Convolutional neural networks for steady flow prediction around 2D obstacles

Junfeng Chen

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à MINES Paris

Réseaux de neurones convolutifs pour la prédiction de flux autour d'obstacles

Soutenue par
Junfeng CHEN
Le 27 janvier 2022

Ecole doctorale n° 364
Sciences Fondamentales et Appliquées

Spécialité
Mathématiques Numériques, Calcul Intensif et Données

Composition du jury :

Anne, JOHANNET Professeur, IMT Alès	*Rapporteur*
Jean-Luc, HARION Professeur, IMT Lille Douai	*Rapporteur*
Emmanuelle, ABISSET-CHAVANNE Professeur, Arts et Métiers ParisTech	*Présidente*
Jonathan, VIQUERAT Ingénieur de recherche, Mines Paris PSL	*Maître de thèse*
Frédéric, HEYMES Maître assistant, IMT Alès	*Co-directeur de thèse*
Elie, HACHEM Professeur, Mines Paris PSL	*Directeur de thèse*

MINES ParisTech | PSL

Acknowledgements

During the past three years, I am lucky to meet a lot of friendly and kind people while preparing my thesis. I would like express my gratitude at this occasion.

First, I would like to deeply thank Jonathan Viquerat and Elie Hachem for their care and patience. With their accompany, it is a pleasure to put myself in the ocean of science and to explore unknown domains. I must give my gratitude to Elie for his unique role in structuring the thesis, and for his dedication to show me the global view of the thesis and help organize the projects. I cannot be more grateful to Jonathan for the countless inspiring and rigorous discussions that we had, as well as for his unreserved guide to form me as an independent researcher.

Next, I would like to thank Anne Johannet and Jean-Luc Harion to be the referees of my thesis. Their comments and constructive suggestions for the manuscript broadens my view on the structure and quality of the thesis. I would like also to thank Emmanuelle Abisset-Chavanne and Frédéric Heymes for their efforts in the jury.

A special thanks to Hassan Ghraieb as we shared three years in the same office, and I will never forget his continuous kindness and trust on me. I wish the best in his defense as well as the work and life in the future. My sincere thank goes to other colleagues in the CFL team: Aakash Patil, Ali-Malek Boubaya, Ramy Nemer, Joe Khalil, Sacha el Aouad, Ghaniyya Medghoul, Lucas Sardo, Franck Pigeonneau, Jérémie Bec and Aurélien Larcher. Their welcome helped me settle down peacefully in the center, and their kindness remains an important memory during the past three years.

My thank continues to Chengdan Xue, Feng Gao, Shaojie Zhang, Han Wang, Zhongfeng Xu, Shengfei Wang. The friendship among us makes me feel home in Sophia Antipolis. I must also thank Qinkai Chen and Qihui Yu, as the time we spent in Paris and other places of France was fascinating and unforgettable, it is a particular treasure to have their friendship in France, especially in the period of Covid-19.

Finally I cannot be more thankful to my parents for their love and continuous support. Our connection is always strong and never affected by the distance between Europe and China. I cannot wait the end of Covid-19 and express my emotion to them face to face.

Contents

List of Figures

List of Tables

1 Introduction

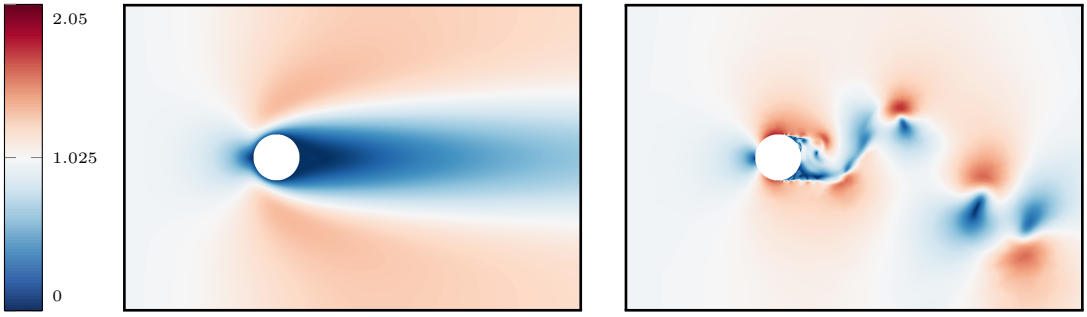
1.1 Fluid dynamics

Fluid dynamics focus on the evolution of density, velocity, pressure and energy of fluids in general, through the conservation laws of mass, forces and energy. These physical principles are condensed in the well-known Navier-Stokes equations, formulated in the early 19-th century by Claude-Louis Navier [Nav23] and George Gabriel Stokes [S+51]. Although the theoretical aspects of the Navier–Stokes (NS) equations have been studied by thousands of mathematicians and physicists, the existence and unicity of its solution have not been proven to this day. In simple configurations, analytical solutions can be obtained, such as for the steady incompressible flow in an infinitely long pipe (*i.e.* Poiseuille flow [Batoo, Poi40]), which displays a paraboloid axial velocity profile. However, the limited number of analytical solutions is not sufficient to fully understand the mechanisms of flow motion.

In 1883, Osborne Reynolds performed experiments consisting in studying the water movement in a long glass pipe with small diameter. Osborne Reynolds's experiments [Rey83] showed that at high velocities, the dyed water flowing through the pipe can lose the paraboloid profile, with the dyed layer broken apart and diffused to arbitrary directions. Reynolds' experiments led to the definition of one of the most important scalar in fluid mechanics, the Reynolds number:

$$Re = \frac{\rho U L}{\mu},$$

with ρ the fluid's density, U the characteristic velocity, L the characteristic length and μ the fluid's dynamic viscosity. The Reynolds number measures the ratio of inertial forces to viscous forces in the flow. At low Reynolds numbers, the flow tends to be laminar, which means that different parts of the fluid move in parallel layers, as shown in figure 1.1a. At high Reynolds numbers (usually $> 10^3$), the flow becomes turbulent, meaning that the fluid continuously experiences irregular fluctuations. Eddies of different sizes are generated and degenerated in turbulent regions of the flow (see figure 1.1b), along with active mass mixture and energy exchange. At large scales, the fluid motion mostly depends on the boundary conditions, while at small scales, the behavior of eddies follows some unified physical pattern independent of geometric factors [Popoo], making turbulent flow a complex multiscale structure. A well-established physical model is the energy cascade theory by Lewis Fry Richardson [RL07], proposed in 1922. In this



(a) Laminar flow **(b)** Turbulent flow

Figure 1.1 | Laminar and turbulent velocity magnitude (m/s) around a 2-D cylinder. At $Re = 15$, the different layers of fluid passing around the cylinder remain parallel. At $Re = 1.5 \times 10^5$, the flow is turbulent, and eddies with different sizes are present behind the cylinder.

model, a simplistic but clear view of the energy budget of turbulence is presented: the energy at larger eddies is transferred to smaller eddies at the occasion of eddy division, until the eddies are so small that the viscous dissipation overpasses the energy transfer. Here, an eddy refers to a swirling structure in turbulent flow, with its moving direction different from the main flow. In 1941, Andreï Nikolaïevitch Kolmogorov quantitatively obtained the smallest scale at which the cascade ceases using dimensional analysis [Kol41]. In incompressible homogeneous isotropic turbulence, there exists an inertial range where the energy transfer obeys a unified law, regardless of other physical properties of the fluids.

Turbulent flows can be encountered at many occasions in the real world. For high-speed airplanes and vehicles, the flow within a thin layer in contact with the solid surface is known to be turbulent. Formally called the turbulent boundary layer [Whi11], it has crucial impact on the stability of the moving body. Turbulence also occurs when two clouds of air with different density, velocity or temperature meet [Stu88]: the frontal surface triggers turbulent eddies, along with intense mass and energy transport. In the atmosphere, the clouds and inhomogeneous earth surface conditions cause uneven sunshine flux, which can generate thermal convective turbulence from the earth surface to cloud tops. Understanding and predicting the events occurring in turbulent flows can hence be beneficial in many situations. However, the complexity and chaos inherent to turbulence make it almost impossible to analyse dynamics from a purely theoretical approach. Indeed, turbulence is often quoted as *"the most important unsolved problem in classical physics"* [EF11].

1.2 Computational fluid dynamics

Experimentation has represented the main approach to study fluid dynamics since Reynolds' tube experiments in 1883. Today, modern captors and imaging technologies make it feasible to recover and precisely measure flow fields in many physical scenarios. Yet, experimentation remains an expensive and time-consuming approach. The preparation of an experiment usually requires much care and time, which is a barrier to the reproduction of experimental results. Experimentation is also not acceptable in inverse designs, because the flow fields are updated every time the control parameter is modified. To this end, computer-based simulations are nowadays widely adopted for both scientific research and industrial practice. As a branch of fluid mechanics, computational fluid dynamics (CFD) exploit numerical methods to obtain physical fields from the resolution of approximated NS equations (in the present thesis, focus is made on velocity and pressure in the NS equations, and other quantities such as temperature or concentration are omitted). Compared to experimental methods, initial and boundary conditions, as

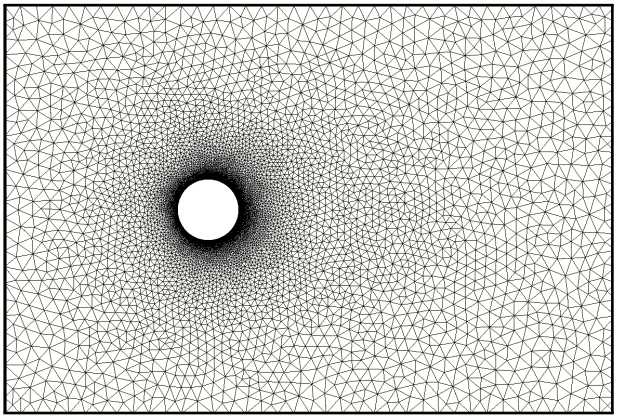


Figure 1.2 | A triangular mesh around a 2-D cylinder used for the finite element method. The elements near the solid surface are refined in order to accurately capture the geometry characteristics and boundary layer dynamics. The elements in the outer region are relatively big to reduce the computation cost.

well as geometric characteristics, can be easily controlled in numerical environments. More, numerical experiments can be repeated without as heavy preparation as for experiments, and multiple instances of a given configuration can be run simultaneously in parallel. For these reasons, CFD has now become an essential tool to study fluid dynamics, alongside experimental setups.

A large range of numerical methods are available to numerically solve NS equations. Among these, the finite difference method (FDM) and the finite element method (FEM) are two widely-spread approaches. In FDM, the discrete NS equations are solved on a domain discretized using uniform rectangular cells. Although the simplicity of implementation is a key advantage of FDM, the method presents major drawbacks when dealing with complex geometries. Indeed, when the physical domain includes curved boundaries, the latter has to be approximated using the axis-aligned spatial discretization, which is a major source of inaccuracy. In some cases, FDM can also fail to capture local features due to the isotropic discretization step size. Although it is possible to reduce the cell size to handle the boundary and inhomogeneity issues, it significantly increases the computational cost of numerical resolution. Therefore, FDM has limited performance when the computation domain is not piece-wise rectangular. In FEM, the space is discretized into small and simple subdomains called elements, whose reunion forms a mesh, as shown in figure 1.2. A set of polynomial basis functions is then defined on the mesh (usually on the nodes), after what the discrete fields can be reconstructed by linear combination of the basis functions. The coefficients of this linear combination are obtained from the numerical resolution of a large sparse linear system, obtained from a so-called variational formulation. Given the intrinsic flexibility of FEM, many type of elements can be used to discretize the computational domain, and often triangles (resp. tetrahedra) are used in 2-D (resp. 3-D) problems, allowing it to naturally account for complex geometries. More, the element size is not expected to be uniform across the computational domain: one can apply local mesh refinement by decreasing the element size to improve the accuracy in the region of interest [BR78]. FEM was first proposed for structural analysis in 1943 by Courant [Cou43], and introduced into fluid dynamics in the 1970s. Today, FEM is a vastly spread method, and has been applied to countless physical problems [Pir89].

The development of CFD promotes the study of turbulent flows through turbulence modelling. To accurately resolve all the scales of turbulent velocity and pressure fields in a CFD computation, the required mesh size must be inferior to the smallest eddy scale, this approach being called direct numerical simulation (DNS). Since the smallest scale decrease exponentially when the Reynolds number rises [Popoo], DNS is barely attainable even for turbulent flows at moderate Reynolds numbers. To this end, various methods have been developed under the umbrella of turbulence modelling. Distinguished by their accur-

acy and their complexity, two of these methods are presented in this section. The first one resolves the Reynolds-averaged Navier–Stokes (RANS) equations, a time-averaged version of the original NS equations. In this framework, physical quantities are decomposed in the sum of a mean quantity and a zero-average fluctuation. Applied to the NS equations, this decomposition leads to a set of equations on the mean flow with an additional fluctuation term, called the Reynolds stress tensor, originating in the non-linear convection term of NS. A RANS model provides a closure [Bou77] to this term, so that the mean flow can be resolved. Besides RANS, large eddy simulation (LES) represents another modelling strategy for turbulent flows. In LES, a low-pass filter is applied to the NS equations, in such a way that only the large scale motion is to be resolved, rather than the full turbulent scales. As in RANS, the filtered nonlinear convection term leads to a non-closed term due to the small scale motion. Since the objective of LES models is to relate this term to large-scale flow, they are also called sub-grid scale models [Sma63, Dea70]. LES is generally more expensive than RANS, but also resolves turbulent features more accurately. Yet, due to its lower cost, RANS is still preferred to LES for industrial applications. According to a 2014 report from NASA [SKA+14], LES will still remain out of reach for large-scale engineering simulations until the year 2030.

1.3 Machine learning

As defined by Azencott [Aze19], machine learning (ML) is about the application of optimization methods to statistical modelling. In practice, ML methods aim at inferring existing relations between quantities from large set of data samples, or exploring the underlying structure of the data set. The simplest ML algorithm is the least squares linear regression, which is an efficient tool used to discover linear constitutive laws in mechanics.

Beyond linear methods, multiple nonlinear algorithms have been proposed for complex data modelization, including generalized linear methods, random forests [Bre01], or neural networks [MP43], each algorithm representing a family of parameterized functions. In the age of big data, these algorithms have been implemented in many different contexts, and their impact on our daily life is rapidly increasing. Some relevant applications include custom advertizing based on web cookies, fraud detection in banks, dynamic pricing in flight tickets, etc. Designing efficient models for such real-life scenarios require a significant amount of feature engineering, *i.e.* algorithmic and data tuning specific to each field and application.

Among all the nonlinear algorithms, neural networks (NN) have been among the most represented in the recent years. A NN refers to a composition of hidden layers of artificial neurons, with each layer being a parametrized nonlinear transformation of its inputs. Its construction makes it capable of approximizing extremely nonlinear functional relations. Of particular interest is the category of deep neural networks (DNNs), which can automatically learn hierarchical features from the raw data without manual feature engineering. A DNN can have many hidden layers, thus large number of trainable parameters, and its training is usually harder than other ML algorithms due to the famous vanishing/exploding gradients issue [PMB13]. Since 2006, DNNs have gained more and more popularity thanks to new optimization technics. Hinton *et al.* [HS06] proposed a smart pre-training method to initialize the trainable parameters; the stochastic gradient descent algorithms [Bot10] handle efficiently non-convex optimization and can lead to globally optimal models. In the meantime, recent toolkits like TensorFlow [AAB+15] and PyTorch [PGC+17] have made it fairly easy to train large DNNs on GPU cards, thus remarkably reducing training times.

The convolutional neural network (CNN) is a particular DNN architecture with embedded convolution operations on 2-D and 3-D tensor inputs. Although first proposed in 1989, CNNs started to be widely

applied to image-like data since the proposition of AlexNet [KSH12] in 2012: with 60 million trainable parameters, AlexNet was trained against a data set with 1.2 million high-resolution images to perform image classification, and reached a test error equals to 17.0%, much better the state-of-the-art which was equal to 28.2%. Training large CNNs against massive high-quality images has now become a standard approach to obtain performant and robust classification models. This technology has already been commercialized in various scenarios like automated driving, facial smartphone unlocking, biomedical image segmentation, among others. The success of CNNs is attracting more and more researchers to develop new concepts and explore potential applications. A recent extension of CNNs is the graph convolutional neural networks (GCNNs), which aim at extending CNNs applications from images to graph-based data. Although its non-Euclidean characteristic makes the convolution on graphs more difficult than on images, graphs are a common data structure in many domains, including social networks, traffics, protein molecules and so forth. In the CFD domain, while the cartesian grids used in FDM can be regarded as image-like data, the meshes used in FEM computations have graph-like structures. In the recent years, both CNNs and GCNNs have seen an increasing interest among CFD researchers.

1.4 Machine learning for computational fluid dynamics

Since the turbulence models are usually obtained from a mixture of physical knowledge and empirical evidence, the concept of data fitting is not new in the CFD domain. Indeed, model calibration is about searching the proper model parameters to enable the model to represent available high-fidelity data. Otherwise, it is possible to regard a model parameter (or other uncertain quantities) as a random variable, whose exact distribution is inversely inferred in order to maximize the likelihood of the observations [XC19].

Combining the tools from machine learning and traditional model calibration [DIX19], it is possible to improve existing turbulence models by exploiting available data from experiments or DNS [PD16, LKT16, WWX17]. In this context, a spatially varying discrepancy term can be introduced into the turbulence model and calibrated through data assimilation. Once this step is completed, a ML model is trained to predict the discrepancy, with local velocity and pressure as inputs. Doing so, the discrepancy model can hence provide local corrections to the original turbulence model. The ML models mentioned here are point-wise: at any location in the computational domain, features based on velocity and pressure are selected and designed as the inputs, after what the model gives a prediction of the discrepancy.

The success of ML-augmented turbulence models also opened the door for other ML algorithms to spread in the CFD community. As a deep learning technique, convolutional neural networks can approximate highly nonlinear functional relation, and at the same time free users from manual feature engineering as in point-wise ML models. In the last few years, investigations were conducted on the use of CNNs for the prediction of aerodynamic force around obstacles [ZSM18, VH20] or full-field steady flow [GLI16, TWPH20, FFT19]. While traditional CNNs require a data set interpolated on cartesian grids, GCNNs can be applied directly on triangular meshes, as shown in [PFSGB21] and [XSSZ21]. Applications considering unsteady flow predictions were also conducted: in [LY19], the authors exploit the flow snapshots at previous time steps to predict future snapshots. In these applications, DNN-based surrogate models are trained to fit the available data set. Hence, compared to solving PDEs with traditional iterative solvers, trained CNNs provide explicit functional relations that only require one forward calculation to predict the output. Conversely, it is important to underline that the applicability range of a trained model is extremely limited compared to regular approaches.

Being essentially a data-driven approach, DNN-based surrogate models for fluid dynamics are like extremely complex black boxes, whose consistency with physical laws is questionable. An emerging

research axe is that of physics-informed neural networks (PINNs). Conversely to regular neural networks approaches, PINNs aim at incorporating physical knowledge into NN-based models. This objective can be reached through incorporating the PDE residuals into the loss used to train the model. Hence, the approach retained to train PINNs does not require to minimize a data-fitting error. Raissi *et al.* [RPK19] demonstrated the interest of PINNs on several textbook problems involving partial differential equations. To do so, they used a fully connected network architecture with time and coordinate information as the inputs, making it easy to obtain the partial derivatives in an automatic differentiation framework. In [WWK21], Wandel *et al.* considered a physics-informed CNN for 3-D fluid simulation. In the latter publication, a CNN architecture is employed to map the flow field at a given time step to its counterpart at the subsequent time step. Being the state-of-art of incorporating physics into NN-based models, PINNs are being applied to more and more fluid problems, including incompressible laminar flow prediction [RSL20], turbulence model development [IDC21], flow and transport in porous media [GT21] and so on.

1.5 Outline

The remaining of this manuscript is structured in the following way:

- Chapter 2 presents the basic notions of machine learning and convolutional neural networks, split into two sections respectively dedicated to network architectures and optimization methods for neural networks. The machine learning tools used in the manuscript are all summed up in this chapter.
- Chapter 3 is dedicated to computational fluid dynamics. The generation of laminar and turbulent flow data sets from CFD tools are covered. The focus is put on the discretization and computational cost, as a reference to the data-driven based surrogate models.
- In Chapter 4, the laminar and turbulent data sets obtained in Chapter 3 are projected to cartesian grids and used to train CNN-based surrogate models. The accuracy and computational cost are reported, with a hyper-parameter study to show the impact of the network's architecture. In the end, differences between laminar and turbulent flow predictions are pointed out, giving more reflections on the relation between the network's modeling capacity and its architecture.
- Chapter 5 presents a novel CNN architecture to make the surrogate model more transparent than a pure blackbox, which is the case of U-net methods used in chapter 2. When doing prediction, a trained CNN with the proposed architecture can detect whether it is doing extrapolation, or give a quantified uncertainty of the predicted velocity and pressure, providing the end-user a metric to evaluate the quality of the predictions.
- In Chapter 6, the laminar data set is projected to body-fitted triangular meshes, and used to train GCNN-based surrogate models. The advantage and disadvantage of using the graph-based method is discussed in detail, through a quantitative comparison with the CNN methods from Chapter 2. Finally as an open topic, an unsupervised physics-informed approach is combined with the proposed GCNN architecture, which provides a steady solver to the incompressible Navier-Stokes equations.

2

Convolutional neural networks

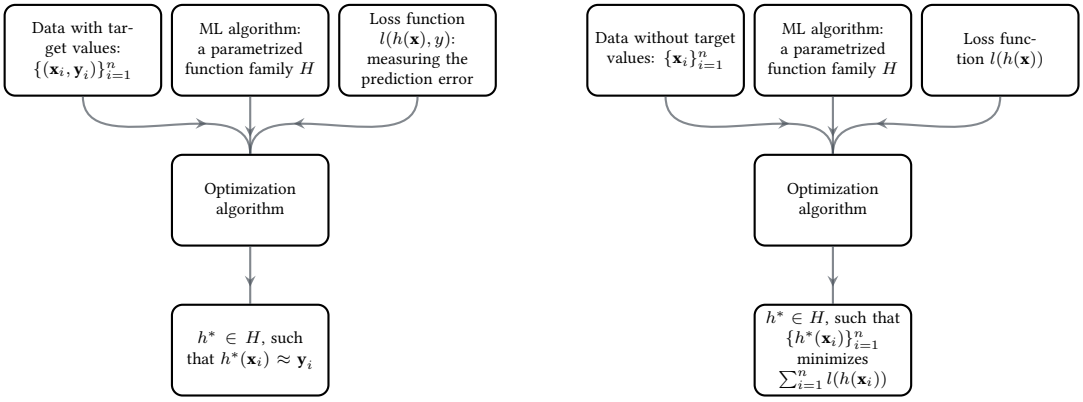
Résumé

Ce chapitre présente les notions et méthodes de base nécessaires à l'utilisation des réseaux de neurones convolutifs (CNN). La première section se concentre sur les concepts généraux de ML, après quoi des généralités sur les neurones artificiels et les réseaux de neurones sont résumées. Étant les outils centraux des chapitres suivants, trois sections distinctes sont réservées aux réseaux de neurones convolutifs, aux autoencodeurs et aux réseaux de neurones convolutifs sur graphes. Ensuite, les aspects de l'entrînement de CNN sont présentés, en mettant l'accent sur l'adaptation aux CNN de l'optimisation basée sur les gradients.

2.1 Machine learning generalities

As defined by Azencott [Aze19], ML is about the application of optimization methods to statistical modelling. In practice, ML methods aim at inferring existing relations between quantities from large set of data samples, or exploring the underlying structure of the data set. The simplest ML algorithm is the least squares linear regression, in which a parameterized linear function is calibrated in order to minimize the error of the model predictions against an observed data set. It is necessary to note that, with the fitting error on the observed data being small, one does not always expect the same error level on non-observed data. In case of overfitting, the prediction can be totally wrong if the predicted case lies outside the boundary of the observations.

Besides the least squares linear regression, more complex machine learning algorithms have been developed to solve advanced problems, among which are generalized linear methods [MN19], tree-based methods [Qui86, Bre01] and neural networks [MP43]. Each algorithm provides a family of functions to modelize the observed data, with the family of neural networks having the largest modeling capacity.



(a) Supervised learning **(b)** Unsupervised learning

Figure 2.1 | General frameworks for supervised and unsupervised machine learning. Supervised learning is concerned with mapping input data with target values, the loss function measuring the prediction error. Unsupervised learning is concerned with data without target values, with a case-by-case different loss function. For example the variance loss when projecting the data into a low-dimensional space is used for principle component analysis.

Indeed, the Universal Approximation Theorem [Cyb89] proves that NNs are able to approximate any continuous function defined on a bounded set. In practice, it means that NNs are able to modelize extremely complex functional relations between the input and output data. It is indeed usually hard to manually quantify how a certain input variable affects the output, for example the impact of temperature on a city's daily electricity consumption. Therefore ML algorithms with large modeling capacity are gaining more and more popularity in many real-life problems.

Besides algorithms, the data is also a crucial part of machine learning. The ML algorithms were originially used to modelize data in the format of scalar and vector. However, other data structures like matrix and graph are also common, which should be flatten into vectors to be compatible with the mentioned ML algorithms. With the invention of convolutional neural networks (CNN) in 1988 [Fuk88], it becomes possible to extend machine learning technics to image-like data. Later, the graph convolutional neural networks (GCNN) can directly modelize data on graphs without destroying the topology of graphs. This thesis considers the application of CNNs and GCNNs in computational fluid dynamics, being more specific, in velocity and pressure prediction on cartesian grids and triangular meshes. These two algorithms will be presented in full details in section 2.2.

Machine learning tasks can be roughly divided into three categories: supervised learning, unsupervised learning and reinforcement learning. Learning is called "supervised" when the data set samples are associated with target values. The objective of supervised learning is therefore to perform accurate predictions of an unknown target value when provided new input data. A supervised model is hence calibrated through minimizing the predicition error over the available data set. Unsupervised learning refers to the tasks where the observed data samples have no target values. Hence, the cost function to be optimized is usually case-dependent. A typical example of unsupervised learning is dimension reduction by principle component analysis (PCA) [Hot33]). PCA calibrates a linear combination of the variables, in order to maximize the variance within the combination of variables. Finally, reinforcement learning is about an agent learning to take adequate actions based on states provided by an external environment, in order to maximize a reward signal. The application of reinforcement learning can be seen in control and robotics, and machine-based game player (for example the game of Go [SHM+16]). Reinforcement learning being out of the scope of the present manuscript, the frameworks of supervised learning and unsupervised learning are presented in figure 2.1.

2.2 From neurons to convolutional neural networks

2.2.1 Neurons and neural networks

In machine learning, an artificial neuron refers to a parameterized nonlinear operator composed of a linear transformation followed by a nonlinear activation (as shown in figure 2.2, although sometimes the activation can be omitted and the neuron degenerates to a linear unit). Let the input vector with d features denoted by $\mathbf{x} \in \mathbb{R}^{1\times d}$. The output of the artificial neuron is computed as:

$$y = \sigma(\mathbf{x} \cdot \mathbf{w} + b), \tag{2.1}$$

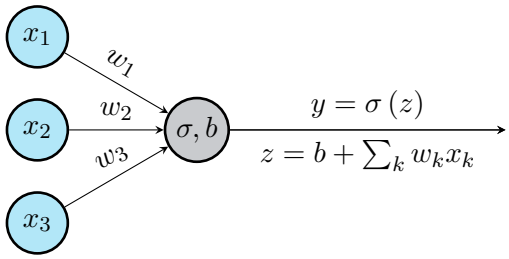
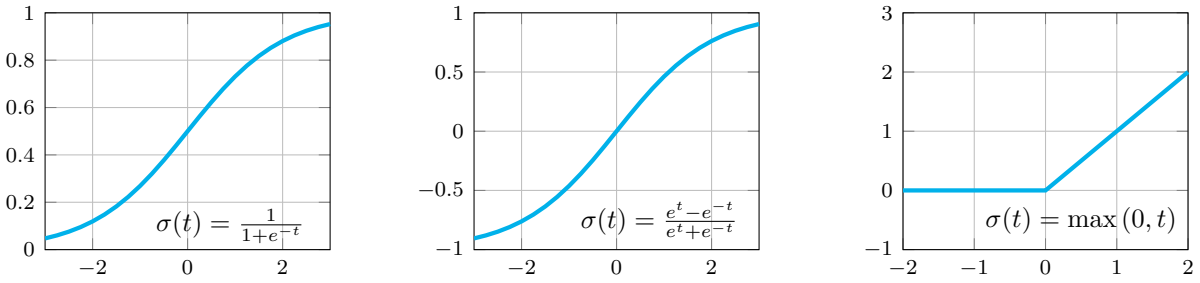


Figure 2.2 | A neuron has a nonlinear activation step after the linear combination of the input features. Neurons are hence adapted to nonlinear problems.

where σ refers to a nonlinear function, while $\mathbf{w} \in \mathbb{R}^{d\times 1}$ and $b \in \mathbb{R}$ refer to the weights and bias of the neuron. Compared to a simple linear unit, a neuron not only extracts information through combining the features of the input vector, but also introduces some nonlinearity. Neurons are hence more adapted to model nonlinear input patterns or input-output relations. In practice, various nonlinear forms are used (see fig 2.3), among which the sigmoid function, the hyperbolic tangent function (tanh), or the rectified linear unit (ReLU) [NH10] are some of the most common.



(a) sigmoid function **(b)** tanh function **(c)** ReLu function.

Figure 2.3 | Examples of popular activation functions. The activation functions are prefered to be differential, so that one can train a neuron by gradient-based optimization algorithms. The ReLu function is not differentiable at $x = 0$, but one can define a subgradient for the optimization [Sho12].

The modeling capacity of an ML algorithm is mostly about how complex a mapping it can represent between its inputs and its outputs. Due to the limited number of parameters and non-linearity, the capacity of a single neuron is not sufficient to represent highly complex mappings. However, by stacking multiple layers of neurons connected together as a network, it is possible to increase the complexity of the input-to-output relation, as shown in figure 2.4. The input vector is passed into a number of independent neurons in the first layer, each of which provides a scalar output. The outputs of all the neurons in the

first layer form a vector, which becomes the input of the following layer of neurons. One can repeat this basic computation block by stacking additional layers. Such a feedforward network of connected neurons is called a multilayer perceptron (MLP) [**?**].

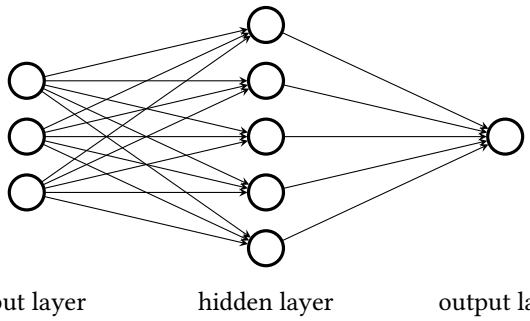


Figure 2.4 | An example multilayer perceptron. The input vector $x \in \mathbb{R}^3$ is regarded as an input layer with 3 neurons. The second layer is called the hidden layer, with 5 hidden nodes. Finally the output layer has 1 neuron, hence the output of this MLP is a scalar in $\mathbb{R}$.

In an MLP, the input layer and the output layer are fixed by the task, for example 3 and 1 neurons are respectively placed in the input layer and the output layer if one trains an MLP to predict children's height from the weight, the age and the gender. The number of hidden layers and their respective width (number of neurons per layer) remain to be customized. Theoretically, it is possible to stack an arbitrary number of hidden layers before the output layer. In each hidden layer, the number of neurons is also flexible. Suppose there are d_l neurons in the $l-$th hidden layer, the output vector $\mathbf{x}_l \in \mathbb{R}^{1\times d_l}$ of this layer is computed as:

$$\mathbf{x}_l = \sigma(\mathbf{x}_{l-1} \cdot \mathbf{W}_l + \mathbf{b}_l), \tag{2.2}$$

where $\mathbf{W}_l \in \mathbb{R}^{d_{l-1}\times d_l}$ and $\mathbf{b}_l \in \mathbb{R}^{1\times d_l}$ are the weight and bias of the $l-$th layer. Each column of $\mathbf{W}_l$ and $\mathbf{b}_l$ refers respectively to the weights and bias of a hidden neuron. For simplicity, here $\sigma(\cdot)$ denotes the activation function applied element-wise to the input vector. If an MLP has a total of L hidden layers with nonlinear activation σ, followed by an output layer without activation (in other words, the activation of the output layer is linear), then the final output $\mathbf{x}_{out}$ can be computed as:

$$\mathbf{x}_{out} = \sigma\left(\cdots\sigma\left(\mathbf{x}\cdot\mathbf{W}_1 + \mathbf{b}_1\right)\cdots\mathbf{W}_L + \mathbf{b}_L\right)\cdot\mathbf{W}_{out} + \mathbf{b}_{out}, \tag{2.3}$$

with $\mathbf{x} \in \mathbb{R}^{1\times d}$ being the input vector, $\mathbf{W}_{out}$ and $\mathbf{b}_{out}$ being the weights and bias of the output layer. The composition pattern of an MLP make it possible to increase the modeling capacity by stacking more hidden layers, which bring more free parameters as well as more nonlinearity. Indeed, the Universal Approximation Theorem [Cyb89] states that:

For any continuous function defined on the $n-$dimensional cube $[0,1]^n$, there exists an MLP with one single hidden layer and sigmoid activation, which can approximate the function to any desired accuracy.

Therefore, MLPs are powerful tools to extract nonlinear pattern or relation from massive data, where the nonlinear activation plays a crucial role in this process. To show the importance of nonlinear activation, consider replacing $\sigma(\cdot)$ in equation (2.3) by the identity function or any affine function. Since the composition of multiple linear operators is equivalent to a single linear operator, the MLP degenerates ultimately to a linear unit. Yet, the universal approximation theorem does not indicate how to

define the exact network architecture for a certain problem. The number and sizes of hidden layers are hence hyper-parameters to be tuned for each problem. In practice, the grid search (testing all the possible combinations of hyper-parameters) is a direct method to find architecture with good performance and acceptable complexity.

The name deep neural networks (DNN) refers to MLPs with many hidden layers, although there exists no strict boundary between DNNs and shallow MLPs. Since each hidden layer extracts higher-level features from the previous layer, DNNs are able to extract very abstract features from the input. Experiments show that DNNs can model complex data with fewer neurons than shallow MLPs at similar performance [Ben09]. DNNs also have advantages in automatic feature engineering: with shallow MLPs, it is usually inevitable to select or design useful features as the input vector. However, the first few layers in DNNs replace the manual feature engineering by parametric feature extraction. The ensemble of weights and bias of DNNs are calibrated so that the DNNs learn to extract useful features from the raw data. For this reason, DNNs have been actively applied to hard problems where people lack profound understanding on the underlying mechanism. For example, Ling *et al.* [LKT16] improved remarkably the accuracy of an existing turbulence closure through training a deep neural network with embedded Galilean invariance.

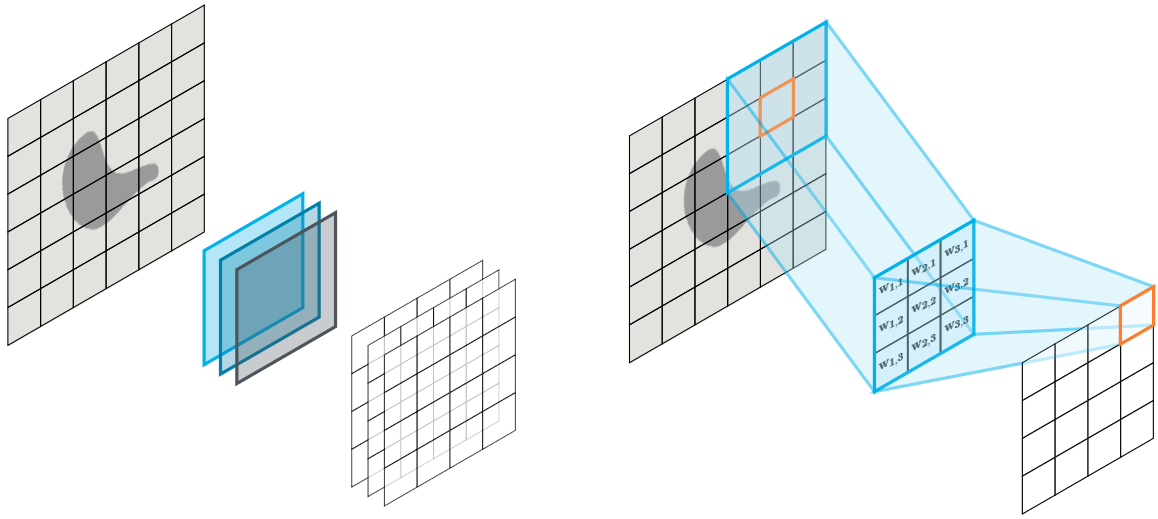
On the other hand, models based on DNNs are hardly interpretable. It is still a challenge to understand why DNNs perform so well on a large range of data-driven problems. Due to their high modelling capacity, DNNs can also easily overfit on the provided observations, and end up with poor performances in a generalization context (*i.e.* apply a calibrated model to non-observed cases). The application of DNNs in physics has received critics due to these disadvantages compared to traditional PDE-based models. Still, practical methods exist to improve the transparency and robustness of DNN-based models, which will be presented in the following chapters.

2.2.2 Convolutional neural networks

In traditional linear regression, it is usually preferred to use independent features in the input vector, so that the calibrated model coefficients indicate the variable importance of the provided features. With deep neural networks, the features are usually dependent, as feature engineering is no more required. In certain cases, the dependence contributes to essential information, as is the case for the neighbouring pixels in an image, which are naturally highly correlated. When using neural networks to process an image, flattening it into a one-dimensional vector implies the loss of this structural information, and makes the feature extraction inefficient. To this end, convolutional neural networks (CNN) were invented [Fuk88] to model data with local features, including not only images, but also speech and time series [LB+95].

CNNs refer to neural networks with convolutional layers, which are still composed of a number of neurons. The input of a convolutional layer is usually a tensor with a shape $h \times w \times c$, with h, w and c being the tensor's height, width and number of channels. For example, a gray image is a tensor with 1 channel, while an RGB image has 3 channels. Rather than performing a dot product with the whole tensor, a neuron in the convolutional layers focuses on a small receptive field, usually of square shape [1], with lateral dimension k. The $k^2 \times c$ components of the tensor are flattened into a vector and processed by the neuron, from which a scalar output is obtained. For every neuron, its receptive field slides over the input tensor, from left to right and from top to down. A scalar output is obtained every time the receptive field moves to a new location. All the scalar outputs are then stacked into a 2-D tensor while respecting the positions of receptive field. Finally the outputs of different neurons are concatenated in the channel

[1] Rectangular receptive fields are also allowed but in practice hardly employed.



(a) A convolutional layer holding three 3×3 kernels, thus producing three activation maps

(b) Convolution of a kernel with the input image

Figure 2.5 | Representation of a convolutional layer structure. Left: structure of a CNN layer holding multiple convolutional kernels applied to the same input, each producing a specific activation map. Right: detail of a convolutional kernel applied to an input image and producing an activation map.

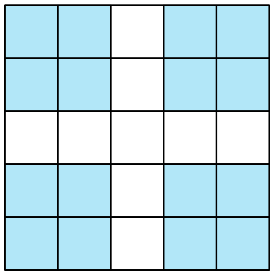
direction to form a 3-D tensor, as is shown in fig 2.5. Following the convention of image processing, the weights of a neuron in the convolutional layers are usually called a convolutional kernel.

Besides the number of kernels, a convolutional layer has 3 other hyper-parameters:

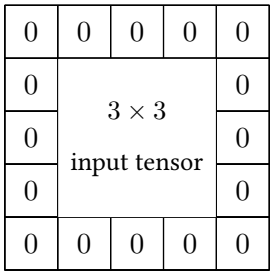
- The *kernel size* k defines how big the receptive field is. In practice, the kernel size k is usually smaller than 5, althouth some works let $k = 7$ [ZF14, SLJ+15] or 11 [KSH12] in the first layer. Generally large kernels are avoided, as the number of kernel's weights scales quadratically with k.
- The *stride* s denotes the number of pixels by which the receptive field slides after each operation (see figure 2.6a). Larger strides lead to less overlapping between neighboring receptive fields, and smaller output size.
- The *padding* operation (see figure 2.6b) increases the height and width of the input tensor by adding zeros to the left/right or up/down sides. With the kernel size being bigger than 1, the output tensor is smaller than the input. By padding zeros around the input, one can control the output size, and more importantly preserve the boundary information. The padding size p refers to the augmentation of height/width on each side.

The shape of the output tensor is determined by these hyper-parameters. If a convolutional layer has d_l kernels, the output size $h_{out} \times w_{out} \times c_{out}$ can be calculated as:

$$\begin{aligned} h_{out} &= \frac{h_{in} - k + 2p}{s} + 1, \\ w_{out} &= \frac{w_{in} - k + 2p}{s} + 1, \\ c_{out} &= d_l, \end{aligned} \tag{2.4}$$



(a) Convolution with $k = 2$, $s = 3$



(b) Padding with $p = 1$

Figure 2.6 | Stride and padding operations. With a stride $s > 1$, the receptive field (the blue pixels) jumps s units every time it slides. Zero padding allows the convolution kernel to cover all the pixels of the input tensor. By properly choosing the padding size p at each side, one can have an output tensor with the same height and width of the input.

with $h_{in} \times w_{in} \times c_{in}$ being the size of the input tensor. For simplicity, an even padding is considered for all the 4 sides. Sometimes, uneven padding is necessary to obtain integer output shape. The number of trainable weights and biases in one layer is then given by:

$$q = (k^2 c_{in} + 1) d_l$$

In comparison with convolutional layers, traditional layers taking all the features as input are called fully-connected layers or dense layers. Convolutional layers have the following advantages over dense layers:

- They can directly extract the local features without destroying the structure of the input tensor. CNNs are ideal for all image-like data, including cartesian grids. Indeed, the finite difference method has strong connections with convolutional layers. Approximating the gradients on a cartesian grid is technically a linear convolution with fixed kernels. For example, the kernels for central difference approximation are shown in figure 2.10.

