POLITECNICO DI TORINO

Corso di Laurea Magistrale

in Ingegneria Aerospaziale

Tesi di Laurea Magistrale

Artificial Neural Networks for surrogating high-fidelity simulations of turbulent flows

Relatore/i. Professoressa Sandra Pieraccini
Candidato/i Giacomo Lorenzini

Anno Accademico 2021/2022

INDICE

Abstract 5

1 Theoretical analysis of Fluid Dynamics 7

1.1 Navier-Stokes equations 7

1.2 Computational Fluid Dynamics 9

1.2.1 Building a CFD simulation 10

1.3 Turbulence modeling 10

1.3.1 Turbulence models.....10

1.3.2 The RANS model – Reynolds equation.....12

1.3.3 The k-ε model.....13

1.3.4 The k-ω model.....14

1.3.5 The k-ω *SST* model.....15

2 Machine Learning.....16

2.1 Introduction to Machine Learning.....16

2.2 Artificial Neural Networks.....17

2.2.1 Neural Network Classification.....18

2.2.2 Activation Function.....20

2.2.2.1 Sigmoid Function.....21

2.2.2.2 Hyperbolic Tangent Function.....22

2.2.2.3 Rectified Linear Unit Function.....23

2.2.3 Backpropagation algorithm - Gradient Descent Method.....26

3 CFD Simulation and NN Training.....31

3.1 The Backward-facing step 31

3.1.1 Geometry.....32

3.1.2 Mesh Setup.....32

3.1.3 CFD solver.....33

3.2 The Feed-Forward Neural Network 35
3.2.1 The Feed-Forward Neural Network Structure....35
3.2.1.1 The dataset.....36
3.2.1.2 Hyperparameters.....37
3.3 The Recurrent Neural Network.....40

4 Results.....43
4.1 Feed-Forward Neural Network.....43
4.1.1 Hyperparameters Tuning.....43
4.1.2 Reynolds number and time influence.....47
4.1.2.1 Reynolds number.....47
4.1.2.2 Time.....54
4.2 Recurrent Neural Network.....60

5 Conclusions and possible improvements 71

6 Bibliography.....72

Abstract

In many problems represented by a mathematical model, the knowledge of the solution at different values of specific parameters is necessary and one way to proceed is to compute a high-fidelity simulation; however, since a high-fidelity simulation is very computational demanding, this is often too time consuming. In these situations, surrogate models can be very useful. The aim of this work is to show viability of Machine Learning as a surrogate model in a specific setting.

The objective is to obtain a reconstruction of a high-fidelity solution of a fluid dynamics problem, depending on both a specific Reynolds number and a specific time, without executing the relative simulation, with the aim of accelerating the case where thousands of simulations are required. In fact, since a Machine Learning based approach will require a conspicuous number of simulations for the training, a trade-off between the number of high-fidelity solutions actually needed and those needed for the training has to be taken into account.

The first two chapters offer an introductory description of the major subjects of this thesis project: Computational Fluid Dynamic and Machine Learning.

1 Theoretical analysis of Fluid Dynamics

1.1 Navier-Stokes Equations

Fluid dynamics is a complex field.

The behaviour of fluids is described through complicated mathematical expressions defined as Navier-Stokes equations. The latter are a generalization of the Eulerian system formulated to explain the flow of adiabatic and inviscid fluids. In fact, Claude-Luis Navier's work, followed by George Gabriel Stokes' studies, introduced the concept of viscosity making it necessary to develop a new system of equations to characterize more complex and real flows.

The Navier-Stokes system (N-S) consists of three different equations that correspond to the conservation (and evolution in the flow) of three fluids' characteristics: mass, momentum and energy. The equations link the three spatial coordinates (x, y, and z) and time (t) with six dependent variables that completely define the flow: pressure, density, temperature and the three velocity components (u, v, w). Being the latter all functions of the four independent variables, the system is a set of Partial Differential Equations (PDE).

As said, the base of the theory are three physical principles; N-S equations derive from the application of Newton's Second Law to the motion of a viscous fluid. Each expression corresponds to the conservation of a different quantity that characterizes the fluid. Considering a fixed control volume, they can be defined as:

- The *Continuity Equation* (or mass balance equation), which states that the mass cannot be generated or destroyed. The written equation relates the change of the mass inside the control volume to the flux of mass entering (or exiting) the volume through its surfaces.
- The *Momentum Balance Equations* (developed in all three spatial coordinates x, y, z) relates the sum of external forces (surface and body forces) acting on an element of fluid to the rate of change of its momentum.
- The *Energy Equation* (funded on the first law of thermodynamics), which states that the energy is conserved. The rate of energy transfer into and out of the fixed volume must equal the rate of change of energy within its boundaries.

The result is a coupled system of the following equations [1]:

$$\begin{cases} \dfrac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) = 0 \\ \dfrac{\partial (\rho \vec{u})}{\partial t} + \nabla \cdot (\rho \vec{u} \vec{u}) = -\nabla p + \nabla \cdot \vec{\tau} + \rho \vec{f} \\ \dfrac{\partial E}{\partial t} + \nabla \cdot (E \vec{u}) = -\nabla \cdot (p \vec{u}) - \nabla \cdot q + \rho \vec{f} \cdot \vec{u} + \nabla \cdot (\vec{u} \cdot \vec{\tau}) \end{cases}$$

Where:

- $\vec{u} = (u, v, w)$
- ρ is the density
- E is the energy, equal to the sum of the internal energy and the kinetic energy of the fluid and it's calculated with the following formula:

$$E = e + \frac{1}{2}\vec{u}^2$$

- q corresponds to the heat transfer and is equal to $q = -k\nabla T$ (Fourier's Law)
- $\vec{f}$ is a generic force field
- $\vec{\tau}$ coincides with the stress tensor and is defined by a matrix of components as follows:

$$\vec{\tau} = \begin{bmatrix} \tau_{11} & \tau_{12} & \tau_{13} \\ \tau_{21} & \tau_{22} & \tau_{23} \\ \tau_{31} & \tau_{32} & \tau_{33} \end{bmatrix}$$

Of which every element is defined by the formula:

$$\tau_{ij} = \delta_{ij} \lambda \nabla \cdot \vec{u} + \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right)$$

The index i indicates the direction in which the stress component acts, the index j specifies the orientation of the surface upon which is acting. δ_{ij} is the Kronecker's Delta, unitary if $i = j$, otherwise equal to zero.

- μ is the dynamic viscosity of the fluid and it is related to the bulk viscosity of the fluid with the equation:

$$\lambda = -\frac{2}{3}\mu$$

Where λ depends on the temperature and pressure of the flow (and is negligible for incompressible flows).

Furthermore, the existence of the viscosity enables to present another fundamental parameter, the Reynolds number. The latter is defined as the ratio of the inertial and viscous forces:

$$\mathrm{Re} = \frac{\rho v L}{\mu}$$

This dimensionless term quantifies the relative importance of the two types of forces, consequently representing the ratio of the contributes of convection and diffusion. For example, high Reynolds numbers (which correspond to turbulent flows) mean that inertial terms affect the flow behavior more than the viscosity – whose effect is to inhibit turbulence – returning a flow field where convection is the predominant transport phenomenon. In the previous formula, v is the velocity of the fluid, while L is the characteristic linear dimension of the considered motion (for example, in the case of a flow around an airfoil, L corresponds to the chord of the airfoil's profile).

All dependent variables appear in the formulas, except for one. In fact, the temperature is hidden in the equation of state that must be used to close the system:

$$pV = nRT$$

Where n is the number of moles and R is the universal constant ($R = 8{,}325\ \frac{J}{mol\ K}$).

The objective of this thesis work involves fully turbulent flows, meaning that only high Reynolds number will be considered. However, the inclusion of turbulence modelling and the third-dimension z have proven to be intractable unless multiple simplifications are applied, making the numerical solution only an approximation of the real result. Moreover, being the solution impossible to find through handmade calculation, the help of computers is necessary; therefore, CFD has become the most important tool in fluid mechanics.

1.2 Computational Fluid Dynamic

Computational Fluid Dynamic (CFD) corresponds to the simulation of fluids using modeling and numerical methods. It is a very wide subject, as it involves branches of both physics and mathematics; given a fluid problem, fluid mechanics is required to know its physical properties and solve it. As everything is defined, Navier-Stokes equations describe the problem to its completion. Since the work must be relegated to a computer, everything needs to be translated into the discretized form. The translators are numerical discretization methods, such as Finite Difference and Finite Volume methods.

CFD's use is to obtain accurate results of physical phenomena more cost-efficiently with respect to experimental testing, as once the pre-processing phase is finalized, the simulations can be run multiple times, reducing the operating costs that characterize the experimental phase (i.e., in a wind tunnel). In addition, being computer simulations easily repeatable, different solutions can be obtained by changing input parameters, reducing both the time and the cost of the overall project. Moreover, it's important to remember that CFD enables to define data at every point of the domain; this is completely different from an experimental test, where the solution is given by the sensors (for example pressure ports and probes) located around the tested object; of course, a higher number of instruments gives more detailed information, but this corresponds to a more expensive material cost. Finally, computational fluid dynamic facilitates flow simulation around large-scale models that would be otherwise

impossible to analyze; in fact, thanks to the possibility of scaling of objects, real dimensions are not binding.

1.2.1 Building a CFD Simulation

Once the fluid dynamic problem is defined, the construction of a CFD analysis can be divided into three major steps.

1. **Preprocessing**, which corresponds to the formation step whose main objective is to correctly define the input parameters and prepare the geometry for the simulation. First, it's necessary to understand the physics of the problem; once this is clarified, the geometry must be determined to properly represent it. The latter part is perhaps the most important, as the domain must be subdivided into multiple elements to compute the governing equations. This mechanism is called *meshing* and coincides with creating the computational grid; its refinement is a key contributor to the final quality of the solution as constructing a finer mesh will mean having more detailed results. The downside of a more accurate solution is a longer simulation, as the refinement affects the computational cost.

2. **Processing**, which is the solving section. Once the geometry is ready, a CFD software computes the governing equations in each cell, by iteratively solving them calculating the residual (which can be considered the 'error' of the CFD simulation). As the *convergence criterion*, defined a priori by the user, is satisfied, the computation stops and the results are ready to be shown.

3. **Postprocessing**, which aims to correctly derive the right conclusions from the model. In this final step the solution is shown by producing images of the evolution of each field of the flow (for example velocity or pressure field), streamlines and vector (or data) plots. The objective is to create effective visualizations of the flow behaviour.

1.3 Turbulence modeling

1.3.1 Turbulence models

The processing step is conducted through iterative calculations to solve the discrete form of Navier-Stokes equations. If the complexity of the problem increases, so does the number of repetitive steps to obtain its solution. This number corresponds to the computational cost of the CFD simulation, meaning the CPU memory needed to perform it. Different factors influence the computational effort; in this work, only the refinement of the mesh is considered: the finer the grid, the higher the memory cost.

The following is a short illustration of the four different computational methods that have been developed in years of study [2] [3].

a. *Direct Numerical Simulation* (*DNS*)

This method solves the set of governing equation for every cell of the mesh without applying any simplification through turbulent models. This means that turbulence is solved in the whole range of both spatial and temporal scales, requiring a large amount of CPU memory. Although this method returns a very accurate solution of the studied phenomenon, it is inapplicable due to the computational cost necessary to determine it. Therefore, advanced (and simplified) models exist.

b. *Reynolds Averaged Navies-Stokes (RANS)*

This approach, developed by Osborne Reynolds, can be used to describe turbulent flows. RANS equations derive from the instantaneous Navier-Stokes equations through the Reynolds decomposition, which refers to separation of the flow variable into two different terms, the mean time averaged component and the fluctuating component. For example, considering the velocity:

$$u(x,t) = \bar{u}(x) + u'(x,t)$$

The simplification leads to a nonlinear stress term (the Reynolds stress tensor, which corresponds to $-\rho\overline{u'_\iota u'_J}$) that must be modelled to obtain the flow solution; this means another equation must be added to close the system.

c. *Large Eddies Simulation* (*LES*)

This model works following the implication of Kolmogorov's theory of self-similarity, which allows to explicitly solve for large eddies and implicitly account for the smaller eddies by using a sub-grid scale. In fact, LES's main idea is to reduce the range of length scale considered by the model via a low-pass filter directly applied to Navier-Stokes equations, whose objective is to remove small-scale information from the numerical solution. The output is an approximate solution characterized by a lower computational cost, as usually the memory usage is mainly due to the solving of the removed smaller scales. However, the filtered data must be acknowledged in those problems where small scales play an important role, as their effect becomes relevant; therefore, they must be resolved through sub-grid models such as Smagorinsky's model or turbulent kinetic energy model. The advantages of LES modelling correspond to a bigger spectrum of applicability and, more importantly, to a moore accurate solutions; the downside is a higher computational cost due to the time step requirements, as the flow can't be considered steady and a finer mesh is necessary in order to capture more details of the flow.

d. *Detached Eddy Simulation* (*DES*)

Finally, there is a model that goes under the name DES, which exploits the benefits of LES modelling while retaining the efficiency of the RANS. The refinement of the mesh is only in proximity of surfaces of interest (as in LES the flow must be completely solved near the wall), largely reducing the total computational cost. The DES approach

offers more insight in the solution and the finer spatial resolution allows to study detailed behaviour of the flow of interest. All of it at a reduced cost compared to a fully-fledged LES approach.

Since the final objective of every developed model is to reduce the computational cost while maintaining a good efficiency in the solution accuracy, different methods could be applied to keep the memory usage low.

This project implements one of these models, combining efficient CFD together with Machine Learning to speed up the process of researching multiple solutions. In fact, the aim is to train an AI to deduce and predict the flow field, to avoid running the solution multiple times, hence lowering the computational cost of the total experiment. However, the training of the AI requires thousands of data, which can only be obtained by running the simulation hundreds of times. This means this idea is applicable only to an experiment that requires a big multitude of solutions of the same problem, which may for example ask to compare different simulation results of the same flow field. Ultimately, using this method is possible only if the solution given by the machine can be considered close to the real result of the simulation. This means that the accuracy of the AI must be very close to perfect to satisfy the user's requirements.

1.3.2 The RANS model – Reynolds equation

The turbulent model used in this thesis project is the RANS; in this approach to turbulence, all the unsteadiness in the flow is averaged out [4]. As said, the flow variables are decomposed in two different terms; the following example considers a single component of the velocity:

$$u_i(x_i, t) = \bar{u}_i(x_i) + u_i'(x_i, t)$$

Where

$$\bar{u}_i(x_i) = \lim_{T\to\infty} \frac{1}{T} \int_0^T u_i(x_i, t)$$

Applying the mean to both the incompressible continuity equations gives the following result:

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0$$

However, considering the same operation for the momentum equation, the result after simplifications is:

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

Where the term in the square brackets corresponds to the sum of three different stresses: the first two depend on the pressure field and the viscosity, while the last term $\rho\overline{u_i' u_j'}$ derives from

the fluctuating velocity field and corresponds to the *Reynolds stress*. The latter is correlated to the turbulent kinetic energy k by the following function.

$$k = \frac{1}{2}\rho\overline{u_i'u_j'}$$

In order to close the system of equations, Reynolds stresses must be approximated. Boussinesq introduced his hypothesis in 1877, by which he linked the Reynolds stress to the turbulent viscosity ν_T:

$$-\overline{u_i'u_j'} = \nu_T\left(\frac{\partial\bar{u}_i}{\partial x_j} + \frac{\partial\bar{u}_j}{\partial x_i}\right) - \frac{2}{3}k\delta_{ij}$$

This means that if a value for ν_T is defined, the system can be solved.

1.3.3 The k-ε model

Before introducing the equation for turbulent kinetic energy, it's important to show ε, the rate of dissipation of turbulent energy [5]:

$$\varepsilon = \nu\overline{\frac{\partial u_i'}{\partial x_j}\frac{\partial u_i'}{\partial x_j}}$$

It follows the equation for the turbulent kinetic energy:

$$\frac{\partial k}{\partial t} + \bar{u}_j\frac{\partial k}{\partial x_j} = -\frac{\partial}{\partial x_j}\left(\frac{1}{2}\overline{u_i'u_j'u_j'} + \frac{1}{\rho}\overline{u_i'p'} - \nu\frac{\partial k}{\partial x_j}\right) - \overline{u_i'u_j'}\frac{\partial\bar{u}_i}{\partial x_j} - \varepsilon$$

Where the material derivative of *k* on the left-hand side equals the sum of five terms:

- *Turbulent transport*, as turbulent fluctuations transport turbulent energy;
- *Pressure diffusion*, which corresponds to another term of transport due to pressure;
- Diffusion of turbulent energy due to the molecular transport process;
- *Production* of turbulent energy, as it represents the rate at which energy is transferred from the mean flow to turbulence;
- *Dissipation* of turbulent energy ε

To close the equation, the terms on the right-hand side must be approximated. The two terms of transport can be rewritten using the *gradient-diffusion hypothesis* as it follows:

$$\frac{1}{2}\overline{u_i'u_j'u_j'} + \frac{1}{\rho}\overline{u_i'p'} = -\frac{\nu_T}{\sigma_k}\frac{\partial k}{\partial x_j}$$

With σ_k as the turbulent Prandtl number ($\sigma_k = 1$) that will be explained in the next page. Moreover, the molecular transport is already a defined result (being ν a known variable) and

the production rate has been transformed using the Boussinesq hypothesis, which means that the dissipation term is the only one that must be elaborated. ϵ can be modelled as:

$$\varepsilon = C_D \frac{k^{3/2}}{l(x)}$$

Where *l(x)* is the characteristic length scale of the turbulence and C_D is a constant. Finally, the turbulent viscosity is:

$$\nu_T = k^{1/2} l(x)$$

Combining the last two equations, ν_T can be rewritten as a function of the kinetic turbulent energy and its dissipation rate:

$$\nu_T = C_\mu \frac{k^2}{\varepsilon}$$

Meaning that the characteristic length can be obtained knowing k and ε (C_μ is a constant and is $C_\mu = 0.09$). This means the solution requires the addition of a second transport equation, giving this specific approach its own name, the two-equations model. In the $k - \epsilon$ model, the second equation is solved for ϵ and is entirely empirical:

$$\frac{\partial \varepsilon}{\partial t} + \bar{u}_j \frac{\partial \varepsilon}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{\nu_T}{\sigma_\varepsilon}\frac{\partial \varepsilon}{\partial x_j}\right) + C_{\varepsilon 1}\frac{\varepsilon}{k}\left[\nu_T\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) - \frac{2}{3}k\delta_{ij}\right]\frac{\partial \bar{u}_i}{\partial x_j} - C_{\varepsilon 2}\frac{\varepsilon^2}{k}$$

Where the constants are considered with their standard value, arrived at by numerous iterations of data fitting for a wide range of turbulent flows [5]:

$$\sigma_k = 1;\ C_\mu = 0.09;\ C_{\varepsilon 1} = 1.44;\ C_{\varepsilon 2} = 1.92;\ \sigma_\varepsilon = 1.3$$

1.3.4 The k-ω model

This model can obtain better results when predicting separate flows or adverse pressure gradient flows. Similar to the k-ε model, the starting point are two partial differential equations resolved for two variables. If the equation for the turbulent kinetic energy k remains the same, the second equation considers ω, which is defined as the specific rate of dissipation of k into thermal energy, or as Wilcox described it in 1970 [6], "it's a frequency characteristic of the turbulence decay process under its self-interaction". The two variables are correlated as "the idea is that ω^2 is the mean square vorticity and k is the kinetic energy of the motion induced by this vorticity", again stated by Wilcox. In particular, the relation is:

$$\omega = \frac{\varepsilon}{C_\mu k}$$

The ω equation had evolved over the past decades. Here follows its latest version developed by Wilcox in his paper [6]. The constitutive equations are:

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$$

$$\bar{S}_{ij} = S_{ij} - \frac{1}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}$$

$$t_{ij} = 2\mu\bar{S}_{ij}$$

$$\rho\tau_{ij} = 2\nu_T\bar{S}_{ij} - \frac{2}{3}\rho k\delta_{ij}$$

$$\nu_T = \frac{\rho k}{\widetilde{\omega}}$$

$$\widetilde{\omega} = max\left\{\omega, C_{lim}\sqrt{\frac{2\bar{S}_{ij}\bar{S}_{ij}}{\beta^*}}\right\} \; where \; C_{lim} = \frac{7}{8}$$

While the governing equations for a generic flow for k and ω are:

$$\frac{\partial}{\partial t}(\rho k) + \frac{\partial}{\partial x_j}(\rho\bar{u}_j k) = \rho\tau_{ij}\frac{\partial\bar{u}_i}{\partial x_j} - \beta^*\rho k\omega + \frac{\partial}{\partial x_j}\left[\left(\mu + \sigma^*\frac{\rho k}{\omega}\right)\frac{\partial k}{\partial x_j}\right]$$

$$\frac{\partial}{\partial t}(\rho\omega) + \frac{\partial}{\partial x_j}(\rho\bar{u}_j\omega) = \alpha\frac{\omega}{k}\rho\tau_{ij}\frac{\partial\bar{u}_i}{\partial x_j} - \beta\rho\omega^2 + \sigma_d\frac{\rho}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial\omega}{\partial x_j} + \frac{\partial}{\partial x_j}\left[\left(\mu + \sigma\frac{\rho k}{\omega}\right)\frac{\partial\omega}{\partial x_j}\right]$$

The closure coefficients are:

$$\alpha = \frac{13}{25};\ \beta^* = \frac{9}{100};\ \sigma = \frac{1}{2};\ \sigma^* = \frac{3}{5};\ \sigma_d = \begin{cases} 0, & \frac{\partial k}{\partial x_j}\frac{\partial\omega}{\partial x_j} \le 0 \\ \frac{1}{8}, & \frac{\partial k}{\partial x_j}\frac{\partial\omega}{\partial x_j} > 0 \end{cases}$$

1.3.5 The *k-ω SST* model

The SST k-ω model combines the previews two, adapting itself in each situation to the considered flow. In fact, it uses the k-ω model in those problems that concern the viscous layer, opting for the k-ε if the considered scenario is a free-stream flow. This is decided based on a function F1, whose value determines which of the two models must be used

2 Machine Learning

2.1 Introduction to Machine Learning

Machine Learning (ML) is a way to use Artificial Intelligence. It was first defined in the 1950s by AI pioneer Arthur Samuel as "the field of study that gives computers the ability to learn without explicitly being programmed" and is now considered a subfield of artificial intelligence. A broader definition sees Machine Learning as the capability of a machine to imitate intelligent human behavior: AI's systems are used to perform complex tasks in a way that is similar to how humans solve problems. This means machines can recognize a visual scene, understand a text written in natural language, or perform an action in the physical world.

