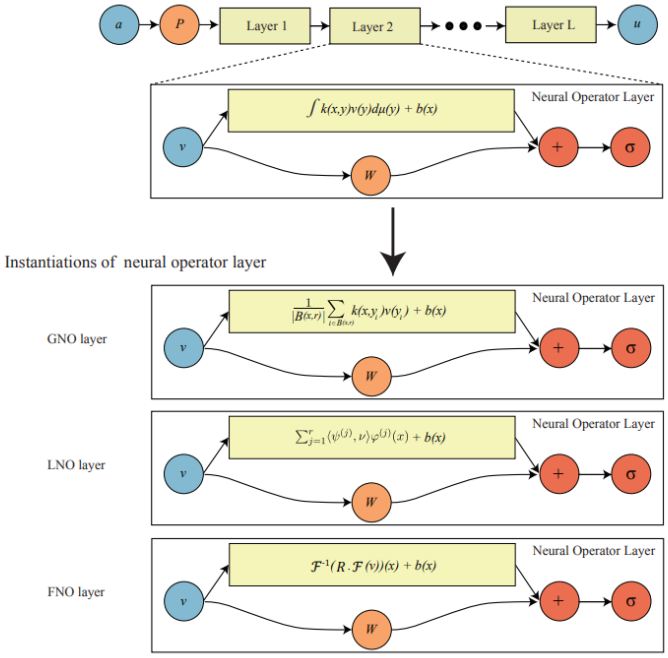


Figure 19: Diagram illustrating the framework of Neural Operators. A NO layer is defined as a way of approximating the kernel integral. In the picture, we can observe the approximation of the kernel integral transform with graph kernel approximation, low-rank approximation, and the Fourier transform [117].

is what it is meant by local. The idea behind the lifting and projection is that we can project to higher dimensions where non-local parts can be learned better [117]. Afterward, the kernel integral layers define the neural operator type to be implemented. The basic form of the integral kernel is:

$$\left(\mathcal{K}_t\left(v_t\right)\right)(x)=\int_{D_t} \kappa^{(t)}(x, y) v_t(y) \mathrm{d} y \quad \forall x \in D_{t+1} \tag{47}$$

What happens afterward is that we need a numerical approximation of this integral kernel to work with, and how we make this approximation defines the type of architecture to be used (see fig. 19). The kernel can be approximated via Graph Neural Networks [139][140], Fourier transform [141], Linear approximation [117], attention mechanism [33][137][114], Chebyshev/Fourier series [59], wavelets [80], among others.

Neural operators are inspired by the theorem of universal approximation of continuous functionals [41] and nonlinear operators [42] with neural networks. The DeepONet (Deep Operator Network) [154] is a special type of neural op-

erator directly inspired by these theorems. According to [154], The theorem of universal approximation for operators reads as follows:

(Universal Approximation Theorem for Operators). Suppose that σ is a continuous non-polynomial function, X is a Banach Space, $K_1 \subset X, K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$, respectively, V is a compact set in $C(K_1), G$ is a nonlinear continuous operator, which maps V into $C(K_2)$. Then for any $\epsilon > 0$, there are positive integers $n,\ p, m,$ constants $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}, w_k \in \mathbb{R}^d,\ x_j \in K_1,\ i = 1, \ldots, n, k = 1, \ldots, p, j = 1, \ldots, m,$ such that:

$$|G(u)(y) - \underbrace{\sum_{k=1}^{p} \sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{\text{branch}} \underbrace{\sigma(w_k \cdot y + \zeta_k)}_{\text{trunk}}| < \epsilon \tag{48}$$

holds for all $u \in V$ and $y \in K_2$.

The DeepONet is derived from the theorem in eq.48 by replacing the two terms in the equation as neural networks. We will have a branch net that evaluates the discretized input function at specific sensor locations and a trunk net that processes the new location where the operator will be evaluated. We can interpret the output of the DeepONet sub-networks as the trunk net being a basis function evaluated at specific locations and the branch net being a coefficient function dependent on the input function's measurements. In the standard DeepONet formulation, both the branch net and trunk net are fully connected MLPs, but they can be other models as CNNs or RNNs for branch nets or POD basis for the trunk net. Unlike the previous general neural operator formulation, the DeepONet is limited to discretizing the input function and the number of sensors since this variable is fixed at the beginning as a hyperparameter. For different types of operators and problems, different amounts of sensors and measured features may be needed (see fig.20).

Neural Operators have been used in a wide variety of applications. They have been used for weather forecasting [184], for simulations of the behavior of viruses [243], for faster sampling of diffusion models [282], bayesian inverse problems [101][138], prediction of chaotic dynamics [142][252], solving PDEs [255][143], solving stochastic PDEs [215], inference of bubble dynamics [145], and an increasing list of other applications. Neural Operators can be used as a general framework to infer operators, even when a theoretical formulation of them doesn't exist like token mixing for computer vision applications [78].

Neural operators still suffer from limitations in terms of scalability and generalization. As with PINNs, there are many variations, and they struggle to achieve small errors on multi-scale problems. Integrating techniques inspired by large models and incorporating more physical biases on the models can help

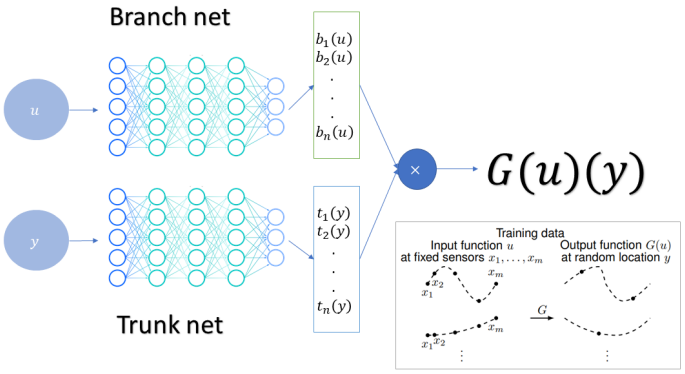


Figure 20: Diagram of DeepONet. The branch net takes as input the input function evaluated at fixed sensor locations, and the trunk net evaluates the operator basis at query locations, which can be arbitrary and discretization-free.

them tackle the goal of making more accurate, scalable, and general architectures for scientific ML.

Neural Differential Equations

Neural Differential Equations are a family of Deep Learning methods inspired by interpreting neural networks as a dynamical system where a hidden state changes with the time/depth of the network. The most simple form is the neural ordinary differential equation (NODE) [40]:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta) \tag{49}$$

The forward pass of the NN becomes the solution to the initial value problem of finding $h(T)$ at a certain time T starting from an initial hidden state $h(0)$. The dynamics of the hidden state are solved through numerical integration methods for solving ODEs. For example, we can interpret residual neural networks as solving eq.49 with the Euler method [213]:

$$h_{t+1} = h_t + f(h(t), t, \theta) \tag{50}$$

To train NODEs, we must backpropagate through the numerical ode solver to calculate its derivative. To do this, we can follow one of these approaches: discretize-then-optimize, optimize-then-discretize, and reversible ODE solvers [106].

Discretize-then-optimize. This method consists of backpropagating through the internal operations of the differential equation solver, given it was written on a framework supporting automatic differentiation. This method is accurate and fast with the downside of being memory hungry as this scales linearly with the number of time-steps used in the discretization of the ODE.

Optimize-then-discretise. This method differentiates the continuous model, thus obtaining the gradients by solving a backward-in-time ODE. This technique is known as the continuous adjoint method[34][15]. Let $h_0 \in \mathbb{R}^d$ and $\theta \in \mathbb{R}^m$. Let $f_\theta : [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ be continuous in t, uniformly Lipschitz in h, and continuously differentiable in h. Let $h : [0, T] \to \mathbb{R}^d$ be the unique solution to

$$h(0) = h_0, \quad \frac{\mathrm{d}h}{\mathrm{d}t}(t) = f_\theta(t, h(t)).$$

Let $L = L(h(T)\ )$ be some scalar function of the terminal value $h(T)$. Then $\frac{\mathrm{d}L}{\mathrm{d}h(t)} = a_h(t)$ and $\frac{\mathrm{d}L}{\mathrm{d}\theta} = a_\theta(0)$, where $a_h : [0, T] \to \mathbb{R}^d$ and $a_\theta : [0, T] \to \mathbb{R}^m$ solve the system of differential equations:

$$a_h(T) = \frac{\mathrm{d}L}{\mathrm{d}h(T)}, \qquad \frac{\mathrm{d}a_h}{\mathrm{d}t}(t) = -a_h(t)^\top \frac{\partial f_\theta}{\partial h}(t, h(t)) \tag{51}$$

$$a_\theta(T) = 0, \qquad \frac{\mathrm{d}a_\theta}{\mathrm{d}t}(t) = -a_h(t)^\top \frac{\partial f_\theta}{\partial \theta}(t, h(t)) \tag{52}$$

The advantage of the adjoint method is that the memory cost of the gradient remains constant independently of the discretization of the differential equation. The downside of the adjoint method is that it involves solving additional differential equations, which adds a computational cost that makes the optimization slower. In addition, the gradient is an approximation itself and depends heavily on the value of the hidden state at time T. Making convergence and training stability difficult when using this approach.

Reversible ODE solvers. The forward and backward passes have the same computational cost with reversible ODE solvers. Examples of reversible ODE solvers are the reversible Heun's method [108] and the asynchronous leapfrog method [171]. These methods are still in their infancy. They provide computational efficiency and a low memory footprint but have limited stability.

Neural differential equations have been used in many applications such as image classification [40], continuous normalizing flows [76], irregularly sampled time-series [210][107][168], generative modeling of continuous time-series [109], system identification [193], modeling physical systems and reduced order modeling [130][125][164][160].

One interesting application of NDEs is the combination of neural networks and PDEs to add inductive biases to the machine learning method. This method

is called universal differential equation (UDE) [194]. They are called universal because they can be used with ODEs, PDEs, and SDEs. This approach consists of substituting parts of the differential equation with an ML approximator, a neural network. The UDE's forward pass consists of solving the differential equation with a numerical method, and in fitting the values with data, the derivatives are calculated with one of the previously mentioned approaches. The general form of the UDE is forced stochastic delay partial differential equation:

$$\mathcal{N}\left[u(t), u(\alpha(t)), W(t), U_\theta(u, \beta(t))\right] = 0 \tag{53}$$

Where $\alpha(t)$ is a delay function and $W(t)$ is a Wiener process. UDEs can be used for symbolic regression (identifying terms of a differential equation), nonlinear regression, reduced order modeling, solving high dimensional equations, model closure discovery, and other uses since these models are very flexible.

It is likely that the intersection of differential equations and neural networks will continue to grow in the future, proportional to the growth of ML applications to scientific problems. Differential equations provide a framework to transfer domain knowledge to ML models to make them respect physical laws, which are computationally efficient and more interpretable.

3 Machine Learning for Computational Fluid Dynamics

3.1 Introduction

After providing an overview of the applications of Deep Learning for Computer Vision, Sequence Modeling, and Differential Equations, this next chapter is devoted to the literature review of Deep Learning applied to Computational Fluid Dynamics. The objectives of this literature review are as follows:

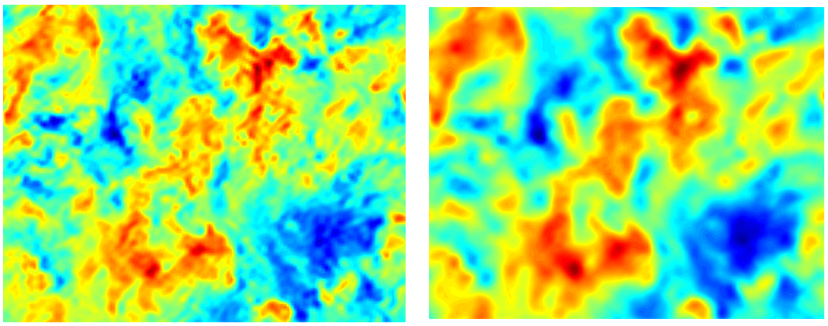
1. Identify the main areas of applications of Machine Learning in CFD.
2. Identify the gaps and main challenges ML applied to CFD faces and concentrate on one of these problems for the rest of the Ph.D. project.

Several reviews address the applications of ML and CFD to fluid mechanics [28][248][11][54][53]. The present review does not attempt to make something different but rather to update and enrich what currently has been done with recent information. The literature review is divided into three major applications: Turbulence modeling, Reduced Order Modeling, and Acceleration of Direct Numerical Simulations (DNS). The following sections discuss datasets and Uncertainty Quantification. Lastly, the closing section highlights future directions and opportunities for the application of ML to CFD.

3.2 Machine Learning for turbulence modeling

The representation of turbulence is a complicated task because of the wide range of active spatial-temporal scales and the chaotic nature of the flow. Due to the growth of computational power, Direct Numerical Simulations can now simulate flows that involve the physics of turbulence to a very precise degree of accuracy [131]. However, techniques that approximate the turbulent flow behavior are preferred in most industrial applications because of their reduced computational cost. The most common of these approaches are RANS (Reynolds-Averaged Navier Stokes) and LES (Large Eddy Simulation) methods. RANS rely on modeling approaches to represent turbulence, so, in essence, they cost considerably less than DNS. RANS models are constructed by applying averaging techniques to the governing equations and require closures to represent the turbulent stresses and scalar fluxes emerging from the averaging process.

The intuition behind RANS is to decompose the quantities of the fluid flow in a sum of the temporally averaged part and fluctuating part. This is called Reynolds' decomposition since it was first proposed by Osborne Reynolds in [203]. The velocity is decomposed in a mean and fluctuating part: $u = \bar{u} + u'$ where:



(a) (b)

Figure 21: a) DNS of homogenous isotropic turbulence b) Filtered DNS with a box filter of size $\Delta = L/32$ applied [153].

$$\begin{aligned} \bar{u} &= \lim_{T\to\infty} \frac{1}{T} \int_0^T u \, dt \\ u' &= u - \bar{u} \\ \bar{u'} &= \lim_{T\to\infty} \frac{1}{T} \int_0^T u' \, dt = 0 \end{aligned} \tag{54}$$

By substituting eq. 54 into the momentum equation for the case of incompressible flow, the resulting equation for the fluid motion is:

$$\rho \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = \rho \bar{f}_i + \frac{\partial}{\partial x_j} \left[-\bar{p}\delta_{ij} + \mu \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} \right) - \rho \overline{u_i' u_j'} \right] \tag{55}$$

In eq. 55 the term $\rho \overline{u_i' u_j'}$ is the Reynolds stress tensor, and this term needs to be closed via modeling.

The modeling of Reynolds' stresses has progressed using a combination of empiricism, intuition, and asymptotic theories and is constrained by requirements of numerical stability and computational costs. These assumptions introduce potential accuracy limitations and reduced credibility on the predictive abilities of the models. Experimental data have been used to calibrate closures and attempt to improve the accuracy of computations. This has been done using statistical inference techniques [191].

In LES, the larger three-dimensional unsteady turbulent motions are directly represented, and the effects of the smaller-scale motions are modeled. The computational cost of LES lies between Reynolds-stress models and DNS. LES is used for flows in which large-scale unsteadiness is significant.

In a more formal sense, we solve the filtered Navier-Stokes equation with LES, and the cutoff frequencies/scales are modeled. In fig. 21, the effect filtering has on the solution of the NS equation can be observed. Filtering smaller scales gives us the fields containing more energy. The velocity vector is decomposed in a filtered and sub-filtered component, $u = \hat{u} + u'$ For incompressible flow, the filtered NS equation is [55]:

$$\frac{\partial \hat{u}}{\partial t} + \hat{u} \cdot \nabla \hat{u} = -\frac{1}{\rho} \nabla \hat{p} + \nu \nabla^2 \hat{u} - \nabla \tau_{ij}^{SGS} \tag{56}$$

Where:

$$\tau_{ij}^{\mathrm{SGS}} = \widehat{u_i u_j} - \hat{u}_i \hat{u}_j \tag{57}$$

The term τ_{ij}^{SGS} is the sub-grid scale tensor. The role of a sub-grid scale stress model is to replace dissipation with the smallest scale eddies. The problem of turbulence modeling of LES and RANS can be considered the same problem from an input-output perspective. However, they vary distinctively. For instance, in RANS, the temporal averaging applies, and the resolution of the equation does not change. In contrast, the LES approach changes depending on the type of filter used, and the corresponding SGS model reacts to it, so the model is sensible to the discretization and filter used.

3.2.1 The problem of data-driven closure modeling

Let's take $\mathcal{N}(\cdot)$ as the system of NS equations. When we apply a filter (temporal or spatial) $\langle \cdot \rangle$, we arrive at an undetermined system $\langle \mathcal{N}(\cdot) \rangle$ result of the uncertainties introduced by the filtering procedure. The model is represented as:

$$\langle \mathcal{N}(\cdot) \rangle = \mathcal{N}(\langle \cdot \rangle) + \mathcal{M}(\cdot) \tag{58}$$

Where $\mathcal{M}$ is the model that depends on a set of independent values $\mathbf{w}$ and has a specific functional form $\mathcal{P}(\mathbf{w})$. The model also considers a set of coefficients $\mathbf{c}$ that adjust the model to the operating conditions [54]. The model now is:

$$\mathcal{M}(\mathbf{w}; \mathcal{P}(\mathbf{w}); \mathbf{c}) \tag{59}$$

In data-driven modeling, the focus is to calibrate the model according to the data θ. Additional parameters are introduced, and the discrepancy δ describes the ability of the model to represent the data. It is a function of the data and a set of features η. The uncertainty on the data is represented by ϵ_θ. The general form of a data-driven model is:

$$\widetilde{\mathcal{M}} = \mathcal{M}\left(\mathbf{w}; \mathcal{P}(\mathbf{w}); \mathbf{c}(\boldsymbol{\theta}); \boldsymbol{\delta}(\boldsymbol{\theta}, \boldsymbol{\eta}); \boldsymbol{\epsilon}_{\boldsymbol{\theta}}\right) \tag{60}$$

The data-driven modeling approach can be done in many ways. We could bypass the explicit functional form of the model and let the model only depend on data, or we could use ML to determine the coefficients of an already established closure model.

3.2.2 Neural Networks for Reynolds Stress Tensor modeling

Neural Networks seem to be a good fit for modeling closures of their expressiveness and universal function approximation capabilities. Ensuring that the learned Reynolds stress models preserve rotational invariance is important. We must ensure that the NN preserves this property by architectural design or optimization. This property is important because the output anisotropy tensor should be rotated by the same angle when the input is rotated. The strategy is to represent the stress tensor by an invariant set of basis [190]. The eddy-viscosity model will be Galilean invariant if it satisfies this equation:

$$\tilde{\boldsymbol{\tau}} = \sum_{n=1}^{10} c^{(n)}(\boldsymbol{\theta}, \boldsymbol{\eta}) \boldsymbol{T}^{(n)} \tag{61}$$

Where $c^{(n)}(\boldsymbol{\theta}, \eta)$ are the coefficients to be determined and $\boldsymbol{T}^{(n)}$ are known functions of the symmetric (R) and anti-symmetric (S) part of the velocity gradient tensor. In [147], an MLP used these features to learn the coefficients that would approximate the anisotropy tensor. In fig. 22, we can observe the neural network designed to approximate the Reynolds stress tensor that depends on the tensor invariants and tensors derived from the velocity gradients. In fig. 23, a result of the TBNN model used on a RANS simulation can be seen. This is an early demonstration that ML approaches for RANS closures can outperform traditional models by producing more accurate mean velocity profiles and capturing physics that previous approaches could not have modeled.

Other authors approach the closure problem differently. Instead of using NNs, they prefer to preserve the algebraic formulation and use symbolic regression techniques like sparse identification [218] and genetic algorithms [259] to fit algebraic models from data. These approaches are attractive because they provide an interpretable algebraic form and are easy to implement to different flow configurations and solvers. However, they have the limitation that they are restricted to the prior knowledge of functional forms and variables. Combining neural networks and symbolic regression is possible, where the NN discovers the hidden/latent variables and models their interaction to produce the results. Then, a symbolic regression algorithm is used to find a symbolic expression that links the latent variables to the output. This has been done by rediscovering simple classical physics laws [46][132] but could be further explored to extract equations from more complex data.

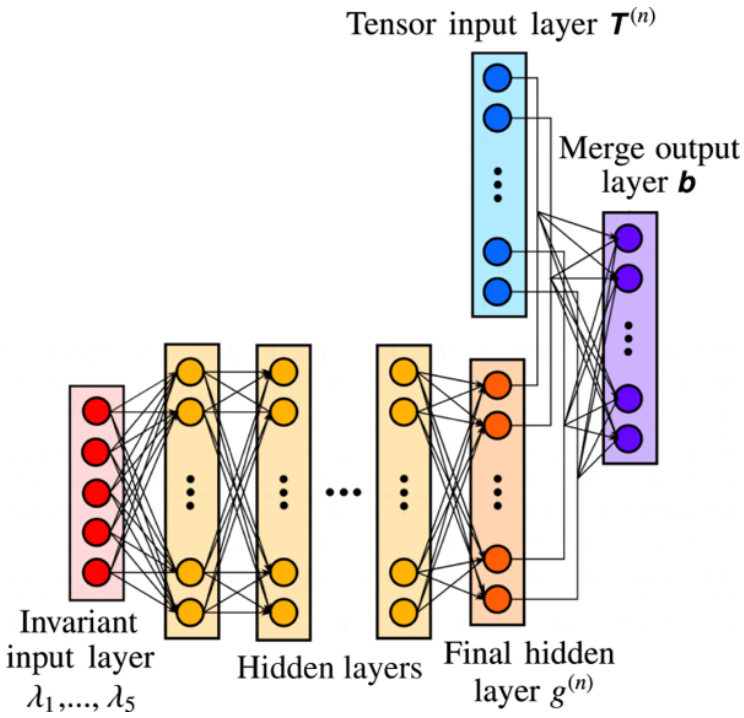


Figure 22: Tensor-Basis Neural Network proposed by [147].

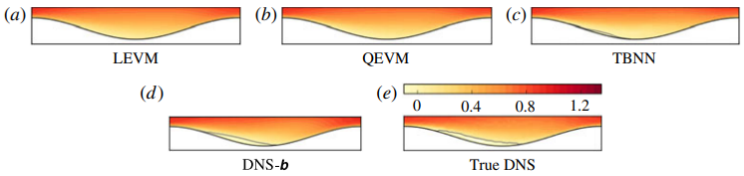


Figure 23: Streamwise velocity in the wavy wall case. Comparison of linear and quadratic eddy viscosity models, TBNN and DNS. TBNN captures better the separation near the wall than other eddy-viscosity models [147].

3.2.3 Machine Learning for Sub-grid scale models

On LES, the closure model must address what happens at the under-resolved scales belonging to the eddies the computational mesh cannot capture. Since LES simulations are commonly performed on grids coarser than DNS, the modeled part accounts for what happens inside the computational cells. For this reason, the closures are called sub-grid scale models (SGS). SGS models aim to model the interaction between large and smaller eddies. Similar to the approach of [147] used for RANS, [158] developed an SGS model for two-dimensional tur-

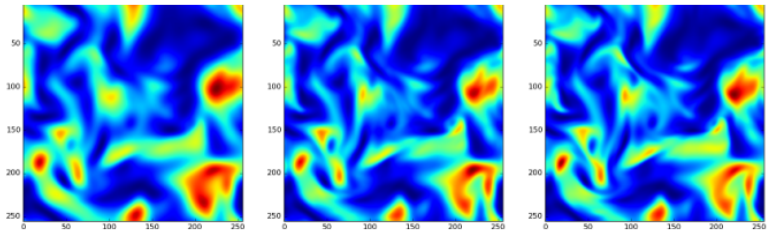


Figure 24: 2D slices of the turbulent kinetic energy of homogeneous isotropic turbulence. On the left is the filtered DNS field, on the center is the field predicted by the ESRGAN [17], and on the left is the true DNS.

bulence that takes as input to an MLP, invariant quantities of the flow.

Computer vision approaches have been applied because of the spatial nature of the SGS modeling problem. Convolutional Neural Network approaches for SGS modeling have been developed by several authors [12][157]. These methods essentially learn to perform super-resolution by approximating the coarser flow solutions to ground truth high-fidelity data. These type of problems has also been tackled by unsupervised methods such as GANs [17] (see fig. 24 for results) and PINNs [99], although not designed directly for SGS modeling, the loss used could be the filtered NS and let the PINN learn the mapping between a first guess of the resolved quantities to the unfiltered terms and corrected values.

The previously seen methods have limitations in data requirements; high fidelity data is needed to train, which may not always be available and limit its generalization capabilities. The second issue is that these models rely on local quantities and may fail on flows with strong long-range correlations across scales. Lastly, CNN-based methods depend on discretization and the use of regular grids. This is limiting for CFD since we need the SGS model to work on different, often unstructured discretizations. One approach that is gaining popularity is adding more inductive biases from the PDE to the models. There are several ways of adding inductive biases to a DL model (see fig. 25). One way to do this is by combining the PDE solver with a residual that is a Machine Learning approximator, like how it is done on Universal Differential Equations [194]. This approach relies on a differentiable physics solver, which means we can backpropagate through the numerical method results and use these gradients to train the closure model end-to-end. By doing this, we need less high-resolution data, and even the neural networks used don't need to be very big as demonstrated it [224](see fig. 26).

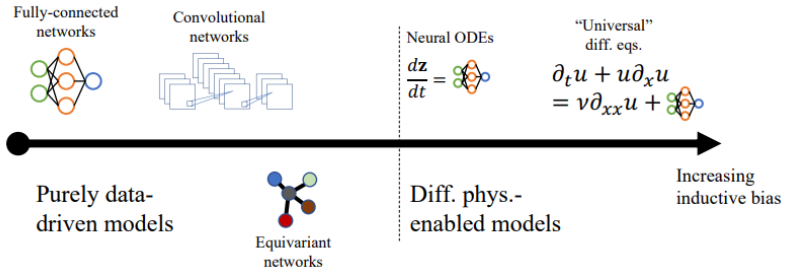


Figure 25: From low to high inductive biases. Purely data-driven models can incorporate inductive biases like locality, shift invariance, or roto-translational equivariance. Differentiable physics models are constrained by known physics, such as dynamical systems or advective and dissipative terms [224].

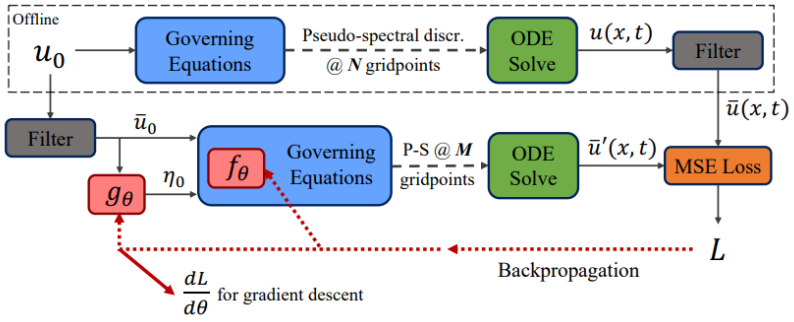


Figure 26: Diagram of the training method in [224]. An initial velocity field is evolved on a DNS-resolved grid and filtered to produce ground truth data. Using the differentiable physics solver, the same initial condition is filtered and evolved for the training loop. The gradients of this process are computed to update the closure term parameters.

3.2.4 Reinforcement Learning for turbulence modeling

SGS models trained with supervised learning have several limitations. To construct the inputs, they are often trained on filtered data from DNS databases. Then, the model is trained to correct the filtered inputs to match the DNS values. The problem with the models trained this way is that they have larger errors when tested on a CFD simulation, even if they were the optimal model for the filtered DNS cases. This happens mainly because the explicit filtering used to build the training data does not correspond to the one used on LES, which can also be implicitly produced by the coarse mesh [12]. These SGS models have not been trained to compensate for the evolution of discrepancies between LES and DNS data, and they may be structurally unstable, accumulate high spatial frequency errors, and diverge from the original trajectory under small

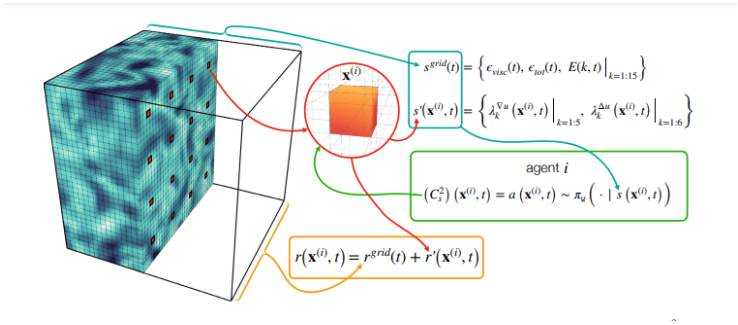


Figure 27: Diagram of MARL coupled with the flow solver. The agents compute dissipation coefficients by sampling a shared policy. The agent's state is defined by local variables, including the invariants of the gradients and Hessian of the velocity field and global variables composed by the energy spectrum. Depending on the accuracy of the LES simulation, the agents receive a reward [5].

perturbations [265][172].

To tackle these challenges, closure models have been developed using reinforcement learning. The idea behind this is that an RL algorithm can actively control the under-resolved simulation by adapting the coefficients of the closure model. The advantage of the RL method is that we do not need DNS reference data to train the model. Instead, the RL agent is trained on a cumulative reward function based on the statistics of the quantities of interest that could be obtained from DNS or experimental data. Multi-agent reinforcement learning (MARL) has been used to learn SGS models [178] (see fig. 27). In that paper, the authors place different agents on an LES simulation that takes as input local flow variables and the average energy spectrum, and the agents' action determines the coefficients of the Smagorinky model. The model is trained using remember and forget experience replay (ReF-ER) [177] and was able to reproduce DNS statistics and to generalize to different Reynolds numbers. This same approach was used later on to produce wall models for LES simulations and tested on LES simulations of turbulent boundary layers and turbulent channel flows, demonstrating the flexibility of this method to different cases and objectives.

Another interesting work applied RL to model a LES simulation's closure using a high-order discontinuous Galerkin scheme. This type of simulation would not work with the SL approach using reference filtered DNS data because the filtering on the DG scheme is a complex one that depends on the numerical fluxes across element faces [122][124][123]. In this work, the authors used a policy network based on 3D CNNs, and it was trained using proximal policy optimization (PPO) [221]. This paper improves the previous approach by only depending on local quantities, being more data efficient. This method could

learn accurate and long-term stable models, beating the analytical models.

Through these examples, we can conclude that RL constitutes a method that matches many of the desirable properties of an optimal closure model. Further investigation needs to be done to improve the generalization and scalability of these methods. As well as other advanced DL approaches for policy models that integrate more physical properties into account. Finally, these models could be extended to other problems arising from the solution of PDEs and other complex physics simulations.

3.3 Machine learning for reduced order and surrogate modeling

As seen in previous sections, the cost of simulating fluid flows is reduced by filtering techniques as done with RANS and LES. These methods have enabled the study of flows that exhibit high Reynolds numbers, which are relevant to industrial applications, flows that would be expensive to simulate using DNS. Even though the cost of simulations has reduced considerably and accurate closure models are being developed, there are cases where a further reduction in the inference speed is needed. Tasks such as design optimization, uncertainty quantification, and control require fast and many evaluations of the flow solver, and even with the current cost of LES and RANS, if we add them up, we would finish needing lots of computational hours. Because of this need, the idea of having surrogate models of these simulations that can reliably estimate quantities of interest for these tasks was born. This section discussed using ML for surrogate models in ML-assisted reduced-order models that combine machine learning with other mathematical techniques. Commonly, ML is used to learn a set of coordinates on a lower dimensional manifold that is used by other methods to evolve these variables in time, and Neural PDE surrogates, where an ML method alone, will be trained to learn the solution of PDEs produced by a numerical solver so they can produce new solutions with other parameters in a fraction of the time.

3.3.1 Machine Learning assisted Reduced Order Models

In fluid dynamics, reduced order models (ROM) rely on complex flows exhibiting a few dominant coherent structures that provide coarse but valuable information about the flow. ROMs describe these structures' evolution, providing a lower-dimensional version of the flow, even lower in dimension than RANS or LES. This makes ROMs suitable candidates for surrogate models that can quickly estimate flow quantities of interest.

The rationale behind developing a ROM consists of two stages: first, we find a set of reduced coordinates that describe the most important flow structures. Second, a differential equation model is identified where the latent co-

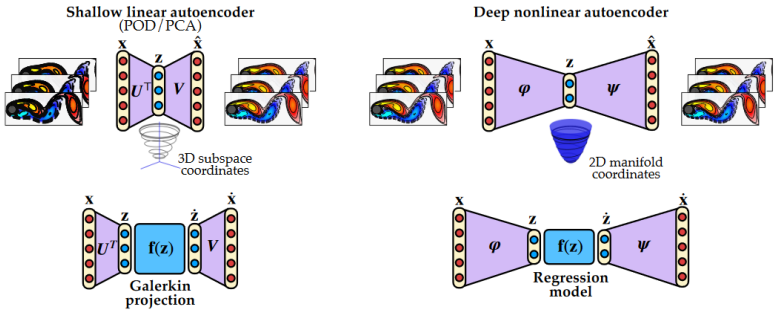


Figure 28: Diagram comparing classic projection-based ROM and ML-ROM. A non-linear manifold is obtained using AEs, and then the modes are evolved using a regression model that can be data-driven too, like DMD or SINDY [248]

ordinates evolve in time. A classic technique for developing a ROM is learning a low-dimensional basis using proper orthogonal decomposition [14] and then obtaining a dynamical system for this trial basis using Galerkin projection of the Navier-Stokes equation onto these modes. This family of methods is called projection-based ROMs and is classified as "intrusive" because it requires a numerical implementation to solve the system of ODEs resulting from it. The other type of ROM is purely data-driven, where a reduced order basis is first identified, and then a linear model is found that evolves the dynamics in time. This is the case of dynamic mode decomposition (DMD) [219][220].

Many methods aim to improve the representation of the reduced order coordinates to improve the current state-of-the-art ROM techniques using ML. The modes obtained by POD exist in a linear subspace that sometimes fails to capture the non-linearities in the Navier-Stokes equation. This becomes important on flows that exhibit multi-scale behavior, where the trajectories exist on a lower dimensional non-linear manifold. One way to obtain a non-linear reduced representation of the flow is through auto-encoders. The representation obtained by auto-encoders can reduce the dimensionality of the flow even further than POD and, at the same time, have less reconstruction error [63][173].

Projection-based ROMs have been built on non-linear manifolds learned by autoencoders. In [129], the authors trained a convolutional autoencoder and used the learned basis on the Galerkin and least-squares Petrov-Galerkin method. These methods produced lower errors than those that used the basis obtained by POD, even if they were optimal. introduced further development of this method [207], Where hyperreduction is achieved using a reduced over-collocation method and teacher-student training. Another method for achieving hyper-reduction is employing shallow masked autoencoders [112]. In figure 28, it can be found the overall difference between projection-based and data-driven ROMs.

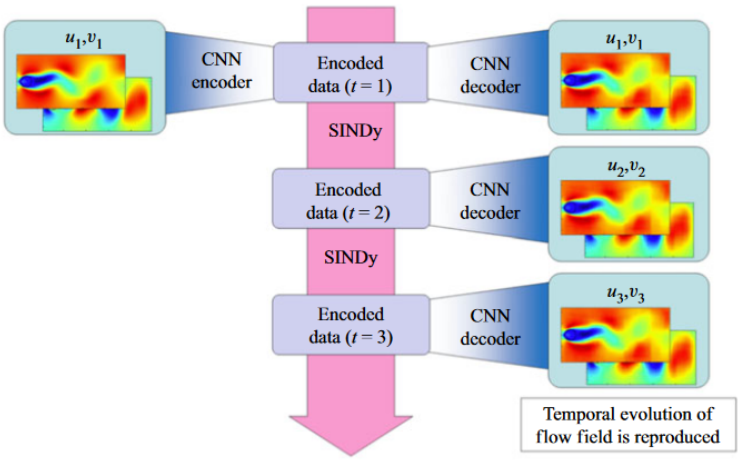


Figure 29: CNN-SINDY based ROM from [66]

Projection-based methods are required to solve a system of ODEs. However, this can be bypassed by learning a regressor that learns the temporal evolution of the latent variables. One of these methods is the sparse identification of non-linear dynamics (SINDY) [29]. This method learns a minimalistic model by fitting observed dynamics to the fewest terms in a library of candidate functions that might describe the dynamics (see Fig. 29). Convolutional AEs were used to learn a reduced set of coordinates for a flow past a cylinder, then its evolution equations were discovered using the SINDY algorithm [66].

Another Neural Network is for finding Koopman operators that linearize a non-linear dynamical system (see fig. 30). The Koopman operator is a linear infinite-dimensional operator that measures the advancement of a non-linear dynamical system and its spectral decomposition into eigenvectors and eigenvalues fully characterizes the system [116][30]. Modern data-driven techniques attempt to find a finite-dimensional approximation of the Koopman operator mainly through dynamic mode decomposition (DMD) [219]. Obtaining this linear representation becomes difficult for high dimensional dynamical systems since they would require optimization on large matrices and many snapshots of data, which would be computationally untractable. Deep Learning can learn representations of high dimensional data, and this was done in [155] where the authors used an autoencoder to learn representations of Koopman eigenfunctions and the linear matrix that evolves the eigenfunctions in time. Koopman-based frameworks have been compared to LSTMs [57] for learning ROMs of turbulence, and the results showed similar performances to LSTMs with other advantages such as less training time and data requirement but at the expense of loss of generality. Another interesting approach used autoencoders to learn

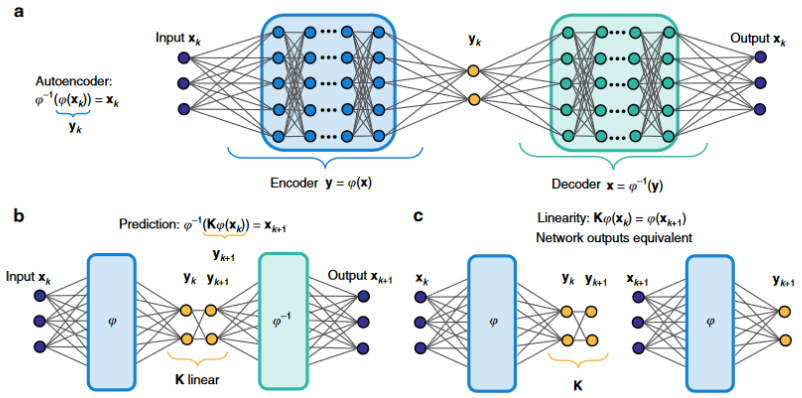


Figure 30: Deep learning framework to identify Koopman eigenfunction $\phi(x)$. The autoencoder learns intrinsic coordinates $y = \phi(x)$ and recovers $x = \phi^{-1}(x)$. An auxiliary subnet is trained to identify a linear Koopman model that satisfies $\mathbf{K}\phi(x_k) = \phi(x_{k+1})$ [155]

Koopman embeddings. The Koopman operator was approximated with a transformer model, increasing the performance of using DMD [72].

Another interesting approach is combining Galerkin projection with differentiable programming. The idea is to approximate the coefficients of a Galerkin model with a POD basis with neural networks. Then, this model is trained as an augmented Neural ODE. By doing this, the model retains physical inductive biases imposed by the differential equation form, so it needs less data and less deep models than other Neural Networks. Neural Galerkin projection has been used to learn ROMs of turbulent flows, and it exhibits numerical stability, which is difficult to achieve for analytical ROMs and fully data-driven models [164].

There are still challenges in developing truly predictive ROMs with good generalization capabilities. The sub-space of reduced dimensionality is represented through linear projection techniques such as POD/PCA and the Non-Linear version with autoencoders. The limitation of AEs is the lack of interpretability and difficulty applying them to irregular meshes and multi-scale data. Other techniques are emerging based on Neural implicit representations that learn a Non-linear representation of data that is mesh-independent [181]. The parameters of a Neural Network describe the non-linear manifold and can learn spatiotemporal representations for high-dimensional data. This approach can be used for Neural Galerkin models making very expressive ROMs for prohibitive problems with previous methods [27][38]. Other methods attempt to learn interpretable latent spaces with beta VAEs [58] and multi-scale representations with GNNs [8].

3.3.2 Neural PDE Surrogates

The last section discussed hybrid ROMs that combined ML with some prior mathematical structures. These approaches are convenient in cases with little but accurate data and well-posed problems where a set of PDEs can describe the dynamics. We might encounter abundant data in certain situations, but it's often incomplete. Alternatively, we may face a challenging scenario where we lack essential information to formulate a well-posed Partial Differential Equation (PDE) problem, or there might be no known PDE governing the system's behavior. In such cases, employing a framework that utilizes data-driven models is more suitable. These models learn the system's dynamics without needing a specific differential equation structure. They have the advantage of being less intrusive and can apply to a wide range of physical systems.

Here, we define a Neural PDE surrogate (see fig. 31) as an ML model that takes as input parameters of a dynamical system like initial state/condition, boundary conditions, and other physical parameters, and it predicts the evolution of the system in time. It differs from traditional ROMs in that the ML model is a black box that is not restricted to the governing equations of the system but instead is a surrogate of the PDE and numerical solver learned from data. The advantage of this formulation is that we can design our models depending on the degree of expressiveness needed. The model can be entirely data-driven or with added inductive biases and regularization terms as needed. This creates a route for crafting models to learn physical systems with greater generalization.

For learning fluid dynamics simulations, several deep learning models have been used. The combination of autoencoders and LSTMs appears many times in the literature to learn simulations of turbulent flows [163][277][57][173][159], these models use the autoencoder to reduce the complexity of the 3D Homogenous Isotropic Turbulence (HIT) simulation and the LSTM to learn the dynamics of the latent variable. Physics is embedded in the learning process by the addition of physics-based regularization terms [165].

Other approaches based on Autoregressive CNNs are preferred over the AE-LSTM framework because they are easier and exhibit more stable training dynamics. The idea behind AR-CNNs is that the CNN model is trained to predict the next step without passing the hidden state to the next prediction or memory, as in LSTMs, and the predicted output is used as input to predict the next one in the sequence. Models based on U-Net have been used since they can learn the multi-scale behavior better than standard CNNs [251][81]. Physics-based constraints have also been used to overcome data limitations and promote better accuracy of the predictions [71]. Other authors have demonstrated that the accuracy of the predictions improves when we learn the difference between consecutive time steps rather than the prediction of the full state by the NN (see fig. 32)[233].

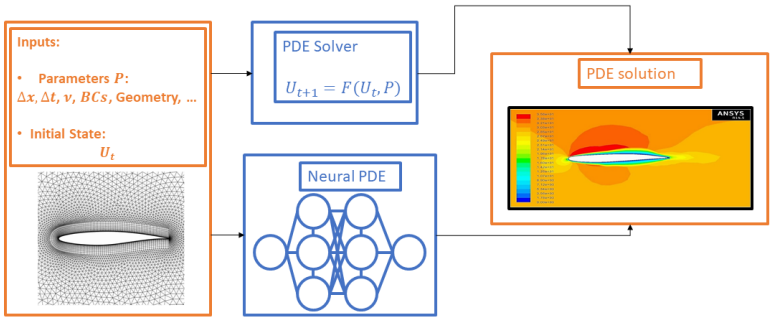


Figure 31: Diagram of how a Neural PDE works. The ML model emulates the PDE solver to predict the system dynamics given a set of conditions we would typically require to solve the equation numerically.

The previous models are fully or partially data-driven. As a consequence, they require big amounts of data to be trained. Training models this way can be costly due to the computational cost of generating the simulation data, especially for complex cases. Physics-informed approaches are attractive Neural PDE surrogate models since they require less data, can be adapted to many equations, and do not require a numerical discretization, unlike CNNs. PINNs have been used for solving the Navier-Stokes equations [199], as well as solutions to the RANS equations [56] and LES simulations [241]. Other approaches similar to PINNs modify the neural network architecture for better learning. These examples include the Deep Galerkin Method [230], Deep Petrov-Galerkin method [223], attention-based neural networks [205] and modified MLPs with Fourier features for multi-scale problems [254]. PINNs have been integrated into software packages that facilitate the implementation of ML algorithms for solving PDEs. Worth mentioning are DeepXDE [154], NVIDIA Modulus [86] and NeuralPDE.jl [287].