0	0	0
$-\frac{1}{2h}$	0	$\frac{1}{2h}$
0	0	0

(a) Partial gradient $\frac{\partial}{\partial x}$

0	$\frac{1}{2h}$	0
0	0	0
0	$-\frac{1}{2h}$	0

(b) Partial gradient $\frac{\partial}{\partial y}$

Figure 2.7 | The kernels used for gradient approximation based on central difference. h is the grid size. Gradient approximation on cartesian grids can be regarded as a special convolution, with no nonlinear activation.

- They require fewer trainable weights. A convolutional kernel has $k^2 c$ weights, while a fully-connected neuron has hwc weights, which can represent a large amount of unknowns when treating high-resolution images. Being parsimonious, CNNs can be stacked into extremely deep architectures, *i.e.* with many hidden layers, while not exceeding hardware limit. GoogLeNet, the winner of the 1st place at the ILSVRC image classification competition in 2014 [SLJ+15], is a CNN architecture with 21 convolutional layers, but only 7 million parameters, the number of kernels in its convolutional layers ranging from 64 to 1024. By using a fully connected network to process the $224 \times 224 \times 3$ input images, 7 million parameters would only allow a network with 47 neurons.
- The output of a convolutional layer is an image-like tensor. This property makes CNNs the proper tools for full-field and end-to-end predictions.

Besides convolutional layers, the second basic type of layers in convolutional neural networks is the pooling layer. A pooling layer takes an input tensor and applies downsampling to reduce its height and width. Average pooling and max pooling [WAH93] are the two most spread approaches, with the latter being more popular in modern CNNs. As is shown in figure 2.8, a max pooling layer has two hyper-parameters: the pooling size and the stride. Similarly to convolutional layers, a receptive field slides over the input tensor with a given stride. In each channel, the pixels in the receptive field are downsampled into the output tensor. By setting the pooling size $m = 2$ and stride $s = 2$, if necessary with zero padding, one can halve the height and width of the input tensor. A max pooling layer usually follows a convolutional layer to preserve only the strongest activations. The pooling layers thus reduce the memory usage and the computation cost in CNNs, and help avoid overfitting by only preserving the most important features. The pooling operation also introduces certain translation invariance to the input tensor. This property is particularly valued in domains such as object detection, as people are interested in the existence of an object and not specifically in its position in the image.

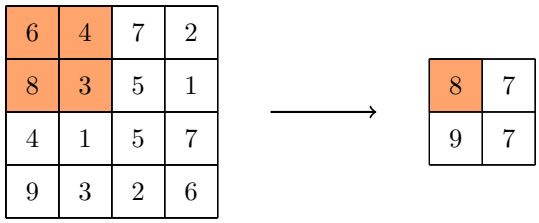


Figure 2.8 | Max pooling with pooling size$=(2,2)$ **and stride**$=2$**.** In the receptive field, only the maximum value is preserved in the output tensor. In CNNs, a max pooling layer is usually used to downsample the feature map of a convolutional layer.

A CNN architecture usually contains a succession of convolutional layers and pooling layers, to extract the hierarchy of features. If the CNN's output is expected to be a scalar or vector, dense layers are used before the output layer. A toy CNN architecture is given as example in figure 2.9. Although this CNN is of particularly limited size, standard CNN architectures can be orders of magnitude larger. For example, the AlexNet [KSH12] has 60 million trainable parameters, while the VGG16 net [SZ15] has a total of 138 million parameters. Complex CNNs have very high modeling capacity, and can learn complex patterns and functional relations from large amounts of data. As some references, the AlexNet and the VGG16 were trained against more than 1.2 million images.

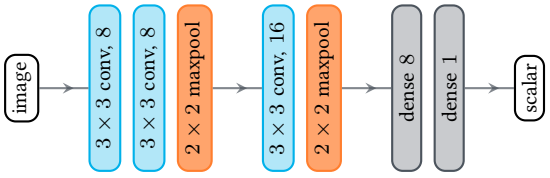
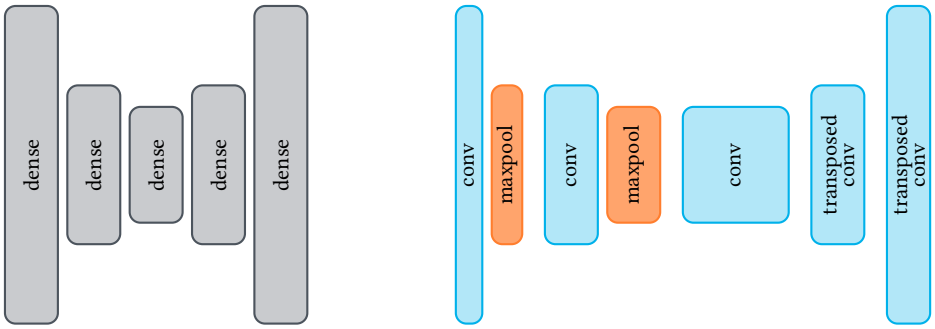


Figure 2.9 | A CNN structure composed of convolutional layers, pooling layers and dense layers. The input image is first passed to 2 consecutive convolutional layers, both with eight 3×3 kernels. Then a 2×2 max pooling layer downsamples the feature map. After this first block is a convolutional layer with 16 kernels, followed by a second pooling layer. The stride size is equal to 2 in all the layers. By doubling the number of kernels in deeper convolutional layers, one can extract more high-level features which are saved in different channels. Finally, two dense layers with 8 and 1 neurons respectively compute a scalar output from the hidden features.

2.2.3 Auto-encoders

An auto-encoder (AE) is a special type of feedforward neural network. The simplest AE has two parts: an encoder branch and a decoder branch (see figure 2.10a), the hidden layers being composed either of dense or convolutional layers. The size of the hidden layers in the encoder branch progressively decrease, until reaching the bottleneck of the architecture, then start increasing symmetrically to reach the original input size at the output. Each hidden layer in the encoder extracts a hierarchy of features and projects them into a lower-dimensional space. At the bottleneck, a compressed representation of the input is obtained, called latent feature. The decoder then reconstructs the input by decoding the latent features. The training of AEs is performed through minimizing the reconstruction error, and therefore does not require data with target values, thus belonging to the class of unsupervised learning techniques.



(a) Autoencoder **(b)** Convolutional autoencoder

Figure 2.10 | Autoencoder and convolutional autoencoder architectures. Autoencoders are feedward neural networks composed of encoders and decoders, which are often symmetric. AEs were created for principle analysis and dimension reduction. An AE extracts a compressed representation of the essential characteristics of the data set, which are restored in the latent features of the middle hidden layer.

A convolutional auto-encoder (CAE) has the same architecture than a traditional auto-encoder, although it is not symmetric in terms of the operations performed in the encoder and decoder branches. In the encoder branch, the dense layer are replaced with convolutional and pooling layers, just as in standard CNNs. In the decoder branch, upsampling operations are required to restore the original dimensions of the input. To do so, a popular method is the transposed convolutional layer, which performs upsampling and convolution at the same time. As is shown in figure 2.11, a transposed convolutional layer has 4 hyper-parameters: the number of kernels c, the kernel size k, the stride s, and the padding size p. The output tensor will have c channels, with each channel being the convolution result between one kernel and the input. For each kernel, the transposed convolution performs the following steps: first, $s-1$ zeros are inserted between every two neighboring pixels in the same channel, which upsamples the input by a factor of s. Then, a normal convolution with kernel size k and an actual stride $s'=1$ is applied to the upsampled tensor. In order to recover the expected output shape, zero padding is necessary, with the padding size p solved from the equations (2.4). If the output shape is expected to be $h_{out} \times w_{out} \times c$, the padding size is computed as:

$$\begin{aligned} 2p_h &= h_{out} - (h_{in}-1)s + k - 2, \\ 2p_w &= w_{out} - (w_{in}-1)s + k - 2, \end{aligned} \tag{2.5}$$

where even padding is assumed on both sides. Uneven padding must be adopted if the calculated padding size is not an integer, which is the case of figure 2.11.

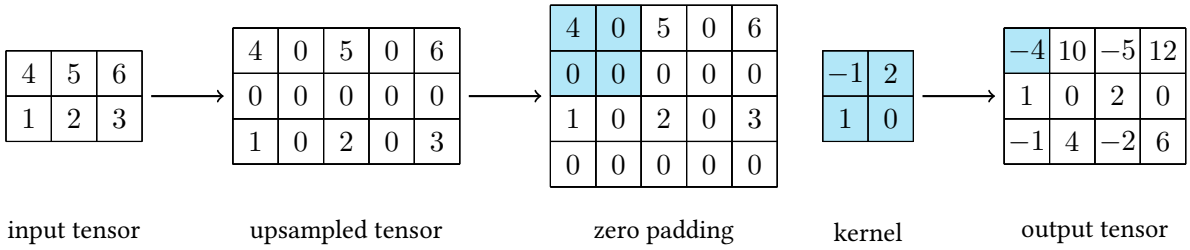


Figure 2.11 | The transposed convolution with kernel size $k = 2$ and stride $s = 2$. Here, the input tensor only has one channel for simplicity. The upsampling inserts $s - 1$ zeros between every two neighboring pixels. Then, zero padding is applied in order to control the output size. Here, the padding size $p = 1$ is determined by the equations 2.4. The upsampled tensor is convoluted with the kernel, whose actual stride is $s' = 1$. The output is a 3×4 tensor, with its pixel values computed by supposing the kernel has zero bias.

AEs were initially designed for principle component analysis [Kra91] and dimension reduction [HS06]. Yet, in the recent years, the application of AEs in anomaly detection processes has increased significantly. Anomaly detection consists in identifying outliers lying outside of the boundaries of the data set: since AEs are trained to approximate the identity map on the available data set, the reconstruction of an input that is too different from other data set elements is likely to be poor. Therefore, the reconstruction error is often used as a measure to detect anomalies. Trained in a supervised framework, convolutional AEs can also be exploited for image processing besides anomaly detection. Indeed, they can be trained for end-to-end prediction tasks, where an input image is mapped to a target image with the same height and width. For instance, CAEs are used in the biomedical domain for image segmentation tasks [RFB15], *i.e.* dividing an image into disjoint parts which are distinguished by the pixel labels inside.

2.2.4 Graph convolutional neural networks

The graph is an unstructured data topology usually denoted by $G(V, E)$, with $V = \{v_i\}_{i=1}^{n_v}$ being a set of nodes and $E = \{e_i\}_{i=1}^{n_e}$ being the set of edges connecting node pairs. The edges can be directed or undirected depending on the connection property. In the present manuscript, only undirected graphs will be considered, and the undirected property will be implicitly assumed for graphs in the remaining chapters. The features on a graph are written into the node feature matrix $\mathbf{X}_V \in \mathbb{R}^{n_v \times d_v}$ and the edge feature matrix $\mathbf{X}_E \in \mathbb{R}^{n_e \times d_e}$, with each row representing the feature vector $\mathbf{x}_v \in \mathbb{R}^{d_v}$ on a node and $\mathbf{x}_e \in \mathbb{R}^{d_e}$ on an edge. An example of graph is shown in figure 2.12

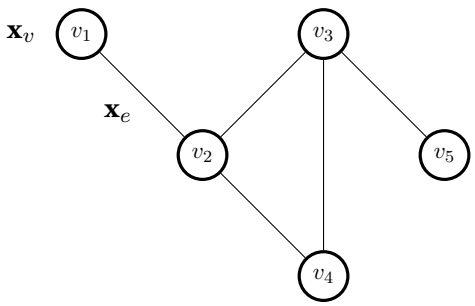


Figure 2.12 | Nodes, edges and features in a graph. The nodes and edges determine the graph's topology. The graph convolution updates the node features by considering the node-node and node-edge interactions.

Social networks, traffic networks, protein molecules and unstructured meshes are some examples of graph-like data. The motivation of graph convolutional neural networks (GCNN) is to extend CNN

technics to general graphs, with the core challenge being to devise a proper convolution operation. The early solution, called the spectral convolution, is inspired by methods of graph signal processing. By means of graph Fourier transform, the spectral convolution turns the convolution into the product of the graph's eigen-decomposed feature matrix and the convolutional kernel in the Fourier space, then followed by a nonlinear activation [DBV16, KW17]. Such a convolution requires an expensive eigen-decomposition and a fixed graph topology, *i.e.* a kernel calibrated on one graph would not be applicable to other graphs. Therefore, the spectral convolution has limited applicable scenarios. To develop a convolution strategy similar to that of standard CNNs, two major obstacles, originating from the unstructured topology [ZTXM19, ZCH+20], must be overcome:

- ◇ While each pixel in an image has 4 direct neighbors, the nodes in a graph can have a variable number of connections to other nodes,
- ◇ The pixels in the images are always indexed from left to right and from top to bottom, but the indexing of nodes and edges in a graph are completely arbitrary.

It is hence not adequate to use a constant convolutional kernel sliding over the whole graph. Furthermore, the kernel must be invariant to the permutation of nodes and edges (convolution methods complying to these characteristics are called spatial convolutions). In the literature, two main frameworks are proposed. In [MBM+17], Monti *et al.* proposed a parametric weighting function to aggregate neighboring features. For a node v with a feature vector $\mathbf{x}_v \in \mathbb{R}^{n_v}$, the convolution with a layer of n'_v kernels is written as:

$$\mathbf{x}_{v',i} = \sigma\left(\sum_{u\in\mathcal{N}(v)} w_i\left(\mathbf{f}(u,v)\right)\mathbf{x}_u\right),\ i = 1,2,...,n'_v \tag{2.6}$$

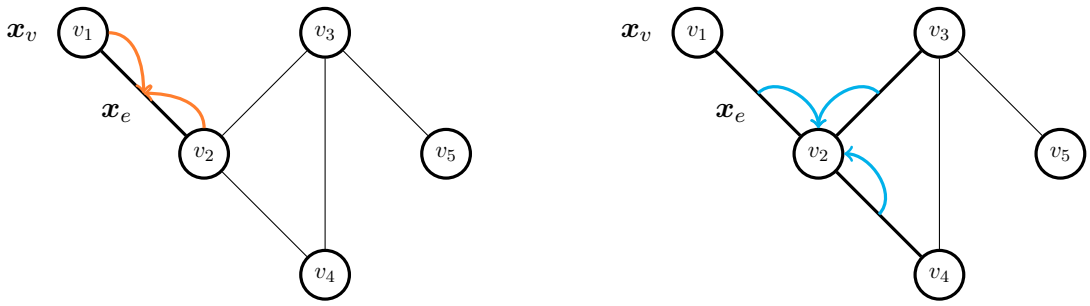
where $\mathbf{x}'_v \in \mathbb{R}^{n'_v}$ is the output feature vector, and $\mathcal{N}(v)$ is the set of neighboring nodes of node v. The most important component of equation (2.6) is the weighting function $w_i(\mathbf{f}(u,v))$, which computes a contribution factor for each node u connected with v. The weighting function w_i is shared by all the edges, so the whole feature matrix is updated by applying (2.6) to all the nodes. The input $\mathbf{f}(u,v)$ is some pseudo-coordinates of u in the neighborhood of v. A simple example of pseudo-coordinates and weighting function can be:

$$\begin{aligned}\mathbf{f}(u,v) &= \left(\frac{1}{\deg(u)}, \frac{1}{\deg(v)}\right)^T,\\ w_i(\mathbf{f}(u,v)) &= \exp\Big(-\frac{1}{2}(\mathbf{f}-\mathbf{f}_i)^T\mathbf{\Sigma}_i^{-1}(\mathbf{f}-\mathbf{f}_i)\Big),\ \text{for } 1\le i\le n'_v\end{aligned} \tag{2.7}$$

with $\deg(v)$ being the number of nodes connected to v. The weighting equation in (2.7) is a parametric Gaussian kernel with diagonal covariance matrix, with $\mathbf{f}_i$, $\mathbf{\Sigma}_i$ being the learnable parameters. For a convolutional layer with n'_v kernels, the total number of trainable parameters would be $4n'_v$.

A second possible convolution strategy is called neural message passing [GSR+17], which was generalized to the graph network (GN) framework by Battaglia *et al.* in [BHB+18]. The latter abandons the idea of weighted summation inherited from traditional CNNs. The message passing procedure is split into two steps (see figure 2.13):

$$\begin{aligned}\mathbf{h}_v &= \sum_{u\in\mathcal{N}(v)} \mathbf{f}_e\left(\mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_{e,uv}\right),\\ \mathbf{x}'_v &= \mathbf{f}_v\left(\mathbf{x}_v, \mathbf{h}_v\right),\end{aligned} \tag{2.8}$$



(a) Passing messages from nodes to edges

(b) Passing message from edges to nodes

Figure 2.13 | Message passing on graphs. The message passing procedure splits the interactions among nodes into two steps: (i) updating edge features by nodes features, and (ii) updating node features by edge features. Both steps require a transformation function which is invariant to arbitrary nodes and edges indexing.

with $\mathbf{x}_{e,uv}$ being the feature vector of the edge connecting nodes u and v. The first step in equation (2.8) employs an edge kernel $\mathbf{f}_e$ to update the edge features. Then around each node, the relevant edge updates are aggregated to node-wise messages $\mathbf{h}_v$. The second step employs a node kernel $\mathbf{f}_v$ to receive the messages and update the node features. In each convolution layer, the kernels $\mathbf{f}_e/\mathbf{f}_v$ are applied to all the edges/nodes. Shallow MLPs are usually employed as the kernel [BPL+16], with the input vector being the concatenation of all the variables. The kernel's output layer remains flexible, so that the dimension of the message $\mathbf{h}_v$ and the updated node feature $\mathbf{x}'_v$ can be customized. However, the message passing framework was first proposed for directed graphs, where the message sender and the message receiver are distinct. Here, we propose a modification to the original framework, to make it symmetric in the context of undirected graphs:

$$\begin{aligned} \mathbf{h}_v &= \sum_{u \in \mathcal{N}(v)} \mathbf{f}_e \left(\frac{\mathbf{x}_u + \mathbf{x}_v}{2}, \frac{|\mathbf{x}_u - \mathbf{x}_v|}{2}, \mathbf{x}_{e,uv} \right), \\ \mathbf{x}'_v &= \mathbf{f}_v \left(\mathbf{x}_v, \mathbf{h}_v \right), \end{aligned} \tag{2.9}$$

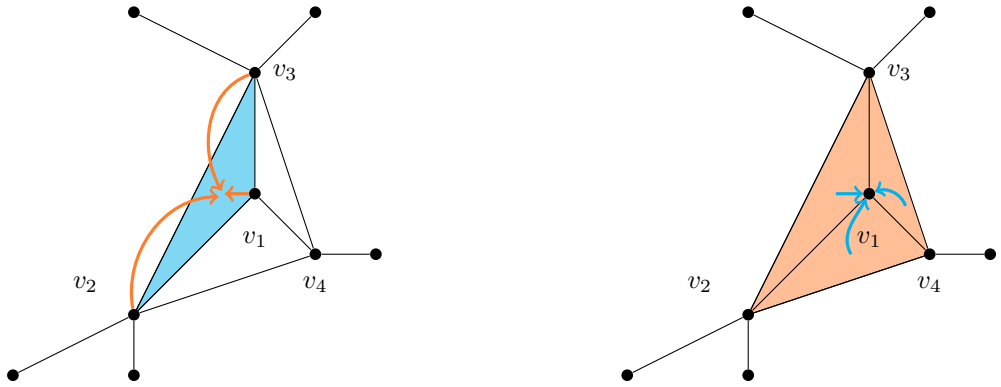
With the proposed symmetrized node features, the edge convolution step becomes invariant to node permutation. The convolutional layers based on equations (2.9) are thus potential tools to modelize data on unstructured meshes used in finite element analysis. Indeed, similar message passing process exists in gradient approximation in FEM. Figure 2.14 shows the average-based method to calculate the first-order gradient of a scalar field defined on the nodes [CHM09]. The first step consists in calculating the gradient in the elements, which is element-wise constant when using linear shape functions. As a simple example in figure 2.14, let the scalar values and coordinates on the nodes be $\{z_i\}_{i=1}^3$ and $\{(x_i, y_i)\}_{i=1}^3$. The interpolated plane $z = ax + by + c$ inside this element can be obtained by solving a set of local equations:

$$ax_i + by_i + c = z_i, \text{ for } i = 1, 2, 3 \tag{2.10}$$

The gradient in the element $(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}) = (a, b)$ is thus a linear interpolation of the node values $\{z_i\}_{i=1}^3$, meaning that this step passes messages from the nodes to the elements. The second step consists in obtaining the node-wise gradient through the simple average of element-wise values:

$$\frac{\partial z}{\partial x}|_v = \frac{1}{\text{card}\left(\mathcal{N}_e(v)\right)} \sum_{e \in \mathcal{N}_e(v)} \left.\frac{\partial z}{\partial x}\right|_e,$$
$$\frac{\partial z}{\partial y}|_v = \frac{1}{\text{card}\left(\mathcal{N}_e(v)\right)} \sum_{e \in \mathcal{N}_e(v)} \left.\frac{\partial z}{\partial y}\right|_e, \tag{2.11}$$

where $\text{card}\left(\mathcal{N}_e(v)\right)$ is the number of neighboring elements of node v. Although the passage of message in the FEM example is nodes $\rightarrow$ elements $\rightarrow$ nodes, the basic idea is the same as the message passing nodes $\rightarrow$ edges $\rightarrow$ nodes in machine learning. Approximating gradients in linear finite element methods can hence be regarded as a linear convolution, with a spatially varying kernel depending on the node coordinates.



(a) Calculate element-wise gradient

(b) Calculate node-wise gradient

Figure 2.14 | Gradient approximation in linear FEM. The gradient in elements are determined by node values, which corresponds to a message passing from nodes to elements. Then around each node, the element-wise gradient are averaged to obtain the node-wise gradient. The averaging step corresponds a message passing from elements to nodes.

2.3 Optimization for convolutional neural networks

2.3.1 Gradient backpropagation and automatic differentiation

Training a neural network is realized through calibrating its parameters using optimization algorithms. In this section, we suppose the model is trained in a supervised setting, with the data set denoted by:

$$(\mathbf{X}, \mathbf{Y}) = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n.$$

While unsupervised learning uses a different loss function, the methods presented here can be generalized without further conditions. A supervised model aims at predicting a target value given an input:

$$\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \boldsymbol{\theta}), \tag{2.12}$$

where $\mathbf{x}$ and $\mathbf{y}$ are the input and output. The trainable parameters are represented by a high-dimensional vector $\boldsymbol{\theta}$, with its dimension being possibly up to several millions for convolutional neural networks. The prediction error is usually evaluated by a loss function l:

$$l(\hat{\mathbf{y}}, \mathbf{y}) = ||\hat{\mathbf{y}} - \mathbf{y}||.$$

For regression problems, the quadratic error and absolute error are common choices for the loss function. The calibration is then realized through minimizing an objective function L defined as the average loss on the data set plus a regularization term:

$$\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathbf{X}, \mathbf{Y}) = \min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} l\left(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}), \mathbf{y}_i\right) + \lambda \Omega(\boldsymbol{\theta}), \tag{2.13}$$

with $\Omega(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_{l_1}$ or $||\boldsymbol{\theta}||_{l_2}$, being respectively called l_1 and l_2 regularization. Minimizing the regularization term Ω tends to pull the parameters to zero, which leads to less complex models and avoids overfitting due to noisy or too small data set [KH92, Ng04]. The regularization rate λ remains a hyper-parameter of the model, and needs to be tuned by the user. Following the basic gradient descent algorithm, trainable parameters are updated iteratively by:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \left(\frac{1}{n} \sum_{i=1}^{n} \frac{\partial l(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}_k), \mathbf{y}_i)}{\partial \boldsymbol{\theta}} + \frac{\partial \Omega}{\partial \boldsymbol{\theta}} \right), \tag{2.14}$$

with $\alpha_k > 0$ being the step size at the $k-$th iteration.

It is well known that the gradient descent leads to a unique minimum if the objective function defined on a convex set is strictly convex [BBV04]. However, in NN-based machine learning, solving the problem (2.13) hardly leads to a unique global minimum due to the prevailing non-convexity of L. Optimization plays hence a crucial role in order to obtain a performant model. Following the update rule given by equation (2.14), at each iteration one needs to evaluate the gradient contributed by the fitting error and the regularization. The gradient contributed by the regularization term is easy to obtain:

$$\frac{\partial \Omega}{\partial \boldsymbol{\theta}} = \begin{cases} 2\boldsymbol{\theta} & \text{if } \Omega = ||\boldsymbol{\theta}||_{l_2}, \\ \text{sign}(\boldsymbol{\theta}) & \text{if } \Omega = ||\boldsymbol{\theta}||_{l_1}, \end{cases} \tag{2.15}$$

which does not depend on the architecture of the neural network. On the other hand, the fitting error involves the model output, making it necessary to consider how the output is computed by the network. For a neural network with N hidden layers $\{\mathbf{f}_i\}_{i=1}^{N}$ and an output layer $\mathbf{f}_{out}$, one can rewrite equation (2.12) as:

$$\begin{aligned} \hat{\mathbf{y}} &= \mathbf{f}_{out}(\mathbf{h}_N, \boldsymbol{\theta}_{out}), \\ \mathbf{h}_i &= \mathbf{f}_i(\mathbf{h}_{i-1}, \boldsymbol{\theta}_i), \text{ for } i = 2, \ldots, N, \\ \mathbf{h}_1 &= \mathbf{f}_1(\mathbf{x}, \boldsymbol{\theta}_1), \end{aligned} \tag{2.16}$$

with $\mathbf{h}_i$ and $\boldsymbol{\theta}_i$ being respectively the output and the trainable parameters of the $i-$th hidden layer. The gradient backpropagation [RHW86] (BP) is a method to compute the the gradient of loss with respect to each trainable parameter of the network. Instead of using finite difference to treating each trainable parameter separately, the BP method does it layer by layer following the chain rule:

- For the output layer,

$$\begin{aligned} \frac{\partial l}{\partial \boldsymbol{\theta}_{out}} &= \frac{\partial l}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \boldsymbol{\theta}_{out}}, \\ \frac{\partial l}{\partial \mathbf{h}_N} &= \frac{\partial l}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}_N}, \end{aligned} \tag{2.17}$$

◇ For $i = N, N-1, \ldots, 2$,

$$\begin{aligned} \frac{\partial l}{\partial \boldsymbol{\theta}_i} &= \frac{\partial l}{\partial \mathbf{h}_i} \frac{\partial \mathbf{h}_i}{\partial \boldsymbol{\theta}_i}, \\ \frac{\partial l}{\partial \mathbf{h}_{i-1}} &= \frac{\partial l}{\partial \mathbf{h}_i} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}, \end{aligned} \tag{2.18}$$

◇ For the first hidden layer,

$$\frac{\partial l}{\partial \boldsymbol{\theta}_1} = \frac{\partial l}{\partial \mathbf{h}_1} \frac{\partial \mathbf{h}_1}{\partial \boldsymbol{\theta}_1} \tag{2.19}$$

In one backprogation, the state gradient $\frac{\partial l}{\partial \mathbf{h}_i}$ is stored in the memory and reused for the layer closer to the input. Through equations (2.17-2.19), the prediction error is propagated backward to the neurons in the first hidden layer.

The BP is an efficient strategy to obtain the exact gradients, as it completely avoids duplicate computation. It is a special case of the so called reverse mode automatic differentiation (AD) used to compute gradients. AD is an algorithmic differentiation method to be distinguished from finite difference and symbolic differentiation. AD is more efficient than finite difference, because it does not apply a small perturbation to compute the partial derivative of each trainable parameter separately, which leads to lots of repeated computations. Compared to symbolic differentiation, AD does not generate complex formulas for gradient computation.

To be more specific, the forward calculation defined by equation (2.16) is broken down to a sequence of primitive operations, including multiplication, square, reciprocal and exponent. These operations can be easily differentiated without doing finite difference. In the implementation of reverse mode AD, the computer opens a 'tape' and does the forward calculation, while watching the intermediate operations and storing the intermediate variables in the memory. Then the operations are differentiated in the backward sense as described by equations (2.17-2.19).

In the domain of neural networks, reverse mode AD has been proven to be the most efficient approach for gradient-based optimization [BPRS18]. In popular platforms of neural network programming, reverse mode AD is pre-implemented so that machine learning practitioners do not need to manually compute the gradients. The training time of large-scale neural networks is also remarkably reduced thanks to the high efficiency of this technique.

2.3.2 Stochastic gradient descent

The standard weight update step (2.14) is also called the batch gradient descent, as it uses an objective function computed over the entire data set. For large data sets and complex neural networks, using batch gradient descent can lead to unacceptably large memory usage and computational time. To overcome this issue, stochastic gradient descent (SGD) was proposed by Bottou *et al.* in [Bot10]. At each iteration of the SGD, the objective function is defined on a single sample for gradient descent:

$$\begin{aligned} &\text{Sample } i \underset{\pi}{\sim} \{1, 2, \ldots, n\}, \\ &\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \frac{\partial l(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}_k), \mathbf{y}_i)}{\partial \boldsymbol{\theta}}, \end{aligned} \tag{2.20}$$

where π denotes the uniform distribution. In practice, the random selection is replaced by using all the data examples one after the other, with a full passage through the data set called one epoch. After each epoch, the data set is shuffled to avoid repeated cycles, and the SGD is usually run for many epochs to sufficiently minimize the error.

While the batch gradient descent method (batch GD) decreases the loss value over the whole data set in the fastest direction, SGD approximates the true gradient by considering a single data sample. Since the sampling follows a uniform distribution, the stochastic gradient is an unbiased estimation of the true gradient:

$$\mathop{\mathbb{E}}_{i \underset{\pi}{\sim} \{1,2,\dots,n\}} \left(\frac{\partial l(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}_k), \mathbf{y}_i)}{\partial \boldsymbol{\theta}} | \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n \right) = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial l(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}_k), \mathbf{y}_i)}{\partial \boldsymbol{\theta}},$$

which means in the long term, SGD also tries to minimize the objective function. The convergence of SGD is usually noisy and requires more iterations compared to batch GD. However, this can become an advantage when training neural networks, as when the objective function is non-convex and possesses many local minima, the noisy gradient approximation allows SGD to escape local minima and continue the search of a global minimum. In the contrary, the batch GD can easily get trapped in a local minimum.

A last alternative, introduced in [LZCS14], is called mini-batch gradient descent. The latter aims at accelerating SGD's convergence while keeping its noisy convergence path. This method uses a subset of the data set to approximate the gradient at each iteration. With enough data in a batch, it provides a descent direction very close to the true gradient:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \frac{1}{m} \sum_{\mathbf{x}_i \in \mathbf{b}} \frac{\partial l(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}_k), \mathbf{y}_i)}{\partial \boldsymbol{\theta}}, \tag{2.21}$$

with $\mathbf{b}$ the current mini-batch and m being the batch size. In the meantime, with the batch being small enough, the randomness is preserved to avoid local minima. The proper batch size remains a hyper-parameter of the method, to be tuned by the user. The batch GD and SGD can be regarded as two extreme cases of mini-batch GD, with respectively n and 1 being the mini-batch sizes. In practice, the data set is split into disjoint mini-batches, which are used for gradient approximation one after the other. A full passage through all the batches is called an epoch, and the data set is shuffled at the start of each new epoch. A sketch of the convergence passages of the presented gradient descent algorithms is shown in figure 2.15.

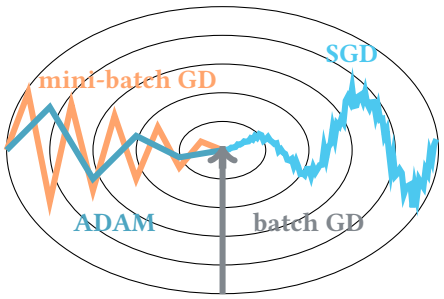


Figure 2.15 | Batch gradient descent, stochastic gradient descent and mini-batch gradient descent. The batch gradient descent converges through the shortest passage. The SGD takes a very noisy passage but arrives at the optimum in the end. Mini-batch gradient descent is a mixture the former two methods.

Still, options exist to accelerate the mini-batch GD. One of the most used algorithm is the adaptive moment estimation (ADAM) optimizer, introduced in [KB15]. Let the mini-batch gradient at iteration k be denoted by $\mathbf{g}_k$, then the neural network parameters are updated by:

$$\begin{aligned} \mathbf{m}_k &= \beta_1 \mathbf{m}_{k-1} + (1-\beta_1)\mathbf{g}_k, \\ \mathbf{v}_k &= \beta_2 \mathbf{v}_{k-1} + (1-\beta_2)\mathbf{g}_k^2, \\ \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \frac{\alpha_k}{\sqrt{\mathbf{v}_k}+\epsilon}\mathbf{m}_k, \end{aligned} \tag{2.22}$$

with $\beta_1 \in [0,1]$ and $\beta_2 \in [0,1]$ being two hyper-parameters, and ϵ a small number used to avoid zero denominator. In equations (2.22), the square $\mathbf{g}_k^2$, the square root $\sqrt{\mathbf{v}_k}$ and the division $\frac{\alpha_k}{\sqrt{\mathbf{v}_k}+\epsilon}$ involving vectors are all element-wise operations. Compared to equation (2.21), the ADAM optimizer introduces two new iterative variables: the moment $\mathbf{m}_k$ and the adaptive term $\mathbf{v}_k$.

- The moment $\mathbf{m}_k$ decides the descent direction. Its intial value is zero, and it is updated at each iteration based on its own value and the gradient. The moment is indeed an exponential moving average of previous gradients, which provides a smoother descent direction to reduce oscillations:

$$\mathbf{m}_k = \sum_{i=1}^{k} \beta_1^{k-i}(1-\beta_1)\mathbf{g}_i.$$

- The adaptive term $\mathbf{v}_k$ brings an element-wise modification to the learning rate α_k, with the scale factor determined by an exponential moving average of the previous partial derivatives [HSS12]. The motivation is to assign different learning rates to neurons in different hidden layers, which is reasonable for the network's hierarchical architecture. Figure 2.15 is the contour near the optimum of some 2-D quadratic function. Closing to the optimum, the objective function has a large derivative in the y-direction, and a relatively small derivative in the x-direction, which is a very common case in multi-dimensional optimization. By normalizing the learning rate α_k by $\sqrt{\mathbf{v}_k}+\epsilon$, the partial derivative in the y-direction is dumped to avoid oscillation, while in the x-direction the step size is magnified to obtain a more aggressive descent. With the adaptive term, using a large learning rate α_k is usually encouraged to accelerate the descent further.

2.3.3 Weights and biases initialization

Weights and biases initializations play an important role in the overall performance of the neural network training. Although there exist no absolute principles, some general rules are listed below:

Zero weights initialization is usually avoided, as it introduces symmetry and usually makes the optimization process fail at the first iteration. Indeed, the neurons in the same hidden layer would have the same contribution in the forward calculation, and would get the same updates in the backpropagation. Therefore, inside each hidden layer, the neurons would always have the same weights and biases in the following iterations, thus reducing dramatically the modeling capacity of the network. To this end, the weights are usually initialized randomly to break the symmetry. Conversely, biases can be simply initialized to zero, as the different neurons contributions are already distinguishable by the random weights.

The initial values should not be too small nor too large. In practice, the initial weights are sampled from given probability distributions, among which the uniform distribution $\mathcal{U}[-t,t]$ and the Gaussian distribution $\mathcal{N}(0,\sigma^2)$ are two commonly used options, with t and σ tunable parameters to modify the variance. Since the gradient backpropagation is based on chain rule, neural networks with very deep architecture can suffer from gradient vanishing or gradient exploding issues, which usually result from too small or too large initial weights. Conversely, initializating weights with adequate magnitudes can accelerate the training process.

In order to let the gradient values be independent of the layer's position (so that all the hidden layers are updated at the same pace), the Glorot method [GB10] suggests to sample the weights from a modified uniform distribution:

$$\mathcal{U}\left[-\sqrt{\frac{6}{d_{l-1}+d_l}}, \sqrt{\frac{6}{d_{l-1}+d_l}}\right], \tag{2.23}$$

or modified Gaussian distribution:

$$\mathcal{N}\left(0, \frac{2}{d_{l-1}+d_l}\right), \tag{2.24}$$

with d_{l-1} and d_l being the dimension of the layer's input and output. The Glorot method can approximately maintain the variance of the layer-wise gradient across the hidden layers. Since the proposed distributions in equations (2.23) and (2.24) are zero-centered, this is equivalent to maintain the magnitude of gradient values in the probabilistic sense. Being originally proposed for hidden layers with a hyperbolic tangent activation, the Glorot method needs a slight scaling on its variance term [HZRS15] for other activations.

2.3.4 Learning rate decay

The learning rate α_k determines the step size taken by the network parameters at the $k-$th iteration of the optimization process. At the beginning, a large learning rate is usually preferred to quickly decrease the loss value. While the loss gradually converges, it is often necessary to use smaller learning rates to smoothly approach the optimum. To this end, a decay schedule for the learning rate usually helps to accelerate the optimization. For example, one can half the learning rate after K iterations, which is called piecewise constant decay:

$$\alpha_k = \alpha_0 / 2^{[k/K]}, \tag{2.25}$$

with $[k/K]$ being the integer part of k/K. Other standard approaches include the exponential decay:

$$\alpha_k = \alpha_0 \mathrm{e}^{-k/K}, \tag{2.26}$$

and the inverse time decay:

$$\alpha_k = \frac{\alpha_0}{1 + k/K}. \tag{2.27}$$

In order to select a proper decay schedule, the initial learning rate as well as the decaying rate remains two hyper-parameters to be tuned by the user. A sketch of different decay schedules is shown in figure 2.16.

2.3.5 Validation and test

When training large scale deep neural networks against a data set, over-fitting is an inevitable issue. Over-fitting means that the model has not only learned generic features from the training data, but also features specific to the samples present in it. This usually leads to a low fitting error on the training subset, but to a poor generalization performance on unseen samples. Over-fitting can have several causes: too complex fitting functions, too small data set, noisy observations... A classical over-fitting example in polynomial interpolation is shown in figure 2.17. The regularization term in equation (2.13) can help

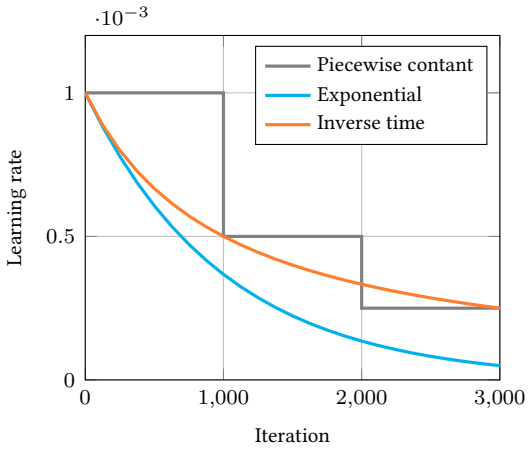


Figure 2.16 | Learning rate decay schedules. The curves are plotted with fixed initial learning rate $\alpha_0 = 0.001$ and decay speed $K = 1,000$.

generate simpler models by limiting the magnitude of the trainable parameters. In the example of figure 2.17, a regularization term can decrease the contribution of the $4-$th and $3-$rd order components, and reduce the prediction error not only in the interval $[0, 5)$ but also in $[5, +\infty)$.

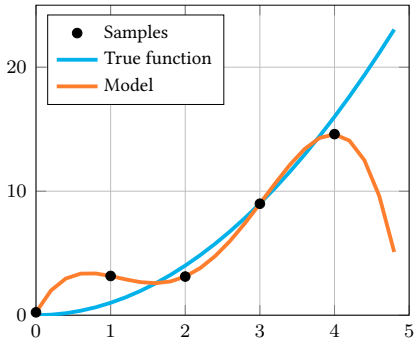
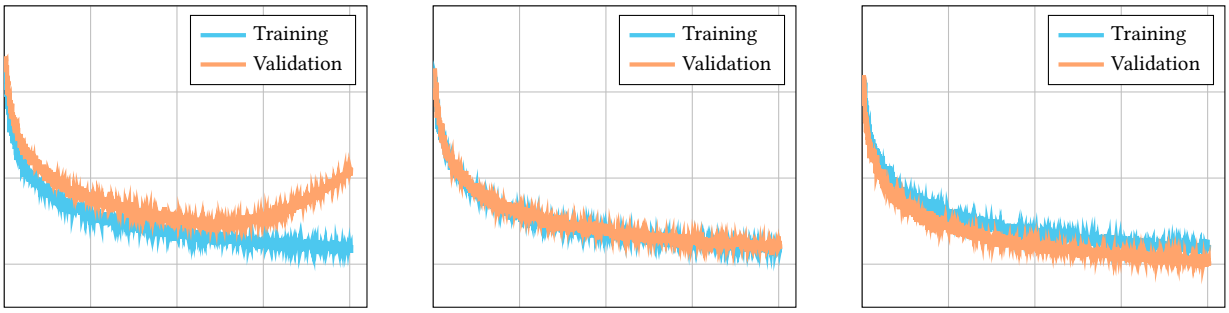


Figure 2.17 | Overfitting in polynomial interpolation. The sample points are generated from a quadratic function, to which a Gaussian noise is added. When using a 4^{th} order polynomial to fit the samples, the error on the samples can be small but the generalization error becomes large.

