ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

SCHOOL OD ENGINEERING AND ARCHITECTURE

DICAM - Department of Civil, Chemical, Environmental, and Materials Engineering

MASTER'S DEGREE IN CHEMICAL AND PROCESS ENGINEERING

MASTER'S THESIS
in

Dynamics And Control Of Chemical Processes M

Reduced order modelling of combustion using convolutional neural network

CANDIDATE:

Fabiola Amaducci

SUPERVISOR:
Prof.
Alessandro Paglianti

CO-SUPERVISRO
Prof.
Alessandro Parente

Academic Year 2019/2020

Session II, October 2020

Alla mia Mamma, al mio Bà...

e a tutti i ragazzi che finiscono ingegneria senza bere caffè

Abstract

It is well known that CFD simulations of a complex combustion system, such as Moderate or Intense Low-oxygen Dilution (MILD) combustion, requires considerable computational resources. This precludes various applications including the use of CFD in real time control systems. The idea of a reduced order model (ROM) was born from the desire to overcome this obstacle. A ROM, if properly instructed, returns the output of a requested CFD simulation in extremely short time. This one is an ideal mechanism with two basic gears: the input size reduction technique and the interpolation method. This project proposes a study on the applicability of convolutional neural network (CNN) as a dimensionality reduction technique. The code written for this purpose will be presented in detail, as well as pre and post processing. A sensibility analysis will be carry out to find out which parameters to adjust and how in order to achieve the optimum. Finally, the network will be compared in its peculiarity and its results with Principal Component Analysis (PCA), the technique used by the BURN group of Libre University of Bruxelles for the same purpose. Moreover with the desire to improve, we went further by trying to overcome the limits dictated by the rules of a legitimate comparation between PCA and CNN. Lastly, the author considers necessary to provide the theoretical basis in order to enrich and support what has just been described. Therefore, you will also find introductions / insights on MILD combustion, CFD of a combustion system, neural networks and the aspects related to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Energy is undoubtedly the single most important factor impacting the prosperity of our society. The need for innovation is particularly important in combustion, considering that the energy derived from burning fossil fuels (coal, petroleum or natural gas) supplies over two thirds of the total world energy needs. Thus, new breakthroughs in clean energy are needed to provide our society with the necessary resources in a way that also protects the environment and addresses the climate change issue. A certain number of new combustion technologies have been proposed in recent years. Among them, Moderate or Intense Low-oxygen Dilution (MILD) combustion is certainly one of the most promising. This one features high fuel flexibility, increased efficiency and low pollution emissions. Even if, unfortunately, information in this field are still sparse.

1.1 MILD combustion

A complete definition of MILD is given by Cavaliere:

A combustion process is named Mild when the inlet temperature of the re-

actant mixture is higher than mixture self-ignition temperature whereas the maximum allowable temperature increase with respect to inlet temperature during combustion is lower than mixture self-ignition temperature.

This means that process evolves in a rather narrow temperature range, which could be placed in an intermediate region between the very fast kinetics of the oxidative undiluted conditions and the relatively slow kinetics linked to low temperature self-ignition regimes. For sure a difference is: in Mild Combustion the process cannot be sustained without preheating the reactants.

The narrow temperature range, under which the process proceeds, allows design, optimisation and adjustment in the process by fine tuning external parameters. These parameters provide controllable shifts of internal parameters in the reactor. In contrast traditional combustion processes are difficult to control because they proceed along temperature excursions of thousands of degrees. Kinetic can change during the completion of the process from low to intermediate or high temperature regimes, the physical parameter, like diffusion, surface tension, can also change abruptly from one to the other. In contrast, Mild Combustion mode is characterised by 'mild' changes and ensures a more gradual evolution.

The definition of Mild Combustion given here is unambiguous because criteria which should be fulfilled to include a process in Mild Combustion are well defined in unequivocal way. Mild Combustion has to be considered a new combustion regime. It is neither a deflagration nor a detonation nor a diffusion flame. It is a combustion process which is a superdiluted explosion or a continuous auto-ignition/explosion. The fluid-dynamic local conditions and thermodynamic constraints under which Mild Combustion develops are quite straightforward. This mode of combustion is achieved through the strong exhaust gas and heat recirculation, achieved by means of the internal aerodynamics of the combustion chamber in conjunction with high-velocity burners. Heat recovery by preheating the oxidant stream can also help in improving thermal efficiency and maintaining the MILD regime.

Mild is the acronym of moderate or intense low-oxygen dilution which is exactly one of the most typical conditions for which the process can be obtained. The relevance of this condition is due to its relatively simple feasibility and that it may be tuned in such a way that it prevents from soot and NOx formation.

MILD is a flameless combustion, it's referred to the outstanding characteristic that no visible emission is detectable in oxidation regions.

In summary, MILD combustion is characterised by elevated reactant temperature and low temperature increase, intensive reactant and product mixing, as well as no audible or visible flame, under ideal conditions. Moreover, MILD combustion delivers very low NOx and CO emissions and high efficiency, with a large flexibility of fuel types. The system is characterised by a more uniform temperature field than in traditional non-premixed combustion, and by the absence of high temperature peaks, thus suppressing NO formation through the thermal mechanism, while ensuring complete combustion and low CO emissions. This uniform field makes the process very unique in the material treatment field because this ensures temperature homogeneity and control on the material surface. Some applications in the steel treatment testifies the feasibility in exploiting such characteristic of the process. The homogeneity makes possible also to control both the combustion process itself and the addition of any chemical which can be beneficial in oxidation process or in its application. In other words the Mild Combustion makes the combustion chamber more similar to other chemical reactors, which are temperature controlled with the consequent benefit to adjust and tune the temperature in a convenient window.

However, what makes such technology very attractive is the large fuel flexibility, being suited for, industrial wastes, biogas and low-BTU fuels. These non-conventional fuels are blends of CH4, CO, H2, N2 and CO2 in variable proportions. In normal non-premixed flame, the generation of a stable flame can be difficult in presence of highly fluctuating compositions and low CH4/H2 contents. Flameless combustion can be a solution, since there is no need to stabilise a flame front, which can turn out to be complicated when the LHV of the burnt fuels is subjected to wide variation.

Usefulness of a combustion process has to be shown along the years and economic constrains sometime obscure long-time convenience. However, this mode has great potentials. This is linked to the fact that combustion process can be restricted to relatively low maximum temperature and temperature increase when Mild Combustion is adopted. The limitation of the maximum temperature can be exploited to limit soot and NOx production as it has been just mentioned. Furthermore, the maximum temperature can be adjusted in such a way that it is lower than that a high temperature metallic material can resist. For example: in the field of combustion engines this leads to an extremely useful freedom degree in the design of a combustion chamber.

MILD combustion technology has been demonstrated for many industrial applications. It was first introduced in industrial furnaces for methane combustion and later extensively investigated for other gaseous fuels like hydrogen and ethanol. Experiments and simulations on MILD oxidation burner have been executed, showing the effects of burner configuration and firing mode on efficiency and emissions. An oxygen enhanced regenerative burner operated in MILD combustion mode has been evalueted. An energy recovery ratio above 80% and NOx emissions below 5 ppm were achieved. Furthermore, prevaporised liquid fuels burning in a reverse-flow MILD combustor under elevated pressures have been studied. They concluded that combustion stability is largely dependent on fuel type and the NOx emission is highly influenced by the operating conditions of pressure, jet velocity and carrier gas. MILD technology can be utilised in gas turbines as well. Experimental and numerical studies on gas turbine under MILD condition, using gaseous fuel are conducted. The effect of pressure, mixing on combustion stability was analysed, indicating that mixing is the key parameter to control and stabilize MILD combustion. Recently, the possibility of using liquid biofuels, diesel and kerosene fuels under MILD condition for gas turbine applications is evaluated. They stated that MILD combustion can potentially substitute conventional gas turbines. Furthermore, experts analysed the potential of oxy-MILD combustion for large scale pulverised coal boilers. Preliminary simulations showed the possibility of efficiency increase of more than 3%. The MILD combustion concept was also extended to hybrid solar thermal devices, which combine concentrated solar radiation with combustion. The integration of MILD combustion in a hybrid solar receiver can lead to increased thermal performances with respect to conventional flames.

1.2 CFD of a MILD combustion system

In recent years, attention has been paid to MILD combustion modelling, due to the very strong turbulence/chemistry interactions of such a combustion regime.This interaction definitely comes from strong mixing, the reduced temperature level typical of this combustion mode and slower reactions (due to the dilution of reactants). There is not a clear separation between large and small scales of turbulence, and reaction can occur over a wide range of scales. Therefore, the chemical reactions proceed in a thick reaction zone, comparable to the integral length scale, leading to a modification of the characteristic scales of the reaction structures. As a consequence, both phenomena must be considered (models based on the scale separation between turbulence and chemistry will fail in predicting the main features of such a combustion regime). Therefore, models that account for finite-rate chemistry must be considered. Furthermore, the usage of detailed kinetic scheme appears mandatory.

In this context the system is solved using Unsteady Reynolds Averaged Navier-Stokes (URANS) simulations in combination with finite-rate chemistry. The Partially-Stirred Reactor (PaSR) model is chosen for turbulence/chemistry interactions. In PaSR, the interaction between turbulence and chemistry is represented with a factor K, which is defined as the ratio between the chemical time scale and the sum of mixing and chemical scales. PaSR models the combustion process as a sequence of reaction and mixing processes in locally uniform regions. Both the chemical and mixing time scales are included in the model explicitly, allowing more comprehensive descriptions on turbulence/chemistry interactions. Therefore, its performances strongly depend on the accurate estimation of mixing and chemical time scales.

Turbulence model

In the context of compressible URANS simulations, the Favre-aver- aged (denoted with $\sim$) governing equations are solved:

$$\frac{\partial \overline{\rho}}{\partial t} + \frac{\partial (\overline{\rho}\widetilde{u}_j)}{\partial x_j} = 0 \tag{1.1}$$

$$\frac{\partial(\overline{\rho}\widetilde{u}_j)}{\partial t} + \frac{\partial(\overline{\rho}\widetilde{u}_j\widetilde{u}_i)}{\partial x_j} = -\frac{\partial \overline{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\overline{\tau_{ij}} - \overline{\rho}\widetilde{u''_i u''_j}\right) \tag{1.2}$$

$$\frac{\partial(\overline{\rho}\widetilde{h})}{\partial t} + \frac{\partial(\overline{\rho}\widetilde{h}\widetilde{u}_j)}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\overline{\rho}\alpha\frac{\partial \widetilde{h}}{\partial x_j} - \overline{\rho}\widetilde{u''_j h''}\right) - \frac{\partial \overline{q_{rj}}}{\partial x_j} + \overline{S}_{hc} \tag{1.3}$$

$$\frac{\partial(\overline{\rho}\widetilde{Y}_i)}{\partial t} + \frac{\partial(\overline{\rho}\widetilde{Y}_i\widetilde{u}_j)}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\left(\overline{\rho}D_{m,i} + \frac{\mu_y}{Sc_t}\right)\frac{\partial \widetilde{Y}_i}{\partial x_j}\right) + \overline{\dot{\omega}}_i \tag{1.4}$$

where:

- ρ is density.
- u is velocity.
- p is pressure.
- h is enthalpy.
- α is thermal diffusivity.
- Sc_t is turbulent Schmidt number
- $D_{m,i}$ is molecular diffusion coefficient for species i in the mixture

Reynolds average conveniently removes fluctuating components from the flow field variables without explicitly defining the spatial length scale used in the averaging operation. Averaging can be performed to extract the large-scale dynamics of the flow field. The key to simulating such large-scale dynamics is to average over the small-scale fluctuations and model the non-linear influence from the small-scale fluctuations, in the governing equations, that can alter the large-scale fluid motion Kajishima [8].

The standard $k - \varepsilon$ model is chosen as turbulence model. It's a two-equation turbulence model and it allows the determination of both, a turbulent length and time scale by solving two separate transport equations. The standard- model in ANSYS Fluent falls within this class of models and has become the workhorse of practical engineering flow calculations in the time since it was proposed by Launder and Spalding. Robustness, economy, and reasonable accuracy for a wide range of turbulent flows explain its popularity in industrial flow and heat transfer simulations. It is a semi-empirical model, and the derivation of the model equations relies on phenomenological considerations and empiricism. The standard-model is a model based on model transport equations for the turbulence kinetic energy and its dissipation rate. The model transport equation for K is derived from the exact equation, while the model transport equation for ε is obtained using physical reasoning and bears little resemblance to its mathematically exact counterpart. In the derivation of the model, the assumption is that the flow is fully turbulent, and the effects of molecular viscosity are negligible. The standard - model is therefore valid only for fully turbulent flows. The turbulence kinetic energy and its rate of dissipation are obtained from the following transport equations:

$$\frac{\partial(\rho k)}{\partial t} + \frac{\partial(\rho k u_i)}{\partial x_i} = \frac{\partial}{\partial x_j}\left[\left(\mu + \frac{\mu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] + G_k + G_b - \rho\varepsilon - Y_M + S_k \quad (1.5)$$

$$\frac{\partial(\rho \varepsilon)}{\partial t} + \frac{\partial(\rho \varepsilon u_i)}{\partial x_i} = \frac{\partial}{\partial x_j}\left[\left(\mu + \frac{\mu_t}{\sigma_\varepsilon}\right)\frac{\partial \varepsilon}{\partial x_j}\right] + C_{1\varepsilon}\frac{\varepsilon}{k}(G_K + C_{3\varepsilon}G_b) - C_{2\varepsilon}\rho\frac{\varepsilon^2}{k} + S_\varepsilon \quad (1.6)$$

where:

- $\widetilde{k}$ is turbulent kinetic energy.
- $\widetilde{\varepsilon}$ is dissipation rate.
- G_k is represents the generation of turbulence kinetic energy due to the mean velocity gradients.

- G_b is the generation of turbulence kinetic energy due to buoyancy.
- Y_M is represents the contribution of the fluctuating dilatation in compressible turbulence to the overall dissipation rate.
- $C_{1\varepsilon}, C_{2\varepsilon}, C_{3\varepsilon}$ are constants.
- $\sigma_k \sigma_\varepsilon$ are the turbulent Prandtl numbers for k and ε , respectively .
- S_k, S_ε and are user-defined source terms

It is based on the eddy viscosity assumption. The unresolved turbulence stresses $\overline{\rho}\widetilde{u_i''u_j''}$ are modelled with the product of an eddy viscosity μ_t and mean flow strain rate S_{ij}^* . The eddy viscosity μ_t in standard $k - \varepsilon$ model is estimated as:

$$\mu_t = \rho C_\mu \frac{\widetilde{k}^2}{\widetilde{\varepsilon}} \tag{1.7}$$

where:

- C_μ is a constant.

Combustion model

Different combustion models exist in the framework of RANS (Reynolds-averaged Navier-Stokes), for example Eddy Dissipation Concept (EDC) model and a newer version of that where is showed that adjusting the EDC coefficients Ctau and Cy from their default value results in significantly improved performance under MILD combustion. Afterwards, Parente proposed functional expression showing the dependency of the EDC coefficients on dimensionless flow parameters, such as Reynolds and Damköhler number. But beside the EDC model, the Partially Stirred Reactor (PaSR) combustion model was proposed for MILD combustion. It was found that EDC fails in providing a reasonable estimation of the ignition region, while improved predictions can be obtained using the PaSR model. So PaSR model has been used in this project.

In the PaSR model, the computational cell is split into two locally uniform zones: one where reactions take place, and another characterized by only mixing. The final species concentration of the cell is determined from the mass exchange between the two zones, driven by the turbulence. A conceptual drawing of the PaSR model is shown in figure 1.1 .

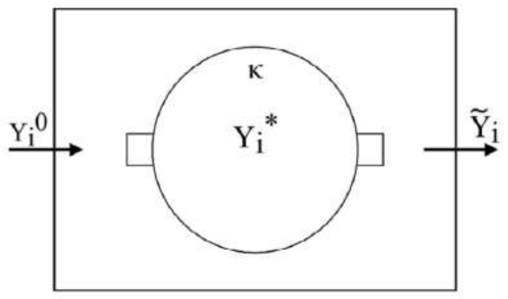


Figure 1.1: Conceptual drawing of the PaSR model

The drawing in figure 1.1 refers to one computational cell, in which Y_i^0 is the initial i_{th} species mass fraction in the non-reactive region, $\widetilde{Y_i}$ is the final averaged i_{th} species mass fraction in the cell and Y_i^* is the i_{th} species mass fraction in the reactive zone. K is the mass fraction of the reaction zone in the computational cell:

$$K = \frac{\tau_C}{\tau_C + \tau_{mix}} \tag{1.8}$$

where τ_C and τ_{mix} are the characteristic chemical and mixing time scales in each cell, respectively.

The mean source term provided to the species transport equation can be expressed as:

$$\overline{\dot{\omega}_i} = K\frac{\widetilde{\rho}(\widetilde{Y_i^*} - Y_i^0)}{\tau^*} \tag{1.9}$$

where τ^* represents the residence time in the reactive structure. In the present work, it's equals to the mixing time scale. In order to get the value of Y_i^*, a time-splitting approach is applied. The reactive zone is modelled as an ideal reactor evolving from Y_i^0, during a residence time τ^*:

$$\frac{dY_i^*}{dt} = \frac{\dot{\omega}_i}{\rho} \tag{1.10}$$

The term $\dot{\omega}_i$ is the instantaneous formation rate of species i. The final integration of $\frac{dY_i^*}{dt}$ is Y_i^*.

τ_{mix} can be estimated following different approaches.

- Kolmogorov time scale: In conventional combustion systems, it is often assumed that reactions happen at the dissipation scales, of the order of the Kolmogorov one[1]. However, in MILD combustion, reactions can occur over a wide range of flow scales, and the use of the Kolmogorov mixing time scale could lead to inaccurate predictions of temperature and species mass fractions. $\tau_{mix} = (\nu/\varepsilon)^{1/2}$
- Integral time scale: $\tau_{mix} = k/\varepsilon$
- fraction of integral time scale: $\tau_{mix} = C_{mix}k/\varepsilon$
- Geometric mean of Kolmogorov and integral time scales: To provide a more accurate evaluation of the mixing time, the whole spectrum of time scales is proposed to consider. A simple approach to achieve this is to take only the two most important time scales, via the geometrical mean of the Kolmogorov and integral time scales.
- Dynamic time scale: the three ways of estimating mixing scales introduced above can be regarded as global approaches. A more sophisti-

[1]Kolmogorov microscales are the smallest scales in turbulent flow $\tau_{mix} = (\nu/\varepsilon)^{1/2}$

cated approach is based on the automatic definition of τ_{mix} based on local properties of the flow field using a dynamic approach.

In our project, Integral time scale with $C_{mix} = 0.5$ has been used (however, despite satisfying predictions, this approach has several drawbacks. Indeed, C_{mix} is not a function of local variables, being arbitrarily chosen and constant in every cell of the domain. This implies that a model sensitivity must be always carried out to use this model, since no a priori method can be used to infer the value of C_{mix}.

τ_C can be evaluated with these approaches:

- Chemical time scale estimation from Jacobian matrix eigenvalues: using the eigenvalues of the Jacobian matrix of the chemical source terms. After the decomposition of the Jacobian matrix, the chemical time scale is estimated with the inverse of the eigenvalues. After removing the dormant species (characterised by infinite time scale values), the slowest chemical time scale is chosen as leading scale for the evaluation of the PaSR parameter K.
- Chemical time scale estimation from formation rates: the decomposition of the source term Jacobian matrix is accurate but time consuming, especially when large scale simulations with much detailed mechanism is used. The formation rate based characteristic time scale evaluation is a simplified approach. Instead of getting the chemical time scale for each species from the Jacobian matrix decomposition, the ratio of species mass fraction and formation rate in the reactive structure is directly used, approximating the Jacobian diagonal terms.
- Chemical time scale estimation from reaction rates: Another simplified

method is based on the reaction rate. Similar to the two approaches above.

The decomposition of the source term Jacobian matrix is the most accurate and time consuming method for the evaluation of the chemical time scale. The approach based on the formation rates provides the best compromise between accuracy and computational cost, while the approach based on reaction rates may lead to inaccurate results as it tends to over-predict the chemical time scales Parente [9]. For these reasons, in our project a chemical time scale estimation from formation rates has been used.

Kinetic mechanism

A sensitivity study was been carried out to select a kinetic scheme, comparing the KEE (17 species and 58 reactions) and GRI-2.11 (31 species and 175 reactions) mechanisms. Being the difference between the two schemes below 3%, KEE was selected for its lower computational cost.

1.3 Reduced order model

Detailed numerical simulations of detailed combustion systems require substantial computational resources, In many engineering applications, complex physical systems can only be described by high-fidelity expensive simulations. The coupling of Computational Fluid Dynamics (CFD) and detailed chemistry is computationally demanding, mainly because of the large number of species and the wide range of chemical times typically involved in complex chemistry, even due to the non-linearity of these problems. Changing the operating conditions, namely the model's input parameters, can drastically change the state of the considered system. Complete knowledge about the investigated system's behaviour for a full range of operating conditions can therefore only be achieved by running these expensive simulations several times with different inputs, until enough observations of the system's state are obtained. We focus on MILD, these combustion systems fall in this category as they are characterized by very complex physical interactions, between chemistry, fluid-dynamics and heat transfer processes. During the last years, several techniques have been proposed for reducing the computational cost because the use of CFD tools in real time is still unrealistic due to the reasons listed above.

One of these techniques is reduced order model. The key feature of reduced order models is their capability for drastically reducing the computational cost, while maintaining a sufficient accuracy from the engineering point of view. ROM represents the behaviour of complex reacting systems in a wide range of conditions, without the need for expensive Computational Fluid Dynamics (CFD) simulations.

In other words: a specific computationally expensive CFD simulation or computer code, referred to as Full-Order Model (FOM) is treated as a black box that generates a certain output $\mathbf{y}$ (e.g. the temperature field) given a set of input parameters $\mathbf{x}$ (e.g. the equivalence ratio) and indicated by Υ:

$$\mathbf{y} = \Upsilon(\mathbf{x}) \tag{1.11}$$

The evaluation of the function Υ usually requires many hours of computational time. After enough observations of the FOM's output are available, $\mathbf{y(xi)}$ $\forall i = 1, ..., M$, a ROM can be trained and the output $\mathbf{y}^*$ for a particular set of unexplored inputs $\mathbf{x}^*$ can be predicted without the need to evaluate $\Upsilon(\mathbf{x}^*)$. The function Υ is therefore approximated by a new function Ψ whose evaluation is very cheap compared to Υ:

$$\mathbf{y}^* = \Upsilon(\mathbf{x}^*) \approx \Psi(\mathbf{x}^*) \tag{1.12}$$

In this context, the availability of physics-based reduced-order models (ROMs) becomes very attractive, to embed the critical aspects of a detailed simulations into simplified relationships between the inputs and outputs that can be used in real time. The development of virtual models, also referred to as digital twins, of industrial systems opens up a number of opportunities, such as the use of data to anticipate the response of a system and brainstorm malfunctioning, and the use of simulations to develop new technologies, i.e. virtual prototyping. A definition of digital twins is *an integrated multi-physics, multi-scale, probabilistic simulation of an as-built system, enabled by digital thread, that uses the best available models, sensor information, and input data to mirror and predict activities/performance over the life of its corresponding physical twin.* Combining CFD simulations with real-time data coming from sensors of a real industrial system to foresee a change in its state is possible only if the prediction of the system's state based on the operating conditions reported by these sensors becomes instantaneous. To do so, a set of training simulations must be generated beforehand, for a wide enough range of possible operating conditions. A physics-based ROM can be then developed in two steps:

- use unsupervised learning to extract the key latent features in the data.
- find a response surface by a supervised learning technique.

Once the mapping between inputs and outputs is embedded in a ROM, the system state can be predicted for new operating conditions, based on real-time data coming from sensors. Without run another new CFD simulation and very quickly.

Parente et al. 2020 have developed a ROM based on the Kriging-PCA approach. In that work, the combination of Principal Component Analysis (PCA) with Kriging has been considered to identify accurate low-order models. PCA is used to identify and separate invariants of the system, the PCA modes, from the coefficients that are instead related to the characteristic operating conditions. Kriging is then used to find a response surface for these coefficients. In this section we will see a little explanation of PCA and Kriging.

PCA

Principal Component Analysis (PCA) offer the potential of preserving the physics of the system while reducing the size of the problem. PCA is a statistical technique used to find a set of orthogonal low-dimensional basis functions, called Principal Components (PCs), to represent an ensemble of high-dimensional data. In other words from the data-set $\mathbf{Y}$ of available simulations, PCA is able to extract a set of basis functions $\Phi = \Phi\left(\Phi_1, \Phi_2, ..., \Phi_q\right)$, with $q < N$ usually, called PCA modes or PC that are invariant with respect to the input parameters $\mathbf{x}$. A set of coefficients $\mathbf{a(x)} = a_1(\mathbf{x}), a_2(\mathbf{x}), ..., a_q(\mathbf{x})$, called PCA scores (or coefficients) and depending on $\mathbf{x}$, is consequently found.

An illustrative example is reported in Figure 1.2, where a temperature spatial field is represented as a set of coefficients $a_1, a_2...a_q$ that weight a set of basis functions, i.e. the PCs. These coefficients are less in number than the original number of variables as $q < N$ and can be interpolated in order to acquire knowledge about the system's state for any unexplored point $x^* \in$D. In order to have more info about PCA see Appendix A.

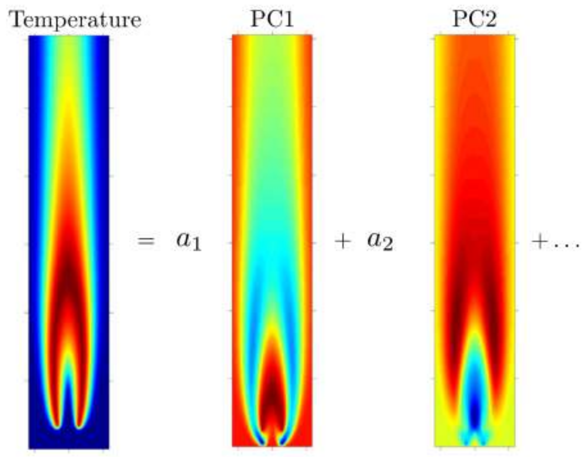


Figure 1.2: Illustrative example of PCA.

Kriging

Kriging is an interpolation method in which every realization a($\mathbf{x}$) is expressed as a combination of a trend function and a residual:

$$a(\mathbf{x}) = \mu(\mathbf{x}) + s(\mathbf{x}) = f^T(x)\beta + z(\mathbf{x}) \tag{1.13}$$

The trend function $\mu(\mathbf{x})$ is a low-order polynomial regression and provides a global model in the input space. The residuals z($\mathbf{x}$) are modelled by a Gaussian process with a kernel or correlation function that depends on a set of hyper-parameters to be evaluated by Maximum Likelihood Estimation (MLE).

This approach can faithfully reproduce the temperature and chemical species fields in a reacting flow simulation.

Now we want to go forward, we ask ourself if there is or not another way to create a digital twin and if it would work better or not. With this purpose, in this project we want to explor a different way to reduce the size of simulation image. Instead of using PCA, we will use Convolutional neural network (CNN). CNN will be explained more deeper in Chapter 2.

Chapter 2

Methods

In each hemisphere of our brain, humans have a primary visual cortex, also known as V1, containing 140 million neurons, with tens of billions of connections between them. And yet human vision involves not just V1, but an entire series of visual cortices - V2, V3, V4, and V5 - doing progressively more complex image processing. We carry in our heads a supercomputer. We humans are stupendously, astoundingly good at making sense of what our eyes show us. But nearly all that work is done unconsciously. And so we don't usually appreciate how tough a problem our visual systems solve.

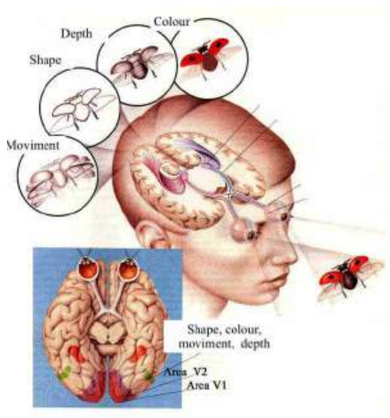


Figure 2.1: Visual cortex

The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program. Simple intuitions about how we recognise shapes turn out to be not so simple to express algorithmically. When you try to make such rules precise, you quickly get lost in a morass of exceptions and caveats and special cases.

Machine learning, a branch of artificial intelligence, to which neural networks belong, arises from this question: could a computer go beyond what we know how to order it to perform and learn on its own how to perform a specified task? Could a computer surprise us? Rather than programmers crafting data-processing rules by hand, could a computer automatically learn these rules by looking at data? This question opens the door to a new programming paradigm. In classical programming, the paradigm of symbolic AI artificial intelligence, humans input rules (a program) and data to be processed according to these rules, and out come answers. With machine learning, humans input data as well as the answers expected from the data, and out come the rules. These rules can then be applied to new data to produce original answers. A machine-learning system is trained rather than explicitly programmed. It's presented with many examples relevant to a task, and it finds statistical structure in these examples that eventually allows the system to come up with rules for automating the task. Following, a practical concept to implement machine learning is presented.

Artificial Neural Network or ANN is a computational model that consists of several processing elements that receive inputs and deliver outputs based on their predefined activation functions. It is inspired by the way the biological nervous system such as brain process information. It is composed of large number of highly interconnected processing elements (neurons) working in unison to solve a specific problem. Neurons are organized into layers that have a specific role. In its simplest form, an ANN can have only three layers of neurons: the input layer (where the data enters the system), the hidden layer (where the information is processed) and the output layer (where the system decides what to do based on the data). But ANNs can get much more complex than that, and include multiple hidden layers. With these base concepts, we can talk about Deep Learning. It's nothing but a specific sub-field of machine learning we were talking about before: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations. The deep in deep learning isn't a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations. How many layers contribute to a model of the data is called the depth of the model. Other appropriate names for the field could have been layered representations learning and hierarchical representations learning. This kind of ANN is called a *deep neural network* (Fig 2.2). We will see a more detailed discussion about these arguments in the next sections.

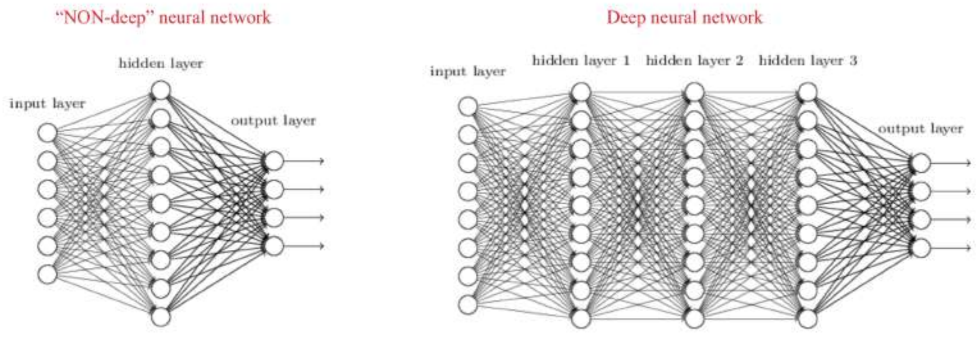


Figure 2.2: Example of Non-deep and deep neural network

2.1 Neurons

The following diagram represents the general model of a neuron which is inspired by a biological neuron.

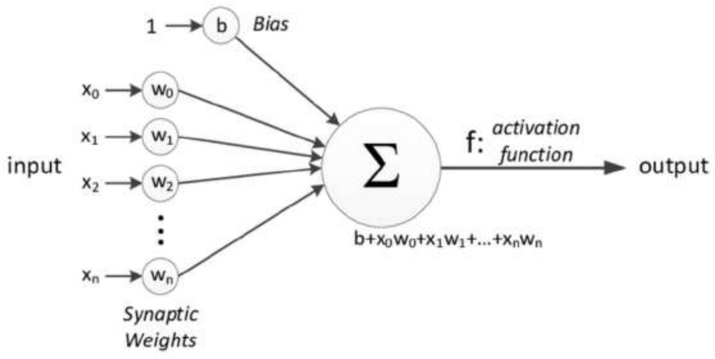


Figure 2.3: Neuron

Rosenblatt proposed a simple rule to compute the output. He introduced weights, $w_1, w_2, ...w_n$ real numbers expressing the importance of the respective inputs to the output. Weight shows the strength of a particular input node and ideally they can be seen as the values assigned to the bonds that connect neurons. The neuron's output called activation (in this case 0 or 1) is determined by whether the weighted sum $\Sigma_j w_j x_j$ is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

$$\text{output} = \begin{cases} 0 \longrightarrow \text{if } \Sigma_j w_j x_j \leq \text{threshold} \\ 1 \longrightarrow \text{if } \Sigma_j w_j x_j > \text{threshold} \end{cases} \tag{2.1}$$

By varying the weights and the threshold, we can get different models of decision-making. The condition $\Sigma_j w_j x_j$ threshold is cumbersome, and we can make two notational changes to simplify it.

- The first change is to write $\Sigma_j w_j x_j$ as a dot product, $w \cdot x \equiv \Sigma_j w_j x_j$, where w and x are vectors whose components are the weights and inputs, respectively.

- The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the neuron's bias, $b \equiv -\text{threshold}$. Using the bias instead of the threshold, the neuron rule can be rewritten:

$$\text{output} = \begin{cases} 0 \longrightarrow \text{if } \Sigma_j w_j x_j + b \leq 0 \\ 1 \longrightarrow \text{if } \Sigma_j w_j x_j + b > 0 \end{cases} \tag{2.2}$$

When you compute a weighted sum like this you might come out with any number. So what is common to do is apply an activation function $\sigma(\Sigma_j w_j x_j + b)$ that compresses the real output into the wanted range. In the previous example Eq 2.2, a Threshold Activation Function (Binary step function) was applied. Depending on the result obtained by the function, the neuron will be more or less activated. For example, we have a neural network used to classify numbers from 0 to 9, as in Fig 2.4. Let's focus our attention to the first neuron (Fig. 2.5), it has the responsibility to identify the presence of a circle. it's connected to all neurons of the previous layer but through different weight for each one. In particular this neuron will be activated only if the previous neurons with biggest weight are active. In Fig 2.6 neuron activation to identify the 9 number is shown.

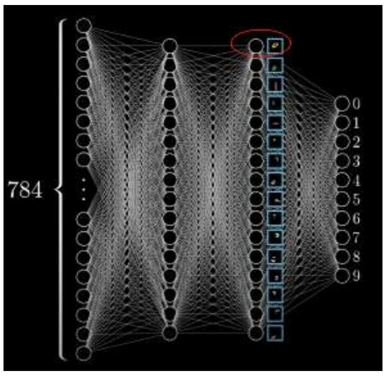


Figure 2.4: NN for identification of numbers(a)

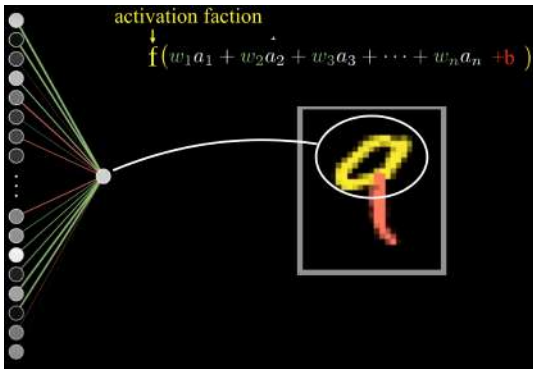


Figure 2.5: NN for identification of numbers (b)

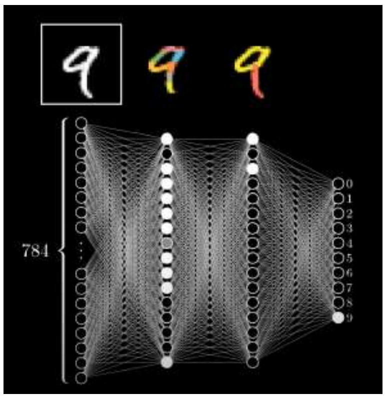


Figure 2.6: NN for identification of numbers (c)

2.2 Activation function

Neural network activation functions (also known as Transfer Function) are a crucial component of deep learning. Activation functions determine the output of a deep learning model, its accuracy, and also the computational efficiency of training a model, moreover it can make or break a large scale neural network. Activation functions also have a major effect on the neural network's ability to converge and the convergence speed, or in some cases, activation functions might prevent neural networks from converging in the first place.

Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated (fired) or not, based on whether each neuron's input is relevant for the model's prediction. Activation functions also help normalize the output of each neuron to a range between 1 and 0 or between -1 and 1. An additional aspect of activation functions is that they must be computationally efficient because they are calculated across thousands or even millions of neurons for each data sample. Modern neural networks use a technique called backpropagation to train the model (further on we will see this concept better) , which places an increased computational strain on the activation function, and its derivative function. Following, some of the most common activation functions.

Binary Step Function

A binary step function is a threshold-based activation function. If the input value is above or below a certain threshold, the neuron is activated and sends exactly the same signal to the next layer.

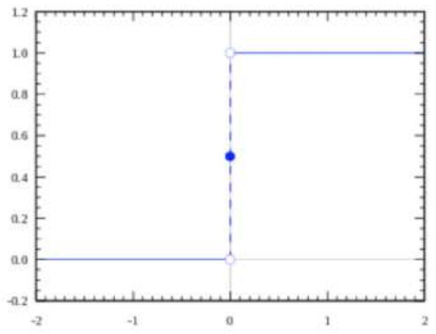


Figure 2.7: Binary Step Function

The problem with a step function is that it does not allow multi-value outputs (for example, it cannot support classifying the inputs into one of several categories).

Linear Activation Function

$$A = cx \tag{2.3}$$

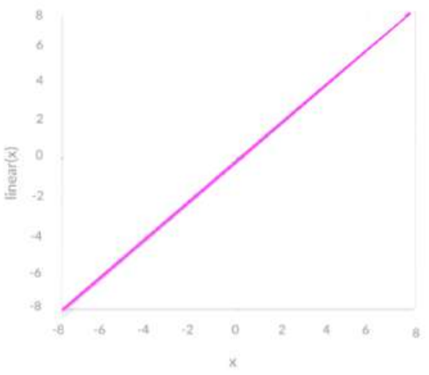


Figure 2.8: Linear Activation Function

It takes the inputs, (applies the weighted sum from neurons), and creates an output signal proportional to the input. In one sense, a linear function is better than a step function because it allows multiple outputs, not just yes and no. However, a linear activation function has two major problems:

- Not possible to use backpropagation (gradient descent) to train the model, the derivative of the function is a constant, and has no relation to the input, X. So it's not possible to go back and understand which weights in the input neurons can provide a better prediction.

- All layers of the neural network collapse into one with linear activation functions, no matter how many layers in the neural network, the last layer will be a linear function of the first layer (because a linear combination of linear functions is still a linear function). So a linear activation function turns the neural network into just one layer.

Modern neural network models use non-linear activation functions. They allow the model to create complex mappings between the network's inputs and outputs, which are essential for learning and modelling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality. Almost any process imaginable can be represented as a functional computation in a neural network, provided that the activation function is non-linear. Non-linear functions address the problems of a linear activation function listed above.

Sigmoid activation function

$$\sigma(x) = \frac{1}{1 + exp(-x)} \tag{2.4}$$

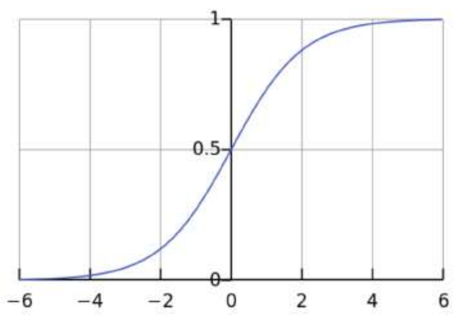


Figure 2.9: Sigmoid activation function

Sigmoid activation function has these advantages:

- Smooth gradient, preventing jumps in output values.
- Output values bound between 0 and 1, normalizing the output of each neuron.
- Clear predictions. For X above 2 or below -2, tends to bring the Y value (the prediction) to the edge of the curve, very close to 1 or 0. This enables clear predictions.

But it has also some disadvantages:

- Vanishing gradient. For very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
- Outputs not zero centered.
- Computationally expensive

Hyperbolic Tangent activation function

$$f(x) = \tanh(x) = \frac{2}{1 + exp(-2x)} - 1 \tag{2.5}$$

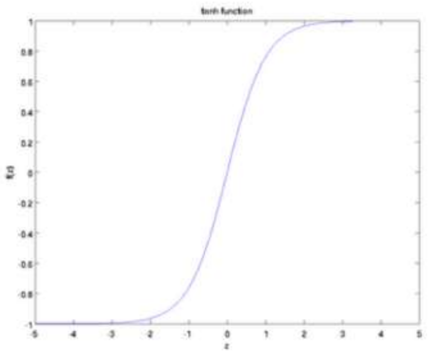


Figure 2.10: Hyperbolic Tangent activation function

This activation function has the same advantages and disadvantages of the previous one. In addiction it has the zero centered advantage that makes it easier to model inputs that have strongly negative, neutral, and strongly positive values.

ReLU (Rectified Linear Unit) activation function

$$f(x) = max(0, x) \tag{2.6}$$

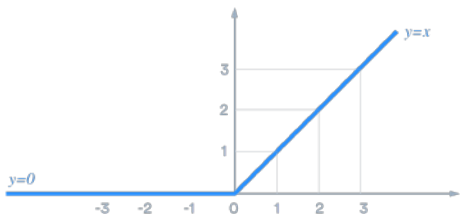


Figure 2.11: ReLU (Rectified Linear Unit) activation function

Sigmoid activation function has these advantages:

- Computationally efficient. allows the network to converge very quickly
- Non-linear. Although it looks like a linear function, ReLU has a derivative function and allows for backpropagation.

The disadvantage is the Dying ReLU problem. When inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.

Leaky ReLU activation function

$$f(x) = max(0.1 * x, x) \tag{2.7}$$

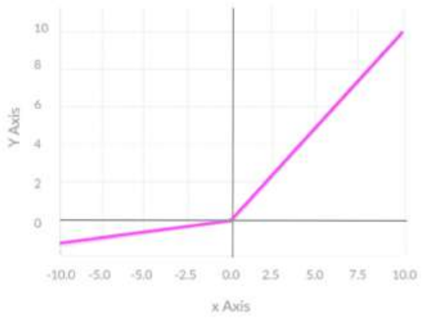


Figure 2.12: Leaky ReLU activation function

The added advantage of this activation function is preventing dying ReLU problem. This variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values. With the disadvantage that obtain inconsistent results for negative value is possible.

2.3 Autoencoder

An autoencoder neural network is an unsupervised Machine Learning algorithm that applies backpropagation, setting the target values to be equal to the input. It compresses the input into a lower-dimensional code and then reconstructs the output from this representation. The code is a compact summary or compression of the input, also called the latent-space representation.

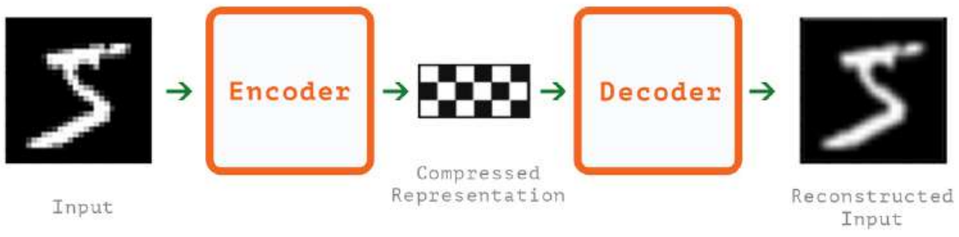


Figure 2.13: Conceptual scheme of an autoencoder

The autoencoder architecture consists of 3 components: encoder, code and decoder. The encoder, which is a fully-connected ANN, compresses the input and produces the code, the decoder then reconstructs the input only using this code. The goal is to get an output identical with the input. Note that the decoder architecture is the mirror image of the encoder. This is not a requirement but it's typically the case. The only requirement is the dimensionality of the input and output needs to be the same. Anything in the middle can be played with.

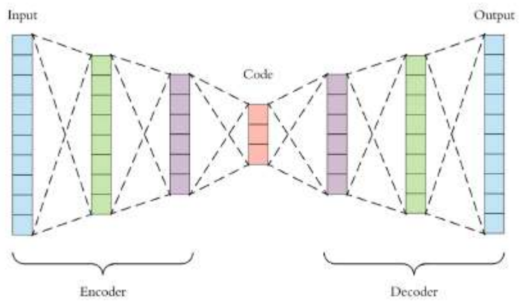


Figure 2.14: Autoencoder architecture

Autoencoders are mainly a dimensionality reduction (or compression) algorithm with a couple of important properties:

- Data-specific: Autoencoders are only able to meaningfully compress data similar to what they have been trained on. Since they learn features specific for the given training data, they are different than a standard data compression algorithm.
- Lossy: The output of the autoencoder will not be exactly the same as the input, it will be a close but degraded representation.
- Unsupervised: autoencoders are considered an unsupervised learning technique since they don't need explicit labels to train on.

In order to train and test autoencoder, define a cost function and a learning method is necessary. We will go deeper into them in the Section 2.4. Now we will talk a little bit about a comparison between autoencoder and PCA.

2.3.1 Autoencoder and PCA

There are a few ways to reduce the dimensions of large data sets to ensure computational efficiency such as backwards selection, removing variables exhibiting high correlation, high number of missing values but by far the most popular is principal components analysis. But a relatively new method of dimensionality reduction is the autoencoder. To summarise, PCA essentially learns a linear transformation that projects the data into another space, where vectors of projections are defined by variance of the data. By restricting the dimensionality to a certain number of components that account for most of the variance of the data set, we can achieve dimensionality reduction. Autoencoders are neural networks that can be used to reduce the data into a low dimensional latent space by stacking multiple non-linear transformations (layers).

If we want to do a comparison:

- PCA is essentially a linear transformation but Autencoders are capable of modelling complex non linear functions. A linearly activated Autoencoder approximates PCA. Mathematically, minimizing the reconstruction error in PCA modeling is the same as a single layer linear Autoencoder. An Autoencoder extends PCA to a nonlinear space.

- PCA features are totally linearly uncorrelated with each other since features are projections onto the orthogonal basis. But autoencoded features might have correlations since they are just trained for accurate reconstruction.

- PCA is able to recognize features that are invariant in space, on the contrary autoencoders learn how to recognize this feature regardless of where it is in the image.

- PCA is faster and computationally cheaper than autoencoders.

- Autoencoder is prone to overfitting due to high number of parameters (though regularization and careful design can avoid this).

- Autoencoders are usually used for large datasets.

Apart from the consideration about computational resources, the choice of technique depends on the properties of feature space itself. If the features have non-linear relationship with each other than autoencoder will be able to compress the information better into low dimensional latent space leveraging its capability to model complex non-linear functions.

With the following examples, the writer is sure that the difference between PCA and autoencoder will be clear. Here we construct two dimensional feature spaces with linear and non-linear relationship between them (with some added noise).

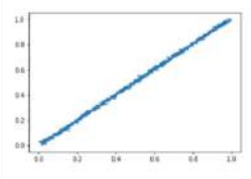
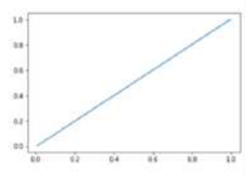
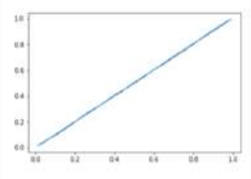
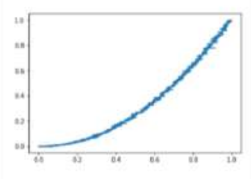
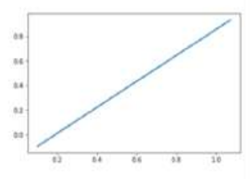
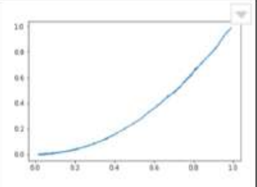
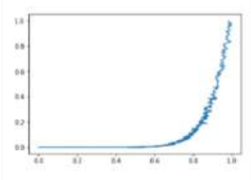
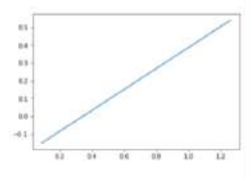
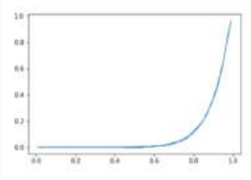
Function	**Feature Space**	**PCA Reconstruction**	**Auto Encoder Reconstruction**
y=mx+c			
y=mx^2+c			
y=mx^8+c			

Figure 2.15: 2D examples

It is evident if there is a non linear relationship (or curvature) in the feature space, autoencoded latent space can be used for more accurate reconstruction. Where as PCA only retains the projection onto the first principal component and any information perpendicular to it is lost. A similar conclusion is possible to obtain conducting experiments in 3D. In case of a curved surface two dimensional PCA is not able to account for all the variance and thus loses information. The projection to the plain that covers the most of variance is retained and other information is lost, thus reconstruction is not that accurate.

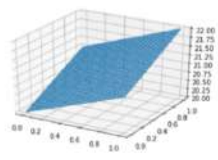
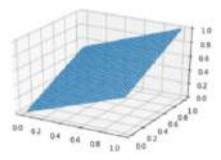
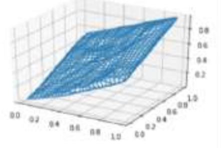
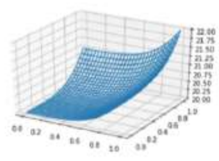
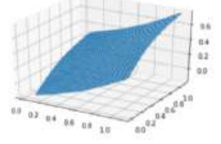
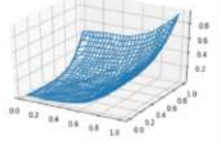
Function	Feature Space	PCA Reconstruction	Auto Encoder Reconstruction
Plane			
Curved Surface			

Figure 2.16: 3D examples

2.4 Learning method

In neural networks, parameters are used to train the model and make predictions. There are two types of parameters:

- Hyperparameters are external parameters set by the operator of the neural network, for example, selecting which activation function to use or the batch size used in training.
- Model parameters are internal to the neural network, for example, neu-

ron weights.

It is needless to say the network is going to perform pretty horribly on a given training example initialising all weights and biases totally randomly. So, first you define a cost function and then an optimisation strategy aim at minimising the cost function.

2.4.1 Cost function

Cost function is a function that measures the performance of a Machine Learning model for given data. Cost Function quantifies the error between predicted values and expected values (called loss) and presents it in the form of a single real number. The loss function (or error) is for a single training example, while the cost function is over the entire training set (or mini-batch for mini-batch gradient descent).

Broadly, loss functions can be classified into two major categories depending upon the type of learning task we are dealing with Regression losses and Classification losses. In classification, we are trying to predict output from set of finite categorical values i.e Given large data set of images of hand written digits, categorizing them into one of 0-9 digits. Regression, on the other hand, deals with predicting a continuous value. Following, we will present some regression cost function. We'll use a^l_j to denote the activation for the j^{th} neuron in the l^{th} layer and we will call the desired output with symbol y.

Mean Absolute Error/L1 Loss

$$MAE = \frac{\sum_{i=1}^{n} |a_i - y_j|}{n} \tag{2.8}$$

It's measured as the average of sum of absolute differences between predictions and actual observations. MAE needs complicated tools such as linear programming to compute the gradients. MAE is robust to outliers since it does not make use of square. MAE doesn't add any additional weight to the distance between points, the error growth is linear.

Mean Bias Error

$$MBE = \frac{\sum_{i=1}^{n} (a_i - y_j)}{n} \tag{2.9}$$

This is much less common in machine learning domain as compared to it's counterpart. This is same as MSE with the only difference that we don't take absolute values. Clearly there's a need for caution as positive and negative errors could cancel each other out.

Mean Square Error/Quadratic Loss/L2 Loss

$$MSE = \frac{\sum_{i=1}^{n}(a_i - y_j)^2}{n} \tag{2.10}$$

As the name suggests, Mean square error is measured as the average of squared difference between predictions and actual observations. It's only concerned with the average magnitude of error irrespective of their direction. However, due to squaring, predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions. MSE errors grow exponentially with larger values of distance. It's a metric that adds a massive penalty to points which are far away and a minimal penalty for points which are close to the expected result. Error curve has a parabolic shape. Plus MSE has nice mathematical properties which makes it easier to calculate gradients. Following, a figure that explains the different behaviour of MAE and MSE.

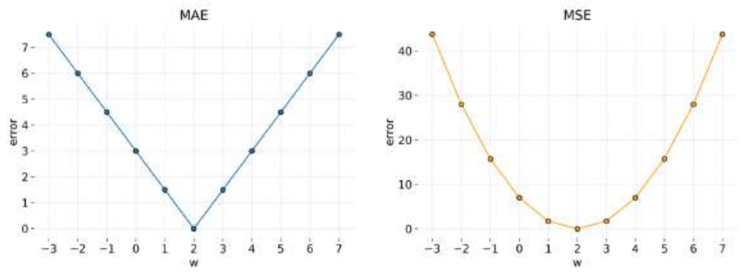


Figure 2.17: MAE and MSE

Mean Absolute Percentage Error

$$MA\%E = \frac{\sum_{i=1}^{n} \frac{|(a_i - y_j)|}{y_i}}{n} * 100 \tag{2.11}$$

Mean Squared Logarithmic Error

$$MA\%E = \frac{\sum_{i=1}^{n} (log(y_j + 1) - log(a_i + 1)^2}{n} \tag{2.12}$$

2.4.2 Optimisation strategy

At the base of optimization strategies there is often backpropagation which is useful for calculating the gradient of the cost function.

The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams. That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble. Today, the backpropagation algorithm is the workhorse of learning in neural networks.

At the heart of backpropagation is an expression for the partial derivative ($\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$) of the cost function C with respect to any weight w (or bias b) in the network. The expression tells us how quickly the cost changes when we change the weights and biases. Let's begin with a notation which lets us refer to weights in the network in an unambiguous way. We'll use w^l_{jk} to denote the weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. So, for example, the diagram below shows the weight on a connection from the fourth neuron in the second layer to the second neuron in the third layer of a network.

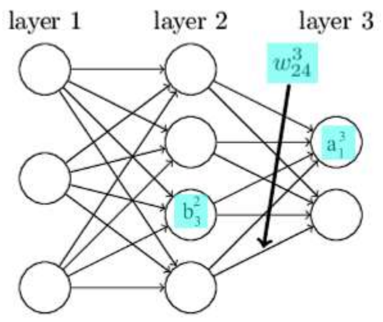


Figure 2.18: Notation of weights

We use a similar notation for the network's biases and activations. Explicitly, we use b^l_j for the bias of the j^{th} neuron in the l^{th} layer. And we use a^l_j for the activation of the j^{th} neuron in the l^{th} layer.

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) \tag{2.13}$$

Then, we need two assumptions about the cost function:

- it can be written as an average $C = \frac{1}{n_L}\sum_x Cx$ over cost functions C_x for individual training examples, x. Where n is number of neurons in the last layer.

- it can be written as a function of the outputs from the neural network

Let's consider a network that has only one neuron per layer and, now, we'll pay attention only for the last two layers ($L-1$ and L). We will call the desired output with symbol y. Assume a MSE function cost is used. So we have:

$$C_0(\text{weights, biases}) = (a^L - y)^2 \tag{2.14}$$

$$z^L = w^L a^{(L-1)} + b^L \tag{2.15}$$

$$a^L = \sigma(z^L) \tag{2.16}$$

The gradient, we want to obtain, is

$$\nabla \mathbf{C} = \begin{bmatrix} \frac{\partial C}{\partial w^1} \\ \frac{\partial C}{\partial b^1} \\ \cdots \\ \cdots \\ \frac{\partial C}{\partial w^L} \\ \frac{\partial C}{\partial b^L} \end{bmatrix} \tag{2.17}$$

In order to calculate each element, backpropagation applies the chain rule.

$$\frac{\partial C_0}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L} \tag{2.18}$$

A simple way to understand what happens in backpropagation is have a look to Fig 2.19 that shows how weights and bias can influence the cost function in this simple network.

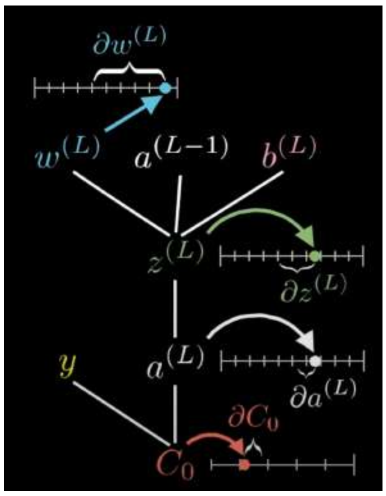


Figure 2.19: Chain rule scheme

Moreover it's valid:

$$\frac{\partial z^L}{\partial w^L} = a^{(L-1)} \tag{2.19}$$

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L) \tag{2.20}$$

$$\frac{\partial C_0}{\partial a^L} = 2(a^L - y) \tag{2.21}$$

The previous expression can be written:

$$\frac{\partial C_0}{\partial w^L} = a^{(L-1)}\sigma'(z^L)2(a^L - y) \tag{2.22}$$

This equation is valid for one training example, but we need to take into account all training example:

$$\frac{\partial C}{\partial w^L} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\partial C_k}{\partial w^L} \tag{2.23}$$

In a similar way, it's valid for partial derivatives of cost function respect to biases and activations.

$$\frac{\partial C_0}{\partial b^L} = \frac{\partial z^L}{\partial b^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C_0}{\partial a^L} = \sigma'(z^L)2(a^L - y) \tag{2.24}$$

$$\frac{\partial C_0}{\partial a^{L-1}} = \frac{\partial z^L}{\partial a^{L-1}} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L} = w^L \sigma'(z^L) 2(a^L - y) \tag{2.25}$$

If there is also another layer before, we have to do a step back to the previous layer (Fig 2.20) and it's easy to find the following expressions:

$$\frac{\partial C_0}{\partial w^{L-1}} = \frac{\partial z^{L-1}}{\partial w^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}} \frac{\partial z^L}{\partial a^{L-1}} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L} \tag{2.26}$$

$$\frac{\partial C_0}{\partial b^{L-1}} = \frac{\partial z^{L-1}}{\partial b^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}} \frac{\partial z^L}{\partial a^{L-1}} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L} \tag{2.27}$$

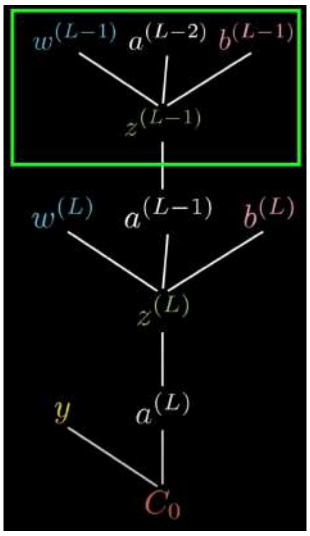


Figure 2.20: Chain scheme (b)

To generalise, in a real neural network we have more neurons per layer and more layers. The influence of weights and biases (through multiple paths) on cost function must be taken into account. The following expressions are the generalisation of the previous ones:

$$C_0 = \sum_{j=0}^{n_L - 1} (a_j^l - y_j)^2 \tag{2.28}$$

$$z_j^l = w_{j,0}^l a_0^{l-1} + w_{j,1}^l a_1^{l-1} + ... + b_j^l \tag{2.29}$$

$$a_j^l = \sigma(z_j^l) \tag{2.30}$$

$$\frac{\partial C_0}{\partial w_{jk}^L} = \frac{\partial z_j^L}{\partial w_{jk}^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial C_0}{\partial a_j^L} \tag{2.31}$$

$$\frac{\partial C_0}{\partial b_j^L} = \frac{\partial z_j^L}{\partial b_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial C_0}{\partial a_j^L} \tag{2.32}$$

The equations of backpropagation are:

$$\frac{\partial C_0}{\partial w_{jk}^l} = a_k^{l-1}\sigma'(z_j^l)\frac{\partial C}{\partial a_j^l} \tag{2.33}$$

$$\frac{\partial C_0}{\partial b_j^l} = \sigma'(z_j^l)\frac{\partial C}{\partial a_j^l} \tag{2.34}$$

with $l = 0, ...N-1$:

$$\frac{\partial C}{\partial a_j^l} = \begin{cases} \sum_{j=0}^{n_{l+1}-1} w_{jk}^{l+1}\sigma'(z_j^{l+1})\frac{\partial C}{\partial a_j^{l+1}} \\ \text{or} \\ \frac{\partial C}{\partial a_j^l} = 2(a_j^L - y_j) \end{cases} \tag{2.35}$$

Backpropagation results are then used by an

Coming back to the overall procedure, the aim of training algorithm is to minimise the cost C(w, b) as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm. There are different types of optimisers, we present only a few ones:

Gradient descent

The way the gradient descent algorithm works is to repeatedly compute the gradient of cost function (with backpropagation), and then to move in the opposite direction, falling down the slope of the valley. If we had only two variable, we can visualize it like this:

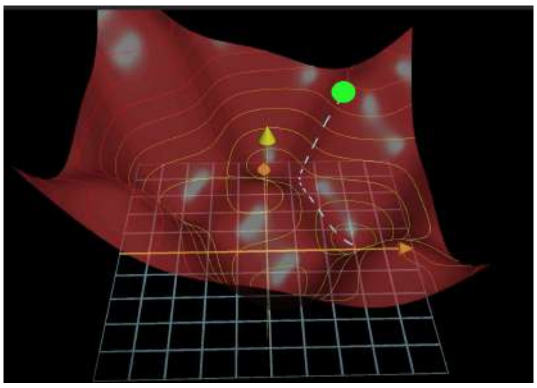


Figure 2.21: Gradient descent

We'll use Eq.2.36 to compute a value for $\Delta\nu$, then move the ball's position ν by that amount:

$$\nu \longmapsto \nu^{'} = \nu - \eta \nabla C \tag{2.36}$$

where η is learning rate: the smaller η is and more accurate we are. At the same time, we don't want η to be too small, since that will make the changes $\Delta\nu$ tiny, and thus the gradient descent algorithm will work very slowly.

Stochastic gradient descent

An idea called stochastic gradient descent can be used to speed up learning. The idea is to estimate the gradient ∇C by computing ∇C for a small sample of randomly chosen training inputs (mini-batch). By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning. A visual example in Fig.2.22.

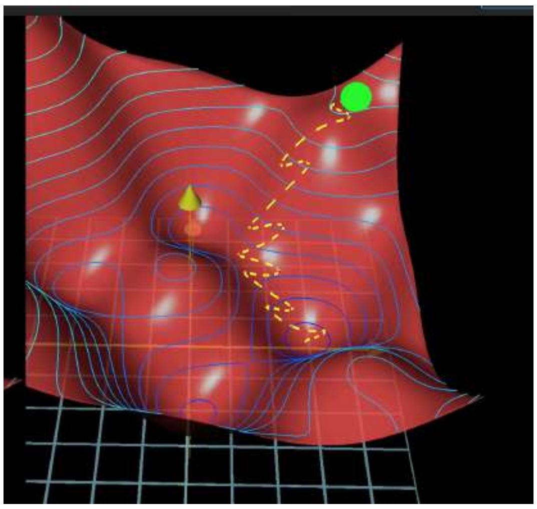


Figure 2.22: Stochastic gradient descent

It's possible to assert:

- SGD helps us to avoid the problem of local minima.
- SGD is much faster than Gradient Descent because it is running each row at a time and it doesn't have to load the whole data in memory for doing computation.
- SGD is generally noisier than typical Gradient Descent, it usually took a higher number of iterations to reach the minima, because of its randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent.

Adam

Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network.

Adam is combining the advantages of two other extensions of stochastic gradient descent. Specifically:

- Adaptive Gradient Algorithm (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).

- Root Mean Square Propagation (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

When Adam was first introduced, people got very excited about its power. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. Some very optimistic charts huge performance gains in terms of speed of training, one is the following:

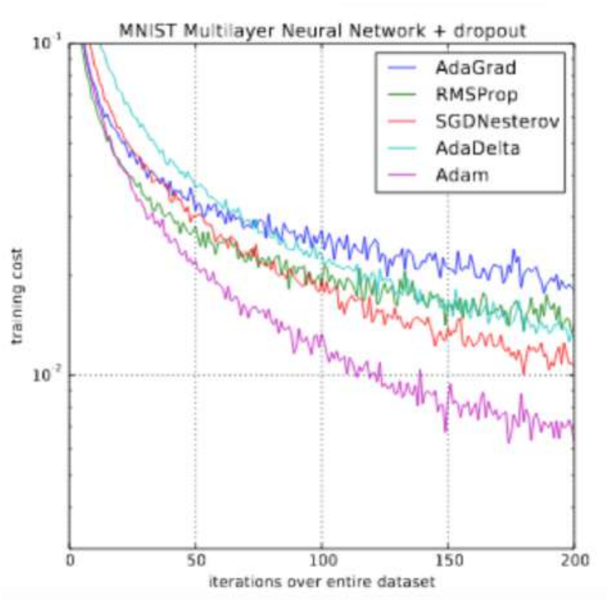


Figure 2.23: Performance chart

However, after a while people started noticing that despite superior training time, Adam in some areas does not converge to an optimal solution, so for some tasks. A lot of research has been done since to analyse the poor generalization of Adam trying to get it to close the gap with SGD.

2.5 Convolutional neural network

A variation of the neural networks is the convolution neural network. ConvNets, as they are sometimes known offer some significant advantages over normal neural nets. The traditional issue is that with big images, with many color channels, is that it quickly becomes computationally infeasible to train some models. What CNN tries to do is transform the images into a form which is easier to process, while still retaining the most important features. This is done by passing a filter over the initial image which conducts matrix (filter) multiplication over a subsection of the pixels in the initial image, it iterates through subsets until it has considered all subsets. The filter aims at capturing the most crucial features, while allowing the redundant features to be eliminated. This passing of a filter over the initial pixels is known as the Convolution Layer.

CNNs do take a biological inspiration from the visual cortex. The visual cortex has small regions of cells that are sensitive to specific regions of the visual field. This idea was expanded upon by a fascinating experiment by Hubel and Wiesel in 1962 where they showed that some individual neuronal cells in the brain responded (or fired) only in the presence of edges of a certain orientation. For example, some neurons fired when exposed to vertical edges and some when shown horizontal or diagonal edges. Hubel and Wiesel found out that all of these neurons were organized in a columnar architecture and that together, they were able to produce visual perception. This idea of specialized components inside of a system having specific tasks (the neuronal cells in the visual cortex looking for specific characteristics) is one that machines use as well, and is the basis behind CNNs. Following, a more detailed overview of what CNNs do.

The first layer in a CNN is always a Convolutional Layer. First thing to make sure you remember is what the input to this conv layer is. Like we mentioned before, the input is a (for example imagine a 2D picture with a dimension of 32 x 32 (x 3 color channel)) array of pixel values. This is easy to accept from the moment you understand that the computer sees the images as a set of numbers to which a color scale is then assigned (Fig. 2.24) .

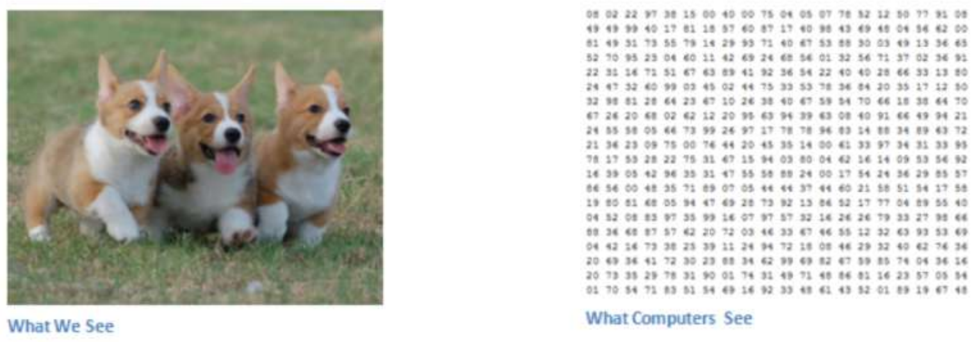


Figure 2.24: Pixel of an image

Now, the best way to explain a conv layer is to imagine a flashlight that is shining over the top left of the image. Let's say that the light this flashlight shines covers a 5 x 5 area. And now, let's imagine this flashlight sliding across all the areas of the input image. In machine learning terms, this flashlight is called a filter(or sometimes referred to as a neuron or a kernel) and the region that it is shining over is called the receptive field. Now this filter is also an array of numbers (the numbers are called weights or parameters). A very important note is that the depth of this filter has to be the same as the depth of the input, so the dimensions of this filter is 5 x 5 x 3. Now, let's take the first position the filter is in for example. It would be the top left corner. As the filter is sliding, or convolving, around the input image, it is multiplying the values in the filter with the original pixel values of the image (aka computing element wise multiplications). These multiplications are all summed up (mathematically speaking, this would be 75 multiplications in total). So now you have a single number. Remember, this number is just representative of when the filter is at the top left of the image. Now, we repeat this process for every location on the input volume. (Next step would be moving the filter to the right by 1 unit, then right again by 1, and so on). Every unique location on the input volume produces a number. After sliding the filter over all the locations, you will find out that what you're left with is a 28 x 28 x 1 array of numbers, which we call an activation map or feature map. The reason you get a 28 x 28 array is that there are 784 different locations that a 5 x 5 filter can fit on a 32 x 32 input image. These 784 numbers are mapped to a 28 x 28 array.

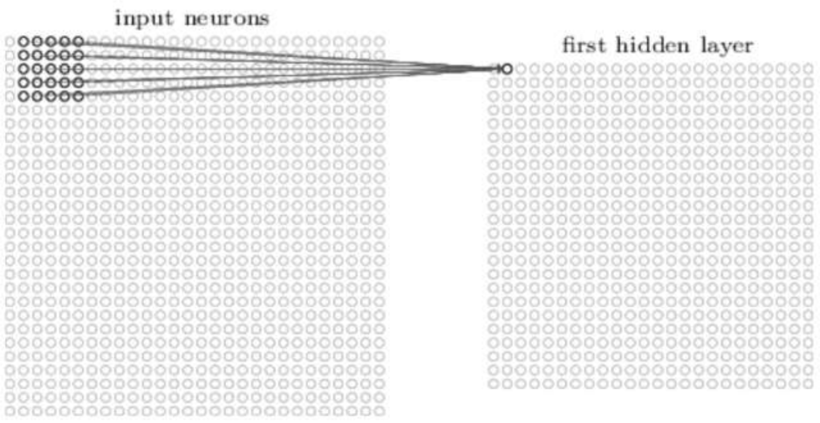
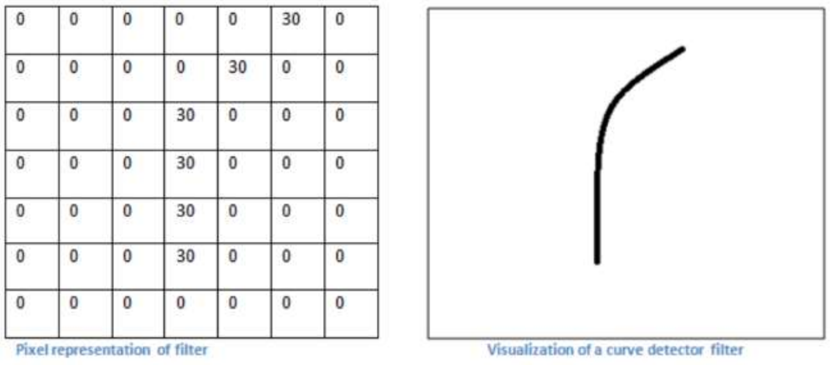


Figure 2.25: Conceptual schema of convolutional operation

Each of these filters can be thought of as feature identifiers like straight edges, simple colours, and curves. Think about the simplest characteristics that all images have in common with each other. Let's say our first filter is 7 x 7 x 3 and is going to be a curve detector. (ignore the fact that the filter is 3 units deep and only consider the top depth slice of the filter and the image, for simplicity.)As a curve detector, the filter will have a pixel structure in which there will be higher numerical values along the area that is a shape of a curve.