PINNs sometimes suffer from training instability [256] and lack of generalization when we want to cover a family of solutions of PDEs with changing parameters and initial and boundary conditions. Inspired by the task of learning the solution operator of PDEs, the Neural Operator models are designed to learn mappings between function spaces instead of learning a single function that satisfies the PDE. We can see examples of Neural Operators applied to CFD like DeepONet applied to learn bubble dynamics [145], Neural Opera-

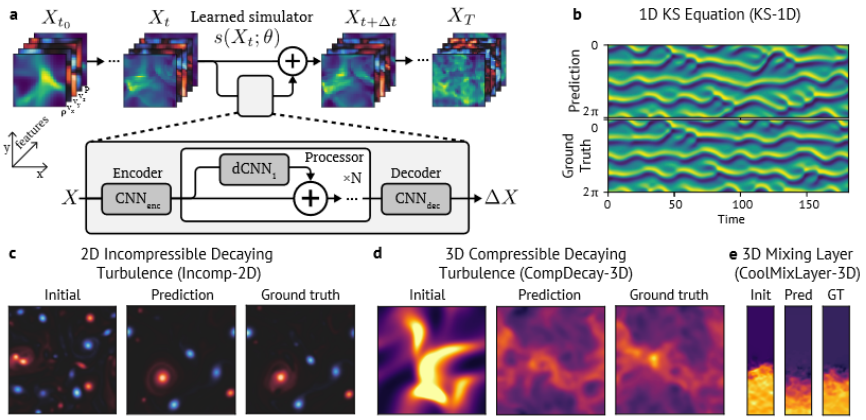


Figure 32: a) Learned Simulator: The model is trained to learn the difference between the previous and next state. In b), the predictions of the 1D KS equation can be seen, c) is a case of 2D decaying turbulence, d) is compressible decaying turbulence, and e) is from a 3D mixing layer [233].

tors for finding solution of Navier-Stokes [141][142][80], applications of LESs for weather modeling [100][184], stochastic PDEs [215] and physics-informed neural operators to parameterize PINNs [252][255][143].

Commonly, PDEs are solved on 3D Euclidean space that involves complex geometries, unstructured grids, or ensembles of particles. Convolutional and feedforward neural networks have difficulty learning in these complex domains. For this reason, there are Neural PDE surrogate models based on GNNs. Graph Networks have been used to learn complex particle fluid simulations [216], learning operators [140][139], message-passing GNNs [22], geometric Clifford algebra networks [211][23] and equivariant neural networks that incorporate symmetries for more data-efficient learning, with applications to Rayleigh-Bénard convection [250] and predicting the flow around a set of particles [227].

There is still a long road to arrive at predictive and robust surrogate models. Different problems require different methods, and some work better than others in certain cases. Considering the cost of generating the data and training for the performance obtained for applications that require narrow error tolerances and have high complexity, it would be very hard to arrive at a good surrogate model. However, as ML advances in techniques and hardware, more opportunities open to implementing better PDE learning frameworks. This requires a big effort in collecting appropriate data, establishing benchmarks, datasets, and good verification and uncertainty quantification procedures.

3.4 Machine learning accelerated DNS

Direct Numerical Simulation (DNS) enables the finding of high-fidelity solutions to the Navier-Stokes equations. With DNS, turbulent flows where all the flow structures are resolved can be simulated. For this to be possible, we must employ fine enough spatial and temporal discretizations to capture all energy-containing scales. This leads to a high computational cost that is proportional to the Reynolds number, making the cost of running such simulations prohibitive after a certain Reynolds Number (Re).

This review's third use of ML for CFD concerns "accelerating" the computational time required to perform DNS. This leads to less time to perform the same calculation, thus representing less cost. The first paradigm consists of solving the Navier-Stokes equation on a coarser grid. This will introduce some numerical errors that an ML model will correct. For example, authors in [284] developed a method for estimating derivatives on a coarse grid with better accuracy than finite differences. This approach was used for solving a 2D passive scalar advection equation. It can be trained end-to-end given a differentiable simulator, similar to [244]. In the same manner, neural networks have been used to improve the results of a fifth-order WENO method by substituting switch functions in shock-capturing methods by a NN [234]. The same authors proposed an LSTM network to correct the time-evolving error in finite difference/finite volume methods [235]. Other authors have worked on improving the accuracy of FVM simulations as well [98]. Some interesting results were found on simulations of a 2D Kolmogorov flow, where an ML model was trained to interpolate coarse-grid DNS to higher resolutions with the desired accuracy while being able to generalize to higher Re [115]. In fig. 33, It can be observed how this method works. At each time step, the NN generates a latent vector at each grid location based on the current velocity field used to output learned interpolation coefficients to calculate the velocity at the target grid points. This method of learned interpolation could also be used in LES, even with an analytical SGS model, to reduce the discretization requirements.

The second paradigm is solving the Poisson equation with deep learning. The Poisson equation is used in operator-splitting techniques, where first, the velocity field is advected, resulting in a new velocity field u^* that doesn't satisfy the incomprehensibility constraint. This velocity is used to find the pressure with the pressure-Poisson equation:

$$\frac{\Delta t}{\rho}\nabla^2 p = -\nabla \cdot \mathbf{u}^* \tag{62}$$

That is then plugged into the momentum equation to find the corrected velocity values. This step is commonly the most computationally expensive step on CFD codes. For this reason, reducing the time required to perform these computations leads to good savings. Motivated by this problem, authors in [2] substituted the Poisson solver with a CNN and tested it on a buoyancy-driven

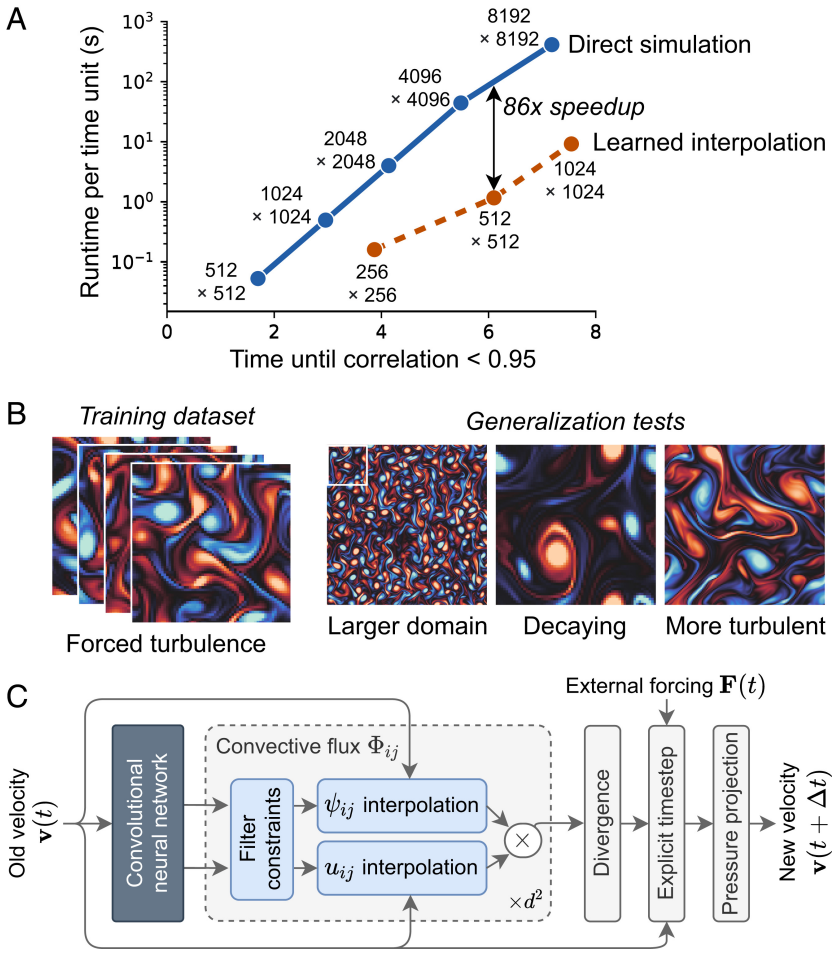


Figure 33: Diagram of ML accelerated DNS by [115]. a) plot of accuracy vs. computational cost. b) Training and validation samples: The model was trained on forced flow and could generalize to a larger domain, decaying turbulence (without forcing), and higher Re. c) Single step of the learned interpolation method.

flow. Having good results at a low Richardson number. Other authors employed CNNs to infer the solution of the Poisson equation on a cartesian grid [180]. Their approach decomposes the original Poisson problem into a homogenous Poisson problem and four inhomogeneous Laplace subproblems, which resulted in errors below 10% of the residual. A recent method used a PINN as a linear Poisson solver [156].

A third paradigm involves reducing the computational domain of the simu-

lation. This is commonly done by the specification of turbulent boundary conditions, which represent the incoming turbulent flow resulting from some previous physical process. ML models have been used to develop inflow turbulence generators that produce realistic turbulent flow, methods based on autoencoders [64], AE-LSTMs [277], GANS [111] and transformers have been implemented. Nevertheless, these models tend to suffer from instability and generalization constraints. Other questions have not been addressed properly, such as the coupling with the current CFD code and generalization to complex flows.

Finally, we want to highlight the use of large language models for performing linear algebra [37] of reinforcement learning for the automatic discovery of faster algorithms [61]. These methods could assist in the development of faster linear solvers for the Poisson equation adapted to modern HPC architectures, develop faster algorithms for the solution of the Navier-Stokes equations altogether, or develop better strategies for using resources on heterogeneous modern HPC hardware.

3.5 Fluid Dynamics Benchmarks and Datasets for Data-Driven CFD

One of the most important parts of a machine learning project is the data. Good data is important since we cannot predict what doesn't exist in the database. A good database should contain sufficient statistically independent and identically distributed samples and be balanced enough not to introduce a bias toward certain classes or labels. ML has advanced partly thanks to the public availability of standard datasets used extensively to test different ML algorithms. Benchmark datasets such as MNIST [128], CIFAR [120], and Imagenet [49] have been used to accelerate the iteration of new ideas and compare them to previous ones. We have discussed before that in fluid dynamics, the cost of data is expensive in terms of production and storage. Nevertheless, this is a scientific field with decades of work where experiments and simulations produce lots of data. The main problem with this type of data is that they are mostly decentralized, uncurated, and often inaccessible to the outside public due to their sometimes confidential nature.

The advantage of open access, good quality data would accelerate the advancement of data-driven approaches for CFD by enabling faster testing of ideas and having metrics and benchmarks to measure our models. We believe that a good database for ML for CFD should be based on the FAIR (Findable, Accessible, Interoperable, Reusable) principles defined in [262]. These principles are:

- **Findable**: The first step in (re)using data is to find them. Metadata and data should be easy to find for both humans and computers. Machine-readable metadata are essential for the automatic discovery of datasets

and services.

1. (Meta)data is assigned a globally unique and persistent identifier.
2. Data is described with rich metadata.
3. Metadata clearly and explicitly include the identifier of the data they describe.
4. (Meta)data is registered or indexed in a searchable resource.

- **Accessible**: Once the user finds the required data, she/he/they need to know how they can be accessed, possibly including authentication and authorization.

  1. (Meta)data is retrievable by their identifier using a standardized communications protocol.
     (a) The protocol is open, free, and universally implementable.
     (b) The protocol allows for an authentication and authorization procedure, where necessary.
  2. Metadata is accessible, even when the data is unavailable.

- **Interporable**: The data usually needs to be integrated with other data. In addition, the data needs to interoperate with applications or workflows for analysis, storage, and processing.

  1. (Meta)data use a formal, accessible, shared, and broadly applicable language for knowledge representation.
  2. (Meta)data use vocabularies that follow FAIR principles.
  3. (Meta)data include qualified references to other (meta)data.

- **Reusable**: FAIR's ultimate goal is to optimize data reuse. To achieve this, metadata and data should be well-described to be replicated and/or combined in different settings.

  1. (Meta)data are richly described with a plurality of accurate and relevant attributes.
     (a) (Meta)data is released with a clear and accessible data usage license.
     (b) (Meta)data is associated with detailed provenance.
     (c) (Meta)data meets domain-relevant community standards.

We have identified a group of datasets for ML for CFD that comply with the FAIR principles. One of the most widely used databases for turbulence study is the John Hopkins Turbulence database [186][136] that contains DNS simulations of forced HIT, turbulent channel flow, boundary layers, and others. In the same manner, the Vinuesa Lab at KTH made public a collection of datasets of fluid simulations [249][247] that contains examples of duct flow, flow

over an airfoil, and turbulent boundary layers. Another dataset uses a graph-based representation of steady-state Navier-Stokes that enables the application of graph-based methods to these problems [19]. This work also includes benchmark metrics based on physical metrics to assess models' performance. This dataset is also extensible by users to include more cases. Similarly, authors in [20] proposed a dataset of 2D RANS simulations over NACA airfoils to test ML models for turbulence modeling or surrogate modeling. This work also provides a set of benchmarks well-suited for the tasks. In a more general sense, there are datasets designed for related problems. PDEBench [237] that has a collection of simulations of different PDEs, which includes compressible and incompressible Navier-Stokes, and PDEArena [81] that includes data-sets of the 2D Navier Stokes, shallow water equations and Maxwell Equations. It also includes a collection of pre-trained CNNs and FNOs on these problems. In terms of universal benchmarks for ML for CFD, there are not many test cases adopted by the community, relying mainly on physical metrics related to the problem, such as drag force, turbulence statistics, mean-squared error, energy spectrum, etc. One of the worth-mentioning efforts is the open challenge in RANS turbulence modeling by the NASA symposium on turbulence modeling, which includes a series of cases to test a turbulence model. This constitutes a good test for an ML model in terms of generalization as it would have to perform across different flow configurations [1]. Further work would need to be done to establish good baseline datasets and benchmarks for ML-drive CFD that cover more flow cases and permit the development of models for different tasks. This will accelerate the research process and foster more collaborations in the same way that has been done with ML applied to other areas.

3.6 Uncertainty Quantification of Machine Learning Methods for Fluid Dynamics

Uncertainty Quantification is the science of quantitative estimation and characterization of uncertainties. In computer simulations, quantifying these uncertainties is a challenging process. The complexity of the uncertainties of simulations arises from the imprecision of the inputs (aleatory uncertainties) and limitations intrinsic in the physics models (epistemic and model-form uncertainties). The first approach in UQ is identifying sources of uncertainties and introducing an appropriate description in probabilistic terms, which then is propagated computationally. UQ aims to determine the confidence interval of the predictions and how the uncertainties affect the variability of the predictions of quantities of interest.

Verification, validation, and uncertainty quantification of models are important in CFD, but this step is commonly bypassed by ML research for CFD. For ML, this is difficult as it is difficult to guarantee convergence rates and error bounds due to the lack of theory of ML models. Also, the interpretation of ML models is a challenge due to the complexity of the models.

In Deep Learning, quantifying uncertainty associated with noisy and limited data and overparameterization is very important in predicting physical systems. The most successful family of UQ methods for DL are based on the Bayesian framework [68][69][127]. Using DL for PDE-related problems also adds the uncertainties relevant to PDE modeling, making a very complicated scenario in Fig. 34 we can observe the sources of uncertainties commonly found in SciML [192].

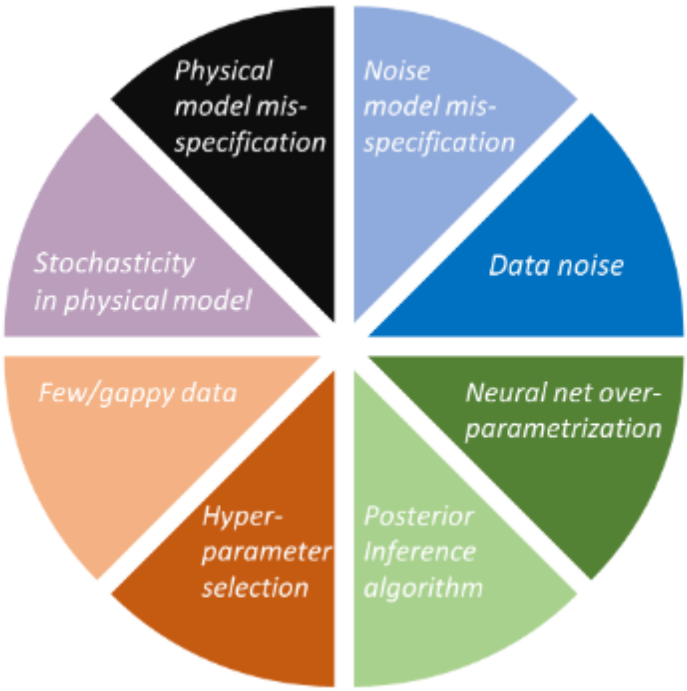


Figure 34: Total uncertainty describing the contributions of data (missing, gappy); physical models(misspecification, stochasticity); neural networks (architecture, hyperparameters, overparameterization); and posterior inference. Aleatoric uncertainty is due to noisy data. Epistemic uncertainty is due to noisy and limited data and NN overparameterization [192].

A big family of methods can be employed for UQ, depending on the nature of the task. Deep Ensemble methods [126] are good but may require training a model many times with different initializations, which is expensive. Hamiltonian Monte Carlo [174] achieves computational accuracy and efficiency for problems with small datasets. For big dataset problems, Langevin Dynamics [260], Mean-field Variational Inference [16], and Monte-Carlo dropout [69] are

more computationally efficient. For solving ODE/PDE problems with and without historical data, the use of a Bayesian Neural Network as the generator of a physics-informed GAN is recommended as surrogate [272][273]. For Neural Operator problems, the physics-agnostic functional prior can be used to quantify uncertainties caused by incomplete data, a deep ensemble to quantify model uncertainties. Inference methods such as Hamiltonian Monte-Carlo and Langevin Dynamics are recommended if the Functional prior is used. Another training model technique that includes uncertainty estimates is the stochastic weight averaging Gaussian method (SWAG) [94]. This involves training a model multiple times and using the averaged weights as the most optimal set of parameters. The SWAG method have been used to Neural PDE problems [283] and surrogate models of fluid flow [167] (see fig. 35).

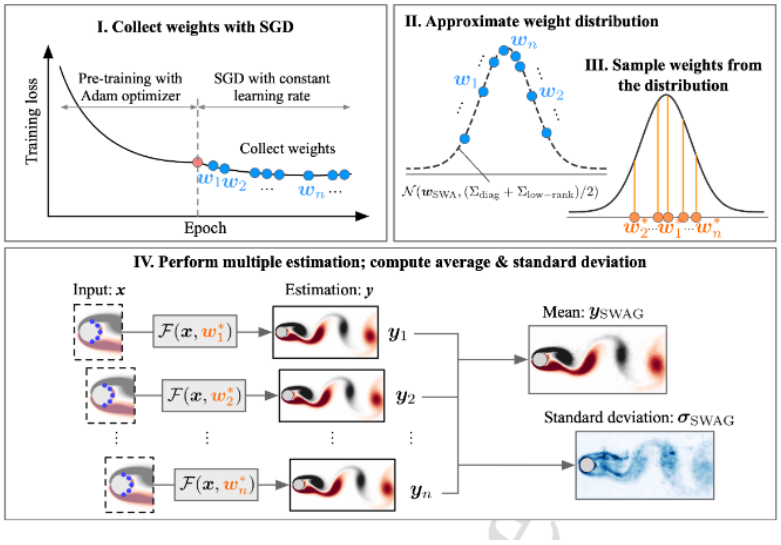


Figure 35: Diagram of SWAG framework for UQ of fluid flow surrogate model [167].

3.7 Challenges and Perspectives

After reviewing the diverse set of applications of Machine Learning to CFD, we have identified the following areas that represent current limitations and opportunities for future work on Machine Learning for CFD:

1. **The need for data**: Machine learning relies on function approximation with high dimensional parametric space. For ML to produce reliable estimations with low variance and bias, sufficient data is needed. As discussed before, CFD data is expensive to produce, and few datasets are available

for public use. More public databases that comply with FAIR principles are needed to reduce ML-CFD research costs and accelerate their advancements. So far, we have datasets covering some canonical turbulence cases and other types of flows. Datasets for multiphase flows, environmental fluids, and multiphysics problems would open up the possibilities of what ML can do for scientific discovery in CFD.

2. **Data-Model Consistency**: An ML model must be robust against training and inference data inconsistencies. In CFD, we commonly train on reference data from DNS, a resolved LES/RANS simulation, or experiments. Then we would like to apply the model in a different context, like different discretization or numerical methods where the inputs would be different than those we used to train. Handling this kind of inadaptation is critical for using ML in CFD. This problem has not been addressed extensively. Most publications on ML for CFD stop on a priori testing of the ML model alone without putting it in production for the task they were designed for with data it may encounter on a CFD application. An area of further work would be applying domain adaptation techniques to handle distribution shifts, reinforcement learning, and transfer learning.

3. **Data-Efficiency**: Due to the complexity of the data, ML models for CFD need to be able to work on scenarios with sparse and noisy data. Methods for making data-efficient models are needed. Developing techniques based on adding physics biases is an important subject for research. As we have seen, making the NNs regularize with physical laws, differential programming, and hybrid PDE/ML solvers constitute practical ways to make models learn better with more accuracy and less data. Incorporating inductive biases in the form of symmetries to make NNs respect conservation laws by design is a direction worth exploring, too.

4. **Interpretability and Explainability**: Knowing what is happening with our ML model, what is learning, what we can learn, how we can fine-tune it, and how many parameters we need are important questions with no satisfactory answers yet. The interpretability of ML models is important for CFD to select the best models for the desired task and verify and validate the models. Symbolic regression can mitigate the interpretability issue by providing an algebraic expression for our ML model. This also helps for easier implementation. Techniques such as disentangling the latent space and attention maps can help us understand what our ML is learning and how the features influence the prediction. AutoML methods could help regarding Neural architecture search for a selected task, facilitating CFD practitioners who want to apply an ML method to their simulations.

5. **Stable Surrogate Models for Multi-Scale and Multi-Physics problems**: Deep Learning is useful for designing surrogate models of fluid simulations with high accuracy and inference time much lower than the

fastest numerical simulation. However, these neural surrogates suffer from stability issues for multi-scale problems like turbulence. Finding ways to promote the stability of predictions is an important area of further development. The use of Deep Learning for surrogate models for problems that exhibit another phenomenon, such as combustion, evaporation, multiple phases, or interaction with structures, has not been tackled too much by previous research. This also adds another complexity that needs models designed to consider these multiple scenarios.

6. **Generalization**: ML models trained for CFD are normally restricted to some flow configuration, with a limited range of parameters and a certain spatiotemporal discretization. Ideally, we would want an ML model to be able to work on different geometries, computational discretization, and simulation parameters. Models such as GNNs and transformers would be more adapted to unstructured grids while also being able to capture the spatiotemporal correlations that would be hard for MLPs. Also, there is the issue of adapting a model training on certain tasks or datasets into a different downstream task. Techniques for few-shot learning and test time optimization can help in this regard and other frameworks mentioned earlier, like transfer learning, reinforcement learning, and domain adaptation.

7. **Integration of Machine Learning frameworks and CFD code**: Most CFD software is written on C, C++, or FORTRAN, and some are not open-source. ML is commonly open-source and done on frameworks written in Python, but a growing Julia ecosystem also exists. This complicates things further since a bridge between the two languages is needed. If doing things that require the ML model and CFD code to interact together, like RL or active learning, the task would become more difficult. Another limitation is that the support of automatic differentiation is limited on CFD codes, which is required for tasks requiring backpropagation through the CFD solver or an adjoint model like Neural Differential Equations. Integration between CFD code and ML needs to be advanced further. Also, implementing new solvers on fast languages like Julia would make the interface between numerical methods and ML easier and faster.

8. **Machine Learning for optimizing Scientific Computing**: ML can help improve CFD indirectly by improving how the calculations are done. ML could be used to automatically discover algorithms that make more efficient use of hardware, intelligent resource management of HPC clusters, and guide data acquisition for CFD problems. ML can also make more efficient algorithms for adaptive step-size computation, adaptive mesh refinement, and shock-wave capturing.

9. **Integration of experimental data and simulations**: In experiments, we can observe the physical phenomena in real life, but it can be difficult to extract this information quantitatively. We can have quantitative measurements with great precision with numerical simulations, but obtaining

a physically accurate model for complex cases is difficult. Taking into account experimental data for well-calibrated simulations is essential for valid predictions. This can be difficult due to the different experimental and CFD data modalities. With ML, we can do the inverse problem of obtaining equation or model parameters from experimental data and reconstructing the flow from it to obtain the information we can't extract through conventional quantitative methods. It would be interesting, for example, to reconstruct the turbulent wind profile on an actual wind farm using ML and use this information as a realistic inflow boundary condition. Or, for instance, enhancing visualization techniques for experiments using ML trained on simulated data. This way, ML can help converge numerical and experimental techniques, facilitating new scientific discoveries.

From the problems listed above, the rest of this Ph.D. work will focus on the problem of stable surrogate/reduced-order models of turbulent flows. The next chapter focuses on implementing and developing a data-driven surrogate model for turbulent flow simulations. For this purpose, two methods will be evaluated: Generative Adversarial Networks and a framework based on Autoencoders coupled with LSTM networks. The main goal is to train a model that can simulate turbulent flow using data coming from high-fidelity simulations. The model should capture the large-scale eddy behavior and be able to extrapolate outside the training data in time. If successful, these models could have applications such as inflow turbulent flow generation, design optimization of systems interacting with these flows, or control.

Part II
Neural Surrogate Models of Turbulent Flows

4 Towards Simulating Turbulence with Deep Learning

4.1 Introduction

Turbulence is a complex and ubiquitous phenomenon in fluid dynamics, often characterized by chaotic and unpredictable behavior. Its understanding and accurate simulation have been long-standing challenges in fluid mechanics, which would significantly impact the design of new technologies and understanding of natural phenomena. Traditionally, the study of turbulence has relied on experiments and computational fluid dynamics (CFD) techniques, which require solving the Navier-Stokes equations to capture the intricate details of turbulent flows. However, these methods often suffer from high computational costs and can reach high accuracy only with appropriate mesh resolution, boundary conditions, and, if needed, turbulence models. As highlighted in the review of the previous chapter, there is a potential for Machine Learning to improve the current techniques for turbulence simulation.

In recent years, advancements in deep learning have shown great promise in tackling complex and data-intensive tasks. Deep learning techniques, particularly Generative Adversarial Networks (GANs) [75] and Recurrent Neural Networks (RNNs) [74] have demonstrated remarkable capabilities in modeling and generating complex signals like images [103] and weather forecasts [225]. This study explores the potential of leveraging these powerful tools to simulate and predict turbulence.

The main objective of this research is twofold: firstly, to investigate experimentally the feasibility of using deep learning methods for simulating turbulence in fluid dynamics, and secondly, to evaluate the performance of these methods as a substitution for numerical simulation and reduced order modeling methods based on POD and projection. Specifically, we focus on two different problems: the first is generating synthetic turbulence, and the second is learning a surrogate model of a turbulence simulation. In the first part, we explore predicting synthetic turbulent signals using Generative Adversarial Networks (GANs). GANs have shown remarkable success in generating realistic and high-dimensional data by training a generator network to produce synthetic samples that resemble actual data. We apply GANs to predict turbulent signals by formulating the problem based on the Langevin equation, a stochastic differential equation commonly used to model particle motion in turbulent flows. We ex-

plore the potential of generating synthetic turbulent flow through a dedicated GAN model and evaluate the results.

The second part of this study revolves around the reduced-order modeling of Homogeneous Isotropic Turbulence (HIT) using Convolutional Long Short-Term Memory (ConvLSTM) networks. HIT is a fundamental form of turbulence that exhibits isotropy and homogeneous properties, making it a suitable target for analysis. We propose an approach that employs an autoencoder for dimensionality reduction of the turbulent flow data, followed by ConvLSTM networks to capture spatiotemporal dependencies and simulate the HIT behavior.

Throughout this research, we conduct experiments to evaluate the performance of the deep learning-based approaches mentioned. Additionally, we discuss the advantages, limitations, and potential applications of Generative Modeling and Supervised Learning techniques applied to surrogate modeling of turbulent flows. The main goal is to analyze the potential and limitations of these methods. In order to pave the way for more efficient and accurate Deep Learning approaches for turbulence simulation methods in the future.

In the subsequent sections, we present a detailed description of the problem statements for GAN-based turbulence generation and ConvLSTM-based reduced-order modeling of HIT. We then discuss the datasets, network architectures, and evaluation metrics used in our experiments. Finally, we present the results of these experiments and provide a comprehensive analysis of the outcomes, leading to meaningful discussions and conclusive remarks.

4.2 Generating Turbulence Signals with Generative Adversarial Networks

4.2.1 Problem Statement

This section aims to use generative methods to generate turbulent flow. The main idea draws inspiration from the statistical description of turbulence, where invariant statistical properties such as moments, autocorrelation functions, probability density functions, and energy spectrum characterize the flow. The methods used in this study are generative machine learning methods that aim to learn the underlying distribution of the turbulent flow and then generate new samples that belong to the distribution and exhibit the same physical and statistical properties as turbulence. We describe this problem as follows:

Problem Statement 4.1 *Being $u(t)$ a turbulent or turbulent signal drawn from a probability distribution $P_{data}(u)$. The goal is to find a model parameterized by θ, that, given a latent variable z drawn from a known prior distribution $P_z(z)$, commonly a Gaussian, serves as a mapping $f(z;\theta) = \hat{u}$ that maps the*

latent variable to an output $\hat{u}$ that belongs to a posterior distribution $P_\theta(\hat{u}|z)$ that is similar to $P_u(u)$.

In other words, the problem to be solved is finding a generative model that can generate turbulent flow with similar characteristics as the data it was trained on. Such a model is trained on a dataset of N samples, $\{u_1, u_2, ..., u_n\}$ that have a probability distribution $P_{data}(u)$. The generative model, as illustrated in Fig. 36, consists of a model parameterized by θ that maps random points of a known probability distribution to a target distribution that should be as close as possible to the target distribution. Since the samples generated by the model do not have a corresponding target in the real data distribution, the parameters of the model are found by minimizing some distance, like the Kullback-Leibler (KL) divergence [121] between the known probability distribution and the distribution of the generated samples.

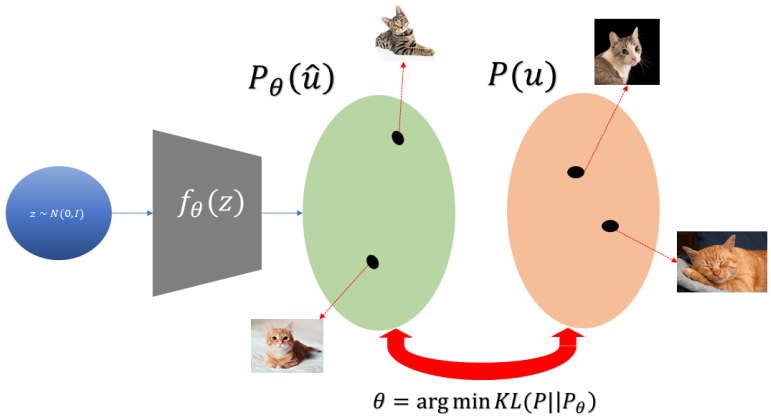


Figure 36: General diagram of a deep generative model. A generator is trained to map a sample from a prior distribution into a sample that follows the same distribution as the training dataset.

There are different types of generative models; since this work focuses on Deep Learning, the class of models that will be used to solve this problem is deep generative models (DGM). A DGM is a model that employs a neural network as a generator. DGMs have recently achieved outstanding performance in the generation of images and text to a very high degree of realism and coherence [105][271][195]. A generative model can follow one of these main approaches: Generative Adversarial Networks, Variational Autoencoders, Normalizing Flows, Autoregressive Models, and Diffusion models; some of these models were described in the previous chapter. The approach selected for this study is the generative adversarial network (GAN) [75]. The reason for choosing GANs

is that they can generate signals with a meager signal-to-noise ratio compared to VAEs and generally generate a blurry output. GANs possess lower inference time and require less computation and data for training than diffusion models [18], making GAN an appropriate choice for the target application, which is the generation of turbulent flow at a lower computational cost than numerical methods.

4.2.2 Improving GAN training with WGAN-GP

The model used in the study is a Generative Adversarial Network. As seen previously, the GAN framework consists of two networks competing until the process arrives at equilibrium. The generator network G represents a mapping from the input noise to the data space. The discriminator network D is a function that outputs a scalar representing the probability that the sample comes from p_{data} rather than the generated distribution. The discriminator D is trained to maximize the probability of assigning the correct label to training samples and the samples generated by G. Simultaneously, G is trained to minimize the probability of the generated sample being classified as fake [75]. This process of two networks trained simultaneously for one beating the other defines a minimax game with a value function $V(G, D)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{u}\sim p_{\text{data}}\,(\boldsymbol{u})}[\log D(\boldsymbol{u})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \quad (63)$$

In a practical sense, to optimize the two networks, two loss functions are needed, the discriminator and the generator loss, defined as:

$$L_{discriminator} = \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{u}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right] \quad (64)$$

$$L_{generator} = \frac{1}{m}\sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \quad (65)$$

For training the model with these losses, the discriminator's parameters are updated by maximizing the $L_{discriminator}$, and then the generator's parameters are updated by minimizing the $L_{generator}$. GANs are known to be unstable in training, making them hard to tune compared to other Deep Learning methods. The most common problems that appear during the training of GANs are:

- **Non-Convergence:** Given the coexistence of two concurrently trained networks, the challenge of achieving GAN convergence emerged as one of the earliest and arguably most formidable issues following its inception. In most cases, attaining the ideal scenario where both networks attain stability and consistently yield congruent outcomes is difficult. One plausible

explanation for this predicament is rooted in the dynamics of the generator's refinement across successive epochs, which inversely impacts the discriminator's efficacy. This phenomenon stems from the discriminator encountering difficulty distinguishing authentic instances from synthetic ones as the generator's output quality improves. Should the generator consistently prevail, the discriminator's accuracy hovers around 50%, akin to a random coin toss. This dynamic poses a significant impediment to the overall convergence of the GAN. As the discriminator's evaluative feedback progressively loses significance over subsequent epochs, generating outputs of equal probability, the generator risks compromising its quality if it persists in training based on these spurious instructional signals.

- **Vanishing Gradients:** This occurrence arises when the discriminator's performance surpasses the generator's to a considerable extent. This incongruity may result in either imprecise updates applied to the discriminator or the attenuation of these updates over time. One of the hypothesized explanations for this phenomenon is attributed to a pronounced penalization of the generator, inducing saturation within the value range after the activation function is applied, ultimately causing the gradient to diminish and causing the generator to no longer continue learning.

- **Mode Collapse:** Mode collapse occurs when the generator fails to provide a variety of outputs covering the whole probability space of the data, sticking only to one of a few modes, thus generating a low variety of samples. This happens when the generator gets stuck in local minima. In other words, it finds the output that would fool the discriminator the most. Meanwhile, the discriminator will always learn to reject this sample until the generator finds another nearby point that can fool the discriminator again. This turns into a never-ending process since the two networks overfit to exploit the short-term weaknesses of their adversary, making the networks unable to converge.

To overcome these problems, several methods have been implemented to improve the training and performance of GANs. One way is to use a different loss function that may prevent one of the aforementioned failure modes. In this study, we will train the GAN using the 1-Wasserstein distance, also called WGAN, to differentiate it from the original GAN formulation [3]. The Wasserstein distance, also known as the Earth-mover distance, can be defined as the minimum cost of transporting mass to transform the distribution q into the distribution p, the cost equal to the mass times the transport distance. The WGAN value function is defined as:

$$\min_G \max_D V(D,G) = \min_G \max_{D \in \mathcal{D}} \mathop{\mathbb{E}}_{\boldsymbol{u} \sim \mathbf{P}_{data}} [D(\boldsymbol{u})] - \mathop{\mathbb{E}}_{z \sim \mathbf{P}_z} [D(G(z)))] \tag{66}$$

Where $\mathcal{D}$ is the set of 1-Lipschitz functions, the WGAN metric is continuous and differentiable almost everywhere, making it possible to train the discriminator until optimality. In this case, the better the discriminator gets, the more reliable it gets and can continue to make the generator learn even when the discriminator is more robust. The main benefits of the WGAN are that it improves the stability of the training process, preventing mode collapse, and the WGAN metric correlates with the generator's convergence and sample quality [3]. One of the limitations of the WGAN is that it requires the discriminator to satisfy the Lipschitz constraint. To enforce this condition, the procedure proposed is to clip the network's weights in a compact bound by c, making the discriminator parameters bounded in $w \in (-c, c)$. This makes the network sensitive to the hyperparameter c, where a wrong choice of this value leads to unstable training. The weight clipping also constrains the discriminator to simple functions, limiting the model's capability for learning complex distributions. A better way to enforce the Lipschitz constraint is by gradient penalty. A differentiable function is 1-Lipschitz if and only if it has gradients with the norm at most 1 everywhere. A method to enforce this property is to add a gradient penalty regularizer that constrains the norm of the gradient of the discriminator with respect to its inputs [79]. The loss of the discriminator then becomes:

$$L = \mathop{\mathbb{E}}_{z\sim\mathbf{P}_z}[D(G(z)))] - \mathop{\mathbb{E}}_{\boldsymbol{u}\sim\mathbf{P}_{data}}[D(\boldsymbol{u})] + \lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{u}}\sim\mathbf{P}_{\hat{\boldsymbol{u}}}}\left[\left(\|\nabla_{\hat{\boldsymbol{u}}} D(\hat{\boldsymbol{u}})\|_2 - 1\right)^2\right] \tag{67}$$

Where λ is the strength of the penalty, generally set to 10. The gradient penalty term is calculated based on a random sample $\hat{\boldsymbol{u}} \sim \mathbf{P}_{\hat{\boldsymbol{u}}}$. This sampling distribution is defined by sampling uniformly along straight lines between pairs of points sampled from the data distribution $\mathbf{P}_{data}$ and the generator distribution $\mathbf{P}_g$. The motivation for this choice is that the optimal discriminator contains straight lines with gradient norm 1 connecting coupled points from the two distributions. The GANs implemented in the experiments will employ the WGAN-GP loss formulation.

4.2.3 GAN with statistical and physical constraints

Complex systems may encounter two distinct kinds of properties: deterministic and statistical. To illustrate, the Navier-Stokes equations that depict the behavior of incompressible fluid motion arise from mass and momentum preservation, constituting deterministic constraints in this context. Nonetheless, the solutions and characteristics of these partial differential equations (PDEs) reveal intricate patterns and statistics due to the complex dynamics they represent. For instance, turbulent flows exhibit non-Gaussian distributions in their velocity increments, and the kinetic energy spectrum of turbulence follows a decay rate of -5/3 within a universal range of wavenumber space. These statistical constraints pertain to the attributes of a collection of system states rather than an individual state.

Generative models for fluid flows have diverse practical applications, including expediting the creation of animated scenes in computer graphics and providing input conditions for simulations of turbulent flows using various methods like direct simulations, large eddy simulations (LES), or hybrid LES/RANS simulations. The primary motivation for this study comes from the current utilization of GANs to generate input conditions for LES and hybrid LES/RANS simulations. The quality of turbulence in the input flow significantly affects the performance of LES simulations, potentially even surpassing the importance of sub-grid scale models in LES. Such input flows are often generated through precursor simulations involving LES in a periodic domain, which can be computationally intensive and introduce artificial periodic behavior. GANs offer a promising approach to generating these input flows. They could also find applications in hybrid LES/RANS simulations, where turbulence fluctuations with specific mean fields and statistics are needed in transitional areas.

The domain knowledge can be incorporated into GANs through additional regularization terms added into the Generator loss function. Studies on enforcing statistical and physical constraints have been done before, and they proved to make the GANs produce more accurate predictions of complex systems that are governed by PDEs such as turbulent flow [266][274]. To enforce statistical constraints, the GAN loss function becomes:

$$L_c(D,G) = L(D,G) + \lambda d\left(\boldsymbol{\Sigma}\left(p_{\text{data}}\right), \boldsymbol{\Sigma}\left(p_{G(z)}\right)\right) \tag{68}$$

$\Sigma(p)$ denotes the covariance structure of a given distributing p, and $d(.,.)$ is the distance between the two covariance structures. Different distances, such as the KL divergence or the Riemannian norm, could be used. In this case, the Froebenius norm is used since it is simpler to compute and provides better stability during training [266]:

$$F\left(\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2\right) = \left\|\boldsymbol{\Sigma}\left(p_{\text{data}}\right) - \boldsymbol{\Sigma}\left(p_{G(z)}\right)\right\|_F \tag{69}$$

Deterministic constraints are added similarly. For example, let u be a quantity that is conserved, according to a certain conservation law in the form:

$$\mathcal{N}(u) = f \tag{70}$$

A GAN will be trained to emulate this physical system. To train this GAN, a regularization term is added to the generator loss:

$$\begin{aligned} L_{\text{C}}(\boldsymbol{D},\boldsymbol{G}) &= L(\boldsymbol{D},\boldsymbol{G}) + \lambda C_{\text{phys}} \\ \text{with} \quad C_{\text{phys}} &= \mathbb{E}_{Z\sim p_z(Z)}[\max(\mathcal{H}(\boldsymbol{G}(Z)),0)], \end{aligned} \tag{71}$$

Being $\mathcal{H}$ the physical constraint functional. For the conservation law presented above, the physical constraint can be computed as the norm of the residual of the equation 70:

$$C_{phys} = \mathbb{E}_{Z\sim p_z(Z)}\|\mathcal{N}(G(z)) - f\|^2 \tag{72}$$

Finally, these constraints are applied to the Generator's loss because they help the networks arrive faster at an equilibrium. When the discriminator arrives at an optimum, the gradients of the generator loss are weak, and the physical and statistical constraints help provide additional gradients that push the learning further.

4.2.4 The Langevin Equation

To primarily assess the GAN in chaotic signals turbulence, a more simple problem is tackled. In the first test, a GAN will be trained to learn a 1D stochastic process. The Langevin equation is a stochastic differential equation used to describe Brownian motion. It was proposed by the French physicist Paul Langevin in 1908. The stochastic process generated by the Langevin equation is the Ornstein-Uhlenbeck (OU) process, and its PDF evolves by the Fokker-Planck equation. In consideration of the case of homogeneous isotropic turbulence, the mean velocity is zero, and all fluid particles are statistically identical. Considering only one velocity component, denoted by *U(t)* is sufficient. The Langevin equation is the SDE:

$$dU(t) = -U(t)\frac{dt}{T_L} + \sqrt{\left(\frac{2\sigma^2}{T_L}\right)}dW(t) \tag{73}$$

Where T_l and σ^2 are positive constants. This equation can be computed through a finite difference method called the Euler-Maruyama method, which is the Euler method with the addition of a normal random variable in place of the *dW(t)* that represents a Wienner stochastic process.

$$U(t+\Delta t) = U(t) - U(t)\frac{\Delta t}{T_L} + \sqrt{\left(\frac{2\sigma^2\Delta t^2}{T_L}\right)}\xi(t) \tag{74}$$

Where $\xi(t)$ is a normalized Gaussian random variable that is independent of itself at different times and which is independent of $U(t)$ at past times, in eq.74 The deterministic drift term $(-U(t)\frac{dt}{T_L})$ causes the velocity to relax toward zero on the timescale T_L, whereas the diffusion term adds a zero-mean random increment of standard deviation $\sigma\sqrt{\frac{2dt}{T_L}}$.

$U(t)$ is a statistically stationary Gaussian Markov process in which continuous sample paths are nowhere differentiable. As a stationary Gaussian process, it is characterized by its mean, variance, and auto-correlation function, which is:

$$\rho(s) = \exp\left(-|s|/T_L\right) \tag{75}$$

If $\rho(s)$ is the Lagrangian velocity auto-correlation function, then the Lagrangian integral timescale is defined by:

$$T_L = \int_0^\infty \rho(s)ds \tag{76}$$

We could verify that eq. 75 is consistent with this definition, so the coefficient T_L in the Langevin equation is indeed the integral timescale of the process [191]. A dataset is built using three $T_l = [0.05, 0.1, 1]s$ to train the GAN. The dataset contains 5000 Langevin processes realizations for each T_l for a duration of $T = 4T_l$ and with a $\Delta t = T_l/52$. An example of the signal of the Langevin equation can be seen in Fig. 37.

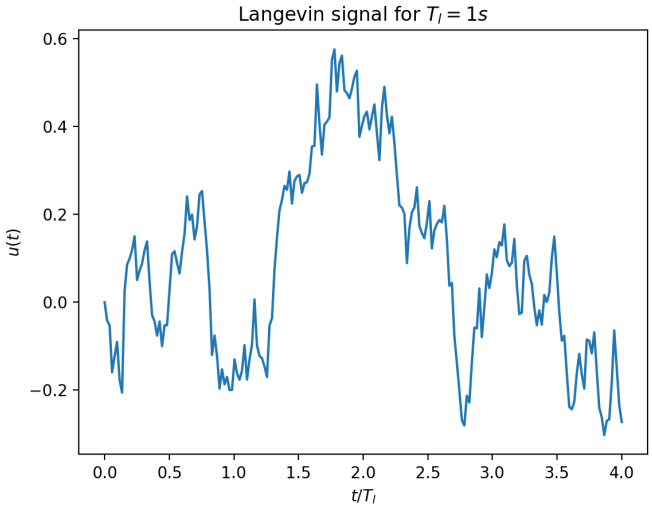


Figure 37: Sample trajectory obtained from the numerical solution of the Langevin equation.