A different approach is to stop the training at a proper stage, so that the training loss is sufficiently reduced, and at the same time one is confident to expect a similar generalization loss. To this end, the observed data set should be split into three subsets for training, validation and testing. The training subset contains the major part of the data set (usually more than 70%), and is used to calibrate the neural network. After each training epoch, the averaged loss value on the validation set is evaluated, which serves as an approximation to the generalization error. In the early stage, both the training loss and validation loss are expected to decrease rapidly. After a certain number of epochs, one may observe three different situations:

- In figure 2.18a, the training loss converges, but the validation loss experiences a rebound after the a descent. This means that the selected NN architecture is too complex in regard of the underlying patterns in the data set, and the training should have been ceased before the rebound. A possible remedy would be to use smaller neural networks, and/or to use a stronger regularization term;

- In figure 2.18b, the training and validation losses are converging similarly. In this case, it is usually assumed that the model is adequate for the considered problem, and would have similar performance when applied to unseen inputs. However, this argument is only valid for interpolation. The extrapolation error, *i.e.* the prediction error when the model is applied to an outlier with respect to the available data set, will not necessarily be of similar level. In order to have good performance on a broader range of data, it is crucial to collect a larger data set with more variability for the model training.
- In figure 2.18c, both the training and validation losses converge, but the training loss is visibly larger than the validation one. this means that the selected architecture is not able to fully infer the underlying pattern in the data set. This case is usually called under-fitting. More layers and neurons should be added to the network to increase its modeling capacity.



(a) Over-fitting **(b)** Reasonable fitting **(c)** Under-fitting

Figure 2.18 | Training loss and validation loss. The validation loss is supervised as an approximation to the generalization error. If the validation loss does not improve anymore, we suppose it is the right moment to terminate the training to prevent over-fitting. This technique is called early stopping.

In practice, the validation loss is used as an early stopping criterion to the training process. A patience parameter is set a priori: if the validation loss is not improved for a certain number of epochs, the training is terminated, and the parameters leading to the smallest validation loss is used for the final model.

Finally the test set is left aside until the training is finished, and is used to evaluate the performance of the trained model. Since the training is biased to the validation set due to the early stopping criterion, the validation performance is usually better than the performance on unseen data. Evaluating the error level on an independent test set provides an more accurate idea of how performant the final model is.

2.4 Some applications of CNNs in CFD

In the recent years, CNNs architectures have been widely exploited in the context of fluid dynamics. In this section, a selection of recent contributions are presented.

In [GLI16], Guo *et al.* trained a convolutional auto-encoder for steady velocity approximation around two-dimensional and three-dimensional obstacles. Instead of a binary representation of the input geometry, the authors used the signed distance function (SDF), which calculates the distance of each pixel to the solid boundary. Using an encoder to extract latent features from the SDF image, the proposed CNN architecture has separate decoders to predict different components of the velocity field. The resolved velocity of lattice Boltzmann solver is used as a reference. The 2-D model is trained against a data set with

$100,000$ samples, with the relative error being less than 3%. The 3-D model is trained against a data set with 0.4 million samples, with the relative error being also lower than 3%.

In [NM20], Nabian *et al.* enchanced the results of Guo *et al.* by penalizing the violation of the divergence-free condition of the velocity, which leads to a physics-regularized loss function:

$$l = ||\mathbf{v}_{\text{pred}} - \mathbf{v}_{\text{true}}||_{l_2}^2 + \lambda||\nabla \mathbf{v}_{\text{pred}}||_{l_2}^2, \tag{2.28}$$

with λ being the regularization rate. Minimizing such a loss function leads to a data-driven surrogate model with embedded physical prior knowledge, and higher prediction accuracy on the test set was reported in the article.

In [TWPH20], Thuerey *et al.* employed a special CAE architecture, called the U-net [RFB15], as surrogate model for Reynolds-averaged Navier-Stokes simulations of airfoil flows. The network takes the inlet velocity and airfoil boundary as inputs, and outputs a 3-D tensor representing the velocity and the pressure fields around the airfoils. The model is trained against a data set with 12.8 steady flow fields, including 1550 different airfoil shapes, a range of Reynolds numbers $Re \in [0.5, 5]$ million and a wide range of attack angles $[-22.5°, 22.5°]$. The prediction error on their test set reaches 2.6% when using a U-net architecture with 30.9 million trainable parameters.

In [NWK20], Wandel *et al.* developed a physics-informed training approach for CNNs, making the obtained model capable of mapping the fluid state from time t to the fluid state at time $t + dt$ in a single forward calculation. This approach is highlighted by the physical loss function, which is defined as:

$$l = \alpha l_d + \beta l_p + \gamma l_b,$$

with the divergence loss l_d, the momentum loss l_p and the boundary loss l_b read:

$$\begin{aligned} l_d &= ||\nabla \mathbf{v}||_{l_2}^2, \\ l_p &= ||\rho\left(\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\mathbf{v}\right) + \nabla p - \mu\Delta\mathbf{v} - f||_{l_2}^2, \\ l_b &= ||\mathbf{v} - \mathbf{v}_b||_{l_2}^2, \end{aligned}$$

where the physical quantities and their gradients are discretized on cartesian grids, and the time derivative is handled by an implicit-explicit scheme. Here α, β and γ are weighting hyper-parameters of loss terms. By minimizing such a loss function, the obtained CNN model thus repects the physical laws described by Navier-Stokes equations, and also the imposed Dirichlet boundary conditions. Furthermore, a pool of the starting fluid states are randomly generated to train the model by minimizing the additioned loss computing by these states, so that the model can be generalized to fluid simulation with arbitrary initial velocity and pressure, making it aware of the evolution of the dynamical system described by the Navier-Stokes equations. The authors generalized this approach to 3-D incompressible fluid simulation in [WWK21].

In [VH20], Viquerat *et al.* trained a CNN for the drag force prediction around arbitrary two dimensional obstacles at a low Reynolds number. The concerned CNN stacks 4 convolution-convolution-pooling blocks, followed by two fully connect layers, and terminated with a linear fully-connected layer of size 1 outputing the predicted drag (see figure 2.19). The authors generated $12,000$ different obstacles through a random algorithm, then resolved the laminar flow around them using a finite element solver, and obtained the drag force. The input of the CNN is an image representing the geometric information of the concerned obstacle. With $296,865$ trainable parameters, the trained model can give accurate drag prediction on unseen obstacles, including NACA airfoils, with a relative error smaller than 2%.

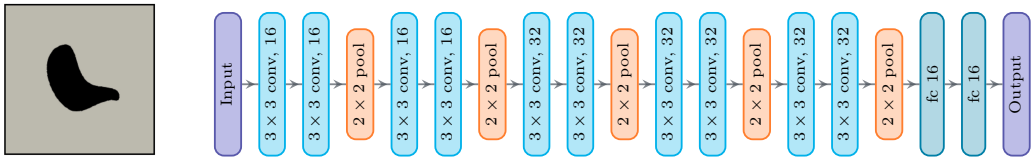


Figure 2.19 | The CNN architecture used for drag prediction in [VH20]. The CNN has 5 repeated conv-conv-pooling blocks. In the first two blocks, 16 kernels are used in the convolutional layers. In the last three blocks, 32 kernels are used. After each block, the image size is divided by 2. The last max-pooling layer is followed by a fully-connected layer with 64 neurons. The architecture is terminated with a linear neuron that outputs the drag prediction.

3 Workflow of Computational Fluid Dynamics

Résumé

Ce chapitre est consacré à la mécanique des fluides numérique classique. Le contenu est organisé en plusieurs sections pour illustrer le cycle complet d'un projet CFD. Les méthodes présentées dans ce chapitre sont utilisées pour générer l'ensemble de données pour les chapitres suivants sur les méthodes d'apprentissage automatique. En tant qu'un phénomène courant dans la vie réelle, l'écoulement externe concerne un fluide passant par un obstacle solide, y compris des avions volant dans l'atmosphère, le vent soufflant au-dessus d'un immeuble, etc. La couche limite fait référence à une fine couche de fluide en contact avec l'interface solide et est reconnue comme la partie la plus intéressante de l'écoulement externe, car elle détermine la force du fluide sur l'obstacle. D'autre part, le caractéristique d'écoulement dans la couche limite dépend de la géométrie, c'est-à-dire des caractéristiques de l'obstacle. Dans ce chapitre, nous générons un ensemble de données d'écoulement externe autour de 2-D obstacles avec différentes caractéristiques géométriques. Reflet de la complexité géométrique, un algorithme est implémenté pour générer $12,000$ formes aléatoires. Des simulations CFD sont ensuite exécutées sur toutes les formes pour obtenir des informations d'écoulement détaillées dans les régimes laminaire et turbulent. L'ensemble de données est ensuite utilisé pour entraîner les réseaux de neurones dans les chapitres suivants.

3.1 Problem setting

The fluid is assumed to be newtonian, like air and water in common industrial situations. The incoming flow has a velocity much smaller than the speed of sound, so the fluid is considered incompressible. The

turbulent regime is attained by controlling the viscosity while fixing the incoming velocity. Since we are pratically interested in the time-averaged flow and the overall impact of the geometric complexity, the turbulence is resolved in the RANS framework, with a chosen turbulence closure presented in section 3.3.2. For both laminar and turbulent flows, the following simplifications are applied: the temperature change is neglected, hence no energy equation is present; the gravity is also neglected for simplicity. By controling and simplifying these factors, we put the very focus on the role of geometry conditions on external flow.

In the first step, n_s random points are drawn in $[0,1]^2$ (the unity is metre), and translated so their center of mass is positioned at $(0,0)$. An ascending trigonometric angle sort is then performed (see figure 3.1a), and the angles between consecutive random points are then computed. An average angle is then computed around each point (see figure 3.1b) using:

$$\theta_i^* = \alpha\theta_{i-1,i} + (1-\alpha)\theta_{i,i+1},$$

with $\alpha \in [0,1]$. The averaging parameter α allows to alter the sharpness of the curve locally, maximum smoothness being obtained for $\alpha = 0.5$.

Then, each pair of points is joined using a cubic Bézier curve, defined by four points: the first and last points, p_i and p_{i+1}, are part of the curve, while the second and third ones, p_i^* and p_i^{**}, are control points that define the tangent of the curve at p_i and p_{i+1}. The tangents at p_i and p_{i+1} are respectively controlled by θ_i^* and θ_{i+1}^* (see figure 3.1c). A final sampling of the successive Bézier curves leads to a boundary description of the shape (figure 3.1d).

The sharpness of the curve features is handled with a positive parameter r that controls the distances $[p_i p_i^*]$ and $[p_{i+1} p_i^{**}]$. For $r = 0$, p_i^* and p_i^{**} respectively coincide with p_i and p_{i+1}, and the curve presents sharp angles at the control points. Intermediate values of r produce smooth curves, with maximal smoothness for $r = 0.5$. When increasing further toward $r = 1$, p_i^* and p_i^{**} tend to coincide with each other, with sharp features to appear near the crossing of the initial and final curve tagents. Finally, for $r > 1$, tangled cases start to appear. A variety of shapes obtained with different values of r can be found in figure 3.2. In the following, r is restricted to the interval $[0,1]$ to avoid tangled shapes.

Using this method, $12,000$ shapes with wide variaty are attained, as some examples shown in figure 3.3.

3.2 Laminar flow

3.2.1 Navier-Stokes equations

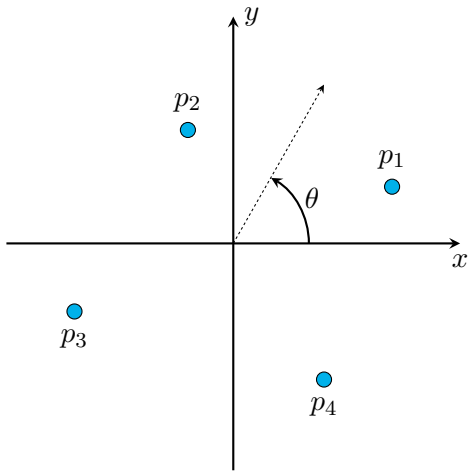
The laminar flow motion of incompressible newtonian fluids is described by the Navier-Stokes (NS) equations:

$$\begin{cases} \rho\left(\partial_t \mathbf{v} + \mathbf{v}\cdot\nabla\mathbf{v}\right) - \nabla\cdot\left(2\eta\boldsymbol{\varepsilon}(\mathbf{v}) - p\mathbf{I}\right) = 0, \\ \nabla\cdot\mathbf{v} = 0, \end{cases} \tag{3.1}$$

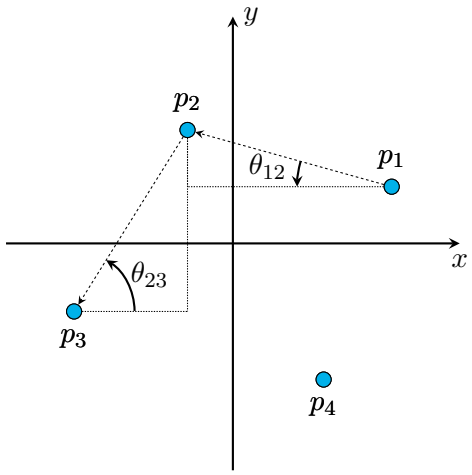
where $t \in [0,T]$ is the time, $\mathbf{v} \in \Omega$ the coordinates, $\mathbf{v}(\mathbf{x},t)$ the velocity, $p(\mathbf{x},t)$ the pressure, ρ the fluid density, η the dynamic viscosity and $\mathbf{I}$ the identity tensor. $\boldsymbol{\varepsilon}(\mathbf{v}) = (\nabla\mathbf{v} + \nabla\mathbf{v}^T)/2$ is the rate-of-strain tensor. $\nabla\cdot\mathbf{v}$ is the expansion rate of the flow, equal to zero here due to the incompressiblty hypothesis.

The Reynolds number of the flow is defined as:

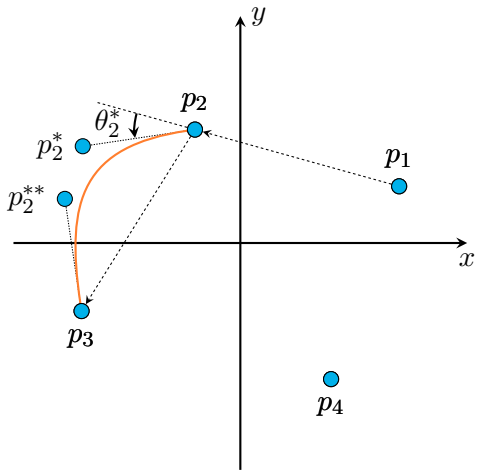
$$Re = \frac{\rho u_0 l}{\eta}, \tag{3.2}$$



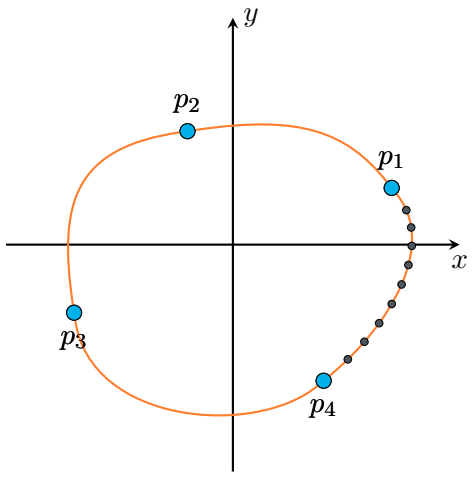
(a) Draw n_s random points, translate them around $(0, 0)$ and sort them by ascending trigonometric angle



(b) Compute angles between random points, and compute an average angle around each point θ_i^*



(c) Compute control points coordinates from averaged angles and generate cubic Bézier curve



(d) Sample all Bézier lines and export for mesh immersion

Figure 3.1 | Random shape generation with cubic Bézier curves.

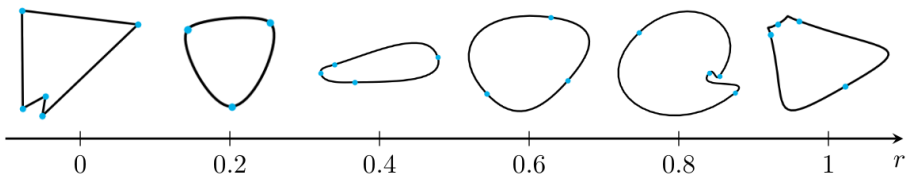


Figure 3.2 | Random shape examples depending on their r value, ranging from 0 to 1. The random points are shown in blue, and their number n_s ranges from 3 to 5, although it is possible to use more. For $r = 0$, one sees the sharp features of the curve on the Bézier points. For intermediate values, smooth curves are obtained. Finally, for values close to 1, sharp features start to appear around the control points (not shown here)

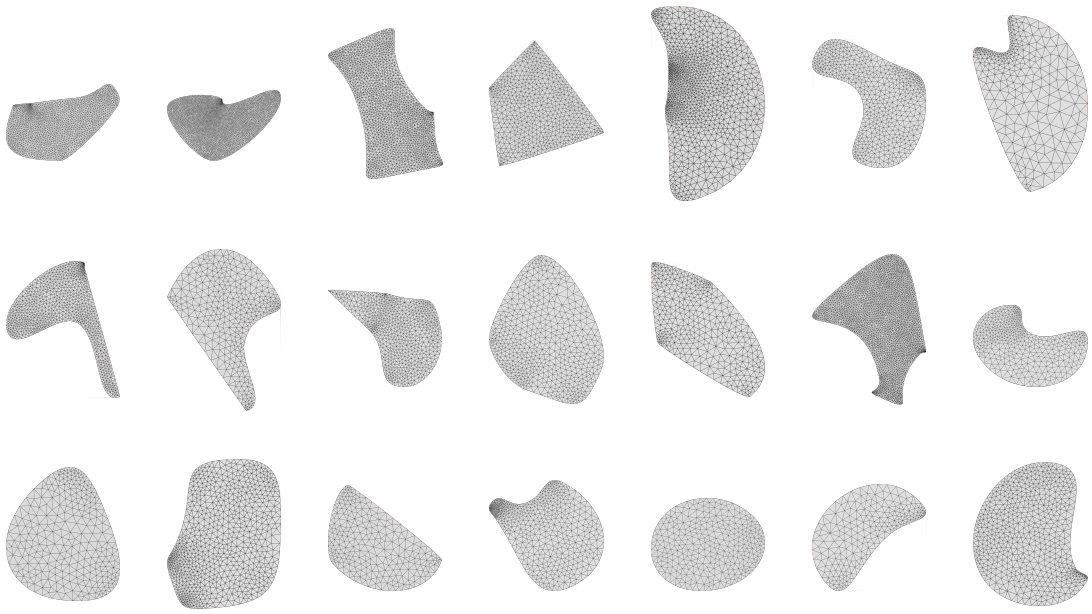


Figure 3.3 | Shape examples drawn from the dataset. A wide variety of shape is obtained using a restrained number of points ($n_s \in [4, 6]$), as well as the local curvature r and averaging parameter α.

where $u_0 = 1\,\mathrm{m/s}$ is the inlet speed of the principle flow, l the characteristic length of the concerned geometry. When the fluid's physical properties are pre-defined, the Reynolds number is totally determined by the geometric characteristics of the studied obstacle. For laminar flow, the density and the dynamic viscosity are set to $1\,\mathrm{kg/m^3}$ and $0.1\,\mathrm{kg/(m \cdot s)}$. The Reynolds number is hence around 10 across the data set.

The obstacle center is placed at the origin of a rectangular domain $[-5, 10]\mathrm{m} \times [-5, 5]\mathrm{m}$, as shown in figure 3.4. The inflow boundary conditions are $\mathbf{v} = (u_0, 0)$. For the lateral boundaries, symmetry conditions $\frac{\partial u}{\partial y} = v = 0$ are used. For the outflow, $\frac{\partial u}{\partial x} = \frac{\partial v}{\partial x} = 0$ together with $p = 0$ are prescribed, and no slip conditions $\mathbf{v} = (0, 0)$ are imposed on the solid surface.

3.2.2 Immersed boundary method for interface description

Classically, the resolution of NS equations around solid obstacle relies on body-fitted methods, where the mesh boundary follows the geometry of the obstacle. These methods require the generation of a full mesh (domain and obstacle) for each computation, which can be both time- and memory-consuming (see figure 3.5a). To overcome this issue, immersed methods based on a unified formulation were introduced that propose to immerge a boundary mesh of the obstacle in a background mesh (see figure 3.5b).

The first step is to compute the signed distance function (level set) [BDC09] of the given geometry to each node of the background mesh:

$$\alpha(\mathbf{x}) = \pm d(\mathbf{x}, \Gamma_{im}), \forall \mathbf{x} \in \Omega, \tag{3.3}$$

with the following sign convention: $\alpha \geq 0$ inside the solid domain defined by the interface Γ_{im}, and $\alpha \leq 0$ outside this domain. Using this function, the fluid-solid interface Γ_{im} is easily identified as the zero iso-value of function α:

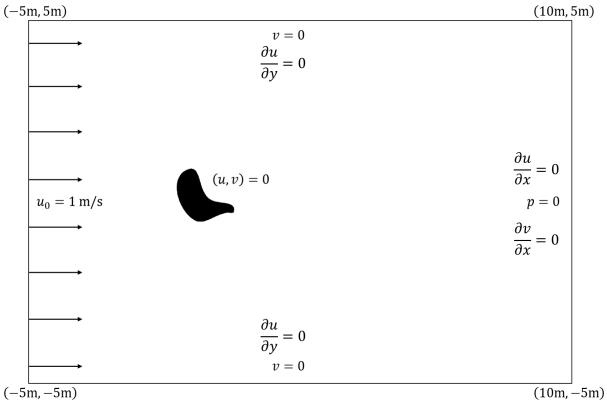
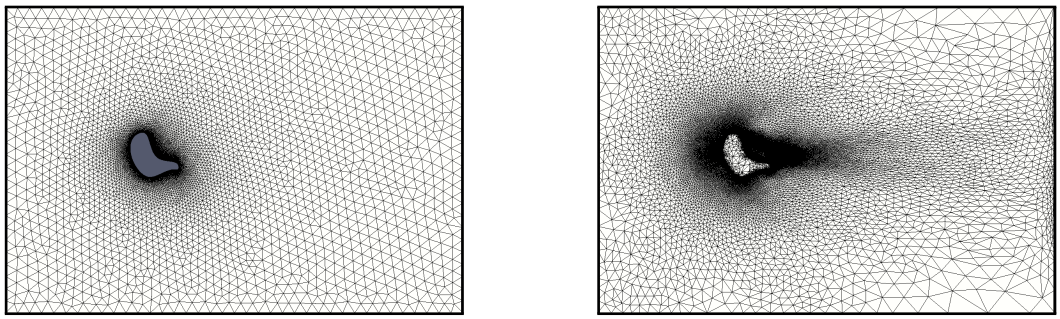


Figure 3.4 | Computation domain and boundary condition. The random shapes are always inside a 1m $\times$ 1m box with the geometry center placed at the origin. The incoming flow has a uniform velocity $u_0 = 1\text{m/s}$. The dynamic viscosity of the laminar flow is $0.1\ \text{kg}/(\text{m} \cdot \text{s})$

.



(a) Body-fitted mesh **(b)** Immersed mesh

Figure 3.5 | Comparison of two different methods for interface representation. The body-fitted mesh requires mesh generation when the studied geometry is changed. The immersed method requires a background mesh for the domain and a mesh for the obstacle. The CFD solver is equiped with a mesh adaptation technic to refine the elements around the solid interface. To run CFD simulations on all the 12 000 shapes, one just need one fixed background mesh while changing the obstacle mesh sequentially.

$$\Gamma_{im} = \{\mathbf{x} \in \Omega, \alpha(\mathbf{x}) = 0\}. \tag{3.4}$$

As explained above, the signed distance function is used to localize the interface of the immersed structure, but it is also used to initialize the desirable properties on both sides of the latter. Indeed, for the elements crossed by the level-set functions, fluid-solid mixtures are used to determine the element effective properties. To do so, a Heaviside function $H(\alpha)$ is defined as follows:

$$H(\alpha) = \begin{cases} 1 \text{ if } \alpha > 0, \\ 0 \text{ if } \alpha < 0, \end{cases} \tag{3.5}$$

With the heaviside equation, one can easily write the unified governing equations for both the fluid phase and solid phase:

$$\begin{cases} \rho^* \left(\partial_t \mathbf{v} + \mathbf{v} \cdot \nabla \mathbf{v}\right) - \nabla \cdot \left(2\eta \boldsymbol{\varepsilon}(\mathbf{v}) + \boldsymbol{\tau} - p\mathbf{I}\right) = 0, \\ \nabla \cdot \mathbf{v} = 0, \end{cases} \tag{3.6}$$

with the mixed quantities defined by:

$$\rho^* = H(\alpha)\rho_s + (1 - H(\alpha))\rho_f,$$
$$\boldsymbol{\tau} = H(\alpha)\boldsymbol{\tau}_s.$$

The subscripts f and s refer respectively to the fluid and to the solid. In the latter equality, $\boldsymbol{\tau}$ acts as a Lagrange multiplier that yields $\boldsymbol{\varepsilon}(\mathbf{v}) = 0$ in the solid [HFCC13]. The boundary conditions of the modified governing equations are the same as described in section 3.2.1, with the no slip conditions on the solid surface extended to nodes inside the obstacles.

3.2.3 Numerical resolution

The modified equations 3.6 are casted into a stabilized finite element formulation, and solved using a variational multiscale (VMS) solver implemented in CimLib [HFCC13]. The CFD solver used is equipped with a remeshing technique [JHVC15] able to track both (i) the fluid/solid interfaces, and (ii) the areas of strong gradients. Based on the laminar nature of the concerned flow, we set the mesh size in the boundary layer to 0.01m. The target number of elements is $100\,000$. This method, exploited in conjunction with mesh immersion, ensures accurate results for the creation of the data set. In practice, a mesh adaptation is applied every 10 iterations, with the time step being 0.5s, and the mesh and flow solutions converge after 40 iterations (see figure 3.6). Given the situation (multiple cheap 2-D simulations), each CFD run was processed on a single core, with 64 shapes running at the same time on Intel Xeon 2.6 GHz cores. The average computation time was 4.8 minutes, and the whole data set was generated in less than 24 hours. The physical computational time was chosen large enough so that the stationary flow was established. A sample of the converged flow fields are shown in figure 3.7.

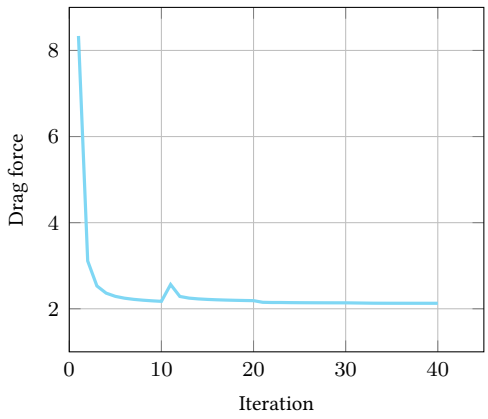
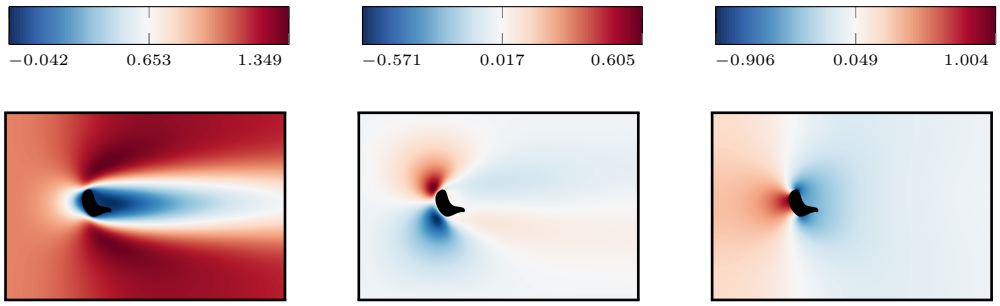


Figure 3.6 | The convergence of drag force (N). The Reynolds number is around 10, which signifies essentially laminar flow in the computational domain, and it takes only 40 iterations for the convergence.



(a) Velocity field - x component **(b)** Velocity field - y component **(c)** Pressure field

Figure 3.7 | Converged laminar velocity field and pressure field obtained from the CFD solver. The converged mesh has been shown in figure 3.5b. The mesh adaptation is applied every 10 iterations. After a total of 40 iterations, the flow fields converge to their stable states.

3.3 Turbulent flow

In this section the Reynolds number is controled by the dynamic viscosity μ, while keeping the inlet velocity u_0 being 1m/s, so that the incompressibility condition is always valid. In order to generate turbulence, the Reynolds number must be high enough, but the required minimum value varies case by case. However, we have a reference in section 1.1, where the turbulent velocity snapshot around a cylinder is shown in figure 1.1b, with the Reynolds number being 1.5×10^5. Accordingly, we choose a bigger value, $Re = 1 \times 10^6$, for the external flow in this section, which is certainly high enough to generate turbulence around the obstacles.

3.3.1 Reynolds-averaged Navier-Stokes equations

In the context of turbulent flows, velocity and pressure experience intense multiscale perturbations in both time and space, leading to an increasing need of computational resources to resolve all the existing time and space scales. From an engineering point of view, the time-averaged flow pattern is also interesting and sufficient for many real-life problems. To this end, the so called Reynolds-averaged Navier–Stokes equations (RANS) can be resolved to directly obtain the time-averaged velocity and pressure fields.The RANS equations are derived by the Reynolds decomposition of the velocity and pressure [Rey95]:

$$\begin{aligned} \mathbf{v}(\mathbf{x}, t) &= \overline{\mathbf{v}}(\mathbf{x}, t) + \mathbf{v}'(\mathbf{x}, t), \\ p(\mathbf{x}, t) &= \overline{p}(\mathbf{x}, t) + p'(\mathbf{x}, t), \end{aligned} \tag{3.7}$$

where the mean field components are defined as time averages of instantaneous fields:

$$\begin{aligned} \overline{\mathbf{v}}(\mathbf{x}, t) &= \lim_{T \to \infty} \frac{1}{T} \int_{t_0}^{t_0+T} \mathbf{v}(\mathbf{x}, t)\, dt, \\ \overline{p}(\mathbf{x}, t) &= \lim_{T \to \infty} \frac{1}{T} \int_{t_0}^{t_0+T} p(\mathbf{x}, t)\, dt, \end{aligned} \tag{3.8}$$

The time average of the fluctuating components $\mathbf{v}'(\mathbf{x}, t)$ and $p'(\mathbf{x}, t)$ is thus zero by definition. By taking the time average of the original Navier-Stokes equations, one gets:

$$\begin{cases} \rho \left(\partial_t \overline{\mathbf{v}} + \overline{\mathbf{v}} \cdot \nabla \overline{\mathbf{v}} \right) - \nabla \cdot \left(2\eta \boldsymbol{\varepsilon}(\overline{\mathbf{v}}) - \rho \mathbf{R} - \overline{p} \mathbf{I} \right) = 0, \\ \nabla \cdot \overline{\mathbf{v}} = 0, \end{cases} \tag{3.9}$$

where the Reynolds stress tensor $\mathbf{R}_{ij} = \overline{\mathbf{v}_i \mathbf{v}_j}$ arises from the nonlinear convection term $\mathbf{v} \cdot \nabla \mathbf{v}$. In their present form, the equations (3.9) are unclosed due to the presence of the Reynolds stress tensor. In order to resolve the time-averaged velocity and pressure, one needs an additional relation to close the system using the mean flow variables $\overline{\mathbf{v}}$ and $\overline{p}$.

3.3.2 Spalart-Allmaras model

The first solution to the closure problem is attributed to Joseph Valentin Boussinesq [Bou77, Sch07] in the year of 1877. The Boussinesq's hypothesis relates the Reynolds stress tensor with the mean flow strain rate by:

$$-\rho \mathbf{R} = 2\eta_t \boldsymbol{\varepsilon}(\overline{\mathbf{v}}) - \frac{2}{3} \rho k \mathbf{I}, \tag{3.10}$$

with $\boldsymbol{\varepsilon}(\mathbf{v}) = (\nabla\overline{\mathbf{v}} + \nabla\overline{\mathbf{v}}^T)/2$ being the rate-of-strain tensor of the mean flow, and $k = \frac{1}{2}tr(\mathbf{R}) = \frac{1}{2}\overline{\mathbf{v}_i\mathbf{v}_i}$ being the turbulence kinetic energy. The quantity $\eta_t > 0$ is called the eddy viscosity, and need to be further modeled. By inserting the equation (3.10) into the original RANS, one gets the following system:

$$\begin{cases} \rho\left(\partial_t\overline{\mathbf{v}} + \overline{\mathbf{v}}\cdot\nabla\overline{\mathbf{v}}\right) - \nabla\cdot\left(2(\eta+\eta_t)\boldsymbol{\varepsilon}(\overline{\mathbf{v}}) - \overline{p}'\right) = 0, \\ \nabla\cdot\overline{\mathbf{v}} = 0, \end{cases} \tag{3.11}$$

where the turbulent kinetic energy is absorbed into the modified pressure $\overline{p}'$. In the following paragraphs, the mean flow variables $\overline{\mathbf{v}}$ and $\overline{p}'$ are denoted by $\mathbf{v}$ and p to simplify the notations. The equations (3.11) are thus rewritten as:

$$\begin{cases} \rho\left(\partial_t\mathbf{v} + \mathbf{v}\cdot\nabla\mathbf{v}\right) - \nabla\cdot\left(2(\eta+\eta_t)\boldsymbol{\varepsilon}(\mathbf{v}) - p\right) = 0, \\ \nabla\cdot\mathbf{v} = 0. \end{cases} \tag{3.12}$$

The eddy viscosity restrains the turbulence through a larger effective viscosity $\eta + \eta_t$. The resolved velocity and pressure from equations (3.12) thus converge to a stable state, which represents the time-averaged flow. The Spalart-Allmaras (SA) model [SA94] is one of the many eddy viscosity-based models. Being designed for aerodynamic flows, it is an adequate model for the computation of turbulent flow around random shapes. It requires to solve a transportation equation to obtain the eddy viscosity field $\eta_t(\mathbf{x}, t)$ at each time step. The concerned equation modelize a working variable $\tilde{\nu}$ as:

$$\frac{\partial\tilde{\nu}}{\partial t} + \mathbf{v}\cdot\nabla\tilde{\nu} - c_{b1}(1-f_{t2})\tilde{S}\tilde{\nu} + \left[c_{w1}f_w - \frac{c_{b1}}{\kappa^2}f_{t2}\right]\left(\frac{\tilde{\nu}}{d}\right)^2 - \frac{c_{b2}}{\sigma}\nabla\tilde{\nu}\cdot\nabla\tilde{\nu} - \frac{1}{\sigma}\nabla\cdot[(\nu+\tilde{\nu})\nabla\tilde{\nu}] = 0, \tag{3.13}$$

with the kinematic viscosity $\nu = \eta/\rho$. The eddy viscosity is retrieved as: $\eta_t = \rho\tilde{\nu}f_{v1}$, with:

$$f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3}, \quad \chi = \frac{\tilde{\nu}}{\nu}, \quad f_{v2} = 1 - \frac{\chi}{1+\chi f_{v1}}, \quad f_{t2} = c_{t3}e^{-c_{t4}\chi^2},$$

$$f_w = g\left[\frac{1+c_{w3}^6}{g^6+c_{w3}^6}\right]^{\frac{1}{6}}, \quad g = r + c_{w2}(r^6 - r), \quad r = \min\left(\frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2}, 10\right),$$

$$\tilde{S} = S + \frac{\tilde{\nu}}{\kappa^2 d^2}f_{v2}, \quad S = \sqrt{2\boldsymbol{\Omega}(\mathbf{v}) : \boldsymbol{\Omega}(\mathbf{v})},$$

where d is the shortest distance to the solid interface, $\kappa = 0.4$ is the von Kármán constant, $\boldsymbol{\Omega}(\mathbf{v}) = (\nabla\mathbf{v} - \nabla\mathbf{v}^T)/2$ is the rotation tensor, and the remaining model coefficients read:

$$c_{b1} = 0.1355, \quad c_{b2} = 0.622, \quad \sigma = 2/3, \quad c_{w1} = \frac{c_{b1}}{\kappa^2} + \frac{1+c_{b2}}{\sigma},$$

$$c_{w2} = 0.3, \quad c_{w3} = 2, \quad c_{t3} = 1.2, \quad c_{t4} = 0.5.$$

The equations (3.12) and (3.13) are solved simultaneously to obtain the flow variables $(\mathbf{v}, p, \tilde{\nu})$. Proper boundary conditions must be imposed on the working variable $\tilde{\nu}$, which will be presented in the next subsection. In case of non-physical negative eddy viscosity recovered by the SA model, the field of η_t should be clipped to reserve the non-negative part. However, a variant [AJ12] of the standard SA model, called the negative SA model, was developed to handle this issue more delicately. The equation (3.13) is replaced when $\tilde{\nu}$ is negative by:

$$\frac{\partial \tilde{\nu}}{\partial t} + \mathbf{v} \cdot \nabla \tilde{\nu} - c_{b1}(1 - c_{t3}) S \tilde{\nu} - c_{w1} \left(\frac{\tilde{\nu}}{d} \right)^2 - \frac{c_{b2}}{\sigma} \nabla \tilde{\nu} \cdot \nabla \tilde{\nu} - \frac{1}{\sigma} \nabla \cdot [(\nu + f_n \tilde{\nu}) \nabla \tilde{\nu}] = 0, \tag{3.14}$$

with $f_n = (c_{n1} + \chi^3)/(c_{n1} - \chi^3)$ and $c_{n1} = 16$. In addition, the turbulent eddy viscosity η_t is set to zero in the regions where the computed value of $\tilde{\nu}$ is negative.

3.3.3 Computational domain and discretization

Since turbulent flows display more complex patterns than laminar ones, the computational domain must be large enough to let the turbulence fully develop. A $[-10, 20]\text{m} \times [-10, 10]\text{m}$ rectangular domain is used to reduce the impact of far-field conditions, as is shown in figure 3.8. The center of the obstacles are always placed at the origin. The boundary conditions for velocity $\mathbf{v}$ and pressure p remain unchanged compared to the laminar flow case. As for $\tilde{\nu}$, no slip condition $\tilde{\nu} = 0$ is imposed on the solid interface, and the inflow boundary condition is set to $\tilde{\nu} = 3\nu$. Fot lateral boundaries, we use the symmetry condition $\frac{\partial \tilde{\nu}}{\partial y} = 0$. For the outflow, $\frac{\partial \tilde{\nu}}{\partial x} = 0$ is prescribed.

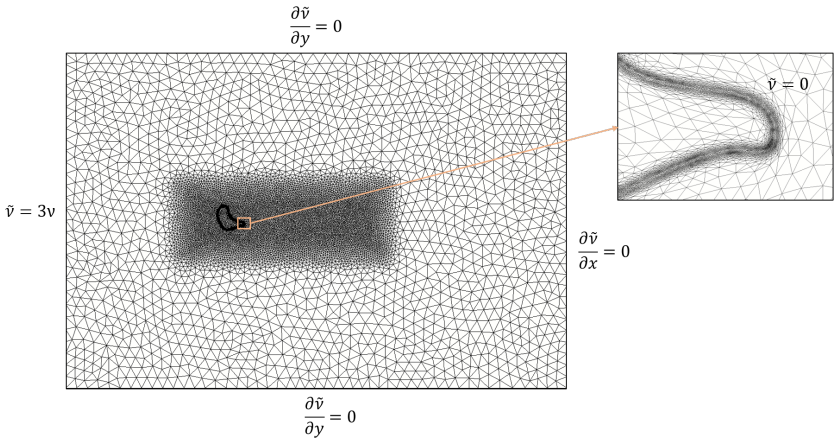


Figure 3.8 | Computational domain and mesh for turbulence simulation around random shapes. The domain is larger than that of laminar flow for the complexity of turbulence. Only the boundary conditions for $\tilde{\nu}$ are indicated as the velocity and pressure follow the same constraints as in laminar flow. The pre-adapted mesh is used for all the 5 Reynolds numbers, with the boundary element size being 5×10^{-5}m.

The dynamic mesh adapatation is replaced by pre-adapted meshes to reduce the resolution time. Such a mesh can be visibly divided into 3 zones (see figure 3.8):

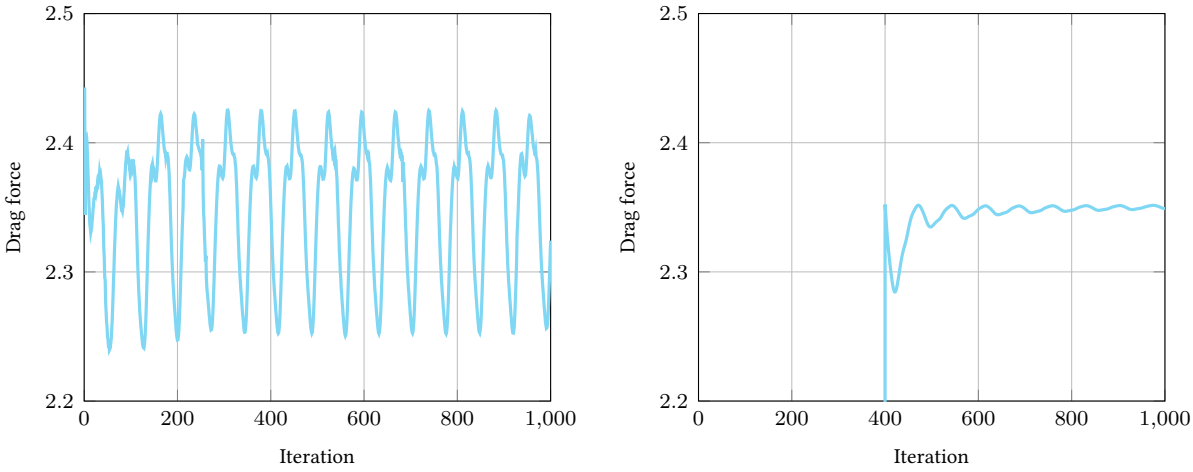
- An outer zone for the main flow.
- A refined zone for the flow close to the obstacles.
- A high-resolution zone for the boundary layer.