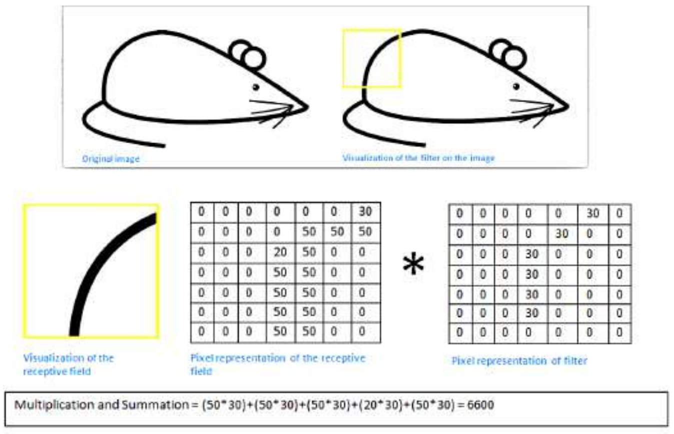


0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter
Visualization of a curve detector filter

Figure 2.26: Convolutional filter (a)

When we have this filter at the top left corner of the input volume, it is computing multiplications between the filter and pixel values at that region. Basically, in the input image, if there is a shape that generally resembles the curve that this filter is representing, then all of the multiplications summed together will result in a large value.



Original image
Visualization of the filter on the image
Visualization of the receptive field
Pixel representation of the receptive field
*
Pixel representation of filter

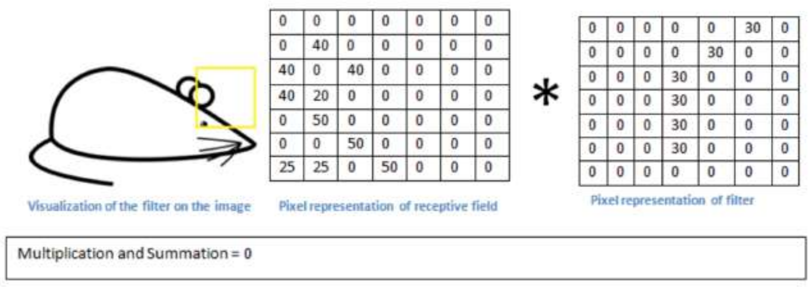
0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600

Figure 2.27: Multiplications between the filter and pixel values (a)

When the filter reaches another part of the image that does not have a feature similar to the filter, it happens as in the figure 2.28.


Visualization of the filter on the image
Pixel representation of receptive field
*
Pixel representation of filter
Multiplication and Summation = 0

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Figure 2.28: Multiplications between the filter and pixel values (b)

The value is much lower. This is because there wasn't anything in the image section that responded to the curve detector filter. The output of this conv layer is an activation map. So, in the simple case of a one filter convolution (and if that filter is a curve detector), the activation map will show the areas in which there at mostly likely to be curves in the picture. In this example, the top left value of our 26 x 26 x 1 activation map (26 because of the 7x7 filter instead of 5x5) will be 6600. This high value means that it is likely that there is some sort of curve in the input volume that caused the filter to activate. The top right value in our activation map will be 0 because there wasn't anything in the input volume that caused the filter to activate (or more simply said, there wasn't a curve in that region of the original image). This is just a filter that is going to detect lines that curve outward and to the right. We can have other filters for lines that curve to the left or for straight edges. The more filters, the greater the depth of the activation map, and the more information we have about the input volume.

Pooling layer

After the convolution layer comes the pooling layer, where the spatial size of the convoluted features will be attempted to be reduced. The reduction in complexity, sometimes known as dimensionality reduction will decrease the computational cost of performing analysis on the data set, allowing the method to be more robust. In this layer, a kernel once again passes over all subsets of pixels of the image. There are two types of pooling kernels which are commonly used. The first one is Max Pooling, which retains the maximum value of the subset. The alternative kernel is average pooling, which does exactly what you'd expect: it retains the average value of all the pixels in the subset. After the pooling phase, the information will hopefully be compressed enough to be used in a regular neural network model.

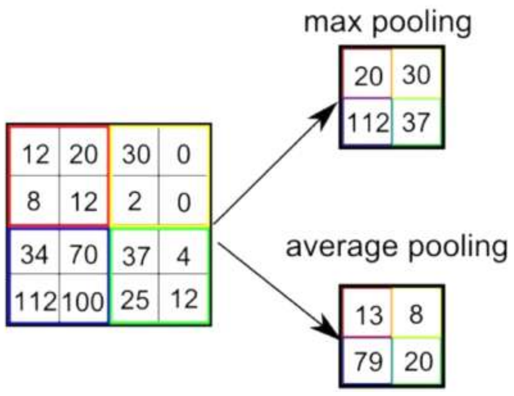


Figure 2.29: Pooling layer

Chapter 3

Test case

3.1 The furnace

The experimental facility consists of a 20 kW nominal power flameless unit, which has a configuration similar to industrial furnaces, in terms of injection profiles, air excess, fuel and air velocity and internal load. It has an integrated metallic finned heat exchanger to extract energy from the flue gases and pre-heat the combustion air. The chamber is made of stainless steel and has a cubic internal section of 700 mm on each side. It is equipped with a ceramic fiber insulation to reduce the heat loss and the external wall temperature of the combustion chamber. Fuel and air are fed co-axially into the combustion chamber through separated jets. The fuel is fed centrally, whereas air is fed through a coaxial gap (OD of 8.2 and 32 mm respectively). The burner can be characterized by a recirculation degree. The combustion chamber is equipped with an air cooling system consisting of four cooling tubes (OD 80 mm), with a length of 630 mm inside the furnace. Varying the air flow allows the combustion chamber to operate at different stable conditions, thus simulating the effect of a variable load. On each vertical wall of the combustion chamber, an opening is available for measurements. Two sides are equipped with a 150x150 mm quartz windows allowing optical access to the in order to detect the chemiluminescent self-emission of OH*. The window can be placed in up to four different positions along the opening length, allowing a complete access to the reactive zone for optical measurements. The openings on the other two sides are closed with an insulated plate coupled with six thermocouple ports, at a related distance of 100 mm. In particular, cold-junction compensated K-type thermocouples are used, in order to measure the wall temperature profile along the height of the furnace. Cooling air inlet/outlet, combustion air and flue gases temperatures are also measured by K-type thermocouples. The furnace temperature (T_f), used as set-point for the burner on/off regulation and the flue-gases temperature (T_{fg}) are given by two shielded N-type thermocouples positioned on the central plane and shifted of 250 mm respect the axis, on the top and bottom wall of the chamber, respectively. They are immersed 20 and 40 mm inside the flow field, respectively.

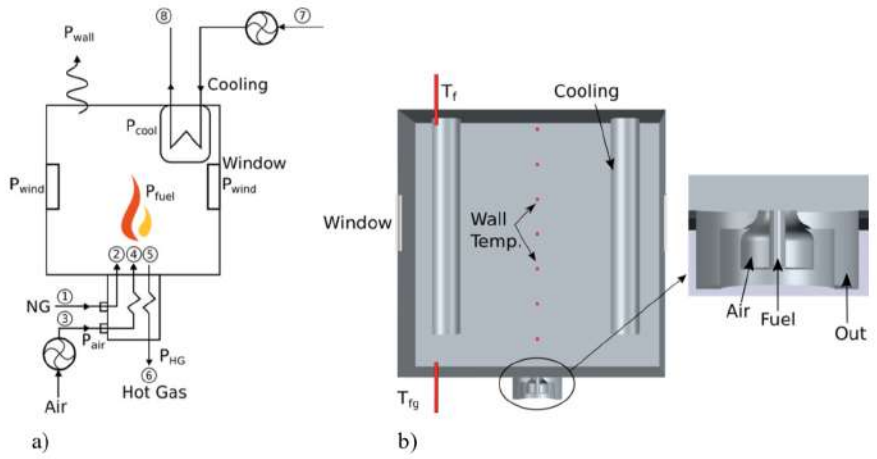


Figure 3.1: Plant scheme (a) and furnace 3D model (b) with positions of measuring

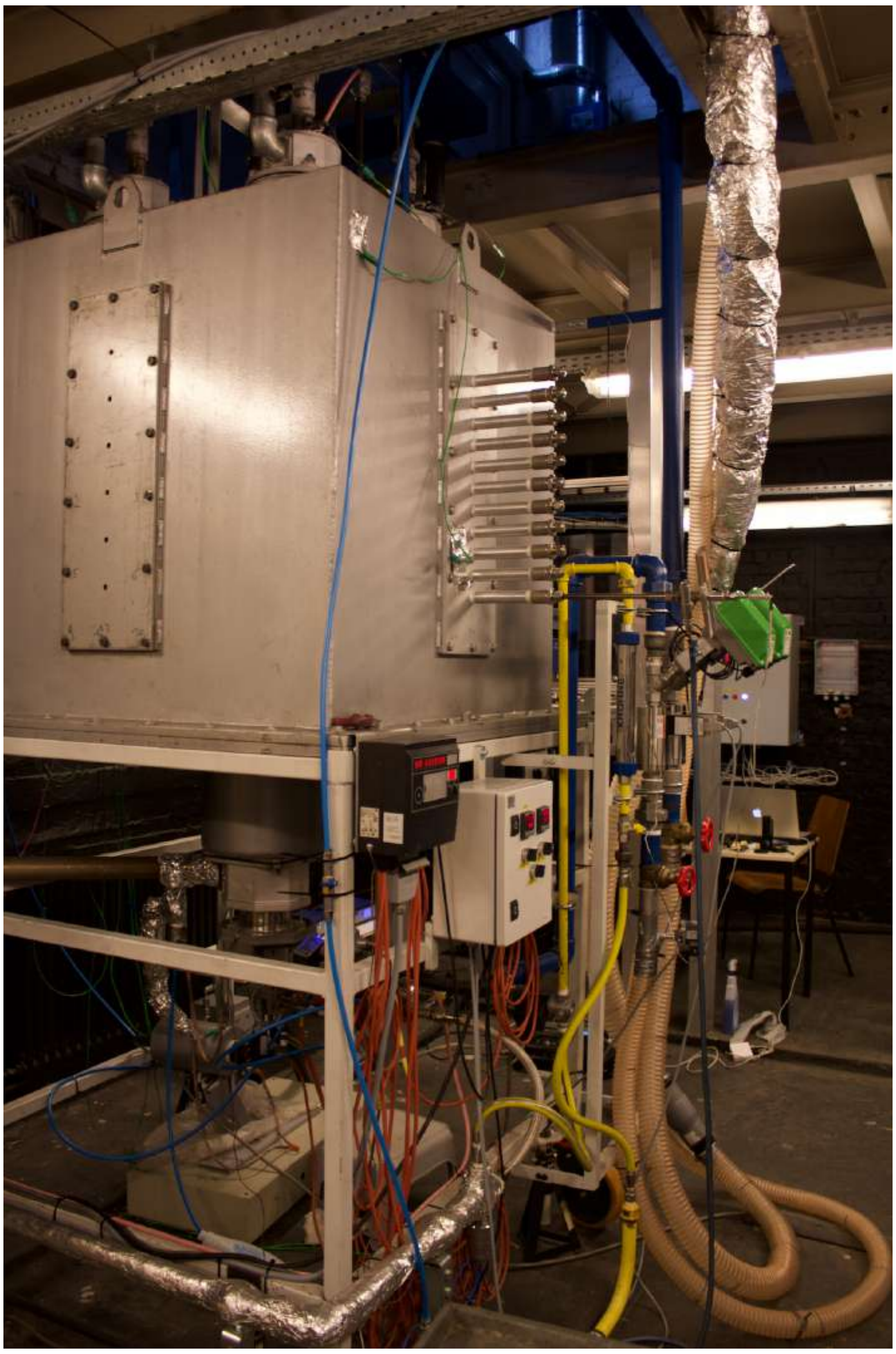


Figure 3.2: BURN group's Furnace

3.2 CFD of BURN's combustion chamber

Like said before, we need some start data-set useful to training the neural network and to validate it. To generate the samples, CFD simulations were carried out by BURN group using the commercial software Ansys Fluent 19.1. A constant heat power of 20 kW was fixed, while the cooling flow rate was set to reach a furnace outlet temperature of Tout=1000°C. Furthermore, the four sides of the furnace were closed with insulated plates. Moreover, a 45° degrees angular sector of the 3D geometry of the furnace was considered, as a result of the symmetry of the problem. The computational grid was first created with tetrahedrons and then converted into polyhedrons. The cooling and window surfaces are modelled as constant negative heat flux surfaces, whose values are set in accordance to the furnace energy balance, while on the lateral wall a conduction/convection condition towards the laboratory air is set. In figures 3.3 and 3.4, there is a representation of the 3D furnace mesh and of the only a 45 degrees angular sector considered in simulations.

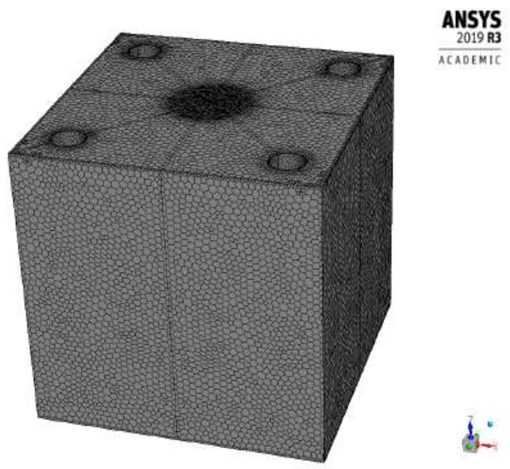


Figure 3.3: Complete 3D mesh

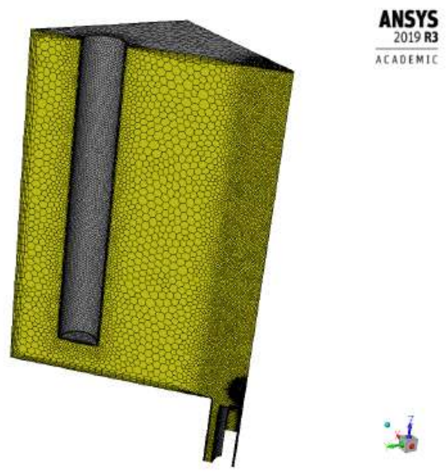


Figure 3.4: Considered domain

The standard k-ε model was used in combination with the PaSR model for turbulence-chemistry interactions. Following, a Cmix of 0.5 was set for the determination of an appropriate mixing scale in a static approach and a chemical time scale estimation from formation rates in the PaSR approach. A sensitivity study was carried out to select a kinetic scheme, comparing the KEE (17 species and 58 reactions) and GRI-2.11 (31 species and 175 reactions) mechanisms. Being the difference between the two schemes below 3% KEE was selected for its lower computational cost. The discrete ordinate (DO) radiation model was used, in combination with the weighted- sum-of-gray-gases (WSGG) model to take into account the radiation properties of the reacting mixture.

In the study three input parameters were considered. This means the digital twin will be able to provide results one time we define these three input. we don't need to choose these among a group of values but the continuity guaranteed by the model (ROM) allows us to choose freely (as far as admitted). The 3 input parameters considered to generate the simulation samples are:

- fuel composition in mole fractions (mixture of methane/hydrogen)
- equivalence ratio
- air injection geometry

A design of experiments (DoE) was established using latin hypercube sampling, varying the input parameters in the range 0-100 % (H2 molar fraction), 0.7-1 (equivalence ratio Φ) and 16-20-25 mm (air injector size). A total of 45 simulations were carried out. The variables of interest selected for the generation of the furnace ROM were the temperature, major species (CH4, H2, O2, H2O, OH), minor species (CO and OH), and pollutants (NO).

ID sim	**air mm**	**x fuel**	**eq ratio**	**ID sim**	**air mm**	**x fuel**	**eq ratio**
12	16	0.60	0.93	2	20	0.45	0.80
17	16	0.50	0.99	30	20	0.20	0.80
18	16	0.10	0.94	3	20	0.90	0.85
23	16	0.30	0.78	5	20	0	0.72
24	16	0.05	0.83	7	20	0.45	0.72
31	16	0.60	0.75	43	20	0.50	0.95
34	16	0.40	0.90	44	20	0.17	0.73
38	16	0.80	0.96	45	20	0.90	0.70
40	16	0.20	0.73	15	25	0.90	0.74
41	16	0.90	0.87	21	25	0.95	0.90
42	16	0.30	0.97	25	25	0.80	0.76
4	16	0.55	0.86	26	25	0.35	0.73
9	16	0.70	0.78	27	25	0.20	0.76
10	20	0.75	0.98	29	25	0.25	0.85

11	20	0.30	0.89	32	25	0.40	0.95
13	20	0.95	0.96	33	25	0.80	0.90
14	20	0.10	0.88	37	25	0.65	0.84
16	20	0.85	0.95	39	25	0.45	0.85
19	20	0.15	0.97	6	25	0.70	0.94
1	20	0.60	0.88	8	25	0.65	0.91
20	20	0.35	0.70	46	25	0.11	0.96
22	20	0.80	0.82	47	25	0.54	0.74
28	20	0.50	0.83				

Table 3.1: DoE

3.3 Preprocessing

In this section we will see how, starting from the dataset provided by the Burn research group, the appropriate input to the convolutional neural network is obtained. In particular, CNN requires in input a matrix of pixels representative of the CFD simulation. We therefore choose to use a single channel color representation, this will be directly expressed with the values of the variable field in the mesh with appropriate normalization.

3.3.1 Dataset

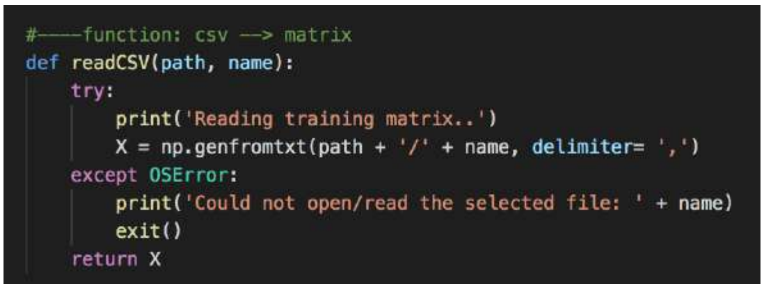
The Burn group of the Libre University carried out 45 CFD simulations of MILD combustion and validated thanks to experimental tests performed on the furnace described above. As already mentioned the simulations differ for the variation of at least one of these three parameters: fuel composition in mole fractions (mixture of methane / hydrogen), equivalence ratio and air injection geometry. See table: 3.1.

Due to the variation of the *air injection geometry* parameter, the research team was forced to create three different meshes so that they are all sufficiently accurate. But the results from which our project starts see a unified mesh and the coordinates associated with it are provided as a csv file in x, y, z. The fields of the calculated variables are also provided through a csv file. In particular, each row corresponds to a CFD simulation. It contains the values of the variables in succession for each grid point (in total 216360). The header row is shown as an example: $T_1, T_2, T_3.....T_{216360}, CH4_1, ...CH4_{216360}$, etc

3.3.2 Code for preprocessing

Preprocessing was done by writing two python scripts to be used in succession. In this section we will only see some glimpses reported but the complete code is available in the appendix B.

Let's have a look to the first script (appendix B.1.1). First of all, every time it will be necessary to import data from a csv file and convert them into a matrix, the function in the figure 3.5 will be used.



```
#----function: csv --> matrix
def readCSV(path, name):
    try:
        print('Reading training matrix..')
        X = np.genfromtxt(path + '/' + name, delimiter= ',')
    except OSError:
        print('Could not open/read the selected file: ' + name)
        exit()
    return X
```

Figure 3.5: Function to import csv

It can be seen from the first graphical representations of the temperature field (as well as of the other variables) that there are clearly incorrect grid points saved in the csv file (see Figure 3.6) because it is physically impossible that there is a discontinuity (from 400 to 1200 degrees) in furnace. Since it was not possible to trace the error, here we have chosen to delete the points with incorrect values from the domain.

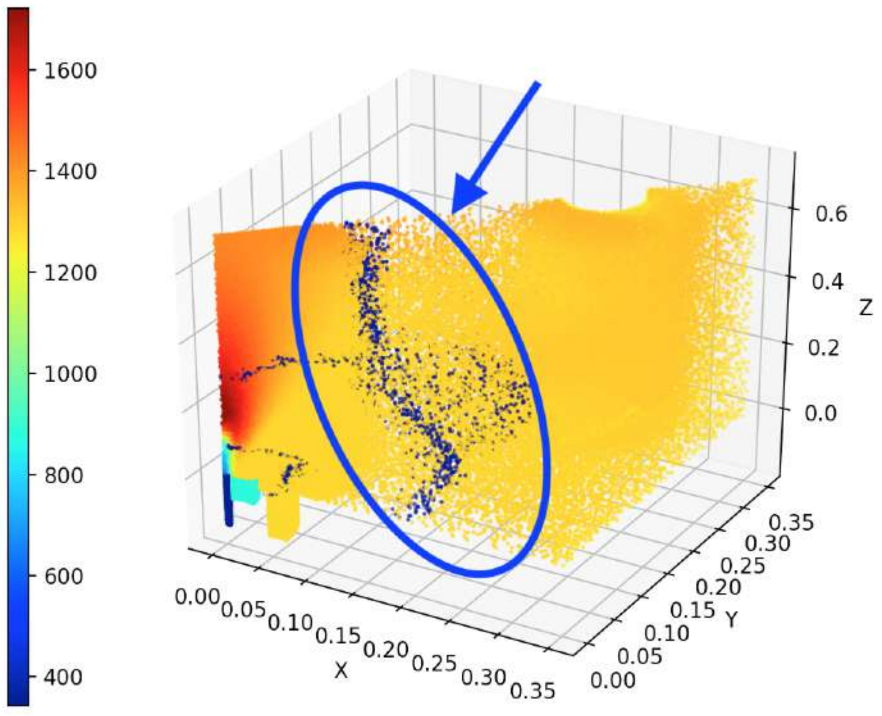
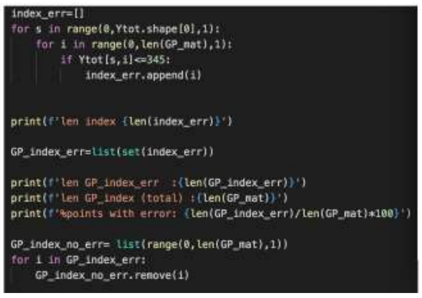


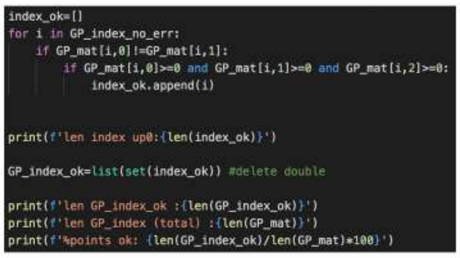
Figure 3.6: Example of an original temperature field, simulation 1

To discern between the grid points, reference was made to the temperature field where the diversity is particularly evident and we deleted all values under 345 K (code in Fig. 3.7 (a)).

Once this is done, the points that belong to the symmetry plane and the points that are outside the study domain are eliminated. In particular, the points with negative coordinates are deleted (Code in Fig 3.7 (b)).



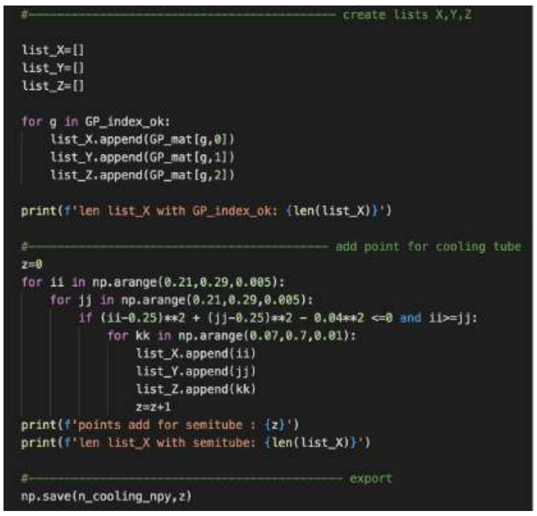
```python
index_err=[]
for s in range(0,Ytot.shape[0],1):
    for i in range(0,len(GP_mat),1):
        if Ytot[s,i]<=345:
            index_err.append(i)


print(f'len index {len(index_err)}')

GP_index_err=list(set(index_err))

print(f'len GP_index_err  :{len(GP_index_err)}')
print(f'len GP_index (total) :{len(GP_mat)}')
print(f'%points with error: {len(GP_index_err)/len(GP_mat)*100}')

GP_index_no_err= list(range(0,len(GP_mat),1))
for i in GP_index_err:
    GP_index_no_err.remove(i)
```



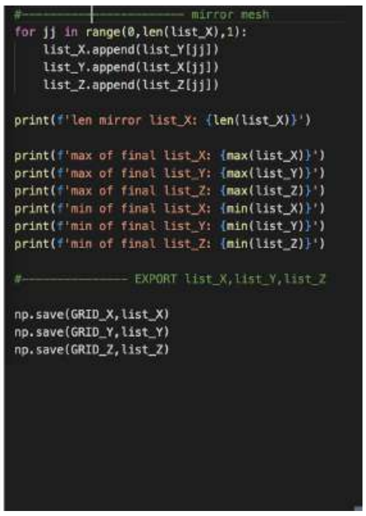
```python
index_ok=[]
for i in GP_index_no_err:
    if GP_mat[i,0]!=GP_mat[i,1]:
        if GP_mat[i,0]>=0 and GP_mat[i,1]>=0 and GP_mat[i,2]>=0:
            index_ok.append(i)


print(f'len index up0:{len(index_ok)}')

GP_index_ok=list(set(index_ok)) #delete double

print(f'len GP_index_ok :{len(GP_index_ok)}')
print(f'len GP_index (total) :{len(GP_mat)}')
print(f'%points ok: {len(GP_index_ok)/len(GP_mat)*100}')
```

Figure 3.7: First script(a) and (b)

After carrying out these actions, we obtain a list of grid points that we consider valid for our study (this list is saved with the name GPindexok). Three lists (X, Y, Z) of the coordinates of the above mentioned grid points are created. These are not yet complete, in fact the coordinates of the points that fill the half pipe excluded from the CFD simulations are added (the number of grid points added z is also saved) (Code in Fig 3.8 (c)). Now it is necessary to mirror the grid so as to obtain a new one (representing a quarter of the furnace) and updating the X, Y, Z lists (Code in Fig 3.8 (d)). The latter are then saved.



```python
#------------------------------------------------ create lists X,Y,Z

list_X=[]
list_Y=[]
list_Z=[]

for g in GP_index_ok:
    list_X.append(GP_mat[g,0])
    list_Y.append(GP_mat[g,1])
    list_Z.append(GP_mat[g,2])

print(f'len list_X with GP_index_ok: {len(list_X)}')

#----------------------------------------------- add point for cooling tube
z=0
for ii in np.arange(0.21,0.29,0.005):
    for jj in np.arange(0.21,0.29,0.005):
        if (ii-0.25)**2 + (jj-0.25)**2 - 0.04**2 <=0 and ii>=jj:
            for kk in np.arange(0.07,0.7,0.01):
                list_X.append(ii)
                list_Y.append(jj)
                list_Z.append(kk)
                z=z+1
print(f'points add for semitube : {z}')
print(f'len list_X with semitube: {len(list_X)}')

#------------------------------------------------ export
np.save(n_cooling_npy,z)
```

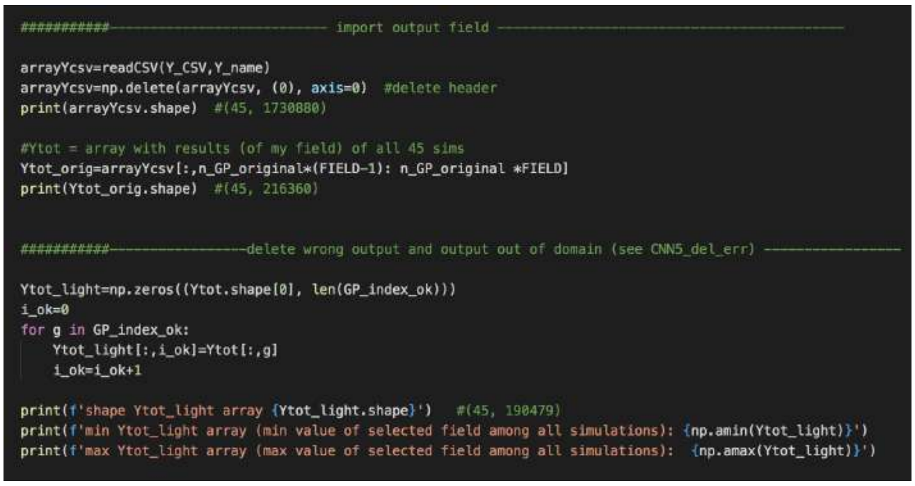


```python
#----------------------------- mirror mesh
for jj in range(0,len(list_X),1):
    list_X.append(list_Y[jj])
    list_Y.append(list_X[jj])
    list_Z.append(list_Z[jj])

print(f'len mirror list_X: {len(list_X)}')

print(f'max of final list_X: {max(list_X)}')
print(f'max of final list_Y: {max(list_Y)}')
print(f'max of final list_Z: {max(list_Z)}')
print(f'min of final list_X: {min(list_X)}')
print(f'min of final list_Y: {min(list_Y)}')
print(f'min of final list_Z: {min(list_Z)}')

#--------------- EXPORT list_X,list_Y,list_Z

np.save(GRID_X,list_X)
np.save(GRID_Y,list_Y)
np.save(GRID_Z,list_Z)
```

Figure 3.8: First script(c) and (d)

With the script described above we worked on the grid points. Now we will work on the values of the variable fields in order to obtain the input matrix to CNN. Also in this second script (appendix B.1.2) the function that translates csv into matrices is used and the results obtained from the previous script are exploited, i.e. the following are imported: the list of approved grid points indices, the number of points added for the half pipe and lists X, Y, Z.

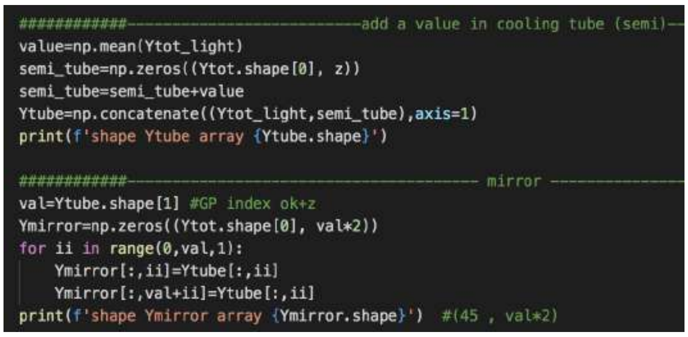
Once the matrix containing the values of the selected variable for all 45 simulations has been imported, we eliminate the values corresponding to the erroneous grid points and those outside the domain with the following script part.



```
###########--------------------------- import output field ---------------------------------------------

arrayYcsv=readCSV(Y_CSV,Y_name)
arrayYcsv=np.delete(arrayYcsv, (0), axis=0)  #delete header
print(arrayYcsv.shape)  #(45, 1730880)

#Ytot = array with results (of my field) of all 45 sims
Ytot_orig=arrayYcsv[:,n_GP_original*(FIELD-1): n_GP_original *FIELD]
print(Ytot_orig.shape)  #(45, 216360)


###########-----------------delete wrong output and output out of domain (see CNN5_del_err) -----------------

Ytot_light=np.zeros((Ytot.shape[0], len(GP_index_ok)))
i_ok=0
for g in GP_index_ok:
    Ytot_light[:,i_ok]=Ytot[:,g]
    i_ok=i_ok+1

print(f'shape Ytot_light array {Ytot_light.shape}')   #(45, 190479)
print(f'min Ytot_light array (min value of selected field among all simulations): {np.amin(Ytot_light)}')
print(f'max Ytot_light array (max value of selected field among all simulations):  {np.amax(Ytot_light)}')
```

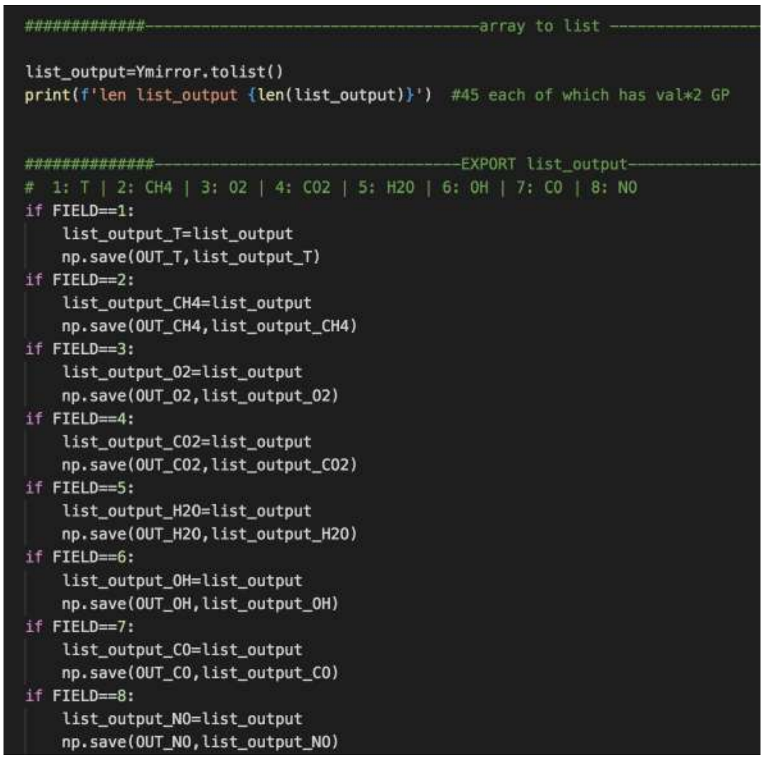
Figure 3.9: Second script (a)

After doing this, let's add a uniform value to the cooling half-pipe. Since the *griddata* function that we will use later would be disturbed by an evident discontinuity between the field of the variable in the furnace and the value linked to the cooling tube and since it has been chosen to have a temperature of cooling tube uniform with that on the wall of the same, we have assigned as arbitrary value the average of the values in the furnace which we expect to be very close to that of the pipe wall. The variable field must therefore be mirrored in accordance with the values of the new grid representing a quarter of the furnace (Code in Fig 3.10).



```
############-------------------------add a value in cooling tube (semi)--
value=np.mean(Ytot_light)
semi_tube=np.zeros((Ytot.shape[0], z))
semi_tube=semi_tube+value
Ytube=np.concatenate((Ytot_light,semi_tube),axis=1)
print(f'shape Ytube array {Ytube.shape}')

############-------------------------------------- mirror --------------
val=Ytube.shape[1] #GP index ok+z
Ymirror=np.zeros((Ytot.shape[0], val*2))
for ii in range(0,val,1):
    Ymirror[:,ii]=Ytube[:,ii]
    Ymirror[:,val+ii]=Ytube[:,ii]
print(f'shape Ymirror array {Ymirror.shape}')  #(45 , val*2)
```

Figure 3.10: Second script (b)



```
#############-----------------------------------array to list ----------------

list_output=Ymirror.tolist()
print(f'len list_output {len(list_output)}')  #45 each of which has val*2 GP


##############-------------------------------EXPORT list_output--------------
#  1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
if FIELD==1:
    list_output_T=list_output
    np.save(OUT_T,list_output_T)
if FIELD==2:
    list_output_CH4=list_output
    np.save(OUT_CH4,list_output_CH4)
if FIELD==3:
    list_output_O2=list_output
    np.save(OUT_O2,list_output_O2)
if FIELD==4:
    list_output_CO2=list_output
    np.save(OUT_CO2,list_output_CO2)
if FIELD==5:
    list_output_H2O=list_output
    np.save(OUT_H2O,list_output_H2O)
if FIELD==6:
    list_output_OH=list_output
    np.save(OUT_OH,list_output_OH)
if FIELD==7:
    list_output_CO=list_output
    np.save(OUT_CO,list_output_CO)
if FIELD==8:
    list_output_NO=list_output
    np.save(OUT_NO,list_output_NO)
```

Figure 3.11: Second script (c)

The obtained matrix is specific for the selected field and has a shape of (n ° sims, n ° grid points). For future purposes it is saved (Code in fig 3.11). To be more clear about current situation, have a look to the 3D plot of the matrix refereed to temperature field.