4.2.5 RNN-GAN for the Langevin Equation

This section describes the architectural details of the GAN used to generate solutions for the Langevin equation. For the generator, an RNN model is used. This choice was made for the ability of RNNs to model sequential problems. The model used is an LSTM since it is designed to learn long-range correlations. The LSTM consists of a single LSTM cell with a hidden dimension of 256. The output of the cell is passed to a linear layer. This model can be seen in figure 38.

The discriminator is based on a CNN. It takes a sequence and outputs a score. The discriminator uses 1D convolution with a kernel of size 3 and a stride of size 2. The input layer has 256 hidden dimensions, and the following layers have $[128, 64, 32]$. The output layer is linear, and LeakyReLU activation

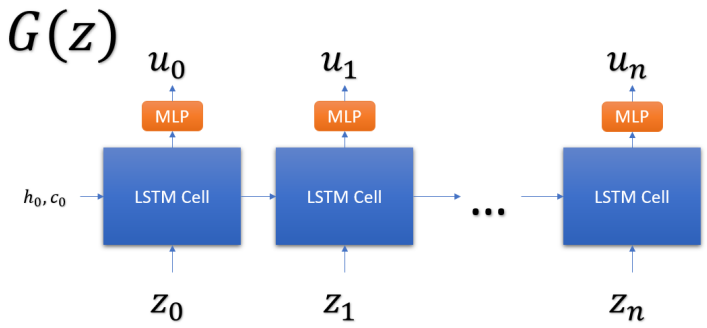


Figure 38: Generator of the RNN-GAN for the Langevin equation.

is used with a negative slope 0.2. The discriminator architecture is seen in figure 39.

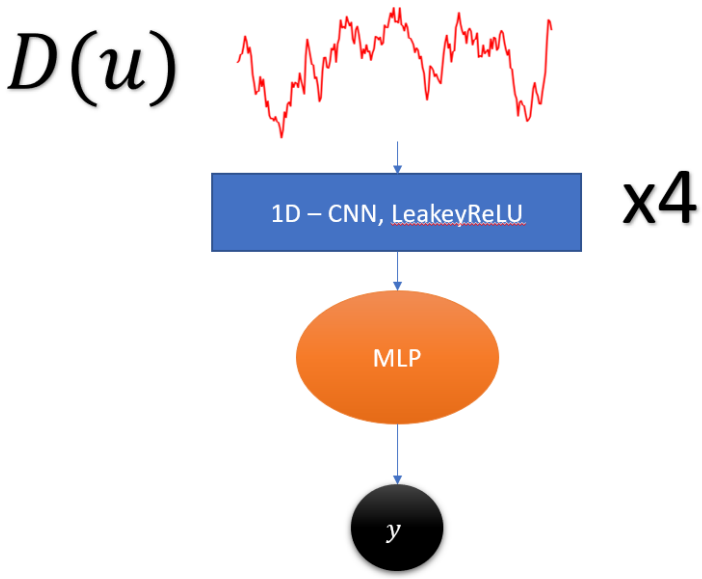


Figure 39: Discriminator of the RNN-GAN for the Langevin equation.

The generator and discriminator are trained using the WGAN-GP framework. The loss functions to be optimized are:

$$L_{disc} = \mathbb{E}[D(G(z))] - \mathbb{E}[D(u)] + \lambda_{gp}\|\nabla_{\hat{u}} D(\hat{u}) - 1\|^2 \tag{77}$$

$$L_{gen} = -\mathbb{E}[D(G(z))] + \lambda_{\mu} L_{\mu} + \lambda_{\Sigma} L_{\Sigma} + \lambda_{\gamma_1} L_{\gamma_1} + \lambda_{Kurt} L_{Kurt} \tag{78}$$

Where the constraint is based on the different order moments between the real signal u and the generated signal $\hat{u} = G(z)$:

$$\begin{aligned} Mean : L_{\mu} &= \|\mu_u - \mu_{\hat{u}}\|^2 \\ Covariance : L_{cov} &= \|\Sigma(u) - \Sigma(\hat{u})\|^2 \\ Skewness : L_{\gamma_1} &= \|\gamma_1(\hat{u})\|^2 \\ Kurtosis : L_{Kurt} &= \|Kurt(u) - Kurt(\hat{u})\|^2 \end{aligned} \tag{79}$$

The mean is the first order statistical moment defined as $\mathbb{E}(u)$, the autocovariance function is calculated as:

$$\Sigma(X) = \frac{\mathrm{E}\left[(X_t - \mu)\left(X_{t+\tau} - \mu\right)\right]}{\sigma^2} \tag{80}$$

The skewness and the kurtosis are third and fourth-order moments, respectively. The skewness quantifies the symmetry of the distribution. In this case, the skewness is enforced to be zero since the PDF of the Langevin signal is a Gaussian. The kurtosis measures the tailedness of a probability distribution of a real-valued random variable. Bothe skewness and kurtosis are computed as follows:

$$\gamma_1 = \mathrm{E}\left[\left(\frac{X-\mu}{\sigma}\right)^3\right] = \frac{\mu_3}{\sigma^3} = \frac{\mathrm{E}\left[(X-\mu)^3\right]}{\left(\mathrm{E}\left[(X-\mu)^2\right]\right)^{3/2}} = \frac{\kappa_3}{\kappa_2^{3/2}} \tag{81}$$

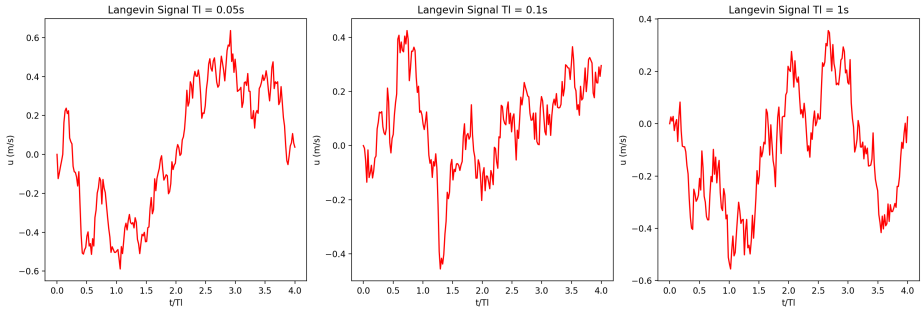
$$\mathbf{Kurt}[X] = \mathrm{E}\left[\left(\frac{X-\mu}{\sigma}\right)^4\right] = \frac{\mathrm{E}\left[(X-\mu)^4\right]}{\left(\mathrm{E}\left[(X-\mu)^2\right]\right)^2} = \frac{\mu_4}{\sigma^4} \tag{82}$$

The GAN is optimized using the Adam optimizer with $\beta_1 = 0, \beta_2 = 0.9$, and a learning rate equal to 0.0001. The batch size is 300, and the model is trained for 160 epochs. The discriminator is trained for every 5 iterations of the generator's update. The values of the regularization terms are $\lambda_{gp} = 10$ and $\lambda_{cov} = 1$.

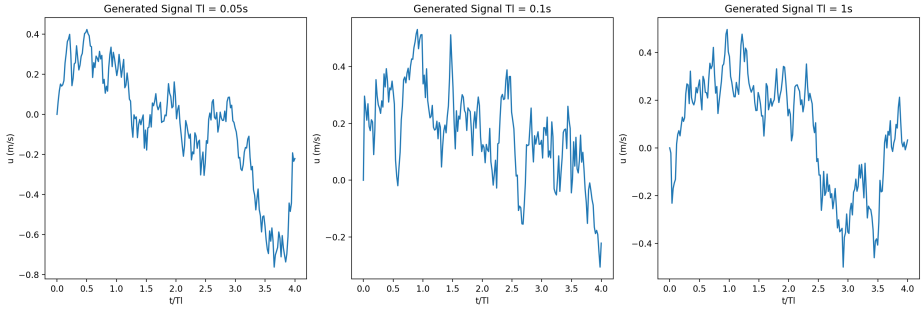
4.2.6 Results of RNN - GAN on the Langevin Equation

The results of the RNN-GAN trained on the Langevin equations are presented in this section. As previously highlighted, the RNN-GAN is trained to predict the Langevin equation for different temporal scales. Figure 40 shows a sample of the generated signals. Since the solution to the Langevin equation is a stochastic process, it is difficult to assess it qualitatively from human visual perception. A

Better method to quantify the accuracy of the predictions is through statistics. In fig. 41, the mean and variance of the real and generated signal. This picture shows that the generated signal's variance follows a similar behavior to the real one and converges to the target values, $\sigma = 0.10$.



(a)



(b)

Figure 40: Comparison between the solution of the a) Langevin equation and the b) GAN generated.

The second quantity to evaluate is the autocorrelation function calculated as:

$$\hat{R}(\tau) = \frac{1}{(n-\tau)\sigma^2} \sum_{t=1}^{n-\tau} (X_t - \mu)(X_{t+\tau} - \mu) \tag{83}$$

The autocorrelation function of the generated and actual signal and the probability density function are seen in Fig. 42. The time scale is calculated for the autocorrelation functions as their integral. The table 1 shows the generated signal's time scale and target. The relative mean-absolute-error is around $3-5\%$ for the three τ_l, meaning the networks' generator is well balanced and does not suffer from mode collapse.

The next test for the GAN is to try to generate a signal for a longer time

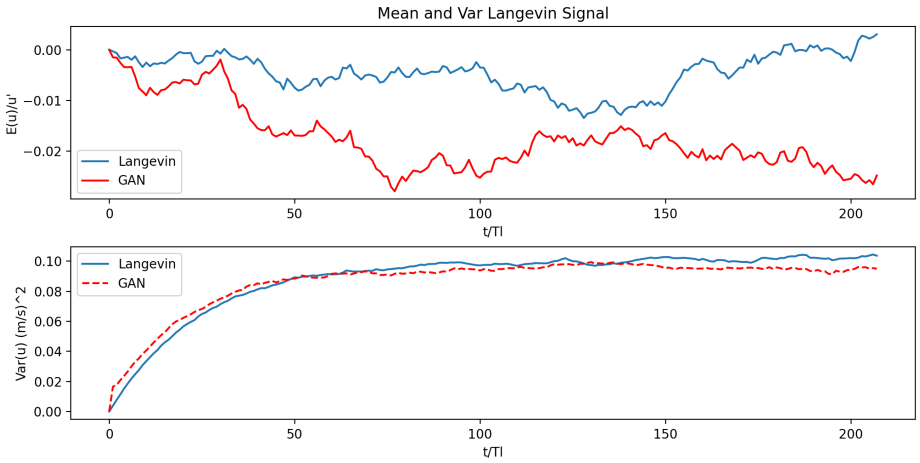
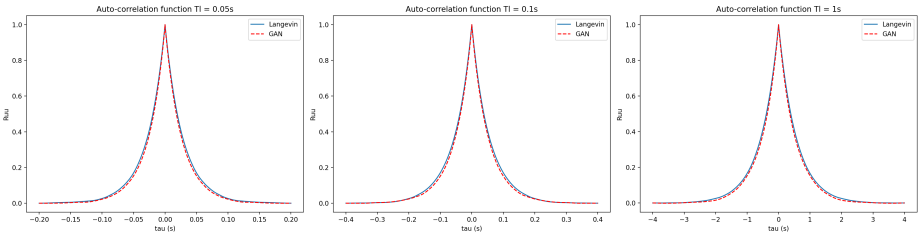


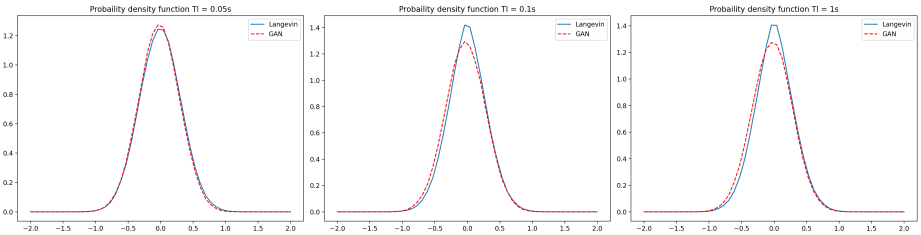
Figure 41: Mean and variance for Langevin signal.

T_l $GAN(s)$	$Tl(s)$	r-MAE
0.052	0.05	0.04
0.105	0.10	0.05
1.028	1.00	0.03

Table 1: Integral time-scale and their relative mean absolute error.



(a)



(b)

Figure 42: a) Autocorrelation function and b) PDF of the Langevin signal.

horizon. The trained generator predicts a Langevin signal with $T_l = 0.05$ for ten times the autocorrelation time. A sample of this result is shown in figure 43 and its mean and variance in fig. 44. The autocorrelation function and PDF are shown in figure 45, and it demonstrates that the functions do not deform, meaning the model can extrapolate, producing a signal with consistent statistics that could be a solution to the Langevin equation.

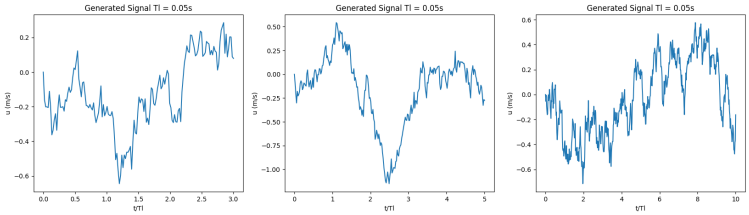


Figure 43: Langevin signal generated by the GAN up to $T = 10T_l$

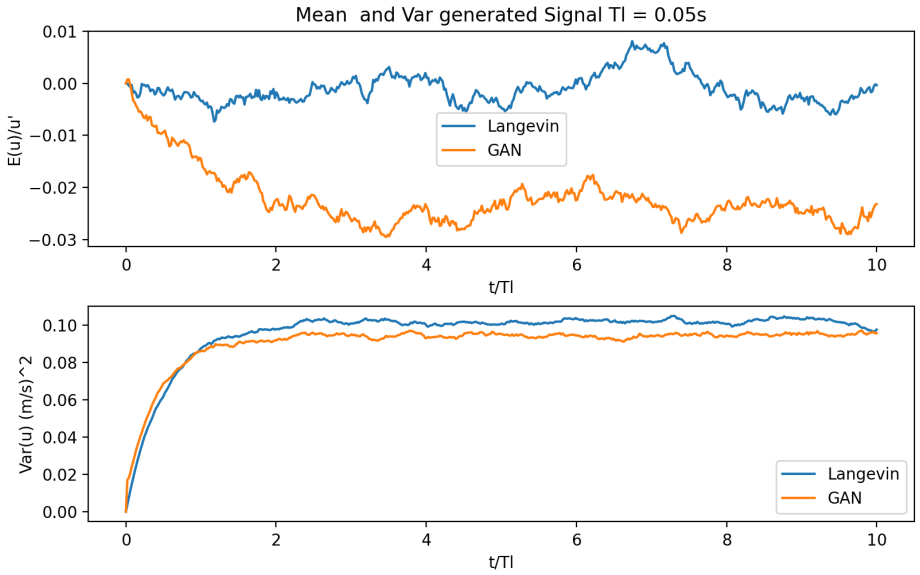
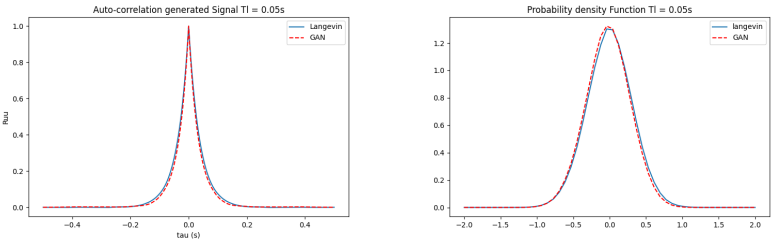


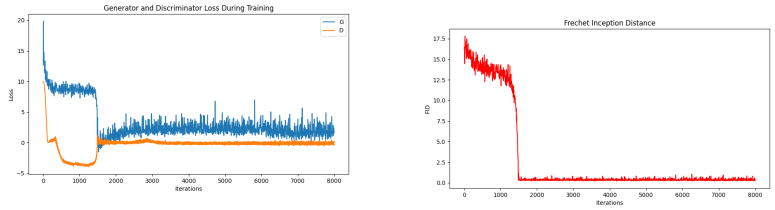
Figure 44: Mean and variance for Langevin signal generated for $T = 10T_l$.

The discriminator and generator loss are monitored during training to determine whether the GAN has fully converged, as seen in figure 46a. The loss of the Discriminator is the 1-Wasserstein distance; if this measure is 0, it indicates that the two distributions, the one of the GAN and the one obtained by solving the Langevin equation, are the same. The generator's loss is not bounded, but the expected behavior is that it is centered around some value, indicating



(a) (b)

Figure 45: a) Autocorrelation function and b) PDF of the Langevin signal at $T = 10T_l$.



(a) Training Loss of the Discriminator and Generator. (b) Frechet's Distance during training.

Figure 46: Training loss and evaluation metric.

convergence. The WGAN loss also correlates with the quality of the generated samples. To assess this, Frechet's distance is computed., which is used in GANs to measure the quality of generated images [88]. The Frechet's distance between two multivariate Gaussians is:

$$FID = \|\mu_u - \mu_{\hat{u}}\|^2 + \mathbf{Tr}(\Sigma_u + \Sigma_{\hat{u}} - 2(\Sigma_u \Sigma_{\hat{u}})^{1/2}) \tag{84}$$

It can be observed in figure 46b that Frechet's distance correlates with the WGAN loss. When the Discriminator loss converges, the quality score drops and is maintained close to the minimum. The values of the WGAN loss and FID are reported in the table 2, the mean after convergence is reported plus its standard deviation.

Disc. Loss	FID
-0.07 ± 0.17	0.37 ± 0.11

Table 2: 1-Wasserstein distance (left) and Frechet's distance (right) during GAN's training.

4.2.7 Synthetic Turbulence Generation

After evaluating the feasibility of Generative Adversarial Networks in generating pseudo-turbulent 0D trajectories governed by the Langevin equation, the next step is to use generative models to generate synthetic turbulent fields. Synthetic turbulence generation has been used before DNS to study particle dispersion and acoustic propagation in Homogenous Isotropic Turbulence. Examples of this can be found in the Literature by Kraichnan [118], who developed a method to generate a divergence-free synthetic turbulence field using a sum of random Fourier Modes. This technique has continued to be used on many applications for initializing DNS simulations [206] and generating inflow boundary conditions [267]. This study will use a technique derived from the original Kraichnan approach to generate instantaneous synthetic turbulence data and train a Generative Adversarial Network designed to substitute this method. The Goal for this is not to propose a new method for synthetic turbulence generation but to take advantage of the relatively low cost of obtaining synthetic data to assess the capacity of generative models to infer new turbulent flows from scratch so that they could be applied later on turbulent flow data pertinent to realistic applications.

As stated earlier, the method used to generate the training data extends the Kraichnan approach. When applied to discrete grids, Kraichnan's method originally had a mass conservation issue. This problem was alleviated by Saad et al. [214], and it is the method used to generate the data. Given a spatially varying velocity field $\boldsymbol{u}(x, y, z) \equiv (u, v, w)$, its Fourier series representation at point $\mathbf{x}$ is:

$$\boldsymbol{u}(\boldsymbol{x}) = 2 \sum_{m=1}^{M} q_m \cos\left(\kappa_m \hat{\boldsymbol{k}}_m \cdot \boldsymbol{x} + \psi_m\right) \hat{\boldsymbol{\sigma}}_m \tag{85}$$

Where M is the number of modes, q_m is the amplitude, $\boldsymbol{k}_m$ is the mth wave number, $\hat{\boldsymbol{k}}_m \equiv (k_{x,m}, k_{y,m}, k_{z,m})$ is the unit direction vector associated with $\boldsymbol{k}_m$ and $\hat{\boldsymbol{\sigma}}_m \equiv (\sigma_{x,m}, \sigma_{y,m}, \sigma_{z,m})$ is a unit direction vector and ψ_m is a phase angle. The method aims to generate a series of M modes at an arbitrary point in space; the method samples a random $\hat{\boldsymbol{k}}_m$ and ψ_m with $_m$ being drawn from the energy spectrum where $q_m = \sqrt{E(\kappa_m)\Delta\kappa}$. The remaining unknown is the direction vector $\hat{\boldsymbol{\sigma}}_m$ that can be obtained by imposing the divergence-free condition. In the Kraichnan approach, the continuous divergence-free condition is used to compute $\hat{\boldsymbol{\sigma}}_m$:

$$\nabla \cdot \boldsymbol{u} = -2 \sum_{m} q_m \kappa_m \hat{\boldsymbol{k}}_m \cdot \hat{\boldsymbol{\sigma}}_m \sin\left(\kappa_m \hat{\boldsymbol{k}} \cdot \boldsymbol{x} + \psi_m\right) = 0; \quad \forall m \in \{0, 1, \ldots, M\} \tag{86}$$

This constraint can be satisfied by defining the vectors $\hat{\boldsymbol{k}}_m$ and $\hat{\boldsymbol{\sigma}}_m$ to be orthogonal:

$$\hat{\boldsymbol{k}}_m \cdot \hat{\boldsymbol{\sigma}}_m = 0, \quad \forall m \in \{0, 1, \ldots, M\} \tag{87}$$

This condition is valid only if the discretization is fine enough so the Fourier Series approximation is close to the continuous limit. For coarser grids, this model leads to divergent velocity fields. To promote the discrete divergence-free constraint, first, the discrete divergence equation is considered in a staggered grid:

$$\begin{aligned} \nabla_d \cdot \boldsymbol{u} = \frac{u_{i+1/2,j,k} - u_{i-1/2,j,k}}{\Delta x} + \frac{v_{i,j+1/2,k} - v_{i,j-1/2,k}}{\Delta y} \\ + \frac{w_{i,j,k+1/2} - w_{i,j,k-1/2}}{\Delta z} \end{aligned} \tag{88}$$

Where:

$$\begin{aligned} u_{i+1/2,j,k} &= u\left(x + \frac{\Delta x}{2}, y, z\right) \\ &= 2\sum_{m=1}^{M} q_m \cos\left[\kappa_m k_{x,m}\left(x + \frac{\Delta x}{2}\right) + \kappa_m k_{y,m} y + \kappa_m k_{z,m} z + \psi_m\right] \sigma_{x,m} \end{aligned} \tag{89}$$

After substitution, the following was obtained:

$$\nabla_d \cdot \boldsymbol{u} = -2\sum_{m=1}^{M} q_m \hat{\boldsymbol{\sigma}}_m \cdot \tilde{\boldsymbol{k}}_m \sin\left(\boldsymbol{k}_m \cdot \boldsymbol{x} + \psi_m\right) \tag{90}$$

Where:

$$\begin{aligned} \tilde{\boldsymbol{k}}_m \equiv \left(\tilde{k}_{m,x}, \tilde{k}_{m,y}, \tilde{k}_{m,z}\right) = \left[\frac{2}{\Delta x}\sin\left(\frac{1}{2}\kappa_m k_{x,m}\Delta x\right),\right. \\ \left.\frac{2}{\Delta y}\sin\left(\frac{1}{2}\kappa_m k_{y,m}\Delta y\right), \frac{2}{\Delta z}\sin\left(\frac{1}{2}\kappa_m k_{z,m}\Delta z\right)\right] \end{aligned} \tag{91}$$

To enforce the divergence-free condition, the following equation is used as a constraint:

$$\hat{\boldsymbol{\sigma}}_m \cdot \tilde{\boldsymbol{k}}_m = 0, \quad \forall m \in \{0, 1, \ldots, M\} \tag{92}$$

This is accomplished by setting the modal direction vector to be:

$$\hat{\boldsymbol{\sigma}}_m = \frac{\hat{\boldsymbol{\zeta}} \times \tilde{\boldsymbol{k}}}{|\hat{\boldsymbol{\zeta}} \times \tilde{\boldsymbol{k}}|} \tag{93}$$

Where $\hat{\boldsymbol{\zeta}}$ is randomly drawn from a uniform distribution. In the end, if the grid spacing approaches zero, $\tilde{\boldsymbol{k}}_m$ will approach $\boldsymbol{\kappa}_m \hat{\boldsymbol{k}}_m$ yielding the correct infinitesimal limit condition. The step-by-step procedure to generate a synthetic turbulent velocity field is described in algorithm 1.

Algorithm 1: Algorithm for generating Synthetic Turbulence on a 3D grid

Input: $M, (L_x, L_y, L_z), (N_x, N_y, N_z), E(k)$, Input data in respective order: number of modes, Domain's dimension, Grid resolution, and Input Spectrum.

Output: Velocity field $\mathbf{u}(x, y, z)$

$\kappa_0 \leftarrow \max\left(\frac{2\pi}{L_x}, \frac{2\pi}{L_y}, \frac{2\pi}{L_z}\right)$ `/* Compute minimum wave number */`

$\kappa_{max} \leftarrow \max\left(\frac{\pi}{\Delta x}, \frac{\pi}{\Delta y}, \frac{\pi}{\Delta z}\right)$ `/* Compute Maximun Wave number */`

$\kappa_m \leftarrow \kappa_0 + \frac{\kappa_{max}-\kappa_0}{M}(m-1)$ `/* Generate a list of` M `equidistant modes */`

foreach κ_m **do**

`/* With` $\varphi_m = \mathcal{U}(0, 2\pi)$ `and` $\theta_m = \cos^{-1}(t)$ `being` $t = \mathcal{U}(-1, 1)$. `*/`

$k_{x,m} = \sin(\theta_m)\cos(\varphi_m)\,; \quad k_{y,m} = \sin(\theta_m)\sin(\varphi_m)\,;$

$k_{z,m} = \cos(\theta_m)$

$$\tilde{\boldsymbol{k}}_m = \left[\frac{2}{\Delta x}\sin\left(\frac{1}{2}\kappa_m k_{x,m}\Delta x\right), \frac{2}{\Delta y}\sin\left(\frac{1}{2}\kappa_m k_{y,m}\Delta y\right), \frac{2}{\Delta z}\sin\left(\frac{1}{2}\kappa_m k_{z,m}\Delta z\right)\right]$$

$\zeta_{x,m} = \sin(\theta_m)\cos(\varphi_m)\,; \quad \zeta_{y,m} = \sin(\theta_m)\sin(\varphi_m)\,;$

$\zeta_{z,m} = \cos(\theta_m)$

$\hat{\boldsymbol{\sigma}}_m = \frac{\hat{\boldsymbol{\zeta}} \times \tilde{\boldsymbol{k}}}{|\hat{\boldsymbol{\zeta}} \times \tilde{\boldsymbol{k}}|}$

$\psi_m = \mathcal{U}(-\pi/2, \pi/2)$

end

return $\boldsymbol{u}(\boldsymbol{x}) = 2\sum_{m=1}^{M} q_m \cos\left(\kappa_m \hat{\boldsymbol{k}}_m \cdot \boldsymbol{x} + \psi_m\right)\hat{\boldsymbol{\sigma}}_m$

The dataset that will be used to train the GAN is generated using the procedure described in algorithm 1. The spectrum that will be used is the Von-Karman Pao spectrum [7]. For this spectrum, 500 boxes of isotropic turbulence are generated with a size of $L = 9\frac{2\pi}{100}$ and a resolution of 128^3 using 1000 Fourier modes, a sample of this velocity field can be observed din Fig. 47.

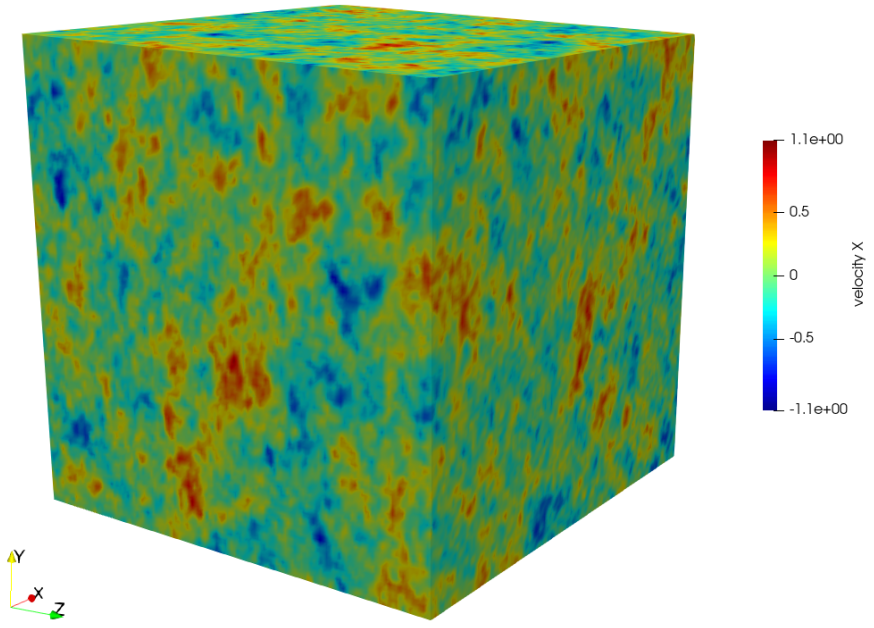


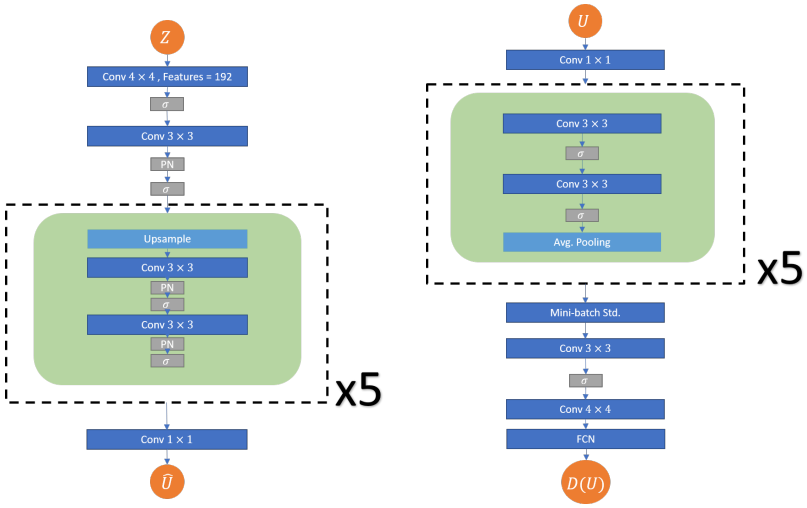
Figure 47: Sample velocity field generated with the proposed synthetic turbulence method.

4.2.8 GAN for generation of 2D slices of synthetic turbulence

The first test of the GAN for synthetic turbulence is to try to generate 2D slices of the 3D velocity field, and this problem is slightly different than the one of the Langevin equation since the goal is to generate instantaneous velocity fields without time dependence. For this reason, the methods that will be presented here are drawn from image generation techniques. Training data for this task will be selected from one of the turbulence spectrums. The VKP spectrum will be used in this case. The GAN architecture will be inspired by one of the models that have achieved good results in image generation tasks. The generator and discriminator architectures are built using the guidelines of the "progressive growing of GANs" paper [103]. This GAN uses a convolutional Generator and Discriminator. The core idea is that the Generator network starts with the first layer generating a 4×4 feature tensor and progressively increases the resolution by a factor of 2. With progressive low-resolution layers being trained first, and as training progresses, the other layers are added; this makes training easier and more stable because more time is spent on learning larger structures, and then the last stage of learning is devoted to refining the details of the image. The Discriminator does the process of the Generator inverted by reducing the resolution through each layer; the last layer uses a standard deviation layer where the mini-batch standard deviation is added as an additional feature map to the last convolutional layer. This layer increases the variation of the generated samples and prevents the GAN from generating only a subset of the training dataset. In progressive growing, the layers are added step by step, but we opt to train all the layers simultaneously for simplicity, and we found no difference in convergence; discarding progressive training has also improved the results for image generation tasks [104]. The activation used is the LeakyReLU activation with a negative slope of 0.2, and the Generator network uses a pixel-wise normalization layer; this prevents the values of the inner representations of the Generator and Discriminator from blowing up. The pixel-wise normalization is implemented as follows:

$$b_{x,y} = a_{x,y} / \sqrt{\frac{1}{N} \sum_{j=0}^{N-1} \left(a_{x,y}^{j}\right)^2 + \epsilon} \tag{94}$$

In this equation, N is the number of features, $a_{x,y}$ is the unnormalized feature vector, and $\epsilon = 10^{-8}$ is a value to prevent numerical errors. The networks are trained using the Adam optimizer with a learning rate of $\alpha = 2 \times 10^{-4}$ and $\beta_1 = 0.0, \beta_2 = 0.99$. The GAN is trained using the WGAN-GP loss as in the previous section. Figure 48 describes the Generator and Discriminator architectures. The resolution is upsampled using nearest-neighbor interpolation, and the downsampling is done by average pooling.



(a) (b)

Figure 48: Diagram of a) Generator and b) Discriminator used to generate 2D slices of Synthetic Isotropic Turbulence.

4.2.9 Results of GAN on 2D slices of synthetic Turbulence

The GAN for 2D synthetic turbulence was trained on slices of the 3D velocity field. 500 velocity fields generated with the VKP spectrum were split on the z axis. These slices are used during training, resulting in a collection of $64,000$ 2D velocity fields with a resolution of 128^2. The model took around $300,000$ iterations to arrive at the convergence of the loss functions. Once the model is trained, the Generator can be used to sample random velocity fields, as shown in Fig 49.

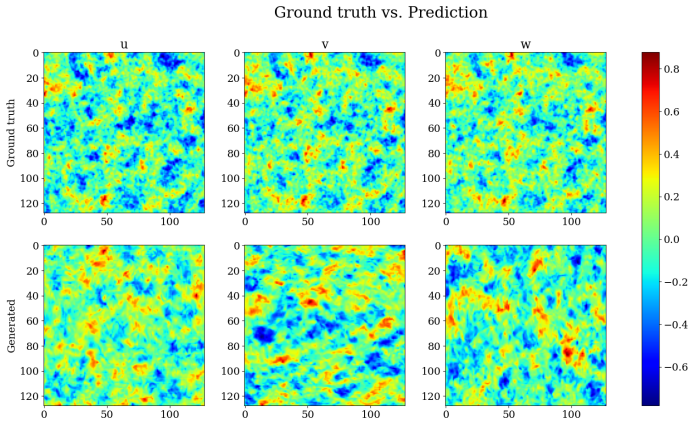


Figure 49: Samples of synthetic turbulence corresponding to a VKP spectrum generated by the GAN.

To characterize the performance of the GAN, we cannot compute a metric like a reconstruction MSE between the velocity fields due to the randomness of the generation process. We chose to evaluate the performance in terms of statistics of the generated turbulent velocity fields. First, the kinetic energy spectrum is computed along one axis. This is done by taking the Fourier transform of the velocity spatial correlation tensor:

$$\begin{aligned} R_{ij}(r) &\equiv \langle u_i(x, y_i) u_i(x + r, y_i) \rangle \\ \phi_{ij}(k_x) &= \frac{1}{2\pi} \int R_{ij}(r) e^{-ik_x r} dr \\ E(k_x) &\equiv 2\pi k_x^2 \sum_i \Phi_{ii}(k_x) \end{aligned} \tag{95}$$

Since the flow is supposed to be isotropic, the statistics are the same in all directions, so the energy spectrum is then averaged over all the spatial locations in y. In Fig. 50 it can be observed the kinetic energy spectrum compared between the GAN and the ground truth data 50. From the spectrum, it can be observed that the GAN well represents the scales present at the resolution studied, and the relative MSE between spectrums is computed, yielding a value of

0.45. An error of 45% may be a large value for the results observed; however, this error could be explained by the oscillations seen in the spectrum of the velocities produced by the synthetic method. In this case, the GAN generates a velocity field with a smoother energy spectrum, but the energy distribution through the scales seems perceptually the same. For this reason, other metrics need to be used to evaluate the performance further.

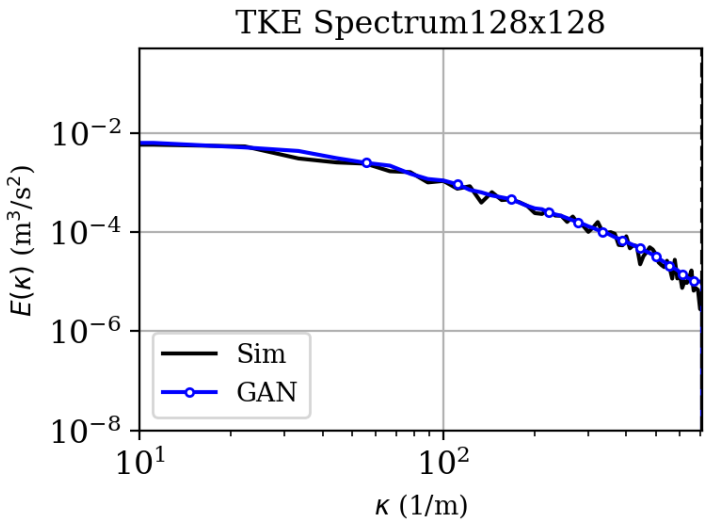


Figure 50: Kinetic energy spectrum for the GAN and synthetic turbulent flow. "Sim" corresponds to the flow spectrum obtained with synthetic turbulence simulation, and "GAN" corresponds to the spectrum of the velocity field sampled by the Generator network.

Next, to describe the turbulent velocities in statistical terms, we compute the estimation of the probability density functions of the velocity. The PDFs of the generated u and the training data can be seen in Fig. 51. There is a slight offset of the PDF of the generated velocities. To quantify this shift, the Wasserstein distance is calculated, which intuitively can be interpreted as the minimum cost of turning one probability into another by transporting points from one to the other. For this case, the Wasserstein distance between the two PDFs is **0.043**.

Finally, the turbulent kinetic energy is computed, which is the total energy produced by the fluctuating part of each component of the velocity:

$$\begin{aligned} u_i' &= u_i - \bar{u_i} \\ TKE &= 1/2 \sum_i u_i'^2 \end{aligned} \tag{96}$$

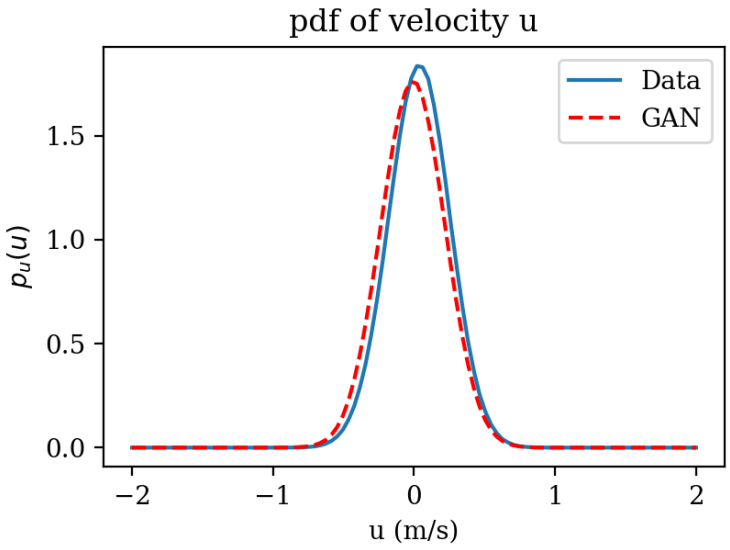


Figure 51: PDFs of velocity field generated by GAN and synthetic turbulence generator.

The relative MSE between the ground truth's TKE (0.1658) and the generated velocities (0.1852) is 0.01. The summary of metrics evaluating the GAN performance in the performance of generating 2D turbulence is summarized in table 3.

Metric	Ground Truth	GAN	Error
$E(k_x)$	-	-	0.45
WD	-	-	0.043
TKE	0.1658	0.1852	0.01
Wall Time	5.5 s	5.0 ms	-

Table 3: Summary of metrics for evaluating GAN on synthetic turbulence generation.

4.2.10 Conclusions on GANs for generating turbulence

We have built and tested GANs for the task of generating turbulent signals, first in 0D with a dataset obtained by solving the Langevin equation. This is similar to the signal that results from measuring the velocity of a point inside a turbulent flow and then on 2D slices of synthetic turbulence obtained by the Fourier method. Through the experiments, it could be verified that GANs are good at learning complex distributions and sampling from them in an efficient

manner. The GANs used in the study were able to capture complex features present on the high wave number frequencies, which correspond to the smaller scale structures of turbulence, something that is difficult for neural networks to do when trained on pixel-wise-based norms in supervised learning. Due to the approach's stochastic nature, evaluation metrics based on statistics were employed to characterize its performance. This verified that the GAN could correctly generate new samples unavailable in the training dataset with the same statistical properties.

However, despite its learning capabilities, GANs have some limitations that hinder their applicability to the problem of fluid flow simulation. The main drawback lies in the difficulty of training; many configurations have to be tried before arriving at one that converges, and this convergence can only be assessed by observing that the values of the losses don't change for a period of time, and by observation of the generated samples which should be in agreement with the domain knowledge. Another one is that since the training is a min-max process, the training dynamics are more complex than supervised learning methods in that the loss values can often arrive at a plateau for a given period of time and then suddenly improve; this makes it hard to define a general early stopping strategy and is heavily problem-dependent. In addition, GANs can fall into failure modes like mode collapse, which greatly affect the model's generalization. Regarding computational cost, GAN sampling can be very efficient once the model is trained, as seen in table 3. Producing 128 2D slices takes around 5.5 ms while the same takes 5.5 s using the synthetic turbulence approach. The main drawback in this sense is the training cost the model requires. First, there is the cost of obtaining the data in the case of turbulent flows; this data needs to be produced either by experiments or numerical methods. Second, GANs need a large amount of data, which may be prohibitive for cases relevant to engineering applications or the scientific community's interest. Third, training can be a lengthy process; for the case of 2D synthetic turbulence, 64,000 samples were used for training and took 300,000 iterations for a duration of around 4 days, being trained using 4 NVIDIA V100 GPUs. In addition, the time it consumes to tune the network hyperparameters has to be considered, which adds overhead in computing hours. This is a significant computational cost that limits the range of applications of this method. This is bad news for generalization since we dealt only with a distribution that belonged to one spectrum; if one wanted to generate synthetic turbulence for arbitrary spectrums, it would require more data and, thus, more time.

For these reasons, we do not proceed with generative modeling of turbulent flows in 3D and space-time since this would add an additional layer of complexity that has not been achieved for generative models; this could be part of an exciting research direction. Recent advances in GANs like StyleGAN [105] and new generative models based on diffusion processes [90] open new possibilities that could be explored for the generative modeling of turbulence, for example, diffusion probabilistic models have already been employed for generating tra-

jectories of lagrangian turbulence, similar to the Langevin equation [135]. For the moment, these questions are left open for future work. In the next part, we will tackle the problem of reduced-order modeling of a turbulent flow using supervised learning. This is a problem that, due to the dimensional complexity and data availability, would be difficult to solve using GANs or other generative approaches. We will employ Autoencoders to learn a latent representation and a recurrent neural network model to learn the temporal evolution of the flow.

4.3 Recuded order modeling of Homogenous Isotropic Turbulence with ConvLSTM

The interest in reducing the computational complexity of the models used to simulate turbulent flows is rooted in the prohibitive costs of the full-order models. Predicting the evolution of the fluid state requires finer temporal and spatial discretizations as the Reynolds number increases. This results in large linear systems with many degrees of freedom requiring important computational power. For many applications, applying direct numerical simulations is unfeasible for the time and money this implies and because more than one simulation is often required to explore the design space and assess risk. Model order reduction is the set of techniques designed to reduce the complexity of the full-order models. These methods can predict the quantities of interest of the physical system at the degree of accuracy needed for the targeted application, with fewer computational resources, while respecting the system's conservation laws and other properties. Projection-based reduced order modeling techniques are a family of methods used for this task. Since these methods consist of projecting the system's equations onto a lower dimensional space previously identified from solutions of the full-order model, they ensure that the ROM inherits the physical properties of the equations. However, there are still two important challenges for projection-based Reduced-order-models (ROMs): one is the identification of the lower-order basis for advection-dominated problems such as turbulent flows, and the other is related to their intrusive nature, meaning that the system that results from applying the equations to the lower dimensional space should be solvable.