The element size at the four sides is around 0.7m, while on the solid interface the element is as small as 5×10^{-5}m, which is enough for turbulent simulation at Re$= 10^6$.

3.3.4 Numerical resolution

The simulation is run from $t = 0$ with a time step being 0.1s. The drag force is surveilled during the simulation, and it is observed that $1,000$ iterations is a good balance for the convergence and computational

time. In practice, the resolved flow variables do not converge to a steady state, but become periodic after several hundreds of iterations (see figure 3.9). In order to obtain a steady flow pattern for every shape, the resolved flow fields are manually averaged over the last 600 iterations (see figure 3.10), because the first 400 iterations are enough for the flow entering the periodic regime. To be more precise, the averaged drag force in figure 3.9b converges very fast to a stable value, thus confirming that an average window of 600 iterations is sufficient.

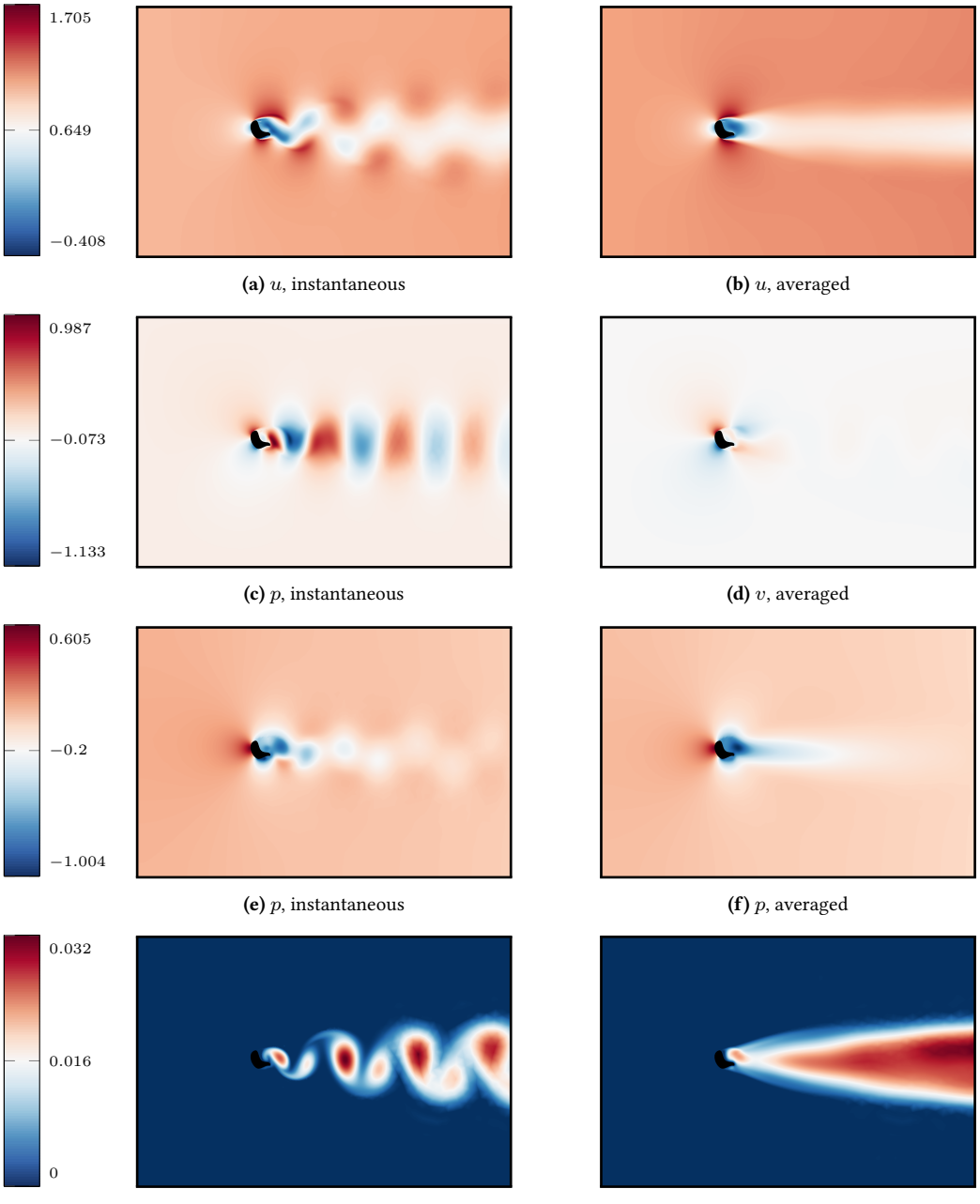


(a) Drag force (N)

(b) Averaged drag force (N)

Figure 3.9 | Drag force on a random shape during the simulation. The flow enters a periodic pattern fastly. In order to obtain a stable flow field at the end of simulation, the flow is averaged since the 400th iteration.

Given the situation (multiple expensive 2-D turbulence simulations), only $2,000$ out of the total $12,000$ random shapes are studied, with each CFD run being processed on 28 Intel Xeon 2.6 GHz cores. Two shapes are simulated at the same time, taking about 3 minutes to finish the simulation. The whole data set was generated in 2 days. A sample of the converged flow fields are shown in figure 3.11. For a better visualization, a closer view focusing on the obstacle is provided, with the window being $4\text{m} \times 6\text{m}$.



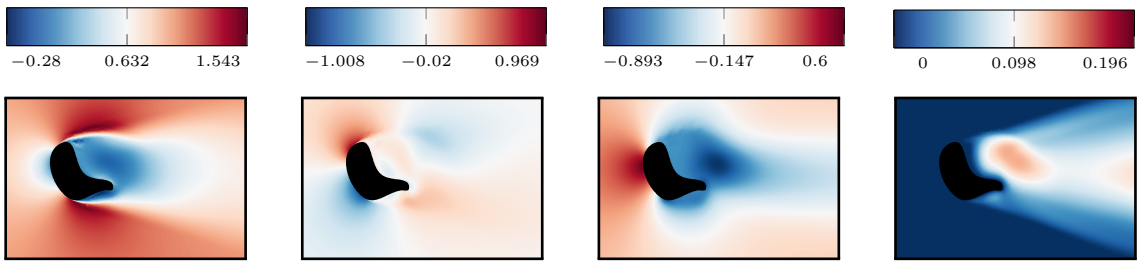
(a) u, instantaneous **(b)** u, averaged

(c) p, instantaneous **(d)** v, averaged

(e) p, instantaneous **(f)** p, averaged

(g) η_t, instantaneous **(h)** η_t, averaged

Figure 3.10 | Turbulent velocity (u, v) and pressure p ($Re = 1 \times 10^6$) around an obstacle. The obstacle is colored in black for a better visulization. The instantaneouos snapshots (on the left) are taken at the end of $1,000$th iterations. The snapshots from the 400th to the $1,000$th iterations are averaged as shown on the right.



(a) u, Re$= 10^6$ **(b)** v, Re$= 10^6$ **(c)** p, Re$= 10^6$ **(d)** η_t, Re$= 10^6$

Figure 3.11 | Averaged turbulent ($Re = 1 \times 10^6$) flow fields $(\mathbf{v}, p, \eta_t)$ around a random shape. The obstacle is colored in black for a better visulization.

4

Convolutional neural networks for fast flow prediction

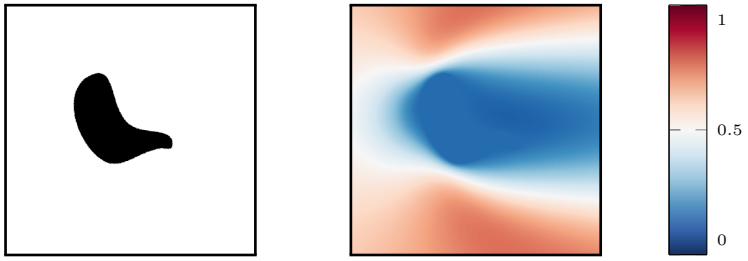
Résumé

Ce chapitre est consacré à l'exploration des applications CNN pour l'estimation de flux autour d'obstacles 2-D. En raison du coût important des solveurs aux EDP traditionnels, la prédiction de flux rapide est devenue un objectif majeur pour les applications industrielles. Dans le design industriel, comme l'optimisation de forme par exemple, les équations de Navier-Stokes doivent être résolues à chaque modification de forme, jusqu'à ce que la géométrie optimale soit trouvée. Dans le cadre de l'apprentissage supervisé, un CNN peut être entraîné contre un ensemble de données ad-hoc, afin d'obtenir un modèle subrogé capable de prédire des quantités d'intérêt, accélérant ainsi remarquablement la conception. Les solveurs aux EDP standard, quant à eux, utilisent des algorithmes itératifs pour calculer les champs de flux satisfaisant aux contraintes physiques. Par conséquent, à condition d'avoir un ensemble de données adéquat et une phase de formation, les modèles subrogés basés sur CNN pourraient considérablement accélérer ces phases de conception.

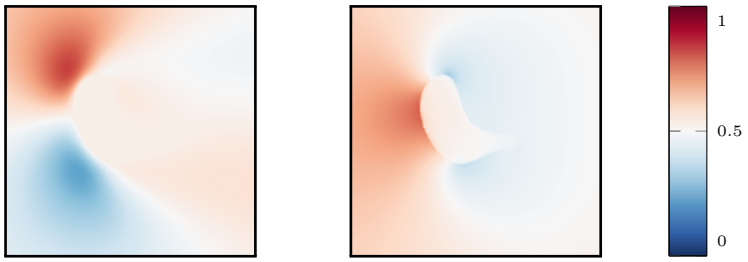
4.1 Laminar flow prediction

4.1.1 data set

In this chapter, a subset of the data set presented in section 3.2 is used. As a reminder, we consider the velocity and pressure fields of 2,000 random shapes at Reynolds 10, to be fitted by a convolutional neural network. The flow fields are saved in triangular meshes, each of which has about 30 000 nodes. In order



(a) Network input **(b)** Velocity field - x component



(c) Velocity field - y component **(d)** Pressure field

Figure 4.1 | The obstacle, velocity field and pressure field for a dataset element. The geometric information is encoded into a binary field, with the solid obstacles being 0, and the non-solid domain being 1. The binary input 4.1a is the input of the CNN. The interpolated velocity field (4.1b, 4.1c) and pressure field (4.1d) make a $401 \times 401 \times 3$ tensor as the output of the CNN. The solid obstacle is not masked in the flow fields.

to be compatible with CNNs, the latter must be interpolated on cartesian grids. With a grid size equal to 0.01m (which is the smallest element size in the triangular meshes), the 10m $\times$ 15m computational domain would be transformed to a 2-D array with 1000×1500 pixels, and the entire data set would require a tremendous amount of memory. Hence, we focus on a cropped domain in the vicinity of the solid obstacles. A 4m $\times$ 4m square window, centered on the obstacles, is adopted to generate the data set used in this chapter. With the aforementioned grid size, the selected window is represented by a 401×401 2-D array. In the data set, each instance holds 4 components, as shown in figure 4.1: the obstacle, the horizontal velocity, the vertical velocity, and the pressure. Finally, the whole data set with $2,000$ instances requires 13.5 GB of storage space.

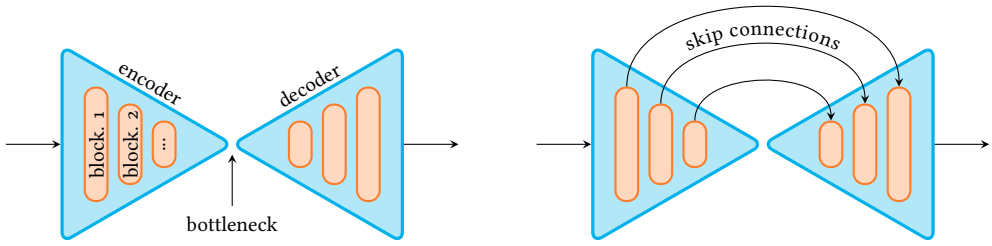
The physical flow fields are normalized and non-dimensionalized before being used to train the CNN. Technically, the normalized velocity and pressure fields $(\overline{u}, \overline{v}, \overline{p})$ can be written as:

$$\begin{aligned} \overline{u} &= (u - u_{\text{min}})/(u_{\text{max}} - u_{\text{min}}), \\ \overline{v} &= (v - v_{\text{min}})/(v_{\text{max}} - v_{\text{min}}), \\ \overline{p} &= (p - p_{\text{min}})/(p_{\text{max}} - p_{\text{min}}), \end{aligned} \tag{4.1}$$

where the maximum and minimum values are taken over the whole data set. The fluid density $\rho = 1\text{kg/m}^3$ and the inlet velocity $u_0 = 1\text{m/s}$ are used for non-dimensionalization, which does not change the values of the velocity and pressure.

4.1.2 CNN architecture

Convolutional autoencoders (CAEs) have been frequently used for full-field flow predictions in the recent years. For example, Guo *et al.* [GLI16] trained CAEs for steady flow prediction around 2-D and 3-D obstacles, while Jin *et al.* [JCCL18] and Lee *et al.* [LY19] trained CAEs for unsteady flow prediction



(a) Convolutional autoencoder architecture

(b) U-net architecture

Figure 4.2 | Sketch of convolutional autoencoder and U-net architectures. Standard CAEs (4.2a) are composed of an encoder and a decoder paths, and can be exploited either for end-to-end regression tasks (in a supervised way, with labels), or for the inference of latent space representations (in an unsupervised way, without labels). U-net autoencoders (4.2b) are a specific class of CAE, in which skip connections are added from the encoder branch to the decoder one in order to mix high-level features from the latent space with low-level one from the contractive path. They usually present a superior level of performance on regression tasks.

around cylinders, *i.e.* to predict the fluid state at the next time step from the the present fluid state. In this section, a variant of CAE, called the U-net, is used for laminar flow prediction around random 2-D shapes. Initially introduced by Ronneberger *et al.* [RFB15] in the field of biomedical image segmentation, they were recently exploited for flow prediction tasks [TWPH20, FFT19], where they were either traine to preidct RANS-solved turbulent data or for super-resolution reconstruction of turbulent flows. U-nets have an autoencoder-like structure, but additionally introduce skip connections between the corresponding convolutional and transposed convolutional layers from the encoder and decoder paths, in order to mix high-level features from the latent space with low-level ones from the encoder. A sketch of such architectures is proposed in figure 4.2.

The U-net architecture exploited in this chapter is composed of a number of convolution-convolution-maxpooling blocks with 3×3 convolutional filters and 2×2 pooling size for the encoder part, and 5 transposed convolution-convolution-convolution blocks with 2×2 transposed convolutional filters and 3×3 convolutional filters for the decoder part. In the encoder branch, the amount of convolutional kernels is doubled after each block, while in the decoder, the amount of kernels is halved after each block. The U-net's architecture is hence symmetric, and scalable through a modification of the number of kernels in the first block. An architecture with 3 convolution-convolution-maxpooling blocks is shown in figure 4.3.

In terms of network architecture, a U-net has two crucial hyper-parameters:

- The number of blocks b in the encoder, which is the same as the number of blocks in the decoder due to the symmetry. It is easy to see the latent feature tensor's size is determined by b: $\frac{h}{2^b} \times \frac{w}{2^b} \times 2^b m$, where h and w are the height and width of the input array, and m is the number of kernels in the first convolutional layer. With more blocks, there are also more skip connections from the encoder to the decoder.
- The number of convolutional kernels m in the first layer. Together with b, it determines the number of trainable parameters due to the architecture's scalability.

The impact of b and m on the U-net's performance will be studied in the section 4.1.4. Other hyper-parameters, including the kernel size, the stride size and the activation function, also have their impacts on the performance. However, they are chosen and fixed following recommendations from the litterature. The details would be presented in the next subsection, together with the optimization aspects of the network training.

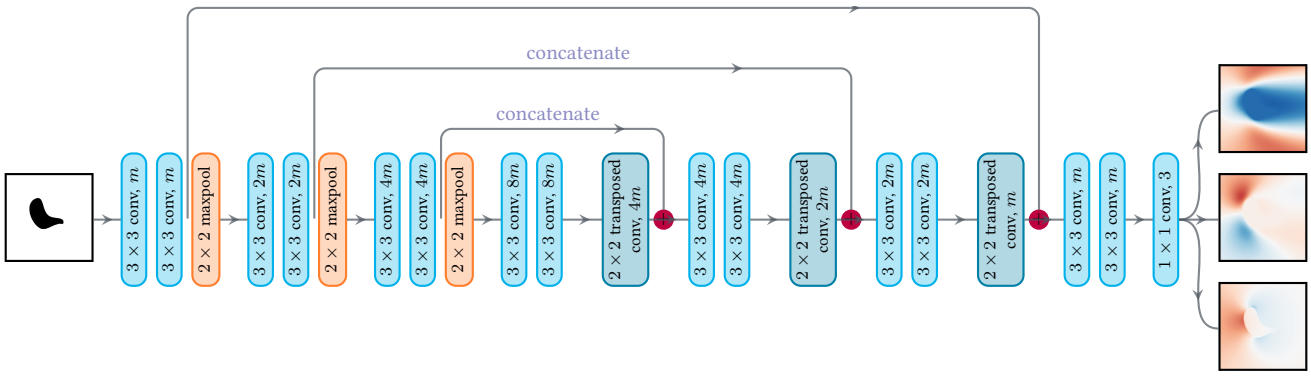


Figure 4.3 | The U-net architecture with 3 convolution-convolution-maxpooling blocks in the encoder. The input image is first passed to 2 consecutive convolutional layers, both with m 3×3 kernels. Then a 2×2 max pooling layer downsamples the feature map. The stride size is 2 in all the layers. By doubling the number of kernels in deeper convolutional layers, one can extract more high-level features which are saved in different channels. The skip connections provide features at different levels to the decoder, which enforce the geometric characteristics in the flow prediction. Finally a linear convolutional layer with 3 kernels compute the output velocity and pressure.

4.1.3 Traininig and evaluation

As a first appraoch, an architecture with 5 blocks in the encoder is trained and evaluated as a proof of concept. With 8 kernels in the first convolutional layer, the network has $1,944,763$ trainable parameters. The U-net is implemented and trained in the Keras API [C+15] with the following configurations: neurons are activated by rectified linear unit (ReLU), batch size equals to 64, weights and bias are initialized by the Glorot normal method, optimized by the Adam optimizer with an initial learning rate equals to 0.001, a batch-wise learning rate decay equals to 0.0002, and early stopping patience is 20 epochs. The loss function is the mean absolute error (MAE), computed as follows:

$$l\left(y^{\text{pred}}, y^{\text{ref}}\right) = \frac{1}{3 \times 401 \times 401} \sum_{\substack{1 \le i \le 3, \\ 1 \le j,k \le 401}} |y^{\text{pred}}_{ijk} - y^{\text{ref}}_{ijk}|, \tag{4.2}$$

with y^{pred} and y^{ref} being the predicted and referential flow fields. The data set is divided to three parts for training, validation and test, with the split ratio being $0.8 : 0.1 : 0.1$. In this manuscript, the loss on a data set, including the validation set or a mini-batch in the training set, refers to the averaged loss values of the instances in the data set. The model is trained 5 times with random initializations, and the one with smallest validation loss is selected, with its training history shown in figure 4.4. The training ceases after 237 epochs, taking $3,630$ seconds on a Nvidia Tesla V100 GPU card, and finally attains a validation loss being 5.7×10^{-3}. Evaluating the selected model on the test set, we observe a test loss equal to 6.03×10^{-3}. By having a closer look at the distribution of the loss on the test set (with 200 shapes), as is shown in figure 4.5, we conclude that the trained model has good performance on the unseen data in the test set. In terms of prediction cost, it takes about 2 seconds to obtain the flow fields around the 200 shapes in the test set, which marks a considerable speedup over the traditional PDE solvers.

A sample prediction is shown in figure 4.6. The prediction on this sample is globally of high quality, with an MAE equals to 7.09×10^{-3}. The flow pattern around the obstacle is reasonable, but visible discrepancies are concentrated near the solid interface. Indeed, the U-net not only predicts the velocity and pressure fields, but also has to reconstruct the obstacle boundary, which cannot be strictly restored due to the discontinuity and the staircasing effect due to the use of a cartesian grid.

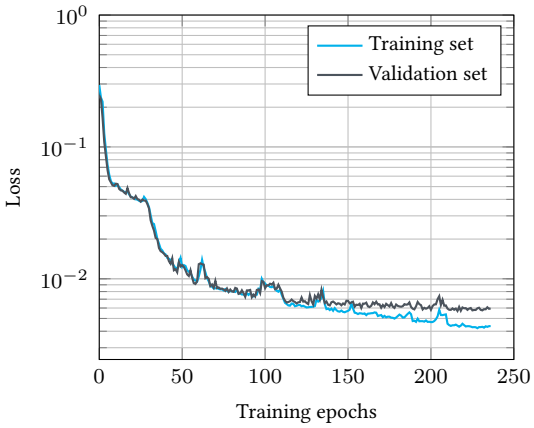


Figure 4.4 | Training history of the U-net model. The training ceases in few than 250 epochs, in order to avoid overfitting.

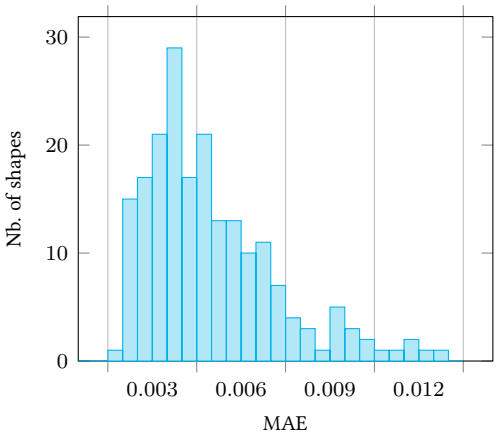
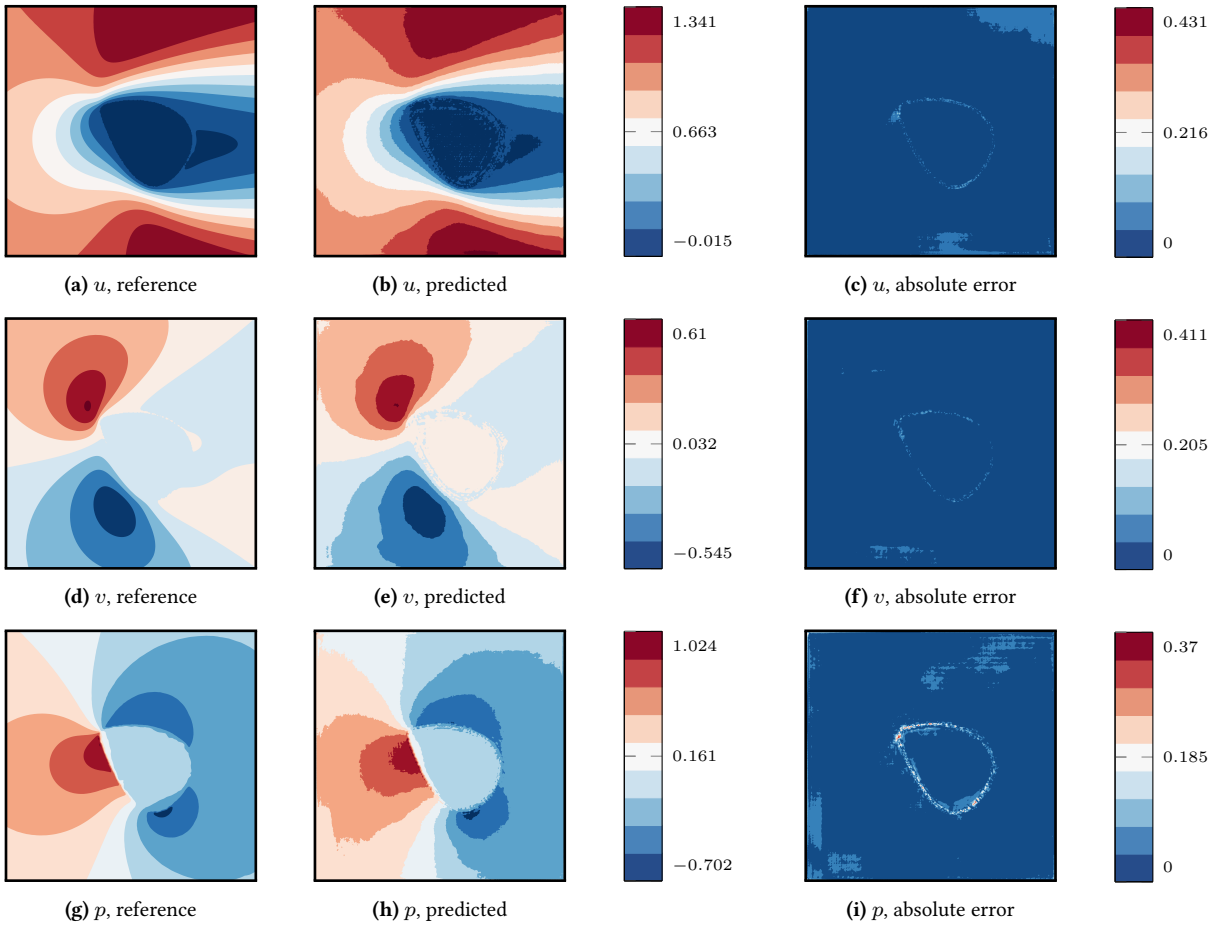


Figure 4.5 | Distribution of the MAE over the test set. The test set contains 200 random Bézier shapes. Only 17 shapes present an MAE superior to 0.01.

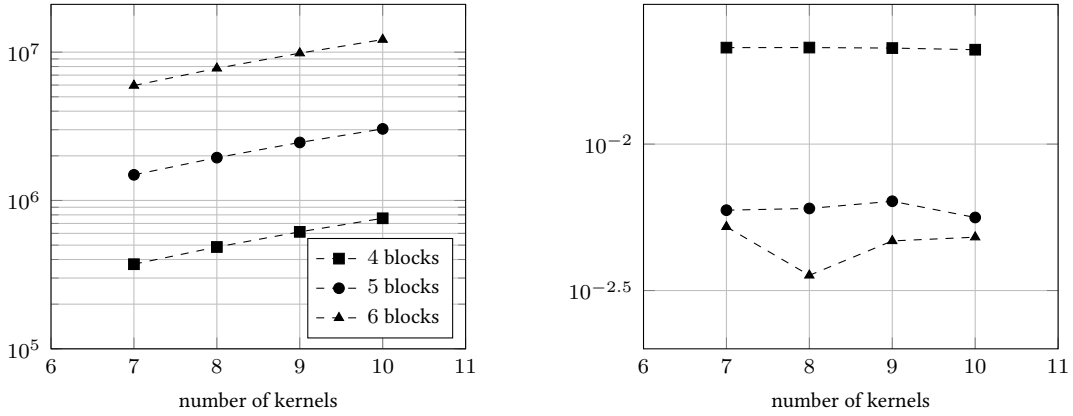


(a) u, reference **(b)** u, predicted **(c)** u, absolute error

(d) v, reference **(e)** v, predicted **(f)** v, absolute error

(g) p, reference **(h)** p, predicted **(i)** p, absolute error

Figure 4.6 | Flow predictions on a Bézier shape from the test set using a U-net model. The horizontal velocity (4.6a, 4.6b, 4.6c), vertical velocity (4.6d, 4.6e, 4.6f) and pressure fields (4.6g, 4.6h, 4.6i) display similar error levels. The trained U-net has 8 filters in its first convolutional layer, leading to 1, 944, 763 trainable parameters in total. The output of the U-net is denormalized to obtain the physical predictions. The error is mostly concentrated in the border of the obstacle.



(a) Number of trainable parameters

(b) MAE on the test set

Figure 4.7 | The impact of hyper-parameters. All the networks are trained following the same settings presented in section 4.1.3, and each architecture is trained 5 times to get rid of the possible impact of initializations.

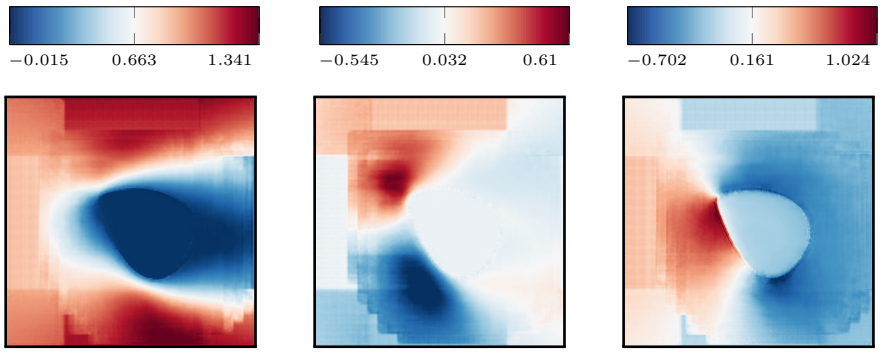
4.1.4 Hyper-parameter selection

Among all the configurable parameters, the number of convolution-convolution-maxpooling blocks b and the number of convolutional kernels m determine how complex the network is. This section is dedicated to a quantitative study of these two hyper-parameters. Three values for b and four values for m are considered, leading to in total 12 configurations of the U-net architecture:

$$b \in \{4, 5, 6\},$$
$$m \in \{7, 8, 9, 10\}.$$

As is shown in figure 4.7a, increasing the number of blocks b significantly increases the number of trainable parameters. In each configuration, the network is trained 5 times with different initializations, and the one with the smallest validation loss is selected as the best model. Then, the averaged MAE on the test set is used to evaluate the performance of the 12 models. The results are given in figure 4.7b. As can be seen, using more blocks in the network remarkably improves performance on the test set, thus underlining the importance of the latent space representation, as the height and width are compressed by a factor of 2^b in the designed autoencoders. However, the marginal utility diminishes when using too many blocks. It is hence not necessary to use more than 6 blocks, considering the additional cost of training increasingly complex neural networks. On the other hand, the number of kernels has no clear impact on the performance.

The predicted velocity and pressure from a U-net with $b = 4$ and $m = 8$ are given in figure 4.8 on a shape from the training set. As can be seen, the segmentation is too coarse to reconstruct the flow fields on a cartesian grid with the step size being 0.01m. One can hence conclude that the capacity of the considered architecture is not large enough to modelize the data set features. All these results confirm the advantage of using deep neural networks to extract hierarchical features. Making the network deeper is an efficient approach to modelize complex nonlinear phenomena in fluid dynamics.



(a) velocity$-u$ **(b)** velocity$-v$ **(c)** pressure$-p$

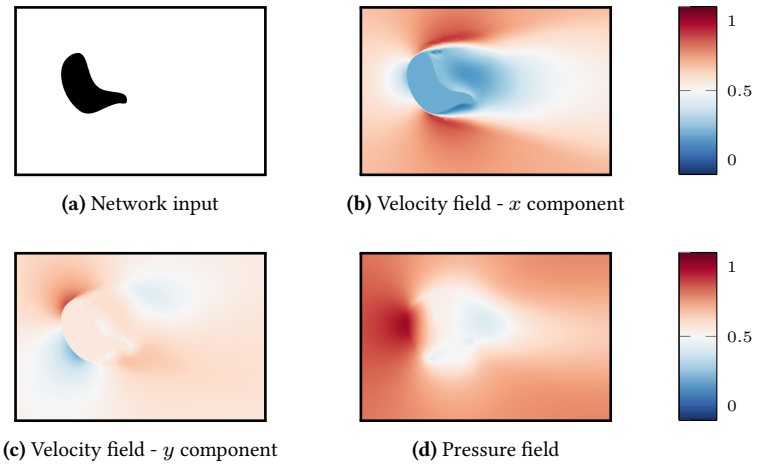
Figure 4.8 | Flow predictions on a Bézier shape from the test set using a U-net model with 4 convolution-convolution-maxpooling blocks. The predicted flow patter shows systematic error, which comes from the lack of learning capacity of the model.

4.2 Turbulent flow prediction

In this section, the U-net approach presented in section 4.1 is applied to turbulent flow. Compared to laminar flow, turbulent flow around obstacles displays more complex patterns around and downstream of the latter. Therefore, the results presented in this section aim at gauging whether convolutional neural networks could be used as surrogate models to Navier-Stokes equations with Spalart-Allmaras turbulence closure.

4.2.1 Data set

Similarly to section 4.1, the velocity and pressure fields around $2,000$ random shapes are used as the data set to be fitted by convolutional neural networks. Unlike the laminar case though, a $4\text{m} \times 6\text{m}$ square window containing the obstacle is adopted to generate the data set for this section. The window covers the region $[-2, 4]\text{m} \times [-2, 2]\text{m}$ in order to include vortices behind the obstacle. The flow fields in the window are projected onto a cartesian grid, the grid size being equal to 0.01m, and the selected window therefore resulting in a 401×601 2-D array. In the data set, each instance has 4 components as shown in figure 4.9: the obstacle, the horizontal velocity, the vertical velocity, and the pressure. The data set is non-dimensionalized and normalized using the same method as section 4.1.1. Finally, the whole data set with $2,000$ instances requires 19.7 GB of disque space. By reducing the grid size to 0.001m, the data set would require 1.92TB of disque space. The boundary element size used for CFD simulation, equal to $5 \times 10^{-5}\text{m}$, would lead to more than 769.5TB of data. For academic interest, the grid size is chosen to be 0.01m, although this leads to an information loss in the boundary layer.



(a) Network input **(b)** Velocity field - x component

(c) Velocity field - y component **(d)** Pressure field

Figure 4.9 | The obstacle, velocity field and pressure field for a dataset element. The geometric information is encoded into a binary field, with the solid obstacles being 0, and the non-solid domain being 1. The binary input 4.9a is the input of the CNN. The interpolated velocity field (4.9b, 4.9c) and pressure field (4.9d) make a $401 \times 601 \times 3$ tensor as the output of the CNN. The solid obstacle is not masked in the flow fields.

4.2.2 Training and evaluation

As the model to fit the data set, a U-net with 7 convolution-convolution-maxpooling and 7 kernels in the first layer is trained, leading to an architecture with $23,822,186$ trainable parameters. All the training configurations are the same as in section 4.1.3. The training ends after 144 epochs, taking 1.07 hours on a Tesla V100 GPU card (see figure 4.10), with the final validation loss and test loss being respectively

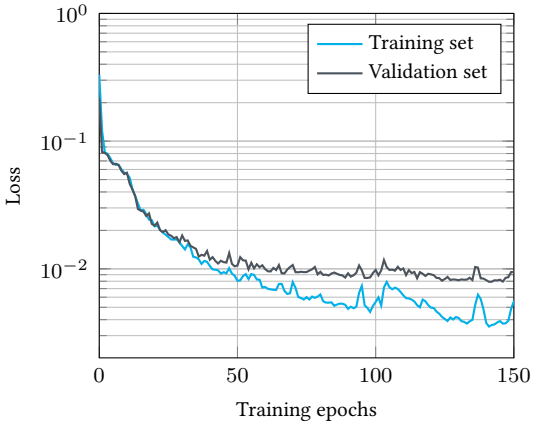


Figure 4.10 | Training history of the U-net model. The training ceases in few than 150 epochs. A slight overfitting is observed, but the test loss remains at the same level of validation loss.

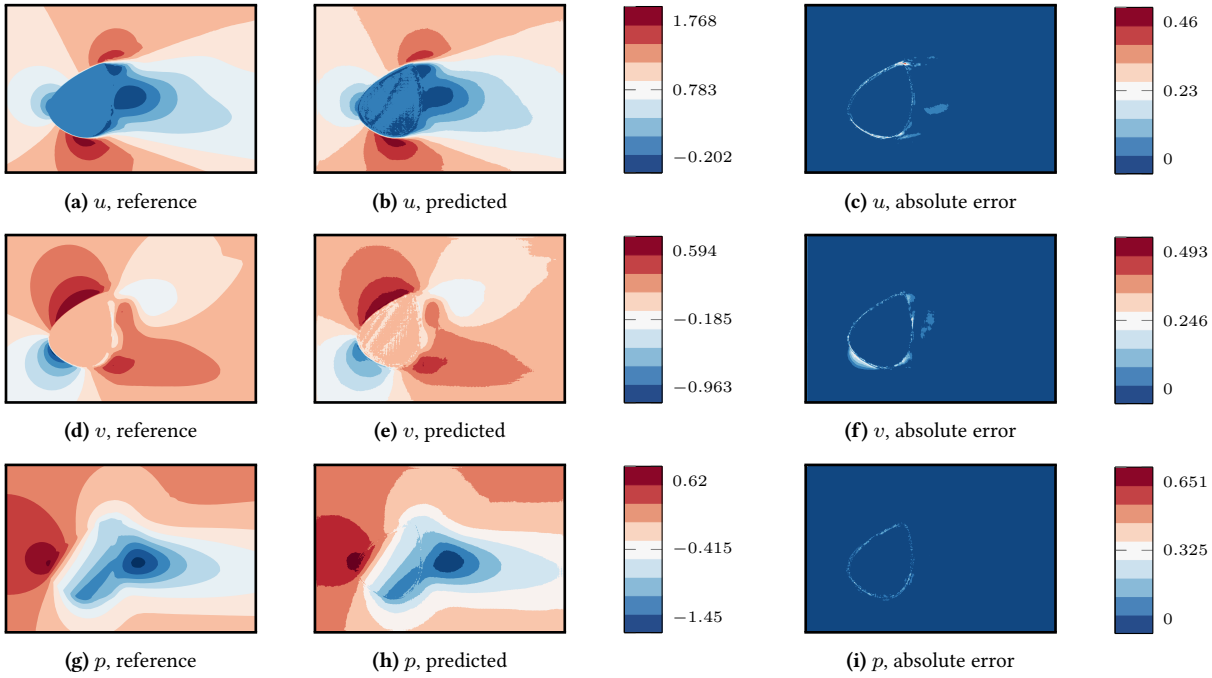
equal to 7.92×10^{-3} and 7.78×10^{-3}. An example of prediction from the test set is given in figure 4.11. The error maps show similar pattern as in the laminar case, with large discrepancy concentrated near the obstacle.

4.2.3 Impact of network depth

Since the number of kernels proved to have little impact on the U-net performance, the number of convolution-convolution-maxpooling blocks b is separately studied in this section. Three values of b are considered: 5,6 and 7. The number of trainable parameters and the averaged MAE on the test set are compared in figure 4.12. Using a $6-$block architecture presents a great advantage, both in performance and model complexity. The predicted velocity and pressure of the $5-$ block architecture are also presented in figure 4.13, where systematic failure of the predictions can be observed, similar to the laminar flow predicted by the $4-$block U-net. Although the $5-$ block architecture is able to give decent laminar flow prediction, its capacity is proved to be not sufficient to modelize turbulent flow. To extract essential latent features for turbulent flow reconstruction around the obstacles, the encoder should have at least 6 blocks, which confirms the more complex nature of turbulent flow than laminar flow.

4.3 Conclusion

The U-net, a special convolutional auto-encoder, was shown to be able to modelize the laminar and turbulent flow around random 2-D obstacles. On the laminar data set, the trained U-net reaches an MAE level about 6×10^{-3} on the test set, which corresponds to a high-quality reconstruction of the velocity and pressure fields, although the fields in the vicinity of obstacles are poorly predicted. The low-quality prediction of the boundary layer is caused by the discontinuity at the interface, as well as the staircasing effect with the cartesian grid. One could use higher-resolution grids to capture the interface, but the training cost would rise rapidly and could easily exceed the hardware limits. On the turbulent data set, the MAE is approximately equal to 8×10^{-3} on the test set, with the predicted flow fields still being accurate except in the boundary layer. However, compared to laminar flow prediction, two negative points must be stressed:



(a) u, reference **(b)** u, predicted **(c)** u, absolute error

(d) v, reference **(e)** v, predicted **(f)** v, absolute error

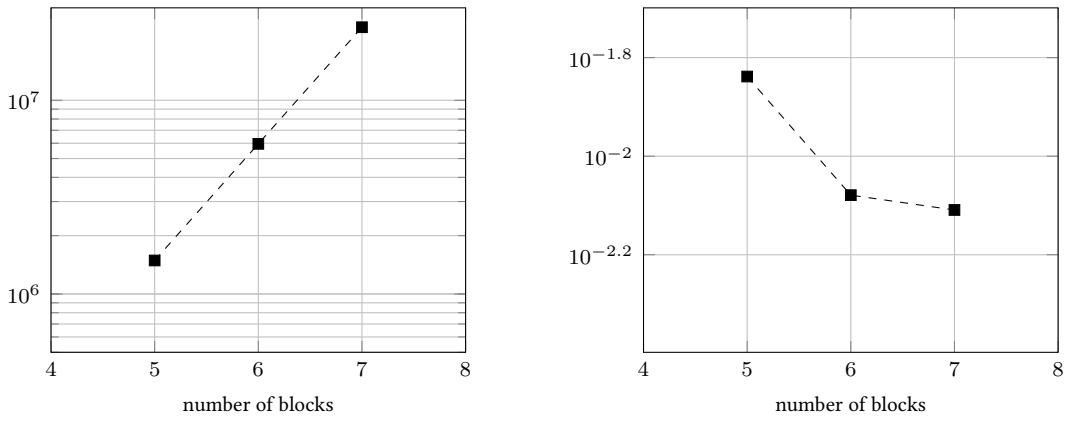
(g) p, reference **(h)** p, predicted **(i)** p, absolute error

Figure 4.11 | Flow predictions on a Bézier shape from the test set using a U-net model. The horizontal velocity (4.11a, 4.11b, 4.11c), vertical velocity (4.11d, 4.11e, 4.11f) and pressure fields (4.11g, 4.11h, 4.11i) display similar error levels. The trained U-net has 7 blocks in the encoder and 7 kernels in its first convolutional layer, leading to $23,822,186$ trainable parameters in total. The output of the U-net is denormalized to obtain the physical predictions. The error is mostly concentrated in the border of the obstacle.