Machine Learning methods can be split in 3 main paradigms:

- Unsupervised Learning: Unsupervised learning uses machine learning algorithms to analyze and cluster unlabeled datasets [7]. These algorithms discover hidden patterns or data groupings without the need for human intervention. Its ability to discover similarities and differences in information make it the ideal solution for exploratory data analysis and image recognition. What happens is that the neural network learns a pattern from the inputs without having any output as reference target.
- Supervised Learning: The neural network receives both the input variables (x) and the correct output variable (Y) and uses an algorithm to learn the mapping function that relates the input to the output.

$$Y = f(x)$$

The goal is to approximate the mapping function so well that when you have new input data (x) you can predict the output variables (Y) for that data. Therefore, it is used in this paper's case study.

- Reinforcement Learning (or semi-supervised learning): The model must achieve a certain objective in a dynamic environment (examples could be driving a vehicle or playing a chess game against an opponent).

2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a subset of the Supervised Learning methods. The principles behind the supervised training process of an ANN are the following: first, the network creates a mathematical model based on the inputs and outputs it elaborates (this is called 'forward' process or 'inference'), then it calculates the 'loss' between the desired and the predicted outputs. The latter corresponds to an error that is used in the so-called 'backward' process to revise and adjust the previous model by modifying its parameters.

The architecture of an ANN reminds the structure of the brain; in fact, the basic unit of neural networks is the '*Neuron*', which is closely linked to the other units of the network in different and complex ways based on the specific goal it must achieve. The *Neuron* is a computational unit characterized by both scalar inputs and outputs. The process inside every neuron is the following: each input unit x is multiplied by its *weight* and then summed with the *bias* (a term that can be assumed as a constant added to the input value) and the result is finally passed through the non-linear function f (also known as the *activation function*) characterizing the neuron, giving the output y. It's the fundamental element of an ANN and it's responsible for all the basic operations; the neuron behaves in different ways, depending on its model. Each neuron model is characterized by its activation function that decides whether the neuron activates or not based on the input it receives.

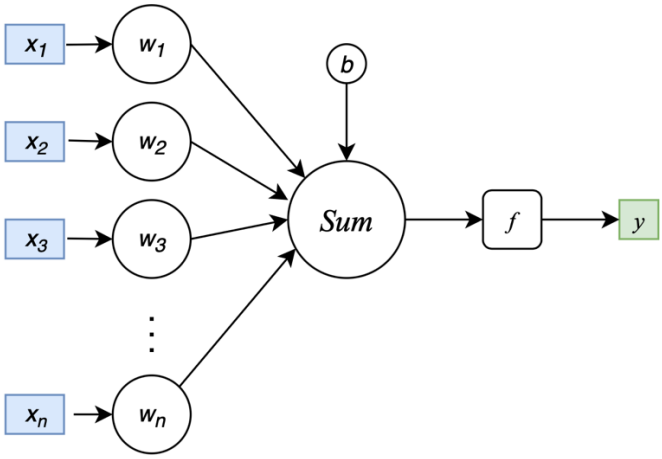


Figure 2.1 – Neuron scheme

ANNs consist of multiple groups of neurons organized in different ways; the basic structure of grouped neurons is called '*Layer*', which coincides with a set of neurons of the same category. The layers of a neural network can be generically divided into 3 main categories based on the specific elaboration step of the layer:

- Input layer: This set of neurons receives the input from the training dataset.
- Hidden layer: It computes the data received from the Input layer, generating parameters
- Output layer: It gives the results of the elaboration.

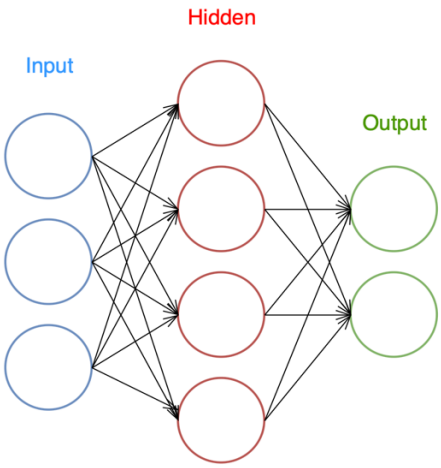


Figure 2.2 – ANN architecture

The architecture depicted above is the basic structure of a neural network, composed by a single input layer, only one hidden layer and one output layer, but it can be expanded by increasing the number of hidden layers.

2.2.1 Neural Network classification

Neurons can be classified depending on the mathematical model that describes their behaviour. There are three different generations:

1. **First generation – Perceptrons**

   The perceptron is a single layer neural network, based on the concept of the neuron of McCulloch-Pitts.

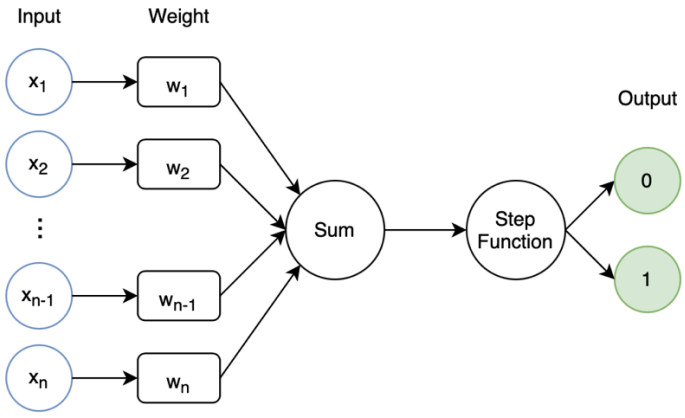


Figure 2.3 - Perceptron

As shown in figure 2.3, the perceptron applies the *step function* to the weighted sum of its inputs, meaning that the neuron will be either active (**1**) or inactive (**0**) depending on its result. If the sum is greater than a threshold value u – chosen a priori by the user – the neuron is activated. The mathematical expression synthetizes it clearly:

$$y = \begin{cases} 1, if\ h = \sum_i x_i w_i \geq u \\ 0, otherwise \end{cases}$$

Being binary classifiers, perceptrons can't be used for continuous data.

2. **Second generation**

The second generation of ANNs substitutes the step function (or binary function) with a continuous activation function *f*. The architecture corresponds to a more modernized perceptron, with the addition of a new element, the bias, as shown in figure 2.4:

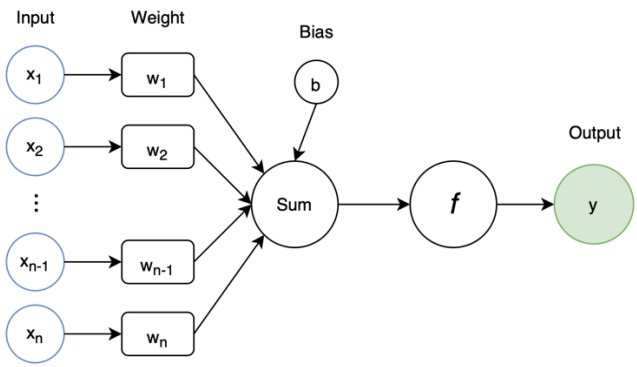


Figure 2.4 – Second generation of ANNs

and the output is calculated as follows:

$$y = f\left(\sum_i x_i w_i - b\right)$$

where f is the activation function and b is the 'Bias'. The activation function is the fundamental part, as it gives the ANN its nonlinear behaviour; this gives the neural network the chance to work on more complex problems.

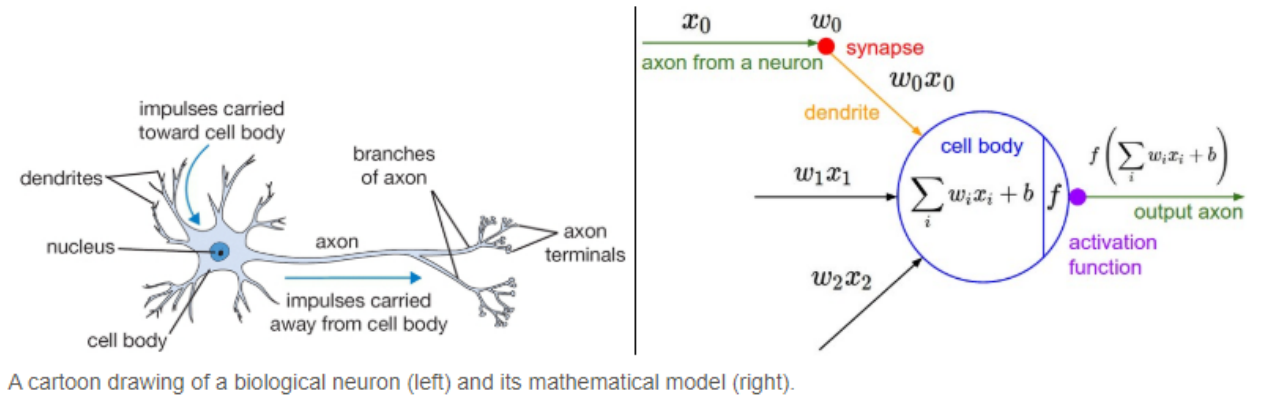
3. **Third generation - Spiking Neural Networks**

Recently, a third way of building Neural Networks has spread as research has tried to create models that are more biologically realistic. Therefore, Spiking Neural Networks (SNNs) have been developed [8]. The main idea is that neurons don't transmit information every propagation (as for example happens in the first two generations of ANNs), rather they fire their signal as their membrane potential - a built-in quality of the neuron – reaches a certain value. This enables a cascade of signals through the

network, where only certain neurons activate. Being so dynamic, this type of ANN is useful for time-dependent problems.

2.2.2 Activation function

The comparison with the neuron-based model is very helpful in order to understand the role of the activation function. In fact, as it happens in the human brain neuron, the transfer function *f* in an ANN decides what is to be signaled to the next neuron unit. It takes outputs from previous layers, transforming it to something that is used in the following neurons.



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Figure 2.5 – Comparison between human neuron and NN's neuron – [9]

The *activation function*'s main objective is limiting the output. The inputs must be restricted to certain values for the Neural Network to work, otherwise there is the possibility of obtaining an output very high in magnitude - as it equals the sum of all inputs – that would correspond to high computational complexity. In fact, if there wasn't a *f* function, the result would just be:

$$y = \sum_i x_i w_i - b$$

Moreover, let's pretend there is a model without transfer function between layers. If this was the case, all neurons would only perform a linear transformation. Since the combination of two linear functions results in an operation that doesn't lose its linearity, all layers would perform the same exact way. Even though the Neural Network becomes simpler, it doesn't have the ability of learning complex models and it only resembles a linear regression operation. Therefore, the transfer function needs to be applied in all layers, and it must be non-linear. It usually differs if the function must be used for the hidden nodes or for the output layer, as the latter depends on the type of prediction required by the Neural Network.

Although *linear* activation functions exist, the focus developed on the *non-linear* is far more important for this project. In fact, non-linear transfer functions manage to allow

backpropagation, which is of key importance in machine learning, as it allows neural networks to compute the weights step-by-step and understand which ones provide a better prediction. Moreover, these functions grant the possibility of multiple layers models, given the non-linearity of the combination of inputs, allowing a higher complexity of the model.

The activation function should have desired characteristics [10, 11]. First, the function must not shift the gradient of the model towards zero, which is what is called the *Vanishing Gradient* problem. Moreover, it must be *zero-centered* without shifting its output consistently in a particular direction, and it should be inexpensive to calculate, as it is applied in each layer. Finally, the transfer function needs to be *differentiable*, which is a necessary requirement since ANNs are trained using the gradient descent process (which will be explained later in this chapter), where the value of the gradient (derivative) influences the later steps of the neural network, so it's important that the value isn't zero.

Here follows the presentation of a few activation functions, given that the application of each one brings to a different solution for the neural network.

2.2.2.1 Sigmoid function

The function is:

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

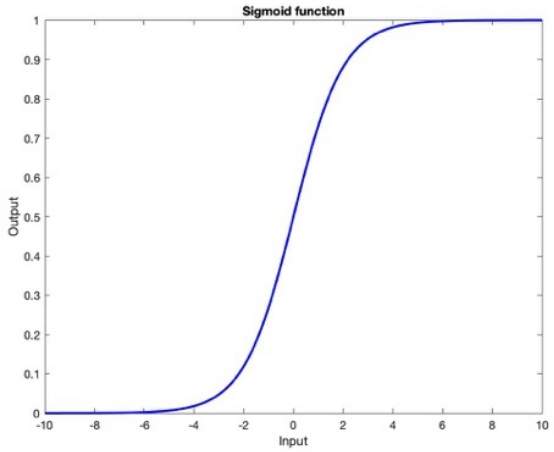


Figure 2.6 – Sigmoid function

This activation function is also called logistic function as it resembles a probability function. Indeed, the sigmoid function outputs values between 0 and 1, the same range of the probability. As figure 2.6 shows, the larger the input, the closer the output value will be to 1; the smaller the input, the closer y is to 0.

The sigmoid function suffers from the vanishing gradient problem for high absolute values of its input; in fact, the study of its derivative in comparison with the primitive function shown in figure 2.7 displays that the value of the derivative is meaningful only between -5 and 5;

outside of this interval the function is almost 0, which means that the neurons in the following step are not activated.

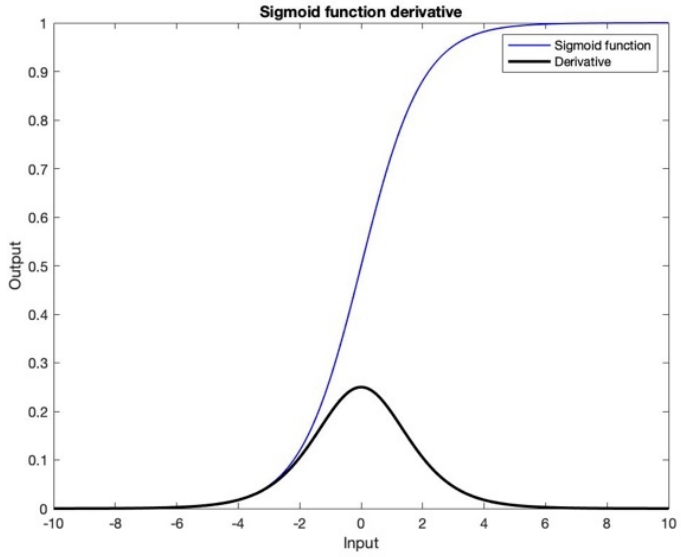


Figure 2.7 – Sigmoid function derivative

2.2.2.2 Hyperbolic tangent function

This function is also referred to as the *tanh function*. It is very similar to the previous one, as it has the same "*S shape*" varying from -1 to 1. The expression is:

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

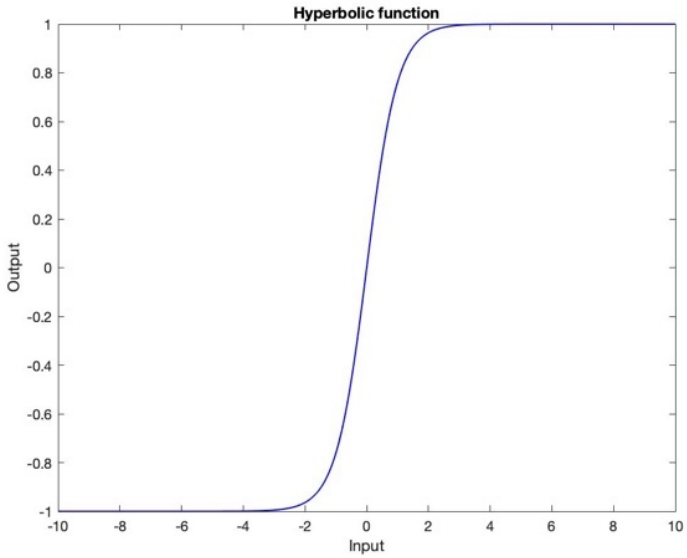


Figure 2.8 – Hyperbolic tangent function

This transfer function has the characteristic of being zero-centered. Moreover, it's very common to apply the *tanh* function to hidden layers, as it makes the learning in the following layers easier. In fact, the output comes out very close to 0, which also helps to center the data.

This function is also affected by the vanishing problem; the figure 2.9 shows the derivative calculated for the values outside the interval $-4 < x < 4$ is very close to zero:

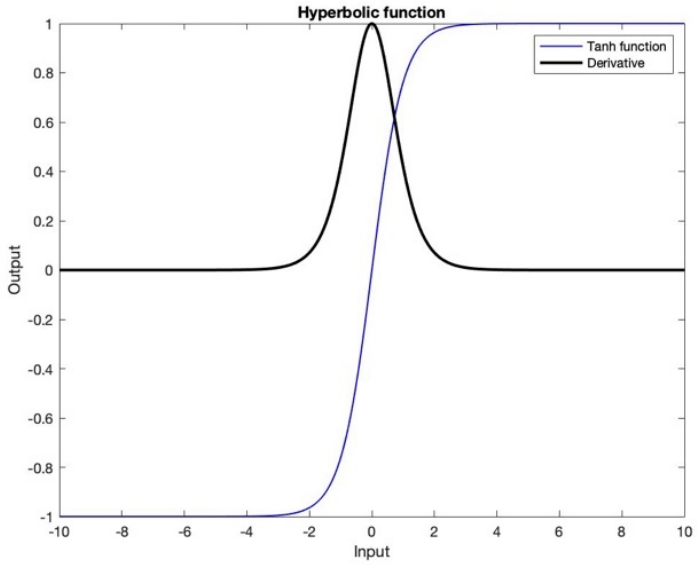


Figure 2.9 – Hyperbolic tangent function derivative

It is important to note that the non-linearity of the tangent function is preferred to the one characterizing the sigmoid function even if its gradient is steeper. This is because it's preferable to have a zero-centered result.

2.2.2.3 Rectified Linear Unit function

This function, also called *ReLU* or *ramp function*, is the most used transfer function. The function is very efficient as there are fewer vanishing gradient problems and the number of neurons active at the same time is reduced; in fact, all nodes that have negative inputs are returned as 0 by the ReLU. The expression is:

$$y = x^{+} = \max(0, x)$$

And it's graphed in figure 2.10:

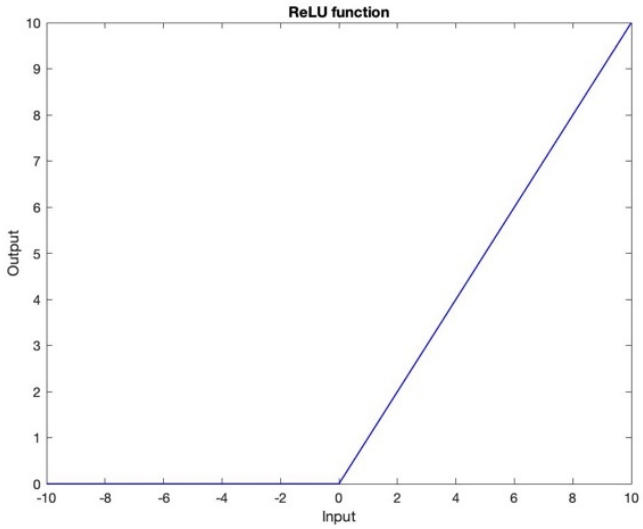


Figure 2.10 – ReLU function

However, this function is characterized by a limitation called *Dying ReLU problem*, which corresponds to the fact that during the backpropagation process some neurons are not updated, meaning that the gradient doesn't flow through them, causing their "death" as they are never activated. In fact, the derivative shows that for all negative values the gradient goes to zero.

Moreover, differently to what was stated earlier, the ReLU function isn't differentiable, as it can be seen from the graph in figure 2.11 that the derivative has two different values when $x = 0$. This may seem like this function isn't eligible for use in gradient-based optimization algorithms, but in practice the gradient descent still performs well enough for machine learning tasks. This is because neural network training algorithms do not usually arrive at a local minimum of the cost function., hence it is acceptable for the minima of the cost function to correspond to points with undefined gradient. A different way of coping with the problem, as suggested in [12], is by replacing the value of the derivative in $x = 0$ with a standard number chosen between the right or left derivative (0 or 1) or their mean (0.5). Finally, it must be said that even though this transfer function is preferred for its computational simplicity, it isn't zero-centered.

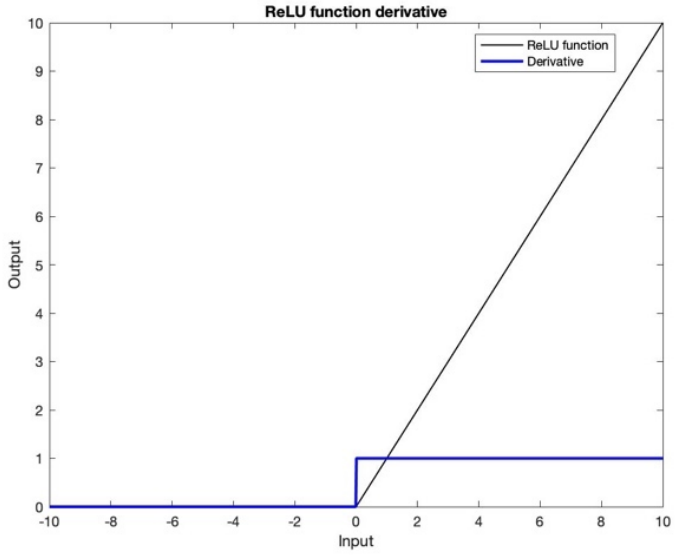


Figure 2.11 – ReLU function derivative

The dying ReLU problem can be mitigated by using an updated ReLU function, called *Leaky ReLU*, which is defined as:

$$y = \begin{cases} x & if\ x > 0 \\ 0.01x & otherwise \end{cases}$$

This function has the same advantages of the ReLU, to which it adds the functionality in the backpropagation process as it can be seen in figure 2.13, the graph representing the derivative of the function.