Machine Learning allows the designing of non-intrusive reduced order models for advection-dominated problems. This can be achieved by learning from data from a non-linear lower-dimensional manifold. Instead of projecting the equations into this new subspace, a second model learns the system's dynamics. The following sections assess the advantage of using neural networks to learn to simulate turbulent fluid flows. This question has been treated previously in publications that applied Deep Learning techniques to learn flow simulations with a reduced complexity [163][63][129]; the purpose of this work is not to provide a new method to accomplish the aforementioned objective but to study through the reproduction the challenges and limitations of applying Deep Learning techniques originally designed for computer vision and Natural Language Processing

for the reduced order modeling of turbulent flows. The task that will be studied is the reduced-order modeling of a Homogeneous Isotropic Turbulence simulation; given limited data obtained from a DNS simulation, the goal for the ML model is to generate new data that conserves the system's physical properties.

4.3.1 Problem Statement

The problem is the reduced-order modeling of turbulence simulations using supervised learning methods. The first stage is the learning of a model that projects the full-order solution to a non-linear reduced-order space:

Problem Statement 4.2 *Being* $\boldsymbol{U}$ *a set of solutions of a given PDE,* $\frac{d\boldsymbol{U}}{dt} = \mathcal{F}(\boldsymbol{U}, \boldsymbol{x})$ *with specific initial and boundary conditions solved in a domain* $D \subset \mathbb{R}^{d_u}$*. The goal is to find a model* f *parameterized by* θ *that projects this solution to a set of latent variables* $\boldsymbol{V}$ *that exist on a lower dimensional space* $D' \subset \mathbb{R}^{d_v}$*, being* $d_v << d_u$*. The model should be able to project points from the latent space to the full-order space through an inverse mapping* $f^{-1} : \boldsymbol{V} \times \theta \rightarrow \boldsymbol{U}$*. The parameters of this model are found through empirical risk minimization of the reconstruction error:*

$$\theta = \arg\min_{\theta} \mathbb{E}\left[\| \boldsymbol{U} - f^{-1}(\boldsymbol{V}; \theta)\|\right] \tag{97}$$

The second stage is finding the model that learns the system's dynamics on the latent space:

Problem Statement 4.3 *Being* $\boldsymbol{U}$ *a set of solutions of a given PDE,* $\frac{d\boldsymbol{U}}{dt} = \mathcal{F}(\boldsymbol{U}, \boldsymbol{x})$ *with specific initial and boundary conditions solved in a domain* $D \subset \mathbb{R}^{d_u}$*. This set of solutions is sampled at regular time intervals* Δt*. A model that can learn the system's dynamics in the latent space* $\boldsymbol{V}$ *obtained by the model* f *must be found. This model, denoted by* g*, learns to predict the next state of the system given the previous one:*

$$\hat{\boldsymbol{V}}_{t+k} = g(\boldsymbol{V}_t; \omega) \tag{98}$$

The model's parameters are found by empirical risk minimization of the reconstruction errors of the predicted trajectories:

$$\omega = \arg\min_{\omega} \mathbb{E}\left[\| \boldsymbol{U}_t - \hat{\boldsymbol{U}}_t\|\right] \tag{99}$$

Where $\boldsymbol{U}_t = \{U_0, U_1, ..., U_N\}$ *is the ordered collection of samples in time and* $\hat{\boldsymbol{U}} = f^{-1}(\hat{\boldsymbol{V}}_t)$ *is the projection of the trajectory in the latent space to the full-order space.*

4.3.2 Background: Model Reduction for Fluid Flow Problems

Reduced order modeling is essential in many numerical workflows. One popular approach for reducing the computational complexity of problems governed by parametric PDEs is reduced basis (RB) methods. Reduced basis methods consist of finding a subspace where the solutions of the PDEs exist but its dimensions are considerably smaller than the full-order model. Although they may require an intensive offline phase where the basis is learned from data, they need less computational resources during the online stage, where the model is used in substitution of the numerical method that approximates the PDE [179].

One of the most common ways to obtain a reduced basis of a fluid flow problem is by decomposing the flow field into a set of modes. The quantities of interest, such as the velocity field, are discretized and assembled into a vector $\mathbf{x}(t) \in \mathrm{R}^n$, called the state vector. Let us consider a set of a linearly independent set of modes $\{\mathbf{v}_1, \ldots, \mathbf{v}_r\}$, where $\mathbf{v}_j \in \mathrm{R}^n$. These modes span a r-dimensional subspace S so $\mathbf{x}(t)$ can be expressed as a linear combination of these modes:

$$\mathbf{x}(t) = \sum_{j=1}^{r} a_j(t)\mathbf{v}_j \tag{100}$$

Or written in matrix form:

$$\mathbf{x}(t) = \mathbf{V}\mathbf{a}(t) \tag{101}$$

Where $\mathbf{V}$ is a rectangular matrix of dimensions $n \times r$, with $n >> r$. The state also evolves in time with dynamics given by:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)) \tag{102}$$

Even though $\mathbf{x}(t)$ lies in S, the equation's right-hand side does not necessarily lie in it. We need to define the dynamics in S. This is done by projecting the right-hand side into the sub-space. The vector in S that is the closest to $\mathbf{x}(t)$ is give by the orthogonal projection $\mathbf{P}\mathbf{x}(t)$, with:

$$\mathbf{P} = \mathbf{V}\left(\mathbf{V}^T\mathbf{V}\right)^{-1}\mathbf{V}^T \tag{103}$$

If the modes are orthonormal, this becomes $\mathbf{P} = \mathbf{V}\mathbf{V}^T$. The dynamics can also be projected with a non-orthogonal projection onto the subspace S, being $\mathbf{W}$, a vector subspace of the same dimensions of S. This projection is defined as:

$$\hat{\mathbf{p}} = \mathbf{V}\left(\mathbf{W}^T\mathbf{V}\right)^{-1}\mathbf{W}^T \tag{104}$$

If the sets $\{\mathbf{v}_j\}$ and $\{\mathbf{w}_j\}$ form a bi-orthogonal set (the inner product $\langle \mathbf{v}_j, \mathbf{w}_k \rangle = \delta_{jk}$)), then the projection becomes $\mathbf{P} = \mathbf{V}\mathbf{W}^T$. Inserting the equation $\mathbf{x}(t) = \mathbf{V}\mathbf{a}(t)$ into the projected dynamics $\dot{\mathbf{x}} = \hat{\mathbf{P}}\mathbf{f}(\mathbf{x})$, we obtain:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{a}(t) = \mathbf{W}^T\mathbf{f}(\mathbf{V}\mathbf{a}(t)) \tag{105}$$

The equation 105 is a reduced-order model. It consists of r equations that describe the evolution of $a(t)$, from which the state can be reconstructed $\mathbf{x}(\mathbf{t}) = \mathbf{V}\mathbf{a}(\mathbf{t})$. When $r << n$, this can potentially reduce computational cost. If $\mathbf{W} = \mathbf{V}$, the projection is orthogonal, and this procedure is called Galerkin projection. If $\mathbf{W}$ is different from $\mathbf{V}$, this is called Petrov-Galerkin projection [209].

One of the key challenges in model reduction is finding the subspaces to project the dynamics. Most techniques for model reduction involve projecting the governing equations onto a linear subspace within the original state space. These subspaces are typically determined using methods like balanced truncation, rational interpolation, the reduced-basis method, and proper orthogonal decomposition (POD), including other variants of POD. However, a significant drawback of constraining the state to evolve within a linear subspace is that it inherently limits the accuracy of the resulting reduced-order model. Linear-subspace ROMs can only achieve high accuracy in low-dimensional models when the problem exhibits a fast-decaying Kolmogorov n-width, such as in diffusion-dominated problems. Unfortunately, many issues of interest have a slowly decaying Kolmogorov n-width, particularly advection-dominated problems [129].

The Kolmogorov n-width (KnW) is a classical concept of approximation theory as it describes the error arising from a projection onto the best possible space of a given dimension $N \in \mathbb{N}$. This error is measured for a class $\mathcal{M}$ of objects because the worst error over $\mathcal{M}$ is considered. Let $\mathcal{M}$ be a subset $\mathcal{M} \subset H$, where H is some Banach space or Hilbert space with norm $\|\cdot\|_H$. Then, the Kolmogorov n-width is defined as:

$$d_N(\mathcal{M}) := \inf_{\substack{V_N \subset H; \\ \dim V_N = N}} \sup_{u \in \mathcal{M}} \inf_{v_N \in V_N} \|u - v_N\|_H \tag{106}$$

Where V_N are linear subspaces. For certain linear problems that arise from parametric PDEs, the Kolmogorov n-width decays exponentially fast (for demonstrations, see [179]):

$$d_N(\mathcal{M}) \leq Ce^{-\beta N} \tag{107}$$

With some constants $C < \infty$ and $\beta > 0$. Problems with fast decay allow selecting a N that allows fewer degrees of freedom than the full-order model while achieving low approximation errors [77]. Advection-dominated problems that exhibit strong non-linearities have a slow decaying KnW, as stated earlier, which includes most of the solutions of the Navier-Stokes equation. Reduced order models for fluid mechanics will work better if the dynamics are projected into non-linear subspaces instead. The literature employs extensions of linear methods to find such non-linear subspaces but requires careful tuning and

doesn't generalize well [209].

On the contrary, the non-linear representations learned by Deep Learning can overcome the KnW limitation [129]. For this reason, an Autoencoder model is implemented in the following subsection to learn a non-linear latent space of reduced dimensions where the dynamics can be learned by another Neural Network, in this case, a Convolutional LSTM.

4.3.3 Homogenous Isotropic Turbulence Dataset

The dataset will be used for the learning problem, is built from a DNS of forced homogenous isotropic turbulence from the John Hopkins Turbulent dataset [186]. The simulations are performed on a 1024^3 periodic grid using a pseudo-spectral parallel code. The simulation solves the incompressible Navier-Stokes equation on the cube:

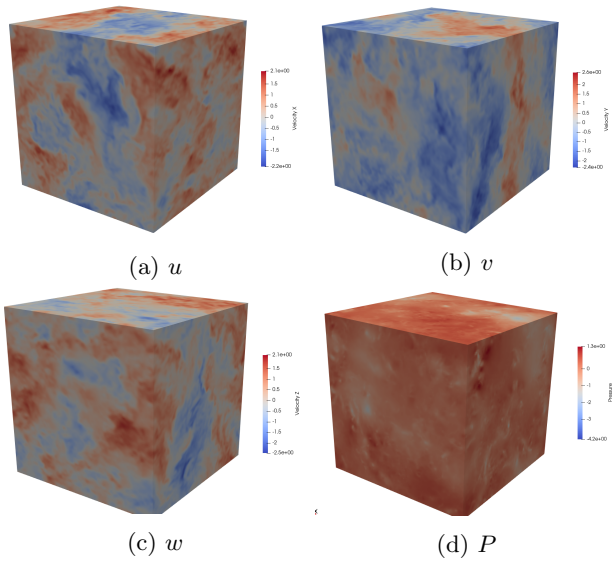
$$\begin{aligned} &\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u}, \\ &\nabla \cdot \mathbf{u} = 0 \end{aligned} \tag{108}$$

Where $\mathbf{u}$ is the velocity vector, p is the pressure, and ν is the kinematic viscosity. The time integration of the viscous term is done analytically using an integrating factor, and the other terms are integrated using a second-order Adams-Bashforth scheme. The nonlinear term is written in vorticity form [32]. The simulation is de-aliased using phase-shift and $2\sqrt{2}/3$ truncation [185]. The Energy is injected in the simulation by keeping constant the total energy in modes so that their wave-number magnitude is less or equal to 2. The divergence-free condition is satisfied due to the spectral representation of the derivatives. The divergence-free condition is satisfied when the velocities are computed from the vorticity. The data in the database is stored after the simulation reaches a statistically stationary state. It contains 5028 data frames, including the three components of the velocity vector and the pressure field. The duration of the stored data is about five large-eddy turnover times.

This dataset is too large to be used for Neural Network training. Because of its high dimensionality (3D + time), it is difficult to fit one time-step of this data on a single GPU, as the memory required for computing the network's gradients scales with the input's dimensions. For this reason, the dataset created is a subsample of this dataset. The resolution is reduced from 1024^3 to 128^3 by filtering the original velocity and pressure fields using a box filter of width $\Delta = 9$. $3,000$ snapshots like this are collected from the database. The dataset information is summarized in table 4. Samples from this dataset are shown in Fig. 52.

HIT dataset sheet	
N_t (number of time-steps)	3000
Resolution	128^3
L (dimensions)	2π
δt (time-step between stored fields)	0.002 s
Re_λ (Taylor-scale Reynolds	418
T_L (Large eddy turnover time)	1.99 s

Table 4: Dataset for 3D Homogenous Isotropic Turbulence.



(a) u (b) v

(c) w (d) P

Figure 52: Velocity components and pressure of a sample of the filtered HIT dataset.

4.3.4 Autoencoder for Dimensionality Reduction

The first stage in building the ML-based ROM is to find a reduced dimensionality representation that permits the learning of the dynamics in it. Autoencoders (AEs) are Machine Learning models used extensively for lossy data compression and dimensionality reduction [166][89]. There is a significant amount of work that has used Convolutional Autoencoders for the dimensionality reduction of fluid flow problems [173][67][58][270]. Convolutional Autoencoders can learn representations that yield lower reconstruction errors and higher reduction rates that approaches based on Principal Components Analysis. However, most of this work has focused on fluid problems in 2D at a low resolution, which permits having large amounts of data and using deep networks. The problem that will be tackled in this work is predicting turbulence in 3D; this limits the amount of data available and the scalability of the models due to

memory constraints. A naive solution to this problem is replacing the standard 2D convolutions in CNNs with 3D convolutions. Still, this has a downside: the number of parameters of the convolutional layers grows cubically. For example, a 2D CNN layer with 64 and 3×3 kernel has 576 learnable weights, and the 3D counterpart will have 1728. This will limit the depth of the networks that can be used, thus reducing the learning capacity and increasing the computational cost.

One way to increase the performance of 3D CNNs without increasing the computational cost is by making the convolution operation more efficient. Depthwise separable convolutions are designed with this purpose in mind. It has been observed that trained CNNs contain many redundant weights. The idea behind separable convolutions is to reduce this by factorizing the convolution operation into separable convolution followed by pointwise multiplication [228]. Depthwise separable convolutions have been used in modern CNN architectures such as Xception [44], and MobileNet [92], which is a CNN designed to fit on module devices with limited capabilities. As stated in the MobileNet paper, Depthwise Separable Convolutions can reduce the computational cost of convolutions by 8-9 times while keeping almost the same accuracy. In 2D, a standard convolutional layer is parameterized by a kernel $\mathbf{K}$ of size $D_K \times D_K \times M \times N$ where D_K is the spatial dimension of the kernel, and M is the number of input channels, and N is the number of output channels. The output feature map of a convolution assuming stride one is calculated as:

$$\mathbf{G}_{k,l,n} = \sum_{i,j,m} \mathbf{K}_{i,j,m,n} \cdot \mathbf{F}_{k+i-1,l+j-1,m} \tag{109}$$

Depthwise separable convolutions use one filter per input channel, and then the features are combined using a 1 convolution to generate new features. This is computed as:

$$\hat{\mathbf{G}}_{k,l,m} = \sum_{i,j} \hat{\mathbf{K}}_{i,j,m} \cdot \mathbf{F}_{k+i-1,l+j-1,m} \tag{110}$$

The cost of the standard convolutional layer depends multiplicatively on the parameters of the convolution operation:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{111}$$

Where the cost of the Depthwise separable convolution is the sum of the $1X1$ convolution and the depthwise convolution:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F \tag{112}$$

From these expressions, the reduction in computation can be expressed as:

$$\begin{aligned} &\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} \\ =&\frac{1}{N} + \frac{1}{D_K^2} \end{aligned} \tag{113}$$

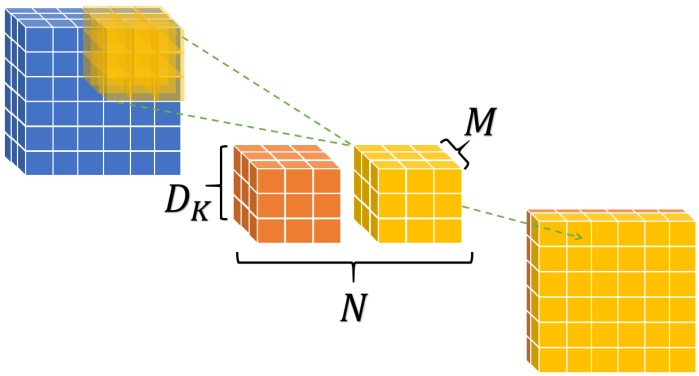


Figure 53: Standard convolutional layer with a 3×3 kernel, 3 input channels and 2 output channels.

So, for a Depthwise separable convolution with kernel 3×3, the computational cost is 9 times less than the standard convolution. To illustrate this further, refer to Fig. 54 for a diagram of an example of a 2D standard convolutional layer and to Fig. 53 for its Depthwise separable equivalent. For this case, it can be observed that the standard convolutional layer possesses 54 learnable weights, whereas the Depthwise separable convolution has 33, 61% of the number of parameters the regular convolution has. The same concept of depthwise separable convolutions is applied for 3D; this has been done before for reconstruction tasks [275]. The 3D Depthwise convolution is represented graphically in Fig. 55.

The next step is designing the decoder and encoder architectures that will be used for the problem at hand. As a reminder, a Convolutional Autoencoder is a neural network model that integrates two subnetworks: an encoder $\mathcal{E}$ that maps the input $\mathbf{U}$ to the latent space $\mathbf{z}$, and a decoder network $\mathcal{D}$ that maps the latent space to the original dimension. The output of the AE is the reconstruction:

$$\hat{\mathbf{U}} = (\mathcal{D} \circ \mathcal{E})(\mathbf{U}) \tag{114}$$

The encoder and decoder networks are based on the ConvNeXt architecture [151]. ConvNeXt is one of the most recent CNN models; it surpasses other CNNs and Vision Transformers on computer vision tasks while requiring less data and computational resources than their attention-based counterparts. The network uses an initial "patchify" layer that is a convolution with kernel size $4 \times 4 \times 4$. After, it is followed by a series of ConvNext residual blocks that are composed of a Depthwise convolution with a kernel size $7 \times 7 \times 7$; this kernel size is considered large compared to the regular kernels of width 3 using in other CNNs, but it has been proved to increase performance without adding more computational strain. After the DW convolution, there are two dense layers; the first increases the number of features by a factor of 4, and the second reduces

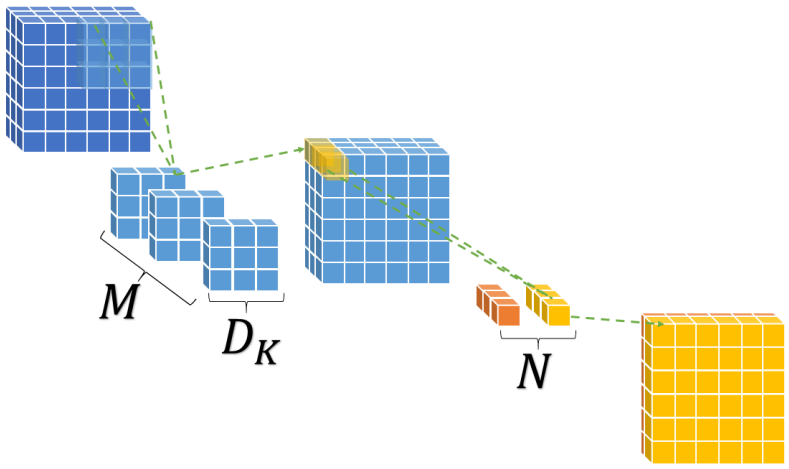


Figure 54: Depthwise separable convolutional layer with a 3×3 kernel, 3 input channels and 2 output channels.

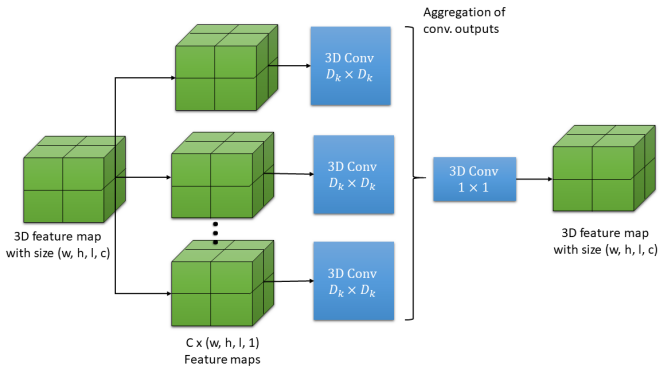


Figure 55: 3D Depthwise Separable Convolution. The input feature map is divided into c feature maps corresponding to each input feature; a convolution is applied to each one with its respective kernel. The output of each convolution is concatenated and then passed to a pointwise convolution.

it back to the output feature size. The structure of the ConvNeXt block can be seen in Fig. 56.

The activation function used is the GELU activation, a smoother version of the ReLU activation used in transformer architectures like BERT [50] and Vision Transformers [51]. GELU showed improvement in the accuracy of Imagenet in the ConvNeXt paper. Regularly, Batch Normalization (BN) is used in CNNs; in

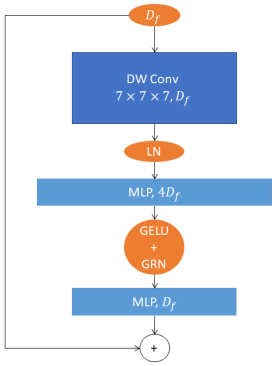


Figure 56: Diagram of a ConvNeXt block.

this case, Layer Normalization substitutes it. LN does not affect performance, is simpler to implement, has no trainable parameters, and does not introduce the problems sometimes BN can [269]. After the GELU activation in the ConvNeXt layer, there is a Global Response Normalization (GRN) layer. The objective of GRN is to prevent feature collapse, which means that there are learned feature maps that are either dead or saturated, not contributing to the global output [263]. Given a spatial feature map X_i the response normalization function is:

$$\mathcal{N}\left(\|X_i\|\right) := \|X_i\| \in \mathcal{R} \rightarrow \frac{\|X_i\|}{\sum_{j=1,\ldots,C} \|X_j\|} \in \mathcal{R} \tag{115}$$

Where $\|X_i\|$ is the $L-2$ norm of the $i-$th channel. This response normalization can be interpreted as the relative importance of a feature with respect to all other channels. The original input response is calibrated using the computed feature normalization score:

$$X_i = X_i * \mathcal{N}\left(\|X_i\|\right) \in \mathcal{R}^{W \times H \times L} \tag{116}$$

To facilitate the optimization, two learnable parameters, γ and β, are added and zero-initialized. A residual connection is also used between the input and output of the GRN layer. This sets the GRN layer to first be an identity function at the beginning of training and then adapt as it is necessary:

$$X_i = \gamma * X_i * \mathcal{N}\left(\|X_i\|\right) + \beta + X_i \tag{117}$$

The overall structure of the Encoder and Decoder networks is a 3D ConvNeXt model with three stages. Each stage has $1, 2, 1$ ConvNeXt blocks, respectively. Between each stage, there is a downsampling or upsampling block, depending on whether the network is the encoder or the decoder. For spatial downsampling, there is a $2 \times 2 \times 2$ Conv layer with a stride of 2, and for upsampling, a transposed convolution is used with the same parameters, LN is used

before each down/up-sampling layer. The features of the encoder at each stage is $(16, 32, 64)$ and for the decoder $(64, 32, 16)$. The output layer is a $1 \times 1 \times 1$ convolution. The AE diagram is seen in Fig 57.

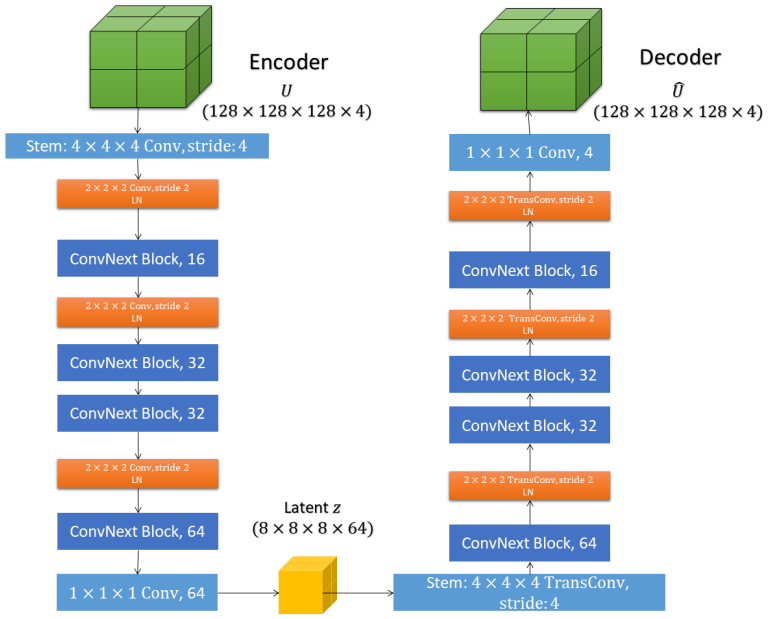


Figure 57: Diagram of Autoencoder for 3D HIT fields.

4.3.5 Latent Space reconstruction with Autoencoder

In this section, the results obtained from the trained Autoencoder are analyzed. The model is trained on $2,500$ snapshots of 3D filtered HIT fields containing 4 variables: three velocity components and pressure. The errors metrics to asses performance are:

- **Relative Root-Mean-Squared error**: to quantify the percentage of reconstruction fidelity or loss of information. this is computed as:

$$RRMSE = \frac{\sum_{i=1}^{N} \sqrt{\|U_i - \hat{U}_i\|^2}}{\sum_{i=1}^{N} \sqrt{\|U_i\|^2}} \tag{118}$$

- **Turbulent Kinetic Energy error:** This error measures the difference between the turbulent kinetic energy spectrum of the ground truth turbulence data and the reconstruction by the AE. The turbulent kinetic energy

spectrum indicates the amount of energy the flow carries at a certain wave number. The energy spectrum in homogenous isotropic turbulence is an invariant statistic and does not change in time, rotations, shifts, or reflections of the flow field [191]. Thus, it is an appropriate measure to asses if the auto-encoder preserves the structures present in the turbulent flow. The kinetic energy spectrum is the integral in wave space of the velocity spectrum function, which is the Fourier transform of the velocity autocorrelation in space:

$$\phi_{i,j}(\mathbf{k}) = \frac{1}{(2\pi)^3} \iiint R_{i,j}(\mathbf{r}) e^{-\mathbf{i}\mathbf{k}.\mathbf{r}} \mathrm{d}\mathbf{r}, \quad R_{i,j}(\mathbf{r}) = \frac{1}{V} \iiint u_j(\mathbf{x}) u_i(\mathbf{x}+\mathbf{r}) \mathrm{d}\mathbf{x}. \tag{119}$$

Directional information is removed in the Fourier space by integrating over all wavenumbers $\mathbf{k}$ of magnitude $k = \|\mathbf{k}\| = \sqrt{k_x^2 + k_y^2 + k_z^2}$. This is expressed as the surface integral over the sphere $\mathcal{S}(k)$ in wave number space centered at the origin with radius k. The energy spectrum is calculated then as:

$$E(\kappa) = \oint \frac{1}{2} \Phi_{ii}(\kappa) \mathrm{d}\mathcal{S}(\kappa) \tag{120}$$

The TKE spectrum error is calculated as the RRMSE between the ground truth data and the reconstruction by the AE.

- **Divergence of the velocity**: The divergence-free condition ($\nabla \cdot U$) is a property that should be satisfied for incompressible fluid flows. This quantity is computed to asses still if the field reconstructed by the AE still satisfies this property.

The errors are reported in table 5. What can be observed is that the model can recover the most important flow structures with an error of 0.1221 and a spectrum error of 0.0620. The spectrum error is lower than the RRMSE, which means that the model fits the higher-energy-containing scales correctly, and the errors reside on smaller scale fluctuations, which contribute less to the overall behavior of the fluid. This is qualitatively observed in figure 59. A sample of the Autoencoder reconstruction can be observed in Fig. 58. These results correspond to the Energy spectrum where the larger structures are conserved in the reconstruction.

Results of AE on HIT reconstruction		
RRMSE	TKE error	Divergence
0.1221	0.0620	0.0021

Table 5: Results of AE for 3D Homogenous Isotropic Turbulence.

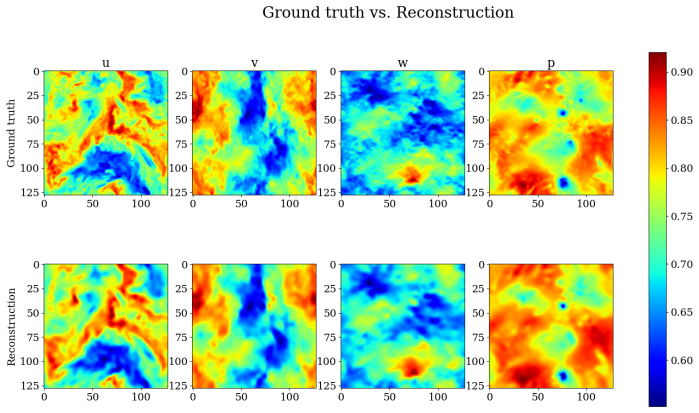


Figure 58: Slice of a turbulent field from the HIT dataset. Ground truth data and reconstruction by Autoencoder.

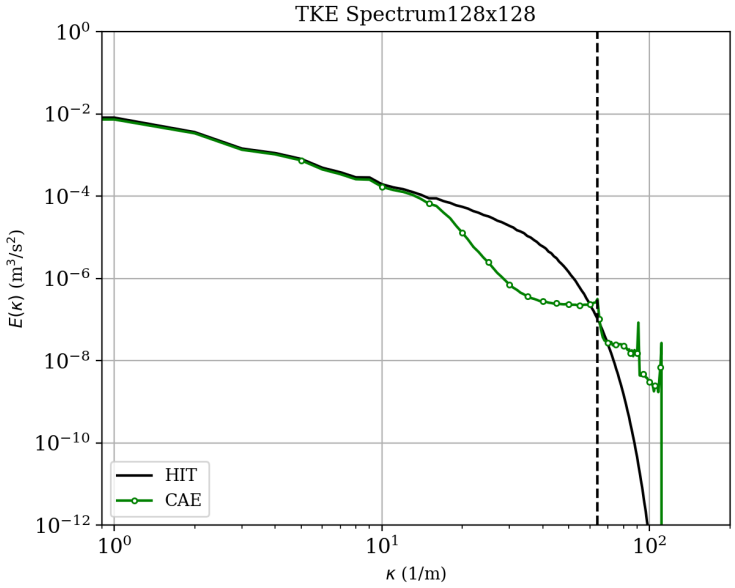


Figure 59: Turbulent Kinetic Energy Spectrum of the HIT flow field. Comparison of the ground truth data and the Autoencoder.

As a final note, the purpose of the Autoencoder here is that of dimensionality reduction. The compression ratio of the autoencoder can be calculated as the relation between the input tensor field dimensions and the dimensions of the latent variable. In this case, it will be:

$$\mathbf{CR} = \frac{128^3 \times 4}{8^3 \times 64} = 4096 \tag{121}$$

Compared to related work, Mohan et al. [163] used a similar AE network for HIT at a resolution of 128^3. They achieved similar performance but a compression rate of 125. Other works have implemented AEs for finding a reduced-order representation of turbulent flow [67][58]. These works tackled 2D slices of turbulent flows, so they did not tackle the memory constraint faced when learning in 3D. For example, the $\beta - VAE$ used in [58] reported a reconstruction accuracy based on the relative MSE around 88%, this error is computed as:

$$E_k = \left(1 - \left\langle \frac{\sum_{i=1}^{n}(u - \tilde{u})^2}{\sum_{i=1}^{n} u^2} \right\rangle\right) \times 100, \tag{122}$$

Where $< . >$ is the ensemble average, and u is the velocity field. The 3D CAE achieves $E_k = 99.5\%$ accuracy using the previous metric definition. It is worth noting that the objective of $\beta - VAE$ is different than the standard AE formulation capabilities, which cannot generate new samples and are not regularized to learn disentangled representations.

4.3.6 Convolutional LSTM for HIT

After learning the lower dimensional representation, the following step is to learn how these latent variables evolve over time. This problem is formulated as a sequence modeling problem, where the goal is that the model can predict the next state of the latent variable given only the previous state of it. For this task, an LSTM model will be used. LSTMs have been used before for performing spatiotemporal prediction of fluid flows [163][277][65]. Even though other models for sequence models exist that have better performance, like transformers and Neural ODEs, recurrent neural networks, in general, are easier to train since they require less data. The implementation is easy due to pre-built modules in the most used deep learning frameworks and have simpler training algorithms. This might explain the reason why, for the fluid mechanics community, it has been an entry model in the development of data-driven surrogate models of fluid flows, and in this study, the same steps will be followed.

The model used will be a Convolutional LSTM (ConvLSTM). The configuration of this model is similar to an LSTM, with the difference that Convolutional layers replace the linear layers in the cell. The ConvLSTM has been used for action recognition, video prediction, and precipitation forecasting [225]. It is appropriate for spatiotemporal sequence problems because the convolutional kernels help preserve spatial correlations present in the problem while also reducing memory costs associated with fully connected layers present in the original LSTM formulation [91]. Being $*$ the convolution operation, the operations inside a ConvLSTM cell are:

$$
\begin{aligned}
i_t &= \sigma\left(W_{xi} * \mathcal{X}_t + W_{hi} * \mathcal{H}_{t-1} + W_{ci} \circ \mathcal{C}_{t-1} + b_i\right) \\
f_t &= \sigma\left(W_{xf} * \mathcal{X}_t + W_{hf} * \mathcal{H}_{t-1} + W_{cf} \circ \mathcal{C}_{t-1} + b_f\right) \\
\mathcal{C}_t &= f_t \circ \mathcal{C}_{t-1} + i_t \circ \tanh\left(W_{xc} * \mathcal{X}_t + W_{hc} * \mathcal{H}_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo} * \mathcal{X}_t + W_{ho} * \mathcal{H}_{t-1} + W_{co} \circ \mathcal{C}_t + b_o\right) \\
\mathcal{H}_t &= o_t \circ \tanh\left(\mathcal{C}_t\right)
\end{aligned}
\tag{123}
$$

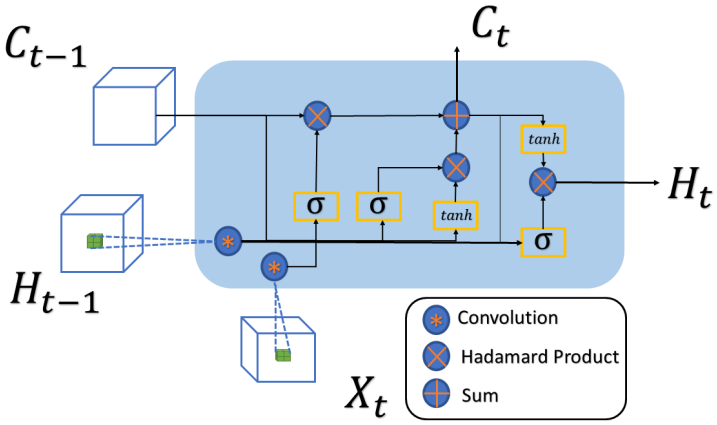
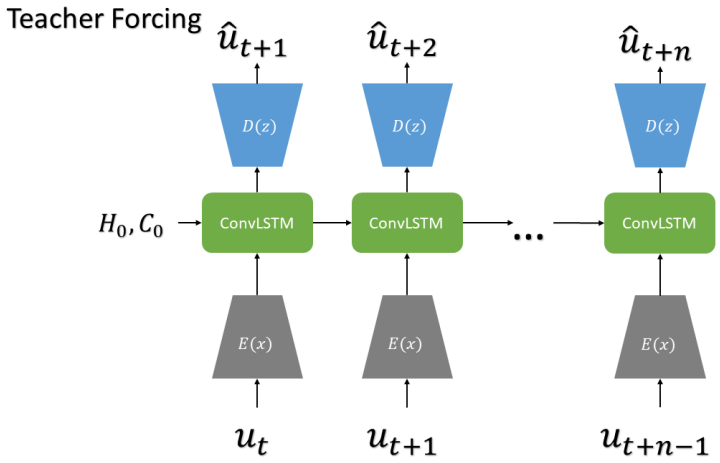
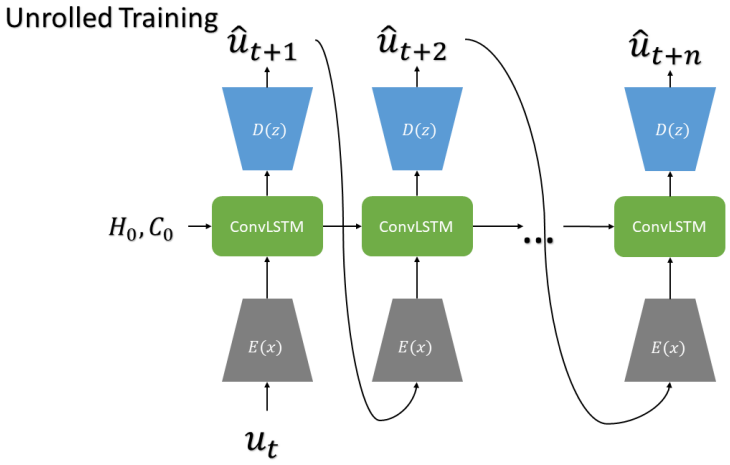


Figure 60: Diagram of a ConvLSTM cell.

This equations are illustrated graphically in Fig. 60. The ConvLSTM model that will be used consists of two stacked ConvLSTM cells with 128 hidden features. The network will use 3D convolutions with 3^3 kernels in this case. The ConvLSTM will operate on the latent representation learned previously by the Autoencoder, reducing computational and memory costs. The Autoencoder is coupled to the ConvLSTM by separating the encoder and decoder network. The encoder is frozen during training using the pre-trained parameters, acting as a pattern recognition module. The ConvLSTM predicts the next step in the sequence for the latent variables, and the decoder projects the prediction to the original feature space. The decoder is fine-tuned during the ConvLSTM training but at a 1×10^{-6} learning rate. This aims to make the decoder robust to possible errors the ConvLSTM might produce in its predictions. The ConvLSTM is trained with the Adam optimizer with a 1×10^{-4} learning rate and cosine learning rate decay with warmup [152]. The model is trained to minimize the mean-squared error between the predicted time step and the ground truth. Teacher forcing is used, meaning that the model is fed the ground-truth input to make the next time step prediction, and this is discarded to predict the next time step. An additional second term allows the model to unroll the predictions for several time steps. For inference, the model is let to generate the next time steps in an autoregressive way. This is shown in Fig. 61.



(a)



(b)

Figure 61: Training and Inference of ConvLSTM: a) Teacher forced training b) Unrolled predictions

The loss function used for training is the combination of teacher-forced predictions and unrolled training:

$$L(\theta) = \|\mathbf{u} - \hat{\mathbf{u}}_{tf}\| + \|\mathbf{u} - \hat{\mathbf{u}}_{ut}\| \tag{124}$$

4.3.7 Results of ConvLSTM on HIT

The dataset is split into shorter sequences for training the ConvLSTM. ConvLSTM was trained on sequences of the flow field containing 20 time steps. The time time-step Δt used for training corresponds to $100\delta t$ of the DNS, which is $0.002s$. The total duration of the sequence is $4s$, which corresponds approximately to 2 Large-Eddy turnover times. After training, the model achieves an RRMSE averaged over time of **0.1305**. Samples of the predicted velocity components and pressure can be observed in Figs. 62 - 65. The figure shows that the predictions contain similar structures, but they diverge from the ground truth as they get unrolled.

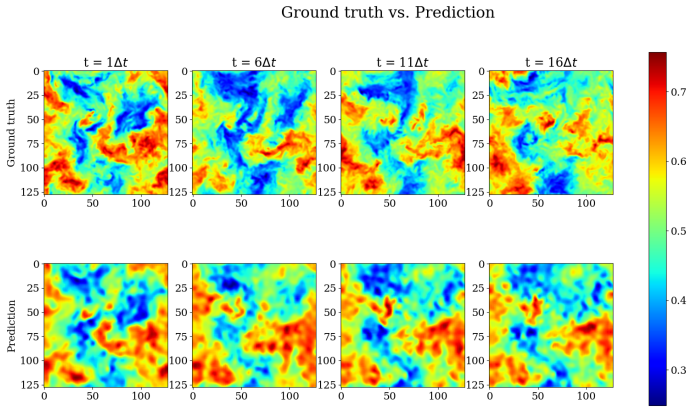


Figure 62: Sample of a prediction made by the ConvLSTM: u

Similarly, the TKE spectrum is computed and averaged over time, as shown in Fig. 66a. From the spectrum, it can be inferred that the ConvLSTM predicts a flow containing the more energetic modes while approximating the smaller scales. The RRMSE of the TKE spectrum is **0.1358**; it is higher than the TKE error obtained by the Autoencoder alone. This is because the predictions deteriorate proportional to the number of time steps. The velocity autocorrelation function can characterize the flow's temporal behavior, as shown in Fig. 66b. The Autocorrelation deviates from the ground truth, and its RRMSE is 0.1974.

The turbulent flow predicted by the ConvLSTM is characterized statistically by calculating invariant statistics averaged over time. The following properties will be used:

- Total Kinetic energy:

$$E_{tot} = \frac{1}{2} \langle u_i u_i \rangle \tag{125}$$

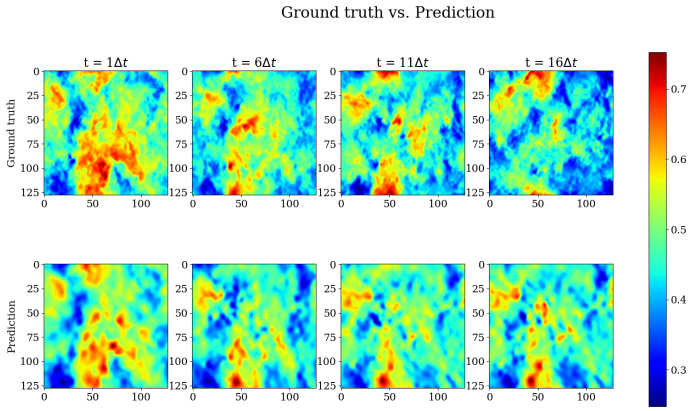


Figure 63: Sample of a prediction made by the ConvLSTM: v

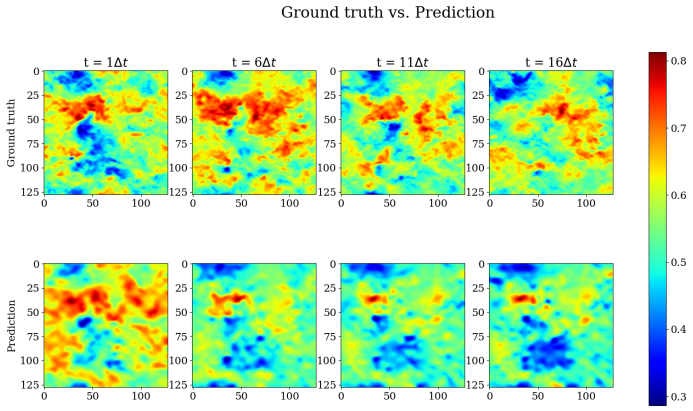


Figure 64: Sample of a prediction made by the ConvLSTM: w

- Rms velocity:

$$u' = \sqrt{(2/3)E_{tot}} \tag{126}$$

- Integral scale:

$$L = \frac{\pi}{2u'^2} \int \frac{E(k)}{k} dk \tag{127}$$

- Large eddy turnover time:

$$T_L = L/u' \tag{128}$$

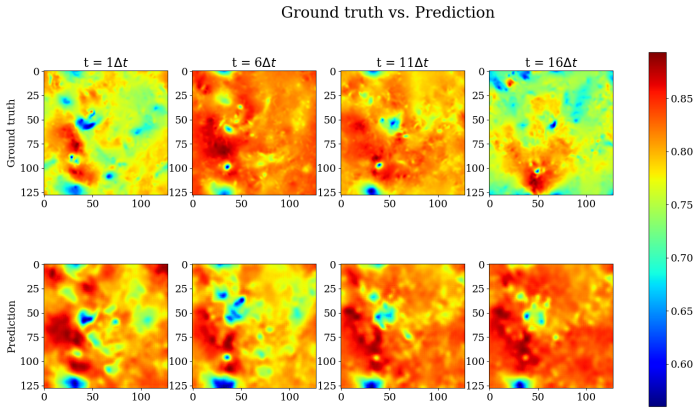
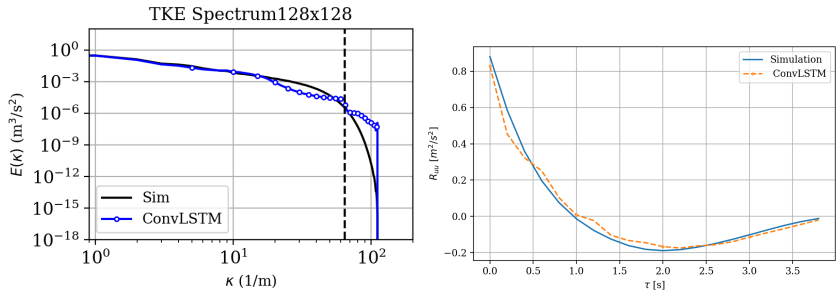


Figure 65: Sample of a prediction made by the ConvLSTM: P



(a) (b)

Figure 66: a) Turbulent Kinetic Energy Spectrum for the ConvLSTM predictions. and b) Autocorrelation of u for the ConvLSTM predictions.