- The MAE level, approximately equal to 8×10^{-3}, is higher than that of the laminar case, equal to 6×10^{-3} of laminar case, although a deeper U-net architecture has been used for turbulence prediction. Additional difficulties could be expected if considering more complex flow patterns.
- The grid size is equal to 0.01m, which implies a much coarser grid compared to the triangular mesh used for CFD simulation. If using finer grids, the size of the data set would augment quadratically, making the training requirements exceed standard hardware capacities.
- Using a smaller grid size should have an impact on the required network depth, the trainining cost, the prediction quality, but this point is not covered due to hardware constraints.

Two hyper-parameters concerning the U-net's architecture are studied: the number of blocks in the encoder and the number of kernels in the each convolutional layer, with both parameters having a direct impact on the latent feature tensor of the auto-encoder architecture. It is evidenced by numerical experiments that there should be enough blocks to extract essential features from the network's input, and that the model's performance is remarkably improved when using more blocks, while the network becomes deeper and the latent feature tensor are further compressed in the height/width axes. On the other hand, using more kernels showed negligible impact on the performace.

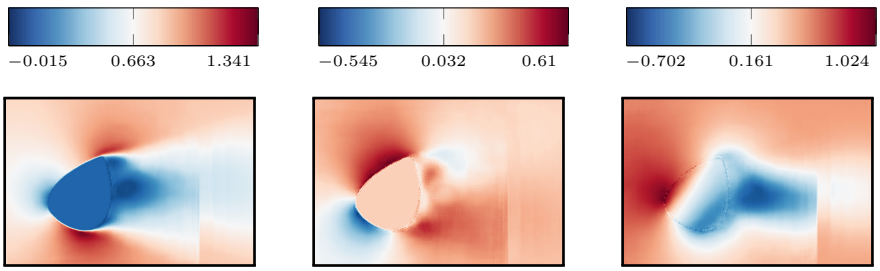
This chapter allowed to introduce the use of CNNs for fast flow prediction, both at low and high Reynolds numbers. In the following chapter, the proposed architecture is modified to perform on-the-fly outlier dectection on top of flow prediction.



(a) Trainable parameters

(b) MAE test set

Figure 4.12 | The impact of hyper-parameters. All the networks are trained following the same settings presented in section 4.1.3, and each architecture is trained 5 times to get rid of the possible impact of initializations.



(a) velocity$-u$

(b) velocity$-v$

(c) pressure$-p$

Figure 4.13 | Flow predictions on a Bézier shape from the test set using a U-net model with 5 convolution-convolution-maxpooling blocks. The predicted flow patter shows systematic error, which comes from the lack of learning capacity of the model.

5

Anomaly detection and uncertainty quantification for the CNN-based surrogate model

Résumé

Au cours des dernières années, les méthodes d'apprentissage profond se sont révélées d'un grand intérêt pour la communauté de la dynamique des fluides numériques, en particulier lorsqu'elles sont utilisées comme modèles subrogés, que ce soit pour la reconstruction d'écoulement, la modélisation de la turbulence ou la prédiction de coefficients aérodynamiques. Dans l'ensemble, des niveaux de précision exceptionnels ont été obtenus, mais la robustesse et la fiabilité des approches proposées restent à explorer, en particulier en dehors de la région de confiance définie par le jeu de données. Dans cette contribution, nous présentons une architecture d'autoencodeur avec double décodeur pour la reconstruction de flux laminaire incompressible avec estimation d'incertitude autour d'obstacles 2-D. L'architecture proposée est formée sur un ensemble de données composé de flux laminaires calculés numériquement autour de $12,000$ formes aléatoires, et applique naturellement une relation quasi-linéaire entre une branche de reconstruction géométrique et le décodeur de prédiction de flux. Sur la base de cette caractéristique, deux processus d'estimation de l'incertitude sont proposés, permettant soit une décision binaire (accepter ou rejeter la prédiction), soit proposant un intervalle de confiance avec la prédiction des l'écoulement (u, v, p). Les résultats sur l'ensemble de données ainsi que des formes jamais vues montrent une forte corrélation positive entre le score de reconstruction et l'erreur quadratique moyenne de la prédiction

d'écoulement. De telles approches offrent la possibilité d'avertir l'utilisateur de modèles entraînés lorsque l'entrée fournie montre un écart trop important par rapport aux données d'entraînement, ce qui rend le modèle subrogé conservateur pour une prédiction d'écloument rapide et fiable.

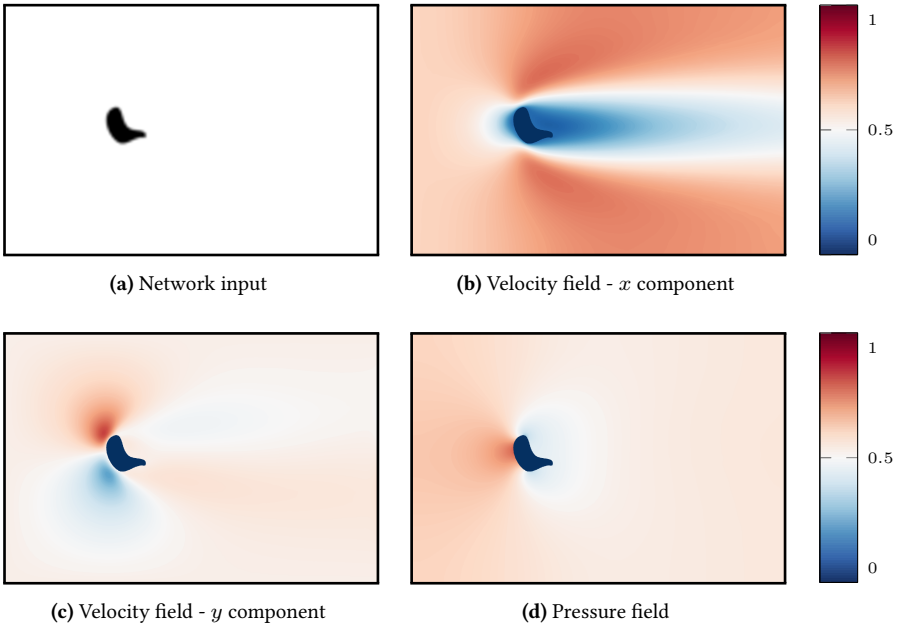
5.1 Introduction

In the previous chapter, numerical experiments were presented to outline the capacities of convolutional neural networks to modelize complex laminar and turbulent flow data sets, and to use them as surrogate models for traditional PDE solvers, in adequate contexts. Still, one must keep in mind that, under stability and convergence assumptions, a PDE solver provides physics-consistent solutions for arbitrary scenarios. On the other hand, the data-driven approach brings nothing about reliability, and that question is barely addressed in the CNN litterature. Conversely, as surrogate models rely on data fitting, their accuracy and robustness cannot be verified outside the boundary of their training data set. In the present chapter, tools are developed to overcome the latter limitation, answering the question: can CNN models be tailored to reliably provide a measure of their own prediction accuracy?

The topological complexity of the input space can make it hard to determine whether or not a given element, provided by an external user, lies within the boundaries of the data set used during training. More, the end user of the model is usually not in possession of the data set, hence approaches based on PCA are not a possible option. While very few approaches were proposed to tackle such issues in the context of CNN-assisted CFD, several outlier detection techniques have been proposed in other domains. Among them, unsupervised methods have attracted much attention, as they do not require labeled data. In particular, several autoencoder (AE) based techniques were developed for medical and industrial applications: in [HHWB02], a fully connected AE with three hidden layers is applied to breast cancer detection; in [CSW+20], a convolutional AE (CAE) is used to detect cracking and spalling defects on concrete structures; in [KLH17], CAEs are used to detect miss-printed logo images on mobile phones, and in [BWAN18], the authors compare several variants of AE on anomaly segmentation in brain magnetic resonance images.

In this chapter, we introduce an autoencoder architecture with a twin-decoder as a possible tool for outlier detection in the context of fluid flow predictions. New contributions include:

- A novel twin-AE architecture displaying a strong correlation between the input reconstruction and the flow prediction error levels by taking advantage of proper skip connections between the two decoder branches. This correlation is found to be almost linear, at the expense of a slightly lower flow prediction accuracy than that of a U-net with similar structure;
- Two uncertainty estimation procedures taking advantage of the latter property: (i) a qualitative procedure based on a user-provided error threshold level, providing binary decisions regarding the prediction (accept or reject), and (ii) a quantitative procedure, providing the user with a error level interval on top of the flow prediction. Results over the considered data set as well as unseen shapes proved these methods to be efficient to detect the applicability limits of the trained model. Both methods rely on simple concepts, and can be easily applied to other end-to-end prediction tasks.



(a) Network input **(b)** Velocity field - x component

(c) Velocity field - y component **(d)** Pressure field

Figure 5.1 | Network input, velocity field and pressure field for a dataset element. The shape is shown in its computational domain (5.1a), along with the computed velocity field (5.1b, 5.1c) and pressure field (5.1d).

5.2 Data set

The data set is composed of $12,000$ shapes, along with their steady-state velocity and pressure fields at $Re = 10$ (see figure 5.1), which are generated by the numerical methods described in section 3.2. The input fields are resized to 2-D 100×150 arrays before being provided to the network, using a grid size 0.1m. The flow fields are normalized and non-dimensionalized using the same method as presented in section 4.1.1, so that the pixel values are in the range $[0, 1]$.

5.3 CNN architecture

The architecture of the twin-decoder proposed in this section is shown in figure 5.2. Its input consists in a boolean 1-channel tensor containing the computing domain and the obstacle. Its outputs are (i) a 1-channel tensor containing the reconstructed input, and (ii) a 3-channels tensor containing the predicted velocity components and pressure fields. The encoder branch consists in stacked convolution-convolution-max-pooling blocks using 3×3 kernel size, with stride size equal to 1, and zero-padding. The convolutional layers exploit ReLU as activation functions. After every pooling operation, the number of kernels used for convolutional layers is doubled, until reaching the bottleneck.

The decoder is composed of two branches, hereafter denoted "shape decoder" and "flow decoder". Both decoder branches are composed of transposed convolution-convolution-convolution blocks, and share similar structures. The transposed convolution layers use 2×2 kernel size, stride size equal to 2, zero padding and ReLU activation. Symmetrically to the encoder branch, the number of kernels is halved after each block in the decoder branches. Finally, a convolutional layer with four 1×1 kernel is applied to set the final number of channels (1 for the shape decoder, and 3 for the flow decoder). The output of our network hence contains 1-channel tensor representing reconstructed input, and a 3-channel tensor representing the velocity and pressure.

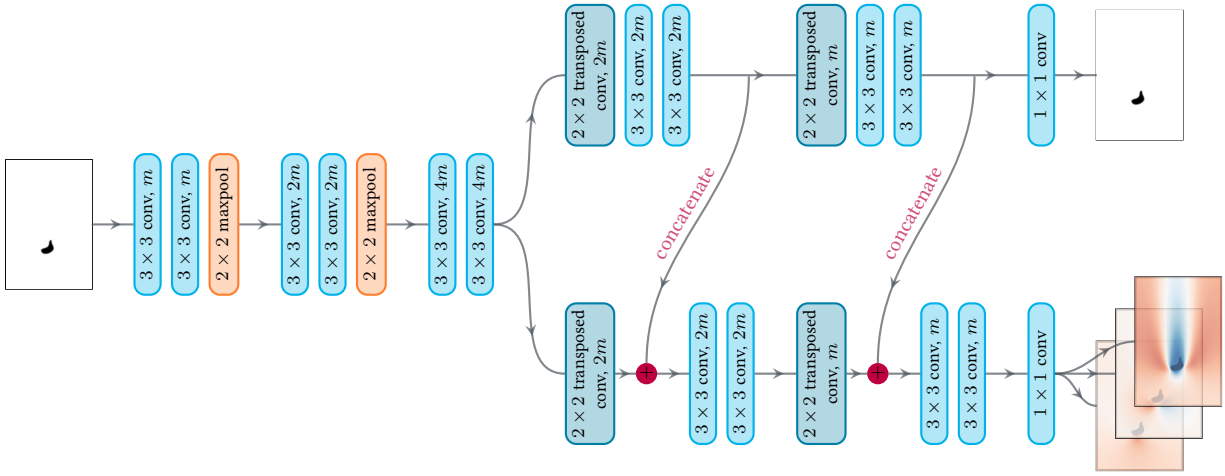


Figure 5.2 | Proposed twin-decoder architecture. The encoder is based on a pattern made of two convolutional layers followed by a max-pooling layer. At each occurence of the pattern, the image size is divided by two, while the number of filters, noted m, is doubled. In both decoder paths, a transposed convolution step is first applied to the input, while the number of filters is halved. The output of this layer in the flow decoder is then concatenated with its mirror counterpart in the shape decoder. Finally, two convolution layers are applied. At the end of the last layer, a 1×1 convolution is applied on each decoder to obtain a final 3D tensor with 4 channels. Every channel has the same dimension as the input.

The key ingredient of the proposed architecture lies in the skip connections that link the shape decoder and the flow decoder. The output of each shape decoder block is concatenated (along the channel axis, to preserve shape) to the output of the transposed convolution layer of each flow decoder block. This idea is similar to that of U-net, except that low-level features do not originate from the encoder branch, but from the reconstruction of the input. Forcing such dependence between the two decoder branches is expected to induce a strong correlation between their performance levels. In essence, by enforcing a relation between the shape reconstruction error and the flow prediction error, the proposed method allows to reject possible outliers based on the reconstruction error. As the latter can be computed for an arbitrary input, the end-user can be warned if the network prediction can be trusted or not. More details on the acceptance/rejection procedure are provided in section 5.5.

5.4 Training procedure

The loss function used to train the twin-decoder architecture is a weighted sum of the shape and the flow decoder losses. Both decoders use the regular mean squared error (MSE) as the loss function:

$$l\left(y^{\text{pred}}, y^{\text{ref}}\right) = \frac{1}{c \times 100 \times 150} \sum_{\substack{1 \leq i \leq c, \\ 1 \leq j \leq 100, \\ 1 \leq k \leq 150}} |y_{ijk}^{\text{pred}} - y_{ijk}^{\text{ref}}|^2, \tag{5.1}$$

with $c = 1$ for the shape decoder and $c = 3$ for the flow decoder. The final loss function used for training is:

$$l_{\text{twin}} = l_{\text{flow}} + \beta l_{\text{shape}}, \tag{5.2}$$

where β is a weighting parameter that remains to be tuned (see section 5.6). The network is trained with the Adam optimizer using an initial learning rate of 1×10^{-3}, which is reduced to 1×10^{-4} after 600

epochs. To prevent overfitting, the validation loss is monitored, and early stopping is used to determine the end of the training. The network parameters are initialized using the Glorot normal method, following [RFB15]. Training is performed using a Nvidia Tesla V100 GPU, using mini-batches of size 128 to limit the required computational resources. As the hyper-parameters will be compared, training times are given in section 5.6.

5.5 Trust level based on shape reconstruction

Given a trained twin-decoder neural network, it is feasible to evaluate the trust level of flow prediction by input reconstruction. As the error levels of the twin decoders are strongly correlated, a qualitative and a quantitative trust-level methods are proposed, both based on the reconstruction error. Their general concepts are proposed in the following sub-sections, while their applications on trained networks are presented in section 5.9. In the following, the MSE errors for flow and shape reconstructions are respectively denoted e_f and e_s.

5.5.1 Qualitative method

In this case, the end-user provides an acceptable MSE error level e_f^* on the flow prediction. The method consists in selecting the associated shape reconstruction error e_s^* that minimizes the probability of taking wrong decisions when supposing that the two error levels are linearly correlated. The threshold shape reconstruction error e_s^* is the solution to the following minimization problem:

$$e_s^* = \underset{e}{\operatorname{argmin}} \frac{1}{N} \left[\operatorname{card}\left(e_s < e \text{ and } e_f > e_f^*\right) + \operatorname{card}\left(e_s > e \text{ and } e_f < e_f^*\right) \right], \tag{5.3}$$

where N is the number of scatter points taken into account. The numerator sums the amount of wrong decisions as the error of both decoders are assumed to be positively correlated. Such a formulation prevents the end-user from making mistakes when accepting or rejecting predictions. An illustration of the method is shown in figure 5.3a.

5.5.2 Quantitative method

With the quantitative method, the flow prediction error e_f and an associated confidence interval δ_f are directly estimated from the input reconstruction error e_s using the following relation:

$$e_f = a e_s + b + \epsilon, \tag{5.4}$$

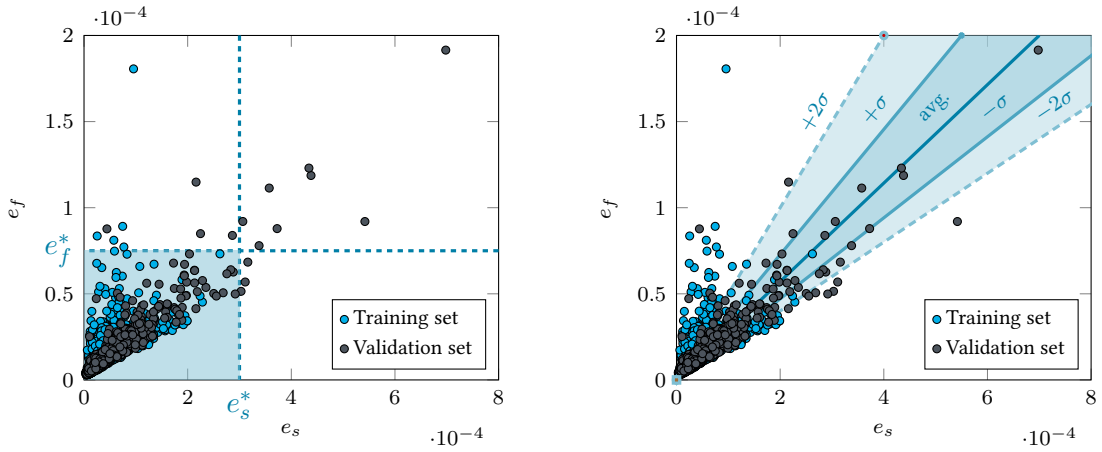
where ϵ follows a normal law of the form:

$$\epsilon \sim \mathcal{N}(0, (c e_s)^2). \tag{5.5}$$

The assumption is supported by the linear pattern of error scatter plots. In essence, indexing the standard deviation on e_s in (5.5) represents the growing uncertainty on flow prediction as shape reconstruction turns worse. Under this formulation, the likelihood of e_f on N scatter points is:

$$\prod_{i=1}^{N} p(e_f^i | e_s^i; a, b, c) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi (c e_s^i)^2}} \exp\left(-\frac{(e_f^i - a e_s^i - b)^2}{2(c e_s^i)^2} \right) \tag{5.6}$$

The optimal parameters a^*, b^* and c^* are obtained by minimizing the negative log likelihood:



(a) Qualitative method

(b) Quantitative method

Figure 5.3 | Description of the qualitative and quantitative methods on the scatter plots of e_f versus e_s on training and validation sets for a reference twin auto-encoder architecture. (Left) Qualitative method: given a threshold e_f^*, the corresponding optimal e_s^* is found by solving problem (5.3). Then, the end-user rejects predictions that produce shape reconstruction errors superior to e_s^*. Only the predictions falling within the bottom left quarter (in orange) are accepted. (Right) Quantitative method: e_f is modeled as an affine function of e_s with an uncertainty interval, as shown in relation (5.4). Based on e_s, the method provides an estimated e_f level, along with a confidence interval for the prediction.

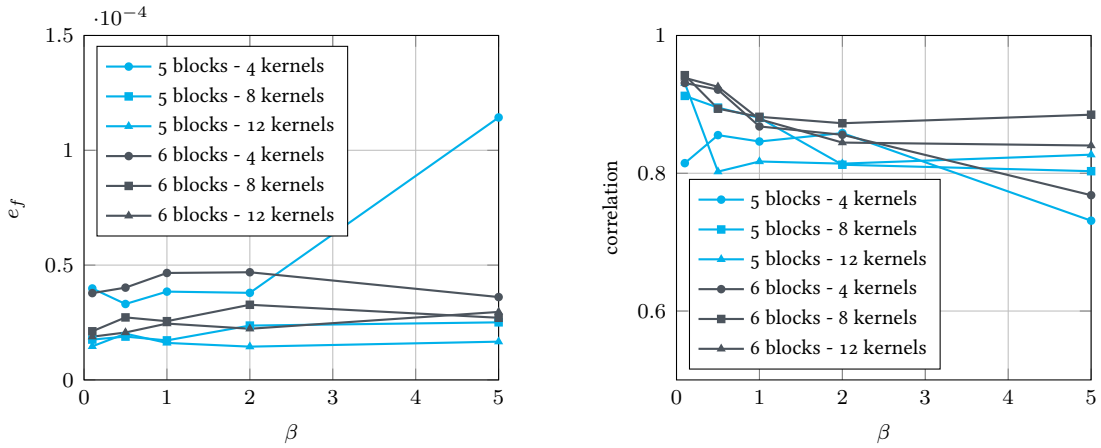
$$(a^*, b^*, c^*) = \underset{a,b,c}{\operatorname{argmin}} \frac{1}{2} \sum_i \left(\frac{e_f^i - a e_s^i - b}{c e_s^i} \right)^2 + \sum_i \log(c e_s^i) + \frac{N}{2} \log(2\pi), \tag{5.7}$$

which is achieved using a gradient descent algorithm. An illustration of the method is shown in figure 5.3b. The minimum is determined by training and validation error, then evaluated on the test set. To the difference of the qualitative method, in which the predictions are either plainly accepted or rejected, such formulation allows one to estimate e_f with a confidence interval δ_f, without a need of pre-selecting a threshold accuracy, thus providing an additional flexibility. As an example, given a shape reconstruction error e_s, the associated flow prediction accuracy level e_f would fall into $[(a-2c)e_s + b, (a+2c)e_s + b]$ with 95% probability.

5.6 Training and hyper-parameter selection

In this section, the impact of three different parameters is explored: (i) the number of convolutional blocks b, (ii) the number of kernels in the first convolutional layer m, and (iii) the weighting parameter β. In total, 30 configurations are studied, with the network being evaluated not only on its flow field prediction performance, but also on the correlation coefficient obtained between e_f and e_s on the validation set. As the different explorations are detailed below, the corresponding results can be found in figure 5.4.

The number of convolutional blocks b in the encoder (or equivalently the number of transposed convolution blocks in the decoders) proved to be determinant regarding the performance of the flow decoder. Based on previous experiments on U-nets, only architectures with 5 or 6 blocks were considered. Architectures with 5 blocks proved to outperform deeper ones in terms of flow prediction, probably due to a too high compression rate in the latent space when using 6 blocks. Adversely, architectures with 6 blocks presented a slightly better correlation coefficient between the two decoder errors.



(a) Flow prediction MSE

(b) Correlation coefficient between e_s and e_f

Figure 5.4 | Hyper-parameter calibration. The performance is evaluated on the validation set. To compare the accuracy of flow prediction between different architectures, the MSE is averaged over the entire validation set.

The number of convolutional kernels m The flow prediction accuracy is highly dependent on the number of convolution kernels used in each blocks. Since the architecture is symmetric and scalable, we use the number of kernels in the first convolutional layer to represent this hyper-parameter. Results show a clear advantage of using 8 kernels over 4, while increasing from 8 to 12 is not as beneficial. Hence, it is not necessary to use too many kernels, as a CNN complexity scales with the square of the kernel number. Similar conclusions hold for the correlation coefficient.

Weighting parameter β The β parameter in the loss function (5.2) proved to have a crucial impact on correlation of the two decoder errors. Five different values were considered (from 0.1 to 5), with larger values giving more weight to the shape decoder during the training process. We found that small β values were very beneficial to the correlation level, while alleviating the differences caused by b. With $\beta = 0.1$, a $5-$block architecture with 12 kernels obtained a correlation level of 93%, which represents a strong linear relation. The impact of β on the flow prediction accuracy is not as clear according to the obtained results.

5.7 Training cost and accuracy

In order to evaluate the computational cost of the training, we provide the best performance attained by each combination with respect to the number of trainable parameters in table 5.1. Each point represents the smallest e_f value obtained by tuning β. By using the $5-$block architecture with 12 kernels, e_f values as low as 1.4×10^{-5} can be reached, requiring 1.4 million parameters, making it the architecture of choice, with the best cost-accuracy ratio. Learning time on a Tesla V100 GPU is approximately 0.7 hours. The training curves for the training and validation subsets are presented in figure 5.5.

On the test set, the proposed model reaches a flow reconstruction accuracy $e_f = 1.38 \times 10^{-5}$, thus showing good generalization capabilities on unseen data. In figure 5.6, a prediction example from the test set is shown, along with the associated exact solution and a pixel-wise error map. As can be seen, the velocity and pressure fields are well recovered by the network, the error being concentrated in the vicinity of the obstacle, *i.e.* in the area of large pressure and velocity gradients. As the obstacle itself only represents a small portion of the predicted field, a second metric is proposed: for each element of the

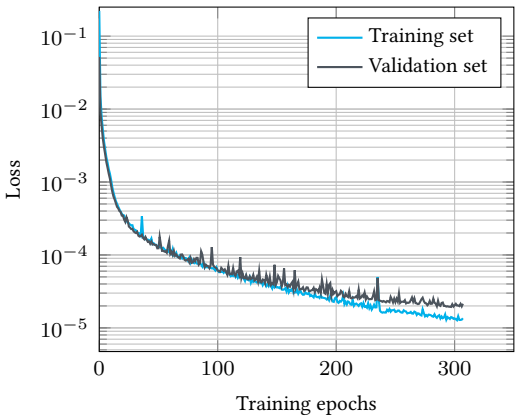


Figure 5.5 | Training history of the model. The training is ceased when the validation loss does not decrease for 10 successive epochs. A slight overfitting can be observed on the trained model.

Table 5.1 | Flow prediction performance obtained for architectures of various complexities. A good ratio must be found between the final achievable accuracy and the total number of learnable parameters, as training time rises dramatically with the network complexity. Here, best performance is obtained when $b = 5$ and $m = 12$, with a total of 1.4 million parameters.

Architecture	Best e_f	Nb. of parameters
5 blocks - 4 kernels	3.3063×10^{-5}	157 208
5 blocks - 8 kernels	1.7216×10^{-5}	627 500
5 blocks - 12 kernels	$\mathbf{1.4491 \times 10^{-05}}$	**1 410 880**
6 blocks - 4 kernels	3.6089×10^{-5}	592 024
6 blocks - 8 kernels	2.1178×10^{-5}	2 365 484
6 blocks - 12 kernels	1.8826×10^{-5}	5 320 384

test set, the mean pixel-wise relative error is computed on a smaller area surrounding the obstacle. The corresponding error distributions are plotted in figure 5.7. Overall, the average relative error is 3.92% for horizontal velocity, 3.57% for vertical velocity and 3.55% for pressure, with very few elements exceeding 6% of relative error, showing again decent generalization capabilities.

5.8 Correlation levels

In this section, the correlation levels obtained between the shape reconstruction and the flow prediction errors are commented. To further show the interest of the twin-AE architecture, two close network structures are considered, namely the dual autoencoder (dual-AE), and the U-net dual autoencoder (U-dual-AE). The dual-AE architecture is simply obtained by removing the skip connections of the twin-AE, while the U-dual-AE exploits skip connections coming from the encoder path instead of the reconstruction path. To ensure a fair comparison, the same configuration is used for all three architectures. A concatenation with constant tensor is applied to the flow decoder of the dual-AE, so its number of parameters is equal to that of the U-dual-AE and the twin-AE.

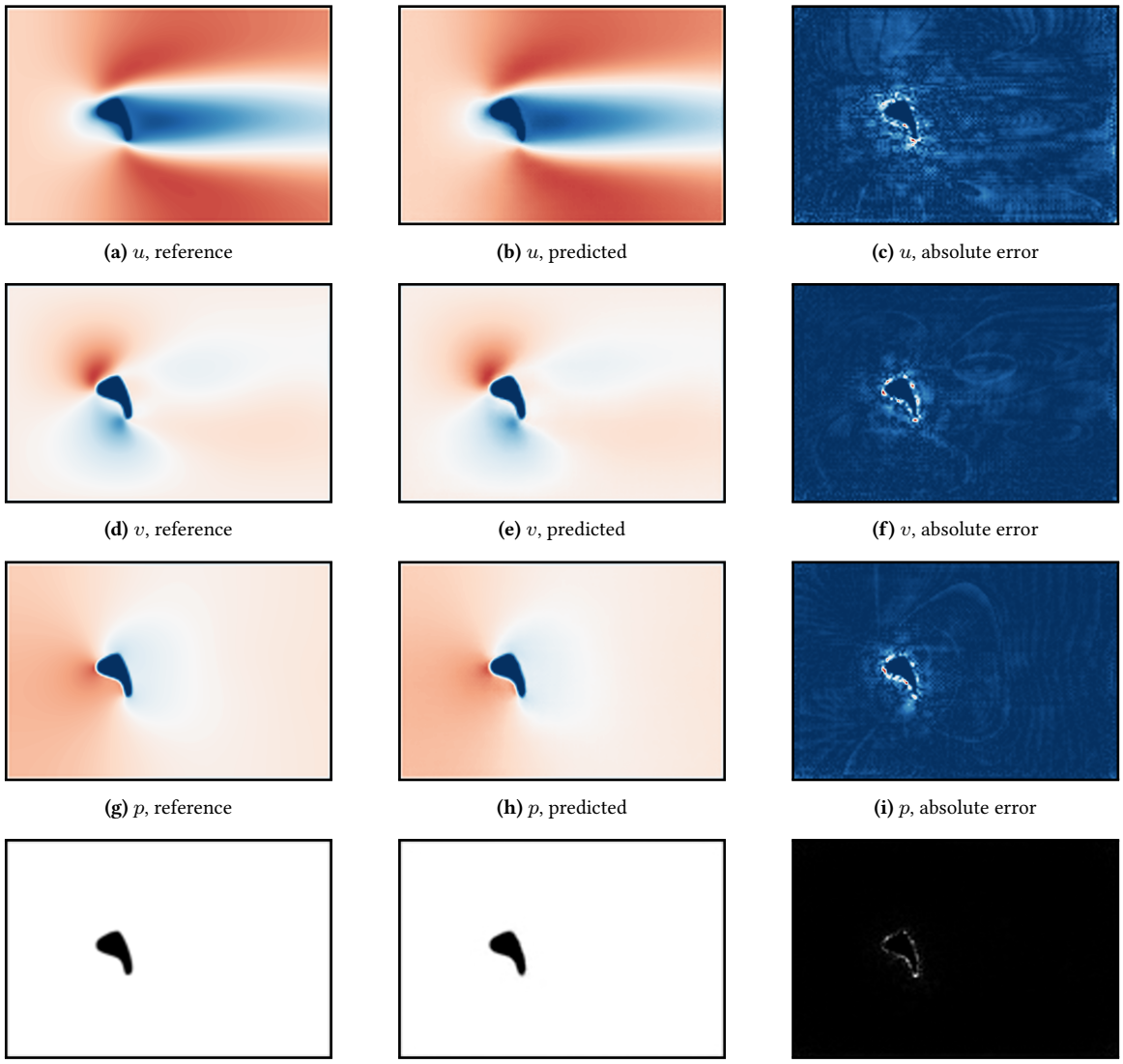
The scatter plots obtained with the twin-AE on the training, the validation and the test sets are respectively shown in figures 5.8a, 5.8b and 5.8c. Associated correlation levels are 0.772, 0.931 and 0.954, meaning that strong linear relations are observed on the validation and test set. Regarding the training set, the weaker correlation is interpreted as a consequence of the slight overfitting observed during training (see figure 5.5). In comparison, the obtained correlation level of the dual-AE on the test set is 0.830, which is significantly weaker than that of the twin-AE, thus proving the interest of the skip connections between the two decoder branches (see figure 5.8d). The twin-AE also has slightly lower relative errors than its dual-AE counterpart (3.92%, 3.57% and 3.55% respectively for u, v, p, against 4.30%, 4.05% and 4.00%). Finally, the U-dual-AE architecture exhibits almost no correlation between e_s and e_f, with a computed correlation level of 0.385 (see figure 5.8e). Adversely, the relative error levels of the U-dual-AE are significantly lower (3.01%, 1.94% and 1.94%), which is in line with results from the literature [CVH19] and the results of previous chapters.

5.9 Trust level based on input reconstruction

This section presents the application of the trust level methods detailed in section 5.5. Again, the underlying concept is to take advantage of the strong correlation between shape and flow reconstruction error levels (see section 5.8) to propose an uncertainty estimation (either qualitative or quantitative) along with the flow prediction.

5.9.1 Qualitative method

A threshold MSE tolerance for the flow reconstruction is provided by the user, and is here chosen to be $e_f^* = 5 \times 10^{-5}$. The corresponding threshold shape reconstruction error e_s^* is obtained by solving the minimization problem (5.3). As the data set size is relatively limited, the problem is solved by an exhaustive search, the results of which are shown in figure 5.9a. One observes that choosing $e_s^* = 1.9 \times 10^{-4}$ minimizes the risks of accepting bad predictions and rejecting good predictions. When testing the procedure on the elements of the validation and testing sets, it is observed that the mistake rate is close to 1% in both cases. In figure 5.9b, the false classification rate on the three subsets is plotted as a function of e_f^*, showing that stricter e_f^* choices inevitably lead to worse performances with this method. The area



(a) u, reference **(b)** u, predicted **(c)** u, absolute error

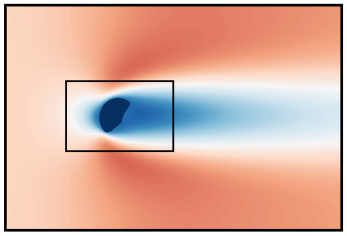
(d) v, reference **(e)** v, predicted **(f)** v, absolute error

(g) p, reference **(h)** p, predicted **(i)** p, absolute error

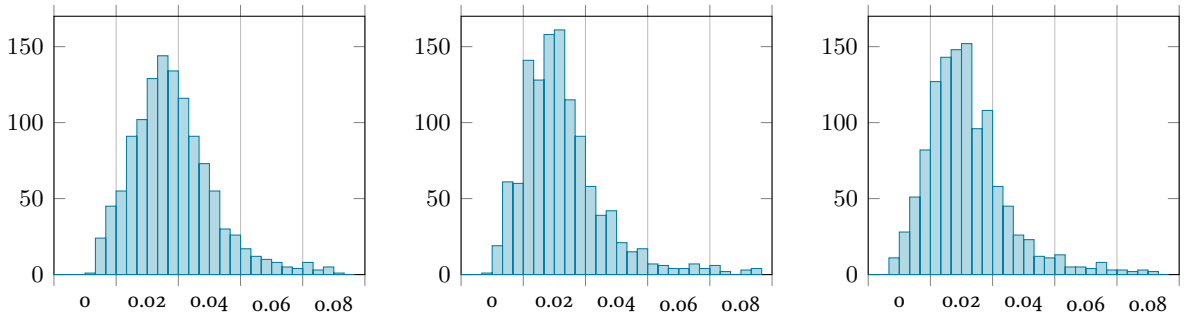
(j) Shape, reference **(k)** Shape, predicted **(l)** Shape, absolute error

Figure 5.6 | Flow and shape predictions around an obstacle from the test set. On this instance, the flow prediction error is $e_f = 1.57 \times 10^{-5}$, with most of the error concentrated on the boundary of the shape, *i.e.* in the area of large pressure and velocity gradients. For the u, v and p predictions, the pixels' value range is still $[0, 1]$. An RGB colormap is used for better visualization

.



(a) Area taken into account to compute relative error histograms.



(b) Relative error for u.

(c) Relative error for v.

(d) Relative error for p.

Figure 5.7 | Relative error for flow predictions over test set. The black rectangle around the obstacle indicates the area on which the u, v and p relative errors are computed. The histograms indicate the error levels obtained when comparing predictions to labels on the 1200 elements of the test set.

of accepted predictions is plotted in figure 5.11a, along with a representation of the elements of the test set.

5.9.2 Quantitative method

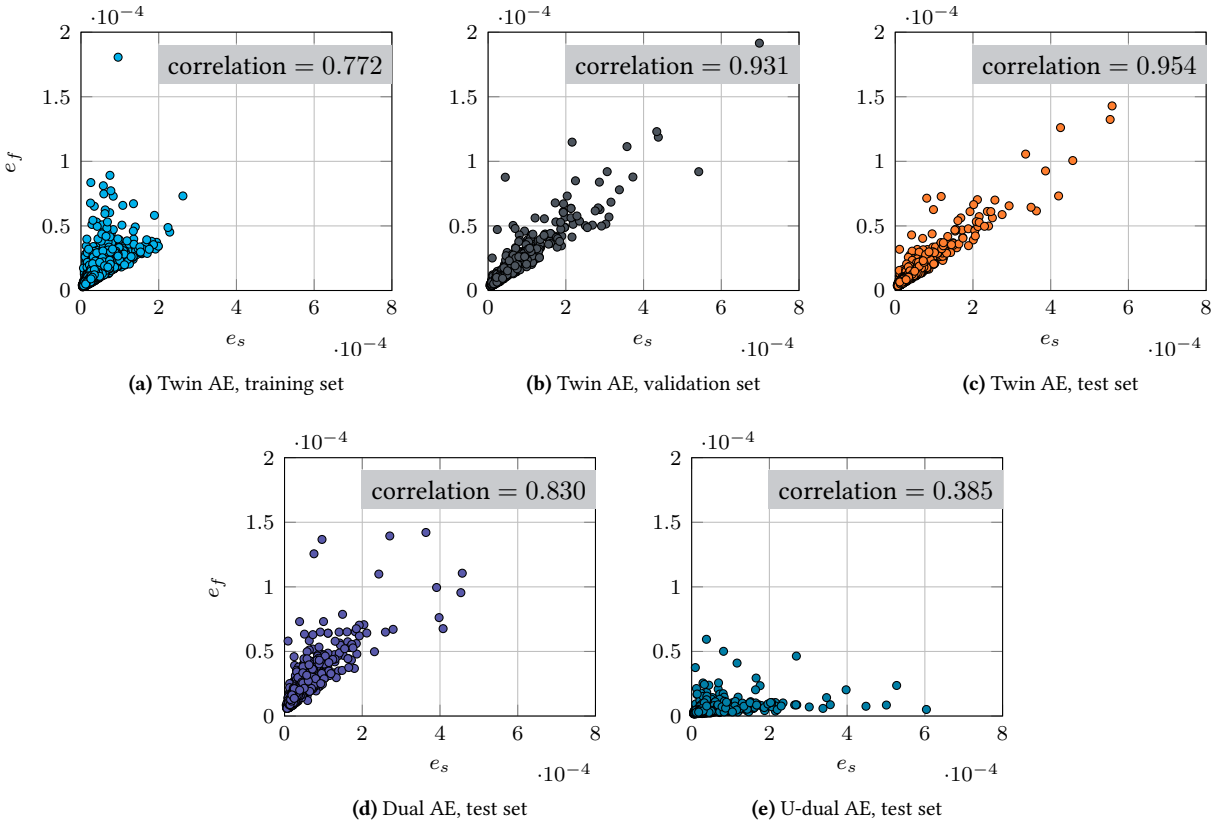
A gradient descent algorithm is used to minimize the negative log-likelihood problem (5.7) over the training set, in order to obtain the optimal parameter set (a^*, b^*, c^*). With an initial value $(a_0, b_0, c_0) = (0.1, 0, 0.1)$, the algorithm converges after 6 iterations (see figure 5.10). The optimal parameters retained are $(a^*, b^*, c^*) = (0.257157, 2.24820 \times 10^{-6}, 0.105841)$, which minimize the negative log likelihood over the validation set. Hence, e_f can be estimated from e_s as:

$$e_f = 0.257157\, e_s + 2.24820 \times 10^{-6} + \mathcal{N}(0, (0.105841\, e_s)^2). \tag{5.8}$$

As shown in figure 5.11b, the regression line and its 1σ confidence interval matches well with the distribution of the test set, with only a handful of (e_f, e_s) couples falling outside of the range. The fact that the confidence interval widens with larger values of e_s translates the increasing scarcity of samples in the test set when e_s rises. For the majority of predictions, though, it provides a good grasp of the flow prediction quality. The 1σ (68% probability) and 2σ (95% probability) confidence intervals for sampled e_s values are provided in table 5.2.