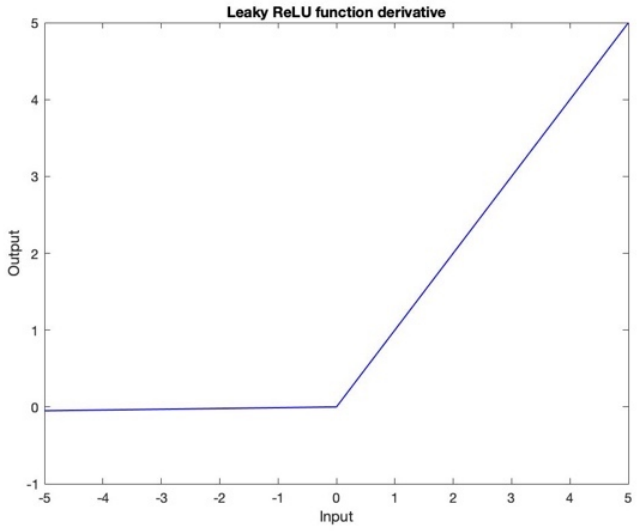


Figure 2.12 – Leaky ReLU function

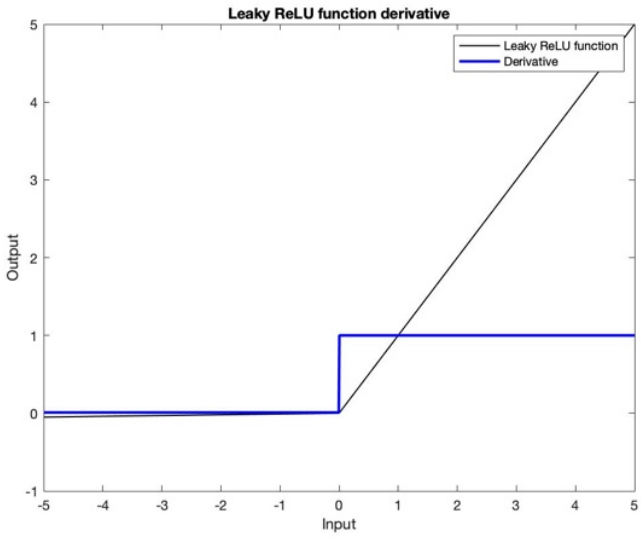


Figure 2.13 – Leaky ReLU function derivative

2.2.3 Backpropagation algorithm – Gradient Descent method

Until now we have not discussed how the ANNs define their parameters during the computational process. In fact, all the elements that have been introduced earlier – weights and biases – must be determined so that the model can be used for predictions. As explained in [7], Machine Learning is solving massive optimization problems. The Neural Network is merely a very complicated function, consisting of millions of parameters, that represents a mathematical solution to a problem. The goal of the training is obtaining the best combination of parameters to minimize a loss function; the value of this loss function gives us a measure of how far from perfect the performance of the network is with a given dataset.

Fitting the Neural Network corresponds to computing the gradient of a predefined loss function with respect to the weights and biases, meaning that the objective is calculating the following partial derivatives:

$$\frac{\partial E}{\partial w} \; and \; \frac{\partial E}{\partial b}$$

Where E is the chosen loss function that will be presented later.

In mathematics the gradient of a function f measures the sensitivity to change of the function value with respect to a change in its argument, which means that finding these derivatives allows the evaluation of the changing of the parameters. In fact, knowing these expressions enables to calculate the value of the weights and the biases step after step with the formula:

$$w_{ij}{}' = w_{ij} - \epsilon \frac{\partial E}{\partial w}$$

$$b_{ij}{}' = b_{ij} - \epsilon \frac{\partial E}{\partial b}$$

Where w_{ij} and b_{ij} correspond to the weight and the bias of the initial step and ϵ is the learning rate. The latter is the trade-off between the rate of convergence and overshooting of the model and must be tuned before launching the algorithm. As the descent direction is already defined by the gradient, the learning rate regulates how big the step in that direction is. $w_{ij}{}'$ and $b_{ij}{}'$ are the values of the weights and biases that must be defined each step and then used for the next parameter calculation; for example, a 2-step training would be the following:

$$Step\ 1 \rightarrow \begin{cases} w_{ij}^1 = w_{ij}^0 - \epsilon \dfrac{\partial E}{\partial w} \\ b_{ij}^1 = b_{ij}^0 - \epsilon \dfrac{\partial E}{\partial b} \end{cases}$$

$$Step\ 2 \rightarrow \begin{cases} w_{ij}^2 = w_{ij}^1 - \epsilon \dfrac{\partial E}{\partial w} \\ b_{ij}^2 = b_{ij}^1 - \epsilon \dfrac{\partial E}{\partial b} \end{cases}$$

In order to achieve optimal results, the correct *error function E* (loss function) must be defined; it tells how much the output of the neural network differs from the expected result found in the database. As stated in [8], "the choice of cost function is tightly coupled with the choice of output unit". Based on [9], the preferred *E* for Regression problems such as the one considered in this project is the *Mean Squared Error* (MSE), which measures the average of the squares of the errors. For example, the MSE calculated after a single step in a neural network would be:

$$E = \frac{1}{2}|y_{NN} - y_{DB}|^2$$

Where y_{NN} is the output value estimated by the Neural Network, while y_{DB} corresponds to the real output value found in the database.

For N neurons it can be written as an average:

$$E = \frac{1}{2N}\sum_{x=1}^{N}|y_{NN}(x) - y_{DB}(x)|^2$$

In alternative, the cross-entropy function could be used, even if this is preferred for classification problems [18], and it is defined as [19]:

$$E_{CE} = \frac{1}{N}\sum_{class}^{N}\left(-log\left[\frac{e^{x_{class}}}{\sum_i e^{x_i}}\right]\right)$$

Where x_{class} is the output with the target class index which is normalized with respect to the sum of the outputs; the result of the logarithm is the averaged over the number of classes.

As E is evaluated, the model can adjust its parameters to get closer to the expected output by knowing its derivatives. Here follows the procedure for a simple neural network (figure 2.14).

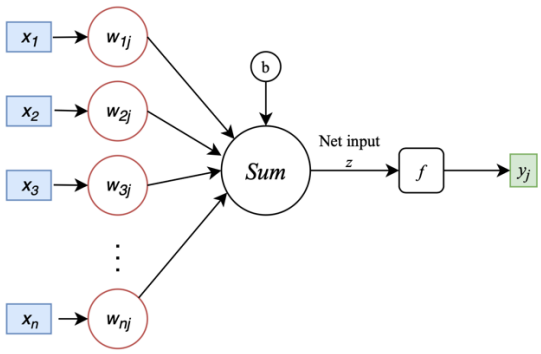


Figure 2.14 – NN scheme

In order to calculate the final output, the different gradients need to be derived every step. The following process is required to obtain a definition for $\frac{\partial E}{\partial w}$ *and* $\frac{\partial E}{\partial b}$ as a function of the error δ_j. First, the chain rule is applied twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}$$

Where z_j is defined as the input for the activation function *f*:

$$z_j = \sum_{i=1}^{n}\left(w_{ij}y_j + b\right) \rightarrow \frac{\partial z_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(w_{ij}y_j + b\right) = y_j$$

Which implies that:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}y_j$$

The derivative of the output *y* is:

$$\frac{\partial y_j}{\partial z_j} = \frac{\partial f(z)}{\partial z}$$

It's important to note that for this reason there is the aforesaid requirement for the activation function to be differentiable and the result depends on the activation function used in the model. For example, in the case of the sigmoid function:

$$\frac{\partial y_j}{\partial z_j} = \frac{\partial f(z)}{\partial z} = f(z)\big(1 - f(z)\big)$$

As the function applied to the sum z_j corresponds to the output of the neuron.

The equation can be rewritten as a function of a new term, that is defined as the error δ_j:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j y_j$$

Where $\delta_j = f'(z)\frac{\partial E}{\partial y_j}$ and can be found using the Hadamard product between the two indicated matrixes $f'(z)$ and $\frac{\partial E}{\partial y_j}$. Once its value is determined, the derivative is quickly computed, so is the new value of the weight.

The same process is followed to define the changing of the bias:

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial b}$$

Where:

$$\frac{\partial z_j}{\partial b} = 1$$

It follows that:

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j} * 1 = \delta_j$$

As in the previous case, the derivative depends on the value of δ_j.

Finally, as said, determining the *local gradient* δ_j enables to calculate the new values of the weights and the biases step by step, setting up the problem for the backpropagation algorithm. Figure 2.15 shows the process that the Neural Network follows for each weight calculation at every step.

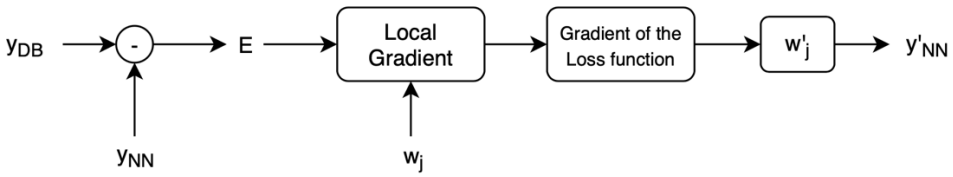


Figure 2.15 – Flowchart for the backpropagation algorithm.

3 CFD Simulations and NN Definition

In this chapter the procedures related to the configuration and setup of the backward facing-step CFD simulations are presented and discussed. The second part involves the analysis of the setup, features and architecture of the NNs developed in this project.

3.1 The Backward-facing step

The case study of the thesis is the benchmarked geometry of the backward-facing step, which is the most representative model for flow separation [20]. In fact, it analyses the development of a 2D motion field occurring when the fluid flow encounters an expansion and separates. As the flow reaches the step, it detaches forming a recirculation zone, due to a stationary vortex.

Figure 3.1 – Recirculation zone in backward-facing step case

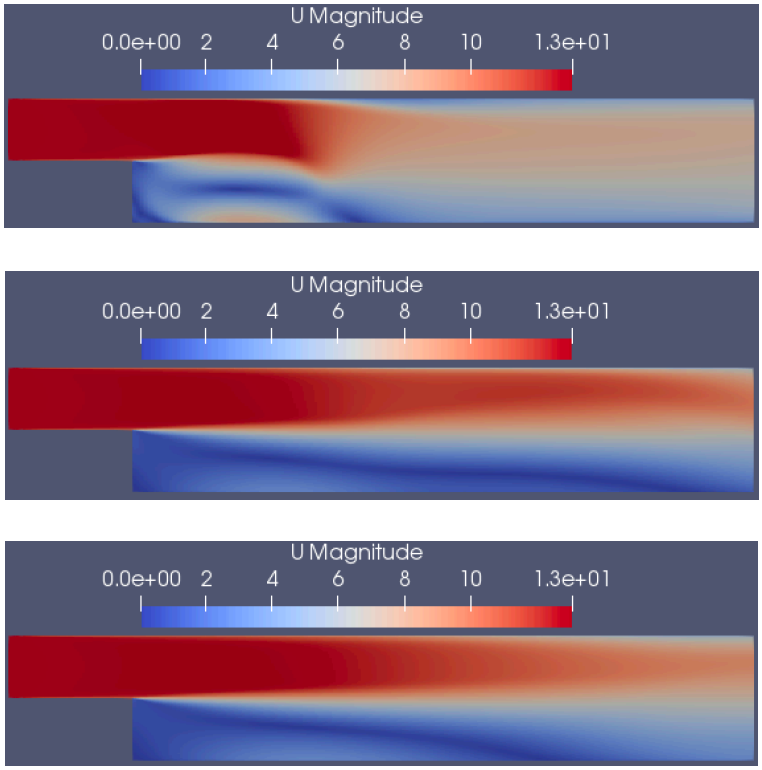


Figure 3.2 – Detail of the CFD simulation which highlights the recirculation zone. From top to bottom, the time of the simulation increases. As the Reynolds number remains constant, t=0.1s in the top scenario, t=0.3s in the middle figure, while t=0.5s in the last image.

The case is bidimensional, incompressible and turbulent.

The software used to create the mesh and run the simulations is OpenFOAM, which is supplied with both the pre- and post-processing environments.

3.1.1 Geometry

The project is characterized by a 2D geometry domain, which is set up by using the blockMesh dictionary. However, the solver needs the figure to be three-dimensional to work; when the 16 nodes that build the geometry are generated, it is important that they create a width along the z-direction. This is simply managed by giving a minimum z-coordinate in both positive and negative direction as the nodes are split in two different planes. The domain is made of three blocks: one corresponds to the inlet box, the remaining two constitute the volume after the step, which is divided in two identical parts.

The resulting geometry is:

Figure 3.5 – 2D geometry

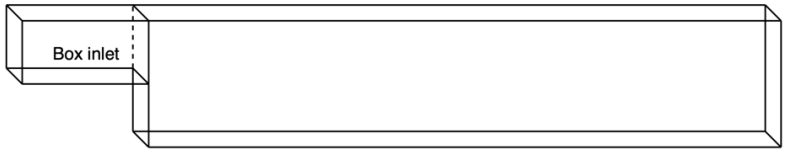


Figure 3.6 – 3D geometry

3.1.2 Mesh setup

The blockMesh utility is fundamental for creating the mesh for the configuration.

As the results of the simulations correspond to the outputs of the training of a Neural Network, the grid must be quite coarse to reduce the computational cost in the second phase of the thesis. Nonetheless, to have reasonable results, the number of cells must not be too small either. Therefore, the mesh is constituted of 50x64 in the inlet block (as the fidelity of the results isn't important in this area), in addition to 300x128 cells in the volume after the step. Consequently, the total of cells is:

$$Cells = 41600$$

It is important to note that the number of cells in the y-direction of the outlet box is 128; this is the number of velocity components the NN will have to predict.

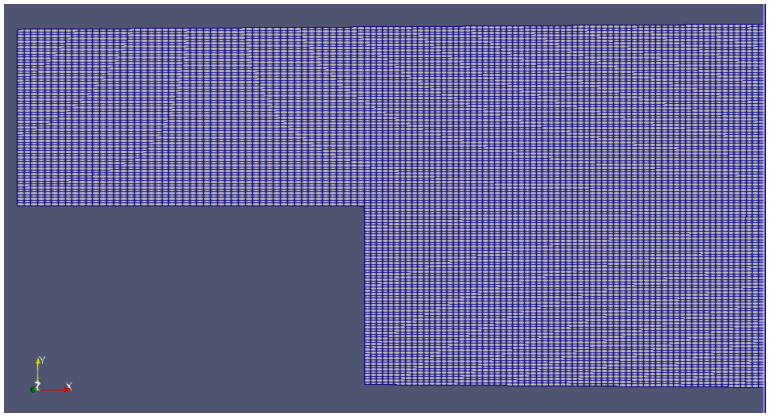


Figure 3.7 – Detail of the mesh grid.

3.1.3 The CFD solver

The solver used to run the simulations needs to have determinate characteristics. From the vast library of incompressible solvers in OpenFOAM, the one that adapts the best must be able to work for different time values, where time is intended as physical time and not number of iterations.

This means that the SimpleFoam solver, the easiest to use, must be discarded as there is no possibility of requesting a specific time value (in terms of seconds). The choice falls onto PisoFoam. PisoFoam is defined as an incompressible solver, able to resolve both laminar and turbulent flow. It's single phase and isothermal, which means that there is no energy equation.

The equations SimpleFoam must solve are [21]:

- Continuity equation

$$\nabla \cdot \vec{u} = 0$$

- Momentum equations:

$$\frac{\partial \vec{u}}{\partial t} + \nabla \cdot (\vec{u}\vec{u}) = -\frac{1}{\rho}\nabla \vec{p} + \nabla \cdot \nu[2S] + F_t$$

Which are not solved separately but combined to solve the PISO – Loop. This model, which in this case operates via RANS (Reynolds-Averaged Navier Stokes equations), neglects the velocity oscillations, instead generating an error solved through the turbulent viscosity equation. The latter models the transport and dissipation of energy neglected by the averaging. The correction is then made on the dynamic viscosity ν, which becomes the sum of two different terms, the physical viscosity and the turbulent viscosity:

$$\nu_{eff} = \nu + \nu_t$$

The turbulent viscosity ν_t is calculated depending on the chosen turbulence model, which is the standard k-ε model for this project already discussed.

The aim of this Feed-Forward Neural Network is to learn the relations between the free-stream conditions of the flow and the value of the velocity components at a predetermined x, which in this case is the non-dimensional value:

Figure 3.8 – Detail of the x position in the geometry.

$$\frac{x}{10L} = 0.15$$

The CFD solution is calculated in 128 cells, meaning there are 128 values for the velocity component x and 128 values for the velocity component y.

The dataset is constituted by 150 simulations with Reynolds number in the range *5000-50000*, with $\Delta Re = 300$. The number of data increases as the time intervals are considered; in fact, we use $\Delta t = 0.2\ s$; since the simulations are run for a total time of 5 seconds, each solution is calculated for 25 intervals.

This makes the dataset used for the training of the Neural Network very large, as the total number of velocity components corresponds to:

$$N = 150 * 25 = 3750$$

This number corresponds to the effective initial dataset; it consists of 3750 set of vectors of velocity components, meaning that each vector comprises 256 elements. However, the final number comprises both the x-component and the y-component, where each vector includes 128 elements in both directions:

$$v_{x+y} \in R^{256} \rightarrow v_{x,y} \in R^{128}$$

3.2 The Feed-Forward Neural Network

Feed-Forward Neural Networks (FFNN) are the most popular and most widely used models in many practical applications. They are called feedforward because information flows through the network and there aren't feedback connections in which outputs of the model are fed back into itself.

A feed-forward neural network is a biologically inspired algorithm [22]. It consists of several simple neuron-like processing units, organized in layers and every unit in a layer relates to all the units in the previous layer [23].

This specific type of neural network is composed of multiple perceptrons and is also called *Multi-layer Perceptron*. The multitude of neurons allows the learning of the nonlinear functions necessary for the gradient descent method. All data enters at the input and passes through the network, layer after layer, until it arrives at the output. The input layer consists of just the inputs to the network. Then follow one or more hidden layers, which consist of any number of neurons, or hidden units, grouped together. Each neuron performs a weighted summation of the inputs which then passes the activation function before reaching the following layer. During normal operation there is no feedback between layers.

It's important to note that in MLPs all neurons of one layer are connected to all neurons of the following one. Moreover, the number of hidden layers becomes a fundamental parameter that influences the nonlinearity of the network and the capability of the scheme to predict the solution; furthermore, the more the number of hidden layers, the more complex the problem the network can solve.

In general, FFNNs are used to understand the link between independent variables, which serve as inputs to the network (in this case the Reynolds number and the time interval), and the independent variables that are designated as outputs.

3.2.1 The Feed-Forward Neural Network Structure

The architecture of the Neural Network is simple, as the information flows forward from inputs to outputs passing the hidden layers; every node is connected to all the nodes of the following layer. The figure below shows the FFNN's layout, highlighting the fact that there are only 2 inputs and 128 couples of output velocity [24]:

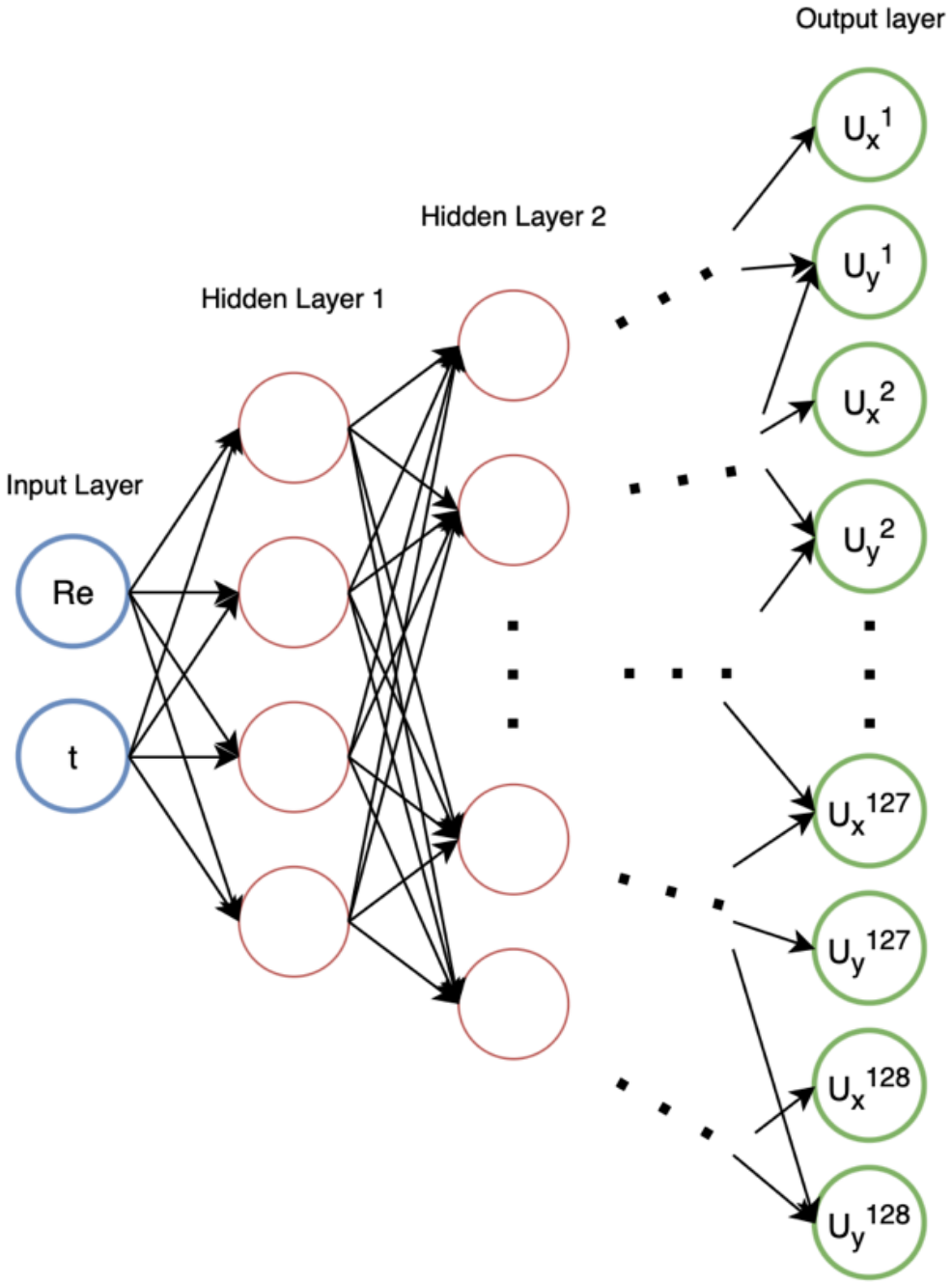


Figure 3.9 – FFNN's architecture

3.2.1.1 The dataset

The dataset for the neural network is defined as the following: to each combination of Reynolds number and specific time instant characterizing the simulation corresponds a vector of velocity components of 256 elements, as reported in table 3.1:

Table 3.1 – Dataset for FFNN, with $N = 3750$.