Statistical Characteristics of turbulence					
Property	DNS	Data	ConvLSTM	ϵ_{DNS}	ϵ_{data}
E_{tot}	0.695	0.662	0.636	0.085	0.039
u'	0.681	0.664	0.651	0.044	0.020
L	1.376	1.457	1.516	0.102	0.041
T_L	2.02	2.19	2.33	0.153	0.062

Table 6: Statistical characteristic of the turbulent flow.

The predicted turbulent flow by the ConvLSTM preserves the statistical properties if compared to the statistics of the dataset and those of the DNS; the comparison of values can be seen in table 6. From the different statistics, the one with the highest error is the large eddy turnover time; this may indicate

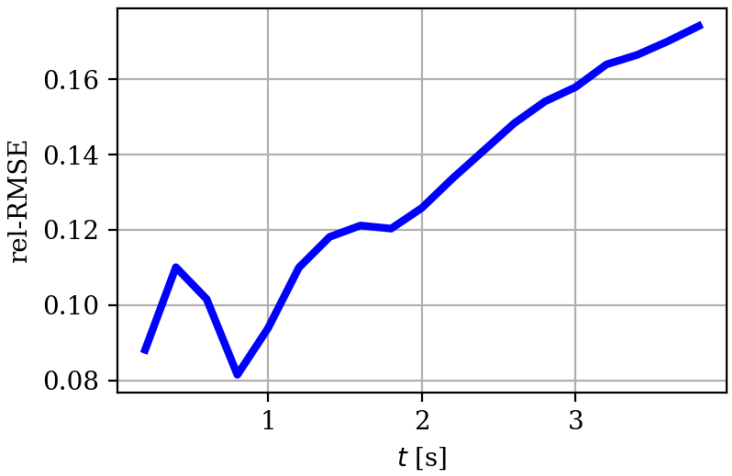


Figure 67: RRMSE with respect to time between Ground truth HIT field and the ConvLSTM prediction.

that the biggest deviation is in the temporal behavior. What is observed in the experiments is that the error keeps increasing as the ConvLSTM advances in time. If one tries to extrapolate for longer sequences bigger than the one used in training, the velocity will stop changing or get stuck in an oscillating state, or the values will blow up. The evolution of the error for $20\Delta t$ is shown in Fig. 67. This error may be a symptom of overfitting, where the model gets stuck predicting closer time steps or an average value that gives a low error for the overall predictions. This problem could be mitigated by adding more data, but it increases the cost of training. Or a simpler model could be used at the risk of losing accuracy in the results.

4.4 Conclusions on AE-ConvLSTM for reduced order modeling of turbulent flows

Using Deep Learning, we have implemented a data-driven, non-intrusive, reduced-order model for a turbulent flow. The framework utilizes a convolutional Autoencoder for dimensionality reduction and a Convolutional LSTM to learn the spatiotemporal evolution of the flow state in the latent space. The target task was to predict the evolution of Homogenous Isotropic Turbulence. The data used comes from a database built from high-fidelity DNS simulation. This task presented important technical challenges; one is to be able to learn from data with very high dimensions and complexity. To solve this issue, we relied on two approaches: the first approach consists of data selection; we filtered the DNS simulation from a 1024^3 into a 128^3 grid and split the simulation contained in the database in sequences of 10 time steps in order to be more manageable

during training. The second approach is to reduce the memory requirements of the Deep Learning architectures by using Depthwise Convolutions for the 3D tensors. The AE model implemented in this study exhibited good performance in agreement with the physical properties of the flow. The model also performs similarly to state-of-the-art dimensionality reduction methods regarding reconstruction fidelity. Even though this AE was designed with the purpose of learning a latent representation in mind, it could be used for other tasks like data compression to save storage space and facilitate file transfer of big turbulence databases. This would require improvements in the error by employing other super-resolution models in order to recover as much fidelity as possible, which is needed for the study of turbulence. In the second stage of training, a ConvLSTM was implemented to learn the flow dynamics in the latent space. The ConvLSTM model was selected because it is more convenient in this case because it learns the spatial correlations at the same time as it learns the temporal dependence of the steps in the sequence. The model was trained to make predictions with a time step equivalent to 100 simulation δt, roughly equal to 1/10th of the large eddy turnover time. If a smaller time step is used, the model may not be able to learn the flow dynamics in short sequences since the time steps contained in the sequence will be highly correlated. The ConvLSTM could make predictions with a distribution of spatial scales in good agreement with that of the ground truth data. The TKE spectrum could verify this. The predictions also had statistical properties close to the dataset and the DNS. However, the temporal behavior diverges from what is expected for this kind of flow, as seen in the autocorrelation function, large eddy turnover time, and the evolution of the RRMSE. The model is probably overfitting in the short-term predictions, unable to generalize to long sequences. This may indicate we need more data to train such a model, which is costly, or that it is not the most appropriate model fit for this task. Another limitation of this approach is the computational cost, even if we have reduced it using the depthwise separable convolutions in the autoencoder. Going beyond the resolution used in the study or training on longer sequences requires much more memory that may not be available on current GPUs. The training of the ConvLSTM model requires around 20h on 4 NVIDIA V100 GPUs, and tuning the hyperparameters requires extensive training iterations. Another major disadvantage we could observe is that the model does not extrapolate well for predicting many time steps. Since implementing this model, other deep learning approaches, especially transformers for sequence problems and Neural Operators for PDE-related tasks, have gained popularity and state-of-the-art results in their respective category. In the next chapter, Neural Operators will be explored to tackle the problem of the spatiotemporal forecasting of turbulent flows. As a physics-inspired method designed to tackle problems related to PDEs, the hypothesis is that it has inductive biases that can leverage more properties from the data, increasing the generalization capabilities. The main interest is to evaluate if these models can achieve numerically stable turbulence predictions over a long period. It is worth noting that is possible to go the other way and explore transformer-inspired methods for the prediction of turbulent flows as has been done for weather prediction

[184][176], with great promise for generalization and accuracy. However, this is left as a perspective since it will require more data and computational resources.

5 Surrogate Models of Turbulence Simulations with Neural Operators

5.1 Introduction

Computer simulations are essential for studying turbulence and designing engineering systems interacting with flows exhibiting this behavior. Simulating turbulent flows to the highest degree of accuracy possible requires computational meshes that capture up to the smallest energy-containing eddy. The amount of active scales on a turbulent flow is proportional to its Reynolds number (Re), so as the Re increases, the computational mesh required becomes finer. This makes, for instance, certain configurations, such as atmospheric flows, where the Re can reach orders of (10^7) computationally prohibitive to calculate. For industrial applications, it is more common to see filtered Navier-Stokes equations such as LES or RANS, which are cheaper to compute and have a mathematical model to compensate for unknown or unfiltered fluctuations. Nevertheless, these simulations remain costly for tasks requiring many functional evaluations of the numerical solution, such as control, design optimization, and uncertainty quantification.

To perform the aforementioned tasks, it is preferred to use surrogate models that can predict quantities of interests of these flows with a required degree of accuracy but in a fraction of the time needed for a CFD method to calculate it. Recently, the use of Deep Learning for developing surrogate models has attracted scientists and engineers because of the capabilities of Deep Learning to capture complex non-linear relationships that are present in turbulence, overcoming limitations that may be present in techniques that rely on linear projection.

This chapter formally describes learning surrogate models of turbulent flow simulations from data, a spatiotemporal regression problem. Afterward, we will discuss the common challenges of learning a predictive reduced-order turbulence model. From the literature, we identify mainly two issues: the conservation of physical laws and numerical stability. We have studied so far GANs and Autoencoders with LSTMs for the prediction of turbulent flows. In the literature, many examples can be found that attempt to learn fluid flow simulations. Still, most of these methods work for a narrow set of cases, and there is not much success in generalization outside the training data distribution. Recently, Physics-Inspired Machine Learning has been gaining interest due to the fact that including physical concepts in the design and the learning process can ameliorate the performance and generalization capabilities of Neural Networks; examples of this can be found in Diffusion models [90], equivariant neural networks [45], hyperbolic/parabolic CNNs [213], Graph Neural Diffusion [35], among others. One of the methods that appeared recently is the Neural Operator [117], which focuses on learning the mapping between function spaces.

This section will explore using Neural Operators to learn the solution operator of the NS equation. Neural Operators, specifically the Fourier Neural Operator (FNO), have been used for learning problems arising from fluid flow simulations [138]. The main objective is to explore using FNO in different configurations to effectively learn the Operator that evolves the fluid flow state in time. The second objective is to propose a framework to train models that can mitigate the issue of numerical stability that appears when trying to extrapolate in time outside the training domain in order to perform long-time prediction of the fluid flow and test them on different Neural Operator models. We train and test the models on 2D Kolmogorov Flow [36] and evaluate if the proposed methodology can overcome the numerical stability issues. The question of whether the Neural Operators can extrapolate to a higher Reynolds number and are invariant to discretization is also addressed. This chapter closes with the conclusions of the proposed methods and draws some perspectives for further inquiry.

5.2 Problem Statement: Learning to Simulate Turbulence from Data

The subject of this chapter is the study of Deep Learning architectures for learning to replicate turbulent flow simulations. The goal for the Deep Learning model here is that given a set of initial conditions and/or other parameters, it can predict the next states of the turbulent flow as a numerical solver would do. We define the problem set as follows:

Problem Statement 5.1 *Being $u(\boldsymbol{x}, t)$ the solution or family of solutions of a non-linear PDE in the form $\frac{du}{dt} = \mathcal{N}(u)$, where $\mathcal{N}$ is a non-linear operator, with boundary conditions $\boldsymbol{x} \in \Omega \subset R^d$ and initial conditions $u(\boldsymbol{x}, 0) = u_0$, where $u_0 \sim P(u_0)$. We want to find an approximation $\hat{u}$ of this solution discretized on spatio-temporal grid $\{\boldsymbol{x}_i \in [0, L_i]\}$ and $\{t_i \in [0, T)\}$ given a Δx and Δt respectively. This approximation is parameterized by a Neural Network $\hat{u} = f(\eta; \theta)$ with parameters θ, that takes as input a set of features η, and whose parameters are found by solving the optimization problem*

$$\theta = \arg\min_{\theta} \mathrm{E}\left[L(u^{(i)}(\boldsymbol{x}_i, t_i), \hat{u}^{(i)}(\boldsymbol{x}_i, t_i),)\right]$$

Where L is a loss function that measures the difference between the data and its approximation, the ground-truth values $u^{(i)}(\mathbf{x}_i, t_i)$ are drawn uniformly from a data set built from the results of a simulation. In the present study, we want to find a model that approximates solutions of the Navier-Stokes equation exhibiting turbulence. To do so, we will remove the explicit time dependence of the Neural Network approximation by employing an auto-regressive model, where the time dependence is implicitly modeled because it takes the previous state of the solution as input features. Its output is the next stage of this one:

$$\hat{u}_{t+\Delta t}(\mathbf{x}) = f\left(\hat{u}_t(\mathbf{x}); \theta\right) \tag{129}$$

In a supervised learning setting, we find the parameters that minimize the mean-squared error between the predicted outputs and their target value. Given a data-set of N samples composed of discrete trajectories of u of length T with a given Δt, the constrained risk minimization objective is:

$$\theta = \arg\min_{\theta} \sum_{i=1}^{N} \sum_{t_j=1}^{T-1} \left\| u_{t_{j+1}}^{(i)} - f(u_{t_j}^{(i)}; \theta) \right\| + \lambda \cdot L_{constraints} \tag{130}$$

Where $\lambda \cdot L_{constraints}$ is a term for enforcing physical, statistical, or stability properties to the model predictions, as we will see later.

5.3 Background

5.3.1 Deep Learning for Reduced Order Models of Fluid Flows

Deep learning is an interesting approach to achieve lower computational requirements of simulations because of their capability of learning non-linear representations that can overcome the limitations of projection-based reduced order models for problems with slowly decaying Kolmogorov n-widths (advection-dominated problems or problems with other strong non-linearities) [129]. One DL approach, inspired by reduced order modeling frameworks, consists of two stages: first, a latent representation for the fluid flow is learned, normally through a Convolutional Autoencoder; then, the evolution of these latent variables is learned with an LSTM model [163][173]. To enforce the model to respect conservation laws, optimization constraints related to desired physical properties can be added [277] or by the addition of layers that enforce divergence-free predictions if the flow is incompressible [165]. A similar approach was used for fluid simulations for computer graphics [261]. In this case, the autoencoder and temporal prediction network (also used an LSTM) are trained end-to-end to have a temporally coherent latent space simultaneously. The latent space is divided as well between vector and scalar fields. LSTMs have also been used in an unsupervised learning setting where a GAN is trained to generate fluid flow from a random variable. In this case, the generator is an LSTM that generates a temporal latent space sequence at the first stage at a second stage; a convolutional decoder projects this latent space to the flow field [111].

One of the disadvantages of this approach is that LSTMs are hard to train and struggle to capture long-range interactions. This translates into difficulty maintaining numerical stability when $T \to \infty$. Models that have shown promise in overcoming this issue are transformers. Transformers have been employed for learning fluid simulations and can be combined with Autoencoders [278][231]. Autoregressive CNNs have been used to learn fluid simulations in the quest for more scalable options with better training dynamics. This bypasses the stage of learning a latent space first, and instead, the model is trained to predict the next time step, similar to what numerical methods do. These models have a lower computational footprint restricted only by GPU memory, which depends on the

dimensionality of the fluid flow state. Auto-regressive U-NETs [251][81] and ResNets [233] have been used to learn a variety of fluid flows. Neural operators [117] are methods inspired by the problem of learning on infinite dimensional spaces, intending to learn forward and inverse problems that arise from PDEs in a discretization invariant way. Neural operators have outperformed CNNs in learning the solution of PDEs [138][143].
Models based on CNNs are restricted to structured grids, but many fluid simulations are performed on unstructured meshes. Models using Graph Neural Networks have been applied to learn lagrangian fluid simulations [217] and simulations on unstructured meshes [187][22].

Physics-Informed Neural Networks (PINNs) can also be used to learn fluid simulations [102]. The advantage of PINNs is that they are suitable for learning from ill-conditioned problems with limited, sparse, and incomplete data. However, the learning of the solution of PDEs using NNs and gradient descent can present training instabilities and convergence issues [256][119][254][253]. PINNs use the weights of a NN to represent the solution function of a differential equation, this can be categorized as a specific application of implicit neural representation (INR), which are neural networks that are used to represent complex signals on arbitrary dimensions [161][182]. INRs are interesting because they are not attached to a specific resolution. They are learned continuous functions that can be sampled at arbitrary resolutions. This is attractive to PDE-related problems since the computational requirement remains constant and invariant to the desired discretization. One drawback of PINNs is that they do not necessarily represent a reduced computational power because they often fail to predict the dynamics outside their training data. An alternative approach is using the INR as a learned non-linear manifold that can be projected using an ODE solver. This permits the construction of non-linear and continuous ROMs [276][39][149][181]. However, these types of methods have some difficulties too. We must train a neural network for a single instance of a PDE solution; thus, its generalization capabilities are difficult. This can be overcome with meta-learning methods or hypernetworks, but further study needs to be done in this area.

The model selected for the study in the rest of this chapter is the Fourier Neural Operator [138]. The main advantages of the Fourier Neural Operator over the other Networks that we exposed previously are:

1. **Invariance to Discretization:** Can be evaluated at any point in the output domain and accepts any number of points of the input domain. The accuracy of the learned operator converges to the continuum operator, even without re-training.

2. **Universal Approximation:** Neural Operators can approximate any continuous operator defined on a compact set of a Banach space. The main

difference between Neural Operators and other Neural Networks classes is that they are expressive enough to learn from different input-output function spaces, are not restricted to learning on a fixed grid representation, and can learn long-range interactions between features. However, the models studied here are evaluated on a uniform grid because of the use of the Fast Fourier Transform. These models could be extended to non-uniform discretizations, but this is left for further work.

5.3.2 Research Objectives

After summarizing the state-of-the-art applications of Deep Learning for the development of new methods of reduced order modeling of fluid flows, several research questions arise:

- What model can we use in each situation?
- What amount of data is needed to learn a reduced representation of a fluid flow?
- What is the accuracy we can get with Deep Learning models?
- How we can train Neural Surrogate Models that are stable and physically accurate.
- What are the generalization capabilities of Deep Learning models when applied to fluid dynamics?

The following sections will address these questions by studying different Deep Learning architectures and training them to reproduce turbulent flow simulations. The key contributions of this chapter are the following:

1. A comparison of Neural Operators for learning turbulent flows in terms of accuracy.
2. A training methodology to promote extrapolation capabilities for predictions outside the time domain contained in the data.
3. Assessment of the limitations of the studied models and guidelines to overcome them possibly.

5.4 Turbulent flow Dataset

5.4.1 2D Kolmogorov Flow

The Kolmogorov flow [36] comes from solving the Navier-Stokes equation with a sinusoidal forcing:

$$\begin{gathered}\frac{\partial \boldsymbol{u}^*}{\partial t^*} + \boldsymbol{u}^* \cdot \boldsymbol{\nabla}^* \boldsymbol{u}^* + \frac{1}{\rho}\boldsymbol{\nabla}^* p^* = \nu \nabla^{*2} \boldsymbol{u}^* + \chi \sin\left(2\pi n y^*/L_y\right)\hat{\boldsymbol{x}} \\ \boldsymbol{\nabla}^* \cdot \boldsymbol{u}^* = 0\end{gathered} \tag{131}$$

Where ρ is the density, η the kinematic viscosity, n an integer describing the scale of the Kolmogorov force, and χ is the forcing amplitude per unit mass of fluid over a doubly periodic domain $[0, L_x] \times [0, L_y]$, $*$ indicates a dimensional quantity. The system is non-dimensionalized by the length-scale $L_y/2\pi$ and time scale $\sqrt{L_y/2\pi\chi}$, then the equations become:

$$\begin{aligned} \frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} \boldsymbol{u} + \boldsymbol{\nabla} p = \frac{1}{Re} \nabla^2 \boldsymbol{u} + \sin(ny)\hat{\boldsymbol{x}}, \\ \boldsymbol{\nabla} \cdot \boldsymbol{u} = 0 \end{aligned} \tag{132}$$

Where the Reynolds number is:

$$Re := \frac{\sqrt{\chi}}{v} \left(\frac{L_y}{2\pi} \right)^{3/2} \tag{133}$$

This equation is solved using doubly periodic boundary conditions in the velocity-vorticity formulation, which is obtained by taking the curl to the momentum equation:

$$\frac{\partial \omega}{\partial t} = \hat{z} \cdot \nabla \times (\boldsymbol{u} \times \omega \hat{z}) + \frac{1}{Re} \nabla^2 \omega - n \cos(ny) \tag{134}$$

Where $\omega \hat{z} := \nabla \times \boldsymbol{u}$. This term is reduced to $-\boldsymbol{u} \cdot \nabla \omega$ since the vortex stretching is null in two dimensions ($\omega \cdot \nabla \boldsymbol{u} = 0$). To solve this equation in two dimensions, the stream function Poisson equation is used to link the velocity to vorticity:

$$\begin{aligned} \boldsymbol{u} = \boldsymbol{\nabla} \times \psi(x, y)\hat{\boldsymbol{z}} \\ -\nabla^2 \psi = \omega \end{aligned} \tag{135}$$

Equations (134) and (135) constitute the Stream-vorticity formulation of NS, and solving these yields an incompressible 2D flow.

For building the dataset, the equation is solved using a pseudo-spectral method implemented in Python using the pseudospectral module of the JAX-CFD package [21][52], where the Poisson equation is solved to find the velocity field. The vorticity is differentiated, and the non-linear term is computed in physical space after it is dealised, then, time is advanced using a Crank-Nicholson scheme update. The data is generated on a 256×256 uniform $(0, 2\pi)^2$ grid and downsampled to 128×128 for learning, the forcing is set to $f(\mathbf{x}, \mathbf{y}) = 4\cos(4(y))$. Each trajectory is initialized with a Gaussian Random field to represent the initial velocity, where the maximum velocity is set to $7m/s$. To build the dataset, solutions are recorded at each $t = 1s$ after the flow becomes stationary, and two datasets are built for different Re:

The Reynolds number quantifies how turbulent the flow is. The higher the number, the more flow exhibits chaotic behavior and is harder to predict. Each dataset is subdivided into sequences of 10-time steps for training, and 10% is

ν (viscosity)	N (number of trajectories)	T (time-steps)	Re (Reynolds' Number)
10^{-2}	1000	200	100
2×10^{-3}	1000	200	500

Table 7: Datasets for 2D Kolmogorov Flow.

saved for validation, which means we will have for each problem $18,000$ trajectories for training and $2,000$ for testing.

5.5 Methodology

5.5.1 Deep Learning Models

The simulations that the neural networks will learn are 2D CFD simulations or 2D slices of simulations in 3D. The models are trained with a fixed time step Δt and fixed resolution $N_x \times N_y$. As stated in Section 2, we want the model to learn how to predict future fluid flow states given only an initial condition. This problem can be classified in ML terms as a spatiotemporal regression problem. In this case, we have a dataset of solutions to the NS equation defined in $\mathbf{R}^2$ that evolves in time. In other words, given an initial condition $u(\mathbf{x}, t_0)$, it is necessary to find a model parameterized by θ that models the solution operator of the NS equation. In many numerical methods, predicting the next time step depends on the solution at one or more previous time steps. This means the model should learn the conditional probability:

$$P_\theta(u) = \prod_{t=0}^{N_k} P_\theta(u_t | u_0, u_1, ..., u_{t-1}) \tag{136}$$

To be used in downstream applications, the learned model should:

- Extrapolate to time horizons that are not present in the dataset.
- Retain the same degree of physical accuracy for all the predictions.
- Generalize to unseen initial conditions.

Many models can be used to model the solution of a PDE. However, obtaining a model that performs with these characteristics for turbulent flow data is challenging. Recurrent Neural Networks (RNN) are initially discarded because they are generally unstable to train due to vanishing/exploding gradients, and they struggle to learn long-range dependencies [74][277]. Physics-informed neural Networks are discarded too, even though they can learn a discretization invariant representation of the solution of a time-dependent PDE, they also present learning difficulties: they usually learn a solution of a PDE for a given set of initial and boundary conditions [199], they present problems in representing multi-scale phenomenon such as turbulence [254][253]. It can be difficult

to generalize to log time horizons [119]. Another critical aspect of the learned models is their data efficiency. RNNs and PINNs lack inductive biases. While PINNs achieve this through the PDE constraints, it results in an objective function that is challenging to optimize. To exploit the inductive bias of the data and to be able to predict solutions for an arbitrary duration, the models selected for this study are Autoregressive models based on Convolutions and Neural Operators.

Neural Operators

Neural Operators are models designed to learn the mapping between function spaces. In the learning setting of this study, the objective is to learn the operator that maps elements from the initial condition function space to the solution function space. By definition, these function spaces are infinite, so if the optimal neural operator learns effectively, the solution operator of a PDE can generalize to arbitrary initial conditions. As we mentioned earlier, a Neural Operator layer is characterized by an integral transform [117]:

$$\left(\mathcal{K}_t\left(v_t;\theta\right)\right)(x) = \int_{D_t} \kappa_\theta^{(t)}(x,y)v_t(y)\mathrm{d}y \quad \forall x \in D_{t+1} \tag{137}$$

Where $\kappa_\theta^{(t)}$ is a learnable kernel function parameterized by a neural network, one way to parameterize a Neural Operator integral transform is by solving the integral in Fourier space. First, it is assumed that the kernel depends on the distance between y and x, $k_\theta(x,y) = k_\theta(x-y)$, so the integral transform becomes a convolution. Then, this convolution is performed in the Fourier domain, yielding the Fourier Neural Operator architecture (FNO) [138]. In the FNO layer, the inputs are lifted to Fourier space using the Fast Fourier Transform algorithm (FFT). Afterward, the Fourier modes are filtered by being multiplied by a learnable kernel matrix. Then, the inverse FFT transforms these outputs back to real space. Let $\mathcal{F}$ denote the Fourier transform of a function $v : D \to \mathbb{C}^n$ and $\mathcal{F}^{-1}$ its inverse:

$$\begin{aligned}\left(\mathcal{F}v\right)_j(k) &= \int_D v_j(x)e^{-2i\pi\langle x,k\rangle}\mathrm{d}x \\ \left(\mathcal{F}^{-1}v\right)_j(x) &= \int_D v_j(k)e^{2i\pi\langle x,k\rangle}\mathrm{d}k\end{aligned} \tag{138}$$

For $j = 1, ..., n$ where $i = \sqrt{-1}$ is the imaginary unit and $< \cdot, \cdot >$ denotes the Euclidean inner product on $\mathbb{R}^d$. By assuming that the kernel is invariant to translation: $k(x,y) = k(x-y)$ for some $k : D \to \mathbb{C}^{m\times n}$ then the integral kernel transform becomes a convolution that can be calculated in Fourier space:

$$\left(\mathcal{K}(v_t)\right)(x) = \mathcal{F}^{-1}(\mathcal{F}(\kappa)\cdot\mathcal{F}(v))(x) \quad \forall x \in D. \tag{139}$$

Then k is parameterized by its Fourier coefficients:

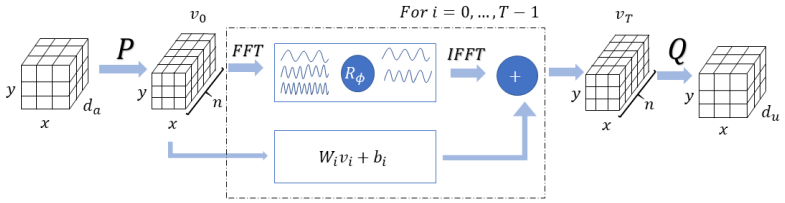


Figure 68: Diagram of an FNO architecture. The input tensor is lifted to a higher dimensional feature space. The FNO layer transforms its input to the Fourier domain, which is filtered and then transformed back to the original domain. This operation is repeated T times before passing to the projection layer that gives the output the dimensionality of the target function space.

$$(\mathcal{K}(v_t))(x) = \mathcal{F}^{-1}\left(R_\phi \cdot \mathcal{F}(v)\right)(x) \quad \forall x \in D \tag{140}$$

A graphic illustrating the FNO architecture can be seen in figure 68. The complex-valued tensor R_ϕ has dimensions $k_{max} \times d_v \times d_v$ and represents a collection of Fourier modes where k_{max} is the maximal number of modes that are truncated from the Fourier series expansion, this is a parameter set up before training the model. Neural networks tend to learn first low-frequency modes and later learn the higher-frequency modes during training [196]. This behavior is known as implicit spectral bias. As the FNO model performs the linear operation in Fourier space, the frequency strength of each frequency mode is related to the spectrum of the resulting model. The FNO also presents spectral bias as it is a property of models trained by first-order learning algorithms. However, learning lower-frequency modes is important because these frequencies represent larger scales that have higher magnitudes than high-frequency parts for dissipative systems such as the Navier-Stokes equation. In an ideal setting, we should choose the maximum number of frequencies learned by the model depending on the smallest scales present in the target NS equation solution. Hence, this requires knowledge of the task and hyperparameter tuning. If the value of k is too small, the model may be unable to capture higher frequency components, leading to overfitting. On the contrary, if the value of k is too large, this may lead the model to overfit.

Increasing the maximum frequency and depth in an FNO can lead to a substantial increase in model size, which translates into an increase in the computational cost of the training stage. One way to increase the network capacity while keeping the memory overhead small is by changing the design of the Neural Operator itself. Following the guidelines presented in [197], an FNO can be designed following the architecture of a U-NET [208]. This model called UNO (U-shaped Neural Operator), is built of an encoder branch that progressively maps the input function to functions defined on smaller domains. And a decoder branch that reverses this operation to generate an appropriate output function with skip connections from the encoder. This encoder-decoder design allows for better multi-scale learning while reducing the computational cost (see fig. 69).

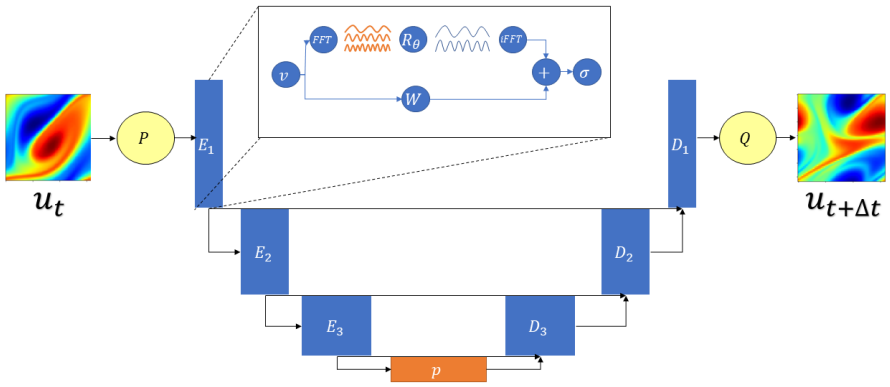


Figure 69: Diagram of the UNO architecture. Each decoder block is an FNO whose output is in a domain smaller than the previous. The inverse is the case for the decoder by adding a skip connection that connects each decoder block to the activations of corresponding encoder blocks. The middle processor layer is a standard FNO layer on the reduced domain.

Another variation we will use for this study is a modified U-NET architecture. The hypothesis is that convolutions may be just what we need to learn fluid dynamics simulations. As seen in previous works [64] [233] [251] [81], variations of ResNETs and U-NETs were implemented to learn fluid flow simulations. The reason convolutions are effective is that they tend to learn local spatial information. Our implementation consists of using convolutional layers on the upsampling and downsampling paths and using Fourier Neural Operator layers in the networks' middle section. Since the FNO layers will operate on a reduced dimensional space, they will be easier to train since there are fewer frequencies to learn. Thus, the size of the filter can be smaller.

In summary, the parameters for the architectures that will be used for this study are:

- **Fourier Neural Operator (FNO)**: Following their original implementation, the FNO used for learning problems has 4 layers, each with 24 Fourier modes. The lifting layer is a fully connected layer with 32 features. The exact number of features is kept along all layers. The Swish activation function is used. This network's total parameters is **9.4M**.
- **U-shaped Neural Operator (UNO)**: The UNO model used has 3 encoder and decoder blocks and 1 processor block. The lifting and projection layers are Fully connected layers with 32 output features. The output layer is a linear layer that outputs to the target variable corresponding number of features. The encoding path reduces the spatial dimension by 3/4 and increases the number of features by 3/2, and the inverse applies to the decoding path. We have three FNO layers for the decoder with $[48, 72, 108]$ features, respectively. The processor layer is just one FNO layer without dimension or feature scaling. The output of each encoder block is skip-connected via concatenation to each decoder block. Every FNO layer in this model has a $k_{max} = 12$ mode. The Gaussian error Linear unit (GELU) is used as activation, and Layer Normalization is used. This model's total parameters is **28M**.
- **Fourier U-NET (U-FNET)**: This network follows a encoder-processor-decoder framework [217][233]. This standard U-NET model has intermediate FNO layers between the encoder and decoder. The encoder and decoder blocks are built with residual blocks (see fig. 70) with two convolutional layers of a kernel of size 3×3, one of the convolutions is zero-initialized, GroupNorm with 1 group, and the GELU activation is used. The down and upsampling is performed using bilinear interpolation. Each block scales the dimensions by a factor of 2. The processor layer is an FNO with 3 layers and $k_{max} = 8$ modes. The output layers are a 3×3 convolutional layer followed by a linear, fully connected layer. The encoding path multiplies the features by $[2, 4, 16]$. The model designed this way has **25.9M** parameters.

5.5.2 Training methodology

Spatio-temporal sequences are being predicted utilizing autoregressive image-to-image models within our study. The process involves training a neural network to discern the subsequent time step in the sequence based on its preceding inputs. Employing the one-step strategy, we exclusively forecast a single time-step, given solely the previous one. This methodology resembles the incremental nature of computational fluid dynamics (CFD) solvers, whereby the system's state is computed step-wise using numerical integration techniques.

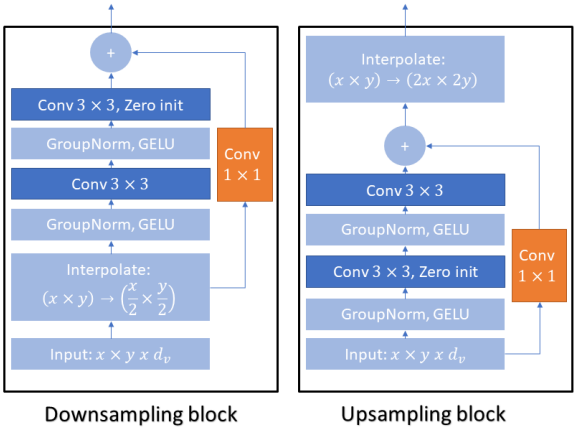


Figure 70: Downsampling and upsampling blocks for the U-FNET architecture.

The models are trained in a supervised setting, wherein the projected sequence is compared to the target sequence using mean squared error. Regularization terms are added to enforce physical accuracy and numerical stability, and these terms are modulated by a constant multiplier λ (see algorithm 2). Due to the datasets consisting of lengthy sequences, complete training on an entire sequence is constrained by the available GPU memory. Moreover, these autoregressive models are susceptible to error propagation as time steps are unfolded during prediction and training, potentially leading to instability. Another technical inconvenience spatiotemporal data faces is that the GPU memory fills up quickly for long sequences. To address this issue, we propose dividing the sequences into shorter lengths. In our investigation, sequences consisting of 10-time steps have proven to strike a favorable balance between memory requirements and stability. This decision can be justified assuming our datasets exhibit ergodicity and statistical stationarity over time. Consequently, we can treat the more minor sequences as independent realizations, thereby enhancing the overall robustness of the model.

Another consideration for training AR neural networks is whether to train the Neural Networks to predict the next state of the solution entirely or to predict the difference between time steps. In previous work, it has been demonstrated that learning the update instead of the complete state in some cases results in better results [217][233]. Also, there is a link with numerical integrators, where the AR model can be seen as an Euler discretization of the dynamics:

Algorithm 2: AutoRegressive NN training

Data: $\{(U_n(x,y,t))\}_{n=1}^{N_{data}}$, a dataset of spatiotemporal fields from a CFD simulation.

Input: θ, The initial parameters of the NN.

Output: $\hat{\theta}$, trained parameters of the NN.

Hyperparameters: lr, learning rate, $N_{epochs} \in \mathbb{N} \in (0, \infty)$

for $i = 1, 2, ..., N_{epochs}$ **do**

  for $n = 1, 2, ..., N_{data}$ **do**

    $\hat{U}_n^{t_0} \leftarrow (\mathrm{U}_n^{t_0};\, \theta)$;

    foreach $U_n^t, t \in (0, T)$ **do**

      $\hat{U}_n^{t+1} \leftarrow \mathrm{G}_\theta(U_n^t)$;

    end

    $loss(\theta)$

    $\leftarrow 1/\mathrm{N}_T \sum ||U_n - \hat{U}_n||^2 + \lambda_1 L_{pde}(U_n, \hat{U}_n) + \lambda_2 L_{stability}(U_n, \hat{U}_n)$;

    $\theta \leftarrow$ **Adam**$(\theta,\, \nabla loss(\theta))$;

  end

end

return $\hat{\theta} = \theta$

$$
\begin{aligned}
\frac{dU}{dt} &= \mathcal{G}_\theta(U) \\
\frac{U^{t_i} - U^{t_{i-1}}}{\Delta t} &= \mathcal{G}_\theta(U^{t_{i-1}}) \\
U^{t_i} &= U^{t_{i-1}} + \mathcal{G}_\theta(U^{t_{i-1}})
\end{aligned}
\tag{141}
$$

The Euler discretization will be used for this study, but that does not mean other update rules could also be applied. This method was used with the U-FNET to differentiate between networks trained to predict the update or the full state. We will call them U-FNET-Euler and U-FNET. Regarding the optimization used, the Adam optimizer was used with gradient clipping with a norm of 1 to prevent exploding gradients [183]. This was observed to be useful during the first stages of learning and allowed the use of a learning rate of 0.001. The models were implemented using JAX [21], flax [84], and optax libraries [87]. The training was carried out on 4 NVIDIA Tesla V100 GPUs with a batch size of 32. This work was granted access to the HPC resources of IDRIS under the allocation 2023-AD011013548R1 made by GENCI.

5.5.3 Gradient Loss term

The multi-scale behavior of turbulent flows is a significant characteristic wherein features across both high and low-frequency ranges hold importance. It should be noted that Neural Networks encounter difficulties when attempting to learn functions with high fluctuations. The Fourier Neural Operator (FNO) has been developed to address this limitation, enabling more effective learning of non-local features. However, suppose optimization solely focuses on the L^2 loss using first-order optimization methods. In that case, the learning process may predominantly capture large-scale behavior, while smaller scales may require more time or may not be learned at all, resulting in noise within this range.

To mitigate this issue, an approach is proposed wherein derivative information is incorporated into the loss function. Including derivative information reduces the errors associated with smaller scales, improving physical approximation. Some researchers have referred to this approach as Sobolev training, as described in the work by Czarnecki et al. (2017) [48]. It is worth noting that this methodology shares similarities with physics-informed loss, as presented by Raissi et al. (2019) [198]. However, in Sobolev training, the focus is on the partial derivatives and relaxing the constraint on the partial differential equations (PDEs) while operating within the supervised learning framework.

For the construction of the Sobolev loss, we consider temporal and spatial derivatives up to the second order, as these derivatives are inherent in the Navier-Stokes equations:

$$L_{pde} = \mathbb{E}\left[\left\|\nabla\hat{U} - \nabla U + \nabla^2\hat{U} - \nabla^2 U + \frac{dU}{dt} - \frac{d\hat{U}}{dt}\right\|^2\right] \tag{142}$$

5.5.4 Promoting stability through regularization

Turbulent flows are characterized by their chaotic nature, rendering them challenging to predict due to their high sensitivity to perturbations. Even a minor deviation from the expected trajectory can result in a significantly different future path, commonly called the butterfly effect. Predicting chaotic systems' evolution poses difficulties for numerical methods due to the rapid propagation of small numerical errors, leading to non-physical outcomes. To address this issue, numerical schemes typically employ robust time-integration techniques and minimize the time step (Δt) to maintain accuracy. On the other hand, numerical methods achieve numerical stability by their numerical dissipation properties that can be inherent to the method or enforced.

Similarly, Neural Networks encounter obstacles when attempting to predict chaotic behavior. Previous studies have utilized Recurrent Neural Networks (RNNs) for predicting attractors in dynamical systems with finite-dimensional state spaces or lower complexity Partial Differential Equations (PDEs) [133][73][70].

However, since understanding neural networks' numerical properties is limited, it is unsure whether a NN can exhibit numerical dissipation or be numerically stable for arbitrary time-series data.

Temporal stability has been a critical concern in training Neural Networks for predicting solutions to the Navier-Stokes and Euler equations, as highlighted in other research papers [216][22][144]. What these papers do essentially is to try to learn dissipative dynamics through regularization. This is similar to training for denoising. The inputs are perturbed with adversarial attacks coming from a different probability distribution. Then, the model is taught to mitigate or suppress this noise by minimizing the distance of the prediction with the added perturbation ϵ and the target state. A stability term is introduced into the loss function following the formulation:

$$\begin{aligned} L_{stability} &= \mathbb{E}[\mathcal{L}(\mathcal{G}_\theta(U^k + \epsilon), U^{k+1})] \\ L_{stability} &= \frac{1}{N_{data}} \frac{1}{N_t} \sum_{n=1}^{N_{data}} \sum_{i=1}^{N_t} \left\| U_n^{t_i} - \mathcal{G}_\theta(U_n^{t_{i-1}} + \epsilon) \right\|^2 \end{aligned} \tag{143}$$

The next step consists of determining how to add the perturbations to the inputs of our model. There are mainly two approaches that have been used to promote stability in Autoregressive Neural Networks. The first method adds Gaussian noise to the inputs with a constant variance [216]. This study uses Gaussian Noise of $\sigma = 0.01$. The other method to add the perturbations is to let the model do unrolled predictions for a small amount of the time steps and then only backpropagate to the last time step. This is called the pushforward method [22], (see fig. 71).

This work will use pushforward and denoising training methods to promote stability. First, models are trained with the pushforward method. When the results converge regarding the training MSE, the stability loss is switched with the denoising method to help enforce numerical dissipation further. In this work, combining both techniques bears a more accurate result than exclusively using one of these losses.

5.6 Results and Discussion

5.6.1 Evaluation Metrics

The evaluation of computational fluid dynamics (CFD) models plays a crucial role in assessing the performance and accuracy of numerical simulations. Deep Learning models used to predict fluid flow also need to be evaluated in a similar rigorous way, beyond the commonly used metrics used in Machine Learning. However, evaluating these deep learning-based CFD models' effectiveness requires carefully selecting and applying appropriate evaluation metrics. The evaluation metrics serve as quantitative measures to assess the quality and reliability of the predictions made by the deep learning models. They provide

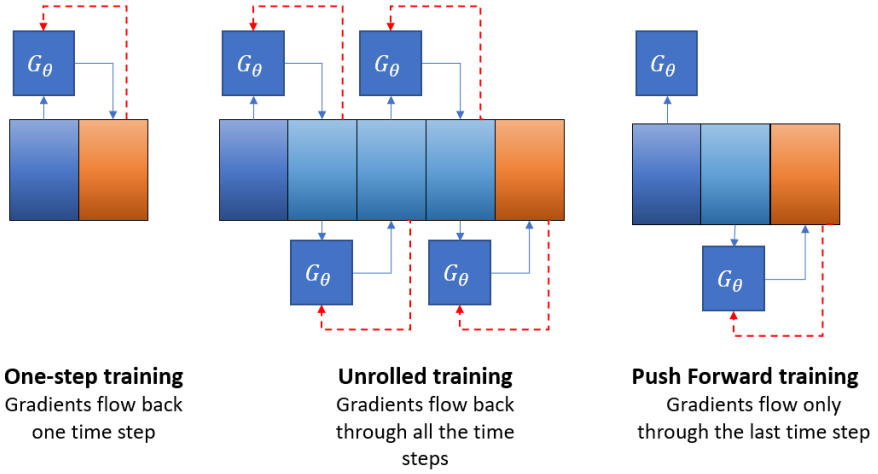


Figure 71: Training strategies. One-step training predicts only the next time step. Unrolled training predicts a whole sequence given only an initial time step. The model is unrolled N times in pushforward training but only backpropagates to the last time step.

valuable insights into the model's ability to capture flow dynamics, accurately simulate complex fluid behavior, and generalize well to unseen data. Moreover, these metrics facilitate the comparison of different models, helping researchers and practitioners make informed decisions regarding model selection and optimization strategies.

In the context of deep learning for CFD, the choice of evaluation metrics is particularly important due to the unique challenges posed by fluid flow simulations. CFD models must accurately represent a wide range of flow features, such as turbulent eddies, vortices, and shock waves, while also capturing the influence of boundary conditions and complex geometries. Furthermore, deep learning models introduce additional considerations, including the impact of architectural choices, training strategies, and regularization techniques on the overall predictive performance. This section aims to provide a comprehensive overview of the evaluation metrics employed in analyzing the models that serve as surrogate models of the fluid simulations object of this work. The metrics used here attempt to measure the accuracy of flow predictions, the ability to capture complex features, and the generalization performance on unseen data. By understanding and utilizing appropriate evaluation metrics, researchers and practitioners in deep learning for CFD can effectively assess their models' performance, identify improvement areas, and contribute to the ongoing advancement of accurate and reliable fluid flow simulations. This section will be valuable for evaluating and benchmarking deep learning models in computational fluid

dynamics, ultimately enhancing our understanding of complex fluid behaviors and their practical applications.