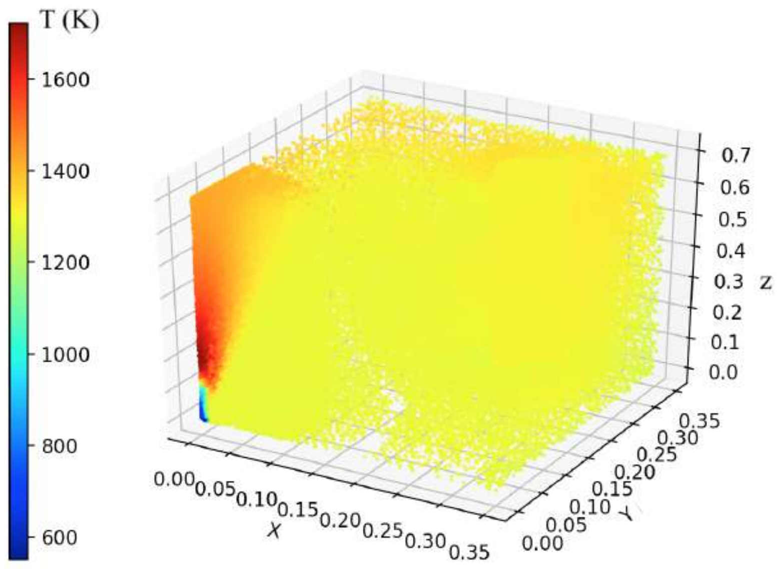
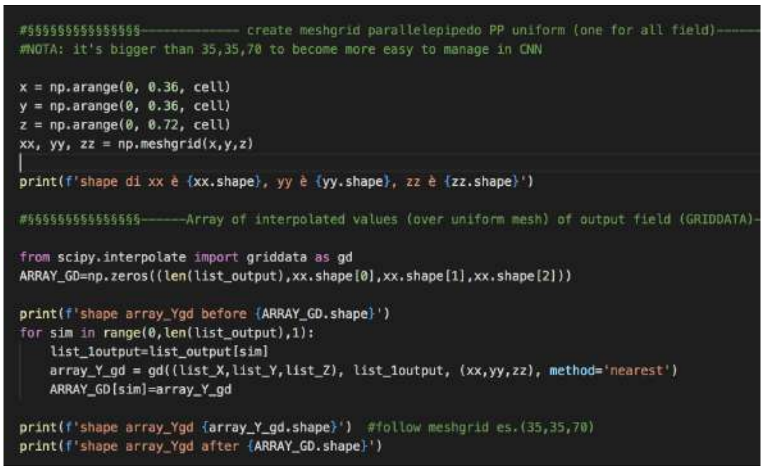


Figure 3.12: Original mesh (plot of temperature field simulation 1)

We have achieved the first fundamental goal of preprocessing. But list output refers to a non-uniform mesh with 393 810 grid points. The convolutional neural network requires a pixel matrix as input, so we need to create a uniform parallelepiped mesh (as a matrix). The fineness of the cell has been chosen so that the number of grid points is similar to the original one, to do this cell parameter is fixed to 0.0067 (314 920 grid points) and to cover all the quarter of furnace dimension are (0.35/0.35/0.72 meters). To built the new mesh we use *meshgrid* function while we use *griddata* function to assign the value of the CFD output variable to each point of the new grid. In griddata the nearest method is applied, this returns the value at the data point closest to the point of interpolation. Then matrix containing interpolated values on the new mesh of the variable selected for the 45 simulations is saved. In fig 3.14 you find a 3D plot of the results of griddata function. Plots, shown in this section,are created through the run of script in appendix B.1.3.



```
#§§§§§§§§§§§§§§§-------------- create meshgrid parallelepipedo PP uniform (one for all field)------
#NOTA: it's bigger than 35,35,70 to become more easy to manage in CNN

x = np.arange(0, 0.36, cell)
y = np.arange(0, 0.36, cell)
z = np.arange(0, 0.72, cell)
xx, yy, zz = np.meshgrid(x,y,z)

print(f'shape di xx è {xx.shape}, yy è {yy.shape}, zz è {zz.shape}')

#§§§§§§§§§§§§§§§------Array of interpolated values (over uniform mesh) of output field (GRIDDATA)-

from scipy.interpolate import griddata as gd
ARRAY_GD=np.zeros((len(list_output),xx.shape[0],xx.shape[1],xx.shape[2]))

print(f'shape array_Ygd before {ARRAY_GD.shape}')
for sim in range(0,len(list_output),1):
    list_1output=list_output[sim]
    array_Y_gd = gd((list_X,list_Y,list_Z), list_1output, (xx,yy,zz), method='nearest')
    ARRAY_GD[sim]=array_Y_gd

print(f'shape array_Ygd {array_Y_gd.shape}')  #follow meshgrid es.(35,35,70)
print(f'shape array_Ygd after {ARRAY_GD.shape}')
```

Figure 3.13: Second script (d)

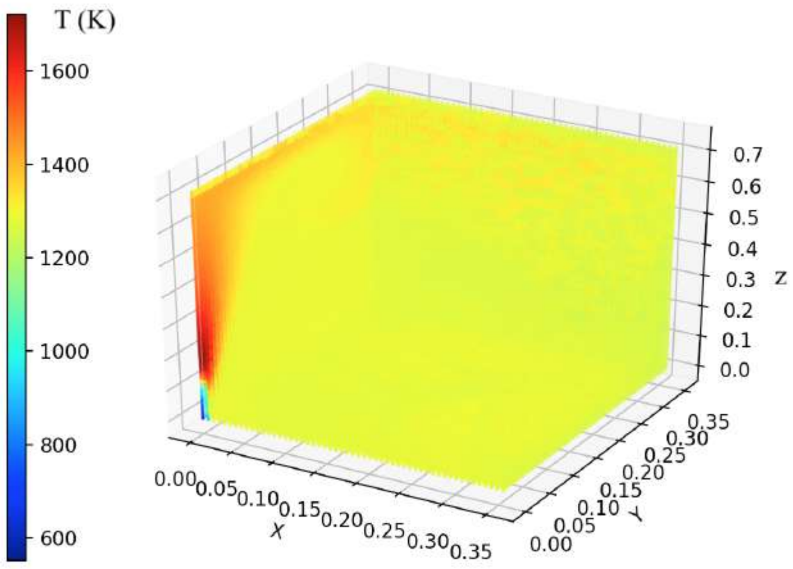
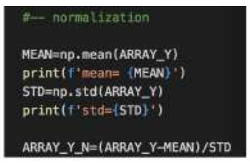


Figure 3.14: Griddata - 0.0067 mesh

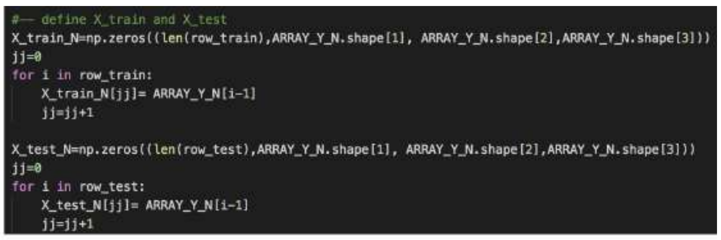
3.4 The core of the project: CNN

To develop our neural network we need two scripts that you will find in the appendix B.2.1 and B.3.1. First lines of the script CNN5.py have purpose of avoiding the randomness associated with the packages used, for this purpose we set the seed of the python numpy package to a value (in our case 9). Thanks to the next chapter it will be clear to you why in this chapter there are already proposed values for the parameters over which the programmer has power, for a brief preview these are the values obtained following an optimization process.

Field values of the selected variable (in our case temperature) are imported through the file created by the script CNN5read.py. These need a normalization to help give more accurate results (for example the one defined as C which subtracts the mean and divides by the standard deviation). (Code in fig. 3.15 (a))



```
#-- normalization

MEAN=np.mean(ARRAY_Y)
print(f'mean= {MEAN}')
STD=np.std(ARRAY_Y)
print(f'std={STD}')

ARRAY_Y_N=(ARRAY_Y-MEAN)/STD
```

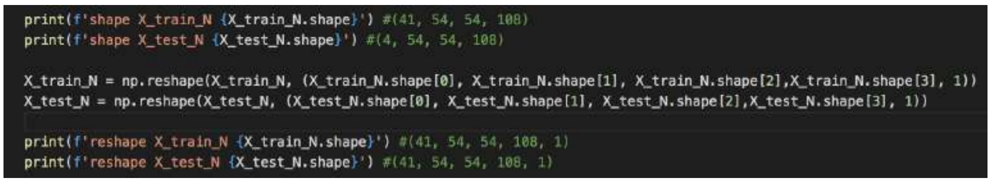


```
#-- define X_train and X_test
X_train_N=np.zeros((len(row_train),ARRAY_Y_N.shape[1], ARRAY_Y_N.shape[2],ARRAY_Y_N.shape[3]))
jj=0
for i in row_train:
    X_train_N[jj]= ARRAY_Y_N[i-1]
    jj=jj+1

X_test_N=np.zeros((len(row_test),ARRAY_Y_N.shape[1], ARRAY_Y_N.shape[2],ARRAY_Y_N.shape[3]))
jj=0
for i in row_test:
    X_test_N[jj]= ARRAY_Y_N[i-1]
    jj=jj+1
```

Figure 3.15: Third script (a) and (b)

As already mentioned, we have 45 CFD simulations available. To be consistent with the work done by the BURN group of the Libre university, only 41 of these will be used to train our neural network, while the other 4 we will be used to test the performance of the network already created. The identification IDs of the simulations belonging to the test category are: 1,22,28,39.

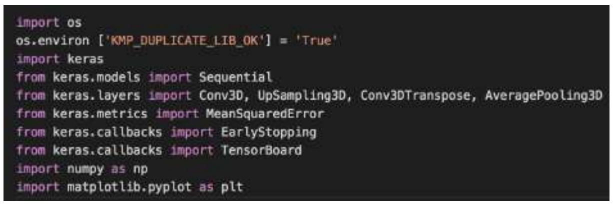
The code in Fig 3.15 (b) is use to divide in two different array the original array of the variable values in accord to the two previous categories. Then the neural network wants that the input array shape explicit the number of channels. To do this we use the reshape function (code in fig 3.16). In our case image pixels are represented by a single number, that we identify with the value of the variable in that grid point but there are other cases (called rgb) in which three channels are used to describe the colour.



```
print(f'shape X_train_N {X_train_N.shape}') #(41, 54, 54, 108)
print(f'shape X_test_N {X_test_N.shape}') #(4, 54, 54, 108)

X_train_N = np.reshape(X_train_N, (X_train_N.shape[0], X_train_N.shape[1], X_train_N.shape[2],X_train_N.shape[3], 1))
X_test_N = np.reshape(X_test_N, (X_test_N.shape[0], X_test_N.shape[1], X_test_N.shape[2],X_test_N.shape[3], 1))

print(f'reshape X_train_N {X_train_N.shape}') #(41, 54, 54, 108, 1)
print(f'reshape X_test_N {X_test_N.shape}') #(41, 54, 54, 108, 1)
```

Figure 3.16: Third script (c)

Now let's get to the heart of our neural network. We rely on the KERAS library for the construction and training of this. Some packages are then imported (code in fig. 3.17).

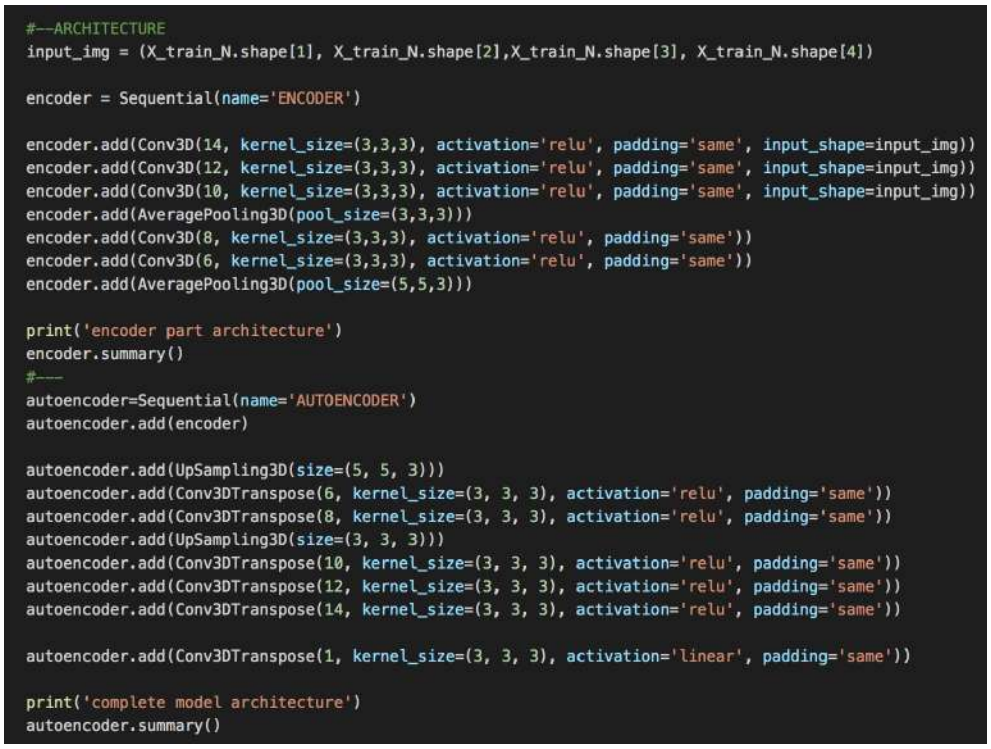


```
import os
os.environ ['KMP_DUPLICATE_LIB_OK'] = 'True'
import keras
from keras.models import Sequential
from keras.layers import Conv3D, UpSampling3D, Conv3DTranspose, AveragePooling3D
from keras.metrics import MeanSquaredError
from keras.callbacks import EarlyStopping
from keras.callbacks import TensorBoard
import numpy as np
import matplotlib.pyplot as plt
```

Figure 3.17: Third script (d)

The next step consists in defining neural network architecture. The first part of the autoencoder will be called ENCODER and it will be developed and saved separately in order to be implemented and to obtain the CODE i.e. the reduced version of the input image. On the whole, however, the architecture takes the name of AUTOENCODER and thanks to the implementation of this, you get the DECODED i.e. the reconstructed version of the input image, this will be compared to the original at each epoch (for training) or each prediction (for test) with the aim of evaluate model accuracy.

The network is created with the Sequential type, this means that it will be a succession of layers that we are going to add progressively with the .add command. As it is easy to notice in the code in fig. 3.18 this succession is constituted for the encoder by layers of conv3D and AveragePooling3D while the respective opposites (Conv3DTranspose and UpSampling3D) are found in the second half i.e. the reconstruction.



```
#--ARCHITECTURE
input_img = (X_train_N.shape[1], X_train_N.shape[2],X_train_N.shape[3], X_train_N.shape[4])

encoder = Sequential(name='ENCODER')

encoder.add(Conv3D(14, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(Conv3D(12, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(Conv3D(10, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(AveragePooling3D(pool_size=(3,3,3)))
encoder.add(Conv3D(8, kernel_size=(3,3,3), activation='relu', padding='same'))
encoder.add(Conv3D(6, kernel_size=(3,3,3), activation='relu', padding='same'))
encoder.add(AveragePooling3D(pool_size=(5,5,3)))

print('encoder part architecture')
encoder.summary()
#---
autoencoder=Sequential(name='AUTOENCODER')
autoencoder.add(encoder)

autoencoder.add(UpSampling3D(size=(5, 5, 3)))
autoencoder.add(Conv3DTranspose(6, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(8, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(UpSampling3D(size=(3, 3, 3)))
autoencoder.add(Conv3DTranspose(10, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(12, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(14, kernel_size=(3, 3, 3), activation='relu', padding='same'))

autoencoder.add(Conv3DTranspose(1, kernel_size=(3, 3, 3), activation='linear', padding='same'))

print('complete model architecture')
autoencoder.summary()
```

Figure 3.18: Third script (e)

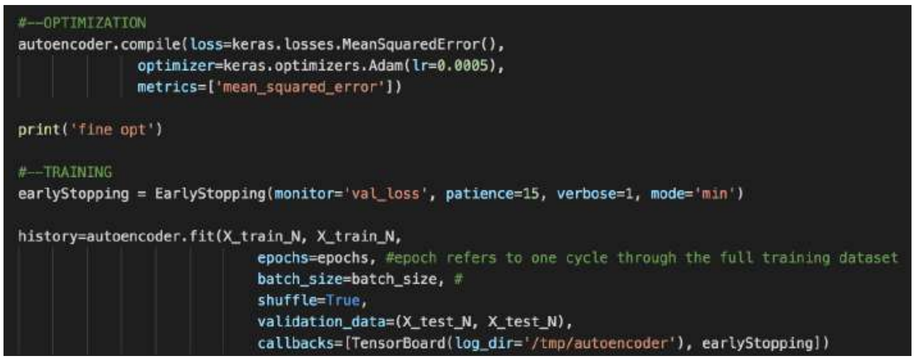
Let's focus on the topics of the individual layers. For the Pooling layers, we have only to decide the pooling size, because we have already decided Average or Max. For the Conv3D layer you will find:

- numbers of filters in a single conv layer. With more filters we usually have more accuracy.
- size of the kernel (if the input imagine is 3D, the size must be a vector with 3 elements, so a cube)
- activation function. There are different activation function as already anticipated (see section 2.2), we choose Relu.
- paddig: Same padding means the size of output feature-maps are the same as the input feature-maps
- input shape

In the next part of the script there is network training. In *.compile* we define:

- the loss function, in our case the mean squared error (see section 2.4.1)
- the type of optimizer used (see section 2.4.2)
- metrics: function that is used to judge the performance of your model. In general, it could have nothing to do with the minimization performed by the optimizer.

In .fit we explain what the inputs are and what the outputs must be compared with (whether they are for training or for testing). Being an autoencoder, the reconstructed image must be compared with the original. Then we decide batch size (i.e. how many sample pass before updating gradient) and epochs (how many cycle with full training dataset execute before stop the training). Because find exactly when to stop the iteration process is not easy (we could do error of over-fitting for example...), we decide to use the function earlystopping. This function monitor a quantity and wait a number of epochs (patient) with no improvement after which training will be stopped. In min mode, training will stop when the quantity monitored has stopped decreasing. In .fit if shuffe is true: it will shuffle your entire dataset first and then make batches according to the batch size.



```
#--OPTIMIZATION
autoencoder.compile(loss=keras.losses.MeanSquaredError(),
                    optimizer=keras.optimizers.Adam(lr=0.0005),
                    metrics=['mean_squared_error'])

print('fine opt')

#--TRAINING
earlyStopping = EarlyStopping(monitor='val_loss', patience=15, verbose=1, mode='min')

history=autoencoder.fit(X_train_N, X_train_N,
                                    epochs=epochs, #epoch refers to one cycle through the full training dataset
                                    batch_size=batch_size, #
                                    shuffle=True,
                                    validation_data=(X_test_N, X_test_N),
                                    callbacks=[TensorBoard(log_dir='/tmp/autoencoder'), earlyStopping])
```

Figure 3.19: Third script (f)

In the final part of this script there is the logarithm plot of both mean squarred error and the value of the loss function for train and test. A typical result in fig. 3.20. In our case they are the same thing.

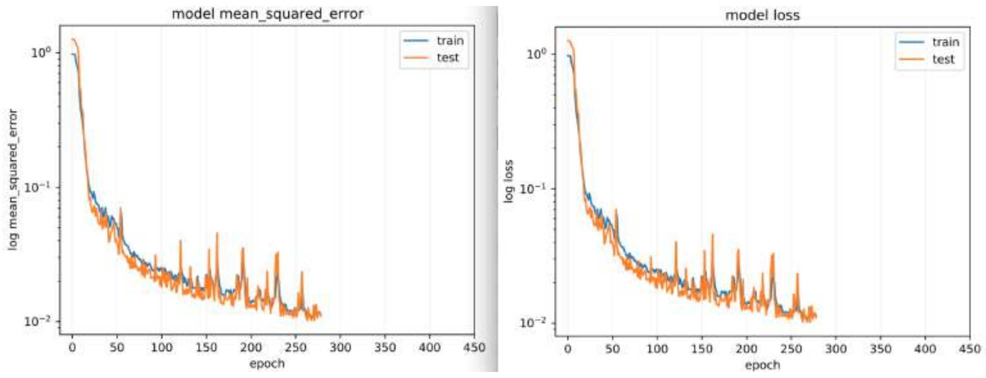
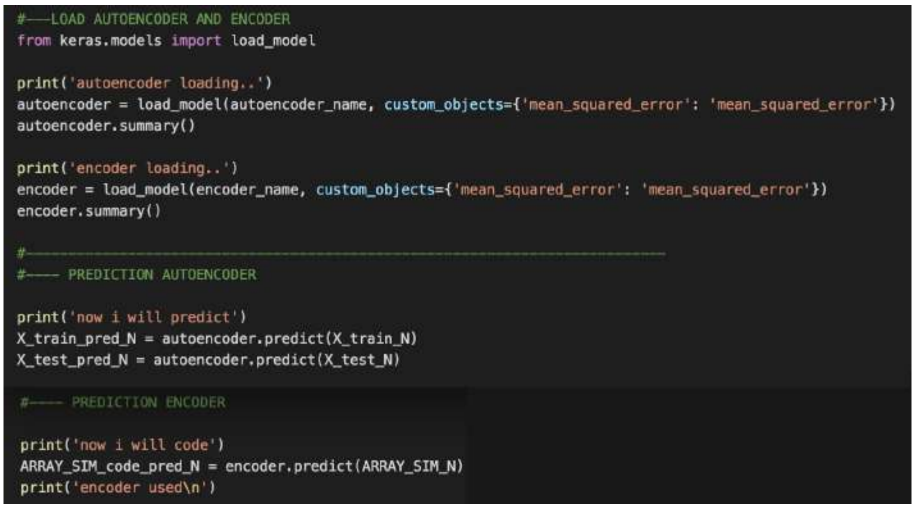


Figure 3.20: Result of third script (g)

Now that we have created and trained the autoencoder we can use a second script (appendix B.3.1) for prediction and post-processing. After loading the array of variable values, we normalize it and extract XtrainN and XtestN once again. The autoencoder and encoder model are loaded and the prediction is performed. As in fig. 3.21. It is added that the prediction was also made for Xtrain but the only one to have importance for the results and future comparisons is that of Xtest, since only in this last case the data were not used for training. The same thing is possible to do with encoder, obtaining the code.



```
#---LOAD AUTOENCODER AND ENCODER
from keras.models import load_model

print('autoencoder loading..')
autoencoder = load_model(autoencoder_name, custom_objects={'mean_squared_error': 'mean_squared_error'})
autoencoder.summary()

print('encoder loading..')
encoder = load_model(encoder_name, custom_objects={'mean_squared_error': 'mean_squared_error'})
encoder.summary()

#------------------------------------------------------------------------------------
#---- PREDICTION AUTOENCODER

print('now i will predict')
X_train_pred_N = autoencoder.predict(X_train_N)
X_test_pred_N = autoencoder.predict(X_test_N)

#---- PREDICTION ENCODER

print('now i will code')
ARRAY_SIM_code_pred_N = encoder.predict(ARRAY_SIM_N)
print('encoder used\n')
```

Figure 3.21: Fourth script (a)

3.5 Post-processing

The performed predictions can be plotted to have a visual representation of the reconstruction. The figure shows the simulation 22 (a zoomed version of the mesh) in the original and reconstructed version. The plots are obtained with the compilation of the script in the appendix B.3.2.

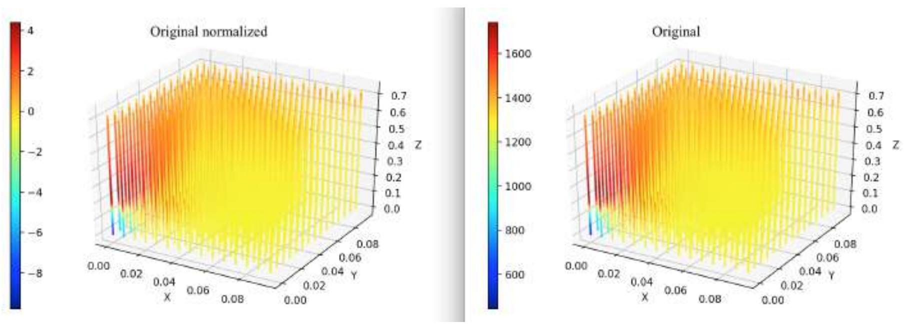


Figure 3.22: Original image

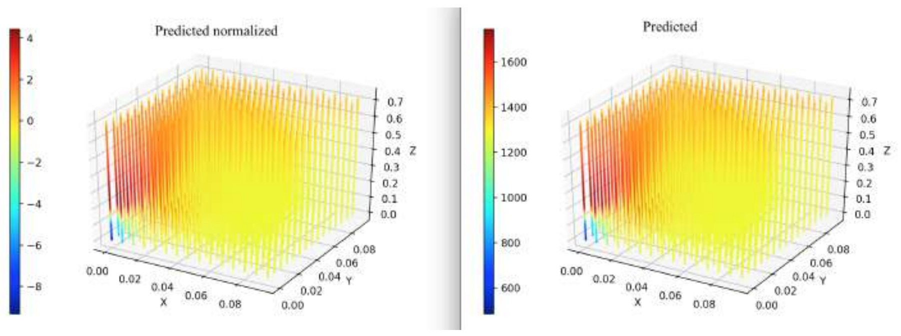


Figure 3.23: Reconstructed image

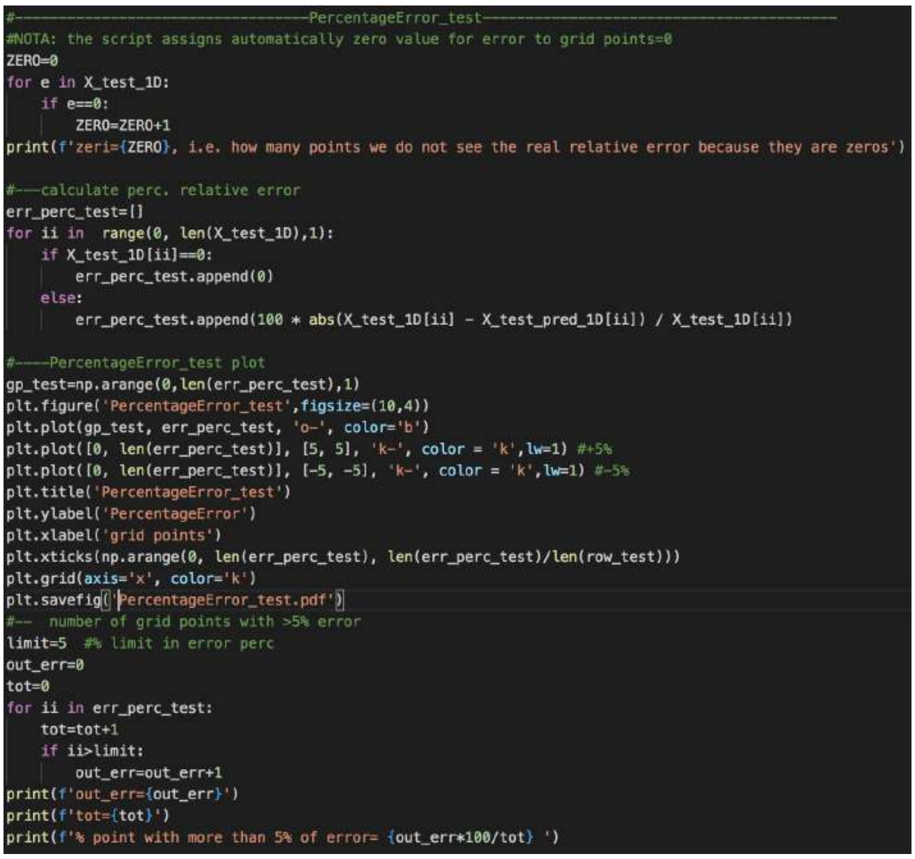
```
ORIGINAL ARRAY (reduced mesh version)= 24 300 elements
CODE= 72 elements array
[0.29625472 0.37266487 0.28748617 0.14876254 0.7973578  0.5016492
0.14867617 0.29581308 0.19254714 0.16508481 0.33441973 0.3622058
0.24576004 0.5525425  0.45719168 0.47793767 0.21135975 0.4223018
0.39432946 0.73621565 0.7937947  0.85891867 0.29021844 0.3456081
0.46777934 0.6791807  0.85057384 0.98540723 0.22922099 0.2476119
0.51817256 0.6312096  0.7843675  0.97529024 0.13252103 0.18274035
0.5625907  0.57876575 0.693001   0.91976064 0.07688414 0.12832482
0.5860489  0.52292705 0.595112   0.8422132  0.04167599 0.09317081
0.5904275  0.47269222 0.512344   0.7587106  0.02462439 0.06994813
0.6064538  0.44367298 0.43814728 0.6893065  0.01488622 0.04969355
0.70766205 0.40865904 0.33373794 0.61953336 0.00908016 0.03039117
1.131802   0.09281466 0.03220464 0.30793083 0.06209738 0.02896498]
```

A second script was also written for the post process, which you can find in the appendix B.3.1. In this second script we will see how to build the parity plot, calculate the percentage error, NRMSE RMSE and R^2. Only parts of the code relating to test simulations will be reported.

Percentage error and parity plot

The relative percentage error is calculated in the following formula and the code used to calculate it is shown in fig 3.24.

$$\text{error} = 100 * abs(X_{original} - X_{predicted})/X_{original} \tag{3.1}$$



```
#----------------------------------PercentageError_test------------------------------------------
#NOTA: the script assigns automatically zero value for error to grid points=0
ZERO=0
for e in X_test_1D:
    if e==0:
        ZERO=ZERO+1
print(f'zeri={ZERO}, i.e. how many points we do not see the real relative error because they are zeros')

#---calculate perc. relative error
err_perc_test=[]
for ii in  range(0, len(X_test_1D),1):
    if X_test_1D[ii]==0:
        err_perc_test.append(0)
    else:
        err_perc_test.append(100 * abs(X_test_1D[ii] - X_test_pred_1D[ii]) / X_test_1D[ii])

#----PercentageError_test plot
gp_test=np.arange(0,len(err_perc_test),1)
plt.figure('PercentageError_test',figsize=(10,4))
plt.plot(gp_test, err_perc_test, 'o-', color='b')
plt.plot([0, len(err_perc_test)], [5, 5], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, len(err_perc_test)], [-5, -5], 'k-', color = 'k',lw=1) #-5%
plt.title('PercentageError_test')
plt.ylabel('PercentageError')
plt.xlabel('grid points')
plt.xticks(np.arange(0, len(err_perc_test), len(err_perc_test)/len(row_test)))
plt.grid(axis='x', color='k')
plt.savefig('PercentageError_test.pdf')
#--  number of grid points with >5% error
limit=5  #% limit in error perc
out_err=0
tot=0
for ii in err_perc_test:
    tot=tot+1
    if ii>limit:
        out_err=out_err+1
print(f'out_err={out_err}')
print(f'tot={tot}')
print(f'% point with more than 5% of error= {out_err*100/tot} ')
```

Figure 3.24: Fourth script (b)

While, a parity plot is a scatterplot that compares experimental data against tabulated data. Each point has coordinates (x, y), where x is the tabulated value, and y is the corresponding experimental value (code in fig 3.26). A line of the equation y = x is added as a reference. When an experimental value equals a tabulated value, the point will lie on the line. The parity plot obtained for all the values of the 4 test simulations are in fig 3.25 . Note that in addition to the *exact prediction* line, lines corresponding to a percentage error of + 5% and -5% have also been added so as to get an idea of the goodness of CNN. It is found that only 0.24% of the data are out of the error range [-5%, + 5%].

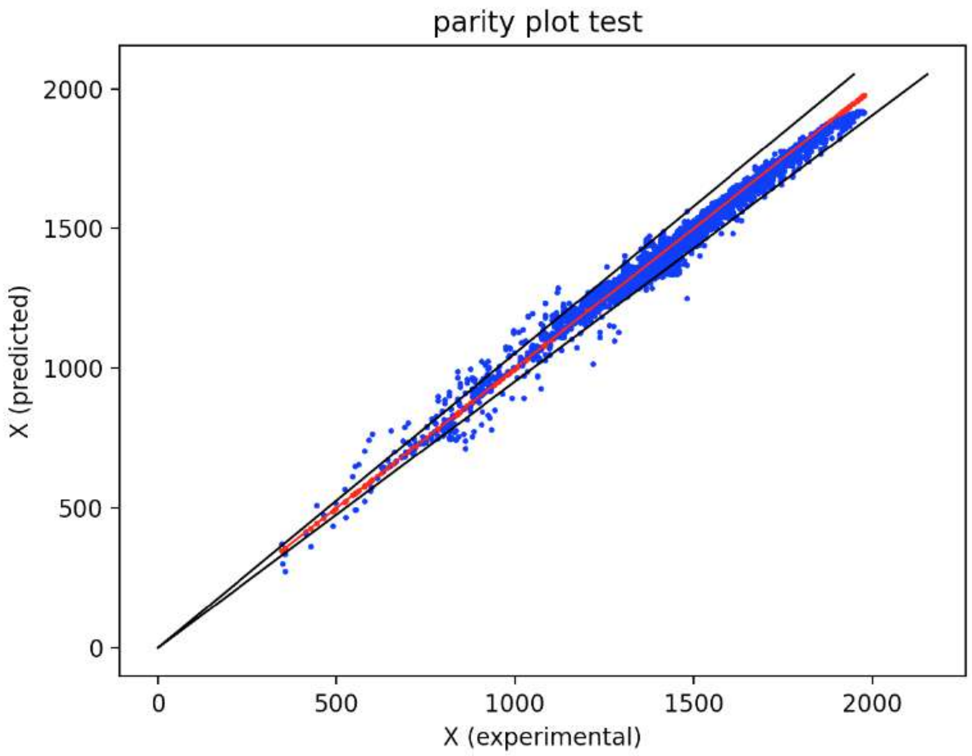
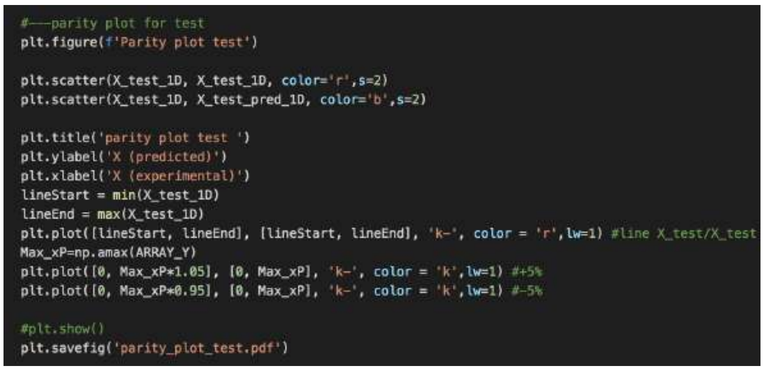


Figure 3.25: Parity plot for test simulations



```
#---parity plot for test
plt.figure(f'Parity plot test')

plt.scatter(X_test_1D, X_test_1D, color='r',s=2)
plt.scatter(X_test_1D, X_test_pred_1D, color='b',s=2)

plt.title('parity plot test ')
plt.ylabel('X (predicted)')
plt.xlabel('X (experimental)')
lineStart = min(X_test_1D)
lineEnd = max(X_test_1D)
plt.plot([lineStart, lineEnd], [lineStart, lineEnd], 'k-', color = 'r',lw=1) #line X_test/X_test
Max_xP=np.amax(ARRAY_Y)
plt.plot([0, Max_xP*1.05], [0, Max_xP], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, Max_xP*0.95], [0, Max_xP], 'k-', color = 'k',lw=1) #-5%

#plt.show()
plt.savefig('parity_plot_test.pdf')
```

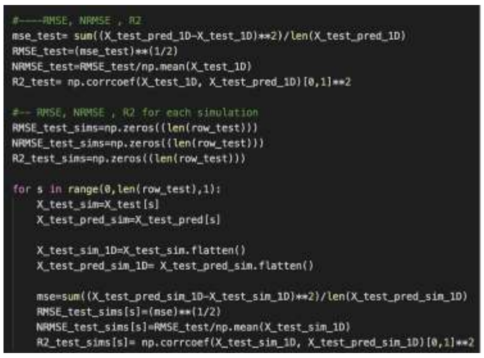
Figure 3.26: Fourth script (c)

RMSE, NRMSE, R^2

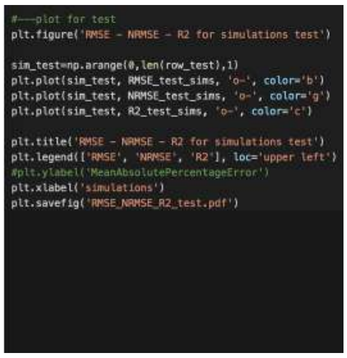
In order to have a single value that can be compared to evaluate the performance of the neural network it was decided to calculate RMSE (ie Root Mean Square Error), NRMSE (Normalized Root Mean Square Error) and R^2 (ie coefficient of determination). Following, equation for NRMSE and RMSE. The code is in fig 3.27.

$$\text{RMSE} = \sqrt{\sum((X_{pred} - X_{orig}) * *2)/N_{all-grid-points}} \tag{3.2}$$

$$\text{NRMSE} = NRMSE/mean \tag{3.3}$$



```
#----RMSE, NRMSE , R2
mse_test= sum((X_test_pred_1D-X_test_1D)**2)/len(X_test_pred_1D)
RMSE_test=(mse_test)**(1/2)
NRMSE_test=RMSE_test/np.mean(X_test_1D)
R2_test= np.corrcoef(X_test_1D, X_test_pred_1D)[0,1]**2

#-- RMSE, NRMSE , R2 for each simulation
RMSE_test_sims=np.zeros((len(row_test)))
NRMSE_test_sims=np.zeros((len(row_test)))
R2_test_sims=np.zeros((len(row_test)))

for s in range(0,len(row_test),1):
    X_test_sim=X_test[s]
    X_test_pred_sim=X_test_pred[s]

    X_test_sim_1D=X_test_sim.flatten()
    X_test_pred_sim_1D= X_test_pred_sim.flatten()

    mse=sum((X_test_pred_sim_1D-X_test_sim_1D)**2)/len(X_test_pred_sim_1D)
    RMSE_test_sims[s]=(mse)**(1/2)
    NRMSE_test_sims[s]=RMSE_test/np.mean(X_test_sim_1D)
    R2_test_sims[s]= np.corrcoef(X_test_sim_1D, X_test_pred_sim_1D)[0,1]**2
```



```
#---plot for test
plt.figure('RMSE - NRMSE - R2 for simulations test')

sim_test=np.arange(0,len(row_test),1)
plt.plot(sim_test, RMSE_test_sims, 'o-', color='b')
plt.plot(sim_test, NRMSE_test_sims, 'o-', color='g')
plt.plot(sim_test, R2_test_sims, 'o-', color='c')

plt.title('RMSE - NRMSE - R2 for simulations test')
plt.legend(['RMSE', 'NRMSE', 'R2'], loc='upper left')
#plt.ylabel('MeanAbsolutePercentageError')
plt.xlabel('simulations')
plt.savefig('RMSE_NRMSE_R2_test.pdf')
```

Figure 3.27: Fourth script (d)

Chapter 4

Results

As already mentioned in the previous chapters, there are some parameters of the neural network (ie the hyperparameters) whose value affects the performance of the autoencoder. In this chapter we will see a sensibility analysis that will try to highlight the influence of hyperparameters on the network (if any), all with the aim of creating a high-performance optimized CNN that can be a valid opponent to the pca (proposed by BURN group). For the sensibility analysis it was decided to work with temperature filed and a dense but reduced mesh of the furnace. This has 24 300 grid points and a dimension of 0.1 * 0.1 * 0.72m (furnace height) as shown in Fig. 4.1. The author is aware of the fact that changing the initial dataset (i.e. considering the entire quarter of the furnace) will also change the neural network that is considered optimized, thus having to repeat a sensibility procedure, but the choice was made by looking for a compromise between the computational cost and the reasonableness / realism of the simulation considered. A few more words about the last concept: mild combustion is characterized by an almost uniform temperature range in the bulk of the furnace. It was therefore decided to reduce the mesh to the only area where there are significant changes in temperature. It will not be difficult to accept that the performance of the network considering the full quarter furnace will not be overly different or in need of major adjustments.