INPUT	OUTPUT
$[Re, t]_1$	$[U_{x_1}, U_{y_1}, U_{x_2}, U_{y_2}, \dots, U_{x_{128}}, U_{y_{128}}]_1$
$[Re, t]_2$	$[U_{x_1}, U_{y_1}, U_{x_2}, U_{y_2}, \dots, U_{x_{128}}, U_{y_{128}}]_2$
⋮	⋮
$[Re, t]_N$	$[U_{x_1}, U_{y_1}, U_{x_2}, U_{y_2}, \dots, U_{x_{128}}, U_{y_{128}}]_N$

First, given the difference of the order of magnitude of the inputs, it is mandatory to standardize the data as the network would otherwise amplify the importance of the Reynolds number (whose order of magnitude is $10^3 - 10^4$) completely neglecting the time instance ($10^{-1} - 10^0$), resulting in incorrect training.

Secondly, the set must be split into three categories for the training to be the most accurate. In fact, there are two competing concerns: if the training data is small, the parameter's estimation will have greater variance, meanwhile if the testing data is too little, the performance of the NN will be less accurate. Therefore, the initial data must be split to have both results balance. Finally, the third subset, called validation subset, is fundamental to avoid snooping (overfitting). Consequently, the 75% of data corresponds to the training set, 15% is dedicated to validation and the remaining 10% is for testing.

It is important to perform the split randomly, otherwise the test won't be reliable as ordering could cause overestimation (when there is some meaningful ordering) or underestimation (if the distribution differs by too much).

3.2.1.2 Hyperparameters

A model hyperparameter is a structure whose value cannot be estimated from data and must be specified by the user. Hyperparameters are used to control the learning process as they don't influence the performance of the model but affect its speed and quality. The time required to train and test a neural network depends upon the value of these parameters.

There isn't a way to define the best values on a given problem, but only the best compromise can be found [25]; therefore, the behaviour of the NN can be studied varying in turn a single hyperparameter while maintaining the others constant. This process is called hyperparameter tuning and results into finding the optimal solution for the NN's structure (the results of the analysis will be shown in the next chapter).

The hyperparameter's choice depends on the task [26]. Here, the varying parameters are:

- **Number of Epochs**: a single epoch equals to training the neural network with all the training data for one cycle [27]; ergo, in an epoch all the data is used exactly once. This number is usually very large, as the model must sufficiently minimize the error of the cost function. Obviously, the higher is the epochs' number, the longer the NN will take to train.

- **Number of layers and nodes**: as the type of architecture of this FFNN is defined from the beginning (being that the output is composed of 256 nodes, the only structure that makes sense to avoid a very large number of neurons is the "funnel architecture" shown in figure 3.9), the variables are the number of hidden layers and the number of nodes that characterizes them. It's important to highlight the difference between a higher number of layers and nodes. In fact, the capability of reproducing more complex problems strictly depends on the number of hidden layers. This is due to the fact that more layers add a higher nonlinearity to the model, being that the activation functions are implemented more than once through the network [28].

  To better show the different solutions adopted and studied for the FFNN model, the various architectures are shown in table 3.2:

Table 3.2 – FFNN's architecture, number of layers and nodes

Hidden Layers					
$h = 1$	$h = 2$	$h = 3$	$h = 4$	$h = 5$	$h = 6$
40					
30	60				
20	40	80			
20	40	60	80		
20	40	60	80	100	
20	40	60	80	100	120
60	60	60	60	60	60

- The **batch size**: it is defined as the number of samples that will propagate through the network. For instance, if the batch size is 50, the algorithm takes the first 50 samples from the training dataset and trains the network (defining its internal parameters). Next, it takes the second 50 samples and trains the network again, *updating the internal parameters*, which is the key point of the gradient descent method. This process goes on until all samples have been propagated. Using a small batch size is optimal for the solution's accuracy, while the downside is prolonging the training time, which means increasing the computational cost. Once again, table 3.3 shows the values used for the tuning of the network:

Table 3.3 – FFNN's grid search, batch size

Batch size					
10	25	40	60	100	150

- The **activation function**: the chosen activation function, defined earlier in chapter 2, is important to accelerate the solution and discard some results that may hinder its accuracy. The choice is important, as the right function reduces to a minimum vanishing gradient problem or, in some scenarios, exploding gradient problems. The tuning considers the following, as we try to consider functions with completely different characteristics:
    - Sigmoid function
    - Rectified Linear Unit function
    - Hyperbolic tangent function

In order to understand the influence of each aforesaid hyperparameter, a *grid search* (also defined as *hyperparameter tuning*) was performed, using the values shown in the tables. The results will be shown in the next chapter.

3.3 The Recurrent Neural Network

Feed-Forward Neural Network allow signals to travel only one way: they are straightforward NN that associate inputs with outputs. This means that there isn't feedback; the output of any layer doesn't affect the same layer.

As they are extended, they become Recurrent Neural Network (RNN). Feedback networks can have signals travelling in both directions by introducing loops; moreover, computations derived from earlier input are fed back into the network, which gives these configurations a kind of memory. RNN are dynamic [29]; their '*state*' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found.

RNN are adapted to work very well with time series data as in this project's scenario [30, 31]. Neurons in these types of networks are characterized by a memory that helps store the already mentioned *state* and the information of previous inputs to generate the following output. This helps if the aim of the network is to find a correlation between a prediction and the changing time. In fact, the CFD solution at the time instant t^* is a consequence of all its precedent solutions ($sol_{t^*} = f(sol_{t<t^*})$ so that this can be considered a sequence of velocity vectors that change as a function of time. The following representation is very explicative, as the '*rolled*' RNN is unraveled and it becomes clear that the future outputs are a function of all inputs:

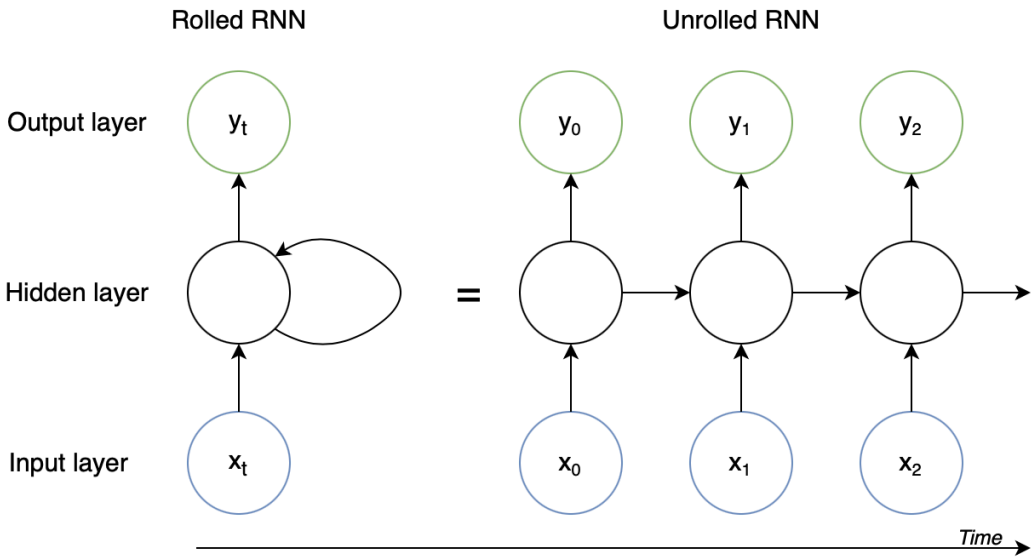


Figure 3.10 – Unrolled RNN – Importance of time sequence.

In this project, this can be translated in:

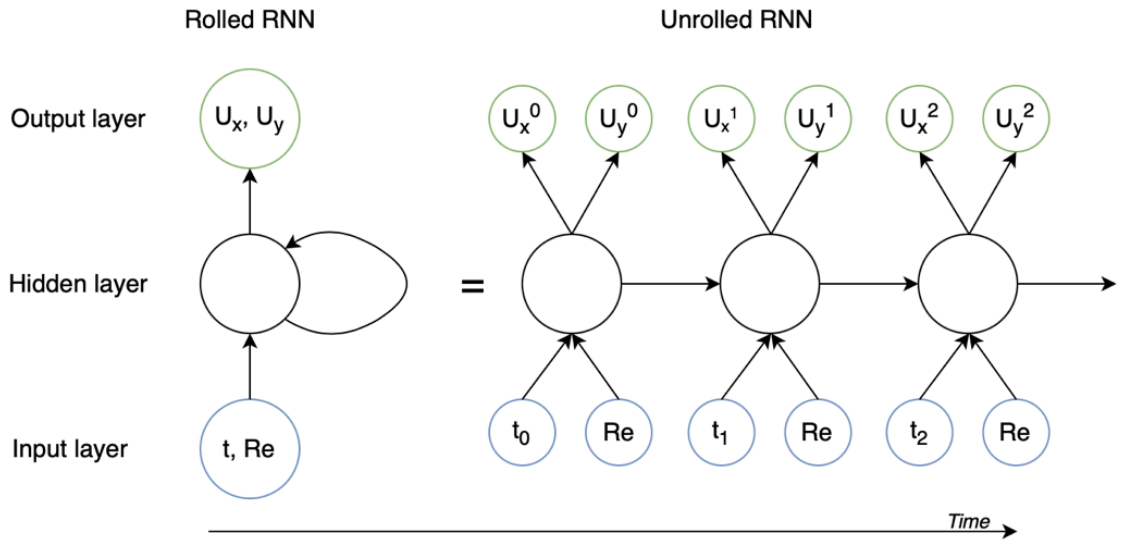


Figure 3.11 – Scheme for RNN.

Where the Reynolds number must remain the same while the Network studies the sequence of velocity components as a function of the time.

Moreover, RNN adopt the same weight within each layer, which is again adjusted step-by-step through the classic backpropagation through time algorithm (*BPTT*), that slightly differs from the precedent one – backpropagation algorithm - as in this case all errors across each timestep are accumulated as the network is unrolled. Once the network is rolled up again, the weights are updated before repeating the same process. Depending on the number of time steps, this mechanism is very consuming. In addition, RNNs suffer from vanishing gradients problems (as in the FFNN's case), and they may run into exploding gradient more often, which is why the number of hidden layers is reduced to a minimum.

Considering the time dependency, it should be able to represent the results with more accuracy with respect to the previously described FFNN.

RNN's hyperparameters were mainly decided based on the literature. In fact the general rule of thumb for the number of hidden layers for RNN problems is 1; two hidden layers are used in case of a very complex problem. It is straightforward that, if the number increases, the precision of the model grows and so does the computational cost; however, literature states that one [30] is a sufficient number to obtain good accuracy results. This RNN is characterized by 1 hidden layer.
The choice of the activation function is between the Sigmoid and the Hyperbolic Tangent as the ReLU – the one used for the Feed Forward NN – suffers from exploding values, meaning that the gradients continue to increase as the backpropagation algorithm proceeds. Being the sigmoid function the less subjected to this problem (looking at the derivative it's very clear why, as the gradient is low for every number considered), it is used in this model.

The tuning process occurs in the exact same way as in the FFNN. However, as the aforementioned hyperparameters (number of hidden layers and the activation function used) were chosen through literature, the only thing that needs to be tuned for the RNN of this project is the neurons number constituting the hidden layer. As said, the RNN is mainly used with a single hidden layer. Following the general rule of thumb to hypothesize the number of units [25]:

$$N_h = \frac{2}{3} * (N_i + N_o) \cong 166$$

This number corresponds to a starting point from which can be guessed a good-enough configuration. In fact, the numbers of nodes tested in the RNN's hidden layer are 120, 140, 160 and 180.

4 Results

In this conclusive Chapter, the results obtained during this thesis project are presented and discussed. The First Section shows the results of the hyperparameters search, while the second one displays the reconstruction of the velocity components with the use of the different NNs implemented in this paper.

4.1 Feed-Forward Neural Network

4.1.1 Hyperparameters Tuning

This section proceeds to show the different analyses undergone to define the hyperparameters that characterize the Feed-Forward Neural Network used.

First, as it is the main variable of the cost of the computation, it is important to search the best epochs value. Since a higher number would give overfitting problems and a lower number would underfit the initial dataset, the choice needs to be optimal. The goal is to minimize the loss function previously specified. Initially, the error falls fast as the model improves substantially, but as the number of epochs increases, there is a loss plateau, which means that the global loss minimum has been reached.

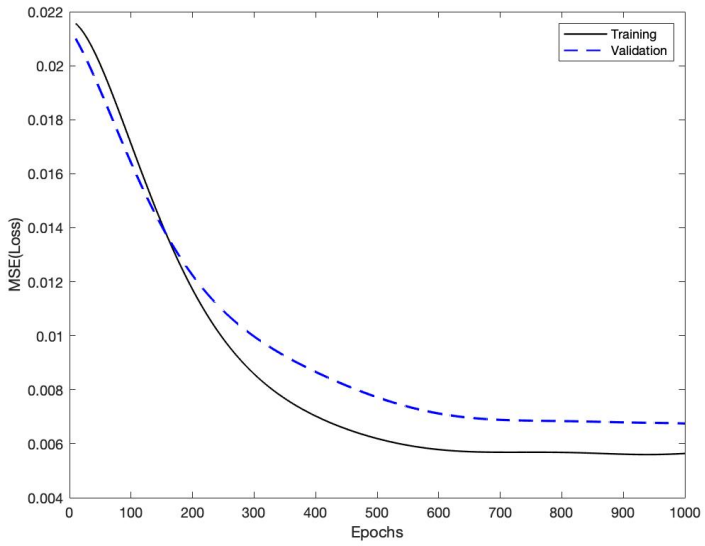


Figure 4.1 The training and validation loss as a function of training epochs. Note that the curves shown are averages of 4 repeated runs.

Therefore, the number of epochs chosen is 600, which is the best trade-off between accuracy and time. In addition, it is important to note that early-stopping was implemented as a regularization during training.

In general, the results present very low value of error as the loss due to the training process has an order of magnitude of $6{,}0 \cdot 10^{-3}$, which makes the model reliable and accurate.

Following the choice of the epochs number, the correct Feed Forward NN's architecture needs to be defined. This means that it must be decided the optimum number of hidden layers and the number of neurons comprising each layer.

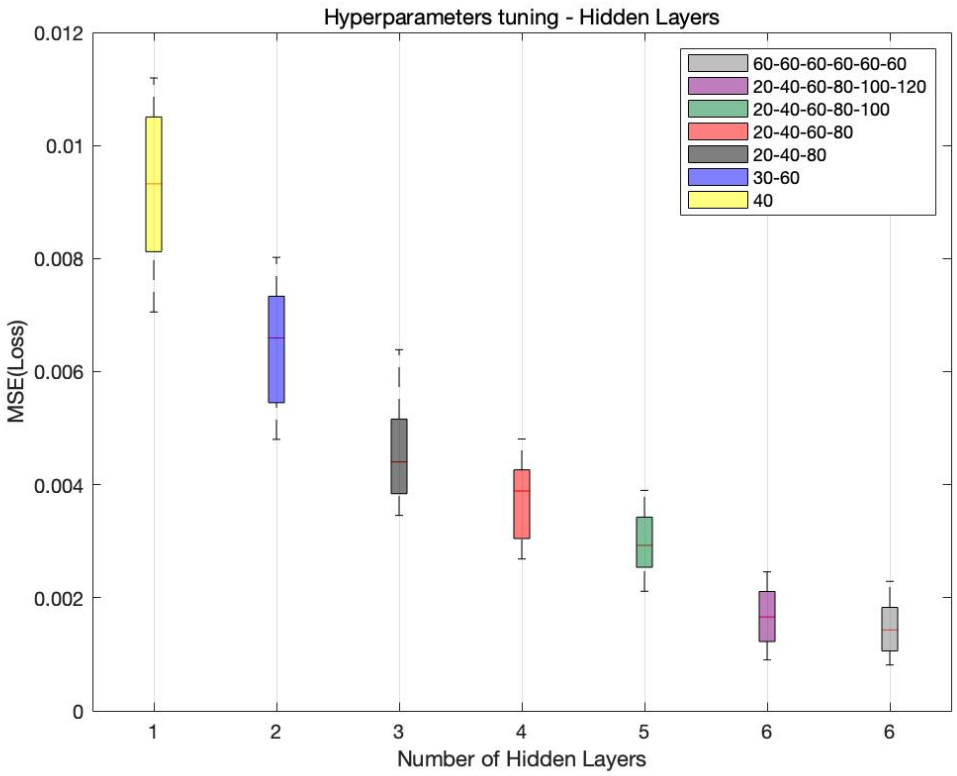


Figure 4.2 Comparison between different NN's architecture.

The following graph compares the different layouts aforementioned. As it could have been expected, the error diminishes if the hidden layers increase. Obviously, the time for the training differs a lot; since it is necessary to find the 'best' trade-off between time complexity and qualified accuracy, the optimal architecture is the one with 5 hidden layers with the respective number of neurons of 20-40-60-80-100.
Consequently, the decided configuration is different from the theorical assumptions, where it was stated that the optimal number of hidden layers for a NN's architecture is three. The choice is due to the augmented precision of the solution by a factor of almost 0.002, which diminishes the loss by half of the best-theorical case.

Before proceeding to the effects of the inputs on the model, it's important to tune for the batch size and the activation function for the chosen configuration. As explained in the previous chapter, the ideal batch size is very small for the solution to be super accurate. Nonetheless, the smaller the batch size, the higher the time cost; the training time increases by almost 1 hour from the model with maximum batch size and the one with a batch size of 10. Once again, it's necessary to search the 'best' trade-off model.

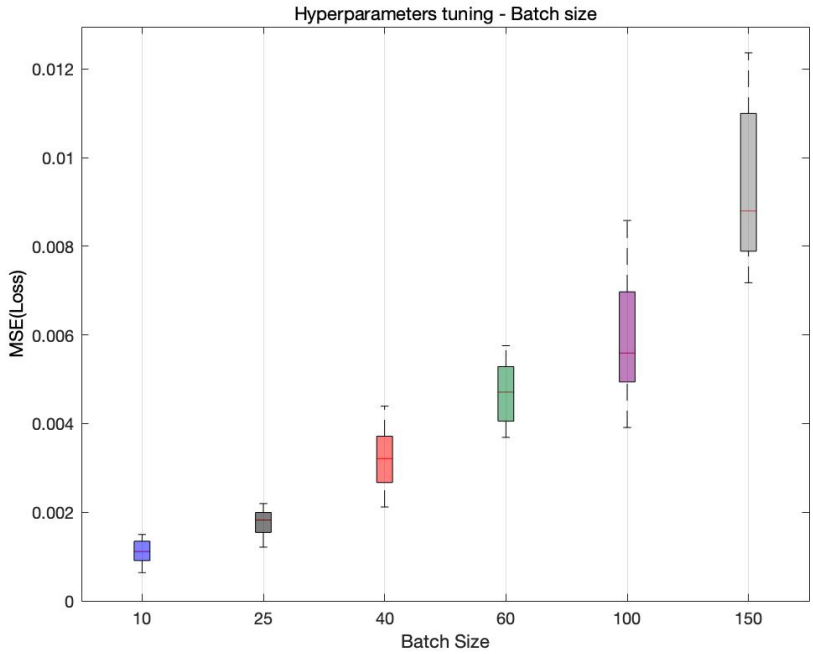


Figure 4.3 Batch size search - Loss as a function of the batch size of the NN.

Looking at the boxplot, the results are the one expected by theory: higher precision corresponds to a smaller batch size. The chosen configuration is both accurate and fast. In fact, the model with a batch size of 40 halves the training time but doubles the loss value compared to the best option.

Finally, the activation function must be determined. The default choice for this type of Neural Network is the ReLU function, as it is simple to implement and works very well; however, it is compared to both the logistic (Sigmoid) and the Hyperbolic Tangent function to test whether it is in fact the correct choice or not.

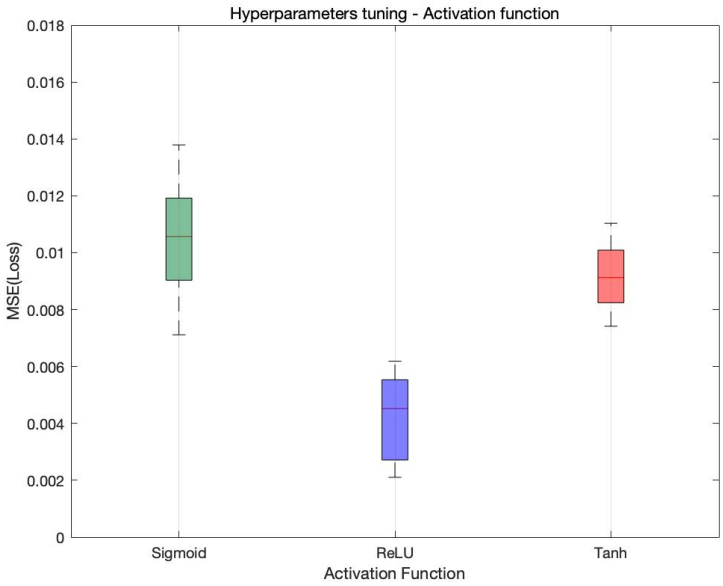


Figure 4.4 Comparison between Activation Functions – Loss as a function of the Activation Functions

As it displayed by the graph, the Rectified Linear Unit is indeed the best Activation function for the model. In the compared cases, the error is amplified by the number of iterations and nodes of the defined Neural Network; probably, with a lower number of layers the other functions would behave in a similar way with a better accuracy.
Note that all hidden layers use the transfer function chosen in the previous paragraph.