The following evaluation metrics were used to assess the performance of the neural networks used in this work:

- **Relative Root Mean Squared Error:** This measure is used to quantify the degree of accuracy in the physical space. It quantifies the degree of fidelity the deep learning model has in predicting the next state of the fluid flow given the previous time step. This error is calculated in the following way:

$$RRMSE = \frac{1}{N_{data}} \frac{1}{N_t} \sum_{n=1}^{N_{data}} \sum_{i=1}^{N_t} \sqrt{\frac{\left\| U_n^{t_i} - \mathcal{G}_\theta(U_n^{t_{i-1}}) \right\|^2}{\|U_n^{t_i}\|^2}} \tag{144}$$

- **Autocorrelation Function:** In turbulent flow, the autocorrelation function provides insights into a particular flow variable's temporal correlation or persistence. Chaotic and irregular fluctuations in velocity, pressure, and other flow properties characterize turbulence. These fluctuations occur over various spatial and temporal scales, challenging analyzing and predicting turbulent behavior. The autocorrelation function quantifies the correlation between a flow variable at a given point in time and its values at different time lags. It measures the similarity or relationship between the variable's past and future values. By examining the autocorrelation function, researchers can gain valuable information about the temporal patterns and organization of turbulent flow phenomena. Mathematically, the autocorrelation function for a flow variable, such as velocity or pressure, is calculated as the normalized covariance between the variable at a specific time instant and its values at different time lags. The covariance represents the statistical measure of how two variables vary together. Normalizing the covariance provides a correlation coefficient ranging from -1 to 1, where 1 indicates a perfect positive correlation, -1 represents a perfect negative correlation, and 0 suggests no correlation. In the case of turbulent flow, the autocorrelation function helps understand fluctuations' temporal behavior. For instance, a high autocorrelation at a specific time lag suggests that the flow variable exhibits a persistent pattern or trend over time. On the other hand, a low autocorrelation indicates that the flow variable's past values do not indicate its future behavior, suggesting randomness or lack of correlation between consecutive values. The autocorrelation function is widely used in turbulence research to study various aspects of flow dynamics, such as turbulence intensity, characteristic time scales, and the presence of coherent structures. It provides a quantitative measure of how long it takes for the flow variable to lose its correlation with past values, allowing researchers to identify dominant time scales and capture important flow features.

In a general sense, the autocorrelation is given by:

$$R_{ff}(\tau) = \int_{-\infty}^{\infty} f(t+\tau)\overline{f(t)}\mathrm{d}t = \int_{-\infty}^{\infty} f(t)\overline{f(t-\tau)}\mathrm{d}t \tag{145}$$

Where τ is the time-lag, and $\overline{f}$ represents the complex conjugate of the signal that, in the cases studied, is the same signal since it is real. The autocorrelation integral is a convolution and can be calculated with its corresponding algorithms. For the discrete case, the autocorrelation of the velocity or vorticity fluctuations is computed as:

$$\hat{R}_{uu}(\tau) = \frac{1}{(n-\tau)\sigma^2} \sum_{t=1}^{n-\tau} \left(U^t - \mu\right)\left(U^{t+\tau} - \mu\right) \tag{146}$$

- **Turbulent Kinetic Energy (TKE) spectrum:** The kinetic energy spectrum is a fundamental concept in turbulent flow analysis. It provides valuable insights into energy distribution across different spatial scales within the flow. It characterizes how the kinetic energy of turbulent motion is distributed as a function of the wavenumber or spatial frequency. In turbulent flows, energy is transferred from large scales to more minor scales through a process known as the energy cascade. This cascade transfers kinetic energy from the larger eddies or structures to smaller eddies, resulting in a range of spatial scales with varying energy levels. The kinetic energy spectrum quantifies this energy distribution across scales and provides information about the dominant energy-containing structures and their interactions. The kinetic energy spectrum is typically obtained by performing a Fourier analysis of the velocity field in a turbulent flow. The velocity field is decomposed into different spatial frequencies or wavenumbers, representing the different scales of motion present in the flow. The Fourier transform of the velocity field yields the kinetic energy spectrum, which describes how much kinetic energy is associated with each wavenumber. The spectrum is often presented in a graph, with the wavenumber on the x-axis and the kinetic energy (or its normalized form) on the y-axis. In an isotropic turbulent flow, where the energy distribution is uniform in all directions, the kinetic energy spectrum exhibits a characteristic shape known as the "Kolmogorov's -5/3 law" or the "inertial range." According to this law, the energy spectrum follows a power-law behavior for a range of intermediate scales with a slope of -5/3. This power-law range represents the range of scales where the energy cascade dominates, and the flow exhibits self-similarity and universal behavior. The kinetic energy spectrum provides valuable information about the energy-containing structures and their contribution to the overall flow dynamics. The high-energy scales correspond to the larger eddies or coherent structures that transfer energy to more minor scales. The low-energy scales represent the dissipative scales, where energy is dissipated into heat due to molecular viscosity. For the 2D case, the one-dimensional TKE spectrum is computed as:

$$\tilde{U}(k_x, k_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} U(x, y) e^{-i2\pi(xk_x + yk_y)} dxdy \tag{147}$$

$$E_U(k) = \int_0^{k_{max}} \left(\int \int_{\Omega_k} \tilde{U}(k_x, k_y) \cdot d\Omega_k \right) dk \tag{148}$$

- **Divergence:** In the context of fluid dynamics, the divergence is a fundamental concept that describes the behavior of a vector field, including the velocity field in a turbulent flow. The divergence represents the spatial variation or convergence/divergence of flow velocities at a given point in the flow field. Mathematically, the divergence of a vector field is calculated as the scalar quantity obtained by taking the dot product of the velocity vector field with the gradient operator:

$$Div(U) = \nabla \cdot U \tag{149}$$

  It represents the flux of the vector field across an infinitesimal control volume centered at a particular point in the flow. The divergence provides information about the flow's expansion or contraction behavior at that point. In the case of turbulent flow, the divergence is particularly relevant as it reflects the local changes in flow rates and volumetric flow in different regions. Positive divergence indicates an outward flow or expansion, while negative divergence suggests an inward flow or contraction. Zero divergence signifies a state of incompressible flow where the flow rates remain constant within the control volume. The divergence field can vary significantly in turbulent flows due to vortices, eddies, and other flow structures. These turbulent features introduce complex flow patterns, resulting in positive and negative divergence regions. The distribution and magnitude of divergence in a turbulent flow field can provide insights into flow behavior, such as the presence of flow separation, recirculation zones, or areas of high flow convergence. The divergence is chosen as a property that has to be respected by the prediction of our model and maintained for long-time horizons when extrapolating to times outside the training data distribution. If the prediction is divergence-free, it is a measure of physical accuracy since the flows studied are incompressible.

5.6.2 Results on 2D Kolmogorov Flow

In this section, the performance of the models is evaluated on the 2D Kolmogorov Flow. First, the errors for a 1-step model prediction can be observed. Table 8 presents the RRMSE and the PDE loss values calculated on test data not used during training. From this, it can be inferred that the models with the U-NET structure have better accuracy than the standard FNO. However, this measures only how close the model is to the ground truth, so it is not enough

Model	RRMSE	L_{pde}
FNO	0.2771	0.0020
UNO	0.2966	0.0132
U-FNET	0.19850	0.0138

Table 8: Errors for each model for 1-step prediction of the 2D Kolmogorov flow.

to assess the numerical stability and physical accuracy of the model.

The next step in the evaluation procedure is to assess qualitatively the predictions made by the models. From fig. 72, it can be seen samples of unrolled predictions made by the neural networks, starting only from an initial solution of the NS equation, the models are let to predict a whole sequence of 200 time steps. The three models exhibit numerical stability up to some extent. Even though there are still errors in the smaller scales, the models can dissipate these errors and not propagate them. As explained earlier, the Neural operator models were trained using the pushforward method and denoising to achieve this stability. The predictions were unrolled for three time steps during the pushforward training phase. The numerical stability was not achieved if the model was trained only on the one-step loss without regularization. After the Neural Operator models, the U-FNET model was implemented and trained. The U-FNET-Euler was trained to predict the update to the fluid flow state using a forward Euler discretization, whether the other models were to approximate the full solution operator. At first, it seemed the U-FNET-Euler was not stable for longer time horizons, but it was observed that the training could be pushed further than the FNO and UNO models, but convergence is slow in comparison. The U-FNET-Euler remained stable for the 200 time steps. Still, the errors started amplifying beyond this time horizon, whether methods trained to predict the full state instead of the residual don't exhibit this behavior.

One thing that can be observed from the vorticity plots is that after some time steps, the trajectory diverges from the ground truth but still looks like turbulence. It is needed to assess if the predictions by these neural networks are still a plausible fluid flow with the same properties and structures as the one that emulates. An important property to validate is that the flow predicted by the neural network exhibits the same temporal behaviors. This is done with the autocorrelation function. In figure 73, the plots compare the autocorrelation function of the flow fields predicted by the model and the one from the CFD code. From these pictures, it can be assessed that the DL models' predictions are close to the original flow data. Still, they produce slightly different turbulent structures in terms of time scale, corresponding to the qualitative comparison in fig.72. To quantify this error, the integral time-scale is calculated, which is the integral of the autocorrelation function. The relative mean absolute error is calculated between the time-scale of the simulated flow by the numerical method and the flow predicted by the neural networks. According to table 9,

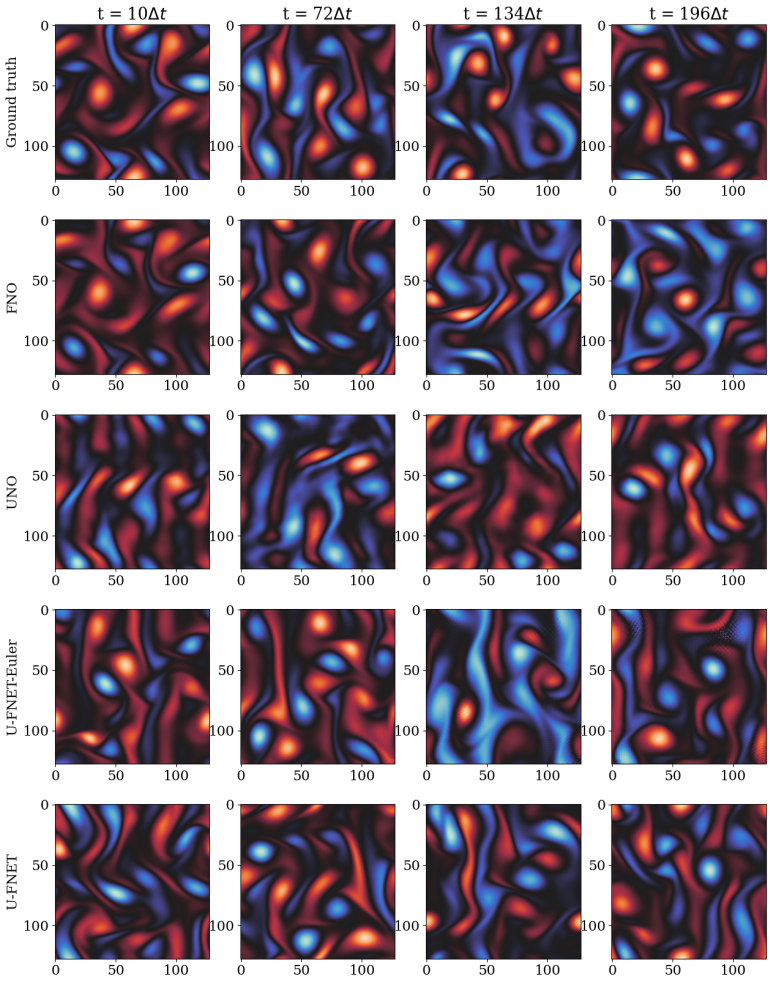


Figure 72: Predictions made by the models on the 2D Kolmogorov flow at $Re = 100$.

the U-FNET achieves lower error in the temporal correlation.

The RRMSE indicates an error in the predictions made by the NN, but it

Model	$\tau_l(s)$	r-MAE
Sim	2.15	–
FNO	1.75 s	0.185
UNO	2.31	0.073
U-FNET-Euler	2.07	0.038
U-FNET	2.13	0.012

Table 9: Integral time-scale and their relative mean absolute error.

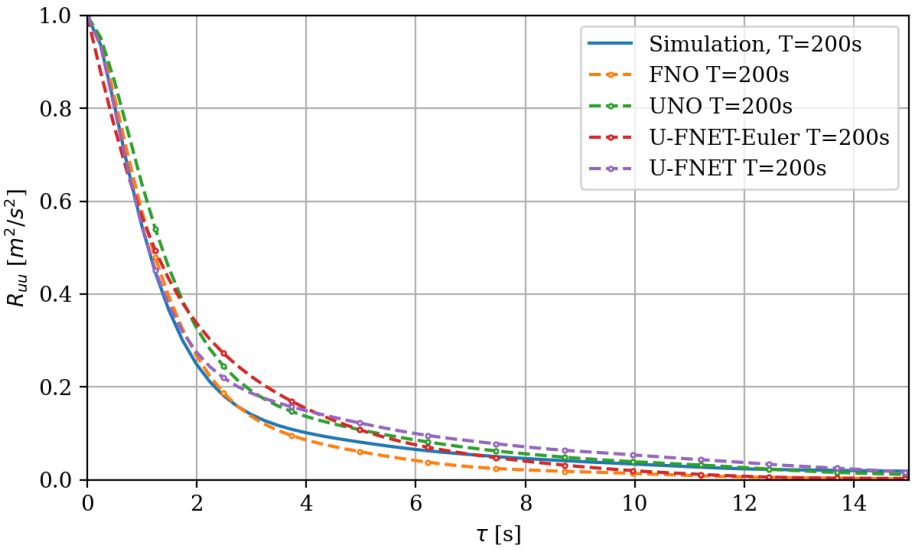


Figure 73: Autocorrelation functions of the vorticity predicted by the different models.

does not give too much information about the statistical and physical properties of the flow. For this reason, the turbulent kinetic energy spectrum is calculated at different time steps to verify that the model has the same distribution of scales. In figure 74 it can be seen the TKE spectrum at 4 different time steps under the 200 available in the dataset: $[t = 10\Delta t, 72\Delta t, 134\Delta t, 196\Delta t]$. The three figures show that the model correctly predicted all the turbulent length scales for the lower time steps. But afterward, the errors amplify, and the models overestimate the smaller length scales' values. The mean absolute error is calculated for lower and higher frequencies. As reported in table 10, the U-FNET retains better accuracy in the lower frequencies, but the error of UNO is lower on the higher frequencies. The cutoff point of higher-lower frequencies is determined as the point at which the spectrum curve starts detaching from the spectrum of the real flow.

Another important property the model should be able to retain is the zero

Model	r-MAE-low	r-MAE-high
FNO	0.1475	79.23
UNO	0.1820	02.95
U-FNET-Euler	0.0793	20.15
U-FNET	0.0901	08.24

Table 10: TKE error for the different models.

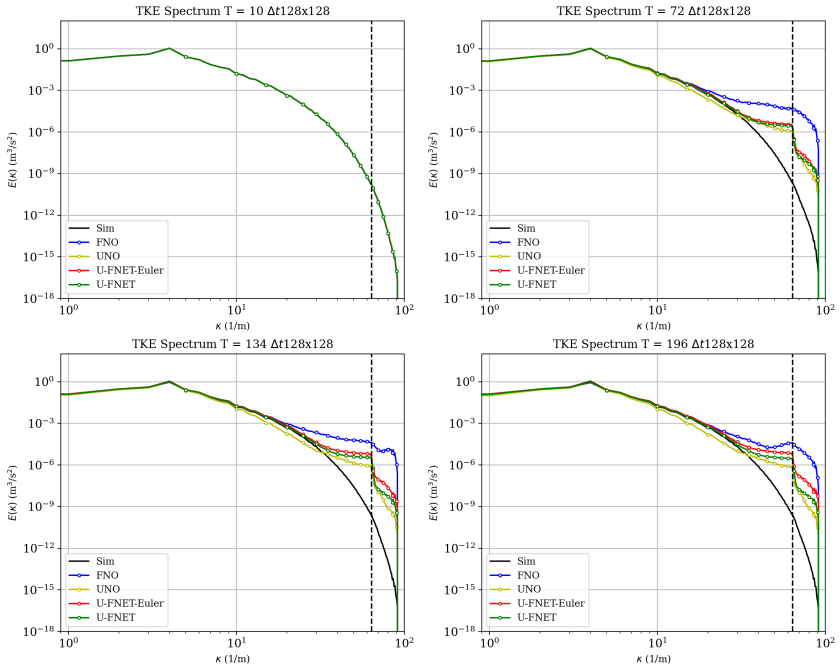


Figure 74: 1D Energy Spectrum of model predictions at different time steps for 2D Kolmogorov flow at $Re = 100$.

divergence of the velocity field. This property is present in incompressible flows and comes from the principle of mass conservation. Since the models are trained using vorticity, the velocities are calculated from the stream function, which is computed from the predicted vorticity:

$$\begin{aligned} \nabla^2 \psi &= -\omega \\ \psi &= -(\nabla^2)^{-1}\omega \\ \mathbf{U} = \nabla \times \psi &= \left(u = \frac{\partial \psi}{\partial y}, v = -\frac{\partial \psi}{\partial x} \right) \\ \nabla \cdot \mathbf{U} &= \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \end{aligned} \tag{150}$$

Learning the vorticity field and getting the velocities from the stream function means learning a divergence-free prediction by definition. The stream function is defined for incompressible flows. The stream function is the volume flux through the curve element between two arbitrary points (AP). Considering a fluid element (see fig. 75) within a Cartesian coordinate system, continuity states that if the flow is incompressible, the flow into the infinitesimally small element must be equal to the flow out of that element. In this element, u is the velocity parallel to the $x-$axis, and v is the velocity parallel to the $y-$axis. The total flow into and out of the element is:

$$\begin{aligned} \delta\psi_{in} &= u\delta y + v\delta x \\ \delta\psi_{out} &= \left(u + \frac{\partial u}{\partial x}\delta x\right)\delta y + \left(v + \frac{\partial v}{\partial y}\delta y\right)\delta x \end{aligned} \tag{151}$$

Having $\delta\psi_{in=\delta\psi_{out}}$ we get:

$$\begin{aligned} u\delta y + v\delta x &= \left(u + \frac{\partial u}{\partial x}\delta x\right)\delta y + \left(v + \frac{\partial v}{\partial y}\delta y\right)\delta x \\ \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= 0 \end{aligned} \tag{152}$$

Which means that the flow field has zero divergences. In fig. 76, the divergence plots for the predicted flow by the different models can be observed. From the figures, it can be inferred that the only models able to predict and maintain divergence-free flow are the UNO and U-FNET models. FNO and U-FNET-Euler can predict divergence-free flow, but at some points, they introduce errors that are not physical, resulting in exploding values of the divergence.

5.6.3 Effect of the loss terms

This section compares the different loss terms' influence on the final result. The U-FNET model is trained on the 2D Kolmogorov flow dataset for this task. In the following three different configurations:

1. **U-FNET-D**: This model is trained only on the mean-squared error (MS) between the one-step prediction and the data in physical space. No additional regularization term is used.

2. **U-FNET-G**: This model is trained using the MSE in physical space and uses gradient information with the gradient loss term. No stability term is used.

3. **U-FNET-S**: This model is trained using the MSE in physical space and uses the stability term with the pushforward method with 3 time steps. The gradient loss term is not used here.

It can be seen from Fig. 77, 78 and 84 that the model used to promote dissipation/stability is very important to learning a flow with proper time correlation

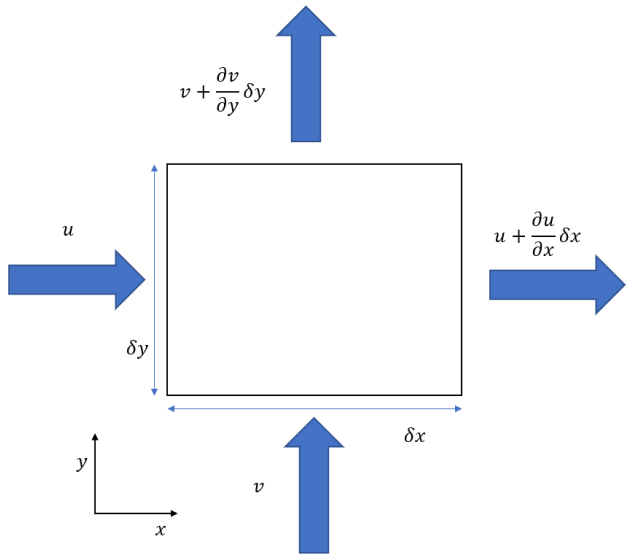


Figure 75: Fluid square element in Cartesian coordinates

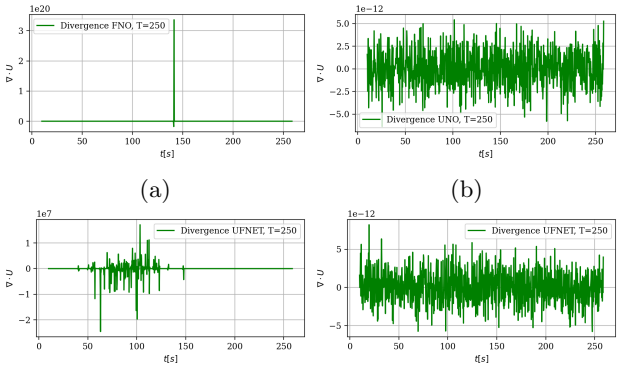
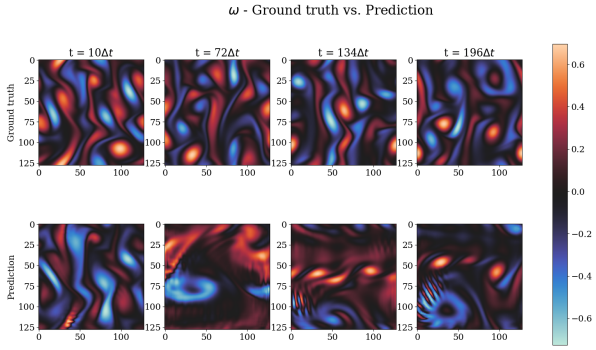
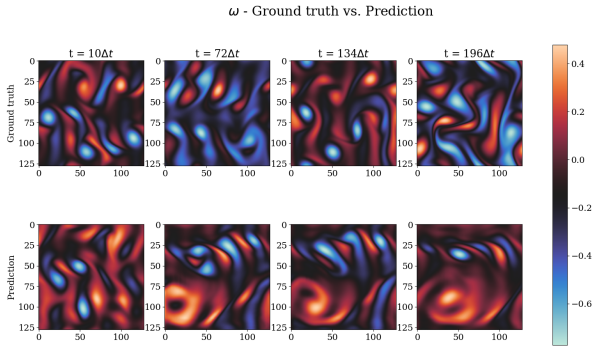


Figure 76: Divergence predicted by a) FNO, b) UNO, and c) U-FNET-Euler d) U-FNET.

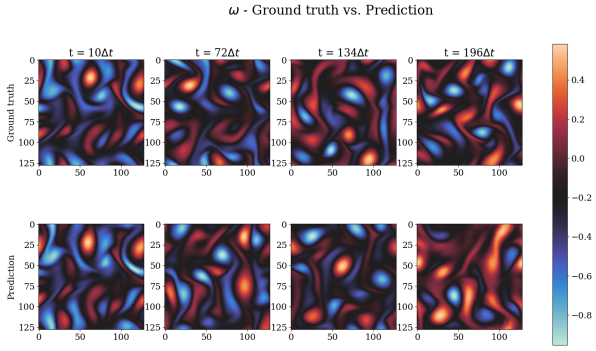
and energy distribution along scales. Just training on data is not enough, and the model predicts a highly correlated flow that is not turbulent. The numerical instability problem doesn't improve when adding the gradient term, but some high-frequency fluctuations are dampened. When the stability term is used for training, the model predicts stable fluid flow with a one-step RRMSE of **0.2204** as reported in table 11, just slightly higher than the U-FNET trained with all the regularization terms and less than the FNO or UNO. The model trained



(a)



(b)



(c)

Figure 77: Vorticity samples predicted by a) U-FNET-D, b) U-FNET-G, and c) U-FNET-S.

Model	RRMSE	L_{pde}
Data-only	0.2048	0.0269
Data+PDE	0.1819	0.0179
Data+Stability	0.2204	0.0296

Table 11: Errors for each model for 1-step prediction of the 2D Kolmogorov flow by U-FNET trained with different regularization.

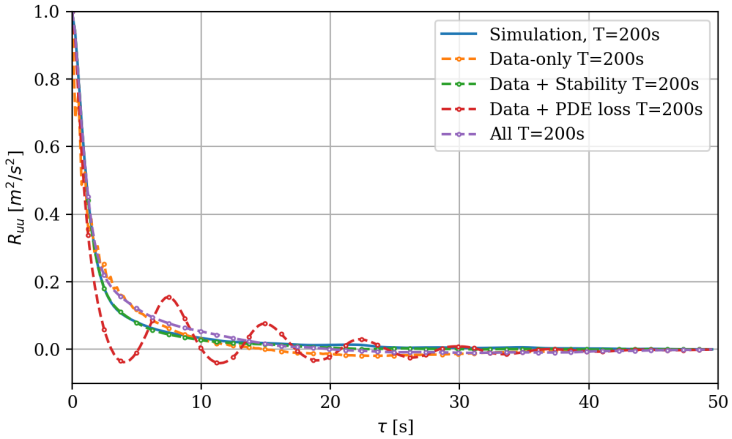


Figure 78: Comparison of Autocorrelation functions for predictions made by U-FNET trained with different regularization.

Model	$\tau_l(s)$	r-MAE
Sim	2.15	–
Data-only	1.67 s	0.185
Data+PDE	1.44	0.328
Data+Stability	1.97	0.084
All	2.13	0.012

Table 12: Integral time-scale and their relative mean absolute error.

with the stability loss can recover similar spectral error to the model trained with the gradient loss term, as seen in figure 84 and table 13. However, the model without the stability term performs poorer in learning the appropriate temporal structure. From these results, it can be concluded that the stability term is necessary for learning models that behave closer to numerical solvers. the contribution of gradient regularization or Sobolev training is to reduce the error further, but it alone does not ensure stability.

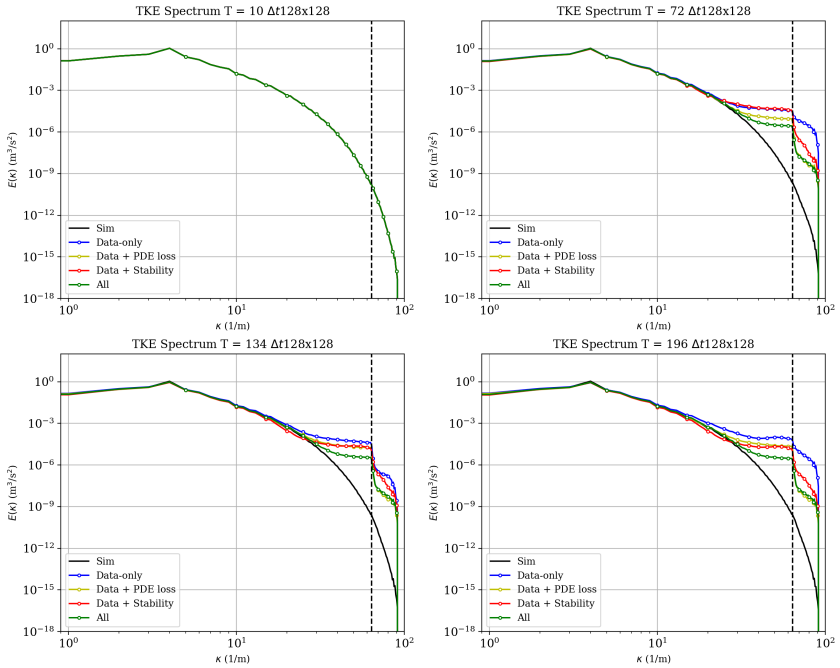


Figure 79: 1D Energy Spectrum of model predictions by U-FNET trained with different regularization.

5.6.4 Learning at a higher Reynolds' number

The next question is which model can predict turbulence at a higher Reynolds number. In the experiments, the only model able to converge was the U-FNET. The other models got stuck at a point where no turbulent fluctuations were produced. To train the U-FNET, transfer learning was used, meaning that the weights of the trained model at $Re = 100$ were used as the initial weights for teaching the model at $Re = 500$. The relative root mean squared error and the gradient loss are seen in table 14.

Next, autocorrelation (fig. 81b), and TKE spectrum fig. 81a are presented. There is good agreement with the simulation data, with errors in values and time scale close to the ones of the U-FNET at a lower Re as it can be seen in tables 14 and 15. However, the errors in the energy spectrum are higher, as seen in table 16. This might be because the flow contains more energy in a bigger range of scales, so the learned frequencies are not enough to compensate for the behavior of higher wave numbers. There is no clear reason why only this model worked for this problem. Being the only difference in the convolutional layers, it can be concluded that this is possibly the reason because the convolutions learn representations based on local features that make learning in the Fourier space easier. This needs further investigation but is a sign that embedding inductive

Model	r-MAE-low	r-MAE-high
Data-only	0.1591	198.32
Data+PDE	0.1136	57.43
Data+Stability	0.1528	41.02
All	0.0907	8.24

Table 13: TKE error for the different models.

Model	RRMSE	L_{pde}
U-FNET	0.1614	0.0343

Table 14: Errors for U-FNET predictions at $Re = 500$.

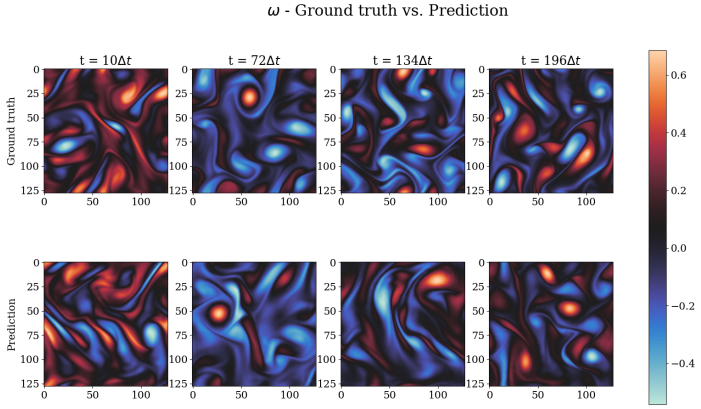
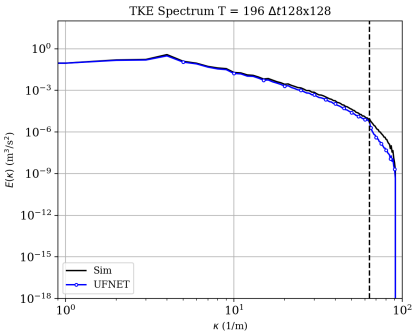
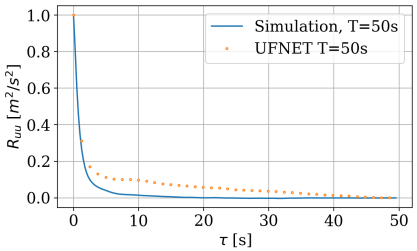


Figure 80: Samples of vorticity predictions by U-FNET at $Re = 500$

biases in the models can help learn complex features.



(a) Energy spectrum at $Re = 500$.



(b) Autocorrelation function of vorticity at $Re = 500$

Model	$\tau_l(s)$	r-MAE
Sim	1.23	–
U-FNET	1.22	0.010

Table 15: Integral time-scale and their relative mean absolute error.

Model	r-MAE-TKE-low	r-MAE-TKE-high
U-FNET	0.6020	112.27

Table 16: Errors for U-FNET predictions at $Re = 500$.

5.6.5 Zero-Shot Super-Resolution at $Re = 100$

One property of Neural Operators is discretization invariance. For a model to be discretization-invariant, it must work for any input function discretization, can be evaluated at any point of the output function, and converge to a continuum operator as the discretization is refined [117]. Other types of neural networks aren't discretization-invariant because they generally take their inputs and outputs as finite-dimensional vectors. Coordinate-based NNs, such as those based on Implicit Neural Representations and PINNs, satisfy the property that they can work on any domain discretization. Still, they fail to approximate between function spaces. Thus, they have limited generalization capabilities. CNNs aren't discretization-invariant because they do not converge with grid refinement due to the change in the receptive field when applied to different resolutions. A common approach to achieving super-resolution using Neural Networks is to include some form of interpolation or upsampling, which bounds the model to a deterministic number of upscaling factors. In the model U-FNET studied here, we use Convolutional Residual blocks with interpolation as layers of the Encoder and Decoder part. This makes the model, not a Neural Operator even though the FNO layers are used in the middle.

In this section, we evaluate the models at a higher input resolution. The models were trained on a downscaled version of the input with a resolution of (128×128) and are now being evaluated on a grid of (256×256) that is the one the flow generated was generated on. Only the FNO could correctly predict the fluid flow at a higher resolution out of the models studied. In addition, the metrics improved compared to its predictions at a lower resolution, as seen in tables 17, 18 and 19. This means that FNO gets closer to the solution operator as the discretization gets refined, satisfying the properties of a discretization-invariant operator.

In Fig. 82, it can be seen the samples of the predicted vorticity for 200 steps look qualitatively as Kolmogorov flow. The autocorrelation function is in good agreement with the ground truth, as seen in Fig. 83a, and the error of the

Model	RRMSE	L_{pde}
FNO	0.2683	0.0016

Table 17: Errors FNO predictions at $Re = 100$ on a (256×256) grid.

integral time scale is 0.011. One interesting behavior of the FNO is that it can sustain stable predictions when predicting at a finer resolution. This is reflected in the divergence of the predicted fields, as shown in Fig. 83b, where the flow is kept divergence-free after 200 time steps.

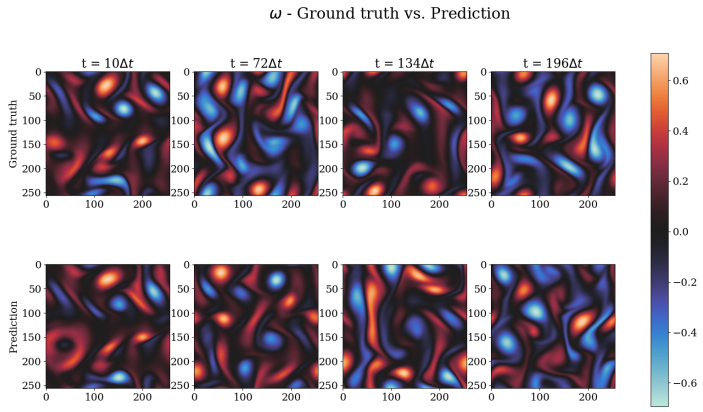
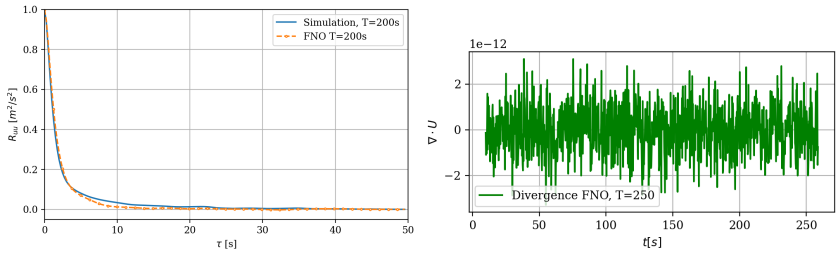


Figure 82: Samples of vorticity predictions by FNO at $Re = 100$ on a (256×256) grid.

Model	$\tau_l(s)$	r-MAE
Sim	2.15	–
FNO	1.92	0.011

Table 18: Integral time-scale and their relative mean absolute error.

The turbulent kinetic energy spectrum at different time steps is shown in Fig. 84. The spectrum starts almost identical to the ground truth, and the errors amplify as the predictions are unrolled until they plateau. Like other Neural Networks trained on gradient descent, FNOs exhibit spectral bias during learning. They tend to fit the lower frequency modes first and learn the higher frequency ones in a later stage or can converge, leaving some of them out. For the data studied, the lower frequency modes are the ones that contain the largest quantity of energy, so learning them is more important. In this case, the model learns to predict the accurate solution fitting these modes and



(a) Autocorrelation functions of the vorticity predicted by FNO at $Re = 100$ on a (256×256) grid.

(b) Divergence of the predictions by FNO at $Re = 100$ on a (256×256) grid.

dissipates the error in the higher frequency components, preventing the accumulation of errors and achieving numerical stability. For this reason, the data used for training FNO must contain the most significant number of modes possible, and the FNO itself must be expressive enough to capture these modes. This might be why the FNO couldn't learn the flow at a higher Re in the experiments. To solve this issue, the FNO layers would need to be wider to learn more frequencies, or the model could be trained at a higher resolution. This is a limitation because as the problem complexities, it increases the computational cost of the model because it needs to be bigger and use higher-dimensional data.

Model	r-MAE-TKE-low	r-MAE-TKE-high
FNO	0.1910	41.04

Table 19: Errors for U-FNET predictions at $Re = 500$.

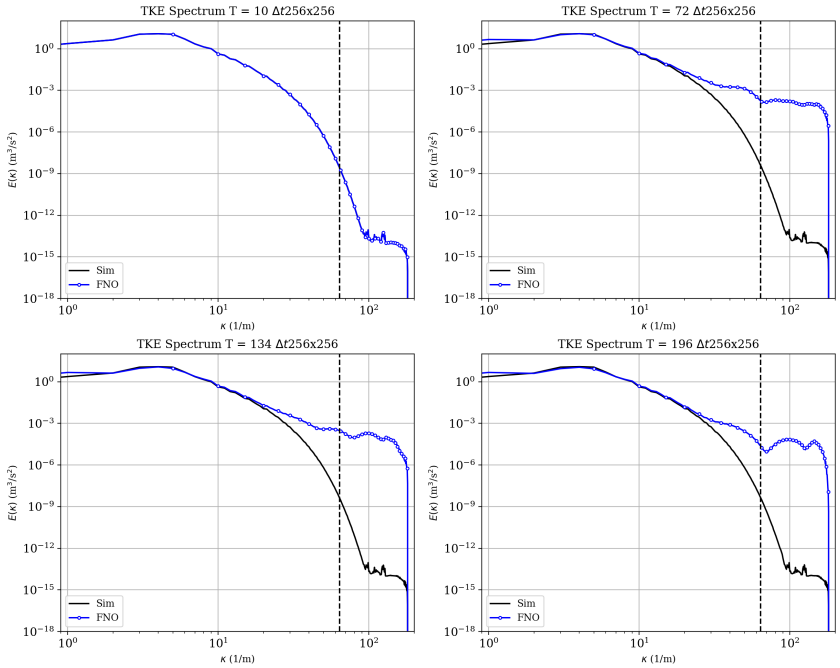


Figure 84: 1D Energy Spectrum of model predictions by FNO at $Re = 100$ on a (256×256) grid.

5.7 Conclusions on Surrogate Modeling of Turbulence with Neural Operators

Neural Operators have been implemented and trained to predict turbulent flows over a long time horizon. We studied configurations based on the Fourier Neural Operator, considered one of the state-of-the-art models in Operator Learning for physics and scientific problems. From the results presented, it can be concluded that training these models to predict spatiotemporal phenomena with chaotic behavior is not an easy task. Three neural network architectures were studied: FNO, UNO, and U-FNET, which mixes encoder-decoder convolutional paths with an FNO processor in the middle. The models were trained on a dataset built with a pseudo-spectral simulation of 2D Kolmogorov flow simulated on a 256^2 uniform grid on a $(2\pi, 2\pi)$ periodic domain, with one dataset containing trajectories at $Re = 100$ and other one at $Re = 500$. Regularization based on gradient information and a stability term to enforce numerical dissipation were used to promote accuracy and numerical stability. The experiments showed that the standard FNO exhibited good accuracy but could not be numerically stable when extrapolating outside the temporal domain present in the training data when predicting at lower resolution. The models with a U-NET-like structure, UNO and U-FNET, could predict longer time horizons without exploding at the training resolution. Out of these two models, the U-FNET reported better

accuracy and could learn more complex flow at a higher Re number. It is interesting to know the mechanisms that produce numerically stable data-driven surrogate models based on neural operators. To clarify this issue, the effect of the regularization terms is investigated by training a U-FNET with different configurations of them: one only trained on l^2 norm, a second one with the addition of the gradient term, and a third one with that addition of the stability term. Through these tests, it was confirmed that the stability term is, in fact, necessary to ensure that the models can extrapolate to longer-range predictions. This property is essential for the development of surrogate models since it allows the collection of data on smaller temporal domains, training the models, and use them to predict the quantities of interest far ahead in the future, thus reducing the computational requirements in different applications that rely on costly computer simulations. However, FNO has an interesting property that CNN models lack: discretization invariance, meaning they can predict at different resolutions. The prediction improves as this discretization is refined. Unlike U-FNET, FNO and UNO couldn't learn the flow at $Re = 500$. We assume it is mainly because the resolution used during training does not contain the most representative modes in terms of frequency. This is a limitation for FNO because it will be challenging to learn from sparse and noisy data, and having high-fidelity data for training can be computationally prohibitive for many complex problems. Future work should aim at improving these limitations. Very recent work has been oriented towards this, introducing incremental spectral learning [281] and factorized FNO that scales the architecture [242]

This study opened more questions regarding the limitations of these models and ways to improve them. The flow configuration used for training is relatively simple and not found in nature. The use of these models on more complex physics like multiphase and turbulence in 3D has been very modest, and it can be assumed that this would require different models and training strategies to tackle the higher dimensionality of these problems. Another important aspect for these models to be useful is the capability to generalize to non-uniform grids and temporal discretization. This problem needs to be addressed in the future to know the resolution limit in relation to the complexity of the data that these models can accomplish. Another important aspect regarding the generalization of neural operators is their robustness for predicting transient phenomena. The models tested here were used on data assumed to be statistically stationary and ergodic. Many problems in physics exhibit transient behavior, which can be more difficult to predict accurately by the same model since there is a change in the data distributions as it advances in time. This could be relevant to applying these models to predict phenomena such as weather, where extreme events drastically change the course of the predictions. Future work, we believe, should be oriented towards models and training methods that can tackle more complex flows with better generality to different parameters, that incorporate more physical knowledge as well as appropriate benchmarking metrics that quantify the trustworthiness of ML as a substitution of numerical methods that are going to be used in practical cases.

Part III
Epilogue

6 Conclusions and Outlook

As this thesis ends, restating the research questions that ignited this work is worthwhile. Given the recent advancements in artificial intelligence technology, thanks greatly to using algorithms based on Neural Networks. The central question of this work is how Deep Learning can then improve the study of Computational Fluid Dynamics. An ongoing and thorough critical assessment of the literature has been carried out for this task. In this review, which is contained in chapter 3 of the manuscript, the applications of ML to CFD have been categorized in three major areas, also in accordance with other reviews found in the literature [28][248]. In addition, we have added a section on the Uncertainty Quantification of ML, an area touched very little by other reviews. Uncertainty quantification is already a scientific field with many applications for determining the reliability of other engineering systems. The research of UQ in ML is growing since it is a tool that helps build trustworthy and safe AI. This will be important for CFD applications because it will measure how reliable the predictions of the state of a fluid flow made by an ML algorithm are. Then, different points addressing the concurrent challenges of applying Machine Learning to enhance CFD were addressed. From the points highlighted in the review, we chose to work on the problem of building data-driven surrogate models of turbulent flows using Deep Learning. The reason for choosing this problem was that turbulent flows are present in many applications, and their simulation requires significant computational resources. By training Deep learning algorithms to simulate these flows, these costs will be reduced and could open pathways to other applications. The main questions behind this work were, first, to determine which models work best at learning the complex multi-scale behavior of turbulence. Second, how can these models be trained to be numerically stable and not propagate errors? Third, what are the capabilities of these models to generalize to other conditions, such as different resolutions and higher Reynolds numbers?

In Chapter 4, we explored some Deep Learning methods that could be used as surrogate models of turbulent flows. We started implementing a GAN to generate samples from the solution given by the Langevin equation, which is a stochastic model of the motion of a particle in a turbulent media. Afterward, we implemented another GAN to generate 2D slices of synthetic turbulence. GANs could effectively learn the complex distribution of turbulence, and the samples generated were of high fidelity. However, GANs present difficulties in

training and huge data requirements. For this reason, it was not possible to scale to more complex cases. After the GAN, supervised learning of turbulent flow simulations was explored using a framework based on an Autoencoder plus a Convolutional LSTM model. An AE was implemented and trained to reduce the problem's dimensionality and learn a latent representation of the flow field. The model was trained on 3D homogenous isotropic turbulence fields coming from a DNS. The AE built was based on the ConvNext models that use Depthwise convolutions with the purpose of reducing the memory requirements of using 3D data, which is an important technical challenge for learning to simulate turbulent flows. The performance of the AE is comparable to other AEs found in the literature designed to learn latent representations of fluid flow fields. However, our work is one of the few that addresses turbulence in 3D. Then, the trained AE was used in the ConvLSTM framework to reduce the dimensions of the spatiotemporal sequences to ease the learning and alleviate the associated costs. The ConvLSTM could predict the turbulent flows for a few time steps and maintain the turbulent statistical properties. However, the model suffered from instability as the predictions were unrolled. Even though we tried to minimize the computational requirements for training this model to achieve better predictions that could learn the proper temporal dynamics and be able to generalize to longer time steps, it would require more data, which is costly to obtain.