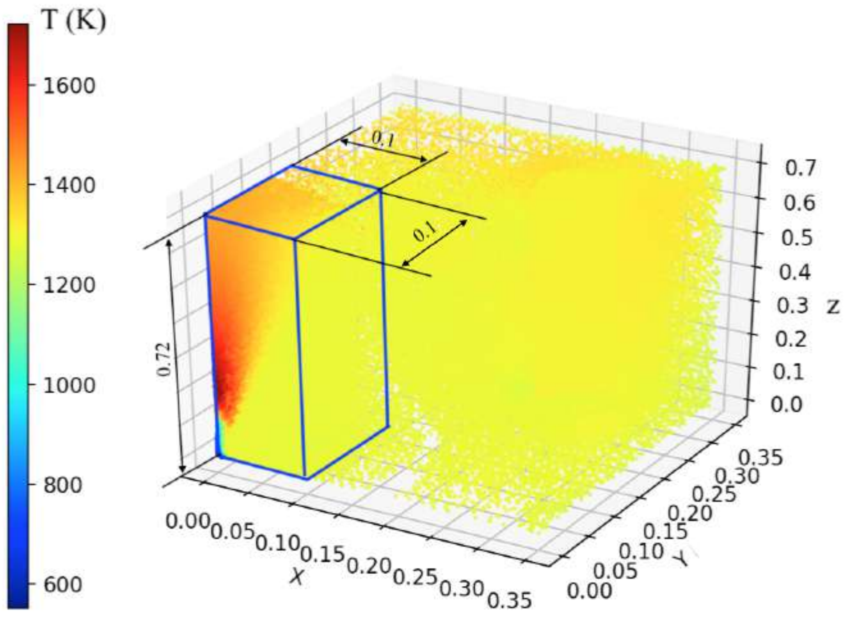


Figure 4.1: Reduced mesh (quotes in meters)

4.1 Sensibility analysis

Now let's get to the heart of the sensibility analysis, each study will be presented indicating the values used for the parameters that will not vary and for what varies, as well as the results in terms of RMSE, NRMSE, R^2. The latter are presented in graphic form, also including the threshold (red line) corresponding to the error obtained with pca approach. It is emphasized that at this stage a comparison between them is important and not so much the evaluation of how optimal they are. Some analysis were performed maintaining a common architecture (in terms of number of layers, filters etc..), shown below. If something will change, it will be the author's responsibility to indicate it to you.

```
#--ARCHITECTURE
# Determine sample shape
input_img = (X_train_N.shape[1], X_train_N.shape[2],X_train_N.shape[3], X_train_N.shape[4])
encoder = Sequential(name='ENCODER')
encoder.add(Conv3D(20, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(MaxPooling3D(pool_size=(3,3,3)))
encoder.add(Conv3D(10, kernel_size=(3,3,3), activation='relu', padding='same'))
encoder.add(MaxPooling3D(pool_size=(5,5,3)))
print('encoder part architecture')
encoder.summary()
#---
autoencoder=Sequential(name='AUTOENCODER')
autoencoder.add(encoder)
autoencoder.add(UpSampling3D(size=(5, 5, 3)))
autoencoder.add(Conv3DTranspose(10, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(UpSampling3D(size=(3, 3, 3)))
autoencoder.add(Conv3DTranspose(20, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(1, kernel_size=(3, 3, 3), activation='sigmoid', padding='same'))
print('complete model architecture')
autoencoder.summary()
```

4.1.1 Normalisation and activation function of final layer

The matrix of the values of the output variable (in our case the temperature) needs normalization before, being sent as input to the neural network. 5 different types of normalization have been identified which we will indicate them with letters.

Normalisation A

$$X_N = \left(\frac{X - mean}{std} - min\right)/max \tag{4.1}$$

Normalisation B

$$X_N = \frac{X}{mean} \tag{4.2}$$

Normalisation C

$$X_N = \frac{X - mean}{std} \tag{4.3}$$

Normalisation D

$$X_N = \frac{X - mean}{std} - min \tag{4.4}$$

Normalisation E

$$X_N = \frac{X - min}{max - min} \tag{4.5}$$

The choice of normalization also determines the range in which the input values are found. In other words:

- Norm B: it redistributes in order to have unitary mean.
- Norm C: also called data standardization, is a process of scaling so that they have a mean value of 0 and a standard deviation of 1.
- Norm D: it maintains the distribution of the normalization C but is translated to positive values only.
- Norm A: it maintains the distribution of the normalization D but it redistributes values in a range between zero and one.

- Norm E: it redistributes values in a range between zero and one.

Consequently, it is necessary to carry out an analysis also by varying the activation function of the last convolutional layer because this determines the range of reconstructed image values, which will be compared with the original ones. Below tables for fixed values and the results in tabular and graphical form. To better see the difference, a graphic version focused on the best results has also been reported.

Fixed parameters	
Batch size	3
Numpy seed	1
Learning rate	0.001
Patience	10
Pooling	max
Cost function	MSE
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.1: Fixed parameters for normalisation and final activation function

	RMSE	**NRMSE**	**R^2**
Norm A			
Sigmoid	30.255	0.022	0.913
Linear	28.288	0.021	0.924
Relu	32.176	0.024	0.905
Norm B			
Sigmoid	102.593	0.075	0.000
Linear	36.239	0.027	0.893
Relu	36.866	0.027	0.874
Norm C			
Sigmoid	71.866	0.053	0.689
Linear	22.168	0.016	0.951
Relu	49.080	0.036	0.849
Norm D			
Sigmoid	928.232	0.682	0.000
Linear	39.306	0.029	0.910
Relu	47.231	0.035	0.906
Norm E			

Sigmoid	35.085	0.026	0.904
Linear	32.140	0.024	0.920
Relu	30.574	0.022	0.912

Table 4.2: Results for normalisation sensibility analysis

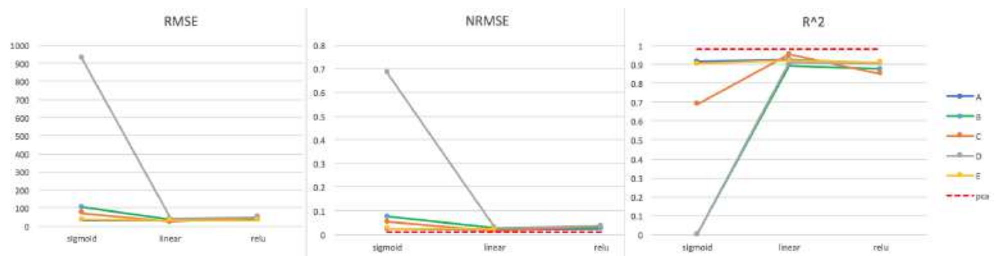


Figure 4.2: Graphs of results for normalisation (a)

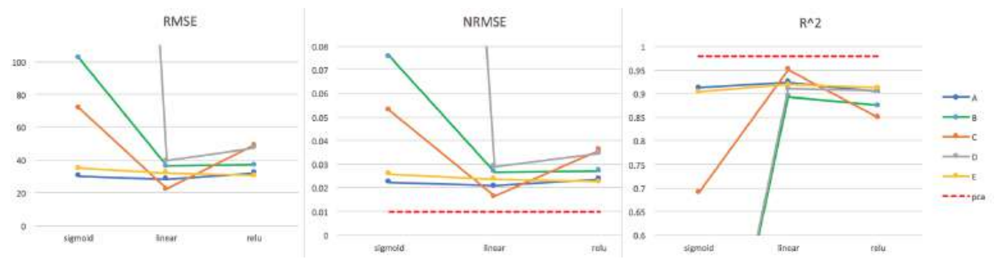


Figure 4.3: Graphs of results for normalisation (b)

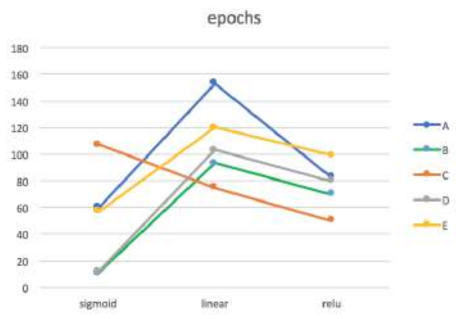


Figure 4.4: Graphs of results (epochs)

It is clear that the best activation function for the last layer is linear in all 5 cases. it is also true that it needs more iterations (epochs) before reaching convergence (except in case C). It is noted that normalization A and E perform very well regardless of the activation function but normalization C with AF linear has the best performance.

4.1.2 Learning rate

The learning rate indicates the step size of change of weights of a deep neural network at each iteration while moving toward a minimum of a loss function and is one of hyperparameters most delicate and important to adjust (tune) to achieve excellent performance on our problem. As said before: the smaller learning rate is and more accurate we are. At the same time, we don't want η to be too small, since that will make the changes tiny, and thus the gradient descent algorithm will work very slowly.

The results obtained confirm what we thought, there is an optimal value for the learning rate, it is not suitable neither to choose one too large nor one too small. Moreover, for values greater than 0.01, convergence was not achieved. The learning rate that we consider best for this case is 0.0005, however probably all values under 0.005 would be good.

In the case of the study of the optimal value for learning rate, a trend in the epochs necessary for convergence is also noted, therefore a significant graph is also shown for these. The higher the learning rate, the less time it takes to reach the end of the process.

Fixed parameters	
Batch size	3
Normalisation	A
Final AF	sigmoid
Numpy seed	1
Patience.	10
Pooling	max

Cost function	MSE
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.3: Fixed parameters for learning rate

LR	RMSE	NRMSE	R^2
0.00001	43.556	0.032	0.814
0.0001	35.017	0.026	0.888
0.0005	32.865	0.024	0.909
0.001	37.727	0.028	0.905
0.005	41.858	0.031	0.850
0.01	102.329	0.075	0.031

Table 4.4: Results for learning rate sensibility analysis

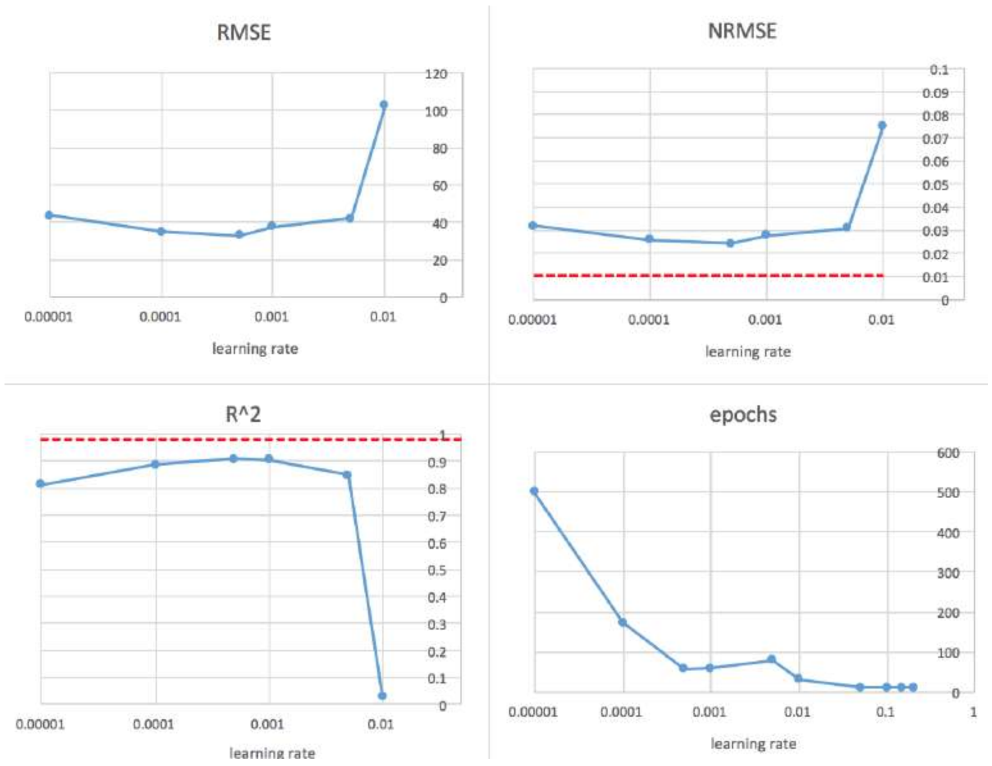


Figure 4.5: Graphs of results for learning rate

4.1.3 Batch size

If the size of the sample or complexity of the network are too much and the epoch cannot be run all together. Epoch is split into batches, and the epoch is run in two or more iterations. Thus, the batch size defines the number of samples that will be propagated through the network each iteration.

Advantages of using a batch size less than number of all samples are:

- It requires less memory. Since you train the network using fewer samples, the overall training procedure requires less memory. That's especially important if you are not able to fit the whole dataset in your machine's memory.
- Typically networks train faster with mini-batches. That's because we update the weights after each propagation.

The disadvantage: the smaller the batch the less accurate the estimate of the gradient will be.

In the results there is not an excessive difference in accuracy by changing the size of the batch, however it is considered as an optimal value 3.

Fixed parameters	
Normalisation	A
Final AF	sigmoid
Numpy seed	1
Learning rate	0.001
Patience.	10
Pooling	max
Cost function	MSE
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.5: Fixed parameters for learning rate

Batch size	**RMSE**	**NRMSE**	**R^2**
2	27.135	0.020	0.931
3	25.587	0.019	0.940
5	28.019	0.021	0.928
10	40.011	0.029	0.871
15	37.589	0.028	0.900
20	46.217	0.034	0.879
25	43.095	0.032	0.857
30	30.488	0.022	0.909
35	32.796	0.024	0.906
40	79.989	0.059	0.488

Table 4.6: Results for batch size sensibility analysis

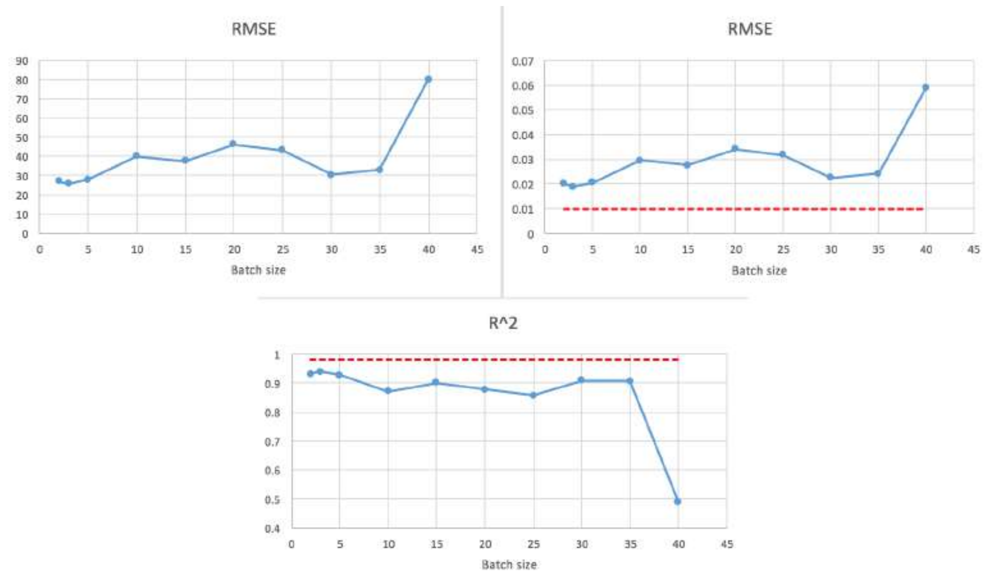


Figure 4.6: Graphs of results for batch size

4.1.4 Patience

Patience is the number of epochs with no improvement after which training will be stopped. We therefore expect that the longer we wait before settling for the result, the better the accuracy will be. In fact, the curve of both errors and R2 seems to reach an asymptote. It is therefore stated that it is necessary to have a high enough number to ensure the optimum. We choose 15.

Fixed parameters	
Batch size	3
Normalisation	A
Final AF	sigmoid
Numpy seed	1
Learning rate	0.001
Pooling	max
Cost function	MSE
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.7: Fixed parameters for learning rate

Patience	**RMSE**	**NRMSE**	**R^2**
1	81.144	0.060	0.518
2	76.873	0.057	0.514
3	44.890	0.033	0.821
4	44.616	0.033	0.868
5	36.608	0.027	0.901
6	38.961	0.029	0.877
7	35.880	0.026	0.914
8	25.443	0.019	0.938
9	43.982	0.032	0.899
10	37.964	0.028	0.915
11	30.907	0.023	0.926
12	35.029	0.026	0.910
13	25.521	0.019	0.941
14	27.726	0.020	0.939
15	23.806	0.018	0.944
16	25.485	0.019	0.939
17	22.547	0.017	0.950
18	25.975	0.019	0.941
19	23.361	0.017	0.949
20	24.660	0.018	0.947
21	25.055	0.018	0.943
22	23.812	0.018	0.945
23	23.464	0.017	0.946
24	22.718	0.017	0.950
25	22.274	0.016	0.951
26	23.617	0.017	0.946
27	23.691	0.017	0.951

28	24.800	0.018	0.947
29	21.787	0.016	0.953
30	28.538	0.021	0.950

Table 4.8: Results for patience sensibility analysis

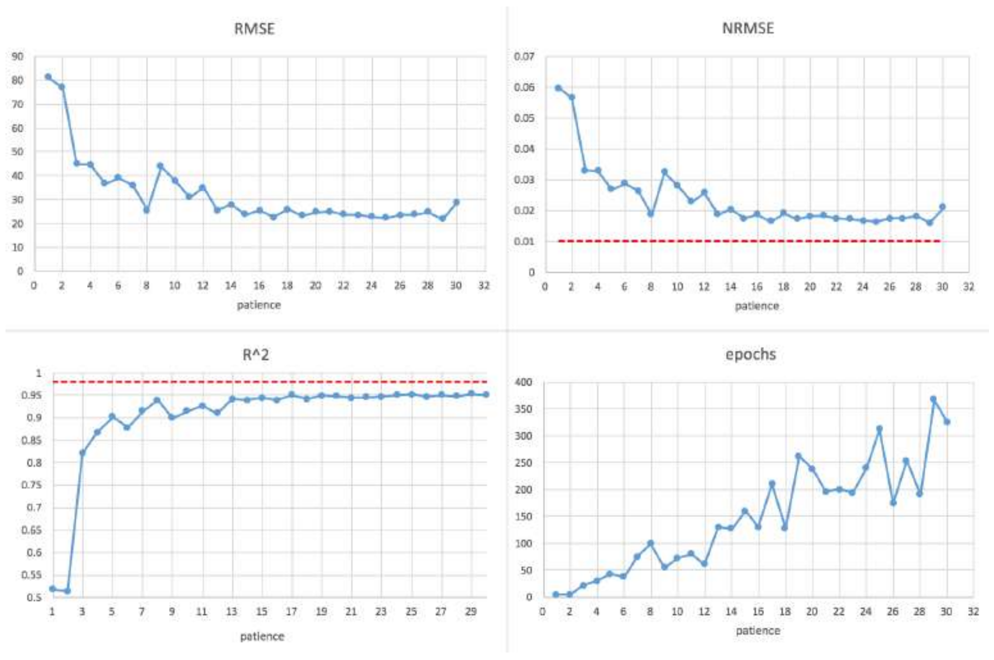


Figure 4.7: Graphs of results for patience

4.1.5 Cost function

The implemented cost functions are: Mean Squared Error, Mean Absolute Error, Mean Absolute Percentage Error and Mean Squared Logarithmic Error. they will not be further explained here, please refer to the section 2.4.1. The MSE cost function is better in performance, MAE and MSLE are not too far apart.

Fixed parameters	
Batch size	3
Normalisation	A
Final AF	sigmoid
Numpy seed	1
Learning rate	0.001

Patience.	10
Pooling	max
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.9: Fixed parameters for learning rate

Cost function	**RMSE**	**NRMSE**	**R^2**
MSE	27.424 0.020	0.933	
MAE	36.592	0.027	0.888
MA%E	49.981	0.037	0.806
MSLE	32.099	0.024	0.902

Table 4.10: Results for cost function sensibility analysis

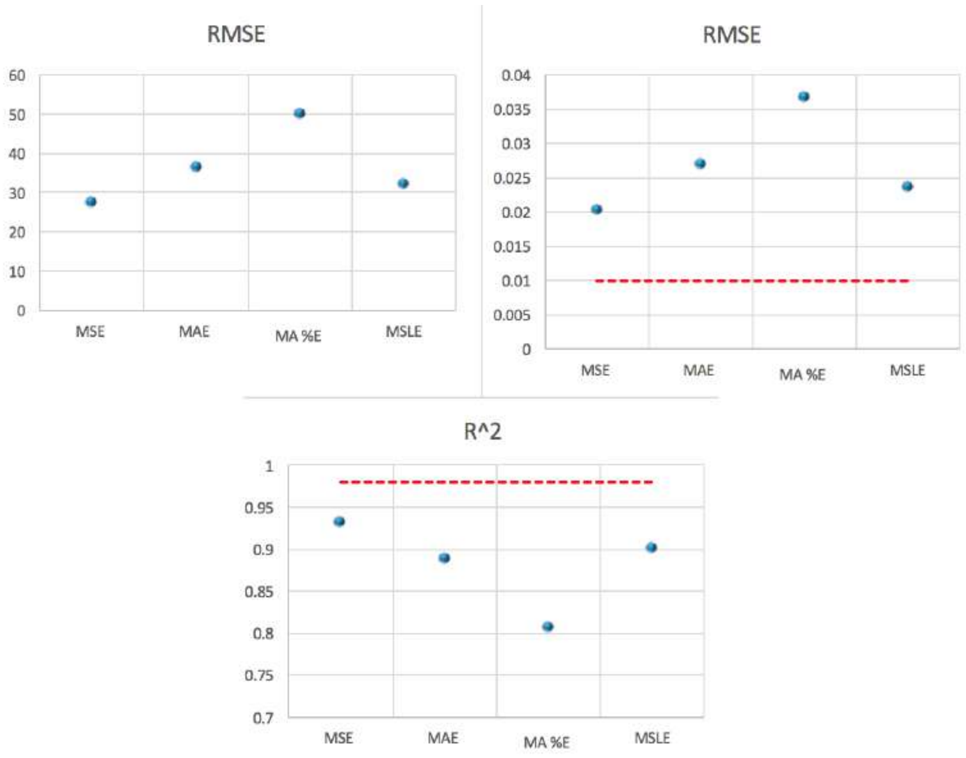


Figure 4.8: Graphs of results for cost function

4.1.6 Pooling

As previously explained, max pooling holds the maximum between the cells belonging to the kernel. While average pooling averages these. Average pooling is chosen as the best of the two.

Fixed parameters	
Batch size	3
Normalisation	A
Final AF	sigmoid
Numpy seed	1
Learning rate	0.001
Patience.	10
Cost function	MSE
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.11: Fixed parameters for learning rate

Filters distribution	**RMSE**	**NRMSE**	**R^2**
20/10-10/20			
Max pooling	32.435	0.024	0.905
Average pooling	26.520	0.019	0.932
30/20-20/30			
Max pooling	32.549	0.024	0.913
Average pooling	25.974	0.019	0.934

Table 4.12: Results for pooling sensibility analysis

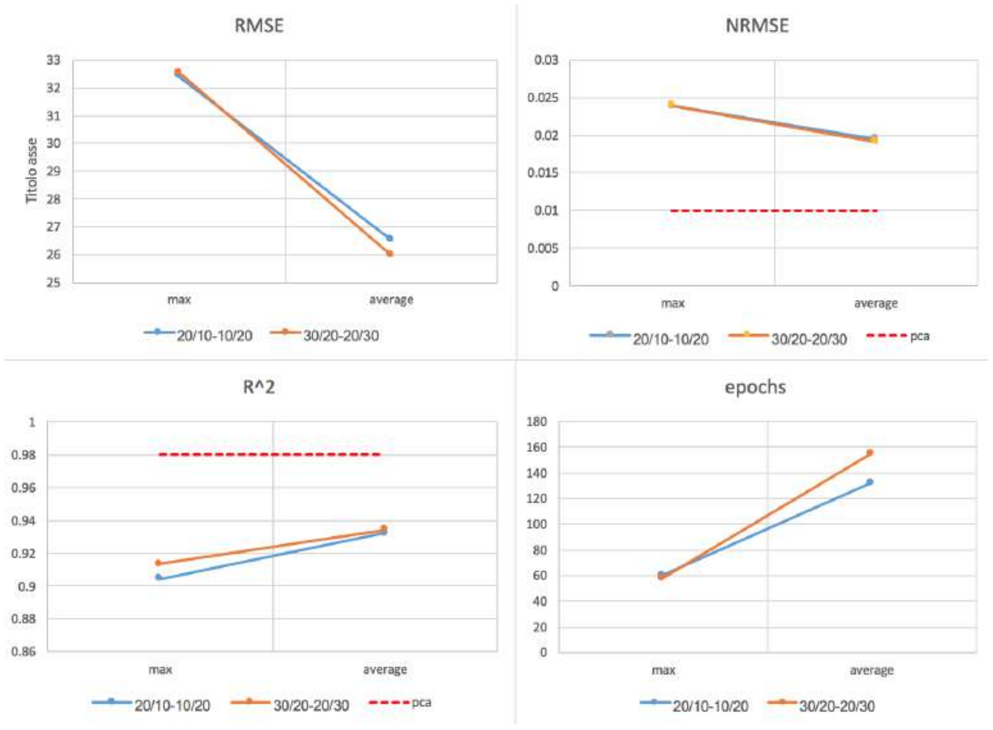


Figure 4.9: Graphs of results for pooling

4.1.7 Optimiser

Among the optimization strategies explained in section 2.4.2, Stochastic gradient descendent, RMSprop and Adam are studied.

RMSprop and Adam give significantly better results than SGD. We choose Adam.

Fixed parameters	
Batch size	3
Normalisation	A
Final AF	sigmoid
Numpy seed	1
Learning rate	0.001
Patience.	10
Pooling	max
Cost function	MSE

Activation	Relu
Code dimension	72

Table 4.13: Fixed parameters for learning rate

Optimiser	**RMSE**	**NRMSE**	**R^2**
SGD	102.657	0.075	0.177
RMS prop	12.674	0.009	0.985
Adam	9.599	0.007	0.991

Table 4.14: Results for optimiser sensibility analysis

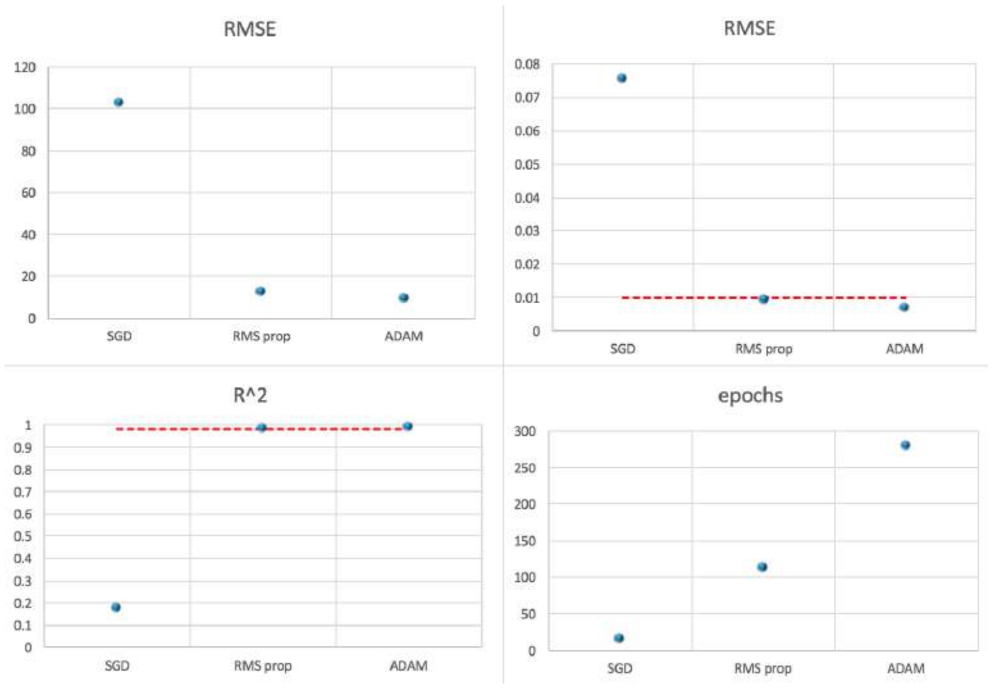


Figure 4.10: Graphs of results for optimiser

4.1.8 Numpy seed

Within the optimization procedure there are parameters that require initialization, but wanting to avoid randomness, the numpy seed is imposed at a fixed value. It is intuitive that depending on the value assigned to it the result will be different, a study is performed to observe how the network responds by varying the seed..

From the results it is not possible to determine a characteristic trend, it is therefore believed that each time the CNN optimized for the other parameters must be subjected to a study of behaviour towards the seed.

Fixed parameters	
Batch size	3
Normalisation	A
Final AF	sigmoid
Learning rate	0.001
Patience.	10
Pooling	max
Cost function	MSE
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.15: Fixed parameters for learning rate

Numpy seed	**RMSE**	**NRMSE**	**R^2**
1	40.073	0.029	0.903
2	31.901	0.023	0.906
3	28.794	0.021	0.922
4	38.153	0.028	0.912
5	26.686	0.020	0.935
6	39.476	0.029	0.884
7	28.765	0.021	0.922
8	39.055	0.029	0.876
9	29.469	0.022	0.924
10	31.364	0.023	0.921

Table 4.16: Results for numpy seed sensibility analysis

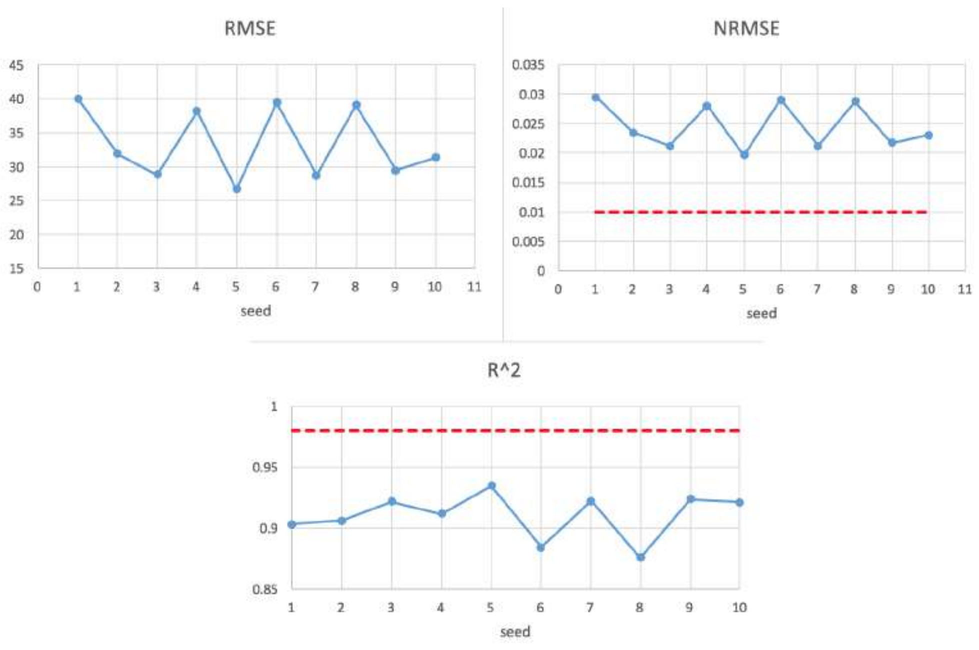


Figure 4.11: Graphs of results for numpy seed

4.1.9 Activation function

The activation function can impact the network's ability to converge and learn for different ranges of input values, and also its training speed. Since the activation function is directly related to the distribution of the input values to the neuron, the behaviour of CNN has been studied both for a normalization of type A (because we have always used this in the previous tests) and C (because it is the best normalization).