5.9.3 Flow prediction on outliers

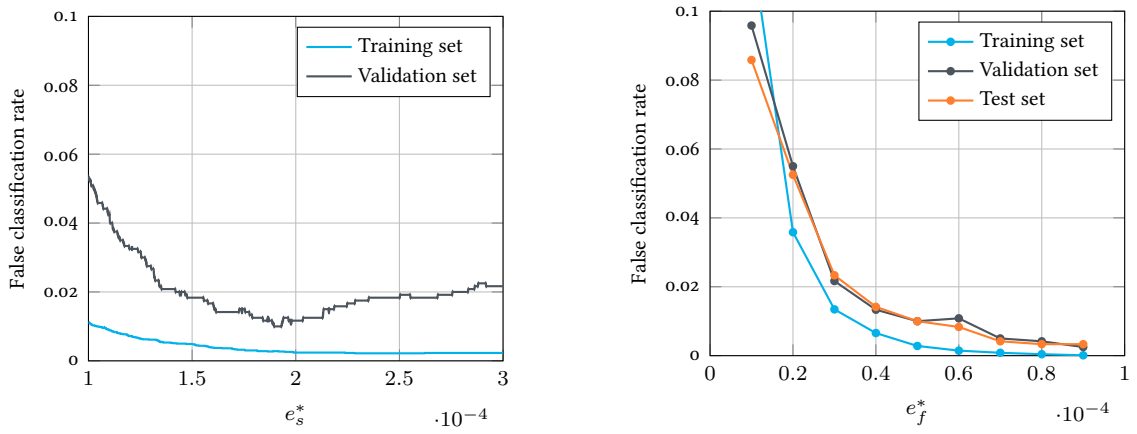
In this section, the capabilities of the qualitative and quantitative methods to detect invalid inputs and outliers are evaluated. To do so, multiple shapes are generated that do not fit within the data set, including polygons with sharp edges (see figure 5.12), shapes included in the data set but misplaced in the input



(a) Twin AE, training set **(b)** Twin AE, validation set **(c)** Twin AE, test set

(d) Dual AE, test set **(e)** U-dual AE, test set

Figure 5.8 | Comparison of scatter plots of e_f versus e_s for different sets and architectures. Top row: the correlation levels obtained with the twin AE on the training, validation and test sets are respectively 0.772, 0.931 and 0.954. Bottom row: dual AE and U-dual AE architectures show weaker correlation levels on the test set, with respective levels of 0.830 and 0.385.



(a) False classification rate as a function of e_s^* for $e_f^* = 5 \times 10^{-5}$.

(b) False classification rate achieved on different subsets for different e_f^* values.

Figure 5.9 | Trust level identification based on the e_s indicator. (Left) The optimal threshold is obtained by minimizing the mistaken classification rate on the training and the validation sets. (Right) The mistake rate rises significantly on all three subsets when decreasing the threshold e_f^* value. For the optimal e_s^* value, the mistake rate on the validation and test sets is approximately 1%.

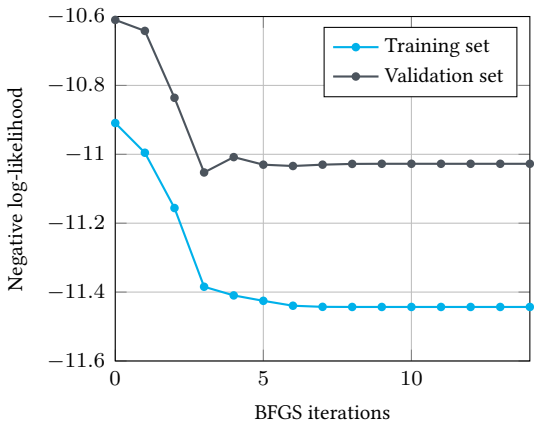
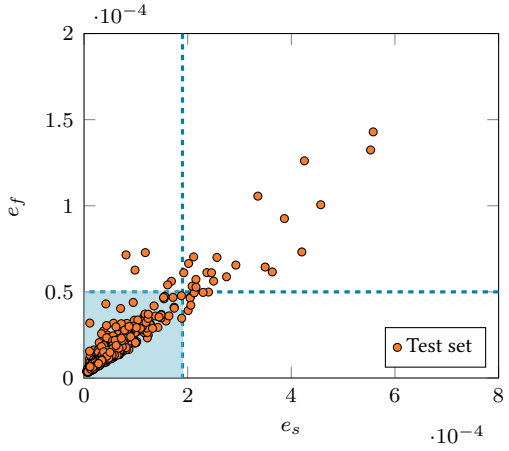
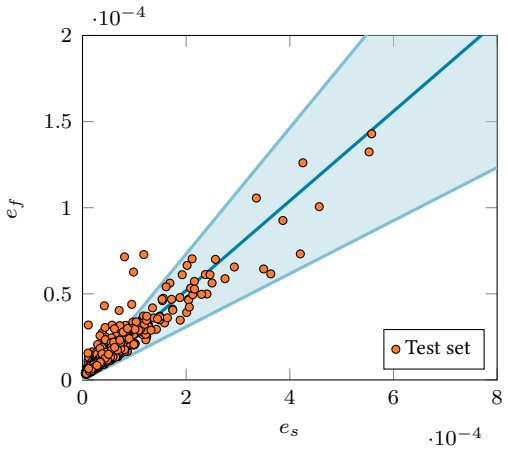


Figure 5.10 | Minimization of the negative log-likelihood problem (5.7) using the BFGS algorithm.



(a) Qualitative method: accepted prediction area when choosing $e_f^* = 5 \times 10^{-5}$ and selecting e_s^* following (5.3).

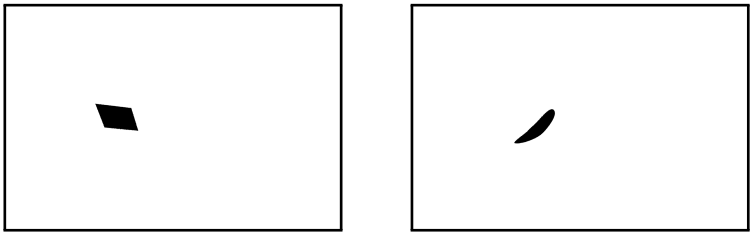


(b) Quantitative method: regression line with its 1σ confidence interval obtained by solving problem (5.7).

Figure 5.11 | Representations of the qualitative and quantitative methods along with the test set scatter plot.

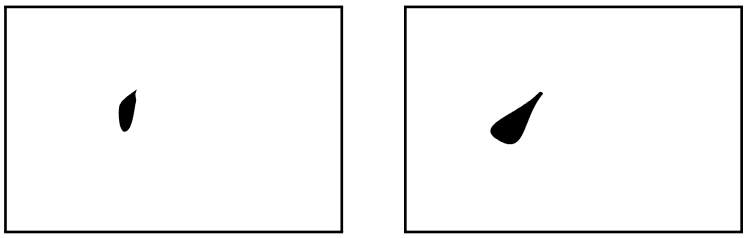
Table 5.2 | Estimating e_f for given e_s values.

e_s	e_f (1σ interval)	e_f (2σ interval)
1×10^{-5}	$[3.76 \times 10^{-6}, 5.88 \times 10^{-6}]$	$[2.70 \times 10^{-6}, 6.94 \times 10^{-6}]$
2×10^{-5}	$[5.27 \times 10^{-6}, 9.51 \times 10^{-6}]$	$[3.16 \times 10^{-6}, 1.16 \times 10^{-5}]$
4×10^{-5}	$[8.30 \times 10^{-6}, 1.68 \times 10^{-5}]$	$[4.07 \times 10^{-6}, 2.10 \times 10^{-5}]$
8×10^{-5}	$[1.44 \times 10^{-5}, 3.13 \times 10^{-5}]$	$[5.89 \times 10^{-6}, 3.98 \times 10^{-5}]$
1.6×10^{-4}	$[2.65 \times 10^{-5}, 6.03 \times 10^{-5}]$	$[9.52 \times 10^{-6}, 7.73 \times 10^{-5}]$
3.2×10^{-4}	$[5.07 \times 10^{-5}, 1.18 \times 10^{-4}]$	$[1.68 \times 10^{-5}, 1.52 \times 10^{-4}]$



(a) Polygon

(b) Bezier shape misplaced by $(+0.5, -0.5)$



(c) Bezier shape misplaced by $(+0.5, +0.5)$

(d) Enlarged Bezier shape

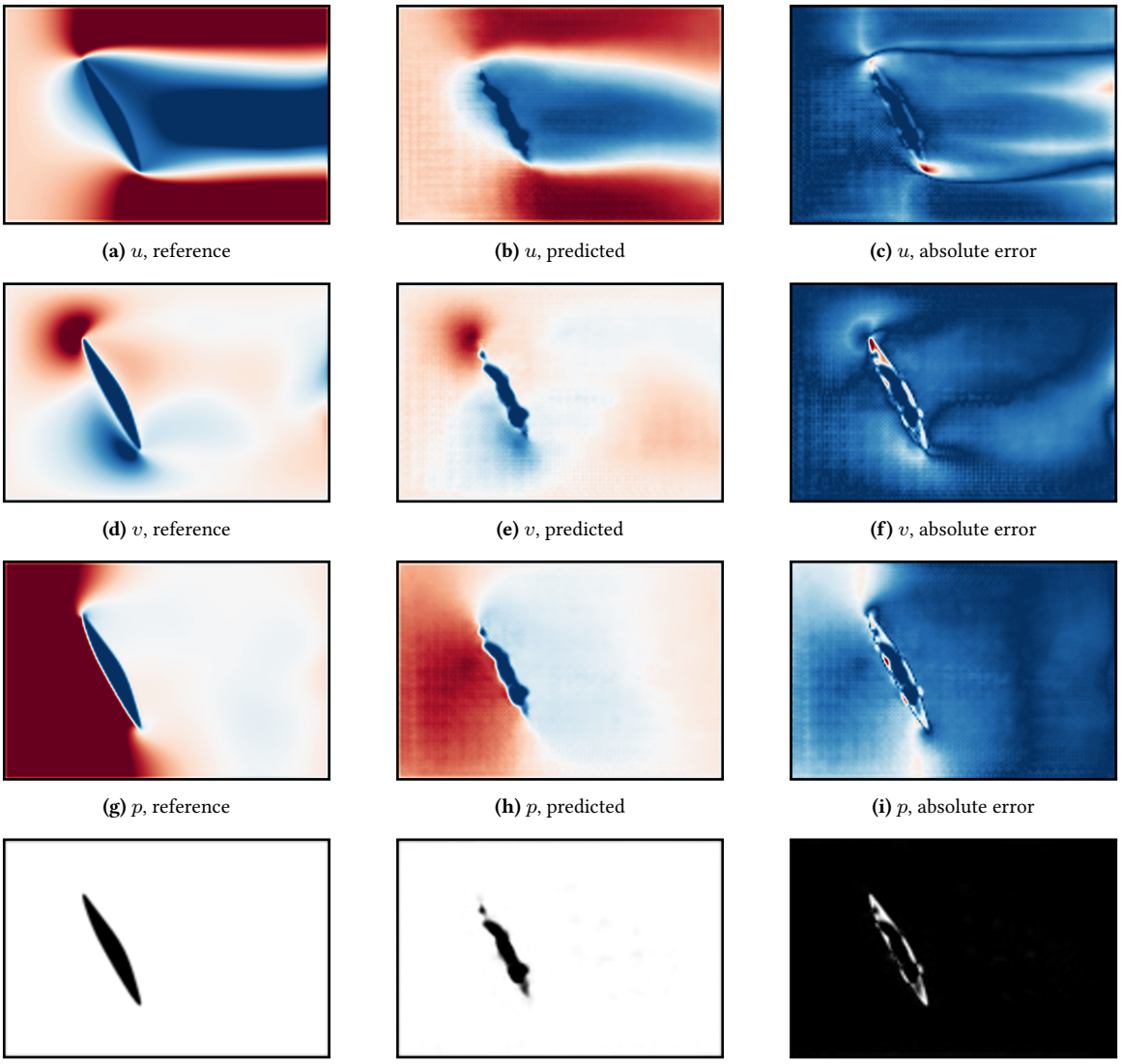
Figure 5.12 | Outlier examples for model evaluation. Polygons are generated with fewer sampling points and sharper edges than the dataset shapes. Misplaced and enlarged shapes have the same curve characteristics as the original data set, but are ill-positioned, or significantly larger than those of the dataset.

Table 5.3 | Model performance on test set and polygons. The comparison is based on the average pixel-level relative error in the 30×45 rectangular zone around the obstacles.

	Shape rel. error	u **rel. error**	v **rel. error**	p **rel. error**
Bezier Test Set	**3.43**%	**3.92**%	**3.57**%	**3.55**%
Polygons	4.16%	6.11%	4.74%	4.70%
Lower right	16.4%	13.5%	17.1%	16.2%
Upper right	16.9%	11.4%	18.2%	14.3%
Enlarged	214%	244%	153%	146%

domain (*i.e.* moved away from the position used in the data set), and enlarged shapes from the data set. In total, 120 outliers are tested. In table 5.3, one can see that the relative errors on the different classes of outliers are systematically larger than those obtained on the data set shapes. While the prediction errors on polygons are only slightly higher that those from the test set, field prediction errors for misplaced shapes are systematically superior to 10%. Enlarged shapes present extremely high reconstruction and prediction errors, higher than 100%. An example of prediction on enlarged Bezier shape is shown in figure 5.13, illustrating the interest of incorporating uncertainty estimation and outlier detection processes in neural network architectures.

In figure 5.14b, the (e_s, e_f) scatter plot of the test set is shown, along with the position of the 120 outliers, both for the qualitative and the quantitative methods. As can be seen, most polygons are located in the accepted region of the qualitative method, indicating that the higher average relative error shown in table 5.3 is caused by a few polygon outliers with large relative errors. When inspecting the polygons set, it was found out that those with largest e_f errors were presenting edges features that were considerably sharper than the others. Almost all the misplaced and enlarged shapes fall into the rejected area of the



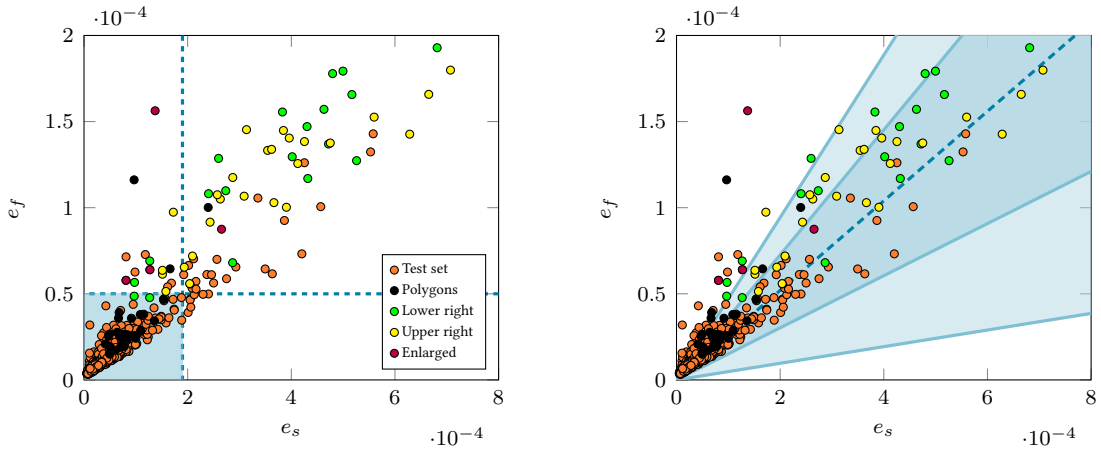
(a) u, reference **(b)** u, predicted **(c)** u, absolute error

(d) v, reference **(e)** v, predicted **(f)** v, absolute error

(g) p, reference **(h)** p, predicted **(i)** p, absolute error

(j) Shape, reference **(k)** Shape, predicted **(l)** Shape, absolute error

Figure 5.13 | Flow and shape predictions around an enlarged Bezier shape. As this input shape is an outlier, the shape reconstruction is poor, and is associated with large prediction errors ($e_f = 1.316 \times 10^{-2}$). For the u, v and p predictions, the color scales are the same as for figure 5.6.



(a) Qualitative method: using the selection criterion $(e_f^*, e_s^*) = (5 \times 10^{-5}, 1.9 \times 10^{-4})$ from previous analysis, 110 bad predictions out of 120 outliers are detected.

(b) Quantitative method: using the regression line from equation 5.8, 106 bad predictions out of 120 outliers fall into the 2σ confidence interval.

Figure 5.14 | Prediction and reconstruction error on the outliers. The qualitative and quantitative methods are illustrated in (a) and (b).

qualitative method, while also matching well the $\pm 2\sigma$ interval of the quantitative method (27 enlarged shapes out of 30 are not shown due to their out-of-range MSE.). Still, a handful of outliers presenting low e_s with large e_f are noticed, indicating that the proposed methods are still missing a small amount of outliers. Overall, the qualitative method efficiently detects most of the outliers, with only 10 out of 120 bad predictions not detected (*i.e.* 91.6% of outliers detected). The $\pm 1\sigma$ interval of the quantitative method covers 85 out of 120 (70.8%) outliers, while $\pm 2\sigma$ interval covers 106 out of 120 (88.3%) outlier predictions. The results of figure 5.14b also indicate that the uncertainty levels of misplaced and enlarged inputs are partly under-estimated.

Overall, the qualitative method efficiently diminishes the risk of mis-use of the trained model, by catching more than 90% of the bad inputs. Still, it remains a limited, binary method, and by construction approximately discards 1% of the data set points. Conversely, the quantitative method also provides adequate confidence range for almost all the elements from the test set, and for nearly 90% of the outliers. However, the provided error intervals for inputs leading to very large e_f errors are underestimated. Finally, similarly to the qualitative one, the quantitative method cannot account for a handful of points presenting large e_f values in conjunction with small e_s values.

5.10 Conclusion

A twin-AE architecture for 2-D incompressible laminar flow prediction with embedded uncertainty estimation was presented. The underlying motivation was to propose a method to naturally incorporate outlier detection and uncertainty estimation in the training procedure, in order to provide a decisional tool to the potential end-user. The embedded uncertainty estimation relies on the coupling of the autoencoder for flow prediction with a second autoencoder for input reconstruction, using well-chosen skip connections. Doing so naturally enforces a quasi-linear relation between the flow prediction error and the input reconstruction error. Building on this particular trait, simple yet effective qualitative and quantitative techniques were proposed to detect outliers and provide uncertainty prediction on any input provided to the trained network.

The proposed architecture was trained on a data set of $12,000$ laminar flows around random 2-D shapes, generated using Bezier curves. After hyper-parameter selection, the correlation coefficient between the reconstruction error and flow prediction error reached 0.95 on the test set. The two methods were then tested on true outliers presenting different flaws (polygonal shapes that did not belong to the data set, shapes from the data set misplaced in the input domain, shapes significantly larger than those of the data set), and proved efficient to either reject shapes with high flow prediction errors, or provide adequate uncertainty range. Still, a handful of outliers remained undetected, or their associated uncertainty range was under-estimated. Possible improvements could be brought to these methods, either by improving the network architecture to improve the flow error/reconstruction error correlation level, or by gaining more control on the data set generation, in order to avoid the inclusion of possible outliers.

These results underline the potential of the proposed approach. Indeed, the implementation of such methods in prediction tasks can significantly lower the risk of the end-user taking decisions based on network predictions using inadequate inputs. Efforts shall be pursued for more accurate input reconstruction. From the experiments on U-Dual-AE, low-level features from its encoder bring great benefits to flow prediction. If the shape decoder of twin-AE provides equivalently beneficial features, we expect an improved flow prediction while keeping the strong correlation between its twin decoders.

6

Graph convolutional neural networks for fast laminar flow prediction

Résumé

Au cours des dernières années, le domaine de la prédiction d'éoulements rapide a été largement dominé par les réseaux de neurones convolutifs basés sur des pixels. Pourtant, l'avènement récent des réseaux de neurones convolutifs sur graphes (GCNN) a attiré une attention considérable dans la communauté de la dynamique des fluides numérique. Dans cette contribution, nous proposons une structure GCNN comme modèle subrogé pour la prédiction d'écoulement laminaire autour d'obstacles bidimensionnels. Contrairement à la convolution traditionnelle sur les images, la convolution sur graphe peut être directement appliquée sur des maillages triangulaires ajustés à la géométrie, ce qui permet un couplage facile avec les solveurs CFD. Le modèle GCNN proposé est calibré sur un ensemble de données composé de flux laminaires calculés par CFD autour de $2,000$ formes 2-D aléatoires. Les niveaux de précision sont évalués sur des champs de vitesse et de pression reconstruits autour d'obstacles hors entraînement, et sont comparés à ceux des architectures U-net, en particulier dans la zone de la couche limite.

6.1 Introduction

The use of convolution in neural networks was originally proposed in the context of image processing [LB+95]. In practice, CNNs can be used to extract local features from grid-based data (including images) by sliding trainable kernels over an input image and performing convolution at each pixel. Yet, when exploiting CNNs in the context of flow predictions, the use of uniform cartesian grids can become a hindrance, due to their poor intrinsic geometrical representation and large associated computational cost. As is seen in previous chapters, CNNs can perform well on flow prediction tasks, but present issues in obstacle reconstruction and boundary layer accuracy. For external and wall-bounded flows, body-fitted or unstructured meshes are usually adopted, in order to respect the geometric characteristics and to accurately capture the physics of the boundary layer. When coupled to CNN-based neural network models, CFD data has to be interpolated from the mesh representation to *ad-hoc* cartesian grids, before being projected back on the mesh, thus leading to precision loss and extended computational load. In [GSW21], Gao *et al.* exploit elliptic coordinate transformation to map from the irregular physical domain to a regular reference domain. This coordinate transformation allows the authors to use a physics-informed CNN to solve partial differential equations in the reference domain, the physical solution in the irregular domain being obtained through an inverse transformation. Compared to fully-connected physics-informed neural networks, the proposed method converges faster and reaches a better accuracy.

A possible alternative consists in designing efficient convolution operations on graph structures, and to apply them directly on unstructured meshes. Extending convolutional neural networks to graphs is of great interest, as many CFD solvers are based on finite element/volume methods, which use unstructured meshes for discretization. In this regard, the design and use of trained graph neural network models can significantly enrich CFD workflows by being used for specific tasks during the resolution process, while still relying on the standard CFD mesh.

Generally speaking, a graph $\mathbf{G}(V, E)$ is composed of a set of nodes V and a set of edges E, representing the connectivity of nodes. A node-level/edge-level feature vector represents the state of each node/edge. The goal of a graph convolution operation is to update the feature vectors through aggregating neighboring information. Two types of graph convolution are to be distinguished, namely spectral convolutions, and spatial convolutions. By means of graph Fourier transform, spectral convolution turns convolution into product between the entire graph's feature matrix and the convolutional kernel [KW17]. In [DABPEK20], the authors combine differentiable PDE solvers and graph convolutional networks (GCN) to perform flow prediction around 2-D airfoils. An initial flow field is first obtained by CFD simulation on a coarse mesh, and is then passed to a GCN for a prediction on a refined mesh. This hybrid method provides better results than coarse CFD simulation alone, and is significantly less computationally-intensive than running a full CFD simulation on a refined mesh. However, spectral convolution techniques are based on a global Fourier transform of the entire graph, and therefore require a fixed graph structure. Hence, spectral convolution has thus limited application in real-world CFD sceneries, and are *de facto* excluded from contexts including mesh adaptation.

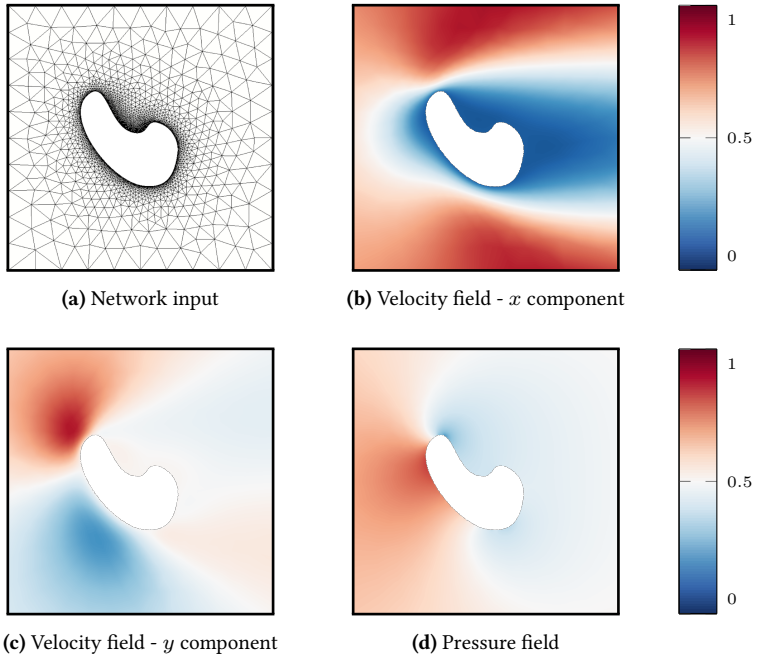
Spatial convolution is more similar to traditional convolution on cartesian data: a convolution kernel operates locally on the node and its neighbours, the kernel parameters being re-used across the entire mesh. A popular implementation of this concept is the message-passing neural network (MPNN), proposed by Gilmer *et al.* for quantum chemistry [GSR+17], which was presented in section 2.2.4. In [OMHF20], a variant of MPNN, called GraphSAGE [HYL17], is applied to drag force prediction from given laminar velocity field around airfoils. On a dataset with 1550 different airfoils and 21 angles of attack, the graph convolution model displays better performance than other machine learning methods. Battaglia *et al.* [BHB+18] propose an alternative method, called graph network (GN) blocks, for general multiple-step graph convolution. A GN block first passes messages from nodes to edges through an edge

convolution kernel, before updating the edge features. The new edge features are then aggregated to the nodes, through sum or other permutation invariant operations, as the edge messages. A node convolution kernel then takes the old node features and edges messages to obtain the new node features. In [SGGP+20], the authors use graph networks-based simulators to learn physical systems of particles, including fluids, rigid solids and deformable materials. The proposed method shows high performance in reconstructing the results of mesh-free solvers. Pfaff *et al.* [PFSGB21] apply graph networks to mesh-based simulations, including incompressible flow around cylinders and compressible flow around airfoils. The proposed neural network is trained to be an accurate incremental simulator, with capability in adapting the mesh. This method also generalizes well to mesh shapes and mesh sizes which are not present in the training set. A slightly different spatial convolution considers parameterized weight functions to aggregate the neighbouring information [MBM+17]. The features and relative position of each node pair are used to compute an edge weight through some trainable weight function, and a single weight function is applied to the entire graph inside one convolution. Since the weight function depends on node features, different aggregation patterns would be obtained at different locations of the graph. Xu *et al.* [XSSZ21] applied this method to adjoint vector modelling for 2-D flow around airfoils. The proposed weight function depends on the edge length and its angle to flow's principle direction. By training a graph convolutional neural network with such weight functions, the authors get similar adjoint vectors to traditional adjoint methods. The predicted adjoint vectors are further used for shape optimization, with resulting shapes being very close to that obtained with direct adjoint methods.

In this chapter, a graph convolutional neural network (hereafter denoted by GCNN) is implemented for laminar flow prediction around random 2-D shapes on triangular meshes. In section 6.2 the data set used in this chapter is presented. In section 6.3, insights about the convolution block and graph neural network architecture are provided. The training details and a global evaluation of the model are provided in section 6.4. In section 6.4.2 and 6.4.3, the flow predictions around a cylinder and an airfoil are compared to baseline CFD results. In particular, the velocity profiles and surface pressure distribution are carefully studied. In order to further show the potential of our method, drag forces are computed from the boundary layer information, and compared to the CFD results in section 6.4.4. After that, a hyper-parameter study is presented in section 6.5 to show the importance of GCNN's architecture design. Then the proposed GCNN approach is compared with the U-net approach in terms of accuracy and computational cost in section 6.6. Finally in section 6.7, as a proof-of-concept of the physics-informed approach, a physics-informed GCNN is trained to predict the flow around the cylinder in an unsupervised approach. The chapter is concluded in section 6.8, and several future possible developments are given.

6.2 Dataset

The data set of $2,000$ random shapes used in section 4.1 is employed again for the graph convolutional neural networks. As we are mainly concerned with the flow patterns near the obstacles, a small domain containing the obstacle is used for the neural networks's inference task, given that a converged triangular mesh such as that presented in figure 3.5b can have more than 30.000 nodes. For each shape, a body-fitted triangular mesh is generated for the domain $[-2, 2]\text{m} \times [-2, 2]\text{m}$. The mesh size is 0.01m on the obstacle's surface, which is the same as the immersed meshes used for CFD simulation. The element size grows to 0.4m at the border of the domain. Depending on the complexity of the shapes, the number of nodes varies from $2,000$ to $4,000$. The velocity and pressure fields from CFD simulations are projected onto these meshes through linear interpolation (see figure 6.1). For the learning phase, normalization and non-dimensionalization over the entire data set are also applied, mapping separately the two coordinates



(a) Network input **(b)** Velocity field - x component

(c) Velocity field - y component **(d)** Pressure field

Figure 6.1 | Triangular mesh, velocity field and pressure field for a dataset element. Body-fitted triangular meshes in a smaller domain (6.1a) are used for training the neural network, along with the interpolated velocity field (6.1b, 6.1c) and pressure field (6.1d).

components, the two velocity components and the pressure into the $[0, 1]$ range. The whole data set is split into a training set with 1600 shapes, a validation set and a test set both with 200 shapes.

6.3 Network architecture

6.3.1 Convolution block

The basic component of the graph convolutional neural network is the convolution block. Here, the proposed convolution block has two components: a two-step graph convolutional layer adapted from the literature [BHB+18, SGGP+20, PFSGB21], followed by a two-step smoothing layer. In this section, the feature matrices on the nodes and the edges are respectively denoted by $\mathbf{X}_V \in \mathbb{R}^{N_V \times d_V}$ and $\mathbf{X}_E \in \mathbb{R}^{N_E \times d_E}$, with N_V and N_E being the number of nodes and edges. d_V and d_E are the dimensions of feature vectors on nodes and edges, which are denoted by $\mathbf{x}_v \in \mathbb{R}^{d_V}$ and $\mathbf{x}_e \in \mathbb{R}^{d_E}$.

Convolutional layer The convolutional layer propagates node-level messages to edges, then aggregates the new edge features and updates the node features according to the following rules:

$$\begin{aligned} \mathbf{x}'_e &= \mathbf{f}_e \left(\frac{\mathbf{x}_{v_1} + \mathbf{x}_{v_2}}{2}, \frac{|\mathbf{x}_{v_1} - \mathbf{x}_{v_2}|}{2}, \mathbf{x}_e \right), \\ \mathbf{x}'_v &= \mathbf{f}_v \left(\mathbf{x}_v, \sum_{e_i \in N(v)} \mathbf{x}'_{e_i} \right), \end{aligned} \tag{6.1}$$

where v_1 and v_2 are the two nodes connected by edge e, and $N(v)$ is the set of neighbouring edges around node v. For both convolution kernels $\mathbf{f}_e$ and $\mathbf{f}_v$, a fully-connected feedforward neural network with one

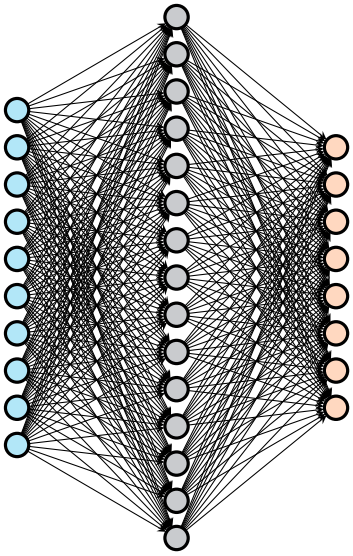


Figure 6.2 | Fully-connected feedforward neural network used as convolution kernel. The vectors on the right hand side in equation (6.1) are concatenated as the inputs of the convolution kernel. In edge convolution, the input dimension is $2d_V + d_E$, while in node convolution, the input dimension is $d_V + d'_E$. The number of hidden nodes is always 128.

hidden layer is used (see figure 6.2), the number of neurons in the hidden layer being set to 128. The number of neurons in the output layer remains modifiable, so the output dimension of the kernel can be customized.

During the edge convolution step, symmetric node features $\frac{\mathbf{x}_{v_1}+\mathbf{x}_{v_2}}{2}$ and $\frac{|\mathbf{x}_{v_1}-\mathbf{x}_{v_2}|}{2}$ are preferred over the raw features $\mathbf{x}_{v_1}$ and $\mathbf{x}_{v_2}$ in order to preserve permutation invariance. The summation in the node convolution step is also invariant to permutation of neighbouring edges. As a consequence, the proposed convolution steps (6.1) are insensitive to nodes and edges indexing.

Smoothing layer The smoothing layer perform an average operation on the output graph. The averaging kernel implemented on triangular meshes is divided into two steps:

$$\begin{aligned} \mathbf{x}''_e &= \frac{\mathbf{x}'_{v1} + \mathbf{x}'_{v2}}{2}, \\ \mathbf{x}''_v &= \frac{1}{\operatorname{card}(N(v))} \sum_{e_i \in N(v)} \mathbf{x}''_{e_i} \end{aligned} \tag{6.2}$$

The smoothing layer is technically also a convolutional layer. The motivation of adding this layer is not for message propagation, but to reduce the spatial variability of the node features. It weighs down the feature map of a convolutional layer through neighbouring compensation. A sequence of convolution-smoothing blocks can hence regularize the output of the networks to reduce the number of local minima of the loss function. The smoothing layer is beneficial to further reducing the training loss and producing continuous flow fields. A similar idea is found in the regularized pooling operation used for convolutional layers on images [OHZU20]. The final output of a convolution block is the new edge feature matrix $\mathbf{X}'_E \in \mathbf{R}^{N_E \times d'_E}$ and the smoothed new node feature matrix $\mathbf{X}''_V \in \mathbf{R}^{N_V \times d'_V}$.

6.3.2 Network architecture

The proposed GCNN architecture is presented in figure 6.3. As can be seen, the node input is composed of three graphs respectively representing (i) the obstacle boundary, (ii) the x coordinates and (iii) the y coordinates, while the average of the node pairs of each edge is regarded as the edge input. Eight convolution-smoothing blocks are stacked to form the graph convolutional neural network, followed by a 1×1 convolution as the output layer. Unless stated otherwise, the intermediate edge and node features dimensions in each graph convolution layer are respectively $(4, 8, 16, 32, 64, 64, 32, 16)$ and $(8, 16, 32, 64, 64, 32, 16, 8)$, as indicated in parenthesis in the graph convolution layers of figure 6.3 (a study of the impact of the features dimensions can be found in section 6.6). The final 1×1 convolutional layer transforms node features in $\mathbb{R}^8$ into vectors in $\mathbb{R}^3$, which represent the predicted velocity and pressure. This symmetric architecture is similar to the encoder-decoder structure used in the U-nets and twin-decoder architectures (see sections 4.1.2 and 5.3).

An important component of the proposed architecture is the use of skip connections from the input graph to the convolutional blocks. The coordinates of the nodes are concatenated to the node features after each smoothing layer. These skip connections provide spatial informations to the edge convolution steps in equation (6.1). The edge convolution hence takes into account the position of the edge center and relative position of the node pairs, allowing the convolution kernel to display different patterns at different locations in the domain.

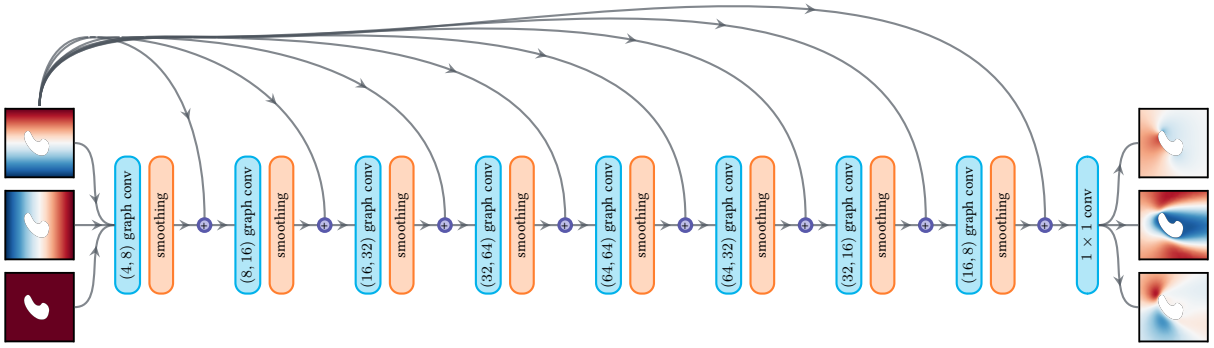


Figure 6.3 | Proposed network architecture. The provided input is composed of three graphs respectively representing (i) the obstacle boundary, (ii) the x coordinates and (iii) the y coordinates. The network has an encoder-decoder structure: in the encoder, the amount of features is doubled after each smoothing step, while in the decoder, it is halved after each smoothing step. For each graph convolution layer, the number of edge and node features are indicated in parenthesis. The network is terminated with a 1×1 convolutional layer to obtain the three-channel output layer, holding respectively the u and v components of the velocity, and the pressure field p. The skip connections allow to inject coordinate informations from the input layer at each edge convolution step of the architecture.

6.4 Training and evaluation

The presented architecture in section 6.3.2 is trained against the dataset described earlier, and evaluated by various standards. Yet, due to their major outcome on the final performance of the model, some hyper-parameters deserve further study. Hence, the impact of the choice of activation function, the presence of smoothing layer, the number of convolution-smoothing blocks and the number of hidden nodes in the convolutional kernel are studied separately in section 6.5. For now, the results of the following section represent a proof of concept of the graph convolutional neural networks as surrogate models in computational fluid dynamics.

6.4.1 Training history

With architecture described in section 6.3.2, the proposed network has $217,853$ trainable parameters, which are calibrated by minimizing an objective function defined on the predicted and the reference flow fields. The mean absolute error (MAE) is adopted as the loss function, and is defined as:

$$\mathcal{L} = \frac{1}{3N_V} \sum_{i=1}^{N_V} \left(\left| u_i^{\text{pred}} - u_i^{\text{ref}} \right| + \left| v_i^{\text{pred}} - v_i^{\text{ref}} \right| + \left| p_i^{\text{pred}} - p_i^{\text{ref}} \right| \right),$$

with the considered fields being normalized to $[0, 1]$, as stated earlier. An in-house TensorFlow implementation is used for the network architecture and training. The convolutional kernels all have 128 hidden neurons, and use the swish function [RZL18] as nonlinear activation:

$$\sigma(x) = \frac{x}{1 + e^{-x}} \tag{6.3}$$

which will be compared with ReLU in later sections. The weights and bias of the kernels are initialized using the Glorot-normal method. The network is trained using the Adam optimizer for at most $1,000$ epochs, with mini-batches of size 64 to limit the required computational resources. Here one mini-batch is a large graph containing 64 disjoint subgraphs, which is different from the mini-batch constitution in CNNs. Since the graphs have different number of nodes, they cannot be simply stacked to form a new axis like images. The learning rate follows a pre-defined epoch-wise decay schedule described as:

$$l_r^e = \frac{l_r^0}{1 + \gamma e}, \tag{6.4}$$

where l_r^e is the learning rate used during epoch number e, and γ is the decay rate. In the experiments, both the initial learning rate and the decay rate were set equal to 0.002. To prevent overfitting, an early stopping criterion is applied: the training is terminated if the average MAE on the validation set is not improved in 60 epochs.

The network is trained 5 times with different random initializations. In the best case, after 914 training epochs, the training and validation losses converge respectively to 7.43×10^{-3} and 7.61×10^{-3}, while the MAE over the test set amounts to 7.70×10^{-3}, as shown in figure 6.4. The proposed model requires approximately 11.96 seconds per epoch on a Tesla V100 GPU card, while it takes only 14 seconds to predict the flow fields around the 200 obstacles in the test set. From the MAE distribution over the test set, shown in figure 6.5, it appears that most elements in the test set are accurately predicted, with a limited amount of shapes exceeding a MAE level of 0.01. This underlines the capabilities of the model to generalize on out-of-training data.

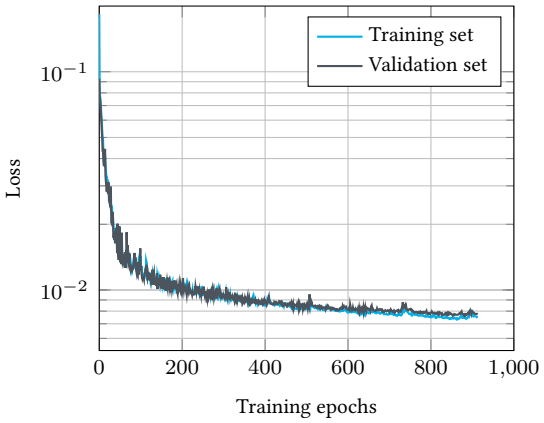


Figure 6.4 | Training history of the model. In fewer than $1,000$ training epochs, the training loss and validation converge. No obvious overfitting is observed.