This lead us in Chapter 5 to explore physics-inspired methods tailored to learn from problems that arise from the solution of partial differential equations, in this case, Neural Operators. We focused mainly on how to train Neural Operators in order to achieve numerically stable predictions over long-time horizons. We employed a strategy based on regularization and obtained good results for the 2D Kolmogorov flow. In some scenarios, the models could generalize to a higher resolution and learn a higher Reynolds number using transfer learning. However, Neural Operators still present limitations. One limitation is still the data requirements of these models. In order to work on cases that represent a bigger range of scales, Fourier Neural Operators need to be scaled up in the number of parameters in order to capture these multi-scale relationships. This increases the model parameters and associated memory costs. In the same line, we found that the more complex the case, the higher the data resolution needs to be. The other constraint of the FNO is that it works on problems on a uniform grid, making it difficult to learn from data with irregular geometries and unstructured meshes.

Before drawing the final curtain on this study, it is worthwhile to reflect on the research methodologies employed and the main challenges of this work. This research project originated at a moment when fast-paced advancements in Artificial Intelligence were being and continue to be made. As a result, this work evolved simultaneously with the literature. At the moment of carrying out this research and writing this thesis, it is possible that some other group of researchers have already or are on the verge of making a contribution that outperforms the methods exposed here or solves the open issues we have discussed.

At the beginning of the Ph.D. project, only a few examples could be found that successfully applied DL to CFD, and those that did it stayed on relatively simple configurations, even simpler than those exposed here. Now, a much bigger array of techniques can be used for different types of problems for CFD, and the community is now considering more possibilities. Still, in the current state, there is significant work to be done for Deep Learning to be applied to problems pertinent to complex real-life applications with fluids and fluid mechanics problems of interest for physicists. Aside from the findings made through the experiments, the real contribution of this work, being one of the first of its kind in our research group, which opened the collaboration between two different research fields. The work presented here could serve as a baseline that opens a path that could be followed by present and future researchers belonging to our own research group and the exterior and my own line of work. If we now turn our attention to future directions, a direct continuation of this work would be:

1. Extension to more complex cases: Implementing neural operators that can handle more complex cases like canonical turbulent flows such as Homogenous Isotropic Turbulence and Channel Flow, and learning data-driven surrogate models of atomization and spray simulations.

2. Develop other methods to ensure numerical stability: As it has been observed, numerical stability was achieved through regularization but is still tied to the capacity of the model to represent the target fluid flow. Other methods could be investigated, like designing more robust architectures to noise or auxiliary super-resolution models to refine the predictions.

3. General model for synthetic turbulence generation: using modern generative modeling techniques, a synthetic turbulence generator could be developed that generates velocity fields for a wider range of conditions such as energy spectrum, turbulence intensity, inlet geometries, and other parameters.

In a more general sense, we could anticipate what would be some of the next steps for ML + CFD:

- **Physics-Inspired Models:** Model that leverages physics knowledge to have inductive biases that boost the expressiveness and learning capabilities for fluid flow problems.

- **More integrations with CFD solvers:** There are a few examples where a neural network is part of CFD software and helps boost its performance. Not only will the network help the numerical method, but it could also learn from it in a continual way by making the solver part of the training loop, where the simulations constitute an environment that keeps feeding the model new information as new cases are simulated. In this scenario, Reinforcement Learning could play a critical role in the control of these simulations.

- **Generalization through Foundation Models:** One thing made clear in all the examples given in this work is that generalization is hard for fluid flows. One approach that has had great success in DL is using large models called foundation models, which are trained on a big corpus of data and then can be fine-tuned for different downstream tasks. This could also be done for fluid mechanics, where foundation models designed for fluid dynamics and trained on high-quality data of canonical fluid flows can learn representations that uncover the fundamental physical laws that govern these phenomena. For then, to be adapted for different tasks and configurations through fine-tuning.

- **More benchmarks and datasets:** This is self-explanatory; having easily accessible datasets and benchmark problems for different tasks that help evaluate the performance of new models would greatly help the speed of the research.

Finally, I would like to thank you, the reader, for your attention until this point. We hope to have conveyed the potential Deep Learning has for Fluid Dynamics and have sparked curiosity for the open research questions available in this field. It is with a high degree of certainty that AI will be a valuable tool at the disposal of scientists to help them make new findings and democratize the use of CFD to a bigger number of practitioners who don't have access to big computational facilities and at the same time, being able to reduce the carbon footprint of these ones. All of this makes a small contribution to the goal of driving humanity to attain a more sustainable future.

References

[1] *2022 Symposium on Turbulence Modeling: Roadblocks, and the Potential for Machine Learning*. URL: https://turbmodels.larc.nasa.gov/turb-prs2022.html (visited on 03/01/2023).

[2] Ekhi Ajuria Illarramendi et al. "Towards an hybrid computational strategy based on Deep Learning for incompressible flows". In: *AIAA AVIATION 2020 FORUM*. AIAA AVIATION Forum. American Institute of Aeronautics and Astronautics, June 2020. DOI: 10.2514/6.2020-3058. URL: https://arc.aiaa.org/doi/10.2514/6.2020-3058 (visited on 02/28/2023).

[3] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. arXiv:1701.07875 [cs, stat]. Dec. 2017. DOI: 10.48550/arXiv.1701.07875. URL: http://arxiv.org/abs/1701.07875 (visited on 01/24/2023).

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. arXiv:1607.06450 [cs, stat]. July 2016. URL: http://arxiv.org/abs/1607.06450 (visited on 04/05/2023).

[5] H. Jane Bae and Petros Koumoutsakos. "Scientific multi-agent reinforcement learning for wall-models of turbulent flows". In: *Nature Communications* 13.1 (Mar. 2022). arXiv:2106.11144 [physics], p. 1443. ISSN: 2041-1723. DOI: 10.1038/s41467-022-28957-7. URL: http://arxiv.org/abs/2106.11144 (visited on 02/21/2023).

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. arXiv:1409.0473 [cs, stat]. May 2016. DOI: 10.48550/arXiv.1409.0473. URL: http://arxiv.org/abs/1409.0473 (visited on 02/03/2023).

[7] Christophe Bailly and Daniel Juve. "A stochastic approach to compute subsonic noise using linearized Euler's equations". In: *5th AIAA/CEAS Aeroacoustics Conference and Exhibit*. Aeroacoustics Conferences. American Institute of Aeronautics and Astronautics, May 1999. DOI: 10.2514/6.1999-1872. URL: https://arc.aiaa.org/doi/10.2514/6.1999-1872 (visited on 09/18/2023).

[8] Shivam Barwey, Varun Shankar, and Romit Maulik. *Multiscale Graph Neural Network Autoencoders for Interpretable Scientific Machine Learning*. arXiv:2302.06186 [physics]. Feb. 2023. DOI: 10.48550/arXiv.2302.06186. URL: http://arxiv.org/abs/2302.06186 (visited on 02/16/2023).

[9] *Batch Normalization & Layer Normalization_batchnormalization_AI_-CSDN*. URL: https://blog.csdn.net/xwd18280820053/article/details/70237664 (visited on 07/10/2023).

[10] Andrea Beck and Marius Kurz. "A Perspective on Machine Learning Methods in Turbulence Modelling". In: *arXiv:2010.12226 [cs]* (Oct. 2020). arXiv: 2010.12226. URL: http://arxiv.org/abs/2010.12226 (visited on 05/17/2021).

[11] Andrea Beck and Marius Kurz. "A perspective on machine learning methods in turbulence modeling". en. In: *GAMM-Mitteilungen* 44.1 (2021). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/gamm.202100002, e202100002. ISSN: 1522-2608. DOI: 10.1002/gamm.202100002. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/gamm.202100002 (visited on 09/26/2023).

[12] Andrea D. Beck, David G. Flad, and Claus-Dieter Munz. "Deep Neural Networks for Data-Driven Turbulence Models". In: *Journal of Computational Physics* 398 (Dec. 2019). arXiv:1806.04482 [physics], p. 108910. ISSN: 00219991. DOI: 10.1016/j.jcp.2019.108910. URL: http://arxiv.org/abs/1806.04482 (visited on 02/20/2023).

[13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. "Representation Learning: A Review and New Perspectives". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8 (Aug. 2013). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 1798–1828. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2013.50.

[14] G Berkooz, P Holmes, and J L Lumley. "The Proper Orthogonal Decomposition in the Analysis of Turbulent Flows". In: *Annual Review of Fluid Mechanics* 25.1 (1993). _eprint: https://doi.org/10.1146/annurev.fl.25.010193.002543, pp. 539–575. DOI: 10.1146/annurev.fl.25.010193.002543. URL: https://doi.org/10.1146/annurev.fl.25.010193.002543 (visited on 02/23/2023).

[15] L. Bittner. "L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, E. F. Mishechenko, The Mathematical Theory of Optimal Processes. VIII + 360 S. New York/London 1962. John Wiley & Sons. Preis 90/–". en. In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 43.10-11 (1963). _eprint: https://onlinelibrary.wiley.com/doi/pdf pp. 514–515. ISSN: 1521-4001. DOI: 10.1002/zamm.19630431023. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.19630431023 (visited on 02/02/2023).

[16] Charles Blundell et al. *Weight Uncertainty in Neural Networks*. arXiv:1505.05424 [cs, stat]. May 2015. DOI: 10.48550/arXiv.1505.05424. URL: http://arxiv.org/abs/1505.05424 (visited on 03/01/2023).

[17] Mathis Bode et al. "Deep learning at scale for subgrid modeling in turbulent flows". In: *arXiv:1910.00928 [physics]* (Oct. 2019). arXiv: 1910.00928. URL: http://arxiv.org/abs/1910.00928 (visited on 10/20/2020).

[18] Sam Bond-Taylor et al. "Deep Generative Modelling: A Comparative Review of VAEs, GANs, Normalizing Flows, Energy-Based and Autoregressive Models". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.11 (Nov. 2022). arXiv:2103.04922 [cs, stat], pp. 7327–7347. ISSN: 0162-8828, 2160-9292, 1939-3539. DOI: 10.1109/TPAMI.2021.3116668. URL: http://arxiv.org/abs/2103.04922 (visited on 08/08/2023).

[19] Florent Bonnet et al. “An extensible Benchmarking Graph-Mesh dataset for studying Steady-State Incompressible Navier-Stokes Equations”. en. In: Apr. 2022. URL: https://openreview.net/forum?id=rqUUi4-kpeq (visited on 03/01/2023).

[20] Florent Bonnet et al. “AirfRANS: High Fidelity Computational Fluid Dynamics Dataset for Approximating Reynolds-Averaged Navier–Stokes Solutions”. en. In: Jan. 2023. URL: https://openreview.net/forum?id=Zp8YmiQ_bDC (visited on 03/01/2023).

[21] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. 2018. URL: http://github.com/google/jax.

[22] Johannes Brandstetter, Daniel Worrall, and Max Welling. “Message Passing Neural PDE Solvers”. In: *arXiv:2202.03376 [cs, math]* (Feb. 2022). arXiv: 2202.03376. URL: http://arxiv.org/abs/2202.03376 (visited on 02/09/2022).

[23] Johannes Brandstetter et al. *Clifford Neural Layers for PDE Modeling*. arXiv:2209.04934 [physics]. Sept. 2022. DOI: 10.48550/arXiv.2209.04934. URL: http://arxiv.org/abs/2209.04934 (visited on 09/13/2022).

[24] Andrew Brock, Jeff Donahue, and Karen Simonyan. *Large Scale GAN Training for High Fidelity Natural Image Synthesis*. arXiv:1809.11096 [cs, stat]. Feb. 2019. URL: http://arxiv.org/abs/1809.11096 (visited on 01/24/2023).

[25] Eoin Brophy et al. *Generative adversarial networks in time series: A survey and taxonomy*. arXiv:2107.11098 [cs]. July 2021. URL: http://arxiv.org/abs/2107.11098 (visited on 01/24/2023).

[26] Tom B. Brown et al. *Language Models are Few-Shot Learners*. arXiv:2005.14165 [cs]. July 2020. DOI: 10.48550/arXiv.2005.14165. URL: http://arxiv.org/abs/2005.14165 (visited on 02/03/2023).

[27] Joan Bruna, Benjamin Peherstorfer, and Eric Vanden-Eijnden. *Neural Galerkin Scheme with Active Learning for High-Dimensional Evolution Equations*. Number: arXiv:2203.01360 arXiv:2203.01360 [cs, math, stat]. May 2022. DOI: 10.48550/arXiv.2203.01360. URL: http://arxiv.org/abs/2203.01360 (visited on 06/23/2022).

[28] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. “Machine Learning for Fluid Mechanics”. en. In: *Annual Review of Fluid Mechanics* 52.1 (Jan. 2020), pp. 477–508. ISSN: 0066-4189, 1545-4479. DOI: 10.1146/annurev-fluid-010719-060214. URL: https://www.annualreviews.org/doi/10.1146/annurev-fluid-010719-060214 (visited on 10/28/2020).

[29] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". en. In: *Proceedings of the National Academy of Sciences* 113.15 (Apr. 2016). Publisher: National Academy of Sciences Section: Physical Sciences, pp. 3932–3937. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.1517384113. URL: https://www.pnas.org/content/113/15/3932 (visited on 06/09/2021).

[30] Steven L. Brunton et al. *Modern Koopman Theory for Dynamical Systems*. arXiv:2102.12086 [cs, eess, math]. Oct. 2021. URL: http://arxiv.org/abs/2102.12086 (visited on 02/24/2023).

[31] Alon Brutzkus and Amir Globerson. "Why do Larger Models Generalize Better? A Theoretical Perspective via the XOR Problem". en. In: *Proceedings of the 36th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, May 2019, pp. 822–830. URL: https://proceedings.mlr.press/v97/brutzkus19b.html (visited on 07/10/2023).

[32] Nianzheng Cao, Shiyi Chen, and Gary D. Doolen. "Statistics and structures of pressure in isotropic turbulence". In: *Physics of Fluids* 11.8 (Aug. 1999), pp. 2235–2250. ISSN: 1070-6631. DOI: 10.1063/1.870085. URL: https://doi.org/10.1063/1.870085 (visited on 08/24/2023).

[33] Shuhao Cao. "Choose a Transformer: Fourier or Galerkin". In: *arXiv:2105.14995 [cs, math]* (Nov. 2021). arXiv: 2105.14995. URL: http://arxiv.org/abs/2105.14995 (visited on 11/23/2021).

[34] Yang Cao, Shengtai Li, and Linda Petzold. "Adjoint sensitivity analysis for differential-algebraic equations: algorithms and software". en. In: *Journal of Computational and Applied Mathematics*. Scientific and Engineering Computations for the 21st Century - Me thodologies and Applications Proceedings of the 15th Toyota Conference 149.1 (Dec. 2002), pp. 171–191. ISSN: 0377-0427. DOI: 10.1016/S0377-0427(02)00528-9. URL: https://www.sciencedirect.com/science/article/pii/S0377042702005289 (visited on 09/01/2022).

[35] Benjamin Paul Chamberlain et al. "GRAND: Graph Neural Diffusion". In: *arXiv:2106.10934 [cs, stat]* (June 2021). arXiv: 2106.10934. URL: http://arxiv.org/abs/2106.10934 (visited on 06/22/2021).

[36] Gary J. Chandler and Rich R. Kerswell. "Invariant recurrent solutions embedded in a turbulent two-dimensional Kolmogorov flow". en. In: *Journal of Fluid Mechanics* 722 (May 2013). Publisher: Cambridge University Press, pp. 554–595. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/jfm.2013.122. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/invariant-recurrent-solutions-embedded-in-a-turbulent-twodimensional-kolmogorov-flow/78CC6B29A670F84CBC79D29408DC2674 (visited on 09/10/2021).

[37] François Charton. *Linear algebra with transformers*. arXiv:2112.01898 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2112.01898. URL: http://arxiv.org/abs/2112.01898 (visited on 02/28/2023).

[38] Peter Yichen Chen et al. *Model reduction for the material point method via an implicit neural representation of the deformation map*. Number: arXiv:2109.12390 arXiv:2109.12390 [cs, math]. Apr. 2022. DOI: 10.48550/arXiv.2109.12390. URL: http://arxiv.org/abs/2109.12390 (visited on 06/23/2022).

[39] Peter Yichen Chen et al. *CROM: Continuous Reduced-Order Modeling of PDEs Using Implicit Neural Representations*. arXiv:2206.02607 [physics]. Mar. 2023. URL: http://arxiv.org/abs/2206.02607 (visited on 05/22/2023).

[40] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. arXiv:1806.07366 [cs, stat]. Dec. 2019. DOI: 10.48550/arXiv.1806.07366. URL: http://arxiv.org/abs/1806.07366 (visited on 08/11/2022).

[41] T. Chen and H. Chen. "Approximations of continuous functionals by neural networks with application to dynamic systems". In: *IEEE Transactions on Neural Networks* 4.6 (Nov. 1993). Conference Name: IEEE Transactions on Neural Networks, pp. 910–918. ISSN: 1941-0093. DOI: 10.1109/72.286886.

[42] Tianping Chen and Hong Chen. "Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems". In: *IEEE Transactions on Neural Networks* 6.4 (July 1995). Conference Name: IEEE Transactions on Neural Networks, pp. 911–917. ISSN: 1941-0093. DOI: 10.1109/72.392253.

[43] Xiaoli Chen et al. "Solving Inverse Stochastic Problems from Discrete Particle Observations Using the Fokker-Planck Equation and Physics-informed Neural Networks". In: *arXiv:2008.10653 [physics, stat]* (Aug. 2020). arXiv: 2008.10653. URL: http://arxiv.org/abs/2008.10653 (visited on 03/18/2021).

[44] François Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. arXiv:1610.02357 [cs]. Apr. 2017. DOI: 10.48550/arXiv.1610.02357. URL: http://arxiv.org/abs/1610.02357 (visited on 08/24/2023).

[45] T. S. Cohen. "Equivariant convolutional networks". en. In: (2021). URL: https://dare.uva.nl/search?identifier=0f7014ae-ee94-430e-a5d8-37d03d8d10e6 (visited on 11/24/2021).

[46] Miles Cranmer et al. "Discovering Symbolic Models from Deep Learning with Inductive Biases". In: *arXiv:2006.11287 [astro-ph, physics:physics, stat]* (June 2020). arXiv: 2006.11287. URL: http://arxiv.org/abs/2006.11287 (visited on 11/10/2020).

[47] *CS1114 Spring 2010 - Introduction to Computing using Matlab and Robotics*. URL: https://www.cs.cornell.edu/courses/cs1114/2011sp/ (visited on 07/10/2023).

[48] Wojciech Marian Czarnecki et al. *Sobolev Training for Neural Networks*. Tech. rep. arXiv:1706.04859. arXiv:1706.04859 [cs] type: article. arXiv, July 2017. DOI: 10.48550/arXiv.1706.04859. URL: http://arxiv.org/abs/1706.04859 (visited on 05/24/2022).

[49] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. ISSN: 1063-6919. June 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[50] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv:1810.04805 [cs]. May 2019. DOI: 10.48550/arXiv.1810.04805. URL: http://arxiv.org/abs/1810.04805 (visited on 02/03/2023).

[51] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. arXiv:2010.11929 [cs]. June 2021. DOI: 10.48550/arXiv.2010.11929. URL: http://arxiv.org/abs/2010.11929 (visited on 02/03/2023).

[52] Gideon Dresdner et al. *Learning to correct spectral methods for simulating turbulent flows*. arXiv:2207.00556 [physics]. July 2022. DOI: 10.48550/arXiv.2207.00556. URL: http://arxiv.org/abs/2207.00556 (visited on 05/30/2023).

[53] Karthik Duraisamy. "Perspectives on Machine Learning-augmented Reynolds-averaged and Large Eddy Simulation Models of Turbulence". In: *arXiv:2009.10675 [physics]* (Jan. 2021). arXiv: 2009.10675. URL: http://arxiv.org/abs/2009.10675 (visited on 05/17/2021).

[54] Karthik Duraisamy, Gianluca Iaccarino, and Heng Xiao. "Turbulence Modeling in the Age of Data". en. In: *Annual Review of Fluid Mechanics* 51.1 (Jan. 2019). arXiv: 1804.00183, pp. 357–377. ISSN: 0066-4189, 1545-4479. DOI: 10.1146/annurev-fluid-010518-040547. URL: http://arxiv.org/abs/1804.00183 (visited on 11/19/2020).

[55] P A Durbin and B A Pettersson-Reif. *Statistical Theory and Modeling for Turbulent Flows; Second Edition*. en. John Wiley & Sons, 2010.

[56] Hamidreza Eivazi et al. "Physics-informed neural networks for solving Reynolds-averaged Navier$\unicode{x2013}$Stokes equations". In: *arXiv:2107.10711 [physics]* (July 2021). arXiv: 2107.10711. URL: http://arxiv.org/abs/2107.10711 (visited on 11/24/2021).

[57] Hamidreza Eivazi et al. "Recurrent neural networks and Koopman-based frameworks for temporal predictions in a low-order model of turbulence". en. In: *International Journal of Heat and Fluid Flow* 90 (Aug. 2021), p. 108816. ISSN: 0142-727X. DOI: 10.1016/j.ijheatfluidflow.2021.108816. URL: https://www.sciencedirect.com/science/article/pii/S0142727X21000461 (visited on 05/18/2021).

[58] Hamidreza Eivazi et al. "Towards extraction of orthogonal and parsimonious non-linear modes from turbulent flows". In: *arXiv:2109.01514 [physics]* (Sept. 2021). arXiv: 2109.01514. URL: http://arxiv.org/abs/2109.01514 (visited on 09/21/2021).

[59] V. Fanaskov and I. Oseledets. *Spectral Neural Operators*. Number: arXiv:2205.10573 arXiv:2205.10573 [cs, math]. May 2022. DOI: 10.48550/arXiv.2205.10573. URL: http://arxiv.org/abs/2205.10573 (visited on 06/23/2022).

[60] Cong Fang, Hanze Dong, and Tong Zhang. "Mathematical Models of Overparameterized Neural Networks". In: *Proceedings of the IEEE* 109.5 (May 2021). Conference Name: Proceedings of the IEEE, pp. 683–703. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.3048020.

[61] Alhussein Fawzi et al. "Discovering faster matrix multiplication algorithms with reinforcement learning". en. In: *Nature* 610.7930 (Oct. 2022). Number: 7930 Publisher: Nature Publishing Group, pp. 47–53. ISSN: 1476-4687. DOI: 10.1038/s41586-022-05172-4. URL: https://www.nature.com/articles/s41586-022-05172-4 (visited on 02/28/2023).

[62] Charles L Fefferman. "EXISTENCE AND SMOOTHNESS OF THE NAVIER–STOKES EQUATION". en. In: ().

[63] Kai Fukami, Koji Fukagata, and Kunihiko Taira. "Assessment of supervised machine learning methods for fluid flows". en. In: *Theoretical and Computational Fluid Dynamics* 34.4 (Aug. 2020), pp. 497–519. ISSN: 1432-2250. DOI: 10.1007/s00162-020-00518-y. URL: https://doi.org/10.1007/s00162-020-00518-y (visited on 03/03/2021).

[64] Kai Fukami et al. "Synthetic turbulent inflow generator using machine learning". In: *Physical Review Fluids* 4.6 (June 2019). Publisher: American Physical Society, p. 064603. DOI: 10.1103/PhysRevFluids.4.064603. URL: https://link.aps.org/doi/10.1103/PhysRevFluids.4.064603 (visited on 10/20/2020).

[65] Kai Fukami et al. "Model Order Reduction with Neural Networks: Application to Laminar and Turbulent Flows". en. In: *SN Computer Science* 2.6 (Sept. 2021), p. 467. ISSN: 2661-8907. DOI: 10.1007/s42979-021-00867-3. URL: https://doi.org/10.1007/s42979-021-00867-3 (visited on 09/14/2023).

[66] Kai Fukami et al. "Sparse identification of nonlinear dynamics with low-dimensionalized flow representations". en. In: *Journal of Fluid Mechanics* 926 (Nov. 2021). Publisher: Cambridge University Press, A10. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/jfm.2021.697. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/sparse-identification-of-nonlinear-dynamics-with-lowdimensionalized-flow-representations/B0A6BC75E087EE8F7B8100CF1185F29A (visited on 02/23/2023).

[67] Kai () Fukami, Taichi () Nakamura, and Koji () Fukagata. "Convolutional neural network based hierarchical autoencoder for nonlinear mode decomposition of fluid field data". In: *Physics of Fluids* 32.9 (Sept. 2020). Publisher: American Institute of Physics, p. 095110. ISSN: 1070-6631. DOI: 10.1063/5.0020721. URL: https://aip.scitation.org/doi/full/10.1063/5.0020721 (visited on 12/10/2020).

[68] Yarin Gal. "Uncertainty in Deep Learning". en. In: (), p. 174.

[69] Yarin Gal and Zoubin Ghahramani. *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning*. arXiv:1506.02142 [cs, stat]. Oct. 2016. DOI: 10.48550/arXiv.1506.02142. URL: http://arxiv.org/abs/1506.02142 (visited on 03/01/2023).

[70] Maximilian Gelbrecht, Niklas Boers, and Jürgen Kurths. "Neural partial differential equations for chaotic systems". en. In: *New Journal of Physics* 23.4 (Apr. 2021), p. 043005. ISSN: 1367-2630. DOI: 10.1088/1367-2630/abeb90. URL: https://iopscience.iop.org/article/10.1088/1367-2630/abeb90 (visited on 01/17/2022).

[71] Nicholas Geneva and Nicholas Zabaras. "Modeling the dynamics of PDE systems with physics-constrained deep auto-regressive networks". en. In: *Journal of Computational Physics* 403 (Feb. 2020), p. 109056. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.109056. URL: https://www.sciencedirect.com/science/article/pii/S0021999119307612 (visited on 06/09/2021).

[72] Nicholas Geneva and Nicholas Zabaras. "Transformers for Modeling Physical Systems". In: *arXiv:2010.03957 [physics]* (Jan. 2021). arXiv: 2010.03957. URL: http://arxiv.org/abs/2010.03957 (visited on 01/25/2021).

[73] William Gilpin. "Chaos as an interpretable benchmark for forecasting and data-driven modelling". In: *arXiv:2110.05266 [nlin]* (Oct. 2021). arXiv: 2110.05266. URL: http://arxiv.org/abs/2110.05266 (visited on 10/13/2021).

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

[75] Ian J. Goodfellow et al. *Generative Adversarial Networks*. arXiv:1406.2661 [cs, stat]. June 2014. URL: http://arxiv.org/abs/1406.2661 (visited on 01/23/2023).

[76] Will Grathwohl et al. *FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models*. arXiv:1810.01367 [cs, stat]. Oct. 2018. URL: http://arxiv.org/abs/1810.01367 (visited on 09/01/2022).

[77] Constantin Greif and Karsten Urban. "Decay of the Kolmogorov N-width for wave problems". en. In: *Applied Mathematics Letters* 96 (Oct. 2019), pp. 216–222. ISSN: 0893-9659. DOI: 10.1016/j.aml.2019.05.013. URL: https://www.sciencedirect.com/science/article/pii/S0893965919301983 (visited on 05/09/2023).

[78] John Guibas et al. "Adaptive Fourier Neural Operators: Efficient Token Mixers for Transformers". In: *arXiv:2111.13587 [cs]* (Nov. 2021). arXiv: 2111.13587. URL: http://arxiv.org/abs/2111.13587 (visited on 11/30/2021).

[79] Ishaan Gulrajani et al. *Improved Training of Wasserstein GANs*. arXiv:1704.00028 [cs, stat]. Dec. 2017. DOI: 10.48550/arXiv.1704.00028. URL: http://arxiv.org/abs/1704.00028 (visited on 01/24/2023).

[80] Gaurav Gupta, Xiongye Xiao, and Paul Bogdan. *Multiwavelet-based Operator Learning for Differential Equations*. arXiv:2109.13459 [cs, math]. Oct. 2021. URL: http://arxiv.org/abs/2109.13459 (visited on 06/16/2023).

[81] Jayesh K. Gupta and Johannes Brandstetter. *Towards Multi-spatiotemporal-scale Generalized PDE Modeling*. arXiv:2209.15616 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2209.15616. URL: http://arxiv.org/abs/2209.15616 (visited on 02/27/2023).

[82] Trevor Hastie, Jerome Friedman, and Robert Tibshirani. *The Elements of Statistical Learning*. en. Springer Series in Statistics. New York, NY: Springer New York, 2001. ISBN: 978-1-4899-0519-2 978-0-387-21606-5. DOI: 10.1007/978-0-387-21606-5. URL: http://link.springer.com/10.1007/978-0-387-21606-5 (visited on 07/10/2023).

[83] Kaiming He et al. *Deep Residual Learning for Image Recognition*. arXiv:1512.03385 [cs]. Dec. 2015. DOI: 10.48550/arXiv.1512.03385. URL: http://arxiv.org/abs/1512.03385 (visited on 01/18/2023).

[84] Jonathan Heek et al. *Flax: A neural network library and ecosystem for JAX*. 2020. URL: http://github.com/google/flax.

[85] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs)*. arXiv:1606.08415 [cs]. July 2020. URL: http://arxiv.org/abs/1606.08415 (visited on 03/27/2023).

[86] Oliver Hennigh et al. "NVIDIA SimNetˆ{TM}: an AI-accelerated multi-physics simulation framework". In: *arXiv:2012.07938 [physics]* (Dec. 2020). arXiv: 2012.07938. URL: http://arxiv.org/abs/2012.07938 (visited on 02/10/2021).

[87] Matteo Hessel et al. *Optax: composable gradient transformation and optimisation, in JAX!* 2020. URL: http://github.com/deepmind/optax.

[88] Martin Heusel et al. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. arXiv:1706.08500 [cs, stat]. Jan. 2018. DOI: 10.48550/arXiv.1706.08500. URL: http://arxiv.org/abs/1706.08500 (visited on 08/10/2023).

[89] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786 (July 2006). Publisher: American Association for the Advancement of Science, pp. 504–507. DOI: 10.1126/science.1127647. URL: https://www.science.org/doi/10.1126/science.1127647 (visited on 08/24/2023).

[90] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising Diffusion Probabilistic Models". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 6840–6851. URL: https://proceedings.neurips.cc/paper/2020/hash/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html (visited on 09/21/2023).

[91] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (visited on 09/29/2022).

[92] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. arXiv:1704.04861 [cs]. Apr. 2017. URL: http://arxiv.org/abs/1704.04861 (visited on 08/24/2023).

[93] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv:1502.03167 [cs]. Mar. 2015. URL: http://arxiv.org/abs/1502.03167 (visited on 04/05/2023).

[94] Pavel Izmailov et al. "Averaging Weights Leads to Wider Optima and Better Generalization". In: *arXiv:1803.05407 [cs, stat]* (Feb. 2019). arXiv: 1803.05407. URL: http://arxiv.org/abs/1803.05407 (visited on 06/28/2021).

[95] Andrew Jaegle et al. *Perceiver: General Perception with Iterative Attention*. arXiv:2103.03206 [cs, eess]. June 2021. DOI: 10.48550/arXiv.2103.03206. URL: http://arxiv.org/abs/2103.03206 (visited on 08/01/2022).

[96] Ameya D. Jagtap and George Em Karniadakis. "Adaptive activation functions accelerate convergence in deep and physics-informed neural networks". In: *arXiv:1906.01170 [physics]* (June 2019). arXiv: 1906.01170. DOI: 10.1016/j.jcp.2019.109136. URL: http://arxiv.org/abs/1906.01170 (visited on 10/29/2020).

[97] Ameya D. Jagtap et al. "Deep Kronecker neural networks: A general framework for neural networks with adaptive activation functions". In: *arXiv:2105.09513 [cs]* (May 2021). arXiv: 2105.09513. URL: http://arxiv.org/abs/2105.09513 (visited on 09/02/2021).

[98] Joongoo Jeon and Sung Joong Kim. "FVM Network to Reduce Computational Cost of CFD Simulation". en. In: (2021).

[99] Chiyu Max Jiang et al. "MeshfreeFlowNet: A Physics-Constrained Deep Continuous Space-Time Super-Resolution Framework". In: *arXiv:2005.01463 [physics, stat]* (Aug. 2020). arXiv: 2005.01463. URL: http://arxiv.org/abs/2005.01463 (visited on 11/25/2020).

[100] Peishi Jiang et al. "Digital Twin Earth – Coasts: Developing a fast and physics-informed surrogate model for coastal floods via neural operators". In: *arXiv:2110.07100 [physics]* (Oct. 2021). arXiv: 2110.07100. URL: http://arxiv.org/abs/2110.07100 (visited on 11/03/2021).

[101] Sebastian Kaltenbach, Paris Perdikaris, and Phaedon-Stelios Koutsourelakis. *Semi-supervised Invertible DeepONets for Bayesian Inverse Problems*. arXiv:2209.02772 [physics, stat]. Sept. 2022. DOI: 10.48550/arXiv.2209.02772. URL: http://arxiv.org/abs/2209.02772 (visited on 09/10/2022).

[102] George Em Karniadakis et al. "Physics-informed machine learning". en. In: *Nature Reviews Physics* 3.6 (June 2021), pp. 422–440. ISSN: 2522-5820. DOI: 10.1038/s42254-021-00314-5. URL: http://www.nature.com/articles/s42254-021-00314-5 (visited on 11/15/2021).

[103] Tero Karras et al. *Progressive Growing of GANs for Improved Quality, Stability, and Variation*. arXiv:1710.10196 [cs, stat]. Feb. 2018. DOI: 10.48550/arXiv.1710.10196. URL: http://arxiv.org/abs/1710.10196 (visited on 01/24/2023).

[104] Tero Karras et al. *Analyzing and Improving the Image Quality of StyleGAN*. arXiv:1912.04958 [cs, eess, stat]. Mar. 2020. URL: http://arxiv.org/abs/1912.04958 (visited on 07/26/2023).

[105] Tero Karras et al. "Alias-Free Generative Adversarial Networks". en. In: (), p. 31.

[106] Patrick Kidger. *On Neural Differential Equations*. arXiv:2202.02435 [cs, math, stat]. Feb. 2022. URL: http://arxiv.org/abs/2202.02435 (visited on 02/02/2023).

[107] Patrick Kidger et al. "Neural Controlled Differential Equations for Irregular Time Series". In: *arXiv:2005.08926 [cs, stat]* (Nov. 2020). arXiv: 2005.08926. URL: http://arxiv.org/abs/2005.08926 (visited on 08/18/2021).

[108] Patrick Kidger et al. *Efficient and Accurate Gradients for Neural SDEs*. arXiv:2105.13493 [cs, math, stat]. Oct. 2021. DOI: 10.48550/arXiv.2105.13493. URL: http://arxiv.org/abs/2105.13493 (visited on 02/03/2023).

[109] Patrick Kidger et al. *Neural SDEs as Infinite-Dimensional GANs*. arXiv:2102.03657 [cs]. May 2021. URL: http://arxiv.org/abs/2102.03657 (visited on 02/03/2023).

[110] Jinwoo Kim et al. *Pure Transformers are Powerful Graph Learners*. arXiv:2207.02505 [cs]. July 2022. DOI: 10.48550/arXiv.2207.02505. URL: http://arxiv.org/abs/2207.02505 (visited on 08/10/2022).

[111] Junhyuk Kim and Changhoon Lee. "Deep unsupervised learning of turbulence for inflow generation at various Reynolds numbers". en. In: *Journal of Computational Physics* 406 (Apr. 2020), p. 109216. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.109216. URL: http://www.sciencedirect.com/science/article/pii/S0021999119309210 (visited on 10/20/2020).

[112] Youngkyu Kim et al. "A fast and accurate physics-informed neural network reduced order model with shallow masked autoencoder". en. In: *Journal of Computational Physics* 451 (Feb. 2022), p. 110841. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2021.110841. URL: https://www.sciencedirect.com/science/article/pii/S0021999121007361 (visited on 02/23/2023).

[113] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Jan. 2017). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 04/13/2022).

[114] Georgios Kissas et al. "Learning Operators with Coupled Attention". In: *arXiv:2201.01032 [physics]* (Jan. 2022). arXiv: 2201.01032. URL: http://arxiv.org/abs/2201.01032 (visited on 02/01/2022).

[115] Dmitrii Kochkov et al. "Machine learning accelerated computational fluid dynamics". In: *arXiv:2102.01010 [physics]* (Jan. 2021). arXiv: 2102.01010. URL: http://arxiv.org/abs/2102.01010 (visited on 02/02/2021).

[116] B. O. Koopman. "Hamiltonian Systems and Transformation in Hilbert Space". In: *Proceedings of the National Academy of Sciences* 17.5 (May 1931). Publisher: Proceedings of the National Academy of Sciences, pp. 315–318. DOI: 10.1073/pnas.17.5.315. URL: https://www.pnas.org/doi/abs/10.1073/pnas.17.5.315 (visited on 02/24/2023).

[117] Nikola Kovachki et al. "Neural Operator: Learning Maps Between Function Spaces". In: *arXiv:2108.08481 [cs, math]* (Sept. 2021). arXiv: 2108.08481. URL: http://arxiv.org/abs/2108.08481 (visited on 09/08/2021).

[118] Robert H. Kraichnan. "Diffusion by a Random Velocity Field". In: *The Physics of Fluids* 13.1 (Jan. 1970), pp. 22–31. ISSN: 0031-9171. DOI: 10.1063/1.1692799. URL: https://doi.org/10.1063/1.1692799 (visited on 09/08/2023).

[119] Aditi S. Krishnapriyan et al. "Characterizing possible failure modes in physics-informed neural networks". In: *arXiv:2109.01050 [physics]* (Nov. 2021). arXiv: 2109.01050. URL: http://arxiv.org/abs/2109.01050 (visited on 03/29/2022).

[120] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". en. In: ().

[121] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (Mar. 1951). Publisher: Institute of Mathematical Statistics, pp. 79–86. ISSN: 0003-4851, 2168-8990. DOI: 10.1214/aoms/1177729694. URL: https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-1/On-Information-and-Sufficiency/10.1214/aoms/1177729694.full (visited on 08/08/2023).

[122] Marius Kurz, Philipp Offenhäuser, and Andrea Beck. "Deep Reinforcement Learning for Turbulence Modeling in Large Eddy Simulations". In: *International Journal of Heat and Fluid Flow* 99 (Feb. 2023). arXiv:2206.11038 [physics], p. 109094. ISSN: 0142727X. DOI: 10.1016/j.ijheatfluidflow.2022.109094. URL: http://arxiv.org/abs/2206.11038 (visited on 02/21/2023).

[123] Marius Kurz et al. "Deep reinforcement learning for computational fluid dynamics on HPC systems". en. In: *Journal of Computational Science* 65 (Nov. 2022), p. 101884. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2022.101884. URL: https://www.sciencedirect.com/science/article/pii/S1877750322002435 (visited on 02/22/2023).

[124] Marius Kurz et al. "Relexi — A scalable open source reinforcement learning framework for high-performance computing". en. In: *Software Impacts* 14 (Dec. 2022), p. 100422. ISSN: 2665-9638. DOI: 10.1016/j.simpa.2022.100422. URL: https://www.sciencedirect.com/science/article/pii/S2665963822001063 (visited on 02/22/2023).

[125] Zhilu Lai et al. "Structural identification with physics-informed neural ordinary differential equations". en. In: *Journal of Sound and Vibration* 508 (Sept. 2021), p. 116196. ISSN: 0022-460X. DOI: 10.1016/j.jsv.2021.116196. URL: https://www.sciencedirect.com/science/article/pii/S0022460X21002686 (visited on 08/18/2021).

[126] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. *Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles*. arXiv:1612.01474 [cs, stat]. Nov. 2017. DOI: 10.48550/arXiv.1612.01474. URL: http://arxiv.org/abs/1612.01474 (visited on 03/01/2023).

[127] Jouko Lampinen and Aki Vehtari. "Bayesian approach for neural networks—review and case studies". en. In: *Neural Networks* 14.3 (Apr. 2001), pp. 257–274. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(00)00098-8. URL: https://www.sciencedirect.com/science/article/pii/S0893608000000988 (visited on 03/01/2023).

[128] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998). Conference Name: Proceedings of the IEEE, pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[129] Kookjin Lee and Kevin T. Carlberg. "Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders". en. In: *Journal of Computational Physics* 404 (Mar. 2020), p. 108973. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.108973. URL: https://www.sciencedirect.com/science/article/pii/S0021999119306783 (visited on 04/13/2021).

[130] Kookjin Lee and Eric J. Parish. "Parameterized Neural Ordinary Differential Equations: Applications to Computational Physics Problems". In: *arXiv:2010.14685 [physics]* (Oct. 2020). arXiv: 2010.14685. URL: http://arxiv.org/abs/2010.14685 (visited on 08/18/2021).

[131] Myoungkyu Lee and Robert D. Moser. "Direct numerical simulation of turbulent channel flow up to". en. In: *Journal of Fluid Mechanics* 774 (July 2015). Publisher: Cambridge University Press, pp. 395–415. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/jfm.2015.268. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/direct-numerical-simulation-of-turbulent-channel-flow-up-to-mathitreittauapprox-5200/3AE84A5A48F83AF294F6CB042AF92DA8 (visited on 02/16/2023).

[132] Pablo Lemos et al. *Rediscovering orbital mechanics with machine learning*. arXiv:2202.02306 [astro-ph]. Feb. 2022. URL: http://arxiv.org/abs/2202.02306 (visited on 12/01/2022).

[133] Mathias Lesjak and Nguyen Anh Khoa Doan. "Chaotic systems learning with hybrid echo state network/proper orthogonal decomposition based model". en. In: *Data-Centric Engineering* 2 (2021). Publisher: Cambridge University Press. ISSN: 2632-6736. DOI: 10.1017/dce.2021.17. URL: https://www.cambridge.org/core/journals/data-centric-engineering/article/chaotic-systems-learning-with-hybrid-echo-state-networkproper-orthogonal-decomposition-based-model/636D2CB1BA6EC278427271CE624F29B2 (visited on 10/27/2021).

[134] Chun-Liang Li et al. *MMD GAN: Towards Deeper Understanding of Moment Matching Network*. arXiv:1705.08584 [cs, stat]. Nov. 2017. URL: http://arxiv.org/abs/1705.08584 (visited on 01/24/2023).

[135] Tianyi Li et al. *Synthetic Lagrangian Turbulence by Generative Diffusion Models*. arXiv:2307.08529 [cond-mat, physics:nlin, physics:physics]. July 2023. DOI: 10.48550/arXiv.2307.08529. URL: http://arxiv.org/abs/2307.08529 (visited on 09/21/2023).

[136] Yi Li et al. "A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence". In: *Journal of Turbulence* 9 (Jan. 2008). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/14685240802376389, N31. DOI: 10.1080/14685240802376389. URL: https://doi.org/10.1080/14685240802376389 (visited on 01/22/2021).

[137] Zijie Li, Kazem Meidani, and Amir Barati Farimani. *Transformer for Partial Differential Equations' Operator Learning*. arXiv:2205.13671 [cs]. Oct. 2022. URL: http://arxiv.org/abs/2205.13671 (visited on 02/01/2023).

[138] Zongyi Li et al. "Fourier Neural Operator for Parametric Partial Differential Equations". In: *arXiv:2010.08895 [cs, math]* (Oct. 2020). arXiv: 2010.08895. URL: http://arxiv.org/abs/2010.08895 (visited on 10/31/2020).

[139] Zongyi Li et al. "Multipole Graph Neural Operator for Parametric Partial Differential Equations". In: *arXiv:2006.09535 [cs, math, stat]* (Oct. 2020). arXiv: 2006.09535. URL: http://arxiv.org/abs/2006.09535 (visited on 11/03/2020).

[140] Zongyi Li et al. "Neural Operator: Graph Kernel Network for Partial Differential Equations". In: *arXiv:2003.03485 [cs, math, stat]* (Mar. 2020). arXiv: 2003.03485. URL: http://arxiv.org/abs/2003.03485 (visited on 11/03/2020).

[141] Zongyi Li et al. "Fourier Neural Operator for Parametric Partial Differential Equations". In: *arXiv:2010.08895 [cs, math]* (May 2021). arXiv: 2010.08895. URL: http://arxiv.org/abs/2010.08895 (visited on 07/06/2021).

[142] Zongyi Li et al. "Markov Neural Operators for Learning Chaotic Systems". In: *arXiv:2106.06898 [cs, math]* (June 2021). arXiv: 2106.06898. URL: http://arxiv.org/abs/2106.06898 (visited on 08/17/2021).