With normalisation A

Fixed parameters	
Batch size	3
Normalisation	A
Final AF	sigmoid
Learning rate	0.001

Patience.	10
Pooling	max
Cost function	MSE
Optimizer	Adam
Code dimension	72

Table 4.17: Fixed parameters for learning rate

Activation function	**RMSE**	**NRMSE**	**R^2**
Relu	25.991	0.019	0.942
Linear	32.570	0.024	0.896
Tanh	42.100	0.031	0.856
Sigmoid	94.935	0.070	0.447
None	30.669	0.023	0.917
Leaky relu	32.907	0.024	0.898

Table 4.18: Results for activation function sensibility analysis (norm A)

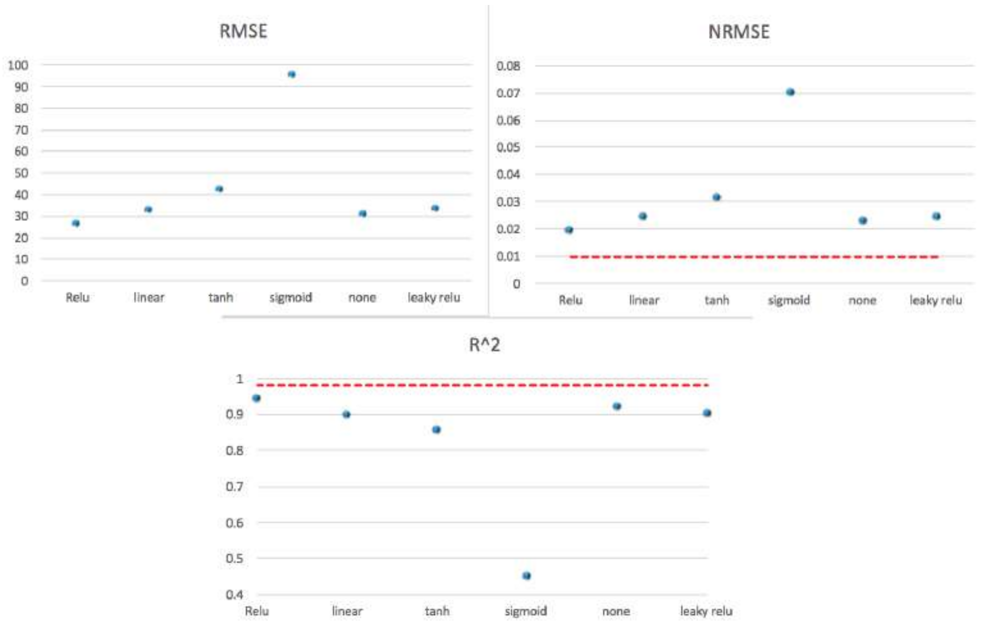


Figure 4.12: Graphs of results for AF norm A

In the case of normalization A and sigmoid activation function of the last layer, the only one that gives performances that differ greatly from the others is the sigmoid AF.

With normalisation C and linear final layer

Fixed parameters	
Batch size	3
Normalisation	C
Final AF	linear
Learning rate	0.001
Patience.	10
Pooling	max
Cost function	MSE
Optimizer	Adam
Code dimension	72

Table 4.19: Fixed parameters for learning rate

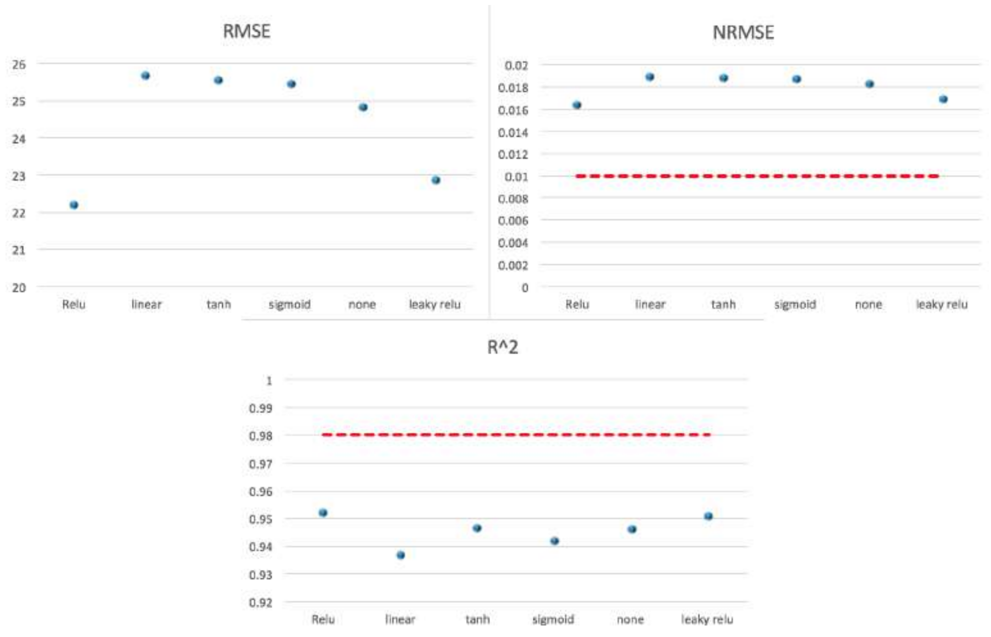


Figure 4.13: Graphs of results for AF norm C

Activation function	**RMSE**	**NRMSE**	**R^2**
Relu	22.176	0.016	0.952
Linear	25.656	0.019	0.936
Tanh	25.517	0.019	0.946
Sigmoid	25.426	0.019	0.942
None	24.810	0.018	0.946
Leaky relu	22.849	0.017	0.951

Table 4.20: Results for activation function sensibility analysis (norm C)

In the case of C normalization and linear activation function of the last layer, there are no marked differences, it is advisable to scan which one to use in each case. We choose relu.

∼∼∼∼∼∼∼∼∼∼∼∼∼∼∼∼∼∼∼∼∼∼

After carrying out these analysis that share the same architecture, we question this too. The studies presented to you are based on questions: Does CNN geometry affect CNN performance? is it possible to identify a trend? The following table, in which the parameters set are indicated, will be considered valid for the subsequent section.

Fixed parameters	
Batch size	3
Normalisation	A
Final AF	sigmoid
Numpy seed	1
Learning rate	0.001
Patience.	10
Pooling	max
Cost function	MSE
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.21: Fixed parameters for studies of CNN architecture

4.1.10 Number of filters

The number of filters indicates how many features a convolutional layer looks for. Each filter corresponds to an output feature map so the more filters a conv layer has, the larger the output size will be. The innermost filters look for features that are less and less intuitive for the human eye. These tests follow architecture in section 4.1, the only thing that is changed is the first value of the convolutional layers or the number of filters of that layer. So for example if the total number in the table is 30 it means that we have conv (10 filters) + max pooling + conv (5 filters) + max pooling and mirror.

Number of filters	**RMSE**	**NRMSE**	**R^2**
6	64.133	0.047	0.596
12	41.118	0.030	0.854
18	47.659	0.035	0.799
24	31.538	0.023	0.902
30	33.910	0.025	0.903
60	31.137	0.023	0.916
72	27.571	0.020	0.932
90	36.612	0.027	0.908
120	28.205	0.021	0.924
180	30.854	0.023	0.925
210	29.341	0.022	0.927
240	32.733	0.024	0.903
270	27.668	0.020	0.927
300	29.076	0.021	0.921
330	36.276	0.027	0.939

Table 4.22: Results for number of filters sensibility analysis

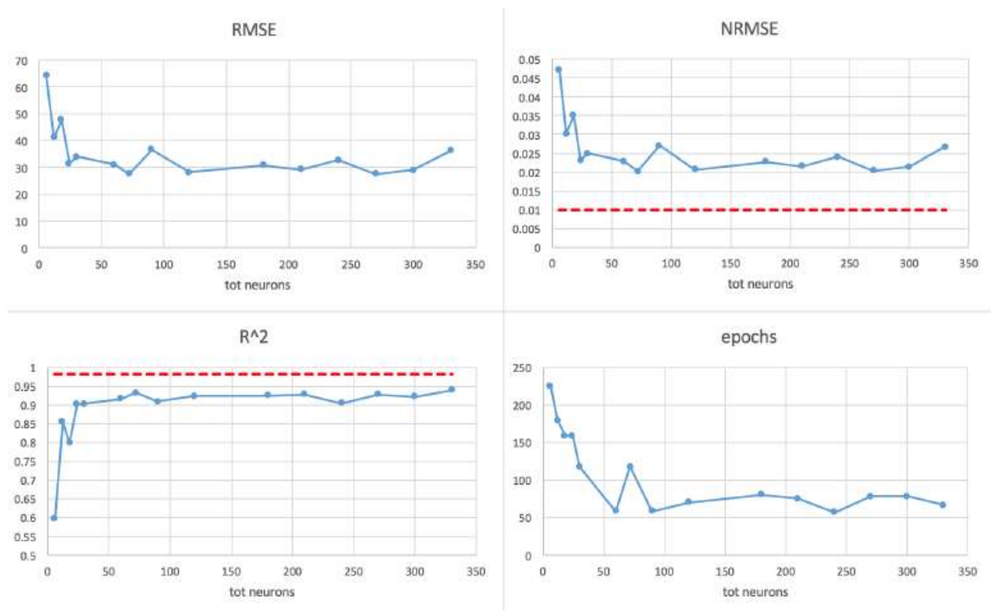


Figure 4.14: Graphs of results for number of filters

In the results it is possible to recognize an increasing trend in accuracy with the increase in the number of filters, but a maximum is reached, which is maintained once the threshold (in our case) of about 60 filters is exceeded. Therefore, ensure that we have an architecture with a sufficient number of filters will be our future responsibility.

4.1.11 Number of convolutional layers

This section is dedicated to understanding whether (with the same number of total filters in the network) it is better to distribute the filters on more or less convolutional layers. There are no particular trends, it is therefore believed that tests have to be made to choose the best configuration for each specific case.

Number of conv layers	**RMSE**	**NRMSE**	**R^2**
6	30.799	0.023	0.912

5	81.607	0.060	0.476
4	36.573	0.027	0.896
3	22.910	0.017	0.952
2	22.993	0.017	0.968
1	26.342	0.019	0.940

Table 4.23: Results for number of conv layers sensibility analysis

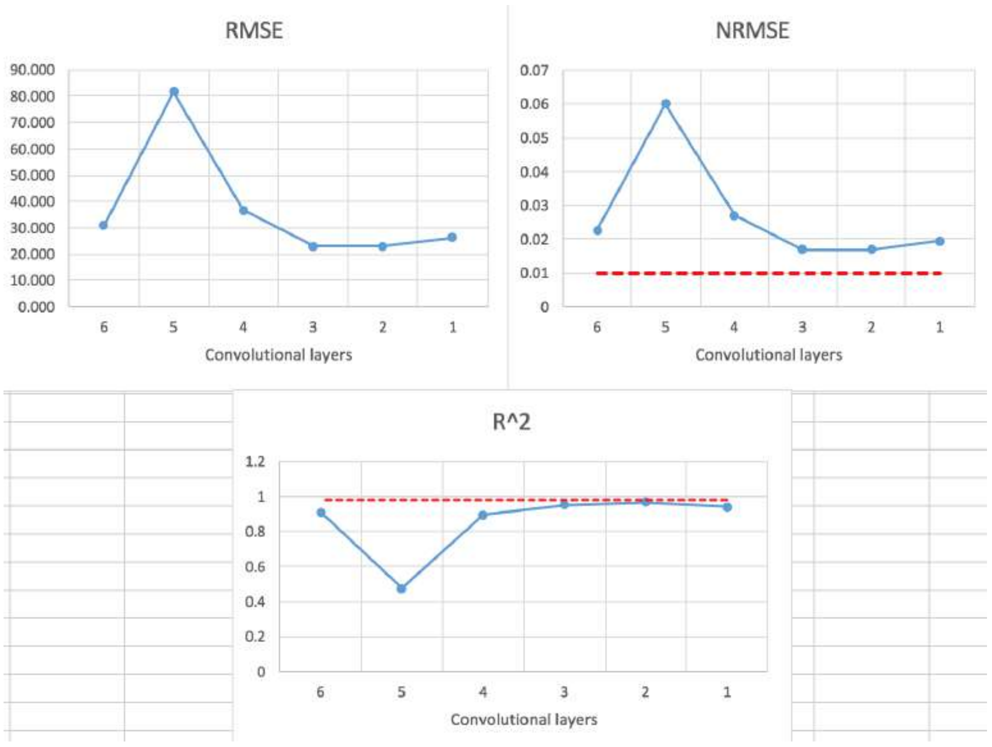


Figure 4.15: Graphs of results for number of conv layers

4.1.12 Symmetry

This section is dedicated to evaluating the influence of the symmetry of the neural network with respect to the code on its performance. In other words, the perfectly symmetrical autoencoder can be seen as a butterfly (centred in code) and whose wings represent the succession of layers with more and more filters moving away from the Code. At the extreme point, we find the butterfly with only one wing (whether it is the coding or decoding one) which represents a scale in the number of filters on one side (the wing) while a homogeneity on the other.

Initially, remembering that the autoencoder is a reduction in size with consequent reconstruction, it was thought that the perfectly symmetrical structure would be the best but the results show that this is not the case, moreover a particular trend is not recognizable so also in this case it is good carry out a targeted analysis for your case study.

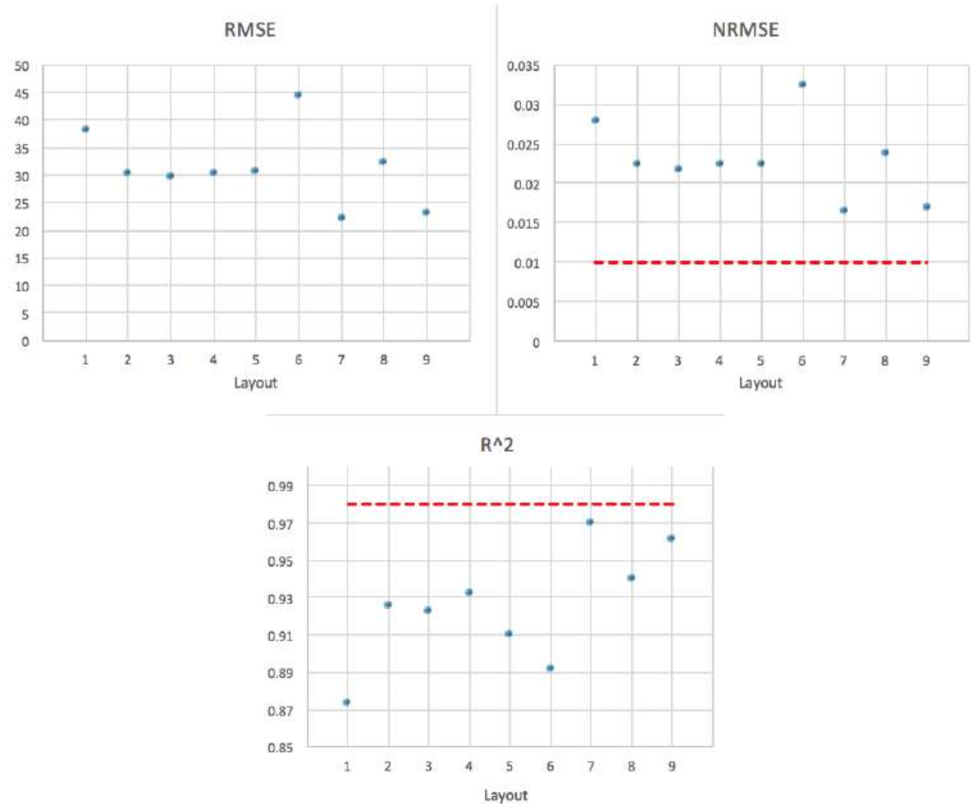


Figure 4.16: Graphs of results for symmetry

	Layout	RMSE	NRMSE	R^2

1↦	98765-55555	38.031	0.028	0.873
2↦	98765-55556	30.402	0.022	0.926
3↦	98765-55567	29.573	0.022	0.922
4↦	98765-55678	30.330	0.022	0.932
5↦	98765-56789	30.509	0.022	0.910
6↦	87655-56789	44.242	0.033	0.892
7↦	76555-56790	22.204	0.016	0.971
8↦	65555-56791	32.210	0.024	0.940
9↦	55555-56792	23.072	0.017	0.961

Table 4.24: Results for symmetry sensibility analysis

4.1.13 Layout

The next three subsections are devoted to the study of the arrangement of the number of filters in a wing of a mirrored architecture.

- Part A studies the importance of symmetry within a wing, using the first layer of max pooling as the center and varying the number of filters in the conv layer preceding and in the one following the pooling.

Parts B and C share a mirror architecture with 5 convolutional layers for each wing. They differ because

- Part B starts with a uniform distribution of the number of filters then, keeping the central layer of the wing fixed, creates an increasingly marked discontinuity in the number of filters between the central and the two pairs of layers that flank it.

- Part C instead has a smoother variation in the number of filters.

A visual representation can help you understand the explanation.

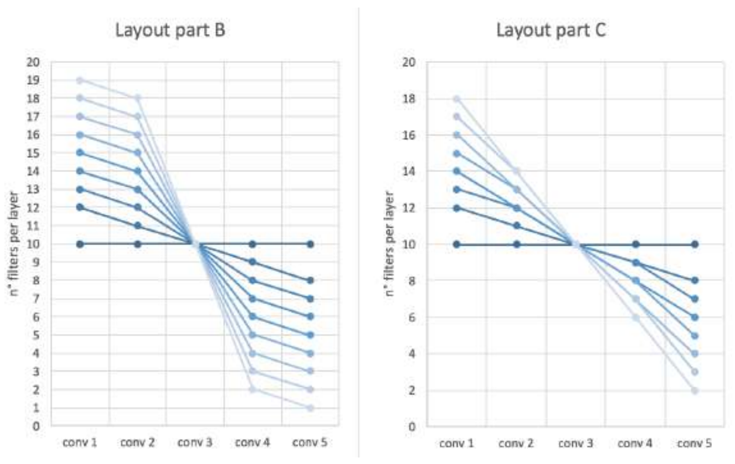


Figure 4.17: Visual representation of the difference between B and C

In none of the three cases, as you will notice in the subsections below, is it possible to identify a trend so it is not possible to make predictions and choose a good architecture a priori. Remember that the total number of filters is important but the arrangement of these must be tailored.

Part A

	Layout	RMSE	NRMSE	R^2
1$\mapsto$	**9*conv - 1*conv- mirror**	92.448	0.068	0.195
2$\mapsto$	**8*conv - 2*conv- mirror**	89.115	0.066	0.361
3$\mapsto$	**7*conv - 3*conv- mirror**	91.393	0.067	0.244
4$\mapsto$	**6*conv - 4*conv- mirror**	88.645	0.065	0.402
5$\mapsto$	**5*conv - 5*conv- mirror**	88.569	0.065	0.364
6$\mapsto$	**4*conv - 6*conv- mirror**	87.257	0.064	0.337
7$\mapsto$	**3*conv - 7*conv- mirror**	97.110	0.071	0.241
8$\mapsto$	**2*conv - 8*conv- mirror**	88.230	0.065	0.305
9$\mapsto$	**1*conv - 9*conv- mirror**	83.079	0.061	0.455

Table 4.25: Results for layout (part A) sensibility analysis

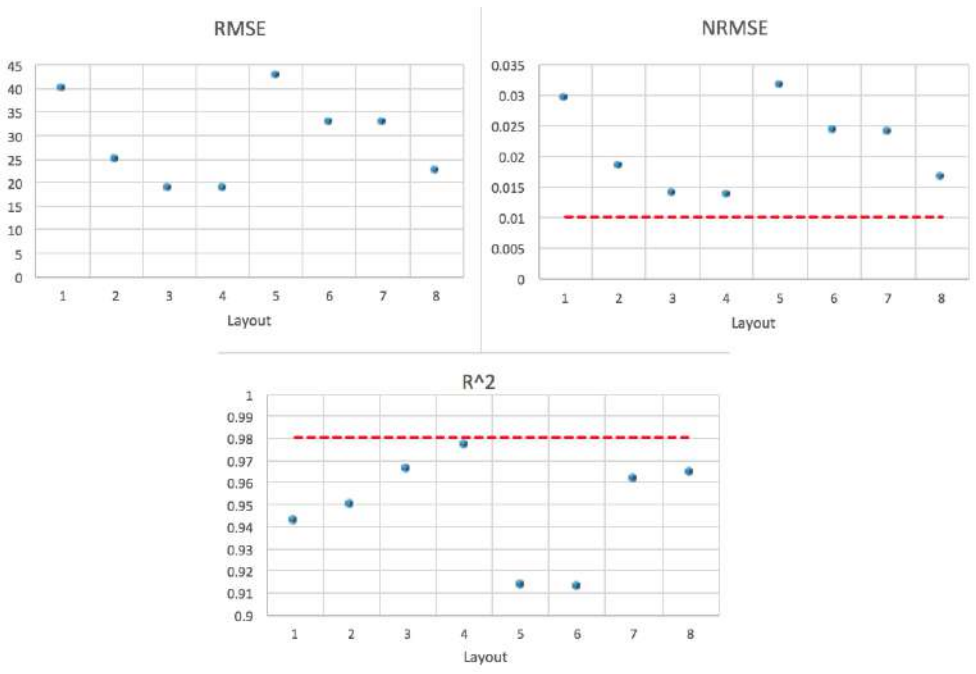


Figure 4.18: Graphs of results for layout (part A)

Part B

	Layout	RMSE	NRMSE	R^2
1↦	**10/10/10 - 10/10 - mirror**	39.890	0.029	0.943
2↦	**12/11/10 - 9/8 - mirror**	24.978	0.018	0.950
3↦	**13/12/10 - 8/7 - mirror**	17.899	0.013	0.975
4↦	**14/13/10 - 7/6 - mirror**	28.726	0.021	0.938
5↦	**15/14/10 - 6/5 - mirror**	33.404	0.025	0.949
6↦	**16/15/10 - 5/4 - mirror**	24.012	0.018	0.958
7↦	**17/16/10 - 4/3 - mirror**	28.027	0.021	0.937
8↦	**18/17/10 - 3/2 - mirror**	21.084	0.016	0.969
9↦	**19/18/10 - 2/1 - mirror**	26.067	0.019	0.954

Table 4.26: Results for layout (part B) sensibility analysis

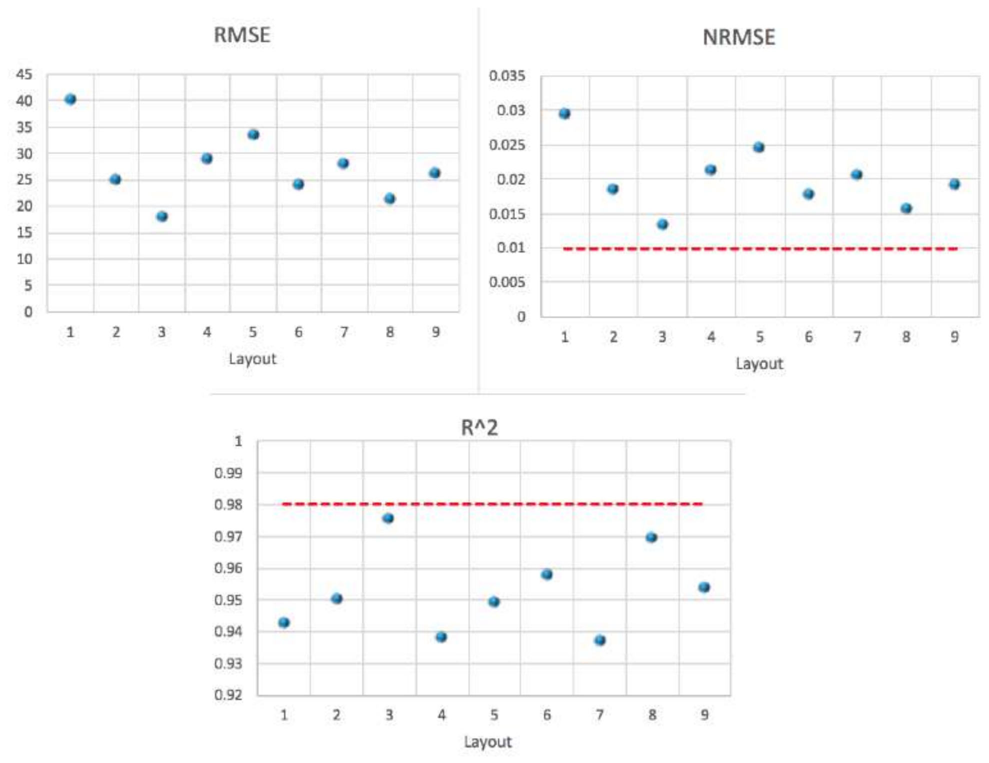


Figure 4.19: Graphs of results for layout (part B)

Part C

	Layout	RMSE	NRMSE	R^2
1↦	**10/10/10 - 10/10 - mirror**	39.890	0.029	0.943
2↦	**12/11/10 - 9/8 - mirror**	24.978	0.018	0.950
3↦	**13/12/10 - 9/7 - mirror**	18.847	0.014	0.966
4↦	**14/12/10 - 8/6 - mirror**	18.680	0.014	0.977
5↦	**15/13/10 - 8/5 - mirror**	42.684	0.031	0.913
6↦	**16/13/10 - 7/4 - mirror**	32.667	0.024	0.913
7↦	**17/14/10 - 7/3 - mirror**	32.662	0.024	0.962
8↦	**18/14/10 - 6/2 - mirror**	22.481	0.017	0.965

Table 4.27: Results for layout (part C) sensibility analysis

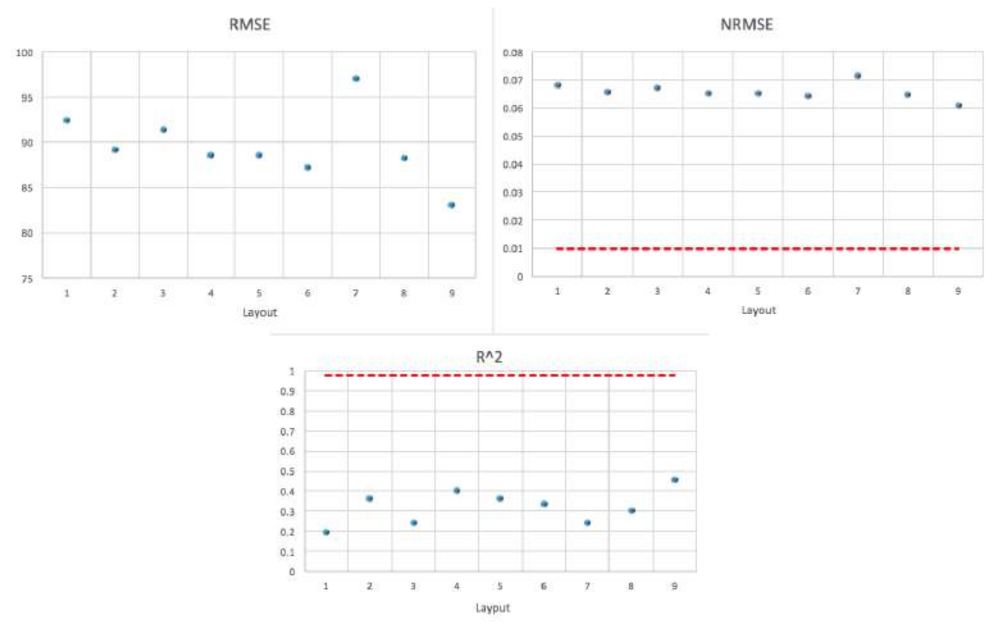


Figure 4.20: Graphs of results for layout (part C)

4.2 Optimised CNN

Thanks to the studies carried out in the section 4.1 , it is possible to identify an optimised convolutional neural network with the following parameters:

Fixed parameters	
Batch size	3
Normalisation	C
Final AF	linear
Numpy seed	9
Learning rate	0.0005
Patience.	15
Pooling	average
Cost function	MSE
Optimizer	Adam
Activation	Relu
Code dimension	72

Table 4.28: Fixed parameters for optimised CNN

As already mentioned, the choice of architecture is not dictated by a specific trend, only for the number of filters it is possible to affirm a general principle. So after a few attempts we got good results for the following:

```
#--ARCHITECTURE
input_img = (X_train_N.shape[1], X_train_N.shape[2],X_train_N.shape[3], X_train_N.shape[4])
encoder = Sequential(name='ENCODER')
encoder.add(Conv3D(14, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(Conv3D(12, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(Conv3D(10, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(AveragePooling3D(pool_size=(3,3,3)))
encoder.add(Conv3D(8, kernel_size=(3,3,3), activation='relu', padding='same'))
encoder.add(Conv3D(6, kernel_size=(3,3,3), activation='relu', padding='same'))
encoder.add(AveragePooling3D(pool_size=(5,5,3)))
print('encoder part architecture')
encoder.summary()
#---
autoencoder=Sequential(name='AUTOENCODER')
autoencoder.add(encoder)
autoencoder.add(UpSampling3D(size=(5, 5, 3)))
autoencoder.add(Conv3DTranspose(6, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(8, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(UpSampling3D(size=(3, 3, 3)))
autoencoder.add(Conv3DTranspose(10, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(12, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(14, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(1, kernel_size=(3, 3, 3), activation='linear', padding='same'))
print('complete model architecture')
autoencoder.summary()
```

CNN results are very satisfying. Think about the fact that PCA has an NRMSE of 0.01. The author recalls this CNN was made for a reduced mesh but also that the temperature variation is limited to the volume under examination. Therefore it can be considered a result worthy of comparison. In any case, the obtained result for complete mesh is also reported. It is noted that this has a lower but good accuracy; thus having sufficient time and computational resources it will be possible to search for the architecture that meets our needs, reasonably aspiring to a result similar to that for the reduced mesh.

Dim code	**RMSE**	**NRMSE**	**R^2**
72	9.5993	0.0071	0.9912

Table 4.29: Results for optimised CNN (reduced mesh)

Dim code	**RMSE**	**NRMSE**	**R^2**
1944	12.0036	0.0092	0.9182
36	12.0036	0.0092	0.9182
6	14.3854	0.0110	0.8765

Table 4.30: Results for optimised CNN (full mesh)

In the table for complete mesh, three tests with different code dimension have been presented for the following reasons: 1944 is the dimension obtained by training exactly the network described above, 36 is the dimension we want to reach to make a legitimate comparison with PCA as described below and 6 is the dimension to show that CNN does not have the same limitations of PCA.

36 - Code dimension CNN

Going back to CNN for the reduced mesh, the size of the code we reached by performing the analysis is 72. The PCA, on the other hand, manages to reach 40 modes (i.e. m-1 with m number of simulations for training). Thus, we run test cases to see the performance of a CNN (with optimized parameters) that arrives to a code of 36 elements. To do this, we manipulate the size of kernels and the amount of pooling layers. For example *ccc*333*cc*556 means 3 conv layers + 1 average pooling layer with kernel 3 3 3 + 2 conv layer + 1 average (5 5 6) and mirror.

	Layout	**RMSE**	**NRMSE**	**R^2**
1$\mapsto$	**ccc 3 3 3 cc 5 5 6**	10.6436	0.0078	0.9894
2$\mapsto$	**cc 3 3 3 c 5 5 2 cc 1 1 3**	19.7683	0.0145	0.9689
3$\mapsto$	**cc 3 3 3 c 5 5 3 cc 1 1 2**	24.7903	0.0182	0.9602
4$\mapsto$	**c 5 5 3 c 1 1 2 c 1 1 3 cc 3 3 1**	15.3471	0.0113	0.9778
5$\mapsto$	**ccc 15 15 9 cc 1 1 2**	35.8867	0.0264	0.8736
6$\mapsto$	**c 3 3 1 c 1 1 3 c 1 1 2 cc 5 5 3**	13.4682	0.0099	0.9862
7$\mapsto$	**c 5 5 1 c 1 1 2 c 1 1 3 cc 3 3 3**	12.3828	0.0091	0.9857

8↦	**ccc 5 5 6 cc 3 3 3**	11.0100	0.0081	0.9890
9↦	**cc 5 5 3 c 3 3 3 cc 1 1 2**	11.7453	0.0086	0.9868

Table 4.31: Results for optimised CNN (reduced mesh) - dim code 36

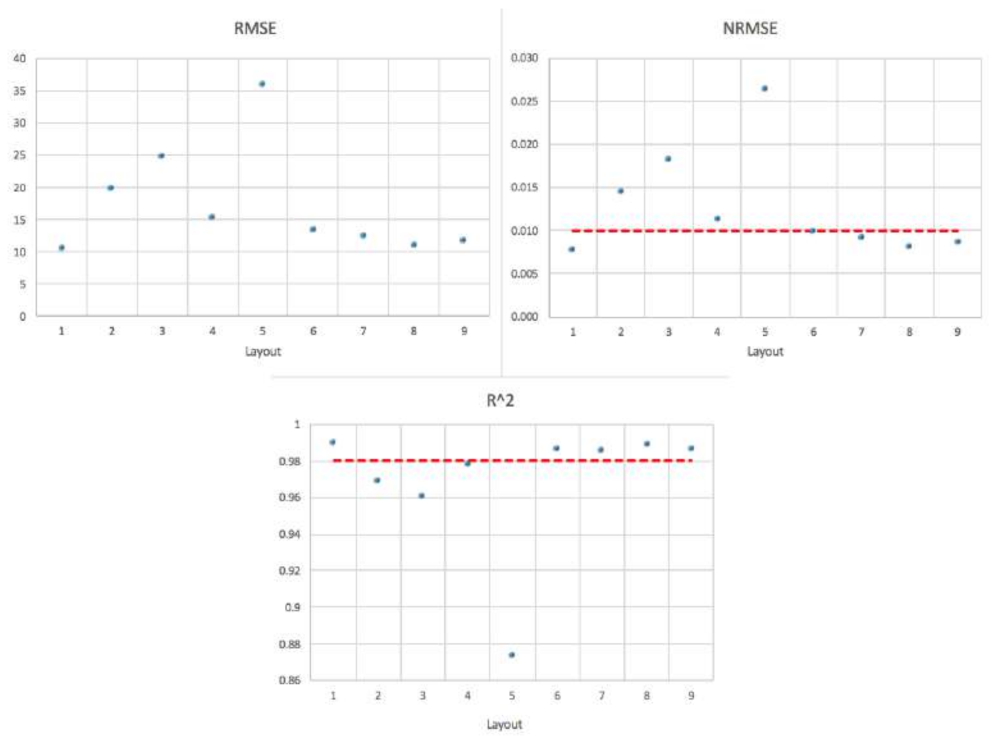


Figure 4.21: Graphs of results for optimisation (dim code 36)

Even if the size of the code has been reduced to 36, it is possible to reach a very high accuracy (R2 = 0.9894 and NRMSE = 0.0078) and always better than PCA accuracy.

Code dimension and accuracy

We want to go further and do some test cases to understand if CNN could give excellent results even for codes with much smaller dimensions. See the following considerations (tab. 4.32).

Code dim	**Layout**	**RMSE**	**NRMSE**	**R^2**
72	ccc 3 3 3 cc 5 5 3	9.5993	0.0071	0.9912
54	ccc 3 3 3 cc 5 5 4	15.1499	0.0111	0.9862
36	ccc 3 3 3 cc 5 5 6	10.6436	0.0078	0.9894
24	ccc 3 3 9 cc 5 5 3	10.7691	0.0079	0.9887
18	cc 3 3 6 c 1 1 3 cc 5 5 3	13.6028	0.0100	0.9853
12	cc 3 3 9 c 1 1 6 cc 5 5 3	11.8877	0.0087	0.9867
6	cc 3 3 6 c 1 1 3 c 5 5 3 c 1 1 4	16.7578	0.0123	0.9734
4	cc 3 3 6 c 1 1 3 cc 5 5 3 (18 17 10 3 2)	14.8703	0.0109	0.9806

Table 4.32: Results for code dimension analysis

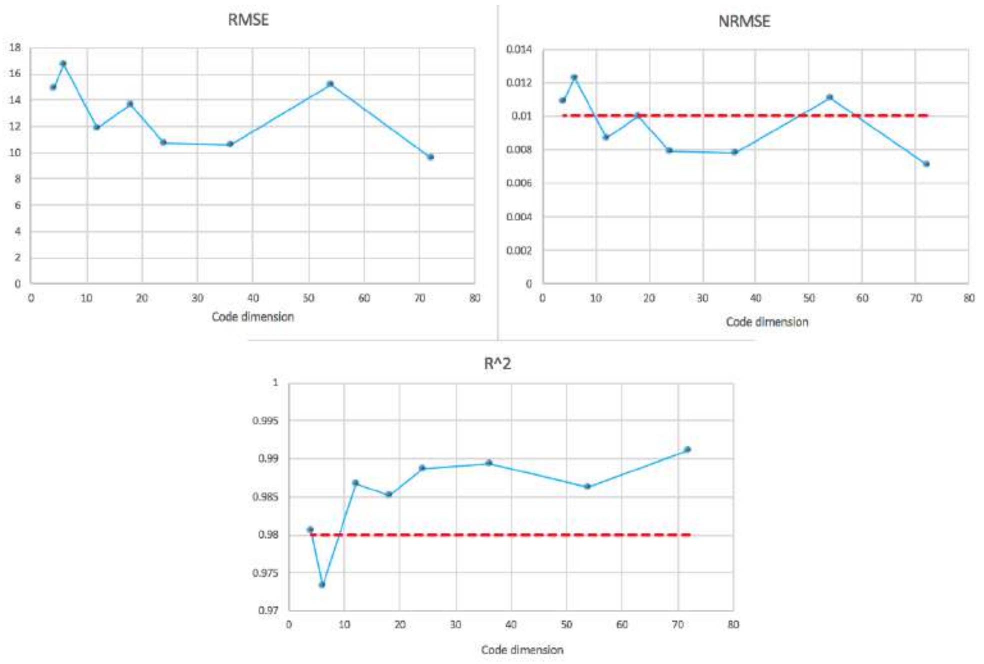


Figure 4.22: Graphs of results for code dimension analysis

It is easy to see from the results that the accuracy is very high even if the code size is definitely less than 36.