The following table summarizes the characteristics of the Feed Forward Neural Network after the tuning:

Table 4.1 – Hyperparameters value of FFNN

Number of Epochs	600
Number of Hidden Layers	5
Architecture (Number of nodes for each layer)	20-40-60-80-100
Batch Size	40
Activation Function	ReLU (Rectified Linear Unit)

4.1.2 Reynolds number and time influence

The following section shows the results of the NN's prediction compared with the actual Direct Numerical Simulation (DNS). As previously displayed, the values of the velocity components (U_x *and* U_y) used to train the NN, and consequently to compare the analyses, are at $x/h = 0.15$, which means the intersect stands at 1 cm from the wall-step.

4.1.2.1 Reynolds number

First, the effect of the Reynolds number is shown. As all simulations were run at a Reynold number higher than $Re = 5000$, the test cases are turbulent solutions. Three different analyses with various Reynolds numbers are taken into consideration. Later, as a bigger variety of solutions is considered, the results are shown into a plot in the form of error value.

Evidently, the cases that are considered must not be in the initial dataset. Note that the time instant doesn't vary; all solutions consider $t = 1.5\ s$. Moreover, as the grid cells are 128 in the y-direction, there will be 128 number of points in the graph.

The chosen Reynolds number are:

- $Re = 7980$ (Figure 4.5, 4.6)
- $Re = 22130$ (Figure 4.7, 4.8)
- $Re = 38420$ (Figure 4.9, 4.10)

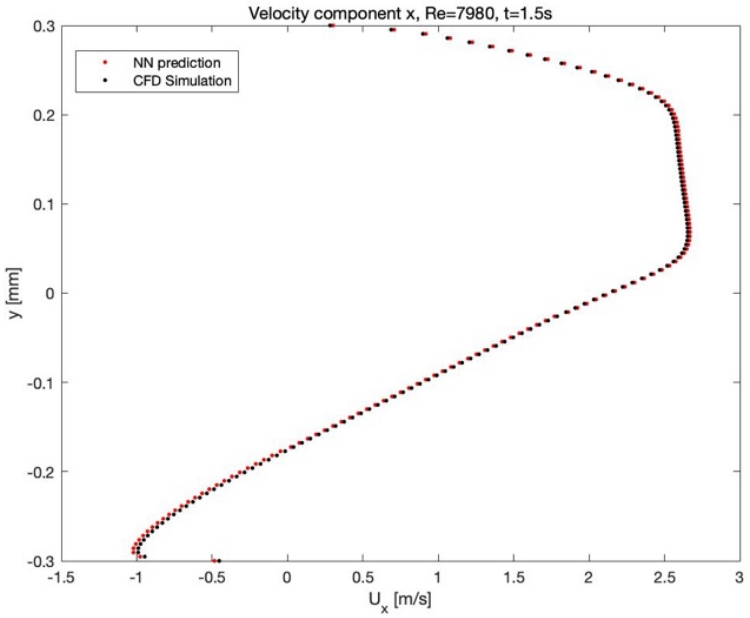


Figure 4.5 - Velocity component x (U_x) for $Re = 7980$, $t = 1.5s$.

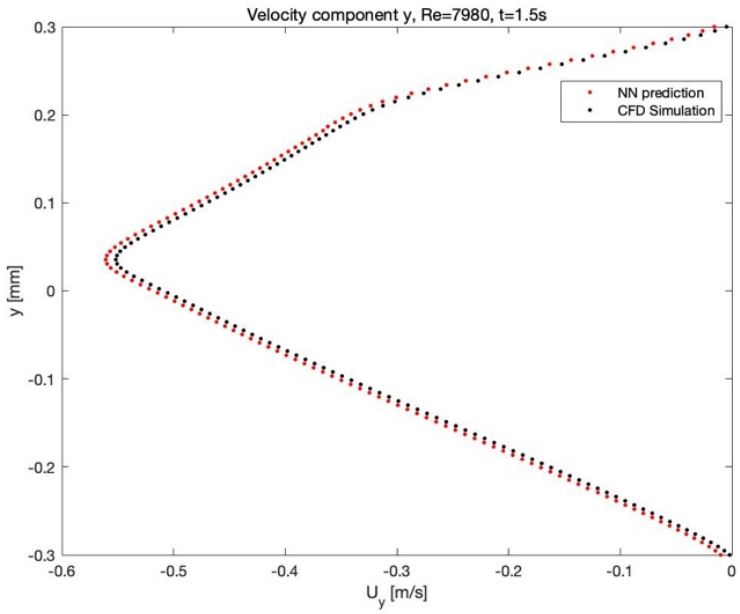


Figure 4.6 – Velocity component y (U_y) for $Re = 7980$, $t = 1.5s$.

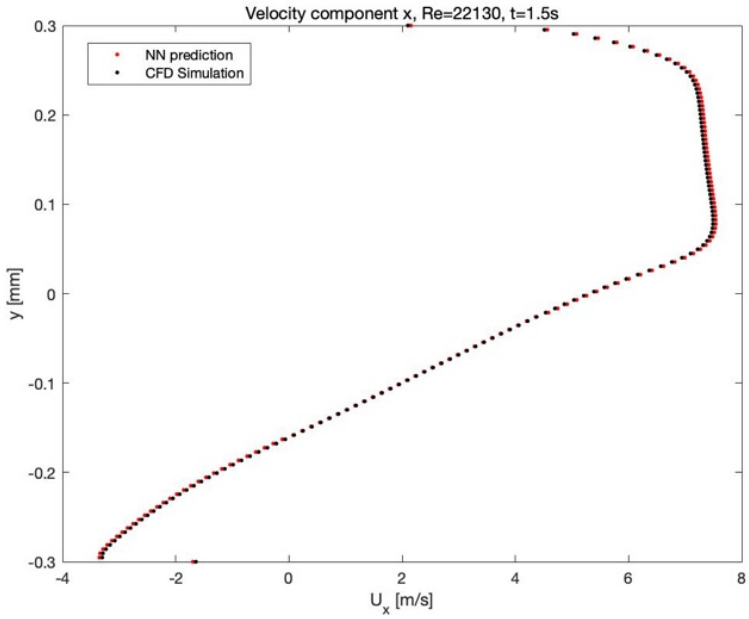


Figure 4.7 - Velocity component x (U_x) for $Re = 22130$, $t = 1.5s$

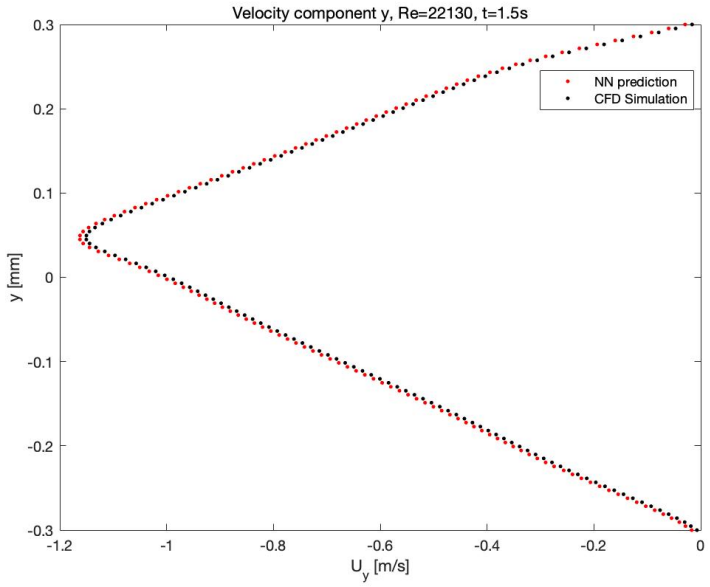


Figure 4.8 – Velocity component y (U_y) for $Re = 22130$, $t = 1.5s$

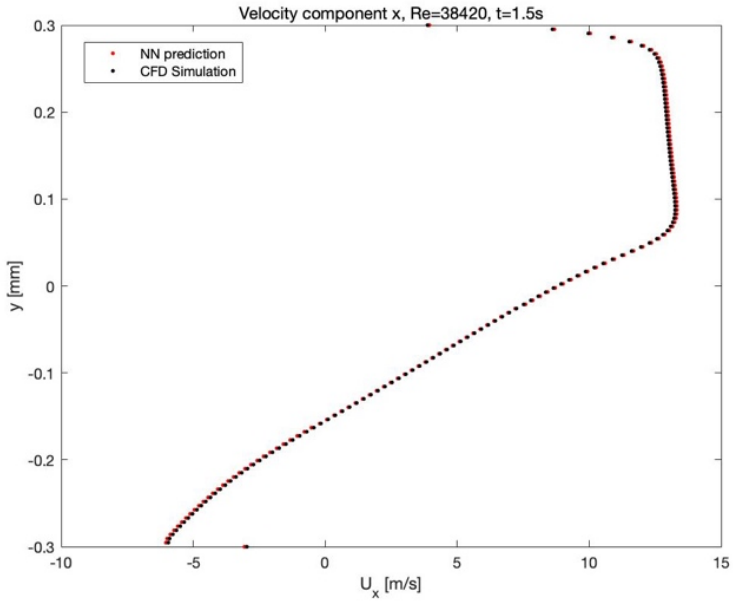


Figure 4.9 – Velocity component x (U_x) for $Re = 38420$, $t = 1.5s$

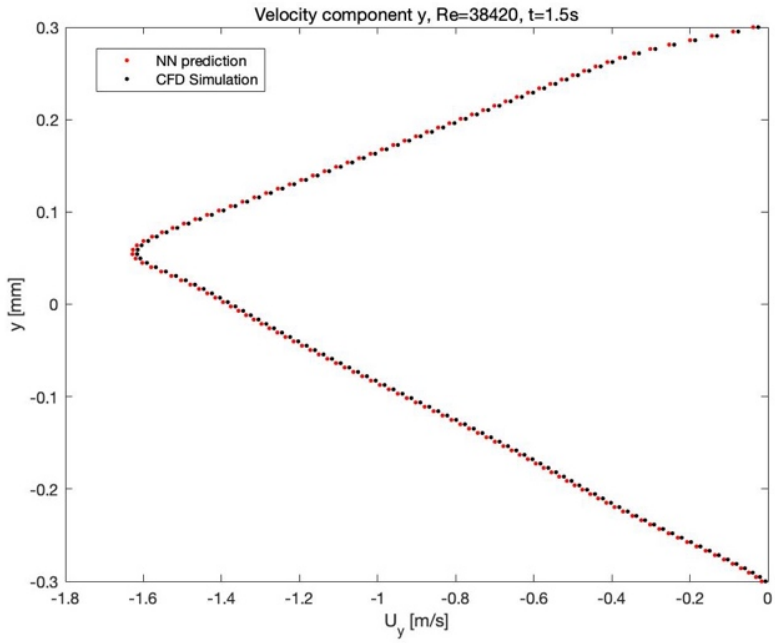


Figure 4.10 – Velocity component y (U_y) for $Re = 38420$, $t = 1.5s$

Generally, in all three cases, the prediction tends to overestimate the solution. In the graphs representing the U_y component, the prediction is always a smaller value, which means it is 'more negative' than the DNS. This is even clearer on the U_x component profile, as there's even a change of sign of the velocity (see figure 4.5, 4.7 and 4.9). The NN's prediction tends to be lower than the CFD's solution values before 0, while it is higher when U_x is positive.

Here, the three test-cases are displayed in the same graph:

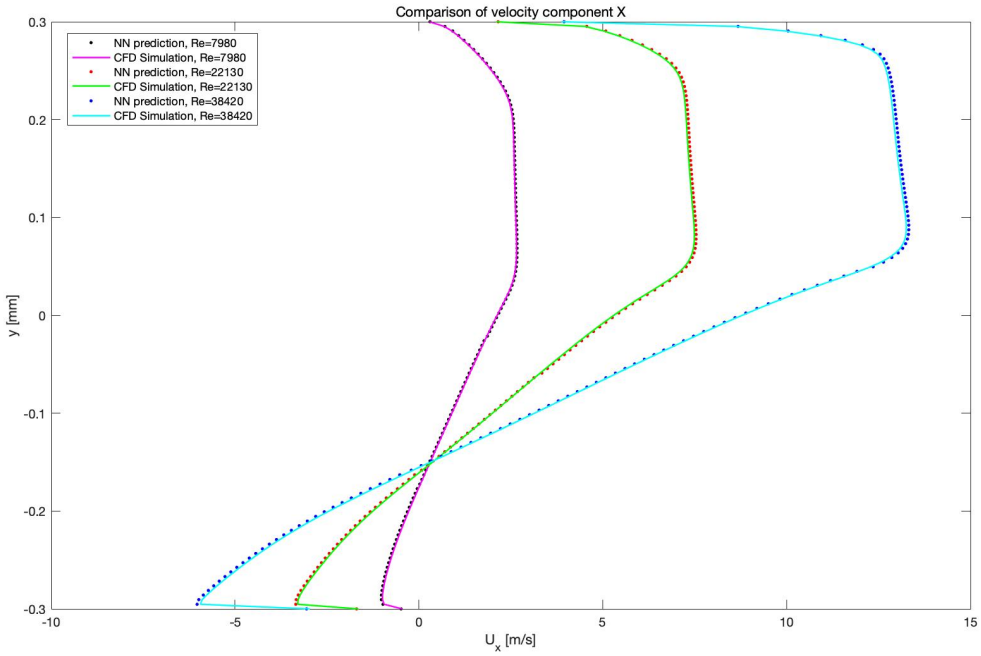


Figure 4.11 – Comparison of x-velocity component U_x for different Reynolds number

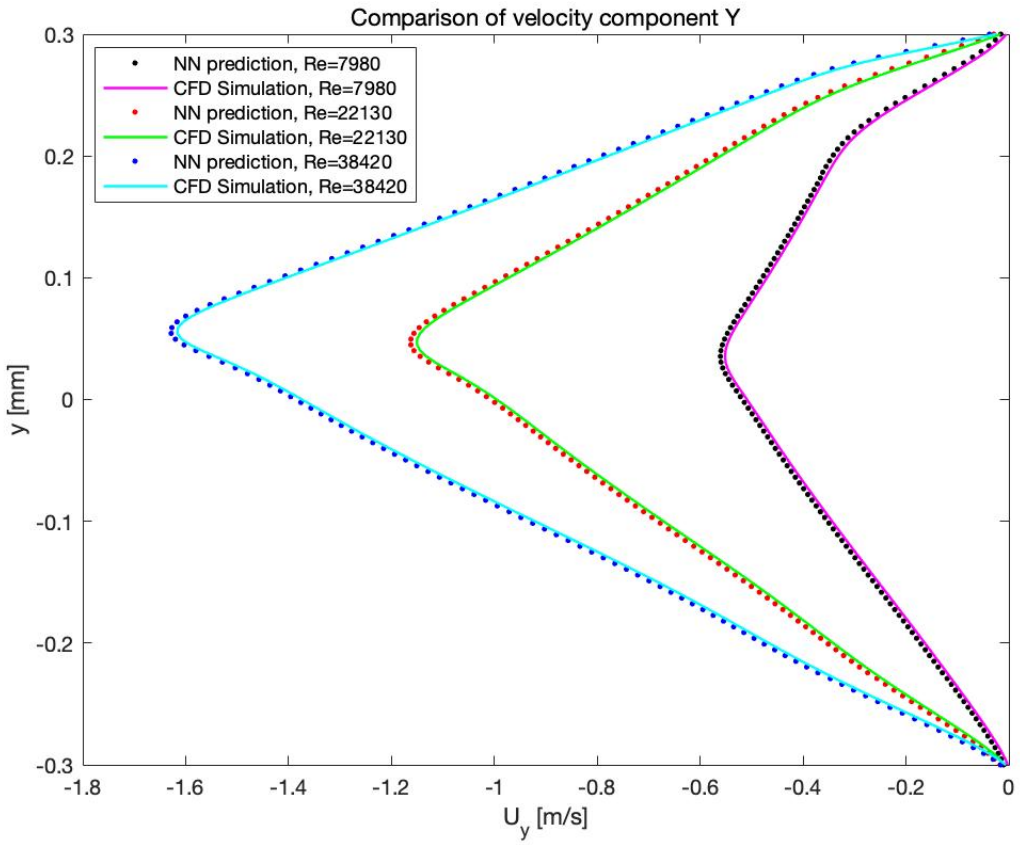


Figure 4.12 – Comparison of y-velocity component U_y for different Reynolds number

The first impression is that for a higher Reynolds number, the prediction moves further from the numerical solution. However, as the velocity for a more turbulent flow is greater, what seems to be a bigger error might not be. To effectively display the numerical loss of different solutions as a function of the Reynolds number, 10 cases are considered.

The error is calculated with the following formulas for each different configuration:

$$\varepsilon_x = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{U_{x_{i_{DNS}}} - U_{x_{i_{Prediction}}}}{U_{x_{i_{DNS}}}}\right) \; with \; N = 128$$

$$\varepsilon_y = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{U_{y_{i_{DNS}}} - U_{y_{i_{Prediction}}}}{U_{y_{i_{DNS}}}}\right) \; with \; N = 128$$

Table 5.2 displays the results:

Table 4.2 – Value of ε_x and ε_y for each Reynolds number

Reynolds number	ε_x	ε_y
7980	0.00567	0.00263
11760	0.00622	0.00291
14540	0.00549	0.00275
18660	0.00504	0.00264
22130	0.00496	0.00252
26930	0.00475	0.00246
30090	0.00464	0.00239
34719	0.00469	0.00244
38420	0.00473	0.00251
43280	0.00481	0.00253

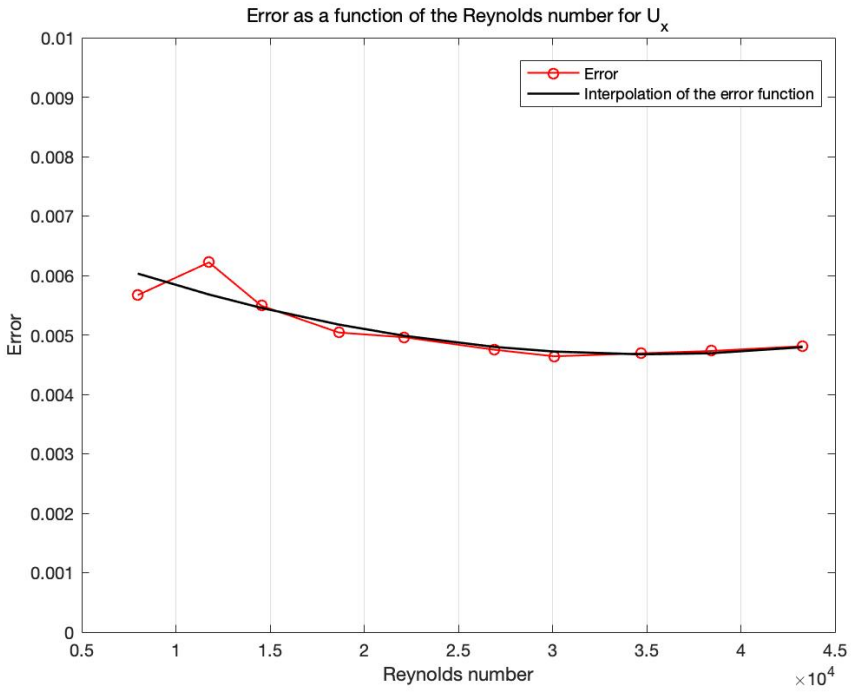


Figure 4.13 – Error of U_x as a function of the Reynolds number.

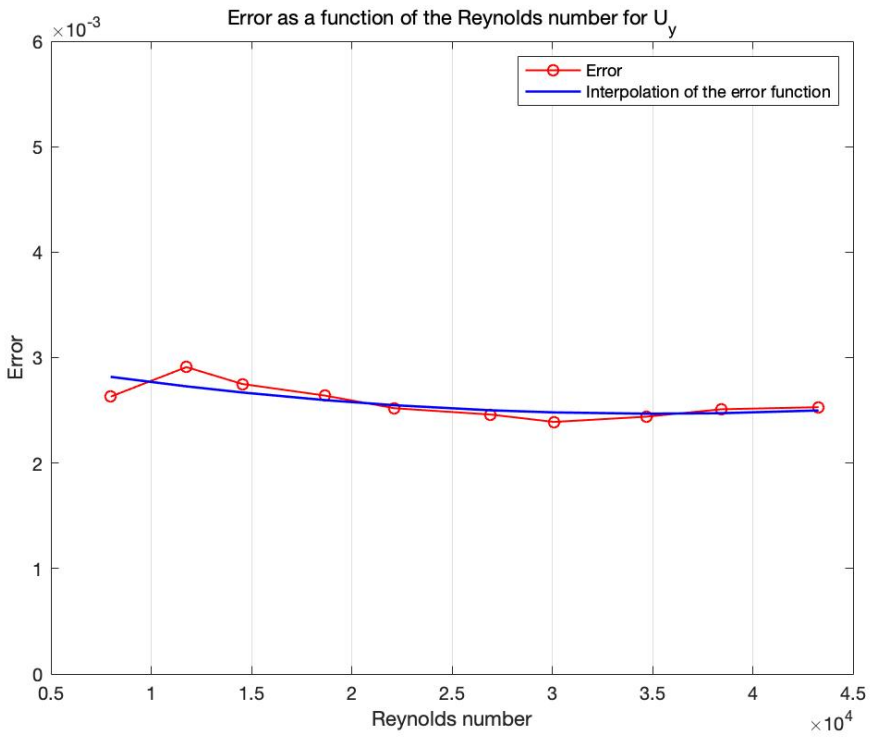


Figure 4.14 – Error of U_y as a function of the Reynolds number.

Both graphs have a minimum for Reynolds number of approximately 30000. Nonetheless, the error function is stable as the Re increases, as might have been expected being that the graphic solution simplifies for more turbulent flows, as the solution doesn't change much once it reaches a higher Reynolds number. This enables to the NN to be more accurate with its solution.

4.1.2.2 Time

In this section, the influence of time is shown.