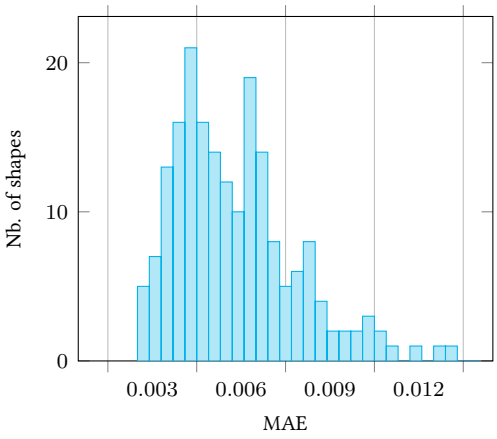
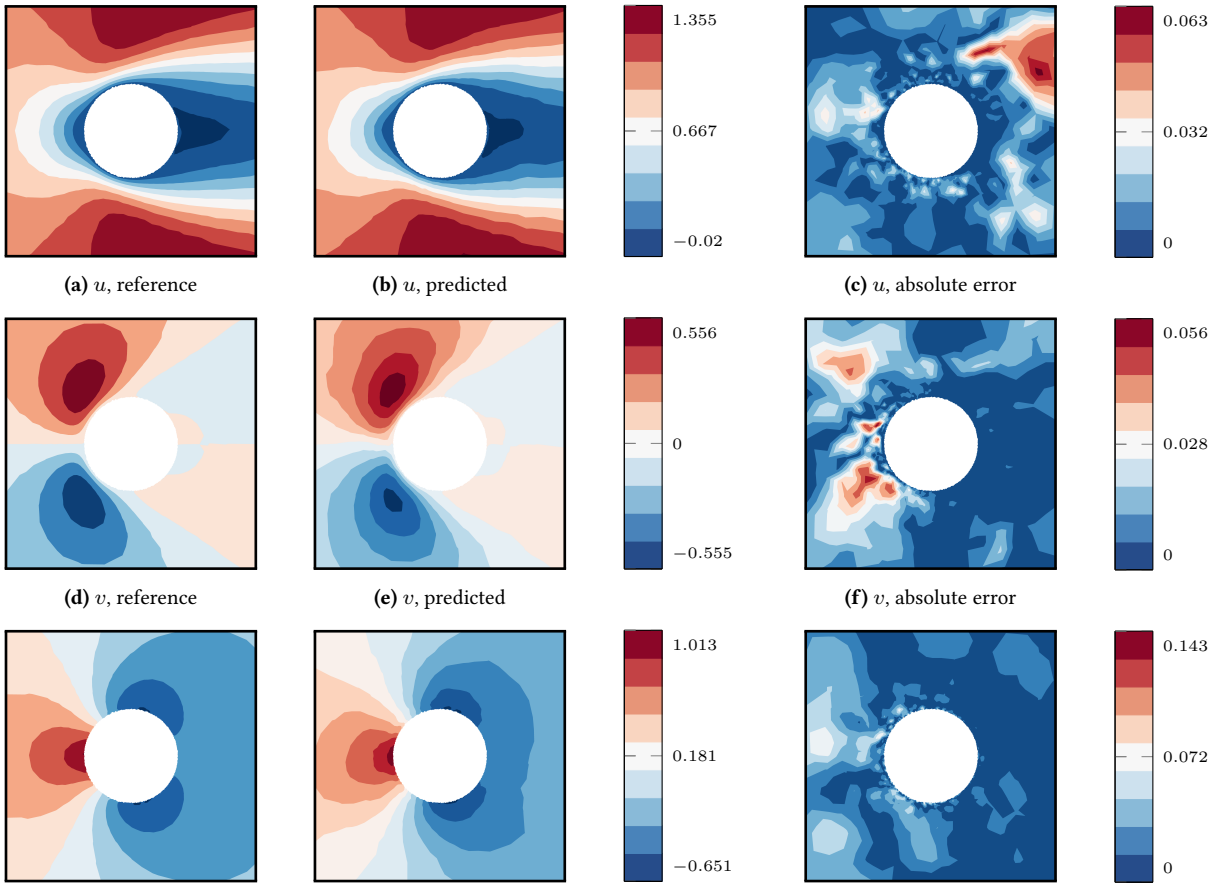


Figure 6.5 | Distribution of the MAE over the test set. The test set contains 200 random Bézier shapes. Only 13% of the shapes present a MAE value larger than 0.01.

6.4.2 Flow prediction around a cylinder

To test the trained GCNN model on an out-of-training input, we consider the velocity and pressure predictions performed on a 2-D cylinder with radius 0.75m placed at the origin, and embedded in a body-fitted triangular mesh with $2,447$ nodes, with a 0.01m mesh size on the cylinder. The MAE of the predicted velocity and pressure fields is 4.3×10^{-3}, which can be explained by the geometric characteristic of the cylinder being close to that of the Bézier shapes from the training set. In figure 6.6, we observe almost symmetric fields with respect to the $x-$axis for both the velocity and the pressure. Although the shapes in the training set are not necessarily symmetric, the trained GCNN is is able to reconstruct a symmetric flow around a cylinder. The velocity profiles and surface pressure distribution displayed in figure 6.7 further confirm the symmetric predictions around a cylinder.



(a) u, reference **(b)** u, predicted **(c)** u, absolute error

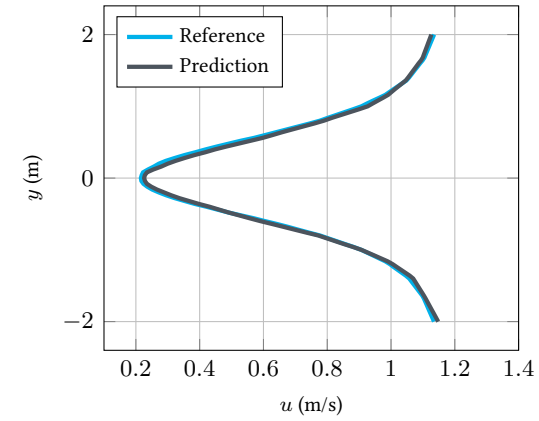
(d) v, reference **(e)** v, predicted **(f)** v, absolute error

(g) p, reference **(h)** p, predicted **(i)** p, absolute error

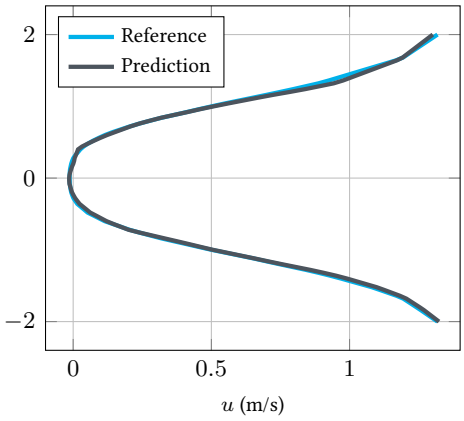
Figure 6.6 | Flow and shape predictions around a cylinder using a GCNN model. The normalized velocity and pressure fields outputted by the network are rescaled to their physical ranges and non-dimensionalized for better demonstration. The reference (6.6a, 6.6d, 6.6g) and predicted fields (6.6b, 6.6e, 6.6h) are very close, with no specific pattern in the MAE maps (6.6c, 6.6f, 6.6i).

6.4.3 Flow prediction around a NACA12 airfoil

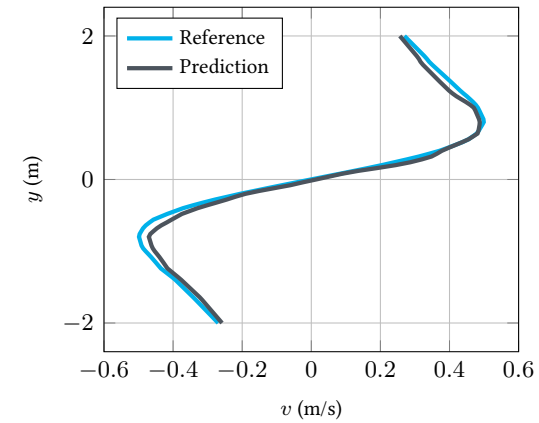
The NACA airfoils from the UIUC database are among the most studied geometries in the CFD community. In the context of this contribution, the prediction on airfoils is regarded as good test of the



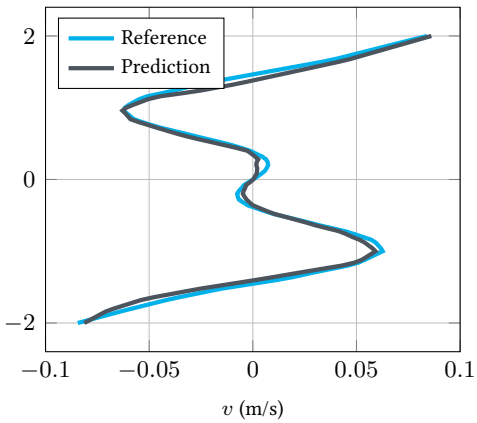
(a) u profile at $x = -1$



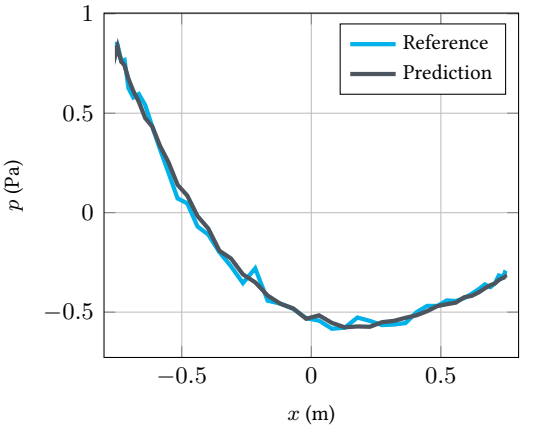
(b) u profile at $x = 1$



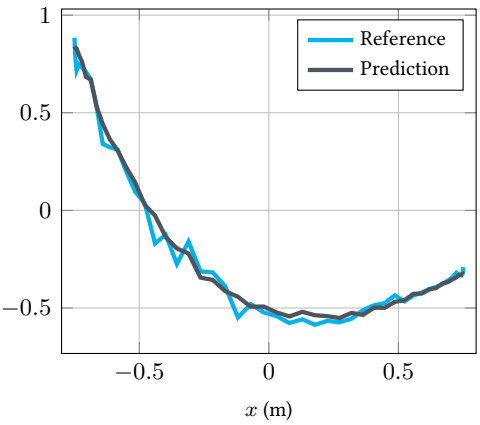
(c) v profile at $x = -1$



(d) v profile at $x = 1$



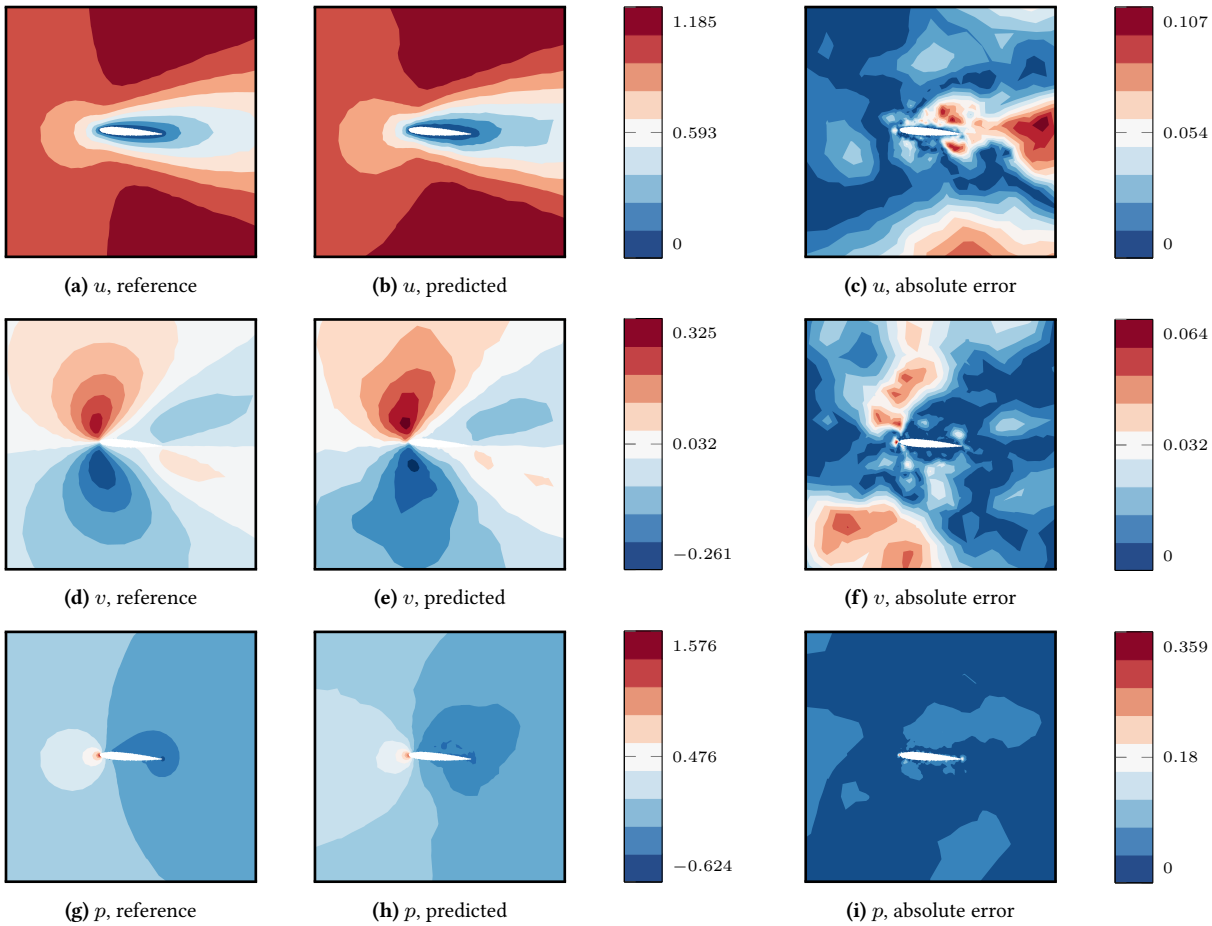
(e) Pressure distribution on the lower surface



(f) Pressure distribution on the upper surface

Figure 6.7 | Velocity and pressure profiles around the cylinder. The velocities are recorded at $x = -1$ (6.7a, 6.7c) and $x = 1$ (6.7b, 6.7d), and display the expected symmetry with respect to the x axis. The pressure is recorded on the lower (6.7e) and upper surfaces (6.7f) of the cylinder. The wiggles in the referential pressure profiles are caused by obstacle boundary resolution by the immersed boundary method, which employs an element size equal to 0.01m. These numerical artefacts could be reduced using smaller boundary elements, although we find they do not impact the neural networks training.

model generalization capabilities on out-of-training shapes, as their geometry is very different from that of Bézier random shapes. The demonstrated airfoil is NACA12, with an angle of attack equal to $4°$ and a chord length equals to 1m. The airfoil is embedded into a triangular mesh made of 1489 nodes. The referential flow fields are from a CFD resolution with the same configurations as the Bézier shapes and the cylinder. The MAE of the predicted fields obtained from the GCNN model is 1.04×10^{-2}, which is similar to the worst prediction levels observed in the test set. Yet, the predicted fields presented in figure 6.8 display reasonable physical features. Although the model overestimates the velocity magnitude, it successfully detects the locations where the vertical velocity changes sign or attains local minimum and maximum, as can be further observed in the plots of figure 6.9. More, the surface pressure distribution is recovered by the model, although with an underestimated pressure magnitude on the leading edge and the trailing edge. The incorrect velocity and pressure magnitude both result from the dissimilarity between flow around the airfoil and flow around the Bézier shapes in the training set. Since the GCNN model in this contribution is trained under the supervised learning framework, it should be conservatively applied to geometries that are not too different from that of the training dataset.

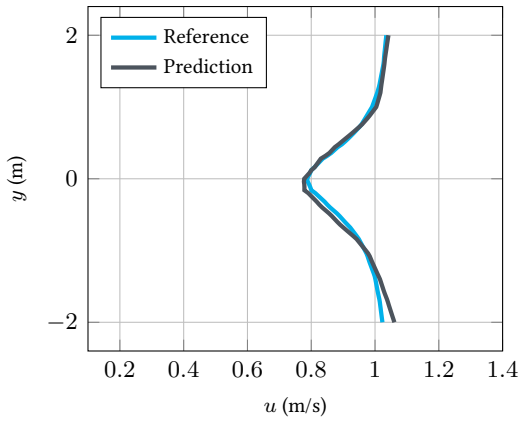


(a) u, reference **(b)** u, predicted **(c)** u, absolute error

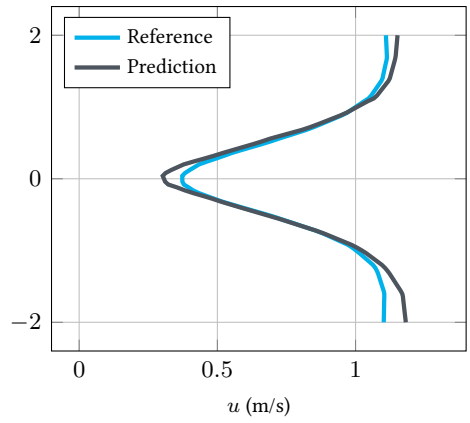
(d) v, reference **(e)** v, predicted **(f)** v, absolute error

(g) p, reference **(h)** p, predicted **(i)** p, absolute error

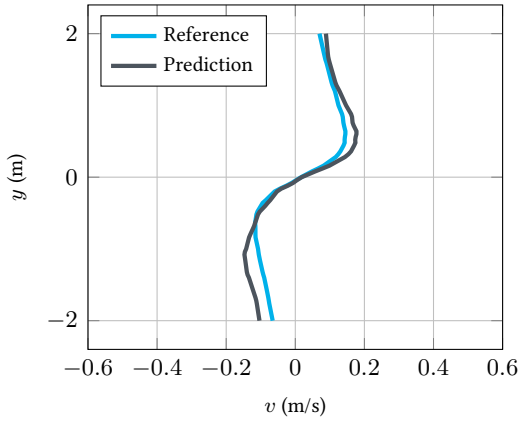
Figure 6.8 | Flow predictions on a NACA12 airfoil using a GCNN model. Although the field amplitudes are not correctly evaluated, the velocity (6.8a, 6.8b, 6.8c, 6.8d, 6.8e, 6.8f) and pressure maps (6.8g, 6.8h, 6.8i) display reasonably physical features, such as sign changes in the vertical velocity or localized high pressure area.



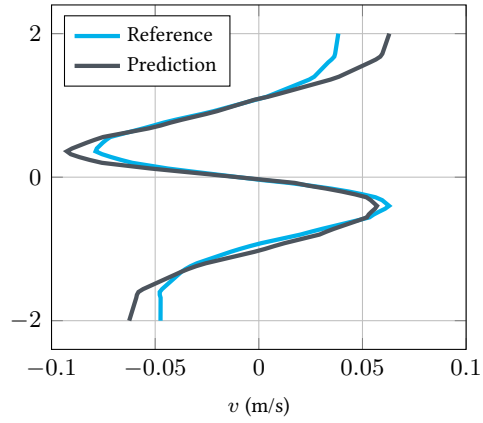
(a) u profile at $x = -1$



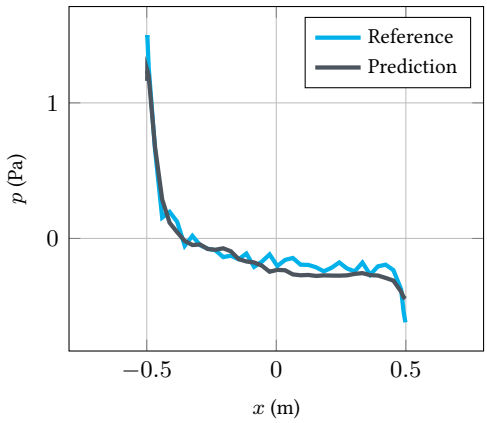
(b) u profile at $x = 1$



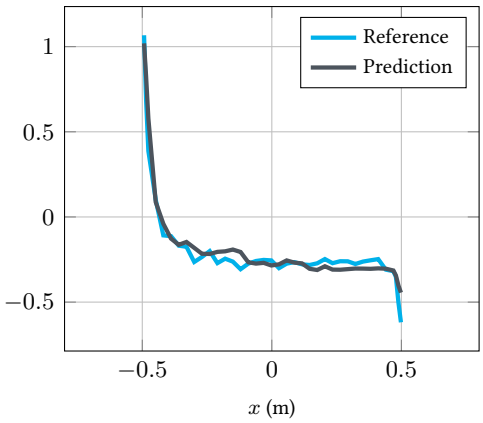
(c) v profile at $x = -1$



(d) v profile at $x = 1$



(e) Pressure distribution on the lower surface



(f) Pressure distribution on the upper surface

Figure 6.9 | Velocity and pressure profiles around the NACA12 airfoil. The velocities are recorded at $x = -1$ (6.9a, 6.9c) and $x = 1$ (6.9b, 6.9d), and display significant errors around the trailing edge. With significant discrepancy of velocity magnitude, the physical pattern of the velocity profiles is still well recovered. The pressure is recorded on the lower (6.9e) and upper surfaces (6.9f) of the airfoil, with remarkable error at the leading and trailing edges.

6.4.4 Drag force prediction

In this section, the accuracy of the drag force prediction is used as a measure of the quality of the predicted boundary layer. The drag force at Re$=10$ is obtained by integrating the edge-wise contribution of pressure and viscous force on the obstacle, as shown in equation (6.5).

$$F_D = \mathbf{e}_x \cdot \oint_S (p \cdot \mathbf{n} + \mu \frac{\partial v_s}{\partial \mathbf{s}}) \, \mathrm{d}s, \tag{6.5}$$

with $\mathbf{e}_x$ being the unit vector in the x-direction, $\mathbf{n}$ and $\mathbf{s}$ repectively the inward- and outward-pointing unit normal vectors, and $\frac{\partial v_s}{\partial \mathbf{s}}$ is the normal gradient of the tangent velocity. The drag forces computed on the 200 shapes of the test set are compared with their reference drag values in figure 6.10. Overall, a good agreement is observed, with an average relative error of 3.43%, and a coefficient of determination equal to 0.987, which is in line with the results of the GraphSage method [OMHF20]. It must be noted that, in the literature, neural networks are usually directly trained on adequate databases to predict drag or lift coefficients. With the drag force computed by reconstructed flow fields on triangular meshes, it is possible to apply the GCNN model to shape optimization in future works.

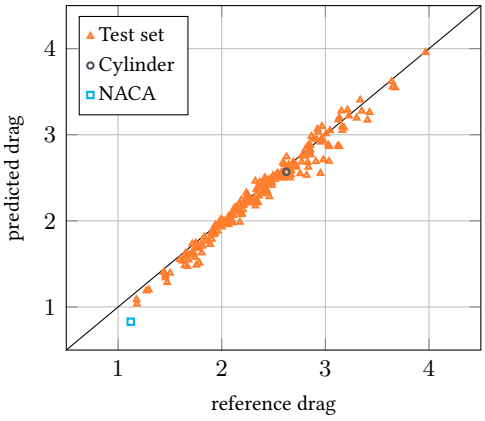
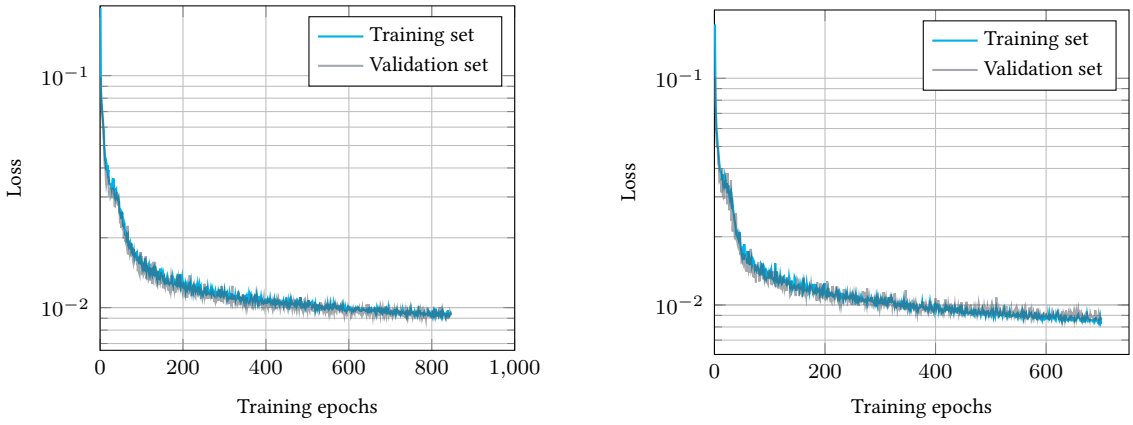


Figure 6.10 | Drag force computed from the boundary layer reconstruction obtained with the graph neural network. The included data contains the 200 Bézier shapes from the test set, along with the cylinder and the NACA0012 airfoil considered in the previous sections. Excellent agreement is observed on most shapes from the test set, with an average relative error of 3.43%. While the cylinder drag is accurately computed, the airfoil drag is off by 26.1% from its reference value.

6.5 Hyper-parameter study

6.5.1 Smoothing layers

To evaluate the advantages of smoothing the outputs of each convolutional layer, an architecture without smoothing layers is trained for comparison. Its architecture is the same as figure 6.3, except that all the smoothing layers are removed. All the training settings remain unchanged, with the network being trained 5 times with different initializations. The model with smallest validation loss is trained 846 epochs before early stopping, as is shown in figure 6.11a. The validation loss reaches 8.93×10^{-3} at the end of training, while the averaged MAE reads 9.18×10^{-3} on the test set. This compares with the reference model (including smoothing layer), which was trained for 914 epochs, resulting in a test loss equal to 7.70×10^{-3}. Therefore, the smoothing layers brings a 15.8% reduction of the prediction error on the test set. In terms of training time, the model without smoothing layers requires 8.5 seconds for training one epoch, leading to 1.99 hours training time. The model with smoothing layers took 3.04 hours, due to the supplementary matrix arithmetics from the smoothing operation (see equation (6.2)).



(a) No smoothing layers

(b) ReLU activation

Figure 6.11 | Training history of the models without smoothing layers (6.11a) and using ReLU activation (6.11b). Both models show inferior performance than the referential model.

6.5.2 Activation function

The swish function is compared to the ReLU activation function when using the same architecture as figure 6.3. The training ends after 701 epochs, taking 1.94 hours on the Tesla V100 GPU card (10.12 seconds per epoch). From ReLu-model's training history, we see the convergence slower than the swish-model. The average MAE on the validation and the test sets are respectively equal to 8.57×10^{-3} and 8.67×10^{-3}. Under the same training configurations, the accuracy of the swish-model is 11.2% higher. Therefore, the swish function represents a better choice than the ReLU one in the proposed graph convolutional neural network. The underlying mechanism of the swish function is explained in the original paper [RZL18]: both swish and ReLU are unbounded to avoid near-zero gradients during back-propagation, but the swish function is smoother than ReLU and non-monotonic: the smoothness of swish function leads to a smooth error landscape, which may make the minimum search easier; being non-monotonic means the presence of negative derivatives and richer gradient information than the ReLu activation, which is helpful to gradient-based optimization.

6.5.3 Multilayer perceptron

Since the convolutional kernel described in figure 6.2 consists in a multilayer perceptron with one hidden layer, the number of hidden and output nodes both condition its complexity. In this section, the impact of the width of the hidden layer is evaluated. As a recall, the reference model holds $217,853$ trainable parameters, using MLPs with 128 nodes in the hidden layer. Two other configurations are considered, with MLPs comprised of 32 and 64 nodes respectively, leading to two GCNN architectures with $54,845$ and $109,181$ trainable parameters. The performance of the three models are compared in table 6.1.

Table 6.1 | Model performance of different convolutional kernels. The comparison is based on the average MAE on the test set.

	Trainable parameters	**Training time (hour)**	**Test MAE**
32	$\mathbf{54,845}$	$\mathbf{1.88}$	8.46×10^{-3}
64	$109,181$	2.39	7.79×10^{-3}
128	$217,853$	3.04	$\mathbf{7.70 \times 10^{-3}}$

As can be observed, increasing the number of hidden nodes improves the GCNN's performance, while also requiring a longer training time. It is observed that using 64 hidden nodes in the MLPs represents a good balance. It must also be underlined that the GCNN still reaches a remarkable performance level with as little as 10^5 trainable parameters. As a reminder, the U-net architecture could easily reach millions of parameters while still displaying large discrepancies near the solid interface. For these reasons, GCNN architectures represent a promising machine learning approach in the context of CFD scenarios. In the following section, GCNNs' performances are compared with that of U-net architectures on similar tasks.

6.6 Comparison with U-nets

U-nets are a popular neural network architecture for end-to-end image prediction, whose performance in the context of flow prediction was evaluated in previous sections. By design, U-nets require grid-based data, such as images. As a consequence, flow fields as well as obstacle boundaries need to be projected onto a regular cartesian grid, implying geometric inaccuracies and higher data storage requirements. In this section, U-nets are quantitatively compared with GCNNs on flow prediction tasks. The U-net architecture is configured as described in section 4.1.2, being symmetric and scalable through a modification of the number of kernels in the first block, the latter being fixed at 5. The relation between the number of filters and the number of trainable parameters is shown in table 6.2. Similarly, in a GCNN, one can modify the node feature dimension of the first convolutional block while proportionally changing the other intermediate feature dimensions.

1^{st} layer	Parameters
7	1,489,162
8	1,944,763
9	2,461,080
10	3,038,113

(a) U-net

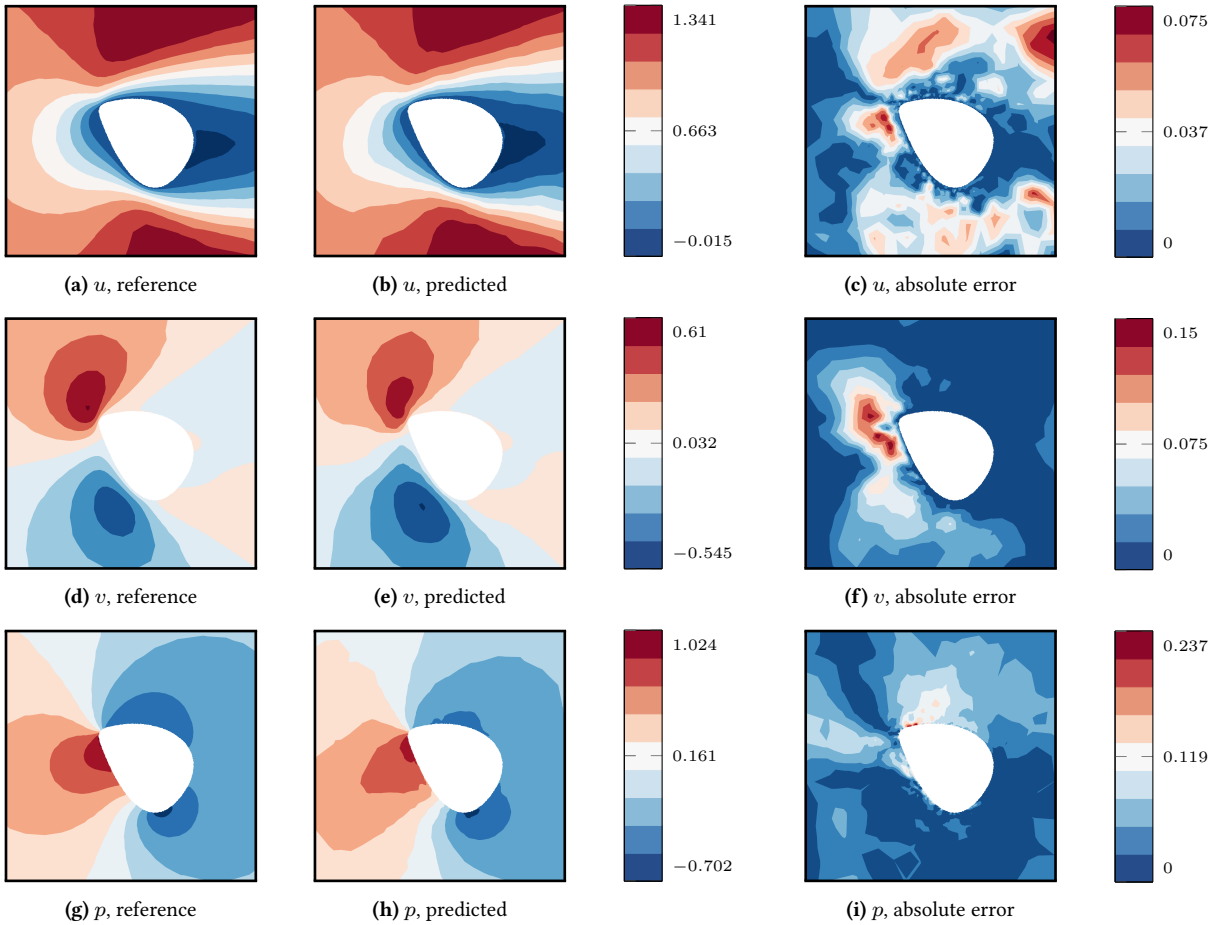
1^{st} layer	Parameters
2	61,190
4	113,411
6	165,632
8	217,853

(b) GCNN

Table 6.2 | Trainable parameters of U-nets and GCNNs. For simplification, both architectures use symmetric and scalable configurations. Hence, their complexity can be represented by the number of convolutional filters or node feature's dimension in the 1^{st} convolutional layer.

In figure 6.12, a sample prediction obtained by the proof-of-concept GCNN model described in section 6.4.1 is presented. As can be observed, the prediction not only presents smaller error levels, but also displays good performance in the boundary layer region, which was not the case of U-net predictions (see figure 4.6). More, the presence of the obstacle is naturally accounted for by the GCNN approach, contrarily to the U-net approach. To perform a fair quantitative comparison with GCNN prediction, the results obtained with the U-net model are interpolated on the triangular meshes. The average MAE on the test set is used for comparison, and is displayed in figure 6.13a as a function of the number of trainable parameters, both for GCNN and U-net. As can be observed, the GCNN requires fewer trainable parameters than the U-net (approximately 1 order of magnitude) to reach a similar accuracy, highlighting the cost-efficient of GCNN in field reconstruction compared to grid-based techniques. Indeed, the GCNN is able to exploit local grid refinement or coarsening to save degrees of freedom, which is a central strength of finite-element and finite-volume CFD methods. It is also worth mentioning that the number of kernels in the first convolutional layer has a positive impact on the U-net's accuracy evaluated on triangular meshes, while it does not change the MAE evaluated on the cartesian grids (see figure 4.7b). We can thus conclude that using more kernels makes the U-nets more aware of the discontinuity at the solid interface. In terms of memory consumption, the GCNN requires 1.05 GB to import the data set (including the nodes coordinates and connectivity, and the velocity and pressure fields), which must be compared to the 13.5 GB required by the U-net architecture for the same task. Hence, the data used for GCNN is more likely to fit into standard GPU cards.

Yet, the main observed drawback of GCNNs is the larger training time required to reached a converged model. Compared to a traditional convolution, the $4-$step graph convolutional block presented in equations (6.1) and (6.2) requires heavy matrix operations, which significantly increases the time required

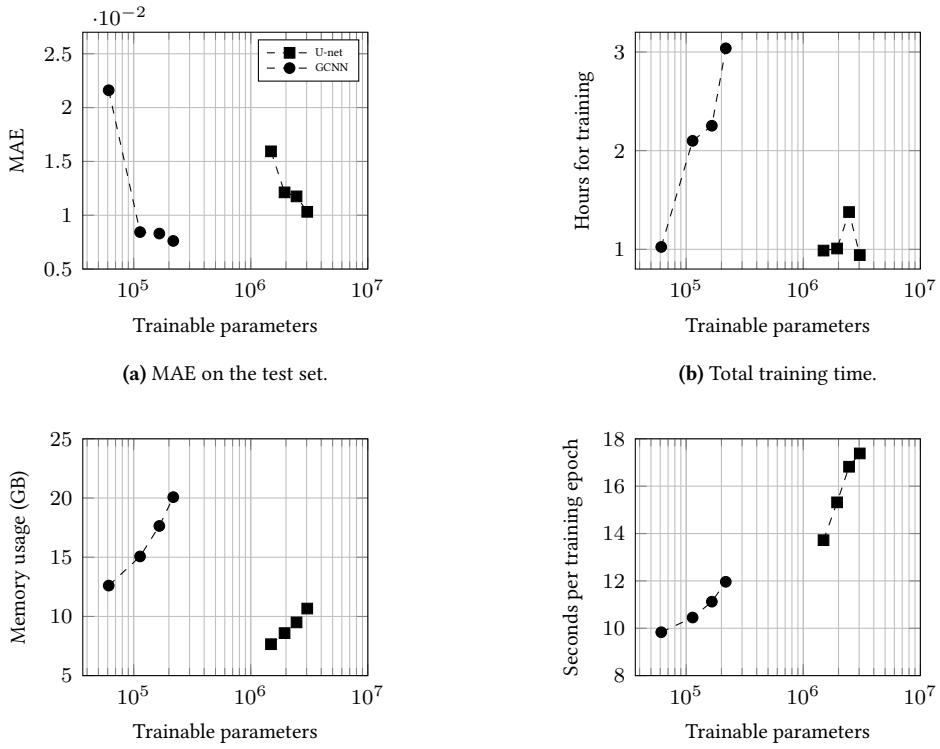


(a) u, reference **(b)** u, predicted **(c)** u, absolute error

(d) v, reference **(e)** v, predicted **(f)** v, absolute error

(g) p, reference **(h)** p, predicted **(i)** p, absolute error

Figure 6.12 | Flow predictions on a Bézier shape from the test set using a GCNN model. The horizontal velocity (6.12a, 6.12b, 6.12c), vertical velocity (6.12d, 6.12e, 6.12f) and pressure fields (6.12g, 6.12h, 6.12i) display similar error levels. The concerned GCNN has 217, 853 trainable parameters in total. The maximum absolute error is much smaller than the U-net method, with no specific pattern displayed in the error maps.



(a) MAE on the test set.

(b) Total training time.

(c) Required memory for gradient backpropagation.

(d) Training time per epoch.

Figure 6.13 | Performance comparison between U-net and GCNN in terms of test set MAE (6.13a), total training time (6.13b), memory usage (6.13c) and per-epoch training time (6.13d). As can be seen, GCNNs require much fewer parameters to reach the same accuracy of U-nets. However, training GCNNs requires more memory to perform gradient backpropagation and is more time-consuming considering their smaller model size.

for a forward network pass. In order to implement automatic differentiation, the generated intermediate variables are not deleted from the memory until one whole back-propagation of the gradients in all the blocks is finished. The required memory to store the intermediate variables are compared in 6.13c when batch size is 64. In spite of the small input data size, GCNNs still require a large amount of memory for the intermediate variables in the training process. The limitation of memory thus does not support using large batch size, which also slows the training. In figure 6.13b and 6.13d, the training time of both methods are compared when batch size is 64. Considering the input size as well as the network's complexity, training GCNNs is more expensive than training U-nets. However, as a newer method than traditional CNNs, there is a lot of room for accelerating the training process.

6.7 Physics-informed graph convolutional neural networks

In the previous sections, the GCNN method has been used to produce surrogate models respecting geometry constraints. To make neural networks aware of physical constraints, this section slightly explores the recent domain of physics-informed neural networks, and this is an opening to future research topics.

6.7.1 Physics-informed neural networks

As was shown in the flow prediction around the NACA airfoil, the trained GCNN model overestimates the velocity magnitude and underestimates the pressure, due to the different geometry characteristics between the airfoil and the Bézier shapes. To improve the model's performance on a geometry lying outside of the boundary of the training set, one should avoid over-fitting in the training stage, which may be realized by adding the l_1 or l_2 regularization term as part of the loss function. However, the regularization rate (see equation (2.13)) remains to be tuned by the user, and it must be dependent on the validation and test set, making the calibrated model biased to the available data. On the other hand, using a larger training data set with more variability can lead to a performant model on a broader range of data, but data preparation and training time also scales linearly with the size of the data set. More, this method still does not jump out of the framework of data fitting, so the obtained model will fail somewhere and even give predictions violating physical common sense.

One alternative to ensure the physical consistency of NN-based surrogate models is the so called physics-informed neural networks (PINN). This approach embeds physical laws described by partial differential equations into the training stage, so that the model prediction approximately satisfies the physical constraints [KKL+21]. Given a spatial-temporal process constrained by the parametrized nonlinear PDE:

$$\partial_t \mathbf{v}(t,\mathbf{x}) + \mathcal{F}(\mathbf{v};\lambda) = 0,\ \mathbf{x}\in\Omega,\ t\in[0,T] \tag{6.6}$$

where $\mathcal{F}$ is a nonlinear operator parametrized by λ, there exists three scenarios to embed the constraints of equations (6.6) in the training stage.

- Proposed by Raissi *et al.* [RPK19], the solution $\mathbf{v}(\mathbf{x},t)$ is predicted by a neural network, whose weights and bias are calibrated through minimizing an objective function based on the deviation to the initial conditions, the boundary conditions and the concerned equations:

$$L = \frac{1}{N_{I/BC}}\sum_{i=1}^{N_{I/BC}} \left|\mathbf{v}(t^i_{\mathbf{v}},\mathbf{x}^i_{\mathbf{v}}) - \mathbf{v}^i\right|^2 + \frac{1}{N_{\mathcal{F}}}\sum_{i=1}^{N_{\mathcal{F}}} \left|\partial_t \mathbf{v}(t^i_{\mathcal{F}},\mathbf{x}^i_{\mathcal{F}}) + \mathcal{F}(\mathbf{v}(t^i_{\mathcal{F}},\mathbf{x}^i_{\mathcal{F}});\lambda)\right|^2 \tag{6.7}$$

  Here, $\{(t^i_{\mathbf{v}},\mathbf{x}^i_{\mathbf{v}},\mathbf{v}^i)\}_{i=1}^{N_{I/BC}}$ are a set of initial and boundary values, and $\{(t^i_{\mathcal{F}},\mathbf{x}^i_{\mathcal{F}})\}_{i=1}^{N_f}$ are a set of points where the PDE constraint $\partial_t\mathbf{v}(t,\mathbf{x}) + \mathcal{F}(\mathbf{v};\lambda) = 0$ is enforced. This approach belongs to the unsupervised framework, *i.e.* it does not require the preparation of a data set, and the trained model can not generalize to the problems with new initial/boundary conditions. However, available observation inside the domain Ω can be included to enhance the training, as is the case of [CWF+21], where 3D flow fields over heated liquid are recoverd from some experimental measurements and known physical equations. Still, this approach is different from the surrogate modeling presented in previous chapters, because our meta models are calibrated on a data set with much variability, and is expected to generalize well on new prediction tasks. Instead, it is more like a novel type of PDE solver, whose convergence and computational cost deserve more studies [MMOH20].