Number of training simulations

Up to now we have excluded 4 simulations over a total of 45 simulations performed in order to reserve them for testing, therefore using only 41 simulations for network training. This was done to have an equal comparison between convolutional neural network and PCA. Now we want to see if CNN is able to achieve excellent results excluding more or less simulations from training (using architecture number 1 of table 4.31).

n°Tsims/45	RMSE	NRMSE	R^2
44	13.4690	0.0100	0.9671
43	9.7857	0.0073	0.9850
42	11.2887	0.0083	0.9851
41	10.6436	0.0078	0.9894
40	23.5759	0.0176	0.9636
39	10.8979	0.0081	0.9871
38	14.7993	0.0110	0.9735
37	14.0888	0.0104	0.9845
36	16.6801	0.0124	0.9698
35	16.9484	0.0125	0.9694
34	13.1715	0.0097	0.9836
33	12.0838	0.0090	0.9825
32	17.2444	0.0129	0.9735
31	13.7219	0.0103	0.9760
30	13.3052	0.0099	0.9781
29	13.7994	0.0103	0.9788
28	15.4626	0.0115	0.9712
27	14.8844	0.0111	0.9741
26	16.0602	0.0120	0.9734
25	21.8316	0.0163	0.9621
24	13.5164	0.0101	0.9778
23	13.9347	0.0104	0.9790
22	13.9186	0.0104	0.9776
21	14.0661	0.0105	0.9774
20	17.1515	0.0128	0.9677
19	16.4944	0.0123	0.9712
18	15.4792	0.0116	0.9725
17	15.4618	0.0115	0.9740
16	17.0356	0.0127	0.9718
15	14.7697	0.0110	0.9756
14	17.1332	0.0128	0.9689
13	21.0428	0.0157	0.9600
12	14.9800	0.0112	0.9743
11	18.1770	0.0136	0.9628
10	23.8475	0.0178	0.9350
9	26.8041	0.0200	0.9252
8	20.8562	0.0156	0.9579
7	17.7008	0.0132	0.9661

6	26.4253	0.0197	0.9234
5	37.4695	0.0280	0.8664
4	28.9187	0.0216	0.9064
3	30.8710	0.0231	0.8904
2	26.1671	0.0195	0.9195
1	132.6666	0.0990	0.1310

Table 4.33: Results for number of training sims analysis

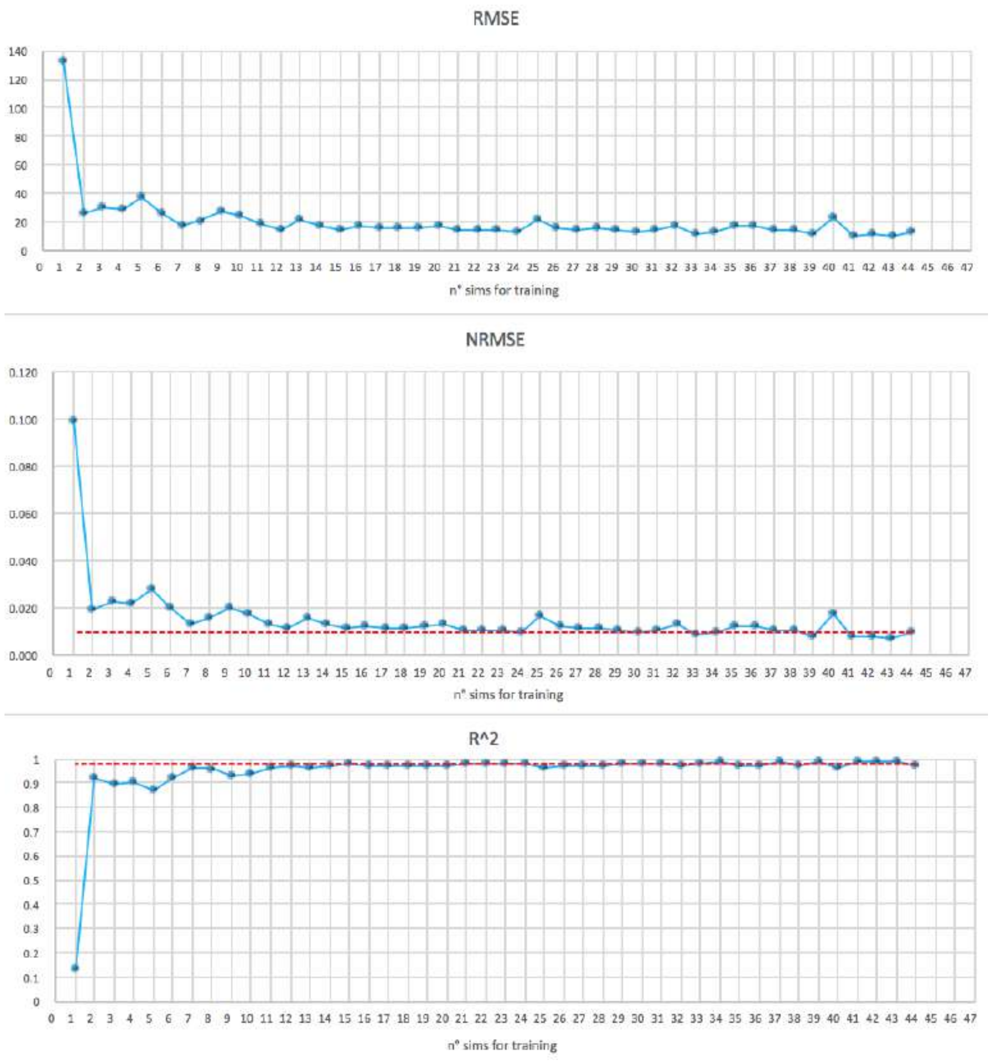


Figure 4.23: Graphs of results for number of training sims analysis

Convolutional network maintains good performance even with a much lower number of training simulations. This is explained if the features necessary for encoding / decoding can already be captured by looking at the few simulations used for training. Finally, we performed the next analysis to understand if the same performances were obtained whatever the simulations chosen as training or if it was necessary to choose wisely. Not being able to consider all the possible combinations, only a few are studied. The author is aware that in this context, couple selection rules are necessary but here we only want to demonstrate the need to properly select the couple. Indeed, in the following results it is clearly noted that the attention in the choice is fundamental for the performance of the network.

Train sims	**RMSE**	**NRMSE**	**R^2**
[**1,28**]	37.1520	0.0278	0.8618
[**45,29**]	37.3948	0.0280	0.8477
[**3,24**]	79.3874	0.0592	0.3205
[**18,32**]	64.1300	0.0479	0.5416
[**5,45**]	51.1451	0.0382	0.7447
[**6,19**]	30.2079	0.0226	0.8979
[**2,17**]	26.6671	0.0199	0.9212
[**26,41**]	26.1671	0.0195	0.9195
[**37,12**]	34.7675	0.0260	0.8687

Table 4.34: Results for different couple of training sims

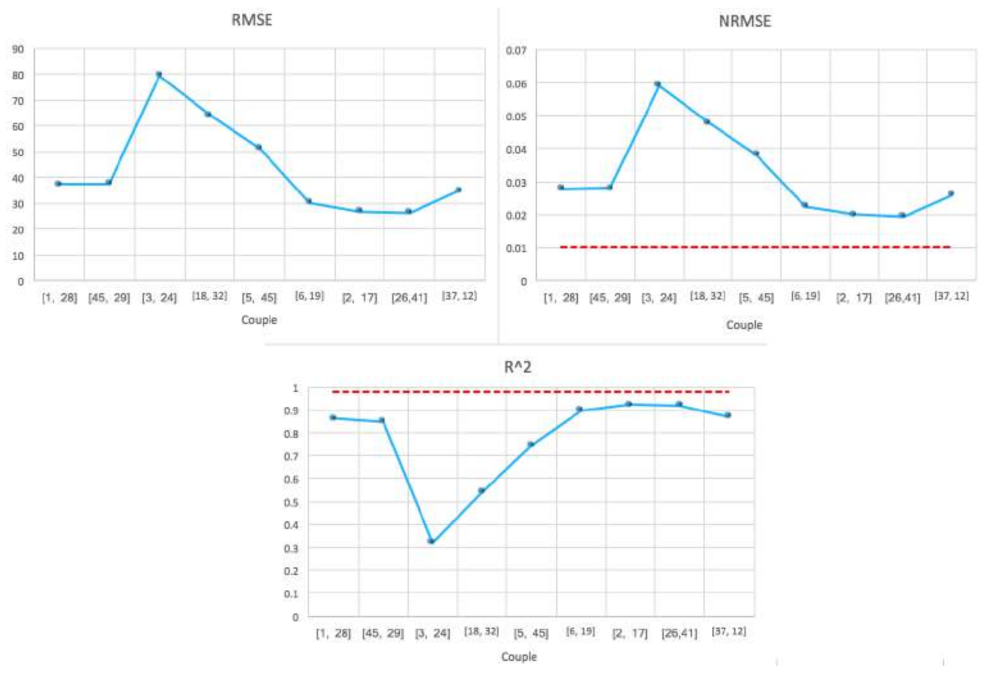


Figure 4.24: Graphs of results for different couple of training sims

Conclusion

Thanks to this study we have understood that CNN is a valid and usable input dimensionality reduction technique in the context of CFD on MILD combustion systems. It is at least as accurate as PCA, which already had excellent results. We also remember the advantages that CNN has over the PCA: Autencoders are capable of modelling complex non linear functions, while the PCA is essentially a linear transformation. PCA is able to recognize features that are invariant in space, on the contrary autoencoders learn how to recognize this feature regardless of where it is in the image.

The project did not stop at the only comparison with PCA but went further. Sensibility analysis has taught us that we could never hope to have excellent performance if we do not adequately choose the following parameters: activation function of the last layer, normalization, learning rate, batch size, patience, cost function, type of pooling, optimizer and the total number of filters. While there are other parameters that must be revised ad hoc such as numpy seed, activation functions and the architecture (n° of convolutional layers, kernel, distribution of filters ..).

A further step was taken in wanting to overcome the limits dictated by the comparison with PCA. Therefore, we found that CNN provides good results both by reducing the encoded image size and providing fewer training simulations for the network than PCA as long as these are conscientiously chosen.

I conclude by considering the completion of the ROM as a future study, associating the most suitable interpolation method with the proposed autoencoder.

Appendix A

PCA - Principal component analysis

Mathematical background

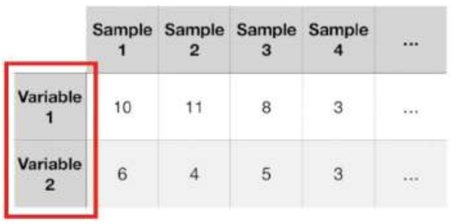
In this section, w'll see some mathematical notations used in PCA.

Given a matrix $\mathbf{Y}$ of size $(m * n)$, Proper Orthogonal Decomposition (POD) seeks $\mathbf{Z}$ of size $(m * q)$ and A of size $(n * k)$ with $k < n$, such that the functional $f(\mathbf{Z},\mathbf{A}) = 1\|\mathbf{Y} - \mathbf{Z}\mathbf{A}^T\|^2$ is minimized, subject to $\mathbf{A}^T\mathbf{A} = \mathbf{I}$, where I is the identity matrix. This problem can be solved by computing the singular value decomposition (SVD) of the matrix $\mathbf{Y}$, which corresponds to finding the eigenvectors and eigenvalues of the matrix $\mathbf{C} = \frac{1}{m-1}\mathbf{Y}^T\mathbf{Y}$. A low-rank approximation of $\mathbf{Y}$ is found as follows $\mathbf{Y} \approx \mathbf{Z}\mathbf{A}^T = \mathbf{Y}\mathbf{A}\mathbf{A}^T$, where the columns of $\mathbf{A}$ of size $(n * q)$ are the POD modes and $\mathbf{Z}$ of size $(m * k)$ is the matrix of POD coefficients. Each column of $\mathbf{Z}$ are the k coefficients for the retained k POD modes so that one particular simulation, or row of $\mathbf{Y}$, can be expressed as $\mathbf{y}(\mathbf{x}) = \sum_{i=1}^{k} \mathbf{a}_i z_i(\mathbf{x})$.

An easy example

I will present what is behind PCA approach with a 2D example. In Figure A.1, there is our dataset. We have to shift the data (centring) and we try to draw a line that fits the data. To do this PCA projects the data onto it and finds the line that maximised the distances $(d_1, d_2, ...d_n)$ from the projected points to the origin or better maximised the SS. The line is called PC1 i.e. first principal component. (Fig.A.3).

$$d_1^2 + d_2^2 + d_3^2 + ... = \text{sum of squared distances } = \text{SS} \tag{A.1}$$



	Sample 1	Sample 2	Sample 3	Sample 4	...
Variable 1	10	11	8	3	...
Variable 2	6	4	5	3	...

Figure A.1: Data for PCA example

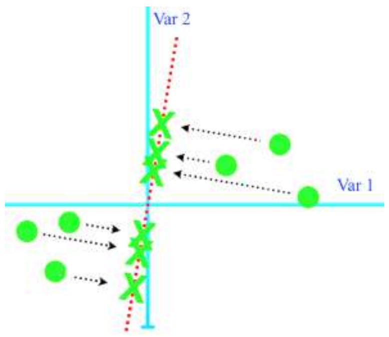


Figure A.2: Projection

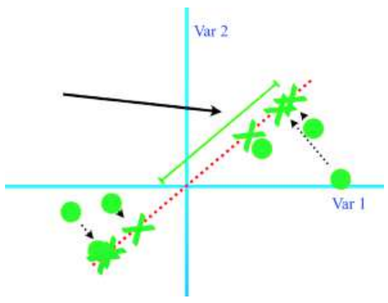


Figure A.3: Distances

Principal components are linear combination of variables. A principal component corresponds to an eigenvector and the SS_{PC1} is the eigenvalue for PC1. The squared root of eigenvalue is the singular value. The proportional contribute (normalised) of each variables in PC1 is called loading score (Figure A.4). PC2 is perpendicular to PC1. If we rotate everything so that PC1 becomes horizontal we obtain PCA plot.

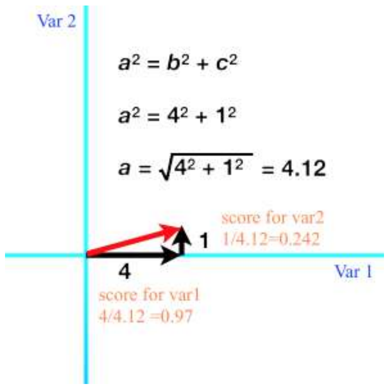


Figure A.4: Scores

This is what PCA done using Singular value decomposition (SVD). If you want to find how much a dataset is influenced by a specific PC you have to calculate Variation like in figure A.5 obtaining a percentage value. The graphical representation of these percentages is called Scree plot (Figure A.6 on the left).

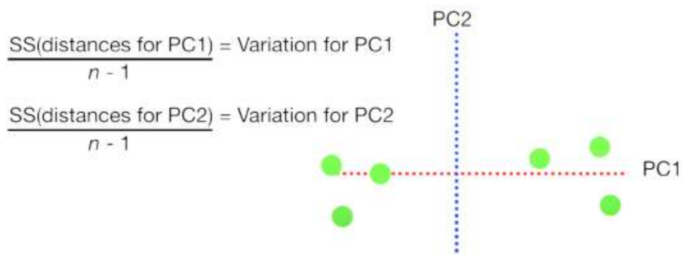

SS(distances for PC1) / n - 1 = Variation for PC1
SS(distances for PC2) / n - 1 = Variation for PC2
PC2
PC1

Figure A.5: Variations

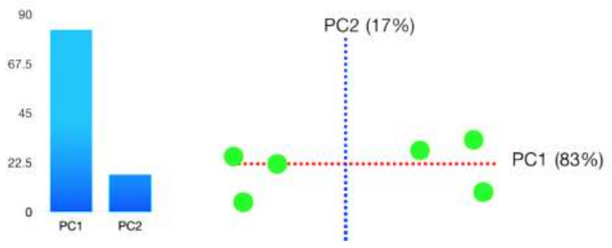


Figure A.6: Scree plot

In a similar way it is possible to work with multidimensional dataset and do some consideration like in Figure A.7.

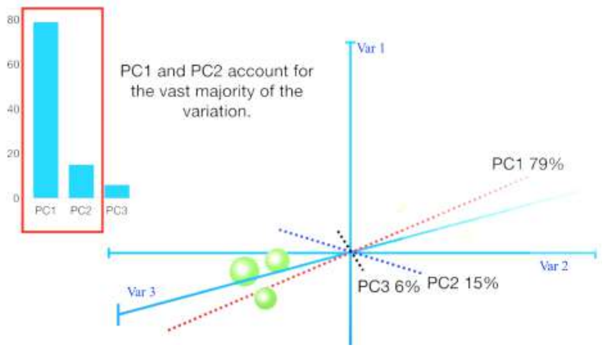


Figure A.7: Multidimensional dataset

Appendix B

Scripts

B.1 Code for preprocessing

B.1.1 CNN5_del_err.py

```
#input: grid points e T field (csv)
#1. remove points with error from grid points
#2. delete points which belong to symmetry plane and out of domain (<0 in the 3 direction )
#3.add point for cooling tube
#4. mirror mesh
#5. save  index_ok_npy;  n_cooling_npy, GRID_X/Y/Z
import numpy as np
#---------------------------------------------------------------------------------------------------------
#-------------------------------------------------INPUT ---------------------------------------------------
#---------------------------------------------------------------------------------------------------------
FIELD_T=1  #don't change!  #1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
GRID_CSV='/Users/Fabi/Desktop/clusterMac/fabiola'
GRID_name='grid-points.csv'
Y_CSV='/Users/Fabi/Desktop/clusterMac/fabiola'
Y_name='Y.csv'
index_ok_npy='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/GP_index_ok.npy'
n_cooling_npy='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/n_cooling.npy'
n_GP_npy='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/n_GP_original.npy'
GRID_X='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/X.npy'
GRID_Y='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/Y.npy'
GRID_Z='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/Z.npy'
summary_GP_txt='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/summary_GP.txt'
#---------------------------------------------------------------------------------------------------------
#-------------------------------------------------FUCTION ---------------------------------------------------
#---------------------------------------------------------------------------------------------------------
```

```
#----function: csv --> matrix
def readCSV(path, name):
try:
print('Reading training matrix..')
X = np.genfromtxt(path + '/' + name, delimiter= ',')
except OSError:
print('Could not open/read the selected file: ' + name)
exit()
return X
#-----------------------------------------------------------------------------------------------------------
#-------------------------------------------------START -----------------------------------------------------
#-----------------------------------------------------------------------------------------------------------
#--------------------------------------------import GRID POINTS----------------------------------------------
GP_mat=readCSV(GRID_CSV,GRID_name)
GP_mat=np.delete(GP_mat, (0), axis=0)
print(f' max GP dir x: {max(GP_mat[:,0])}')   #0.35
print(f' max GP dir y: {max(GP_mat[:,1])}')   #0.35
print(f' max GP dir z: {max(GP_mat[:,2])}')   #0.7000002381
print(f' min GP dir x: {min(GP_mat[:,0])}')   #0.001760897526
print(f' min GP dir y: {min(GP_mat[:,1])}')   #-7.347880795e-19
print(f' min GP dir z: {min(GP_mat[:,2])}')   #-0.149999998
print(f'number of GP is {len(GP_mat)}') #216360
#-------------------------------------------- export
n_GP =len(GP_mat)
np.save(n_GP_npy,n_GP)
#----------------------------------------- import Tmep output ------------------------------------------
arrayYcsv=readCSV(Y_CSV,Y_name)
arrayYcsv=np.delete(arrayYcsv, (0), axis=0)
print(arrayYcsv.shape)  #(45, 1730880)
Ytot=arrayYcsv[:,len(GP_mat)*(FIELD_T-1): len(GP_mat) *FIELD_T]
print(f'min Y_tot (ie for all sims) {np.amin(Ytot)}')
print(f'max Y_tot (ie for all sims) {np.amax(Ytot)}')
print(f'shape Y_tot: {Ytot.shape}')  #(45, 216360)
#------------------°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°________________________
#-----------------------------------------modified field---------------------------------------------
#-------------------°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°_________________________
#-------------------------------------------- identify points corresponding to wrong output
index_err=[]
for s in range(0,Ytot.shape[0],1):
for i in range(0,len(GP_mat),1):
if Ytot[s,i]<=345:
index_err.append(i)
print(f'len index {len(index_err)}')
GP_index_err=list(set(index_err)) #delate double points
print(f'len GP_index_err  :{len(GP_index_err)}')
print(f'len GP_index (total) :{len(GP_mat)}')
print(f'%points with error: {len(GP_index_err)/len(GP_mat)*100}')
GP_index_no_err= list(range(0,len(GP_mat),1))
for i in GP_index_err:
```

```
GP_index_no_err.remove(i)
#-------------------------------------------  delete points which belong
#to symmetry plane and out of domain (<0 in the 3 direction )
index_ok=[]
for i in GP_index_no_err:
if GP_mat[i,0]!=GP_mat[i,1]:
if GP_mat[i,0]>=0 and GP_mat[i,1]>=0 and GP_mat[i,2]>=0:
index_ok.append(i)
print(f'len index up0:{len(index_ok)}')
GP_index_ok=list(set(index_ok)) #delete double
print(f'len GP_index_ok :{len(GP_index_ok)}')
print(f'len GP_index (total) :{len(GP_mat)}')
print(f'%points ok: {len(GP_index_ok)/len(GP_mat)*100}')
#-------------------------------------------- export
np.save(index_ok_npy,GP_index_ok)
#------------------------------------------- create lists X,Y,Z
list_X=[]
list_Y=[]
list_Z=[]
for g in GP_index_ok:
list_X.append(GP_mat[g,0])
list_Y.append(GP_mat[g,1])
list_Z.append(GP_mat[g,2])
print(f'len list_X with GP_index_ok: {len(list_X)}')
#------------------------------------------ add point for cooling tube
z=0
for ii in np.arange(0.21,0.29,0.005):
for jj in np.arange(0.21,0.29,0.005):
if (ii-0.25)**2 + (jj-0.25)**2 - 0.04**2 <=0 and ii>=jj:
for kk in np.arange(0.07,0.7,0.01):
list_X.append(ii)
list_Y.append(jj)
list_Z.append(kk)
z=z+1
print(f'points add for semitube : {z}')
print(f'len list_X with semitube: {len(list_X)}')
#-------------------------------------------- export
np.save(n_cooling_npy,z)
#-------------------------------------------- mirror mesh
for jj in range(0,len(list_X),1):
list_X.append(list_Y[jj])
list_Y.append(list_X[jj])
list_Z.append(list_Z[jj])
print(f'len mirror list_X: {len(list_X)}')
print(f'max of final list_X: {max(list_X)}')
print(f'max of final list_Y: {max(list_Y)}')
print(f'max of final list_Z: {max(list_Z)}')
print(f'min of final list_X: {min(list_X)}')
print(f'min of final list_Y: {min(list_Y)}')
print(f'min of final list_Z: {min(list_Z)}')
#------------------------------------------- EXPORT list_X,list_Y,list_Z
np.save(GRID_X,list_X)
np.save(GRID_Y,list_Y)
np.save(GRID_Z,list_Z)
#-------------------------------------------- grid plot
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
fig = plt.figure('original')
ax = fig.add_subplot(111, projection='3d')
im=ax.scatter(list_X, list_Y, list_Z, c='b', s=1.5)
ax.set_xlabel('X ')
ax.set_ylabel('Y ')
ax.set_zlabel('Z ')
plt.show()
#---------------------------------------------------------------------------------------------------------
#------------------------------------------------SAVE TXT----------------------------------------------
#---------------------------------------------------------------------------------------------------------
with open(summary_GP_txt,'w') as txt:
txt.write(f'original number of grid points: {len(GP_mat)}\n\n')
txt.write(f'final number of grid points: {len(list_X)}\n\n')
txt.write(f'n GP for semitube (cooling): {z}\n\n')
txt.write(f'max value in X {max(list_X)} in Y {max(list_Y)} in Z {max(list_Z)}\n\n')
txt.write(f'min value in X {min(list_X)} in Y {min(list_Y)} in Z {min(list_Z)}\n\n')
txt.write(f'GP_index_ok (good GP - semi mesh) {len(GP_index_ok)}\n\n')
txt.write(f'%points with error from csv: {len(GP_index_err)/len(GP_mat)*100}\n\n')
txt.write(f'max of final list_X: {max(list_X)}\n\n')
txt.write(f'max of final list_Y: {max(list_Y)}\n\n')
txt.write(f'max of final list_Z: {max(list_Z)}\n\n')
txt.write(f'min of final list_X: {min(list_X)}\n\n')
txt.write(f'min of final list_Y: {min(list_Y)}\n\n')
txt.write(f'min of final list_Z: {min(list_Z)}\n\n')
txt.close()
```

B.1.2 CNN5read.py

```
# use after CNN5_del_err
# 1. read  X,Y,Z  data and field (es.T)
# 2. delete: wrong variable values, var. values with coordinate <0
#     and which belong to simmetry plane (in according to CNN5_del_err)
# 3. add a value for cooling semi-tube and it mirrors the mesh
# 4. for a choosen field (including all sims), it creates a matrix (that is an
#      interpolation of original mesh on a uniform PP mesh). This one is
#      good for CNN input
#    This one is a matrix which element are values of the field
#    (as colour of a 3D image)
# NOTA= value for cooling tube = mean
# NOTA2= this scripts is usefull to create OUT, the second part (ARRAy_Y creation)
```

```
#is possible to do in another script (light_griddata.py)
#-----------------------------------------------------------------------------------------------------------
#--------------------------------------------------INPUT ---------------------------------------------------
#-----------------------------------------------------------------------------------------------------------
FIELD=1   #1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
cell= 0.0067  # dim cell in meters
#existing file
Y_CSV='/Users/Fabi/Desktop/clusterMac/fabiola'
Y_name='Y.csv'
index_ok_npy='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/GP_index_ok.npy'
n_cooling_npy='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/n_cooling.npy'
n_GP_npy='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/n_GP_original.npy'
GRID_X='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/X.npy'
GRID_Y='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/Y.npy'
GRID_Z='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/Z.npy'
#path
OUT_T='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_T.npy'
OUT_CH4='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_CH4.npy'
OUT_O2='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_O2.npy'
OUT_CO2='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_CO2.npy'
OUT_H2O='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_H2O.npy'
OUT_OH='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_OH.npy'
OUT_CO='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_CO.npy'
OUT_NO='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_NO.npy'
cell_folder=f'/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/cell{cell}'
ARRAY_Y_npy= cell_folder+'/ARRAY_Y'+f'/ARRAY_FIELD{FIELD}.npy'
#------
import numpy as np
#-----------------------------------------------------------------------------------------------------------
#--------------------------------------------------FUCTION -----------------------------------------------
#-----------------------------------------------------------------------------------------------------------
# csv ---> np.array
def readCSV(path, name):
try:
print('Reading training matrix..')
X = np.genfromtxt(path + '/' + name, delimiter= ',')
except OSError:
print('Could not open/read the selected file: ' + name)
exit()
return X
#---------------------------°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°-----------------
#----------------------------------------------------START ----------------------------------------------
#-------------------------°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°---------------------
#----------------------------------------------------------------------------------------------------
#---------------------------------------------import grid -------------------------------------------
#----------------------------------------------------------------------------------------------------
n_GP_original = np.load(n_GP_npy)
list_X = np.load(GRID_X)
list_Y = np.load(GRID_Y)
```

```
list_Z = np.load(GRID_Z)
GP_index_ok = np.load(index_ok_npy)
z = np.load(n_cooling_npy)
#---------------------------------------------------------------------------------------------------
#---------------------------------------------OUTPUT ----------------------------------------------
#--------------------------------------------------------------------------------------------------
###########--------------------- import output field -----------------------------------------
arrayYcsv=readCSV(Y_CSV,Y_name)
arrayYcsv=np.delete(arrayYcsv, (0), axis=0)  #delete header
print(arrayYcsv.shape)  #(45, 1730880)
#Ytot = array with results (of my field) of all 45 sims
Ytot=arrayYcsv[:,n_GP_original*(FIELD-1): n_GP_original *FIELD]
print(Ytot.shape)  #(45, 216360)
###########---------delete wrong output and output out of domain (see CNN5_del_err) -------
Ytot_light=np.zeros((Ytot.shape[0], len(GP_index_ok)))
i_ok=0
for g in GP_index_ok:
Ytot_light[:,i_ok]=Ytot[:,g]
i_ok=i_ok+1
print(f'shape Ytot_light array {Ytot_light.shape}')   #(45, 190479)
print(f'min Ytot_light array (min value of selected field among all simulations): {np.amin(Ytot_light)}')
print(f'max Ytot_light array (max value of selected field among all simulations):  {np.amax(Ytot_light)}')
############-------------add a value in cooling tube (semi)---------------------------------
value=np.mean(Ytot_light)
semi_tube=np.zeros((Ytot.shape[0], z))
semi_tube=semi_tube+value
Ytube=np.concatenate((Ytot_light,semi_tube),axis=1)
print(f'shape Ytube array {Ytube.shape}')
############-------------------------- mirror ------------------------------------------------
val=Ytube.shape[1] #GP index ok+z
Ymirror=np.zeros((Ytot.shape[0], val*2))
for ii in range(0,val,1):
Ymirror[:,ii]=Ytube[:,ii]
Ymirror[:,val+ii]=Ytube[:,ii]
print(f'shape Ymirror array {Ymirror.shape}')  #(45 , val*2)
#############--------------------------array to list -----------------------------------------
list_output=Ymirror.tolist()
print(f'len list_output {len(list_output)}')  #45 each of which has val*2 GP
##############-------------------EXPORT list_output-----------------------------------------
#  1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
if FIELD==1:
list_output_T=list_output
np.save(OUT_T,list_output_T)
if FIELD==2:
list_output_CH4=list_output
np.save(OUT_CH4,list_output_CH4)
if FIELD==3:
list_output_O2=list_output
np.save(OUT_O2,list_output_O2)
```

```
if FIELD==4:
list_output_CO2=list_output
np.save(OUT_CO2,list_output_CO2)
if FIELD==5:
list_output_H2O=list_output
np.save(OUT_H2O,list_output_H2O)
if FIELD==6:
list_output_OH=list_output
np.save(OUT_OH,list_output_OH)
if FIELD==7:
list_output_CO=list_output
np.save(OUT_CO,list_output_CO)
if FIELD==8:
list_output_NO=list_output
np.save(OUT_NO,list_output_NO)
#-----------------------------------------------------------------------------------------------
#----------------------------------------- MESH and GRIDDATA-----------------------------------
#-----------------------------------------------------------------------------------------------
#§§§§§§§§------ create meshgrid parallelepipedo PP uniform (one for all field)-----------------
#NOTA: it’s bigger than 35,35,70 to become more easy to manage in CNN
x = np.arange(0, 0.36, cell)
y = np.arange(0, 0.36, cell)
z = np.arange(0, 0.72, cell)
xx, yy, zz = np.meshgrid(x,y,z)
print(f’shape di xx è {xx.shape}, yy è {yy.shape}, zz è {zz.shape}’)
#§§§§§§§§---Array of interpolated values (over uniform mesh) of output field (GRIDDATA)--------
from scipy.interpolate import griddata as gd
ARRAY_GD=np.zeros((len(list_output),xx.shape[0],xx.shape[1],xx.shape[2]))
print(f’shape array_Ygd before {ARRAY_GD.shape}’)
for sim in range(0,len(list_output),1):
list_1output=list_output[sim]
array_Y_gd = gd((list_X,list_Y,list_Z), list_1output, (xx,yy,zz), method=’nearest’)
ARRAY_GD[sim]=array_Y_gd
print(f’shape array_Ygd {array_Y_gd.shape}’)  #follow meshgrid es.(35,35,70)
print(f’shape array_Ygd after {ARRAY_GD.shape}’)
#§§§§§§§§----------------------EXPORT ARRAY_Y --------------------------------------------
#  1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
if FIELD==1:
ARRAY_T=ARRAY_GD
np.save(ARRAY_Y_npy,ARRAY_T)
print(f’ARRAY_T shape {ARRAY_T.shape}’)
if FIELD==2:
ARRAY_CH4=ARRAY_GD
np.save(ARRAY_Y_npy,ARRAY_CH4)
print(f’ARRAY_CH4 shape {ARRAY_CH4.shape}’)
if FIELD==3:
ARRAY_O2=ARRAY_GD
np.save(ARRAY_Y_npy,ARRAY_O2)
print(f’ARRAY_O2 shape {ARRAY_O2.shape}’)
```

```
if FIELD==4:
ARRAY_CO2=ARRAY_GD
np.save(ARRAY_Y_npy,ARRAY_CO2)
print(f'ARRAY_CO2 shape {ARRAY_CO2.shape}')
if FIELD==5:
ARRAY_H2O=ARRAY_GD
np.save(ARRAY_Y_npy,ARRAY_H2O)
print(f'ARRAY_H2O shape {ARRAY_H2O.shape}')
if FIELD==6:
ARRAY_OH=ARRAY_GD
np.save(ARRAY_Y_npy,ARRAY_OH)
print(f'ARRAY_OH shape {ARRAY_OH.shape}')
if FIELD==7:
ARRAY_CO=ARRAY_GD
np.save(ARRAY_Y_npy,ARRAY_CO)
print(f'ARRAY_CO shape {ARRAY_CO.shape}')
if FIELD==8:
ARRAY_NO=ARRAY_GD
np.save(ARRAY_Y_npy,ARRAY_NO)
print(f'ARRAY_NO shape {ARRAY_NO.shape}')
```