[143] Zongyi Li et al. "Physics-Informed Neural Operator for Learning Partial Differential Equations". en. In: *arXiv:2111.03794 [cs, math]* (Nov. 2021). arXiv: 2111.03794. URL: http://arxiv.org/abs/2111.03794 (visited on 11/15/2021).

[144] Zongyi Li et al. *Learning Dissipative Dynamics in Chaotic Systems*. arXiv:2106.06898 [cs, math]. Sept. 2022. URL: http://arxiv.org/abs/2106.06898 (visited on 04/13/2023).

[145] Chensen Lin et al. "A seamless multiscale operator neural network for inferring bubble dynamics". en. In: *Journal of Fluid Mechanics* 929 (Dec. 2021). Publisher: Cambridge University Press. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/jfm.2021.866. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/seamless-multiscale-operator-neural-network-for-inferring-bubble-dynamics/D516AB0EF954D0FF56AD864DB2618E94 (visited on 11/24/2021).

[146] Tianyang Lin et al. *A Survey of Transformers*. arXiv:2106.04554 [cs]. June 2021. DOI: 10.48550/arXiv.2106.04554. URL: http://arxiv.org/abs/2106.04554 (visited on 02/03/2023).

[147] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. "Reynolds averaged turbulence modelling using deep neural networks with embedded invariance". en. In: *Journal of Fluid Mechanics* 807 (Nov. 2016). Publisher: Cambridge University Press, pp. 155–166. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/jfm.2016.615. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/reynolds-averaged-turbulence-modelling-using-deep-neural-networks-with-embedded-invariance/0B280EEE89C74A7BF651C422F8FBD1EB (visited on 02/17/2023).

[148] Zachary C. Lipton. *The Mythos of Model Interpretability*. arXiv:1606.03490 [cs, stat]. Mar. 2017. DOI: 10.48550/arXiv.1606.03490. URL: http://arxiv.org/abs/1606.03490 (visited on 07/10/2023).

[149] Burigede Liu et al. "A learning-based multiscale method and its application to inelastic impact problems". en. In: *Journal of the Mechanics and Physics of Solids* 158 (Jan. 2022), p. 104668. ISSN: 0022-5096. DOI: 10.1016/j.jmps.2021.104668. URL: https://www.sciencedirect.com/science/article/pii/S0022509621002982 (visited on 06/23/2022).

[150] Liyuan Liu et al. *On the Variance of the Adaptive Learning Rate and Beyond*. arXiv:1908.03265 [cs, stat]. Oct. 2021. URL: http://arxiv.org/abs/1908.03265 (visited on 04/05/2023).

[151] Zhuang Liu et al. *A ConvNet for the 2020s*. arXiv:2201.03545 [cs]. Mar. 2022. URL: http://arxiv.org/abs/2201.03545 (visited on 08/25/2023).

[152] Ilya Loshchilov and Frank Hutter. *SGDR: Stochastic Gradient Descent with Warm Restarts*. arXiv:1608.03983 [cs, math]. May 2017. DOI: 10.48550/arXiv.1608.03983. URL: http://arxiv.org/abs/1608.03983 (visited on 09/15/2023).

[153] Hao Lu, Christopher J. Rutland, and Leslie M. Smith. "A priori tests of one-equation LES modeling of rotating turbulence". In: *Journal of Turbulence* 8 (Jan. 2007). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/14685240701493947, N37. DOI: 10.1080/14685240701493947. URL: https://doi.org/10.1080/14685240701493947 (visited on 07/10/2023).

[154] Lu Lu, Pengzhan Jin, and George Em Karniadakis. "DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators". In: *Nature Machine Intelligence* 3.3 (Mar. 2021). arXiv:1910.03193 [cs, stat], pp. 218–229. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00302-5. URL: http://arxiv.org/abs/1910.03193 (visited on 02/01/2023).

[155] Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. "Deep learning for universal linear embeddings of nonlinear dynamics". en. In: *Nature Communications* 9.1 (Nov. 2018). Number: 1 Publisher: Nature Publishing Group, p. 4950. ISSN: 2041-1723. DOI: 10.1038/s41467-018-07210-0. URL: https://www.nature.com/articles/s41467-018-07210-0 (visited on 02/24/2023).

[156] Stefano Markidis. *The Old and the New: Can Physics-Informed Deep-Learning Replace Traditional Linear Solvers?* arXiv:2103.09655 [physics]. July 2021. URL: http://arxiv.org/abs/2103.09655 (visited on 08/08/2022).

[157] Mitsuaki Matsuo et al. "SUPERVISED CONVOLUTIONAL NETWORKS FOR VOL- UMETRIC DATA ENRICHMENT FROM LIMITED SECTIONAL DATA WITH ADAPTIVE SUPER RESOLUTION". en. In: (2021), p. 5.

[158] R. Maulik et al. “Subgrid modelling for two-dimensional turbulence using neural networks”. en. In: *Journal of Fluid Mechanics* 858 (Jan. 2019). Publisher: Cambridge University Press, pp. 122–144. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/jfm.2018.770. URL: http://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/abs/subgrid-modelling-for-twodimensional-turbulence-using-neural-networks/10EDED1AEAA52C35F3E3A3BB6DC218C1 (visited on 12/08/2020).

[159] Romit Maulik, Bethany Lusch, and Prasanna Balaprakash. “Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders”. In: *Physics of Fluids* 33.3 (Mar. 2021). Publisher: American Institute of Physics, p. 037106. ISSN: 1070-6631. DOI: 10.1063/5.0039986. URL: https://aip.scitation.org/doi/full/10.1063/5.0039986 (visited on 06/08/2021).

[160] Romit Maulik et al. “Time-series learning of latent-space dynamics for reduced-order model closure”. en. In: *Physica D: Nonlinear Phenomena* 405 (Apr. 2020), p. 132368. ISSN: 0167-2789. DOI: 10.1016/j.physd.2020.132368. URL: https://www.sciencedirect.com/science/article/pii/S0167278919305536 (visited on 08/18/2021).

[161] Lars Mescheder et al. *Occupancy Networks: Learning 3D Reconstruction in Function Space*. Number: arXiv:1812.03828 arXiv:1812.03828 [cs]. Apr. 2019. DOI: 10.48550/arXiv.1812.03828. URL: http://arxiv.org/abs/1812.03828 (visited on 06/23/2022).

[162] Tom M. Mitchell. *Machine Learning*. en. McGraw-Hill series in computer science. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[163] Arvind Mohan et al. “Compressed Convolutional LSTM: An Efficient Deep Learning framework to Model High Fidelity 3D Turbulence”. In: *arXiv:1903.00033 [nlin, physics:physics]* (Mar. 2019). arXiv: 1903.00033. URL: http://arxiv.org/abs/1903.00033 (visited on 10/20/2020).

[164] Arvind T. Mohan, Kaushik Nagarajan, and Daniel Livescu. *Learning Stable Galerkin Models of Turbulence with Differentiable Programming*. Number: arXiv:2107.07559 arXiv:2107.07559 [nlin, physics:physics]. July 2021. DOI: 10.48550/arXiv.2107.07559. URL: http://arxiv.org/abs/2107.07559 (visited on 06/23/2022).

[165] Arvind T. Mohan et al. “Embedding Hard Physical Constraints in Neural Network Coarse-Graining of 3D Turbulence”. In: *arXiv:2002.00021 [physics]* (Feb. 2020). arXiv: 2002.00021. URL: http://arxiv.org/abs/2002.00021 (visited on 11/26/2020).

[166] Masaki Morimoto et al. “Convolutional neural networks for fluid flow analysis: toward effective metamodeling and low-dimensionalization”. In: *arXiv:2101.02535 [physics]* (Jan. 2021). arXiv: 2101.02535. URL: http://arxiv.org/abs/2101.02535 (visited on 03/03/2021).

[167] Masaki Morimoto et al. "Assessments of epistemic uncertainty using Gaussian stochastic weight averaging for fluid-flow regression". en. In: *Physica D: Nonlinear Phenomena* (July 2022), p. 133454. ISSN: 0167-2789. DOI: 10.1016/j.physd.2022.133454. URL: https://www.sciencedirect.com/science/article/pii/S0167278922001828 (visited on 07/26/2022).

[168] James Morrill et al. "Neural Rough Differential Equations for Long Time Series". In: *arXiv:2009.08295 [cs, math, stat]* (June 2021). arXiv: 2009.08295. URL: http://arxiv.org/abs/2009.08295 (visited on 08/19/2021).

[169] Saviz Mowlavi and Saleh Nabi. "Optimal control of PDEs using physics-informed neural networks". In: *arXiv:2111.09880 [physics]* (Nov. 2021). arXiv: 2111.09880. URL: http://arxiv.org/abs/2111.09880 (visited on 11/23/2021).

[170] W. James Murdoch et al. "Definitions, methods, and applications in interpretable machine learning". In: *Proceedings of the National Academy of Sciences* 116.44 (Oct. 2019). Publisher: Proceedings of the National Academy of Sciences, pp. 22071–22080. DOI: 10.1073/pnas.1900654116. URL: https://www.pnas.org/doi/10.1073/pnas.1900654116 (visited on 07/10/2023).

[171] Ulrich Mutze. *An asynchronous leapfrog method II*. arXiv:1311.6602 [math]. Apr. 2016. URL: http://arxiv.org/abs/1311.6602 (visited on 02/03/2023).

[172] B. T. Nadiga and D. Livescu. "Instability of the perfect subgrid model in implicit-filtering large eddy simulation of geostrophic turbulence". In: *Physical Review E* 75.4 (Apr. 2007). Publisher: American Physical Society, p. 046303. DOI: 10.1103/PhysRevE.75.046303. URL: https://link.aps.org/doi/10.1103/PhysRevE.75.046303 (visited on 02/21/2023).

[173] Taichi ( ) Nakamura et al. "Convolutional neural network and long short-term memory based reduced order surrogate for minimal turbulent channel flow". In: *Physics of Fluids* 33.2 (Feb. 2021). Publisher: American Institute of Physics, p. 025116. ISSN: 1070-6631. DOI: 10.1063/5.0039845. URL: https://aip.scitation.org/doi/full/10.1063/5.0039845 (visited on 03/03/2021).

[174] Radford M. Neal. *MCMC using Hamiltonian dynamics*. arXiv:1206.1901 [physics, stat]. May 2011. DOI: 10.1201/b10905. URL: http://arxiv.org/abs/1206.1901 (visited on 03/01/2023).

[175] Y. Nesterov. "A method of solving a convex programming problem with convergence rate mathcal {O}(1/kˆ {2})". In: *Sov. Math. Dokl.* Vol. 27. 1986.

[176] Tung Nguyen et al. *ClimaX: A foundation model for weather and climate*. arXiv:2301.10343 [cs]. Jan. 2023. DOI: 10.48550/arXiv.2301.10343. URL: http://arxiv.org/abs/2301.10343 (visited on 02/03/2023).

[177] Guido Novati and Petros Koumoutsakos. *Remember and Forget for Experience Replay*. arXiv:1807.05827 [cs, stat]. May 2019. DOI: 10.48550/arXiv.1807.05827. URL: http://arxiv.org/abs/1807.05827 (visited on 02/22/2023).

[178] Guido Novati, Hugues Lascombes de Laroussilhe, and Petros Koumoutsakos. "Automating turbulence modelling by multi-agent reinforcement learning". en. In: *Nature Machine Intelligence* 3.1 (Jan. 2021). Number: 1 Publisher: Nature Publishing Group, pp. 87–96. ISSN: 2522-5839. DOI: 10.1038/s42256-020-00272-0. URL: https://www.nature.com/articles/s42256-020-00272-0 (visited on 02/12/2021).

[179] Mario Ohlberger and Stephan Rave. *Reduced Basis Methods: Success, Limitations and Future Challenges*. arXiv:1511.02021 [math]. Jan. 2016. DOI: 10.48550/arXiv.1511.02021. URL: http://arxiv.org/abs/1511.02021 (visited on 05/05/2023).

[180] Ali Girayhan Özbay et al. "Poisson CNN: Convolutional neural networks for the solution of the Poisson equation on a Cartesian mesh". en. In: *Data-Centric Engineering* 2 (2021). Publisher: Cambridge University Press, e6. ISSN: 2632-6736. DOI: 10.1017/dce.2021.7. URL: https://www.cambridge.org/core/journals/data-centric-engineering/article/poisson-cnn-convolutional-neural-networks-for-the-solution-of-the-poisson-equation-on-a-cartesian-mesh/8CDFD5C9D5172E51B924E9AA1BA253A1 (visited on 02/28/2023).

[181] Shaowu Pan, Steven L. Brunton, and J. Nathan Kutz. *Neural Implicit Flow: a mesh-agnostic dimensionality reduction paradigm of spatio-temporal data*. Number: arXiv:2204.03216 arXiv:2204.03216 [cs]. Apr. 2022. DOI: 10.48550/arXiv.2204.03216. URL: http://arxiv.org/abs/2204.03216 (visited on 06/23/2022).

[182] Jeong Joon Park et al. *DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation*. arXiv:1901.05103 [cs]. Jan. 2019. DOI: 10.48550/arXiv.1901.05103. URL: http://arxiv.org/abs/1901.05103 (visited on 05/22/2023).

[183] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. arXiv:1211.5063 [cs]. Feb. 2013. DOI: 10.48550/arXiv.1211.5063. URL: http://arxiv.org/abs/1211.5063 (visited on 06/14/2023).

[184] Jaideep Pathak et al. "FourCastNet: A Global Data-driven High-resolution Weather Model using Adaptive Fourier Neural Operators". In: *arXiv:2202.11214 [physics]* (Feb. 2022). arXiv: 2202.11214. URL: http://arxiv.org/abs/2202.11214 (visited on 04/19/2022).

[185] G. S. Patterson and Steven A. Orszag. "Spectral Calculations of Isotropic Turbulence: Efficient Removal of Aliasing Interactions". en. In: *The Physics of Fluids* 14.11 (Nov. 1971), pp. 2538–2541. ISSN: 0031-9171. DOI: 10.1063/1.1693365. URL: https://pubs.aip.org/pfl/article/14/11/

2538/942698/Spectral-Calculations-of-Isotropic-Turbulence (visited on 08/24/2023).

[186] Eric Perlman et al. "Data exploration of turbulence simulations using a database cluster". In: *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. Nov. 2007, pp. 1–11. DOI: 10.1145/1362622.1362654.

[187] Tobias Pfaff et al. *Learning Mesh-Based Simulation with Graph Networks*. arXiv:2010.03409 [cs]. June 2021. DOI: 10.48550/arXiv.2010.03409. URL: http://arxiv.org/abs/2010.03409 (visited on 04/13/2023).

[188] Mary Phuong and Marcus Hutter. *Formal Algorithms for Transformers*. arXiv:2207.09238 [cs]. July 2022. DOI: 10.48550/arXiv.2207.09238. URL: http://arxiv.org/abs/2207.09238 (visited on 08/19/2022).

[189] B. T. Polyak. "Some methods of speeding up the convergence of iteration methods". en. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (Jan. 1964), pp. 1–17. ISSN: 0041-5553. DOI: 10.1016/0041-5553(64)90137-5. URL: https://www.sciencedirect.com/science/article/pii/0041555364901375 (visited on 03/28/2023).

[190] S. B. Pope. "A more general effective-viscosity hypothesis". en. In: *Journal of Fluid Mechanics* 72.2 (Nov. 1975). Publisher: Cambridge University Press, pp. 331–340. ISSN: 1469-7645, 0022-1120. DOI: 10.1017/S0022112075003382. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/more-general-effectiveviscosity-hypothesis/86456F12CB23C8D2D9A2021CBB7FB732 (visited on 02/20/2023).

[191] Stephen B. Pope. *Turbulent Flows*. Cambridge: Cambridge University Press, 2000. ISBN: 978-0-521-59886-6. DOI: 10.1017/CBO9780511840531. URL: https://www.cambridge.org/core/books/turbulent-flows/C58EFF59AF9B81AE6CFAC9ED16486B3A.

[192] Apostolos F. Psaros et al. *Uncertainty Quantification in Scientific Machine Learning: Methods, Metrics, and Comparisons*. arXiv:2201.07766 [cs]. Jan. 2022. URL: http://arxiv.org/abs/2201.07766 (visited on 08/29/2022).

[193] Alessio Quaglino et al. "SNODE: Spectral Discretization of Neural ODEs for System Identification". In: *arXiv:1906.07038 [cs]* (Jan. 2020). arXiv: 1906.07038. URL: http://arxiv.org/abs/1906.07038 (visited on 08/18/2021).

[194] Christopher Rackauckas et al. "Universal Differential Equations for Scientific Machine Learning". In: *arXiv:2001.04385 [cs, math, q-bio, stat]* (Nov. 2021). arXiv: 2001.04385. URL: http://arxiv.org/abs/2001.04385 (visited on 01/17/2022).

[195] Alec Radford et al. "Improving Language Understanding by Generative Pre-Training". en. In: (), p. 12.

[196] Nasim Rahaman et al. "On the Spectral Bias of Neural Networks". en. In: *Proceedings of the 36th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, May 2019, pp. 5301–5310. URL: https://proceedings.mlr.press/v97/rahaman19a.html (visited on 08/28/2023).

[197] Md Ashiqur Rahman, Zachary E. Ross, and Kamyar Azizzadenesheli. *U-NO: U-shaped Neural Operators*. Number: arXiv:2204.11127 arXiv:2204.11127 [cs]. May 2022. DOI: 10.48550/arXiv.2204.11127. URL: http://arxiv.org/abs/2204.11127 (visited on 06/23/2022).

[198] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". en. In: *Journal of Computational Physics* 378 (Feb. 2019), pp. 686–707. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.10.045. URL: http://www.sciencedirect.com/science/article/pii/S0021999118307125 (visited on 10/20/2020).

[199] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. "Hidden Fluid Mechanics: A Navier-Stokes Informed Deep Learning Framework for Assimilating Flow Visualization Data". In: *arXiv:1808.04327 [physics, stat]* (Aug. 2018). arXiv: 1808.04327. URL: http://arxiv.org/abs/1808.04327 (visited on 10/28/2020).

[200] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *arXiv:1710.05941 [cs]* (Oct. 2017). arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941 (visited on 11/27/2020).

[201] Gabrielle Ras et al. *Explainable Deep Learning: A Field Guide for the Uninitiated*. arXiv:2004.14545 [cs, stat]. Sept. 2021. URL: http://arxiv.org/abs/2004.14545 (visited on 07/10/2023).

[202] S. J. Raudys and A. K. Jain. "Small sample size effects in statistical pattern recognition: recommendations for practitioners and open problems". English. In: IEEE Computer Society, Jan. 1990, pp. 417, 418, 419, 420, 421, 422, 423–417, 418, 419, 420, 421, 422, 423. DOI: 10.1109/ICPR.1990.118138. URL: https://www.computer.org/csdl/proceedings-article/icpr/1990/00118138/12OmNyKa64x (visited on 06/30/2023).

[203] Osborne Reynolds. "IV. On the dynamical theory of incompressible viscous fluids and the determination of the criterion — Philosophical Transactions of the Royal Society of London. (A.)" In: (1895). URL: https://royalsocietypublishing.org/doi/10.1098/rsta.1895.0004 (visited on 12/23/2022).

[204] Alexander Rives et al. "Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences". In: *Proceedings of the National Academy of Sciences* 118.15 (Apr. 2021). Publisher: Proceedings of the National Academy of Sciences, e2016239118. DOI: 10.1073/pnas.2016239118. URL: https://www.pnas.org/doi/10.1073/pnas.2016239118 (visited on 02/03/2023).

[205] Ruben Rodriguez-Torrado et al. "Physics-informed attention-based neural network for solving non-linear partial differential equations". In: *arXiv:2105.07898 [cs]* (May 2021). arXiv: 2105.07898. URL: http://arxiv.org/abs/2105.07898 (visited on 08/18/2021).

[206] R S Rogallo and P Moin. "Numerical Simulation of Turbulent Flows". In: *Annual Review of Fluid Mechanics* 16.1 (Jan. 1984). Publisher: Annual Reviews, pp. 99–137. ISSN: 0066-4189. DOI: 10.1146/annurev.fl.16.010184.000531. URL: https://www.annualreviews.org/doi/10.1146/annurev.fl.16.010184.000531 (visited on 05/31/2021).

[207] Francesco Romor, Giovanni Stabile, and Gianluigi Rozza. "Non-linear Manifold Reduced-Order Models with Convolutional Autoencoders and Reduced Over-Collocation Method". en. In: *Journal of Scientific Computing* 94.3 (Mar. 2023), p. 74. ISSN: 0885-7474, 1573-7691. DOI: 10.1007/s10915-023-02128-2. URL: https://link.springer.com/10.1007/s10915-023-02128-2 (visited on 02/15/2023).

[208] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *arXiv:1505.04597 [cs]* (May 2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597 (visited on 11/12/2020).

[209] Clarence W. Rowley and Scott T.M. Dawson. "Model Reduction for Flow Analysis and Control". en. In: *Annual Review of Fluid Mechanics* 49.1 (Jan. 2017), pp. 387–417. ISSN: 0066-4189, 1545-4479. DOI: 10.1146/annurev-fluid-010816-060042. URL: https://www.annualreviews.org/doi/10.1146/annurev-fluid-010816-060042 (visited on 05/05/2023).

[210] Yulia Rubanova, Ricky T. Q. Chen, and David Duvenaud. *Latent ODEs for Irregularly-Sampled Time Series*. arXiv:1907.03907 [cs, stat]. July 2019. DOI: 10.48550/arXiv.1907.03907. URL: http://arxiv.org/abs/1907.03907 (visited on 09/01/2022).

[211] David Ruhe et al. *Geometric Clifford Algebra Networks*. arXiv:2302.06594 [cs]. Feb. 2023. URL: http://arxiv.org/abs/2302.06594 (visited on 02/15/2023).

[212] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". en. In: *Nature* 323.6088 (Oct. 1986). Number: 6088 Publisher: Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 03/27/2023).

[213] Lars Ruthotto and Eldad Haber. "Deep Neural Networks Motivated by Partial Differential Equations". In: *arXiv:1804.04272 [cs, math, stat]* (Dec. 2018). arXiv: 1804.04272. URL: http://arxiv.org/abs/1804.04272 (visited on 02/04/2022).

[214] Tony Saad et al. "Scalable Tools for Generating Synthetic Isotropic Turbulence with Arbitrary Spectra". In: *AIAA Journal* 55.1 (Aug. 2016). Publisher: American Institute of Aeronautics and Astronautics, pp. 327–331. ISSN: 0001-1452. DOI: 10.2514/1.J055230. URL: https://arc.aiaa.org/doi/10.2514/1.J055230 (visited on 02/01/2021).

[215] Cristopher Salvi and Maud Lemercier. "Neural Stochastic Partial Differential Equations". In: *arXiv:2110.10249 [cs]* (Oct. 2021). arXiv: 2110.10249. URL: http://arxiv.org/abs/2110.10249 (visited on 11/03/2021).

[216] Alvaro Sanchez-Gonzalez et al. *Learning to Simulate Complex Physics with Graph Networks*. arXiv:2002.09405 [physics, stat]. Sept. 2020. URL: http://arxiv.org/abs/2002.09405 (visited on 01/11/2023).

[217] Alvaro Sanchez-Gonzalez et al. "LEARNING GENERAL-PURPOSE CNN-BASED SIMULA- TORS FOR ASTROPHYSICAL TURBULENCE". en. In: (2021), p. 12.

[218] Martin Schmelzer, Richard P. Dwight, and Paola Cinnella. "Discovery of Algebraic Reynolds-Stress Models Using Sparse Symbolic Regression". en. In: *Flow, Turbulence and Combustion* 104.2 (Mar. 2020), pp. 579–603. ISSN: 1573-1987. DOI: 10.1007/s10494-019-00089-x. URL: https://doi.org/10.1007/s10494-019-00089-x (visited on 02/17/2023).

[219] Peter J. Schmid. "Dynamic mode decomposition of numerical and experimental data". en. In: *Journal of Fluid Mechanics* 656 (Aug. 2010). Publisher: Cambridge University Press, pp. 5–28. ISSN: 1469-7645, 0022-1120. DOI: 10.1017/S0022112010001217. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/dynamic-mode-decomposition-of-numerical-and-experimental-data/AA4C763B525515AD4521A6CC5E10DBD (visited on 08/02/2022).

[220] Peter J. Schmid. "Dynamic Mode Decomposition and Its Variants". In: *Annual Review of Fluid Mechanics* 54.1 (2022). _eprint: https://doi.org/10.1146/annurev-fluid-030121-015835, pp. 225–254. DOI: 10.1146/annurev-fluid-030121-015835. URL: https://doi.org/10.1146/annurev-fluid-030121-015835 (visited on 07/20/2022).

[221] John Schulman et al. *Proximal Policy Optimization Algorithms*. arXiv:1707.06347 [cs]. Aug. 2017. DOI: 10.48550/arXiv.1707.06347. URL: http://arxiv.org/abs/1707.06347 (visited on 02/22/2023).

[222] Philippe Schwaller et al. "Molecular Transformer: A Model for Uncertainty-Calibrated Chemical Reaction Prediction". In: *ACS Central Science* 5.9 (Sept. 2019). Publisher: American Chemical Society, pp. 1572–1583. ISSN: 2374-7943. DOI: 10.1021/acscentsci.9b00576. URL: https://doi.org/10.1021/acscentsci.9b00576 (visited on 02/03/2023).

[223] Yong Shang, Fei Wang, and Jingbo Sun. *Deep Petrov-Galerkin Method for Solving Partial Differential Equations*. arXiv:2201.12995 [cs, math]. Jan. 2022. URL: http://arxiv.org/abs/2201.12995 (visited on 07/19/2022).

[224] Varun Shankar, Romit Maulik, and Venkatasubramanian Viswanathan. *Differentiable Turbulence*. arXiv:2307.03683 [physics]. July 2023. URL: http://arxiv.org/abs/2307.03683 (visited on 07/10/2023).

[225] Xingjian Shi et al. "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting". In: *arXiv:1506.04214 [cs]* (Sept. 2015). arXiv: 1506.04214. URL: http://arxiv.org/abs/1506.04214 (visited on 12/15/2020).

[226] A. Shnirelman. "On the nonuniqueness of weak solution of the Euler equation". en. In: *Communications on Pure and Applied Mathematics* 50.12 (1997). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291097-0312%28199712%2950%3A12%3C1261%3A%3AAID-CPA3%3E3.0.CO%3B2-6, pp. 1261–1286. ISSN: 1097-0312. DOI: 10.1002/(SICI)1097-0312(199712)50:12<1261::AID-CPA3>3.0.CO;2-6. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-0312%28199712%2950%3A12%3C1261%3A%3AAID-CPA3%3E3.0.CO%3B2-6 (visited on 07/04/2023).

[227] B. Siddani, S. Balachandar, and R. Fang. "Rotational and reflectional equivariant convolutional neural network for data-limited applications: Multiphase flow demonstration". In: *Physics of Fluids* 33.10 (Oct. 2021). Publisher: American Institute of Physics, p. 103323. ISSN: 1070-6631. DOI: 10.1063/5.0066049. URL: https://aip.scitation.org/doi/10.1063/5.0066049 (visited on 10/27/2021).

[228] Laurent Sifre. "Rigid-Motion Scattering For Image Classification". PhD thesis. CMAP - Ecole Polytechnique, 2014.

[229] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv:1409.1556 [cs]. Apr. 2015. URL: http://arxiv.org/abs/1409.1556 (visited on 06/23/2023).

[230] Justin Sirignano and Konstantinos Spiliopoulos. "DGM: A deep learning algorithm for solving partial differential equations". In: *Journal of Computational Physics* 375 (Dec. 2018). arXiv: 1708.07469, pp. 1339–1364. ISSN: 00219991. DOI: 10.1016/j.jcp.2018.08.029. URL: http://arxiv.org/abs/1708.07469 (visited on 02/12/2021).

[231] Alberto Solera-Rico et al. *β-Variational autoencoders and transformers for reduced-order modelling of fluid flows*. arXiv:2304.03571 [physics]. Apr. 2023. URL: http://arxiv.org/abs/2304.03571 (visited on 04/13/2023).

[232] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. *Unsupervised Learning of Video Representations using LSTMs*. arXiv:1502.04681 [cs]. Jan. 2016. DOI: 10.48550/arXiv.1502.04681. URL: http://arxiv.org/abs/1502.04681 (visited on 01/20/2023).

[233] Kimberly Stachenfeld et al. *Learned Coarse Models for Efficient Turbulence Simulation*. arXiv:2112.15275 [physics]. Apr. 2022. DOI: 10.48550/arXiv.2112.15275. URL: http://arxiv.org/abs/2112.15275 (visited on 09/19/2022).

[234] Ben Stevens and Tim Colonius. "Enhancement of shock-capturing methods via machine learning". In: *Theoretical and Computational Fluid Dynamics* 34.4 (Aug. 2020). arXiv:2002.02521 [physics], pp. 483–496. ISSN: 0935-4964, 1432-2250. DOI: 10.1007/s00162-020-00531-1. URL: http://arxiv.org/abs/2002.02521 (visited on 02/28/2023).

[235] Ben Stevens and Tim Colonius. *FiniteNet: A Fully Convolutional LSTM Network Architecture for Time-Dependent Partial Differential Equations*. arXiv:2002.03014 [physics, stat]. Feb. 2020. DOI: 10.48550/arXiv.2002.03014. URL: http://arxiv.org/abs/2002.03014 (visited on 02/28/2023).

[236] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". en. In: *Proceedings of the 30th International Conference on Machine Learning*. ISSN: 1938-7228. PMLR, May 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html (visited on 03/28/2023).

[237] Makoto Takamoto et al. *PDEBENCH: An Extensive Benchmark for Scientific Machine Learning*. arXiv:2210.07182 [physics]. Feb. 2023. DOI: 10.48550/arXiv.2210.07182. URL: http://arxiv.org/abs/2210.07182 (visited on 03/01/2023).

[238] Mingxing Tan and Quoc V. Le. *EfficientNetV2: Smaller Models and Faster Training*. arXiv:2104.00298 [cs] version: 3. June 2021. URL: http://arxiv.org/abs/2104.00298 (visited on 01/18/2023).

[239] Matthew Tancik et al. "Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains". In: *arXiv:2006.10739 [cs]* (June 2020). arXiv: 2006.10739. URL: http://arxiv.org/abs/2006.10739 (visited on 02/11/2021).

[240] Neil C. Thompson et al. *The Computational Limits of Deep Learning*. arXiv:2007.05558 [cs, stat]. July 2022. URL: http://arxiv.org/abs/2007.05558 (visited on 07/10/2023).

[241] Yifeng Tian et al. *Lagrangian Large Eddy Simulations via Physics Informed Machine Learning*. arXiv:2207.04012 [physics]. Aug. 2022. DOI: 10.48550/arXiv.2207.04012. URL: http://arxiv.org/abs/2207.04012 (visited on 02/21/2023).

[242] Alasdair Tran et al. *Factorized Fourier Neural Operators*. Number: arXiv:2111.13802 arXiv:2111.13802 [cs]. Nov. 2021. DOI: 10.48550/arXiv.2111.13802. URL: http://arxiv.org/abs/2111.13802 (visited on 06/16/2022).

[243] Anda Trifan et al. *Intelligent Resolution: Integrating Cryo-EM with AI-driven Multi-resolution Simulations to Observe the SARS-CoV-2 Replication-Transcription Machinery in Action*. en. Pages: 2021.10.09.463779 Section: New Results. Oct. 2021. DOI: 10.1101/2021.10.09.463779. URL: https://www.biorxiv.org/content/10.1101/2021.10.09.463779v1 (visited on 02/01/2023).

[244] Kiwon Um et al. "Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers". In: *arXiv:2007.00016 [physics]* (June 2020). arXiv: 2007.00016. URL: http://arxiv.org/abs/2007.00016 (visited on 10/20/2020).

[245] *Understanding Convolutional Neural Networks: A Complete Guide*. en-US. Jan. 2023. URL: https://learnopencv.com/understanding-convolutional-neural-networks-cnn/ (visited on 06/23/2023).

[246] Ashish Vaswani et al. "Attention Is All You Need". In: *arXiv:1706.03762 [cs]* (Dec. 2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762 (visited on 11/10/2020).

[247] R. Vinuesa et al. "Turbulent boundary layers around wing sections up to Rec=1,000,000". en. In: *International Journal of Heat and Fluid Flow* 72 (Aug. 2018), pp. 86–99. ISSN: 0142-727X. DOI: 10.1016/j.ijheatfluidflow.2018.04.017. URL: https://www.sciencedirect.com/science/article/pii/S0142727X17311426 (visited on 03/01/2023).

[248] Ricardo Vinuesa and Steven L. Brunton. *The Potential of Machine Learning to Enhance Computational Fluid Dynamics*. Tech. rep. arXiv:2110.02085. arXiv:2110.02085 [physics] type: article. arXiv, Oct. 2021. DOI: 10.48550/arXiv.2110.02085. URL: http://arxiv.org/abs/2110.02085 (visited on 05/20/2022).

[249] Ricardo Vinuesa, Philipp Schlatter, and Hassan M. Nagib. "On minimum aspect ratio for duct flow facilities and the role of side walls in generating secondary flows". In: *Journal of Turbulence* 16.6 (June 2015). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/14685248.2014.996716, pp. 588–606. DOI: 10.1080/14685248.2014.996716. URL: https://doi.org/10.1080/14685248.2014.996716 (visited on 03/01/2023).

[250] Rui Wang, Robin Walters, and Rose Yu. "Incorporating Symmetry into Deep Dynamics Models for Improved Generalization". In: *arXiv:2002.03061 [cs, math, stat]* (Mar. 2021). arXiv: 2002.03061. URL: http://arxiv.org/abs/2002.03061 (visited on 04/19/2021).

[251] Rui Wang et al. "Towards Physics-informed Deep Learning for Turbulent Flow Prediction". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 1457–1466. ISBN: 978-1-4503-7998-4. DOI: 10.1145/3394486.3403198. URL: https://doi.org/10.1145/3394486.3403198 (visited on 10/20/2020).

[252] Sifan Wang and Paris Perdikaris. "Long-time integration of parametric evolution equations with physics-informed DeepONets". In: *arXiv:2106.05384 [physics]* (June 2021). arXiv: 2106.05384. URL: http://arxiv.org/abs/2106.05384 (visited on 08/17/2021).

[253] Sifan Wang, Yujun Teng, and Paris Perdikaris. "Understanding and mitigating gradient pathologies in physics-informed neural networks". In: *arXiv:2001.04536 [cs, math, stat]* (Jan. 2020). arXiv: 2001.04536. URL: http://arxiv.org/abs/2001.04536 (visited on 12/01/2020).

[254] Sifan Wang, Hanwen Wang, and Paris Perdikaris. "On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks". In: *arXiv:2012.10047 [cs, stat]* (Dec. 2020). arXiv: 2012.10047. URL: http://arxiv.org/abs/2012.10047 (visited on 04/23/2021).

[255] Sifan Wang, Hanwen Wang, and Paris Perdikaris. "Learning the solution operator of parametric partial differential equations with physics-informed DeepOnets". In: *arXiv:2103.10974 [cs, math, stat]* (Mar. 2021). arXiv: 2103.10974. URL: http://arxiv.org/abs/2103.10974 (visited on 04/23/2021).

[256] Sifan Wang, Xinling Yu, and Paris Perdikaris. "When and why PINNs fail to train: A neural tangent kernel perspective". en. In: *Journal of Computational Physics* 449 (Jan. 2022), p. 110768. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2021.110768. URL: https://www.sciencedirect.com/science/article/pii/S002199912100663X (visited on 01/30/2023).

[257] Yongji Wang et al. "Asymptotic self-similar blow up profile for 3-D Euler via physics-informed neural networks". In: *arXiv:2201.06780 [physics]* (Mar. 2022). arXiv: 2201.06780. URL: http://arxiv.org/abs/2201.06780 (visited on 04/20/2022).

[258] Zhengwei Wang, Qi She, and Tomás E. Ward. "Generative Adversarial Networks in Computer Vision: A Survey and Taxonomy". en. In: *ACM Computing Surveys* 54.2 (Mar. 2022), pp. 1–38. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3439723. URL: https://dl.acm.org/doi/10.1145/3439723 (visited on 01/24/2023).

[259] J. Weatheritt and R. D. Sandberg. "The development of algebraic stress models using a novel evolutionary algorithm". en. In: *International Journal of Heat and Fluid Flow* 68 (Dec. 2017), pp. 298–318. ISSN: 0142-727X. DOI: 10.1016/j.ijheatfluidflow.2017.09.017. URL: https://www.sciencedirect.com/science/article/pii/S0142727X17303223 (visited on 02/20/2023).

[260] Max Welling and Yee Whye Teh. "Bayesian learning via stochastic gradient langevin dynamics". In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Madison, WI, USA: Omnipress, June 2011, pp. 681–688. ISBN: 978-1-4503-0619-5. (Visited on 03/01/2023).

[261] Steffen Wiewel et al. *Latent Space Subdivision: Stable and Controllable Time Predictions for Fluid Flow*. arXiv:2003.08723 [cs, stat]. Mar. 2020. DOI: 10.48550/arXiv.2003.08723. URL: http://arxiv.org/abs/2003.08723 (visited on 04/12/2023).

[262] Mark D. Wilkinson et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data* 3 (Mar. 2016), p. 160018. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4792175/ (visited on 03/01/2023).

[263] Sanghyun Woo et al. *ConvNeXt V2: Co-designing and Scaling ConvNets with Masked Autoencoders*. arXiv:2301.00808 [cs]. Jan. 2023. URL: http://arxiv.org/abs/2301.00808 (visited on 08/25/2023).

[264] BigScience Workshop et al. *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*. arXiv:2211.05100 [cs]. Mar. 2023. DOI: 10.48550/arXiv.2211.05100. URL: http://arxiv.org/abs/2211.05100 (visited on 05/23/2023).

[265] 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 (July 2018). arXiv:1801.02762 [physics], p. 074602. ISSN: 2469-990X. DOI: 10.1103/PhysRevFluids.3.074602. URL: http://arxiv.org/abs/1801.02762 (visited on 02/21/2023).

[266] Jin-Long Wu et al. "Enforcing statistical constraints in generative adversarial networks for modeling chaotic dynamical systems". en. In: *Journal of Computational Physics* 406 (Apr. 2020), p. 109209. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.109209. URL: https://www.sciencedirect.com/science/article/pii/S0021999119309143 (visited on 03/04/2021).

[267] Xiaohua Wu. "Inflow Turbulence Generation Methods". In: *Annual Review of Fluid Mechanics* 49.1 (2017). _eprint: https://doi.org/10.1146/annurev-fluid-010816-060322, pp. 23–49. DOI: 10.1146/annurev-fluid-010816-060322. URL: https://doi.org/10.1146/annurev-fluid-010816-060322 (visited on 10/27/2020).

[268] Yuxin Wu and Kaiming He. *Group Normalization*. arXiv:1803.08494 [cs]. June 2018. URL: http://arxiv.org/abs/1803.08494 (visited on 04/05/2023).

[269] Yuxin Wu and Justin Johnson. *Rethinking "Batch" in BatchNorm*. arXiv:2105.07576 [cs]. May 2021. URL: http://arxiv.org/abs/2105.07576 (visited on 08/25/2023).

[270] Jiayang Xu and Karthik Duraisamy. "Multi-level Convolutional Autoencoder Networks for Parametric Prediction of Spatio-temporal Dynamics". In: *Computer Methods in Applied Mechanics and Engineering* 372 (Dec. 2020). arXiv: 1912.11114, p. 113379. ISSN: 00457825. DOI: 10.1016/j.

cma.2020.113379. URL: http://arxiv.org/abs/1912.11114 (visited on 07/23/2021).

[271] Ling Yang et al. *Diffusion Models: A Comprehensive Survey of Methods and Applications*. arXiv:2209.00796 [cs]. Oct. 2022. DOI: 10.48550/arXiv.2209.00796. URL: http://arxiv.org/abs/2209.00796 (visited on 11/10/2022).

[272] Liu Yang, Dongkun Zhang, and George Em Karniadakis. "Physics-Informed Generative Adversarial Networks for Stochastic Differential Equations". In: *arXiv:1811.02033 [cs, math, stat]* (Nov. 2018). arXiv: 1811.02033. URL: http://arxiv.org/abs/1811.02033 (visited on 05/11/2021).

[273] Yibo Yang and Paris Perdikaris. "Adversarial uncertainty quantification in physics-informed neural networks". en. In: *Journal of Computational Physics* 394 (Oct. 2019), pp. 136–152. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.05.027. URL: http://www.sciencedirect.com/science/article/pii/S0021999119303584 (visited on 10/20/2020).

[274] Zeng Yang, Jin-Long Wu, and Heng Xiao. *Enforcing Deterministic Constraints on Generative Adversarial Networks for Emulating Physical Systems*. arXiv:1911.06671 [physics, stat]. Nov. 2020. URL: http://arxiv.org/abs/1911.06671 (visited on 08/09/2023).

[275] Rongtian Ye, Fangyu Liu, and Liqiang Zhang. *3D Depthwise Convolution: Reducing Model Parameters in 3D Vision Tasks*. arXiv:1808.01556 [cs]. Aug. 2018. URL: http://arxiv.org/abs/1808.01556 (visited on 08/24/2023).

[276] Yuan Yin et al. *Continuous PDE Dynamics Forecasting with Implicit Neural Representations*. arXiv:2209.14855 [cs, stat]. Feb. 2023. DOI: 10.48550/arXiv.2209.14855. URL: http://arxiv.org/abs/2209.14855 (visited on 05/22/2023).

[277] Mustafa Z. Yousif, Linqi Yu, and HeeChang Lim. "Physics-guided deep learning for generating turbulent inflow conditions". en. In: *Journal of Fluid Mechanics* 936 (Apr. 2022). Publisher: Cambridge University Press, A21. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/jfm.2022.61. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/physicsguided-deep-learning-for-generating-turbulent-inflow-conditions/5AC04D36A013FD8661CB8381860ADC8C (visited on 10/13/2022).

[278] Mustafa Z. Yousif et al. "A transformer-based synthetic-inflow generator for spatially developing turbulent boundary layers". en. In: *Journal of Fluid Mechanics* 957 (Feb. 2023). Publisher: Cambridge University Press, A6. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/jfm.2022.1088. URL: https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/transformerbased-syntheticinflow-generator-for-spatially-developing-turbulent-boundary-layers/E58DB7B8F3C0F8FB223C6488F9CBB34D (visited on 02/24/2023).

[279] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. arXiv:1212.5701 [cs]. Dec. 2012. URL: http://arxiv.org/abs/1212.5701 (visited on 04/05/2023).

[280] Bowen Zhang et al. *StyleSwin: Transformer-based GAN for High-resolution Image Generation*. en. arXiv:2112.10762 [cs]. July 2022. URL: http://arxiv.org/abs/2112.10762 (visited on 08/10/2022).

[281] Jiawei Zhao et al. *Incremental Fourier Neural Operator*. arXiv:2211.15188 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2211.15188. URL: http://arxiv.org/abs/2211.15188 (visited on 12/01/2022).

[282] Hongkai Zheng et al. *Fast Sampling of Diffusion Models via Operator Learning*. arXiv:2211.13449 [cs]. Nov. 2022. URL: http://arxiv.org/abs/2211.13449 (visited on 12/01/2022).

[283] Yinhao Zhu et al. "Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data". en. In: *Journal of Computational Physics* 394 (Oct. 2019), pp. 56–81. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.05.024. URL: http://www.sciencedirect.com/science/article/pii/S0021999119303559 (visited on 10/20/2020).

[284] Jiawei Zhuang et al. "Learned discretizations for passive scalar advection in a 2-D turbulent flow". In: *arXiv:2004.05477 [cond-mat, physics:physics]* (Nov. 2020). arXiv: 2004.05477. URL: http://arxiv.org/abs/2004.05477 (visited on 03/02/2021).

[285] Juntang Zhuang et al. *AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients*. arXiv:2010.07468 [cs, stat]. Dec. 2020. URL: http://arxiv.org/abs/2010.07468 (visited on 04/05/2023).

[286] Zongren Zou and George Em Karniadakis. *L-HYDRA: Multi-Head Physics-Informed Neural Networks*. arXiv:2301.02152 [physics]. Jan. 2023. DOI: 10.48550/arXiv.2301.02152. URL: http://arxiv.org/abs/2301.02152 (visited on 01/11/2023).

[287] Kirill Zubov et al. *NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations*. arXiv:2107.09443 [cs]. July 2021. DOI: 10.48550/arXiv.2107.09443. URL: http://arxiv.org/abs/2107.09443 (visited on 02/28/2023).