◇ In the framework of supervised learning, the residuals of PDEs are added to the fitting error as a regularization term, leading to a loss function in the form of:

$$l = ||\mathbf{v}_{\text{pred}} - \mathbf{v}_{\text{true}}||_{l_2}^2 + \lambda_1 ||\partial_t \mathbf{v}_{\text{pred}}(t, \mathbf{x}) + \mathcal{F}(\mathbf{v}_{\text{pred}}; \lambda)||_{l_2}^2 + \lambda_2 l_{I/BC}, \tag{6.8}$$

with $l_{I/BC}$ being the deviation to the initial and boundary conditions, and (λ_1, λ_2) being the regularization rate. The data set is composed of many solutions $\mathbf{v}$ obtained from different initial and boundary conditions, and the objective function used to train the neural network is the average of the loss function over the data set, making this method an improved version of the those already presented in the manuscript. As is already mentioned in section 2.4, Nabian *et al.* [NM20] showed the advantage of using physics-driven regularization terms by adding the residual of zero-divergence condition of the velocity to the velocity prediction error. In some literatures [ZLS20, GW20], this hybrid training strategy is also called 'physics-guided data-driven modeling'.

◇ The third method employs a 'weakly-supervised learning' strategy. It is different from the unsupervised one, as it is able to generalize to new prediction tasks; it is however not a pure supervised approach, as no ground-truth need to be prepared for the training process. Instead, a set of inputs are randomly generated, after what the training is realized through minizing the batch-wise physical loss. In [SFG+18], Sharma *et al.* obtained a CNN-based model mapping the boundary condition to the solution of the Laplacian equation on a square domain, making the CNN a solver to this boundary value problem. Similarly, Wandel *et al.* [NWK20, WWK21] trained CNNs to map the fluid state at time t to the fluid state at time $t + dt$, which is realized by generating a pool of random initial states, then minimizing the additioned residuals of the Navier-Stokes equations computed by the pool of predicted fluid states at next time step, making the CNN aware of the evolution of the dynamical system defined by Navier-Stokes equations.

6.7.2 Solving the Navier-Stokes equations by a physics-informed graph convolutional neural network

As the last work of the manuscript, and an opening for future research topics this section aims at showing the potential of physics-informed neural networks for the coupling of physics, numerics and machine learning. Following the unsupervised framework proposed by Raissi *et al.* [RPK19], we use the GCNN architecture presented in section 6.3 to recover the steady velocity and pressure fields around a 2-D cylinder, which is discretized on a triangular mesh with $N = 9,294$ nodes shown in figure 6.14.

The cylinder's radius is 0.75m, with its center placed at the origin, and the boundary conditions being the same as those imposed for the laminar flow resolution in section 3.2.1. Since we are going to obtain the steady flow fields as the outputs of a GCNN, the concerned physical and boundary constraints are:

$$\begin{cases} \rho\, \mathbf{v} \cdot \nabla \mathbf{v} - \nabla \cdot (2\eta \boldsymbol{\varepsilon}(\mathbf{v}) - p\mathbf{I}) = 0 \text{ on } \Omega, \\ \nabla \cdot \mathbf{v} = 0 \text{ on } \Omega, \\ \text{Dirichlet}(\mathbf{v}, p) = 0 \text{ on } \partial\Omega_D, \\ \text{Neumann}(\mathbf{v}, p) = 0 \text{ on } \partial\Omega_N, \end{cases} \tag{6.9}$$

where no time derivative is present. To minimize the residuals of these constraints, the following optimization problem is solved in the automatic differention environment of Tensorflow:

The input graph is the triangular mesh with the coordinates as its node features, which is transformed by the GCNN to the velocity and pressure values. Right after the GCNN's output layer, the Dirichlet boundary conditions are imposed on the flow predictions, following a series of differentiable operations:

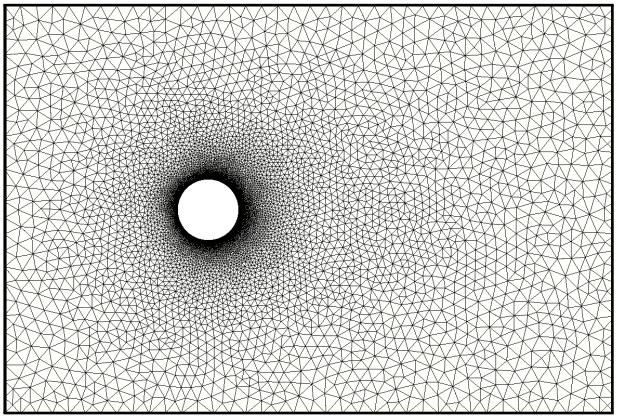


Figure 6.14 | A triangular mesh around a 2-D cylinder. In total there are $9,294$ nodes, with the element size near the solid surface being 0.01m.

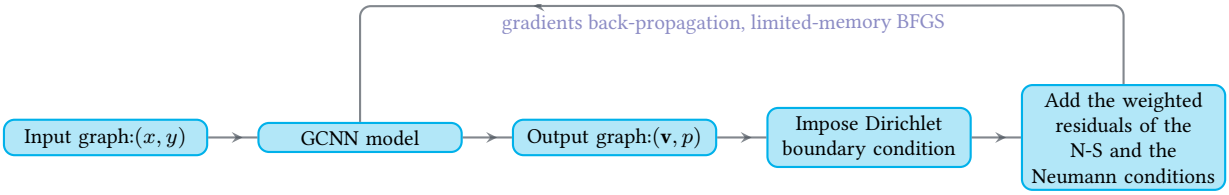


Figure 6.15 | The workflow of physics-informed graph convolutional neural network. The GCNN model, the imposition of Dirichlet boundary conditions, and the residual calculation are all differentiable operations, so that the gradients with repect to the trainable parameters can be easily obtained in the automatic differentiation framework of Tensorflow. The gradients are used by the limited-memory BFGS algorithm to update the network's parameters.

$$
\begin{aligned}
u &\leftarrow u * (1 - \mathbb{1}_{\text{in+solid}}) + u_0 * \mathbb{1}_{\text{in}}, \\
v &\leftarrow v * (1 - \mathbb{1}_{\text{in+solid+top+bottom}}), \\
p &\leftarrow p * (1 - \mathbb{1}_{\text{out}}),
\end{aligned}
\tag{6.10}
$$

where the identity fuction $\mathbb{1}$ represents a mask indicated by its subscripts. After the imposition of Dirichlet boundary conditions, the flow fields are used to calculate the residuals of the NS equations as well as the Neumann boundary conditions, with the required gradients $(\nabla \mathbf{v}, \nabla p)$ estimated by the average-based method presented in equations (2.10,2.11). Finally, the loss value is a weighted sum of the node-wise residuals, with the weight of each node determined by the local element size:

$$
w_i = \frac{\sum_{e \in \mathcal{N}(i)} \text{area}(e)}{\sum_{j \in \{1,2,...,N\}} w_j}, \text{ for } i \in \{1, 2, ..., N\}, \tag{6.11}
$$

where $\text{area}(e)$ is the area of an element e in the neighbourhood of node i. The automatic differentiaton setting allows to easily obtain the gradients with respect to the trainable parameters, which is used for parameter update by a quasi-Newton optimization algorithm, named by limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) [LN89, DLT+17, NW06]. For a differentiable function $l = f(\boldsymbol{\theta})$, $\boldsymbol{\theta}$ is updated at iteration k by:

$$
\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \mathbf{H}_k \nabla f_k, \tag{6.12}
$$

where α_k is the step size chosen by a line search [Wol69, Wol71], $\mathbf{H}_k$ is the approximated inverse Hessian, and ∇f_k is the gradient with repect to $\boldsymbol{\theta}$. Instead of inversing the Hessian matrix explicitly, which is too expensive when $\boldsymbol{\theta}$ is high-dimensional, the L-BFGS algorithm approximates $\mathbf{H}_k$ through a much less memory-consuming approach. To do the approximation, the algorithm stores the last m updates of three iterative quantities:

$$\begin{aligned}\mathbf{s}_k &= \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k,\\ \mathbf{y}_k &= \nabla f_{k+1} - \nabla f_k,\\ \rho_k &= 1/(\mathbf{y}_k^T \mathbf{s}_k),\end{aligned}$$

after what these saved vectors are used to estimate $\mathbf{H}_k \nabla f_k$ through the following process:

Algorithm 1: L-BFGS

$\mathbf{q} = \nabla f_k$
for $i = k-1$ *to* $k-m$ **do**
    $\alpha_i \leftarrow \rho_i \mathbf{s}_i^T \mathbf{q}$
    $\mathbf{q} \leftarrow \mathbf{q} - \alpha_i \mathbf{y}_i$
endfor
initialize $\mathbf{H}_k^0 = \frac{\mathbf{s}_{k-1}^T \mathbf{y}_{k-1}}{\mathbf{y}_{k-1}^T \mathbf{y}_{k-1}} \mathbf{I}$
$\mathbf{r} = \mathbf{H}_k^0 \mathbf{q}$
for $i = k-m$ *to* $k-1$ **do**
    $\beta \leftarrow \rho_i \mathbf{y}_i^T \mathbf{r}$
    $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{s}_i(\alpha_i - \beta)$
endfor
return $\mathbf{H}_k \nabla f_k = \mathbf{r}$

As a true second-order optimization algorithm, L-BFGS has better convergence performance than the ADAM optimizer used in previous chapters. It is an ideal option for the PI-GCNN solver, as the unsupervised learning framework does not fit a huge data set, implying that enough memory space is available for the implementation of automatic differentiation. Conversely, L-BFGS remains unrealistic for supervised learning with huge data set and complex network architectures, for which first-order algorithms are the proper choice.

6.7.3 The performance of PI-GCNNs

Following the workflow in figure 6.15, the physical loss is calculated by the non-dimensionalized flow fields, and is minimized by the same GCNN architecture of section 6.3, with a total of $217,341$ parameters. The weights are initalized by the Glorot normal method, and the optimization is stopped if $||\nabla f_{k+1} - \nabla f_k||_{l_2} < 10^{-9}$ at iteration k, with a maximum of $300,000$ iterations.

On a NVIDIA Tesla V100 GPU card, the training takes 3.7 hours, leading to a final loss equal to 6.3×10^{-3}. As is shown in the training curve of figure 6.16, the loss decreases fast during the first $50,000$ iterations, then reach a relatively flat level, although the loss value is continuously reduced. The CFD results are used to compute a referential loss value, which is equal to 7.48×10^{-3}, thus a bit larger than that of the GCNN's outputs. In order to reach the same loss level as the CFD method, the GCNN should be trained for about 1.58 hours. Still, the training time is much longer than the resolution time of the CFD solver, which takes only 5 minutes on a single Intel Xeon 2.6 GHz core (see section 3.2.3 for more details about the CFD resolution).

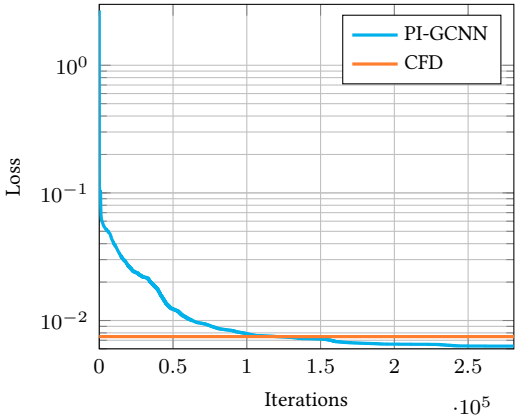
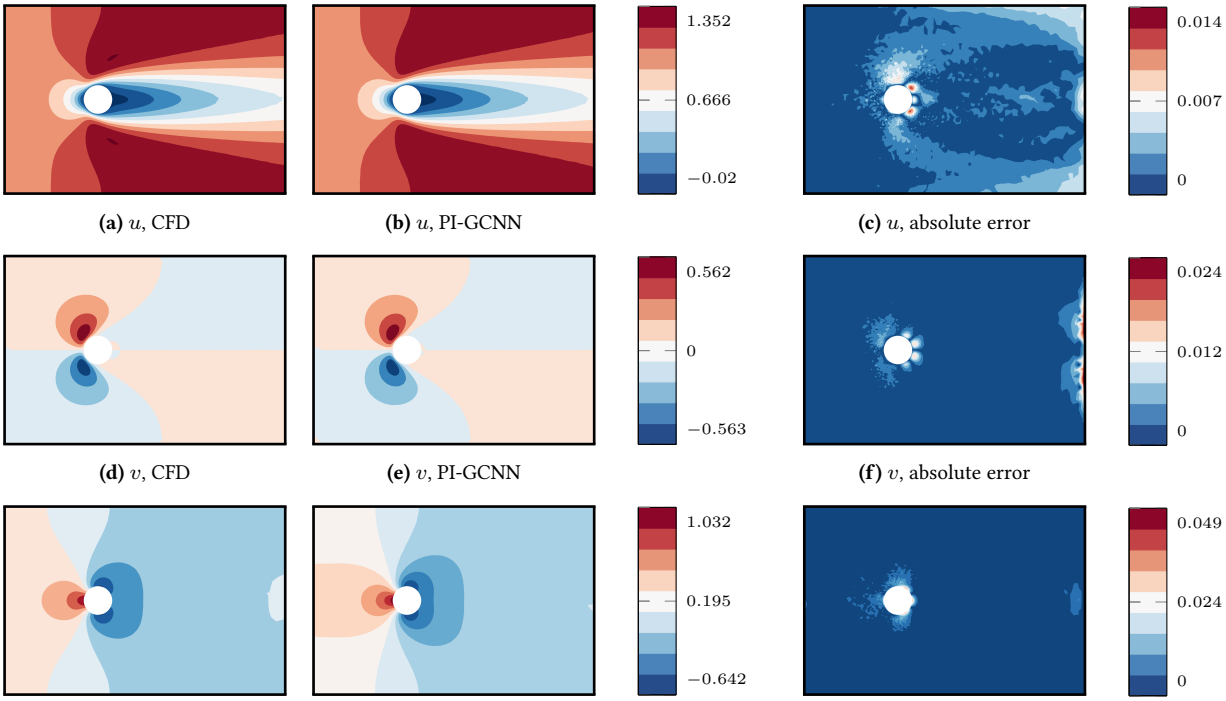


Figure 6.16 | Training history of the pysics-informed GCNN. It takes about 1.58 hours to reduce the physical loss to the same level of CFD results.

The predicted velocity and pressure fields are compared to the CFD results in figure 6.17. As is observed, the prediction is of very high quality in terms of continuity and absolute error, making this work a proof-of-concept of using GCNNs as an unsupervised ML solver for steady incompressible NS equations in the laminar regime. The application to more complex problems, like turbulent flow with more important multiscale property, remains to be validated in the future. On the other hand, the physics-driven regularized method combined with supervised learning is also promising, as it owns the hybrid advantages of fast prediction and physical consistency.



(a) u, CFD **(b)** u, PI-GCNN **(c)** u, absolute error

(d) v, CFD **(e)** v, PI-GCNN **(f)** v, absolute error

(g) p, CFD **(h)** p, PI-GCNN **(i)** p, absolute error

Figure 6.17 | Velocity and pressure resolved by a PI-GCNN. In the left column (6.17a, 6.17d, 6.17g) are resolved by a CFD solver. In the center column, (6.17b, 6.17e, 6.17h) are resolved by the PI-GCNN. From the error maps in the right column, one can confirm that the PI-GCNN's results are on par with those of the traditional CFD solver.

6.8 Conclusion for GCNNs

In this chapter, graph convolutional neural networks for 2-D incompressible laminar flow prediction were introduced. Compared to traditional convolution neural networks, the graph based method works directly on body-fitted triangular meshes, making it a method of choice for coupling with finite element numerical simulations. The proposed convolution method preserves permutation invariance, and is therefore insensitive to node and edge indexing. In the framework of supervised learning, the GCNN network was trained on a dataset of $2,000$ laminar flows around random 2-D shapes, generated using Bézier curves, resulting in an averaged MAE on the test set at 7.70×10^{-3}, with no evidence of overfitting. The resulting model was proved to efficiently reconstruct velocity and pressure fields around a cylinder, respecting the symmetry induced by the input geometry. In particular, the predicted flow remained highly accurate even in the boundary layer of the obstacle. The predicted flow around a significantly different shape, namely a NACA12 airfoil, presented higher MAE levels and ill-estimated amplitudes, while still respecting the physics of the reference flow. The predicted velocity and pressure fields were further used to compute the drag force by integration of the fields in the boundary layer, resulting in an average relative error equal to 3.43% on the test set. Finally, the GCNN model performance was compared to that of a standard U-net model. The GCNN architecture proved its superiority in terms of accuracy and model complexity, although its required training time was significantly longer. As a challenge, the aspect of optimization deserves more studies to improve the training convergence and computational cost. Yet, the versatility of GCNN and its capabilities to work on body-fitted meshes with arbitrarily complex geometric characteristics make it a method of choice for the generation of accurate, fast prediction models coupled to computational fluid dynamics.

In the framework of unsupervised learning, a physics-informed GCNN is used to obtain the steady flow fields around a cylinder, through minimizing the residuals of the NS equations discretized on a triangular mesh. The obtained velocity and pressure solutions are on par with the results from the traditional CFD solver, proving that the latter can be represented by a single forward calculation of a GCNN. As the intersection of physics, numerics and machine learning, embeding physical knowledge into GCNNs are likely to produce performant and robust sorrogate models for CFD solvers.

7 Outlook

In this chapter, we go over the content of the present manuscript, and point out the future possible works toward more CNN-based surrogate model for steady flow estimation with better performance and robustness.

7.1 Summary

The work in this manuscript begins from a general introduction of fluid dynamics, computational fluid dynamics, machine learning and the emerging domain of applying ML methods to traditional CFD. To provide a solid background to the novelties of the manuscript, the second chapter is dedicated to the basic concepts of machine learning and convolutional neural networks, from the view of both modeling and optimization methods. After that, the third chapter employs a traditional solver for Navier-Stokes equations, to generate two data sets of laminar and turbulent flow around random 2-D obstacles.

The obtained data sets are first used to train U-nets for fast flow prediction. Through numerical experiments, U-nets prove to be architectures with sufficient modelling capacity to fit not only the laminar data set, but also the turbulent one. The obtained surrogate models can predict accurately the velocity and pressure fields around 2-D obstacles within 0.05 seconds, representing a considerable speed up compared to tradition PDE solvers. However, large prediction errors are usually observed in the boundary layer, as the U-net method requires interpolating the original data set on cartesian grids, on which it also has to reconstruct the solid/fluid boundary. The staircasing effect, together with the possibly fine grid size, are thus barriers to improvements of the method accuracy.

In the following chapter, a novel auto-encoder architecture with a twin-decoder structure is proposed, in an attempt to improve the reliability of neural networks prediction regarding to interpolation or extrapolation. Such a network architecture allows to evaluate the prediction accuracy through a metric based on the input reconstruction. With the surrogate model obtained in this framework, the end user can either accept or reject the predicted flow fields based on criteria established on the original dataset, thus safeguarding the user from using the model outside of its reliability range, or be provided with a well-calibrated uncertainty level. Such a functionality adds additional transparency to the usual black-box approach, usually reproached to vanilla CNN models.

In the next chapter, a graph convolutional neural network architecture is proposed for laminar flow prediction around random 2-D obstacles. Unlike the U-net method, GCNNs can work on unstructured data, so the flow fields are interpolated on body-fitted triangular meshes, with the boundary element size being the same as the grid size used for U-nets. The GCNNs are trained to predict velocity and pressure values on the nodes, and display higher accuracy than the U-net method, with exceptional performance within the boundary layer prediction. Furthermore, thanks to the parcimonious representation of fluid quantities on triangular meshes, the data set used to train GCNNs is far more compact than the data set for U-nets. The proposed graph-based approach is a meaningful step in order to bridge the gap existing in the efficient coupling of finite element methods and machine learning approaches.

Finally, a preliminary effort is made on the physical consistency of GCNN-based models. A physics-informed GCNN is exploited as a direct solver to the steady Navier-Stokes equations, with the laminar velocity and pressure fields being the outputs of the network, which is trained through minimizing the residuals of the physical and boundary constraints. The fields obtained with the PI-GCNN are very close to those of traditional finite element solvers, meaning that the traditional solver's implicit resolution process can be represented by a forward calculation of a graph convolutional neural network. In the future, this approach deserves more consideration as a new modeling methodology, possibly leading to a new class of direct solvers for PDE problems.

7.2 Future works

The topics presented in this manuscript give rise to a number of possible further developments, lying at the intersection of computational fluid dynamics, convolutional neural networks and optimization. We close this manuscript with a short discussion of these topics.

7.2.1 Outlier detection for GCNN-based surrogate models

Unlike physical models including the Navier-Stokes equations, data-driven surrogate models usually display poor performance on inputs which lie outside of the boundary of their training data set. In the present manuscript, a methodology based on auto-encoders was proposed, and proved to be efficient in the context of CNNs. An extension of this work to graph convolutional neural networks could represent an interesting research topic.

7.2.2 Optimization aspects of GCNNs

GCNNs are the ideal modeling methods for computational fluid dynamics, as they prove to reconstruct accurately the boundary layer in external flow around 2-D obstacles. With a minimum of network complexity, GCNNs attain better performance than traditional CNNs, making them promising and parsimonious surrogate models. However, the training costs of both the supervised and the unsupervised models are higher than that of traditional CNNs. A better understanding of the training process (*i.e.* the gradient back-propagation with GCNNs) is required to reduce the training cost to a level comparable to CNNs. For example, a diverse pool of activation functions besides the swish function could be tested, and there may exist better options as the convolutional kernels rather than using multilayer perceptrons. Alternatively, since the graph convolution method used in this manuscript is based on the neural message passing framework, other convolution frameworks also deserve trials and could result in different training characteristics.

7.2.3 GCNNs for turbulence

With the success on laminar flow prediction, it would be interesting to train a GCNN as the surrogate model for RANS-resolved turbulent flow. Similarly to the U-net approach, one can expect difficulties to fit the turbulent data set with GCNNs. However, with the grid size being 0.01m in the U-net approach, the boundary element size can be reduced to 5×10^{-5}m thanks to the parsimonious discretization of triangular meshes. Consequently, graph-based surrogate models are more suitable for turbulent flow prediction in spite of the optimization difficulties, which are not yet clear considering the multiscale properties of turbulent flow.

7.2.4 Physics-driven regularization for GCNNs

Finally, the physics-driven regularization approach for supervised learning would be an improvement to the surrogate models presented in this manuscript. Unlike the iterative solver based on physics-informed unsupervised learning, physics-informed supervised learning allows the training of a surrogate model which predicts the flow fields in one single forward calculation, and respects the physical constraints, making this approach hold hybrid advantages in efficiency as well as physical consistency. By combining the physics-constrained approach with graph convolutional neural networks, we expect surrogate models which respects both geometry and physics, thus providing more realistic results, similar in quality to that of finite element solvers.

Publications

Research articles

- *A twin-decoder structure for incompressible laminar flow reconstruction with uncertainty estimation around 2D obstacles*; Junfeng Chen, Jonathan Viquerat, Frederic Heymes and Elie Hachem, **Neural Comput & Applic** (2022). https://doi.org/10.1007/s00521-021-06784-z
- *Graph neural networks for laminar flow prediction around random two-dimensional shapes*; Junfeng Chen, Elie Hachem and Jonathan Viquerat, **Physics of Fluids** 33, 123 607 (2021) https://doi.org/10.1063/5.0064108

Oral presentations

- *U-shape convolutional neural networks for fast flow prediction at low Reynolds number*; Complex Day, Nice, France (March 2019)
- *Graph convolutional neural networks for flow prediction around random 2-D shapes*; MMLDT-CSET, San Diego, USA (September 2021)

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[AJ12] Steven R Allmaras and Forrester T Johnson. Modifications and clarifications for the implementation of the spalart-allmaras turbulence model. In *Seventh international conference on computational fluid dynamics (ICCFD7)*, volume 1902. Big Island, HI, 2012.

[Aze19] Chloé-Agathe Azencott. *Introduction au machine learning*. Dunod, 2019.

[Bat00] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library. Cambridge University Press, 2000.

[BBV04] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[BDC09] Julien Bruchon, Hugues Digonnet, and Thierry Coupez. Using a signed distance function for the simulation of metal forming processes: Formulation of the contact condition and mesh adaptation. from a lagrangian approach to an eulerian approach. *International journal for numerical methods in engineering*, 78(8):980–1008, 2009.

[Ben09] Yoshua Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009.

[BHB+18] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv*, 2018.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[Bou77] J. Boussinesq. Essai sur la théorie des eaux courantes. *Comptes rendus de l'Académie des Sciences*, 23:1–680, 1877.

[BPL+16] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray kavukcuoglu. Interaction networks for learning about objects, relations and physics. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 4509–4517, Red Hook, NY, USA, 2016. Curran Associates Inc.

[BPRS18] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[BR78] I. Babuvška and W. C. Rheinboldt. Error estimates for adaptive finite element computations. *SIAM Journal on Numerical Analysis*, 15(4):736–754, 1978.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[BWAN18] C. Baur, B. Wiestler, S. Albarqouni, and N. Navab. Deep autoencoding models for unsupervised anomaly segmentation in brain mr images. In *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries*, page 161–16. Springer, 2018.

[C+15] Francois Chollet et al. Keras, 2015.

[CHM09] Carlos D Correa, Robert Hero, and Kwan-Liu Ma. A comparison of gradient estimation methods for volume rendering on unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):305–319, 2009.

[Cou43] R. Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bulletin of the American Mathematical Society*, 49(1):1 – 23, 1943.

[CSW+20] J.K. Chow, Z. Su, J. Wu, P.S. Tan, X. Mao, and Y.H. Wang. Anomaly detection of defects on concrete structures with the convolutional autoencoder. *Advanced Engineering Informatics*, 45:101105, 2020.

[CVH19] J. Chen, J. Viquerat, and E. Hachem. U-net architectures for fast prediction of incompressible laminar flows, 2019.

[CWF+21] Shengze Cai, Zhicheng Wang, Frederik Fuest, Young Jin Jeon, Callum Gray, and George Em Karniadakis. Flow over an espresso cup: inferring 3-d velocity and pressure fields from tomographic background oriented schlieren via physics-informed neural networks. *Journal of Fluid Mechanics*, 915, 2021.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[DABPEK20] Filipe De Avila Belbute-Peres, Thomas Economon, and Zico Kolter. Combining differentiable PDE solvers and graph neural networks for fluid flow prediction. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 2402–2411. PMLR, 13–18 Jul 2020.

[DBV16] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29:3844–3852, 2016.

[Dea70] James W. Deardorff. A numerical study of three-dimensional turbulent channel flow at large reynolds numbers. *Journal of Fluid Mechanics*, 41(2):453–480, 1970.

[DIX19] Karthik Duraisamy, Gianluca Iaccarino, and Heng Xiao. Turbulence modeling in the age of data. *Annual Review of Fluid Mechanics*, 51:357–377, 2019.

[DLT+17] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*, 2017.

[EF11] I. Eames and J. B. Flor. New developments in understanding interfacial processes in turbulent flows. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 369(1937):702–705, 2011.

[FFT19] Kai Fukami, Koji Fukagata, and Kunihiko Taira. Super-resolution reconstruction of turbulent flows with machine learning. *Journal of Fluid Mechanics*, 870:106–120, 2019.

[Fuk88] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[GLI16] Xiaoxiao Guo, Wei Li, and Francesco Iorio. Convolutional neural networks for steady flow approximation. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 481–490, 2016.

[GSR+17] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[GSW21] Han Gao, Luning Sun, and Jian-Xun Wang. Phygeonet: Physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state pdes on irregular domain. *Journal of Computational Physics*, 428:110079, 2021.

[GT21] Cedric Fraces Gasmi and Hamdi Tchelepi. Physics informed deep learning for flow and transport in porous media, 2021.

[GW20] Zhi Geng and Yanfei Wang. Physics-guided deep learning for predicting geological drilling risk of wellbore instability using seismic attributes data. *Engineering Geology*, 279:105857, 2020.

[HFCC13] Elie Hachem, Stephanie Feghali, Ramon Codina, and Thierry Coupez. Immersed stress method for fluid–structure interaction using anisotropic mesh adaptation. *International Journal for Numerical Methods in Engineering*, 94(9):805–825, 2013.

[HHWB02] S. Hawkins, H. He, G. Williams, and R. Baxter. Outlier detection using replicator neural networks. In *Data Warehousing and Knowledge Discovery*, pages 170–180. Springer Berlin Heidelberg, 2002.

[Hot33] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[HSS12] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8):2, 2012.

[HYL17] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[IDC21] Arsen S. Iskhakov, Nam T. Dinh, and Edward Chen. Integration of neural networks with numerical solution of pdes for closure models development. *Physics Letters A*, 406:127456, 2021.

[JCCL18] X. Jin, P. Cheng, W. L. Chen, and H. Li. Prediction model of velocity field around circular cylinder over various reynolds numbers by fusion convolutional neural networks based on pressure on the cylinder. *Physics of Fluids*, 30(4), 2018.

[JHVC15] G. Jannoun, E. Hachem, J. Veysset, and T. Coupez. Anisotropic meshing with time-stepping control for unsteady convection-dominated problems. *Applied Mathematical Modelling*, 39(7):1899–1916, 2015.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[KH92] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

[KKL+21] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

[KLH17] M. Ke, C. Lin, and Q. Huang. Anomaly detection of logo images in the mobile phone using convolutional autoencoder. In *4th International Conference on Systems and Informatics (ICSAI)*, pages 1163–1168, 2017.

[Kol41] Andrey Nikolaevich Kolmogorov. The local structure of turbulence in incompressible viscous fluid for very large reynolds numbers. *Cr Acad. Sci. URSS*, 30:301–305, 1941.

[Kra91] Mark A Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243, 1991.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[LB+95] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[LKT16] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807:155–166, 2016.

[LN89] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[LY19] Sangseung Lee and Donghyun You. Data-driven prediction of unsteady flow over a circular cylinder using deep learning. *Journal of Fluid Mechanics*, 879:217–254, 2019.

[LZCS14] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[MBM+17] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5115–5124, 2017.

[MMOH20] Craig Michoski, Miloš Milosavljević, Todd Oliver, and David R Hatch. Solving differential equations using deep neural networks. *Neurocomputing*, 399:193–212, 2020.

[MN19] Peter McCullagh and John A Nelder. *Generalized linear models*. Routledge, 2019.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[Nav23] Claude Louis Marie Henri Navier. Mémoire sur les lois du mouvement des fluides. *Mémoires de l'Académie des sciences de l'Institut de France*, pages 389–440, 1823.

[Ng04] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78, 2004.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[NM20] Mohammad Amin Nabian and Hadi Meidani. Physics-driven regularization of deep neural networks for enhanced engineering design and analysis. *Journal of Computing and Information Science in Engineering*, 20(1):011006, 2020.

[NW06] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[NWK20] Michael Weinmann Nils Wandel and Reinhard Klein. Unsupervised deep learning of incompressible fluid dynamics. *CoRR*, abs/2006.08762, 2020.

[OHZU20] Takato Otsuzuki, Hideaki Hayashi, Yuchen Zheng, and Seiichi Uchida. Regularized pooling. In *Artificial Neural Networks and Machine Learning – ICANN 2020*, pages 241–254, Cham, 2020. Springer International Publishing.

[OMHF20] Francis Ogoke, Kazem Meidani, Amirreza Hashemi, and Amir Barati Farimani. Graph convolutional neural networks for body force prediction, 2020.

[PD16] Eric J Parish and Karthik Duraisamy. A paradigm for data-driven predictive modeling using field inversion and machine learning. *Journal of Computational Physics*, 305:758–774, 2016.

[PFSGB21] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 2021.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch, 2017.

[Pir89] Olivier Pironneau. *Finite element methods for fluids*. Wiley Chichester, 1989.

[PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1310–III–1318. JMLR.org, 2013.

[Poi40] Jean-Léonard-Marie Poiseuille. Recherches expérimentales sur le mouvement des liquides dans les tubes de très petits diamètres. *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, pages 961–967, 1840.

[Pop00] Stephen B. Pope. *Turbulent Flows*, page 3–9. Cambridge University Press, 2000.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[Rey83] Osborne Reynolds. XXIX. An experimental investigation of the circumstances which determine whether the motion of water shall be direct or sinuous, and of the law of resistance in parallel channels. *Philosophical Transactions of the Royal Society of London*, 174:935–982, 1883.

[Rey95] 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.)*, 186:123–164, 1895.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[RL07] Lewis Fry Richardson and Peter Lynch. *Weather Prediction by Numerical Process*. Cambridge Mathematical Library. Cambridge University Press, 2 edition, 2007.

[RPK19] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[RSL20] Chengping Rao, Hao Sun, and Yang Liu. Physics-informed deep learning for incompressible laminar flows. *Theoretical and Applied Mechanics Letters*, 10(3):207–212, 2020.

[RZL18] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2018.

[S+51] George Gabriel Stokes et al. *On the effect of the internal friction of fluids on the motion of pendulums*, page 1–10. Pitt Press Cambridge, 1851.

[SA94] P. SPALART and S. ALLMARAS. A one-equation turbulence model for aerodynamic flows. *Recherche Aerospatiale*, 1:5–21, 1994.

[Sch07] François G. Schmitt. About boussinesq's turbulent viscosity hypothesis: historical remarks and a direct evaluation of its validity. *Comptes Rendus Mécanique*, 335(9):617–627, 2007. Joseph Boussinesq, a Scientist of bygone days and present times.

[SFG+18] Rishi Sharma, Amir Barati Farimani, Joe Gomes, Peter Eastman, and Vijay Pande. Weakly-supervised deep learning of heat transport via physics informed loss. *arXiv preprint arXiv:1807.11374*, 2018.

[SGGP+20] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. Learning to simulate complex physics with graph networks, 2020.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[Sho12] Naum Zuselevich Shor. *Minimization methods for non-differentiable functions*, volume 3. Springer Science & Business Media, 2012.

[SKA+14] Jeffrey Slotnick, Abdollah Khodadoust, Juan Alonso, David Darmofal, William Gropp, Elizabeth Lurie, and Dimitri Mavriplis. Cfd vision 2030 study: a path to revolutionary computational aerosciences, 2014.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[Sma63] J. Smagorinsky. General Circulation Experiments with the Primitive Equations. *Monthly Weather Review*, 91(3):99, January 1963.

[Stu88] Roland B. Stull. *An Introduction to Boundary Layer Meteorology*. Atmospheric Sciences Library. Springer, 1988.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[TWPH20] Nils Thuerey, Konstantin Weißenow, Lukas Prantl, and Xiangyu Hu. Deep learning methods for reynolds-averaged navier–stokes simulations of airfoil flows. *AIAA Journal*, 58(1):25–36, 2020.

[VH20] Jonathan Viquerat and Elie Hachem. A supervised neural network for drag prediction of arbitrary 2d shapes in laminar flows at low reynolds number. *Computers & Fluids*, 210:104645, 2020.

[WAH93] John J Weng, Narendra Ahuja, and Thomas S Huang. Learning recognition and segmentation of 3-d objects from 2-d images. In *1993 (4th) International Conference on Computer Vision*, pages 121–128. IEEE, 1993.

[Whi11] Frank M. White. *Fluid Mechanics*. McGraw-Hill Companies, Inc., 2011.

[Wol69] Philip Wolfe. Convergence conditions for ascent methods. *SIAM review*, 11(2):226–235, 1969.

[Wol71] Philip Wolfe. Convergence conditions for ascent methods. ii: Some corrections. *SIAM review*, 13(2):185–188, 1971.

[WWK21] Nils Wandel, Michael Weinmann, and Reinhard Klein. Teaching the incompressible navier–stokes equations to fast neural surrogate models in three dimensions. *Physics of Fluids*, 33(4):047117, 2021.

[WWX17] Jian-Xun Wang, Jin-Long Wu, and Heng Xiao. Physics-informed machine learning approach for reconstructing reynolds stress modeling discrepancies based on dns data. *Physical Review Fluids*, 2(3):034603, 2017.

[XC19] Heng XIAO and Paola CINNELLA. Quantification of model uncertainty in RANS simulations: A review. *Progress in Aerospace Sciences*, 108:1–31, July 2019.

[XSSZ21] Mengfei Xu, Shufang Song, Xuxiang Sun, and Weiwei Zhang. A convolutional strategy on unstructured mesh for the adjoint vector modeling. *Physics of Fluids*, 33(3):036115, 2021.

[ZCH+20] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[ZLS20] Ruiyang Zhang, Yang Liu, and Hao Sun. Physics-guided convolutional neural network (phycnn) for data-driven seismic response modeling. *Engineering Structures*, 215:110704, 2020.

[ZSM18] Yao Zhang, Woong Je Sung, and Dimitri N Mavris. Application of convolutional neural network to predict airfoil lift coefficient. In *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, page 1903, 2018.

[ZTXM19] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):1–23, 2019.

RÉSUMÉ

Au cours des dernières années, les réseaux de neurones ont suscité un grand intérêt dans la communauté de la dynamique des fluides computationnelle, en particulier lorsqu'ils sont utilisés comme modèles subrogés, que ce soit pour la reconstruction de l'écoulement, la modélisation de la turbulence ou pour la prédiction des coefficients aérodynamiques. Cette thèse considère l'utilisation de réseaux de neurones convolutifs, une catégorie spéciale de réseaux de neurones conçus pour les images, comme modèles subrogés pour la prédiction de l'écoulement stationnaire autour d'obstacles 2D. Les modèles subrogés sont calibrés dans le cadre de l'ajustement des données, avec l'ensemble de données préparé par des solveurs qualifiés aux équations de Navier-Stokes et projeté sur des grilles cartésiennes. Une fois calibrés, les modèles montrent une grande précision en termes de prédiction de vitesse et de pression, même autour d'obstacles non vus lors de la calibration. Dans l'étape suivante, une nouvelle architecture de réseaux de neurones convolutifs est proposée pour la détection d'anomalies et la quantification de l'incertitude, permettant au modèle subrogé de savoir s'il effectue une interpolation ou une extrapolation tout en faisant la prédiction. Avec ces méthodes, l'utilisateur d'un réseau de neurones calibré peut soit décider d'accepter ou non une prédiction, soit avoir une estimation quantifiée de l'erreur de prédiction. La troisième contribution consiste à utiliser des réseaux de neurones convolutifs sur graphes comme modèles subrogés pour prédire la vitesse et la pression sur des maillages triangulaires, qui présentent des avantages significatifs dans la représentation géométrique par rapport aux grilles cartésiennes. Grâce au raffinement du maillage proche des interfaces solides, le modèle basé sur des graphes peut donner une prédiction de couche limite plus précise que les réseaux de neurones convolutifs traditionnels. La dernière partie de cette thèse considère l'intégration des connaissances physiques dans la calibration d'un réseau de neurones convolutifs sur graphe, qui est calibré en minimisant le résidu des équations de Navier-Stokes sur un maillage triangulaire. La vitesse et la pression prédites autour d'un cylindre sont de très haute qualité par rapport aux résultats des solveurs numériques qualifiés. N'étant pas dans le cadre de l'ajustement des données, cette approche fournit un nouveau solveur d'équations aux dérivées partielles, et mérite plus de travail sur sa convergence et son coût de calcul.

MOTS CLÉS

réseaux de neurones convolutifs, modèles subrogés, dynamique des fluides numériques, élément fini, quantification de l'incertitude, convolution sur graphes

ABSTRACT

Over the past few years, neural networks have arisen great interest in the computational fluid dynamics community, especially when used as surrogate models, either for flow reconstruction, turbulence modeling, or for the prediction of aerodynamic coefficients. This thesis considers using convolutional neural networks, a special category of neural networks designed for images, as surrogate models for steady flow prediction around 2D obstacles. The surrogate models are calibrated in the framework of data fitting, with the data set prepared by high-fidelity solvers to Navier-Stokes equations and projected onto cartesian grids. Once calibrated, the models show high accuracy in terms of velocity and pressure prediction, even around obstacles not seen during the calibration. In the next step, a new architecture of convolutional neural networks is proposed for anomaly detection and uncertainty quantification along with the steady flow prediction, making the surrogate model aware whether it is doing interpolation or extrapolation while doing prediction. With these methods, the user of a calibrated neural network can either decide whether to accept a prediction or not, or have a quantified estimation of the prediction error. The third contribution is to use graph convolutional neural networks as surrogate models to predict velocity and pressure on triangular meshes, which have significant advantages in geometry representation compared to cartesian grids. Thanks to the mesh refinement close to the solid interfaces, the graph-based model can give more accurate boundary layer prediction than traditional convolutional neural networks. The last part of this thesis considers integrating physical knowledge into the calibration of a graph convolutional neural network, which is calibrated by minimizing the residual of Navier-Stokes equations on a triangular mesh. The predicted velocity and pressure around a cylinder are of very high quality when compared to the results of high-fidelity numerical solvers. Being not in the framework of data fitting, this approach provides a novel solver to partial differential equations, and deserves more work on its convergence and computational cost.

KEYWORDS

convolutional neural networks, surrogate models, computational fluid dynamics, finite element, uncertainty quantification, graph convolution