By the previous analysis, the best-case scenario is a fully turbulent flow. Consequently, time is tested on the configuration with $Re = 38420$. Three time-inputs are taken into account to show graphics results; later, as in the precedent study, multiple solutions are considered in order to establish the error function.

Time instants are:

- $t = 1\ s$

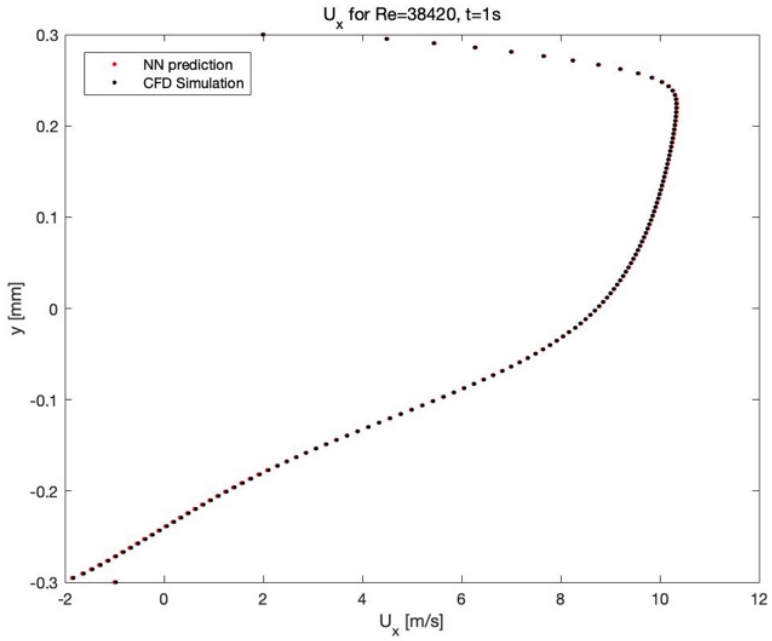


Figure 4.15 - Velocity component x (U_x) for $Re = 38420$, $t = 1s$

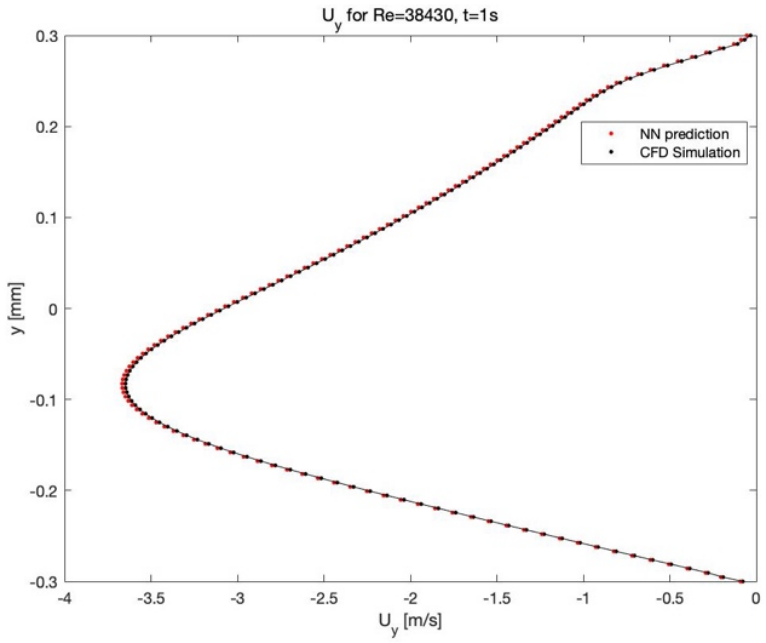


Figure 4.16 - Velocity component y (U_y) for $Re = 38420$, $t = 1s$

- $t = 3\ s$

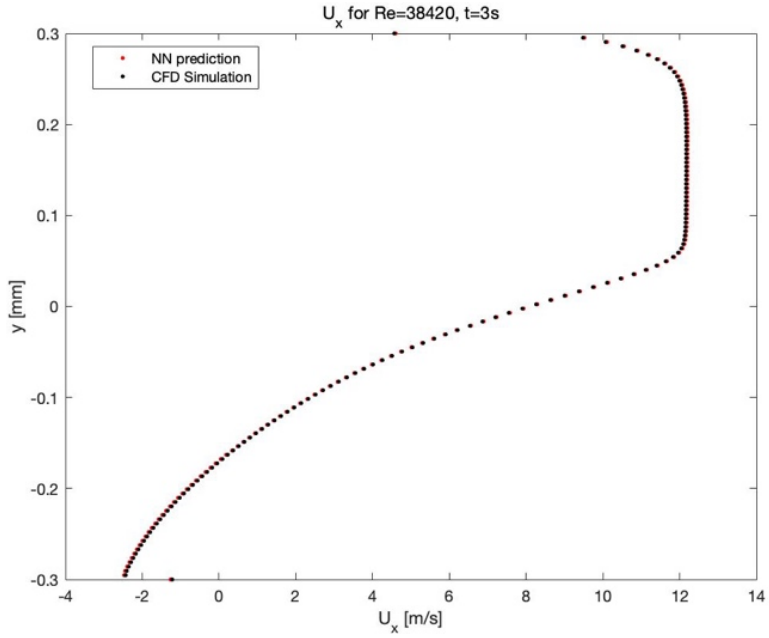


Figure 4.17 - Velocity component x (U_x) for $Re = 38420$, $t = 3s$

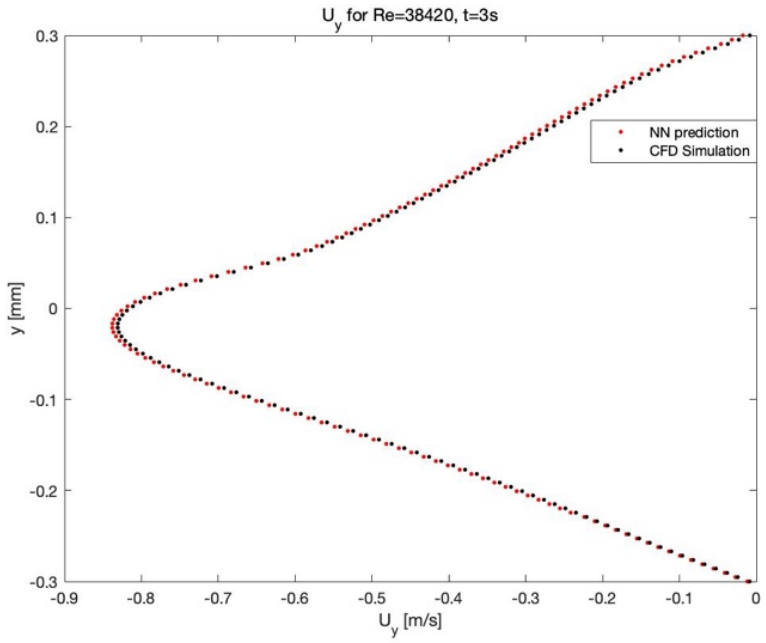


Figure 4.18 - Velocity component y (U_y) for $Re = 38420$, $t = 3s$

- $t = 5\ s$

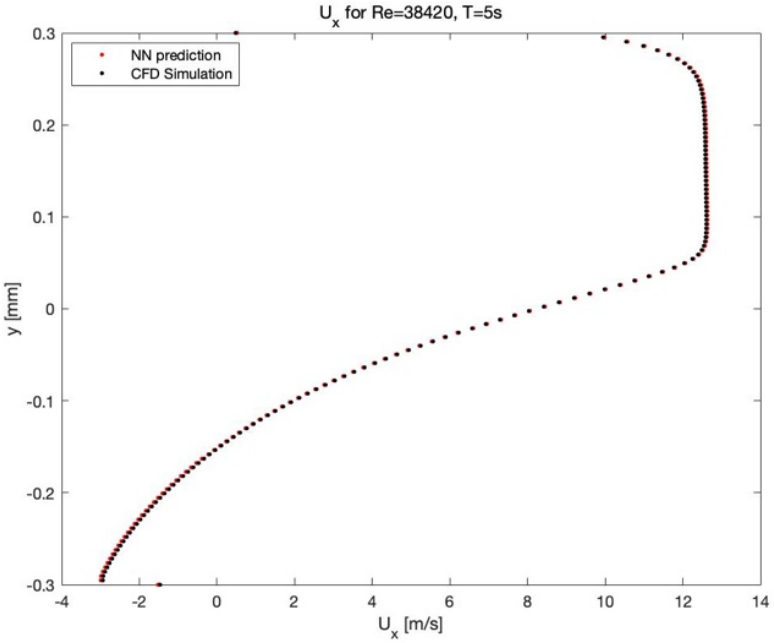


Figure 4.19 - Velocity component x (U_x) for $Re = 38420$, $t = 5s$

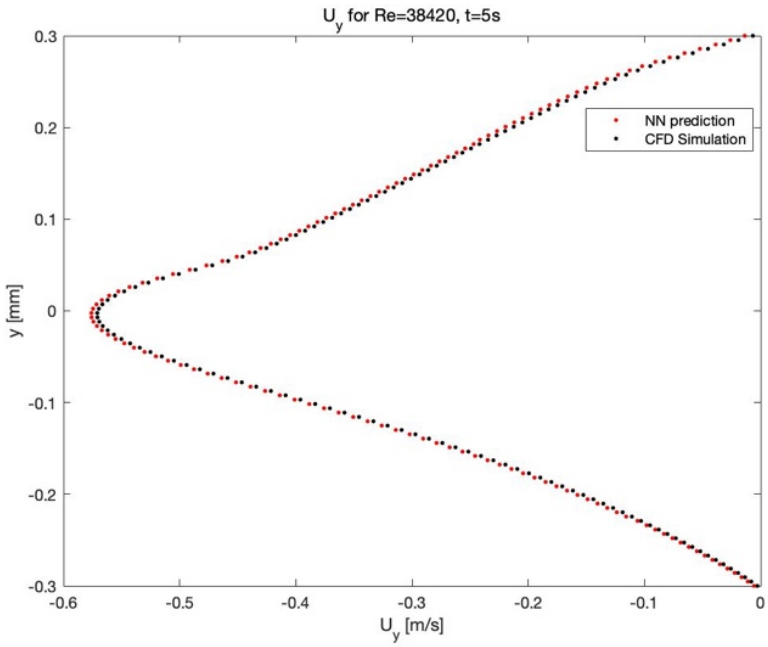


Figure 4.20 - Velocity component y (U_y) for $Re = 38420$, $t = 5s$

In all configurations, in the graphs representing the U_y component, the prediction seems to be less precise than the one of U_x. In addition, the trend that characterizes the Reynolds number's discussion hasn't changed. The NN's prediction tends to be lower than the DNS's values before 0, while it is higher when U_x is positive.

Here, it follows the graph with the 3 solutions for both U_x and U_y, compared with the real case studied with CFD. In order, from top to bottom, the simulations represent 1s, 3s and 5s.

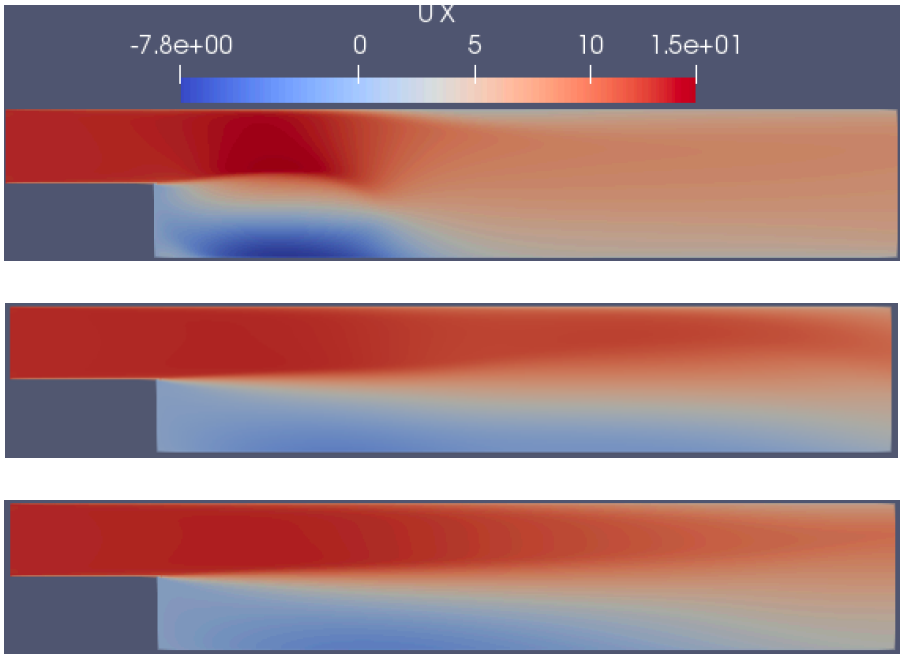


Figure 4.21 - Comparison of x-velocity component U_x for different time values – CFD representation.

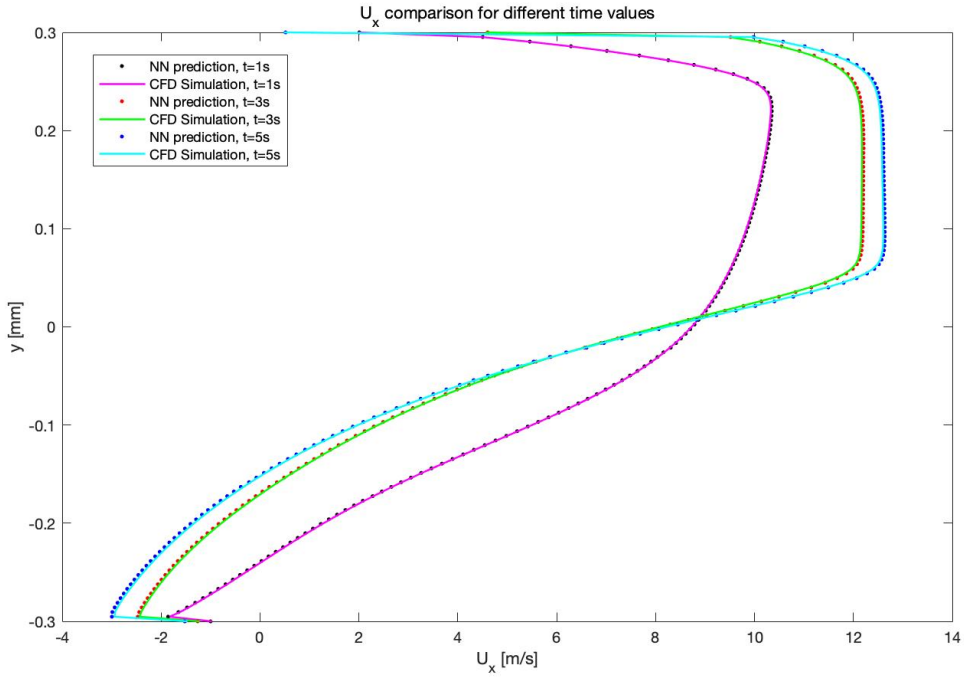


Figure 4.22 – Comparison of x-velocity component U_x for different time values at $Re = 38420$.

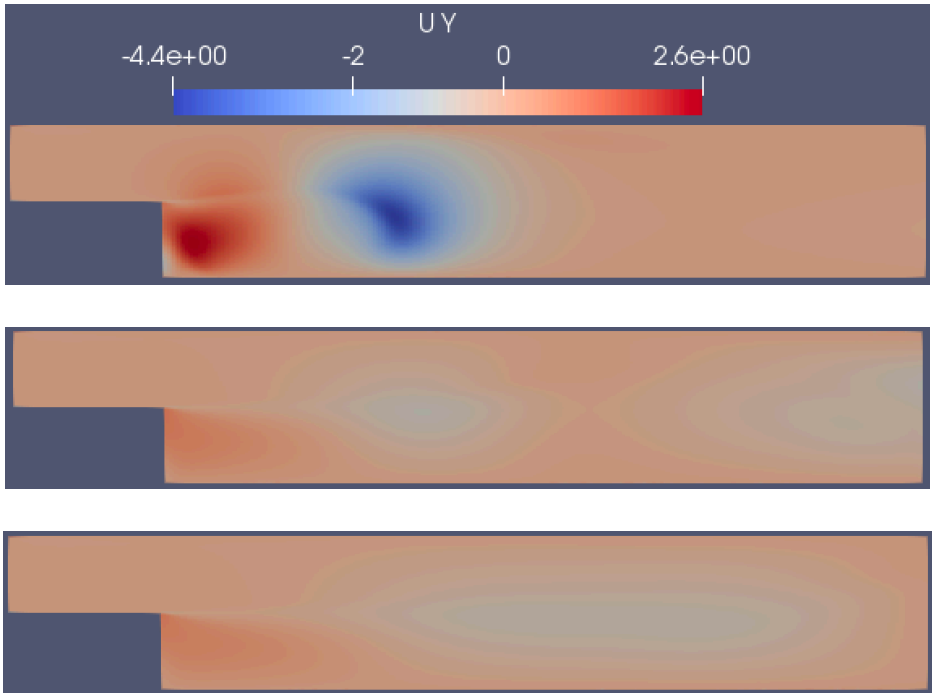


Figure 4.23 - Comparison of y-velocity component U_y for different time values – CFD representation.

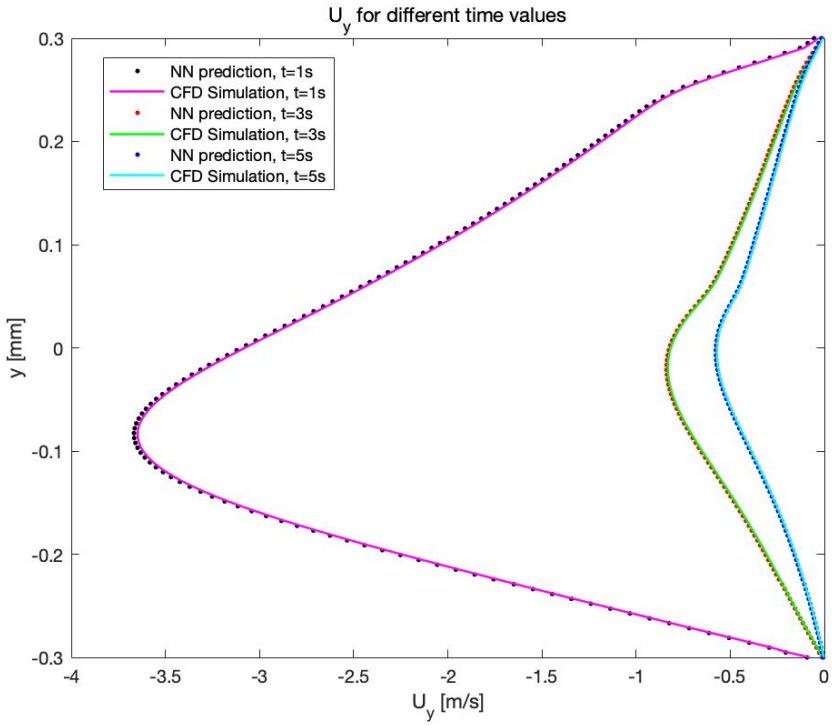


Figure 4.24 – Comparison of y-velocity component U_y for different time values at $Re = 38420$.

By looking at the last two graphs, it seems like there aren't any differences from one time instant to the other in terms of prediction precision. Especially, in the U_x figure, the predictions seem to retrace the velocity profile of the CFD's solution. However, in order to complete the analysis accurately, the error function is graphed as in the previous case (figure 4.25 and figure 4.26); in this model, the independent variable is time.

Using the same precedent formulas, the data is collected in Table 4.3:

Table 4.3 – Value of ε_x and ε_y for each time instant

Time [s]	ε_x	ε_y
0.5	$2.35 * 10^{-4}$	$3.73 * 10^{-4}$
1	$2.29 * 10^{-4}$	$3.68 * 10^{-4}$
1.5	$2.33 * 10^{-4}$	$3.65 * 10^{-4}$
2	$2.31 * 10^{-4}$	$3.66 * 10^{-4}$
2.5	$2.28 * 10^{-4}$	$3.63 * 10^{-4}$
3	$2.30 * 10^{-4}$	$3.61 * 10^{-4}$
3.5	$2.23 * 10^{-4}$	$3.64 * 10^{-4}$
4	$2.27 * 10^{-4}$	$3.63 * 10^{-4}$
4.5	$2.29 * 10^{-4}$	$3.62 * 10^{-4}$
5	$2.30 * 10^{-4}$	$3.64 * 10^{-4}$

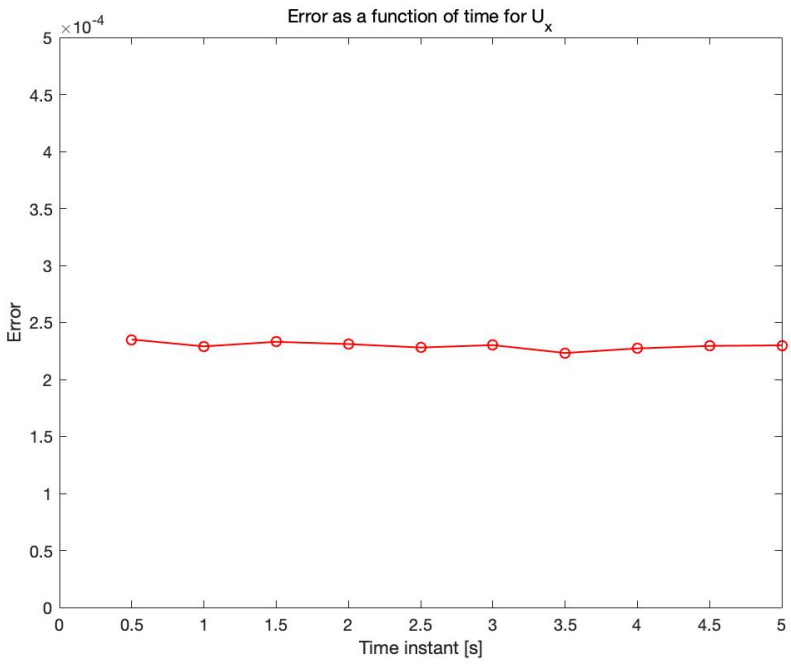


Figure 4.25 – Error of U_x as a function of time of simulation.

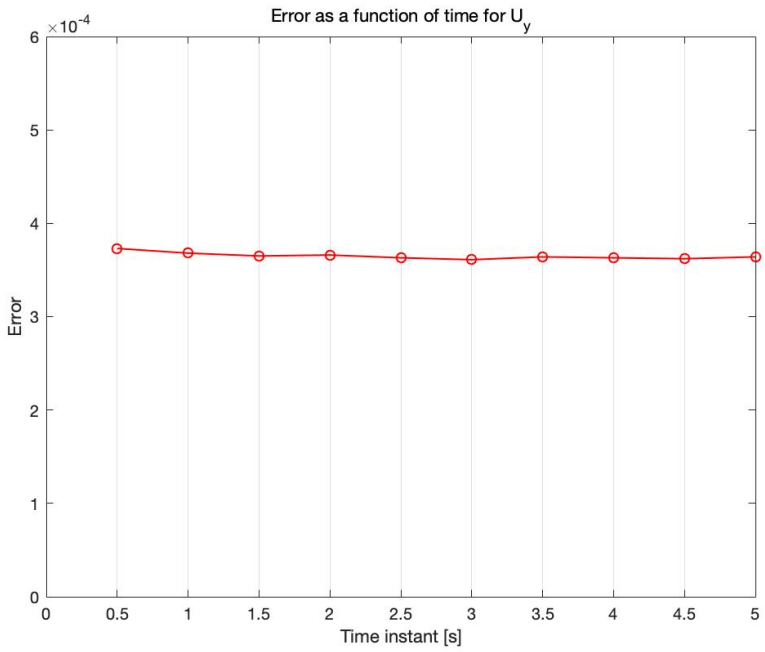


Figure 4.26 – Error of U_y as a function of time of simulation.

This result shows that the error is very low time dependent.

4.2 Recurrent Neural Network

This section concerns the use of the more advanced ANN, defined and described in the previous chapters. The results obtained using the Recurrent Neural Network are compared with the outcomes of the last two sections. This model, whose outputs are influenced by both the inputs and the outputs of different configurations, should be more precise as it reconstructs the solution step-by-step.

As already explained, a RNN is very hard to optimize and to train, consequently some hyperparameters were chosen based on the knowledge of previous research. However, both the number of epochs and the number of nodes for the hidden layer needs to be searched.

Regarding the number of epochs, here follows the graph of its tuning:

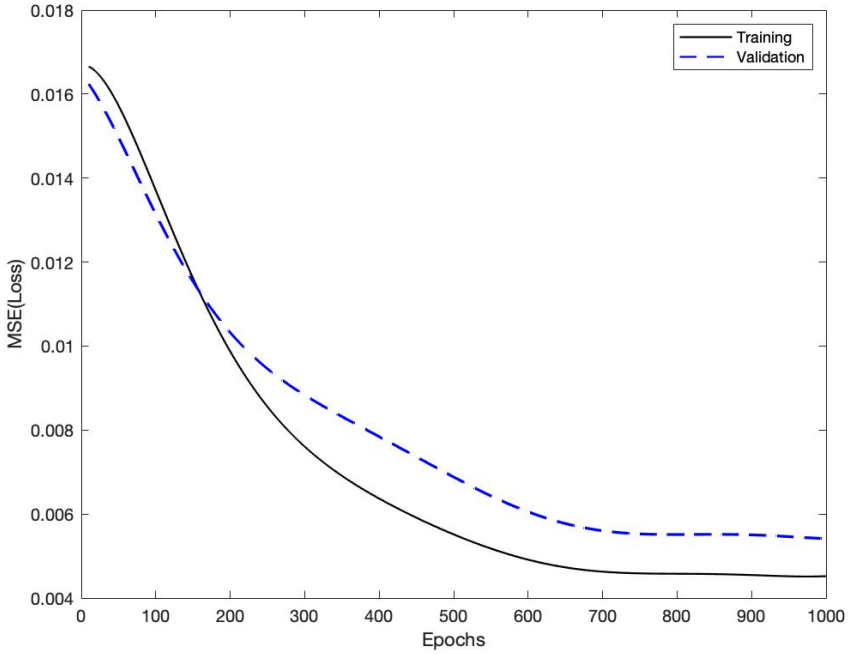


Figure 4.27 - The training and validation loss as a function of training epochs for a RNN.

Consequence is that the chosen number of epochs is 650, which allows both sufficient accuracy and considerable time saving. It is important to note that early-stopping was implemented as a regularization during training so to reduce the number of iterations if the solution is acceptable. The batch size is taken as 32, the dense layer is characterized by 5 units and the dropout value is 30% (to reduce the time of the training).

Now, the number of neurons in the hidden layer must be defined. With the given hyperparameters, tests were run for four different architectures, following the same logic of the Feed Forward NN tuning; the aim is to find the minimum for the loss function.

The results are shown with a boxplot.

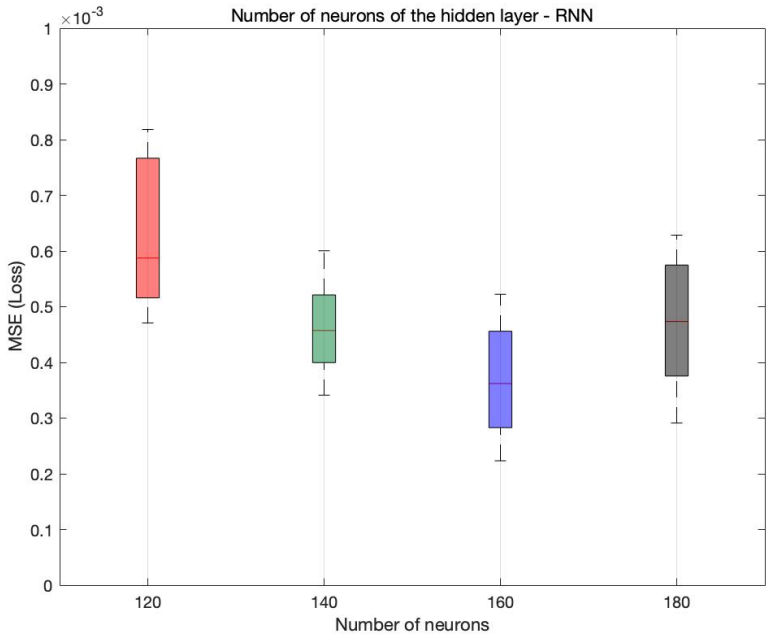


Figure 4.28 – Comparison between architectures – Loss as a function of the number of neurons.

The figure shows the optimal configuration is with 160 neurons. However, due to the time complexity of the model, and given that the loss function is very low for all solutions, the chosen architecture is the one with 140 units.

To summarize, the hyperparameters of the Recurrent Neural Network are shown in Table 4.4.

Table 4.4 – Hyperparameters value of RNN

Number of Epochs	650
Number of Hidden Layers	1
Architecture (Number of nodes for the layer)	140
Batch Size	30
Activation Function	Sigmoid Function
Dropout value	30%

In order to analyze the quality of the RNN, the same previously-run-tests should be performed and the results compared with the earlier ones. As a visual comparison isn't sufficient due to the proximity of the data, graphs figuring the error as a function of the Reynolds number and time instant will be provided; the same inputs will be used to reproduce them.

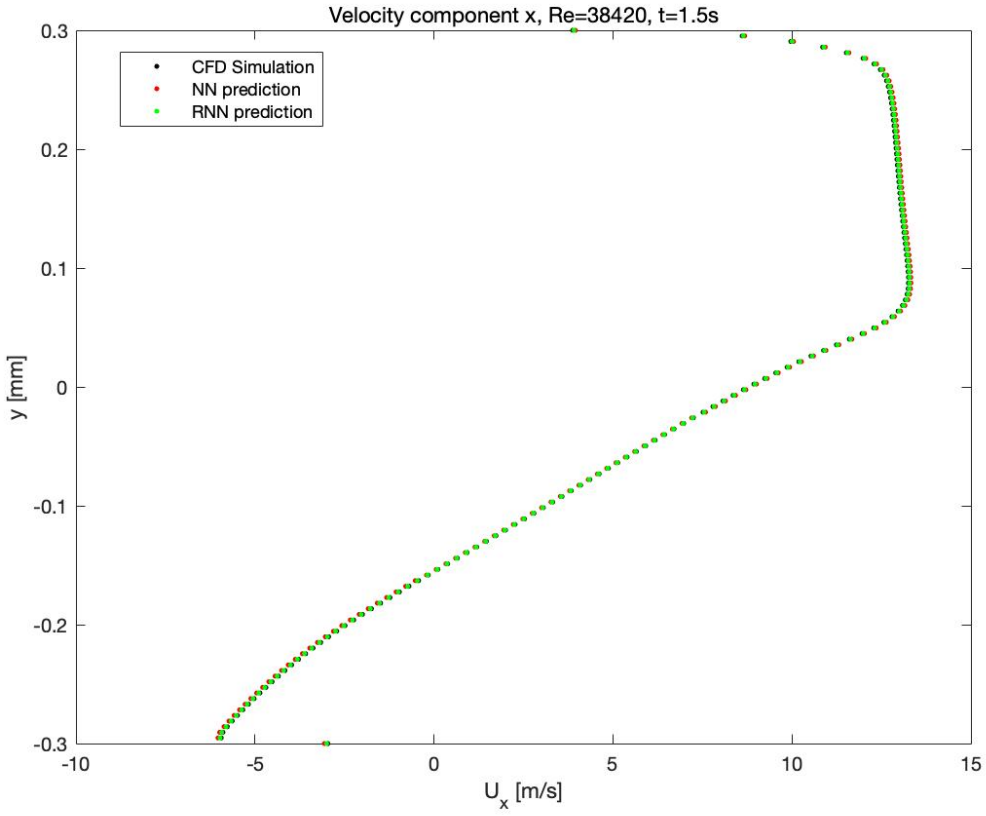


Figure 4.29 - Velocity component x (U_x) for $Re = 38420$, $t = 1.5s$ – Comparison between FFNN and RNN

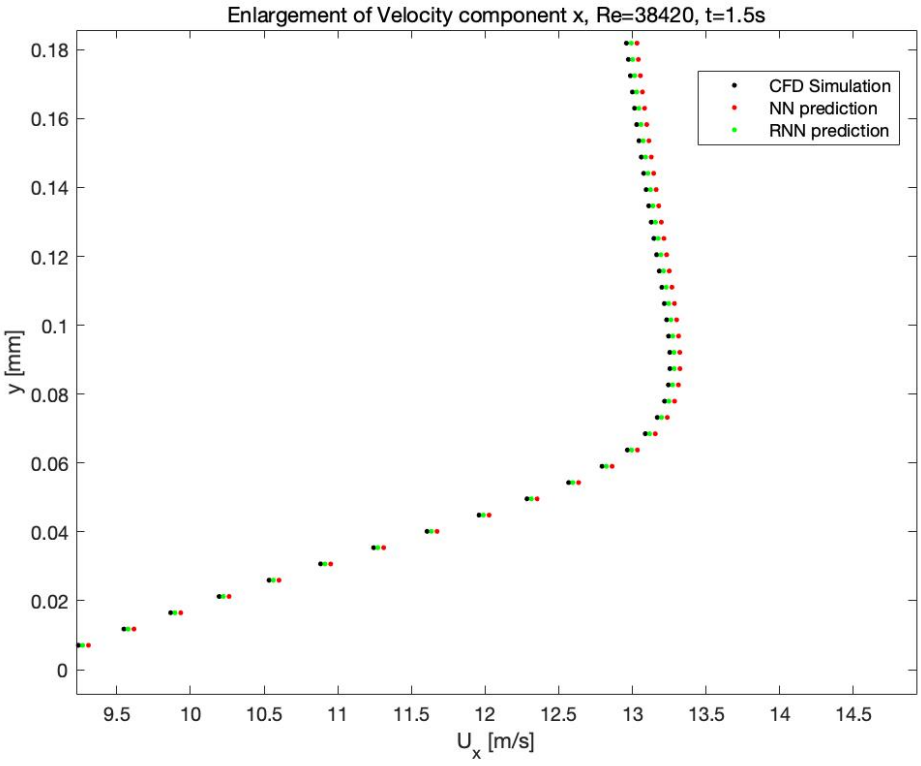


Figure 4.30 – Enlargement of (U_x) for $Re = 38420$, $t = 1.5s$ – Comparison between FFNN and RNN

As expected, the prediction of the Recurrent Neural Network is closer to the DNS (figure 4.29 and 4.30).

For the velocity component (U_y):

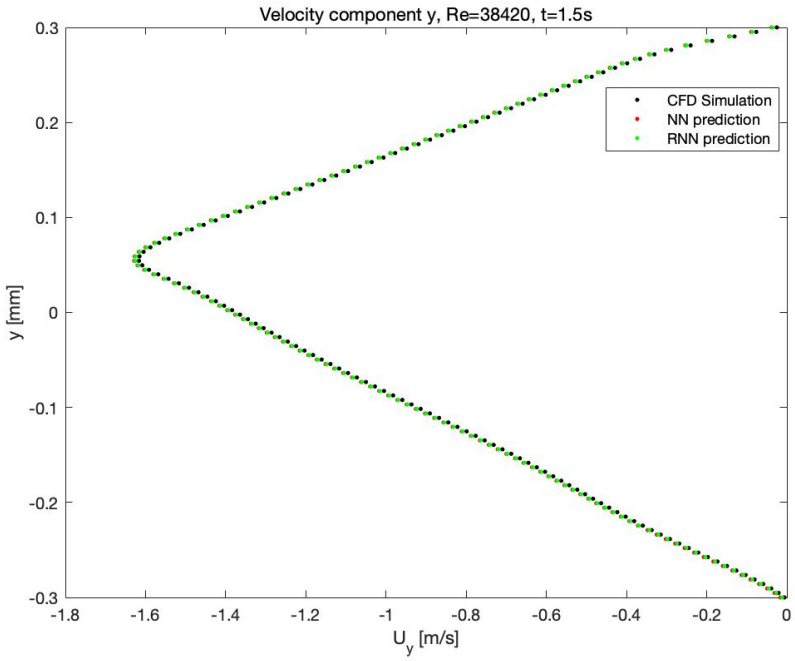


Figure 4.31 - Velocity component y (U_y) for $Re = 38420$, $t = 1.5s$ – Comparison between FFNN and RNN

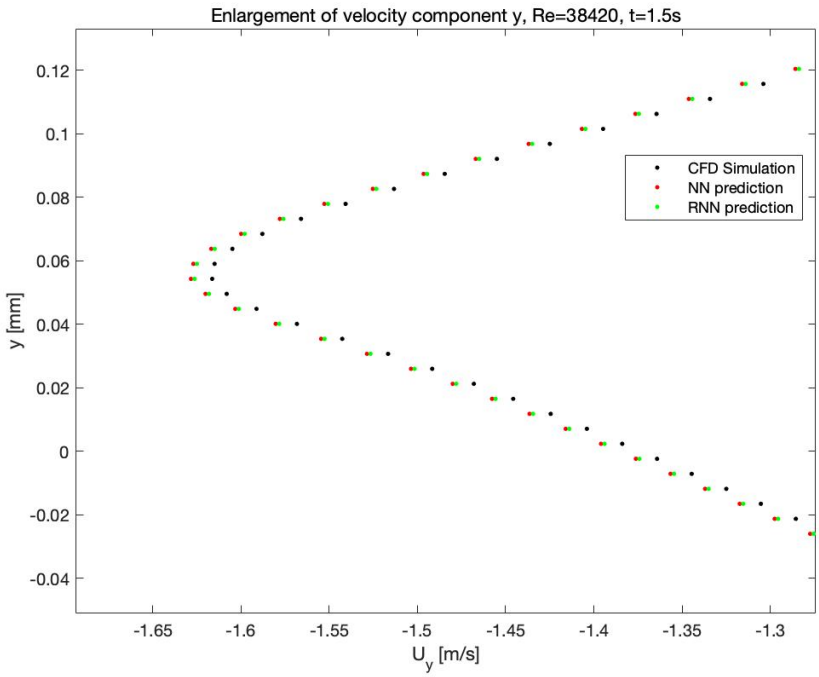


Figure 4.32 – Enlargement of (U_y) for $Re = 38420$, $t = 1.5s$ – Comparison between FFNN and RNN

The same conclusion can be drawn looking at the results for the U_y component (see figure 4.31 and 4.32): the RNN's prediction is slightly closer to the CFD solution than the FFNN's (*NN* in the graphs above).

Now, being the RNN ideal for sequential data, it is interesting to see its prediction when the independent variable is the time. Therefore, after showing the results for a varying Reynolds number (figure 4.29, 4.30, 4.31 and 4.32), now the time instant becomes the changing factor while the Reynolds number is considered a consant.

For the horizontal velocity:

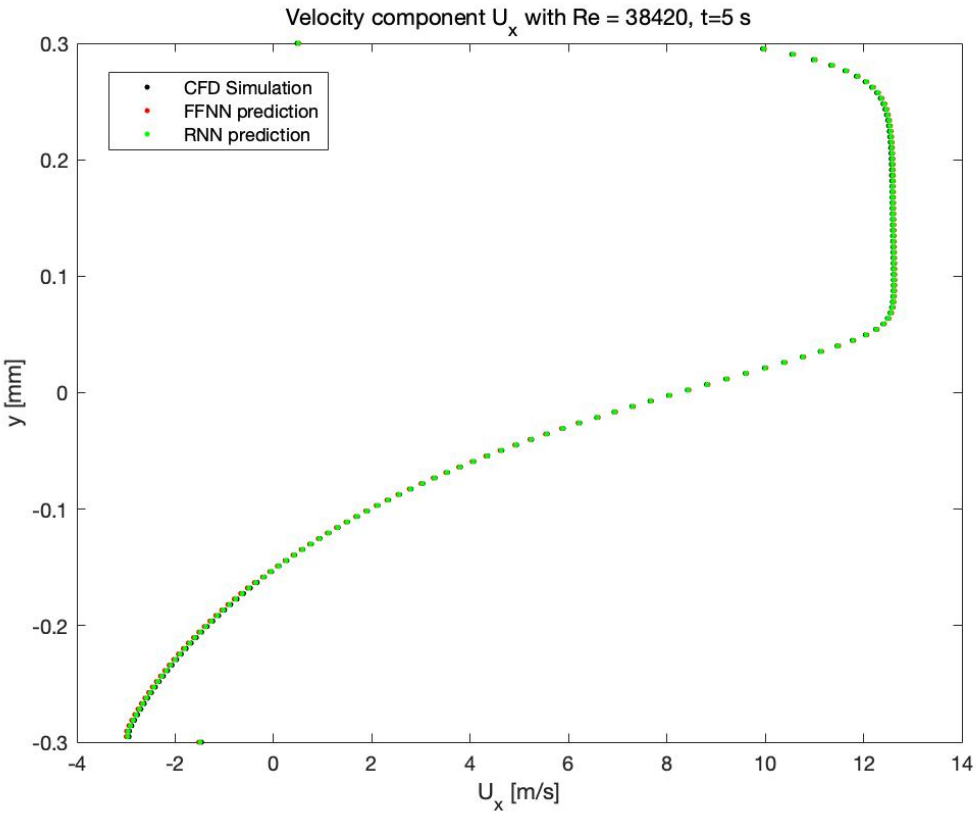


Figure 4.33 - Velocity component x (U_x) for $Re = 38420$, $t = 5s$ – Comparison between FFNN and RNN

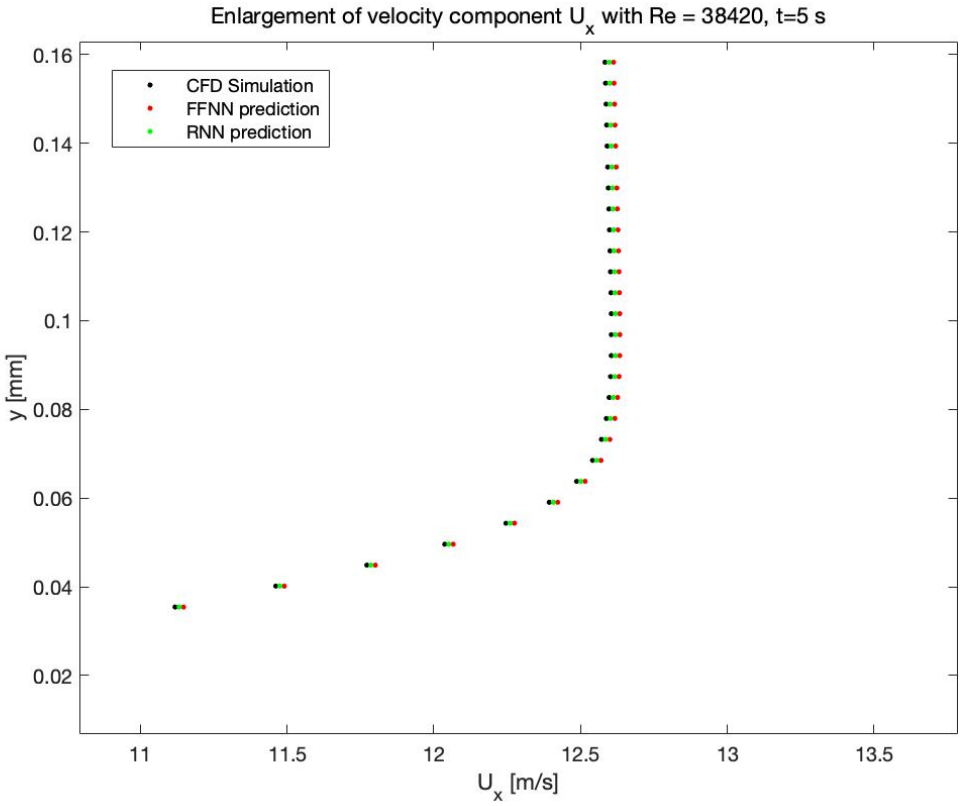


Figure 4.34 – Enlargement of (U_x) for $Re = 38420$, $t = 5s$ – Comparison between FFNN and RNN

While for the vertical velocity:

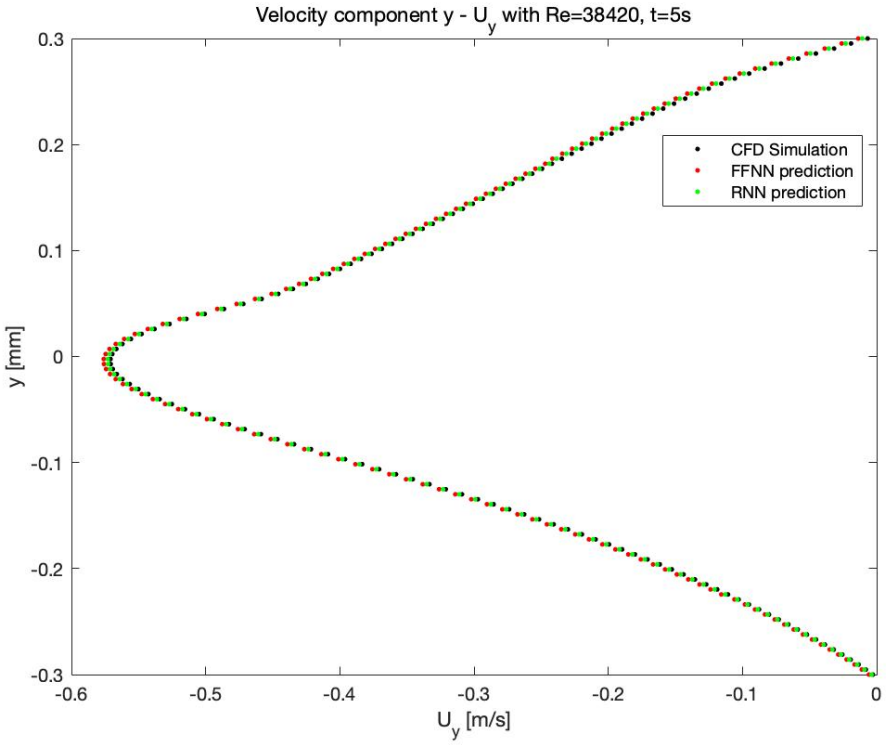


Figure 4.35 - Velocity component y (U_y) for $Re = 38420$, $t = 5s$ – Comparison between FFNN and RNN

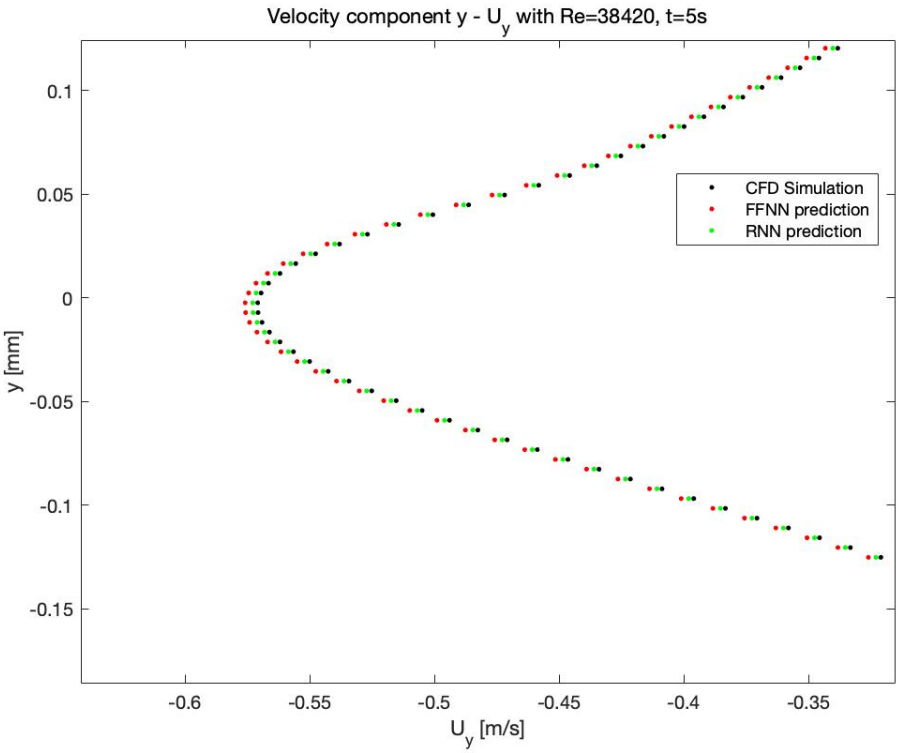


Figure 4.36 – Enlargement of (U_y) for $Re = 38420$, $t = 5s$ – Comparison between FFNN and RNN

As expected, the prediction of the Recurrent Neural Network is very close to the CFD simulation's result. This is due to the sequential characteristic of the model; in fact, as it reconstructs the predicted solution, the result at $t = 5s$ is deducted by studying all outputs from $t = 0s$ to $t = 4.9s$, giving the Network the opportunity to learn the time influence more correctly.
To ensure that the new model has a better accuracy, the error-dependency of the configuration is calculated for the same values of Reynolds number and time. The results are shown in table 5.5 and 5.6:

Table 4.5 – Value of ε_x and ε_y for each Reynolds number

Reynolds number	ε_x	ε_y
7980	$2.17* 10^{-3}$	$1.29* 10^{-3}$
11760	$2.31* 10^{-3}$	$1.38* 10^{-3}$
14540	$2.13* 10^{-3}$	$1.33* 10^{-3}$
18660	$2.11* 10^{-3}$	$1.25* 10^{-3}$
22130	$2.02* 10^{-3}$	$1.23* 10^{-3}$
26930	$1.97* 10^{-3}$	$1.19* 10^{-3}$
30090	$1.99* 10^{-3}$	$1.21* 10^{-3}$
34719	$2.04* 10^{-3}$	$1.22* 10^{-3}$
38420	$2.02* 10^{-3}$	$1.18* 10^{-3}$
43280	$2.03* 10^{-3}$	$1.20* 10^{-3}$

Table 4.6 – Value of ε_x and ε_y for each time instant

Time [s]	ε_x	ε_y
0.5	$9.29* 10^{-5}$	$1.12* 10^{-4}$
1	$9.38* 10^{-5}$	$1.08* 10^{-4}$
1.5	$9.33* 10^{-5}$	$1.06* 10^{-4}$
2	$9.16* 10^{-5}$	$1.02* 10^{-4}$
2.5	$9.04* 10^{-5}$	$1.01* 10^{-4}$
3	$9.01* 10^{-5}$	$0.97* 10^{-4}$
3.5	$8.96* 10^{-5}$	$0.95* 10^{-4}$
4	$8.90* 10^{-5}$	$0.94* 10^{-4}$
4.5	$8.87* 10^{-5}$	$0.91* 10^{-4}$
5	$8.81* 10^{-5}$	$0.88* 10^{-4}$

Four graphs display the results compared with the FFNN's architecture.

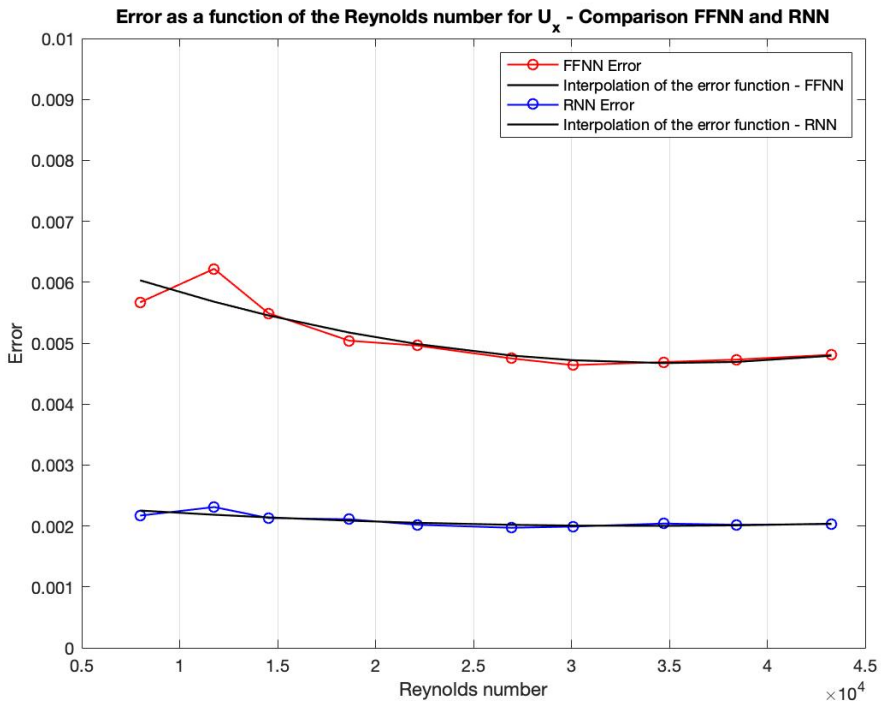


Figure 4.37 – Error of U_x as a function of the Reynolds number – Comparison between Feed Forward NN and Recurrent Neural Network

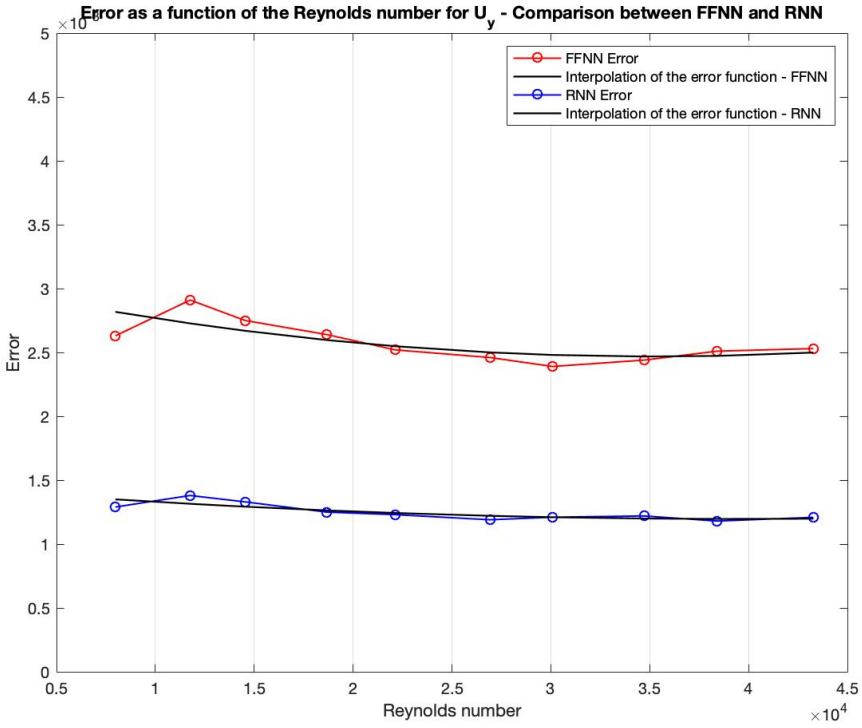


Figure 4.38 – Error of U_y as a function of the Reynolds number – Comparison between Feed Forward NN and Recurrent Neural Network

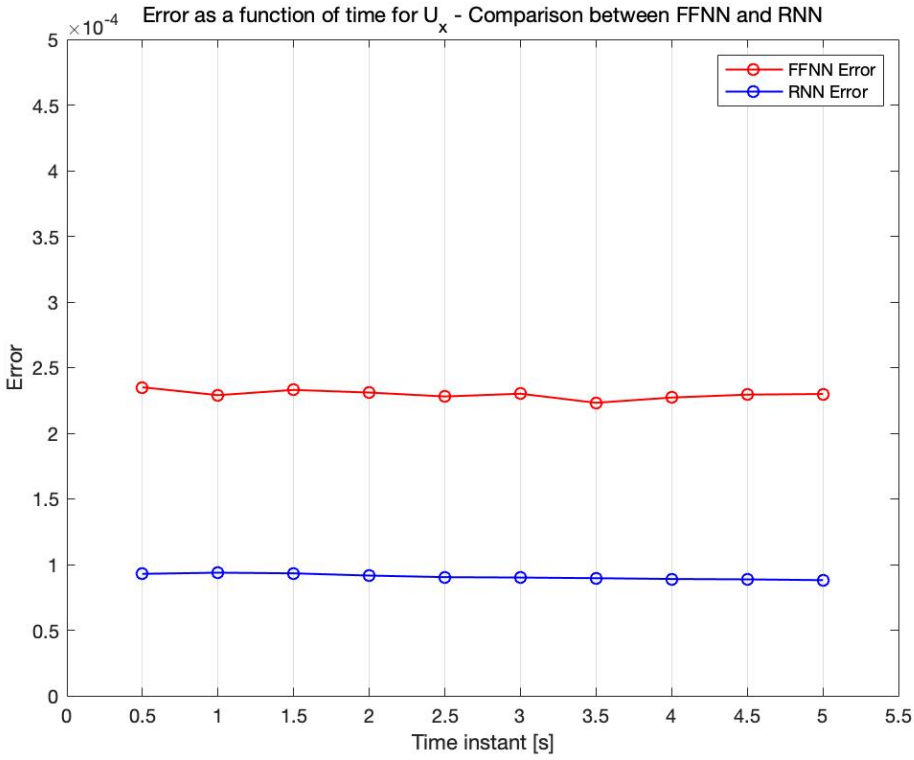


Figure 4.39 – Error of U_x as a function of time of simulation - Comparison between Feed Forward NN and Recurrent Neural Network

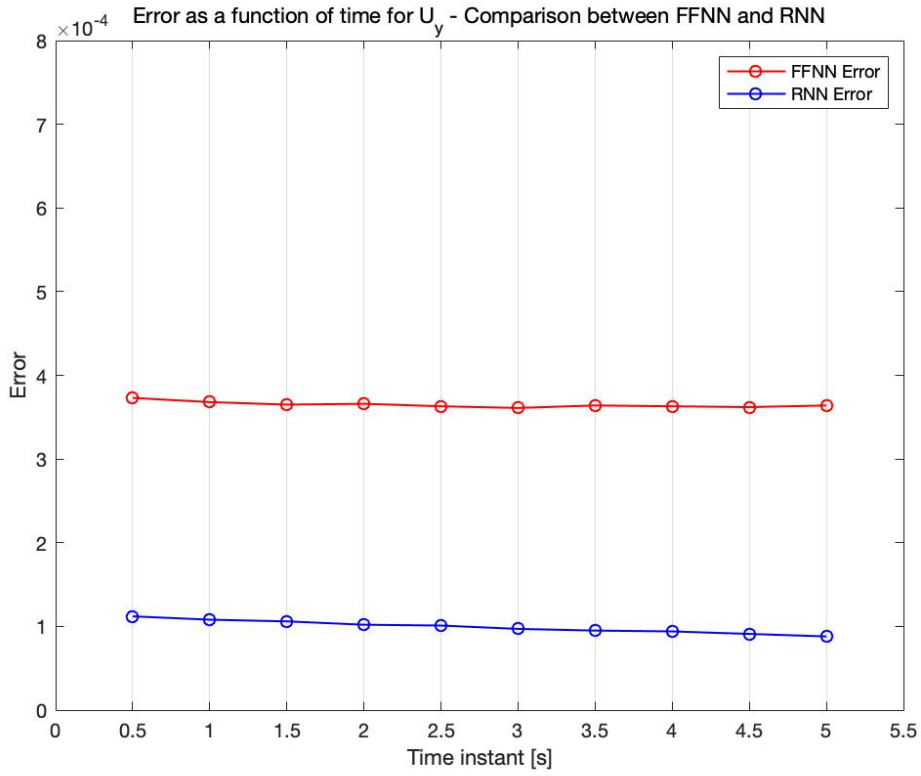


Figure 4.40 – Error of U_y as a function of time of simulation - Comparison between Feed Forward NN and Recurrent Neural Network

As the figures display, the RNN reduces the error in every case.
Note that in both time dependent studies, the value of error keeps diminishing as the time advances. The reason, as already explained, is due to the sequential model in use. Finally, for a more complex Neural Network, there is a visual gain in all predictions; these results come with a bigger computational cost. Once again, it is important to find the 'best' trade-off to reach the solution.

5 Conclusions and possible improvements

According to the study of this thesis both FFNN and RNN were able to predict and reproduce the results with quite high accuracy. The latter can surely be improved by changing some elements.

First, both the FFNN and the RNN have been trained trying to minimize the computational cost, being that the time taken for the definition of weights and biases could be increased by changing the hyperparameters. For example, the number of epochs has been chosen as the best compromise between error loss function evaluation and time needed to complete the total number of iterations. Choosing a higher number means having a more precise prediction, with the downside of taking longer to be developed. The same idea was applied to most of the choices made while tuning, for the batch size and the architecture, and even for the number of neurons in each hidden layer.

Second, a higher number of dataset samples coincides with a better accuracy, provided that the data is divided correctly into the three aforementioned categories: training, validation, and testing. This number only depends on the quantity of simulations run before proceeding with the coding for the neural networks.

Finally, the comparison between the two different types of NNs highlights the advantages of the more modern architecture of the Recurrent Neural Network. The dependence on the time of the simulation results gives the edge to this type of network as it behaves better with time sequences. Looking at all results, the error function calculated using the output values is more than halved when computing it for the RNN. Furthermore, the scenario that considers the loss function dependence on the time instant, the error is even lower.

In summary, with the right approach, this method may expand the boundaries of efficient simulation in computational fluid dynamics, as with Machine Learning-accelerated CFD [32] it is possible to solve expensive simulations much faster and with an acceptable accuracy without additional costs. Moreover, this technique is applicable to both physical tasks (for example airplane or automotive design) and numerical climate prediction (time dependent solutions), which indicates the vast chances it may have. Nonetheless, the undergoing improvements on the deep learning models will allow the same data prediction with an even lower computational cost.

6 Bibliography

- [1] NASA, *Navier-Stokes Equations*. https://www.grc.nasa.gov/www/k-12/airplane/nseqs.html
- [2] idealsimulation, *Turbulence Models in CFD*, https://www.idealsimulations.com/resources/turbulence-models-in-cfd/
- [3] D.C. Wilcox. *Turbulence Modeling for CFD, Third Edition*, 2006
- [4] E. Furbo. *Evaluation of RANS turbulence models for flow problems with significant impact of boundary layers*, Examensarbete, 2010
- [5] Wikipedia contributors. *K-epsilon turbulence model* – Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/K-epsilon_turbulence_model
- [6] D.C. Wilcox. *Formulation of the k-ω Turbulence Model Revisited*, AIAA Journal, Vol. 46, No. 11, 2008
- [7] IBM Cloud Education, *Unsupervised Learning*, https://www.ibm.com/cloud/learn/unsupervised-learning, 2020.
- [8] S. Ghost-Dastidar, H. Adeli. Chapter: *Third Generation Neural Networks: Spiking Neural Networks*, Advances in Computational Intelligence (pp.167-178). August 2009.
- [9] Stanford University, *Deep Learning for Computer Vision*, https://cs231n.github.io/neural-networks-1/
- [10] Machine Learning Mastery. *How to Choose an Activation Function for Deep Learning*. https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/ *by J. Brownlee, 2021*
- [11] A. Farzad, H. Mashayekhi, H. Hassanpour. *A comparative performance analysis of different activation functions in LSTM networks for classification*, Neural Computing and Applications pp. 2507-2521, 2017
- [12] SebastianRaschka, *Why is the ReLU function not differentiable at x=0?* https://sebastianraschka.com/faq/docs/relu-derivative.html
- [13] Wikipedia contributors. *Backpropagation* – Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Backpropagation
- [14] Towards Data Science. *Understanding Backpropagation Algorithm.* https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd, by S. Kostadinov, 2018
- [15] Machine Learning Mastery. *A Gentle Introduction to Backpropagation Through Time*. https://machinelearningmastery.com/gentle-introduction-backpropagation-time/ by J. Brownlee, 2017
- [16] PaperspaceBlog, *Intro to optimization in deep learning: Gradient Descent.* https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/
- [17] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*, page 181, The MIT Press, 2016.
- [18] Machine Learning Mastery. *Loss and Loss Functions for Training Deep Neural Networks*. https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/
- [19] PyTorch. https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
- [20] L. Chen, K. Asai, T. Nonomura, G. Xi, T. Liu. *A review of Backward-Facing Step (BFS) flow mechanisms, heat transfer and control.* Thermal Science and Engineering Progress, Volume 6, pp 194-216, 2018
- [21] OpenFOAM User Guide, version 9. https://openfoam.org

- [22] Science Direct. *Feedforward Neural Networks*, https://www.sciencedirect.com/topics/chemical-engineering/feedforward-neural-networks
- [23] V7labs. *The Essential Guide to Neural Network Architectures*, https://www.v7labs.com/blog/neural-network-architectures-guide, by P. Baheti, 2022
- [24] R. K. Brouwer. *Feed-forward neural network for one-to-many mappings using fuzzy sets*, Neurocomputing, Volume 57, pp. 345-360, 2004
- [25] Towards Data Science. *Choosing the right Hyperparameters for a simple LSTM using Keras*. https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046, by K. Eckhardt, 2018
- [26] Towards Data Science. *Simple Guide to Hyperparameter Tuning in Neural Networks*. https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594, by M. Stewart, 2019.
- [27] StackOverflow, *Epoch vs Iteration when training neural networks*. https://stackoverflow.com/questions/4752626/epoch-vs-iteration-when-training-neural-networks, 2011
- [28] M. Uzair, N. Jamil. *Effects of Hidden Layers on the Efficiency of Neural Networks*, IEEE 23rd International Multitopic Conference (INMIC), 2020
- [29] IBM Cloud Education, *Recurrent Neural Networks*, https://www.ibm.com/cloud/learn/recurrent-neural-networks, 2020
- [30] Towards Data Science. *Recurrent Neural Networks and Natural Language Processing*. https://towardsdatascience.com/recurrent-neural-networks-and-natural-language-processing-73af640c2aa1, by C. Thomas, 2019
- [31] Analytics Vidhya page. *Recurrent Neural Networks (RNN's) and Time Series Forecasting*, https://medium.com/analytics-vidhya/recurrent-neural-networks-rnns-and-time-series-forecasting-d9ea933426b3, *by K. Kohli, 2020*
- [32] D. Kochkov, Jamie A. Smith, A. Alieva, S. Hoyer. *Machine learning-accelerated computational fluid dynamics*, PNAS 2021