B.1.3 test_read.py

```
#script usefull to test CNN5_read
#----------------------------------------------------------------------------------------------------------
#----------------------------------------------INPUT-------------------------------------------------------
#----------------------------------------------------------------------------------------------------------
FIELD=1
sim=1
cell=0.0067 #according to CNN5_read
#don't modify
GRID_X='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/X.npy'
GRID_Y='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/Y.npy'
GRID_Z='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/GRID/Z.npy'
OUT_T='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_T.npy'
OUT_CH4='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_CH4.npy'
OUT_O2='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_O2.npy'
OUT_CO2='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_CO2.npy'
OUT_H2O='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_H2O.npy'
OUT_OH='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_OH.npy'
OUT_CO='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_CO.npy'
OUT_NO='/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/OUT/OUT_NO.npy'
cell_folder=f'/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/cell{cell}'
ARRAY_Y_npy= cell_folder+'/ARRAY_Y'+f'/ARRAY_FIELD{FIELD}.npy'
#----- pkgs
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
```

```
print(f'field= {FIELD} where 1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO')
print(f'simulation number {sim}')
#-----------------------------------------------------------------------------------------------------
#-------------------------------------------- meshgrid -----------------------------------------------
#-----------------------------------------------------------------------------------------------------
#---- MESHGRID parallelepipedo PP
x = np.arange(0, 0.36, cell)
y = np.arange(0, 0.36, cell)
z = np.arange(0, 0.72, cell)
xx, yy, zz = np.meshgrid(x,y,z)
fig = plt.figure('plot mesh PP')
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xx, yy, zz, s=1)
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
#plt.show()
#-----------------------------------------------------------------------------------------------------
#-----------------------------------------------ARRAY_Y (griddata) -----------------------------------
#-----------------------------------------------------------------------------------------------------
#------------------IMPORT ARRAY IN OUTPUT A CNN5 READ
print('load array...')
#  1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
if FIELD==1:
ARRAY_T = np.load(ARRAY_Y_npy)
print('field=T')
print(f'shape ARRAY_T {ARRAY_T.shape}')
ARRAY_Y=ARRAY_T
if FIELD==2:
ARRAY_CH4=np.load(ARRAY_Y_npy)
print('field=CH4')
print(f'ARRAY_CH4 shape {ARRAY_CH4.shape}')
ARRAY_Y=ARRAY_CH4
if FIELD==3:
ARRAY_O2=np.load(ARRAY_Y_npy)
print('field=O2')
print(f'shape ARRAY_O2 {ARRAY_O2.shape}')
ARRAY_Y=ARRAY_O2
if FIELD==4:
ARRAY_CO2=np.load(ARRAY_Y_npy)
print('filed=CO2')
print(f'ARRAY_CO2 shape {ARRAY_CO2.shape}')
ARRAY_Y=ARRAY_CO2
if FIELD==5:
ARRAY_H2O=np.load(ARRAY_Y_npy)
print('field=H2O')
print(f'ARRAY_H2O shape {ARRAY_H2O.shape}')
ARRAY_Y=ARRAY_H2O
if FIELD==6:
```

```
ARRAY_OH=np.load(ARRAY_Y_npy)
print('filed=OH')
print(f'ARRAY_CH4 shape {ARRAY_OH.shape}')
ARRAY_Y=ARRAY_OH
if FIELD==7:
ARRAY_CO=np.load(ARRAY_Y_npy)
print(f'ARRAY_CO shape {ARRAY_CO.shape}')
ARRAY_Y=ARRAY_CO
if FIELD==8:
ARRAY_NO=np.load(ARRAY_Y_npy)
print(f'ARRAY_NO shape {ARRAY_NO.shape}')
ARRAY_Y=ARRAY_NO
print(f'shape ARRAY_Y {ARRAY_Y.shape}')
print(f'min value of ARRAY_Y (so of all simulations - griddata value) is {np.amin(ARRAY_Y)}')
print(f'max value of ARRAY_Y (so of all simulations - griddata value) is {np.amax(ARRAY_Y)}')
#------------------------------GRIDDATA: estrarre la griddata e fare il plot di conferma
array_Y_gd=ARRAY_Y[sim-1]
print(f'shape array_Ygd {array_Y_gd.shape}')
print(f'number of grid points (griddata)= {array_Y_gd.shape[0]*array_Y_gd.shape[1]*array_Y_gd.shape[2]}')
print(f'min value of ARRAY_Y_gd (so of the {sim} simulation - griddata value) is {np.amin(array_Y_gd)}')
print(f'max value of ARRAY_Y_gd (so of the {sim} simulation - griddata value) is {np.amax(array_Y_gd)}')
from scipy.interpolate import griddata as gd
fig = plt.figure('interpolazione')
ax = fig.add_subplot(111, projection='3d')
im=ax.scatter(xx, yy, zz, c=array_Y_gd, cmap='jet', vmin=np.amin(array_Y_gd), vmax=np.amax(array_Y_gd),s=1)
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
cax = fig.add_axes([0.05, 0.1, 0.02, 0.8])
fig.colorbar(im, orientation='vertical', cax=cax)
#plt.show()
#-----------------------------------------------------------------------------------------------------
#-------------------------------------------------ORIGINAL--------------------------------------------------
#-----------------------------------------------------------------------------------------------------
#--------------------------------REAL: plot del reale output
print('load list X,Y,Z ...')
list_X = np.load(GRID_X)
list_Y = np.load(GRID_Y)
list_Z = np.load(GRID_Z)
print('load list_output_X ...')
#  1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
if FIELD==1:
list_output_T = np.load(OUT_T)
list_output=list_output_T
if FIELD==2:
list_output_CH4 = np.load(OUT_CH4)
list_output=list_output_CH4
if FIELD==3:
list_output_O2 = np.load(OUT_O2)
```

```
list_output=list_output_O2
if FIELD==4:
list_output_CO2 = np.load(OUT_CO2)
list_output=list_output_CO2
if FIELD==5:
list_output_H2O = np.load(OUT_H2O)
list_output=list_output_H2O
if FIELD==6:
list_output_OH = np.load(OUT_OH)
list_output=list_output_OH
if FIELD==7:
list_output_CO = np.load(OUT_CO)
list_output=list_output_CO
if FIELD==8:
list_output_NO = np.load(OUT_NO)
list_output=list_output_NO
print(f'min value of list_output (so of all simulation - real value) is {np.amin(list_output)}')
print(f'max value of list_output (so of all simulation - real value) is {np.amax(list_output)}')
list_1output=list_output[sim-1]
print(f'len(list_1output) (so number of real grid point) : {len(list_1output)}')
print(f'min value of list_1output (so of {sim} simulation - real value) is {min(list_1output)}')
print(f'max value of list_1output (so of {sim} simulation - real value) is {max(list_1output)}')
fig = plt.figure('original')
ax = fig.add_subplot(111, projection='3d')
im=ax.scatter(list_X, list_Y, list_Z, c=list_1output, cmap='jet', ...
vmin=min(list_1output), vmax=max(list_1output),s=1)
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
cax = fig.add_axes([0.05, 0.1, 0.02, 0.8])
fig.colorbar(im, orientation='vertical', cax=cax)
plt.show()
```

B.2 Code for the core of the project

B.2.1 CNN5.py

```
#use after CNN5read.py
# you obtain autoencoder and encoder
#--- pkgs
from numpy.random import seed
seed(9)   #avoid different solution for the same architecture
import os
import time
Day= time.strftime("_%d_%m_%Y")
print(f'Today is {Day}')
```

```
Time= time.strftime("_%H_%M_%S")
print(f'Time is {Time}')
import numpy as np
#------------------------------------------------------------------------------------------------------
#----------------------------------------------INPUT---------------------------------------------------
#------------------------------------------------------------------------------------------------------
FIELD=1
Norm='C'
cell=0.0067
row_train=[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27,29,30,31,32,33,34,35,36,37,38,
40,41,42,43,44,45]
row_test=[1,22,28,39]
epochs=500
batch_size=3
#don't modify
#---
cell_folder=f'/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/Scell{cell}'
autoencoder_path_folder=cell_folder+'/AUTOENCODER'+f'/AUTOENCODER_{FIELD}/AUTOENCODER_{FIELD}_'
+str(FIELD)+Day+Time
results_file=cell_folder+f'/AUTOENCODER/AUTOENCODER_{FIELD}/results.txt'
row_train_npy=cell_folder+f'/AUTOENCODER/AUTOENCODER_{FIELD}/AUTOENCODER_{FIELD}_'+str(FIELD)
+Day+Time+'/row_train.npy'
row_test_npy=cell_folder+f'/AUTOENCODER/AUTOENCODER_{FIELD}/AUTOENCODER_{FIELD}_'+str(FIELD)
+Day+Time+'/row_test.npy'
ARRAY_Y_npy= cell_folder+'/ARRAY_Y'+f'/ARRAY_FIELD{FIELD}.npy'
#---
norm_folder= cell_folder+'/ARRAY_Y'+f'/Norm{Norm}'
Mean_npy= norm_folder+f'/mean/mean_{FIELD}.npy'
Std_npy= norm_folder+f'/std/std_{FIELD}.npy'
#------------------------------------------------------------------------------------------------------
#-------------------------------------------------START -----------------------------------------------
#------------------------------------------------------------------------------------------------------
os.mkdir(autoencoder_path_folder)
os.chdir(autoencoder_path_folder)
print(f'os get dir: {os.getcwd()}')
#------------------------------------------------------------------------------------------------------
#------------------------------------------------get input field---------------------------------------
#------------------------------------------------------------------------------------------------------
print(f'field= {FIELD} where 1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO')
print('load array...')
#  1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
if FIELD==1:
ARRAY_T = np.load(ARRAY_Y_npy)
print(f'shape ARRAY_T {ARRAY_T.shape}')
ARRAY_Y=ARRAY_T
if FIELD==2:
ARRAY_CH4=np.load(ARRAY_Y_npy)
print(f'ARRAY_CH4 shape {ARRAY_CH4.shape}')
ARRAY_Y=ARRAY_CH4
```

```
if FIELD==3:
ARRAY_O2=np.load(ARRAY_Y_npy)
print(f'shape ARRAY_O2 {ARRAY_O2.shape}')
ARRAY_Y=ARRAY_O2
if FIELD==4:
ARRAY_CO2=np.load(ARRAY_Y_npy)
print(f'ARRAY_CO2 shape {ARRAY_CO2.shape}')
ARRAY_Y=ARRAY_CO2
if FIELD==5:
ARRAY_H2O=np.load(ARRAY_Y_npy)
print(f'ARRAY_H2O shape {ARRAY_H2O.shape}')
ARRAY_Y=ARRAY_H2O
if FIELD==6:
ARRAY_OH=np.load(ARRAY_Y_npy)
print(f'ARRAY_CH4 shape {ARRAY_CH4.shape}')
ARRAY_Y=ARRAY_OH
if FIELD==7:
ARRAY_CO=np.load(ARRAY_Y_npy)
print(f'ARRAY_CO shape {ARRAY_CO.shape}')
ARRAY_Y=ARRAY_CO
if FIELD==8:
ARRAY_NO=np.load(ARRAY_Y_npy)
print(f'ARRAY_NO shape {ARRAY_NO.shape}')
ARRAY_Y=ARRAY_NO
print(f'shape ARRAY_Y {ARRAY_Y.shape}')
print(f'{len(row_train)} sims over { ARRAY_Y.shape[0]} for training')
print(f'{len(row_test)} sims over { ARRAY_Y.shape[0]-len(row_train)} for test ')
print(f'min value of ARRAY_Y (so of all simulations - griddata value) is {np.amin(ARRAY_Y)}')
print(f'max value of ARRAY_Y (so of all simulations - griddata value) is {np.amax(ARRAY_Y)}')
#-- normalization
MEAN=np.mean(ARRAY_Y)
print(f'mean= {MEAN}')
STD=np.std(ARRAY_Y)
print(f'std={STD}')
ARRAY_Y_N=(ARRAY_Y-MEAN)/STD
print(f'shape ARRAY_Y_N {ARRAY_Y_N.shape}')
print(f'min value of ARRAY_Y_N (so of all simulations - griddata value) normalized is {np.amin(ARRAY_Y_N)}')
print(f'max value of ARRAY_Y_N (so of all simulations - griddata value) normalized is {np.amax(ARRAY_Y_N)}')
#--------------EXPORT normalization -----------------------
np.save(Mean_npy,MEAN)
np.save(Std_npy,STD)
#-- define X_train and X_test
X_train_N=np.zeros((len(row_train),ARRAY_Y_N.shape[1], ARRAY_Y_N.shape[2],ARRAY_Y_N.shape[3]))
jj=0
for i in row_train:
X_train_N[jj]= ARRAY_Y_N[i-1]
jj=jj+1
X_test_N=np.zeros((len(row_test),ARRAY_Y_N.shape[1], ARRAY_Y_N.shape[2],ARRAY_Y_N.shape[3]))
jj=0
```

```
for i in row_test:
X_test_N[jj]= ARRAY_Y_N[i-1]
jj=jj+1
print(f'shape X_train_N {X_train_N.shape}') #(41, 54, 54, 108)
print(f'shape X_test_N {X_test_N.shape}') #(4, 54, 54, 108)
X_train_N = np.reshape(X_train_N, (X_train_N.shape[0], X_train_N.shape[1], X_train_N.shape[2],X_train_N.shape[3], 1))
X_test_N = np.reshape(X_test_N, (X_test_N.shape[0], X_test_N.shape[1], X_test_N.shape[2],X_test_N.shape[3], 1))
print(f'reshape X_train_N {X_train_N.shape}') #(41, 54, 54, 108, 1)
print(f'reshape X_test_N {X_test_N.shape}') #(41, 54, 54, 108, 1)
print(f'min value of X_train_N (so of all train simulations) normalized is {np.amin(X_train_N)}')
print(f'max value of X_train_N (so of all train simulations) normalized is {np.amax(X_train_N)}')
print(f'min value of X_test_N (so of all test simulations) is {np.amin(X_test_N)}')
print(f'max value of X_test_N (so of all test simulations) is {np.amax(X_test_N)}\n')
#--------------EXPORT dataset -----------------------
np.save(row_train_npy,row_train)
np.save(row_test_npy,row_test)
#----------------------------------------------------------------------------------------------------------
#----------------------------------------START CNN AUTOENCODER-----------------------------------
#----------------------------------------------------------------------------------------------------------
#---pkgs
import os
os.environ ['KMP_DUPLICATE_LIB_OK'] = 'True'
import keras
from keras.models import Sequential
from keras.layers import Conv3D, UpSampling3D, Conv3DTranspose, AveragePooling3D
from keras.metrics import MeanSquaredError
from keras.callbacks import EarlyStopping
from keras.callbacks import TensorBoard
import numpy as np
import matplotlib.pyplot as plt
#--ARCHITECTURE
input_img = (X_train_N.shape[1], X_train_N.shape[2],X_train_N.shape[3], X_train_N.shape[4])
encoder = Sequential(name='ENCODER')
encoder.add(Conv3D(14, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(Conv3D(12, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(Conv3D(10, kernel_size=(3,3,3), activation='relu', padding='same', input_shape=input_img))
encoder.add(AveragePooling3D(pool_size=(3,3,3)))
encoder.add(Conv3D(8, kernel_size=(3,3,3), activation='relu', padding='same'))
encoder.add(Conv3D(6, kernel_size=(3,3,3), activation='relu', padding='same'))
encoder.add(AveragePooling3D(pool_size=(5,5,3)))
print('encoder part architecture')
encoder.summary()
#---
autoencoder=Sequential(name='AUTOENCODER')
autoencoder.add(encoder)
autoencoder.add(UpSampling3D(size=(5, 5, 3)))
autoencoder.add(Conv3DTranspose(6, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(8, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(UpSampling3D(size=(3, 3, 3)))
autoencoder.add(Conv3DTranspose(10, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(12, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(14, kernel_size=(3, 3, 3), activation='relu', padding='same'))
autoencoder.add(Conv3DTranspose(1, kernel_size=(3, 3, 3), activation='linear', padding='same'))
print('complete model architecture')
autoencoder.summary()
#--OPTIMIZATION
autoencoder.compile(loss=keras.losses.MeanSquaredError(),
optimizer=keras.optimizers.Adam(lr=0.0005),
metrics=['mean_squared_error'])
print('fine opt')
#--TRAINING
earlyStopping = EarlyStopping(monitor='val_loss', patience=15, verbose=1, mode='min')
history=autoencoder.fit(X_train_N, X_train_N,
epochs=epochs, #epoch refers to one cycle through the full training dataset
batch_size=batch_size, #
shuffle=True,
validation_data=(X_test_N, X_test_N),
callbacks=[TensorBoard(log_dir='/tmp/autoencoder'), earlyStopping])
#--SAVE
encoder.save('encoder')
autoencoder.save('autoencoder')
print('encoder and autoencoder saved')
#----------------------------------------------------------------------------------------------------------
#--------------------------------------------PLOT MSE AND LOSS--------------------------------------
#----------------------------------------------------------------------------------------------------------
print(history.history.keys())
# summarize history for mean_squared_error
plt.figure('summarize history for mean_squared_error')
plt.plot(history.history['mean_squared_error'])
plt.plot(history.history['val_mean_squared_error'])
plt.title('model mean_squared_error')
plt.ylabel('log mean_squared_error')
plt.xlabel('epoch')
plt.yscale('log')
plt.xticks(np.arange(0, epochs, 0.1*epochs))
plt.grid(axis='x', color='0.95')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('plot_mean_squared_error.pdf')
#plt.show()
# summarize history for loss
plt.figure('summarize history for loss')
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('log loss')
plt.xlabel('epoch')
plt.yscale('log')
plt.xticks(np.arange(0, epochs, 0.1*epochs))
plt.grid(axis='x', color='0.95')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('plot_loss.pdf')
#plt.show()
#-----------------------------------------------------------------------------------------------------
#-----------------------------------------------SAVE TXT----------------------------------------------
#-----------------------------------------------------------------------------------------------------
with open('summary.txt','w') as txt:
txt.write(f'os get dir: {os.getcwd()}\n\n')
txt.write(f'sim for train: {len(row_train)}\n')
txt.write(f'sim for train: {row_train}\n\n')
txt.write(f'sim for test: {len(row_test)}\n')
txt.write(f'sim for test: {row_test}\n\n\n')
txt.write(f'epochs: {epochs}\n\n')
txt.write(f'batch_size: {batch_size}\n\n')
autoencoder.summary(print_fn=lambda x: txt.write(x + '\n'))
encoder.summary(print_fn=lambda x: txt.write(x + '\n\n\n'))
txt.close()
```

B.3 Code for post-processing

B.3.1 CNN5load.py

```
# use after CNN5.py
# ---- prediction
# load array and normalisation
# define X_train and X_test (normalised and not) and flatten for plot
# load of autoencoder and encoder
# predict X_train_N and X_test_N wit autoencoder, then the "decoded" will be denormalised and flattened
# ---- post-prediction
# parity plot (normaliseded and not)
# NMRSE, RMSE , R^2 (normalised and not) -->plot
# %relative error (only for not normalised) --> plot
#--- pkgs
import os
import numpy as np
from numpy.random import seed
seed(9)   #avoid different solution for the same architecture
import os
os.environ ['KMP_DUPLICATE_LIB_OK'] = 'True'
import keras
from keras.metrics import MeanSquaredError
import matplotlib.pyplot as plt
#-----------------------------------------------------------------------------------------------------
#---------------------------------------------INPUT---------------------------------------------------
#-----------------------------------------------------------------------------------------------------
```

```
FIELD=1
cell=0.0067
Norm='C'
#to be the same of CNN5
row_train=[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27,29,30,
31,32,33,34,35,36,37,38,40,41,42,43,44,45]
row_test=[1,22,28,39]
name_autoencoder_folder= '/AUTOENCODER_1_1_17_08_2020_16_59_13'
#don't modify
#---
cell_folder=f'/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/Scell{cell}'
autoencoder_path_folder= cell_folder+f'/AUTOENCODER/AUTOENCODER_{FIELD}' + name_autoencoder_folder
autoencoder_name='autoencoder'
encoder_name='encoder'
row_train_npy=autoencoder_path_folder+'/row_train.npy'
row_test_npy=autoencoder_path_folder+'/row_test.npy'
ARRAY_Y_npy= cell_folder+'/ARRAY_Y'+f'/ARRAY_FIELD{FIELD}.npy'
#---
norm_folder= cell_folder+'/ARRAY_Y'+f'/Norm{Norm}'
Mean_npy= norm_folder+f'/mean/mean_{FIELD}.npy'
Std_npy= norm_folder+f'/std/std_{FIELD}.npy'
#-----------------------------------------------------------------------------------------------------------
#----------------------------------------------START--------------------------------------------------------
#-----------------------------------------------------------------------------------------------------------
os.chdir(autoencoder_path_folder)
print(f'os get dir: {os.getcwd()}')
print(f'field= {FIELD} where 1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO')
#-----------------------------------------------------------------------------------------------------------
#------------------------------------------------ORIGINAL---------------------------------------------------
#-----------------------------------------------------------------------------------------------------------
print('load array...')
#  1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
if FIELD==1:
ARRAY_T = np.load(ARRAY_Y_npy)
print(f'shape ARRAY_T {ARRAY_T.shape}')
ARRAY_Y=ARRAY_T
if FIELD==2:
ARRAY_CH4=np.load(ARRAY_Y_npy)
print(f'ARRAY_CH4 shape {ARRAY_CH4.shape}')
ARRAY_Y=ARRAY_CH4
if FIELD==3:
ARRAY_O2=np.load(ARRAY_Y_npy)
print(f'shape ARRAY_O2 {ARRAY_O2.shape}')
ARRAY_Y=ARRAY_O2
if FIELD==4:
ARRAY_CO2=np.load(ARRAY_Y_npy)
print(f'ARRAY_CO2 shape {ARRAY_CO2.shape}')
ARRAY_Y=ARRAY_CO2
if FIELD==5:
```

```
ARRAY_H2O=np.load(ARRAY_Y_npy)
print(f'ARRAY_H2O shape {ARRAY_H2O.shape}')
ARRAY_Y=ARRAY_H2O
if FIELD==6:
ARRAY_OH=np.load(ARRAY_Y_npy)
print(f'ARRAY_CH4 shape {ARRAY_CH4.shape}')
ARRAY_Y=ARRAY_OH
if FIELD==7:
ARRAY_CO=np.load(ARRAY_Y_npy)
print(f'ARRAY_CO shape {ARRAY_CO.shape}')
ARRAY_Y=ARRAY_CO
if FIELD==8:
ARRAY_NO=np.load(ARRAY_Y_npy)
print(f'ARRAY_NO shape {ARRAY_NO.shape}')
ARRAY_Y=ARRAY_NO
print(f'shape ARRAY_Y {ARRAY_Y.shape}')
print(f'{len(row_train)} sims over { ARRAY_Y.shape[0]} for training')
print(f'{len(row_test)} sims over { ARRAY_Y.shape[0]-len(row_train)} for test ')
print(f'min value of ARRAY_Y (so of all simulations - griddata value) is {np.amin(ARRAY_Y)}')
print(f'max value of ARRAY_Y (so of all simulations - griddata value) is {np.amax(ARRAY_Y)}')
#-- normalization
MEAN=np.load(Mean_npy)
STD=np.load(Std_npy)
print(f'mean= {MEAN}')
print(f'std={STD}')
ARRAY_Y_N=(ARRAY_Y-MEAN)/STD
print(f'shape ARRAY_Y_N {ARRAY_Y_N.shape}')
print(f'min value of ARRAY_Y_N (so of all simulations - griddata value) normalized is {np.amin(ARRAY_Y_N)}')
print(f'max value of ARRAY_Y_N (so of all simulations - griddata value) normalized is {np.amax(ARRAY_Y_N)}')
#-- define X_train and X_test (normalised)
row_train=np.load(row_train_npy)
row_test=np.load(row_test_npy)
X_train_N=np.zeros((len(row_train),ARRAY_Y_N.shape[1], ARRAY_Y_N.shape[2],ARRAY_Y_N.shape[3]))
jj=0
for i in row_train:
X_train_N[jj]= ARRAY_Y_N[i-1]
jj=jj+1
X_test_N=np.zeros((len(row_test),ARRAY_Y_N.shape[1], ARRAY_Y_N.shape[2],ARRAY_Y_N.shape[3]))
jj=0
for i in row_test:
X_test_N[jj]= ARRAY_Y_N[i-1]
jj=jj+1
print(f'shape X_train_N {X_train_N.shape}')
print(f'shape X_test_N {X_test_N.shape}')
X_train_N = np.reshape(X_train_N, (X_train_N.shape[0], X_train_N.shape[1],
 X_train_N.shape[2],X_train_N.shape[3], 1))
X_test_N = np.reshape(X_test_N, (X_test_N.shape[0], X_test_N.shape[1], X_test_N.shape[2],X_test_N.shape[3], 1))
print(f'reshape X_train_N {X_train_N.shape}')
print(f'reshape X_test_N {X_test_N.shape}')
print(f'min value of X_train_N (so of all train simulations) normalized is {np.amin(X_train_N)}')
print(f'max value of X_train_N (so of all train simulations) normalized is {np.amax(X_train_N)}')
print(f'min value of X_test_N (so of all test simulations) is {np.amin(X_test_N)}')
print(f'max value of X_test_N (so of all test simulations) is {np.amax(X_test_N)}\n')
#de-normalization for postprocessing
X_train= X_train_N*STD +MEAN #de-norm
X_test= X_test_N*STD +MEAN #de-norm.
print(f'min value of X_train_N (so of all train simulations) is {np.amin(X_train)}')
print(f'max value of X_train_N (so of all train simulations) is {np.amax(X_train)}')
print(f'min value of X_test (so of all test simulations) is {np.amin(X_test)}')
print(f'max value of X_test (so of all test simulations) is {np.amax(X_test)}\n')
#flatten for postprocessing
X_train_N_1D=X_train_N.flatten()
X_test_N_1D=X_test_N.flatten()
X_train_1D=X_train.flatten()
X_test_1D=X_test.flatten()
print(f'shape X_train_N_1D: {X_train_N_1D.shape}')
print(f'shape X_test_N_1D: {X_test_N_1D.shape}')
print(f'shape X_train_1D: {X_train_1D.shape}')
print(f'shape X_test_1D: {X_test_1D.shape}\n')
#----------------------------------------------------------------------------------------------------------
#-------------------------------------------------AUTOENCODER PREDICTION-----------------------------------
#----------------------------------------------------------------------------------------------------------
#---LOAD AUTOENCODER AND ENCODER
from keras.models import load_model
print('autoencoder loading..')
autoencoder = load_model(autoencoder_name, custom_objects={'mean_squared_error': 'mean_squared_error'})
autoencoder.summary()
print('encoder loading..')
encoder = load_model(encoder_name, custom_objects={'mean_squared_error': 'mean_squared_error'})
encoder.summary()
#---------------------------------------------------------------------------
#---- PREDICTION AUTOENCODER
print('now i will predict')
X_train_pred_N = autoencoder.predict(X_train_N)
X_test_pred_N = autoencoder.predict(X_test_N)
print('autoencoder used\n')
print(f'shape X_train_pred_N: {X_train_pred_N.shape}')
print(f'shape X_test_pred_N: {X_test_pred_N.shape}')
print(f'min value of X_train_pred_N (so of all train simulations) normalized is {np.amin(X_train_pred_N)}')
print(f'max value of X_train_pred_N (so of all train simulations) normalized is {np.amax(X_train_pred_N)}')
print(f'min value of X_test_pred_N (so of all test simulations) normalized is {np.amin(X_test_pred_N)}')
print(f'max value of X_test_pred_N (so of all test simulations) normalized is {np.amax(X_test_pred_N)}\n')
#----- de-normalization for post processing
X_test_pred=X_test_pred_N*STD +MEAN
X_train_pred=X_train_pred_N*STD +MEAN
print(f'min value of X_train_pred (so of all train simulations) is {np.amin(X_train_pred)}')
print(f'max value of X_train_pred (so of all train simulations) is {np.amax(X_train_pred)}')
print(f'min value of X_test_pred (so of all test simulations) is {np.amin(X_test_pred)}')
print(f'max value of X_test_pred (so of all test simulations) is {np.amax(X_test_pred)}\n')
#----- flatten of predicted for post-processing
print('flatten...')
X_test_pred_N_1D=X_test_pred_N.flatten()
X_train_pred_N_1D=X_train_pred_N.flatten()
print('flatten (normalized value) ...')
X_test_pred_1D=X_test_pred.flatten()
X_train_pred_1D=X_train_pred.flatten()
print(f'shape X_test_pred_1D: {X_test_pred_1D.shape}')
print(f'shape X_train_pred_1D: {X_train_pred_1D.shape}')
#-----------------------------------------------------------------------------------------------------
#-------------------------------------------------POST PREDICTION----------------------------------------
#-----------------------------------------------------------------------------------------------------
#-------------------------------------parity plot--------------------------------------
#---train
plt.figure(f'Parity plot train')
plt.scatter(X_train_1D, X_train_1D, color='r',s=2)
plt.scatter(X_train_1D, X_train_pred_1D, color='b', marker='X',s=2)
plt.title('parity plot train')
plt.ylabel('X (predicted)')
plt.xlabel('X (experimental)')
lineStart = min(X_train_1D)
lineEnd = max(X_train_1D)
plt.plot([lineStart, lineEnd], [lineStart, lineEnd], 'k-', color = 'r',lw=1) #line X_test/X_test
Max_xP=np.amax(ARRAY_Y)
plt.plot([0, Max_xP*1.05], [0, Max_xP], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, Max_xP*0.95], [0, Max_xP], 'k-', color = 'k',lw=1) #-5%
#plt.show()
plt.savefig('parity_plot_train.pdf')
#---test
plt.figure(f'Parity plot test')
plt.scatter(X_test_1D, X_test_1D, color='r',s=2)
plt.scatter(X_test_1D, X_test_pred_1D, color='b',s=2)
plt.title('parity plot test ')
plt.ylabel('X (predicted)')
plt.xlabel('X (experimental)')
lineStart = min(X_test_1D)
lineEnd = max(X_test_1D)
plt.plot([lineStart, lineEnd], [lineStart, lineEnd], 'k-', color = 'r',lw=1) #line X_test/X_test
Max_xP=np.amax(ARRAY_Y)
plt.plot([0, Max_xP*1.05], [0, Max_xP], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, Max_xP*0.95], [0, Max_xP], 'k-', color = 'k',lw=1) #-5%
#plt.show()
plt.savefig('parity_plot_test.pdf')
#---------------------------------PercentageError_test----------------------------------------
#NOTA: the script assigns automatically zero value for error to grid points=0
ZERO=0
for e in X_test_1D:
if e==0:
```

```
ZERO=ZERO+1
print(f'zeri={ZERO}, i.e. how many points we do not see the real relative error because they are zeros')
#---calculate perc. relative error
err_perc_test=[]
for ii in  range(0, len(X_test_1D),1):
if X_test_1D[ii]==0:
err_perc_test.append(0)
else:
err_perc_test.append(100 * abs(X_test_1D[ii] - X_test_pred_1D[ii]) / X_test_1D[ii])
err_perc_train=[]
for ii in  range(0, len(X_train_1D),1):
if X_train_1D[ii]==0:
err_perc_train.append(0)
else:
err_perc_train.append(100 * abs(X_train_1D[ii] - X_train_pred_1D[ii]) / X_train_1D[ii])
#----PercentageError_test plot
gp_test=np.arange(0,len(err_perc_test),1)
plt.figure('PercentageError_test',figsize=(10,4))
plt.plot(gp_test, err_perc_test, 'o-', color='b')
plt.plot([0, len(err_perc_test)], [5, 5], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, len(err_perc_test)], [-5, -5], 'k-', color = 'k',lw=1) #-5%
plt.title('PercentageError_test')
plt.ylabel('PercentageError')
plt.xlabel('grid points')
plt.xticks(np.arange(0, len(err_perc_test), len(err_perc_test)/len(row_test)))
plt.grid(axis='x', color='k')
plt.savefig('PercentageError_test.pdf')
#----PercentageError_train plot
gp_train=np.arange(0,len(err_perc_train),1)
plt.figure('PercentageError_train',figsize=(15,4))
plt.plot(gp_train, err_perc_train, 'o-', color='c')
plt.plot([0, len(err_perc_train)], [5, 5], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, len(err_perc_train)], [-5, -5], 'k-', color = 'k',lw=1) #-5%
plt.title('PercentageError_train')
plt.ylabel('PercentageError')
plt.xlabel('grid points')
plt.xticks(np.arange(0, len(err_perc_train), len(err_perc_train)/len(row_train)))
plt.grid(axis='x', color='k')
plt.savefig('PercentageError_train.pdf')
#--  number of grid points with >5% error
limit=5  #% limit in error perc
out_err=0
tot=0
for ii in err_perc_test:
tot=tot+1
if ii>limit:
out_err=out_err+1
print(f'out_err={out_err}')
print(f'tot={tot}')
print(f'% point with more than 5% of error= {out_err*100/tot} ')
#------------------------------------- RMSE, NRMSE , R2 ------------------------------------
#----RMSE, NRMSE , R2 (over all sims)
mse_test= sum((X_test_pred_1D-X_test_1D)**2)/len(X_test_pred_1D)
mse_train= sum((X_train_pred_1D-X_train_1D)**2)/len(X_train_pred_1D)
RMSE_test=(mse_test)**(1/2)
RMSE_train=(mse_train)**(1/2)
print(f'RMSE_test {RMSE_test}')
print(f'RMSE_train {RMSE_train}')
NRMSE_test=RMSE_test/np.mean(X_test_1D)
NRMSE_train=RMSE_train/np.mean(X_train_1D)
print(f'NRMSE_test: {NRMSE_test}')
print(f'NRMSE_train: {NRMSE_train}')
R2_test= np.corrcoef(X_test_1D, X_test_pred_1D)[0,1]**2
R2_train= np.corrcoef(X_train_1D, X_train_pred_1D)[0,1]**2
print(f'R2_test {R2_test}')
print(f'R2_train {R2_train}')
#-- RMSE, NRMSE , R2 for each simulation
RMSE_test_sims=np.zeros((len(row_test)))
NRMSE_test_sims=np.zeros((len(row_test)))
R2_test_sims=np.zeros((len(row_test)))
for s in range(0,len(row_test),1):
X_test_sim=X_test[s]
X_test_pred_sim=X_test_pred[s]
X_test_sim_1D=X_test_sim.flatten()
X_test_pred_sim_1D= X_test_pred_sim.flatten()
mse=sum((X_test_pred_sim_1D-X_test_sim_1D)**2)/len(X_test_pred_sim_1D)
RMSE_test_sims[s]=(mse)**(1/2)
NRMSE_test_sims[s]=RMSE_test/np.mean(X_test_sim_1D)
R2_test_sims[s]= np.corrcoef(X_test_sim_1D, X_test_pred_sim_1D)[0,1]**2
print(f'RMSE_test_sims{RMSE_test_sims}')
print(f'NRMSE_test_sims{NRMSE_test_sims}')
print(f'R2_test_sims{R2_test_sims}')
RMSE_train_sims=np.zeros((len(row_train)))
NRMSE_train_sims=np.zeros((len(row_train)))
R2_train_sims=np.zeros((len(row_train)))
for s in range(0,len(row_train),1):
X_train_sim=X_train[s]
X_train_pred_sim=X_train_pred[s]
X_train_sim_1D=X_train_sim.flatten()
X_train_pred_sim_1D= X_train_pred_sim.flatten()
mse=sum((X_train_pred_sim_1D-X_train_sim_1D)**2)/len(X_train_pred_sim_1D)
RMSE_train_sims[s]=(mse)**(1/2)
NRMSE_train_sims[s]=RMSE_train/np.mean(X_train_sim_1D)
R2_train_sims[s]= np.corrcoef(X_train_sim_1D, X_train_pred_sim_1D)[0,1]**2
print(f'RMSE_train_sims{RMSE_train_sims}')
print(f'NRMSE_train_sims{NRMSE_train_sims}')
print(f'R2_train_sims{R2_train_sims}')
#---plot for test
```

```
plt.figure('RMSE - NRMSE - R2 for simulations test')
sim_test=np.arange(0,len(row_test),1)
plt.plot(sim_test, RMSE_test_sims, 'o-', color='b')
plt.plot(sim_test, NRMSE_test_sims, 'o-', color='g')
plt.plot(sim_test, R2_test_sims, 'o-', color='c')
plt.title('RMSE - NRMSE - R2 for simulations test')
plt.legend(['RMSE', 'NRMSE', 'R2'], loc='upper left')
#plt.ylabel('MeanAbsolutePercentageError')
plt.xlabel('simulations')
plt.savefig('RMSE_NRMSE_R2_test.pdf')
#---plot for train
plt.figure('RMSE - NRMSE - R2 for simulations train')
sim_train=np.arange(0,len(row_train),1)
plt.plot(sim_train, RMSE_train_sims, 'o-', color='b')
plt.plot(sim_train, NRMSE_train_sims, 'o-', color='g')
plt.plot(sim_train, R2_train_sims, 'o-', color='c')
plt.title('RMSE - NRMSE - R2 for simulations train')
#plt.ylabel('MeanAbsolutePercentageError')
plt.xlabel('simulations')
plt.legend(['RMSE', 'NRMSE', 'R2'], loc='upper left')
plt.savefig('RMSE_NRMSE_R2_train.pdf')
#plt.show()
#-----------------------------------------------------------------------------------------------------
#----------------------------POST PREDICTION for normalized value-------------------------------------
#-----------------------------------------------------------------------------------------------------
#-------------------------------------parity plot (norm)-------------------------------------
#----parity plot (norm)
#---test
plt.figure(f'Parity plot test (norm)')
plt.scatter(X_test_N_1D, X_test_N_1D, color='r',s=2)
plt.scatter(X_test_N_1D, X_test_pred_N_1D, color='b',s=2)
plt.title('parity plot test(norm) ')
plt.ylabel('X (predicted)')
plt.xlabel('X (experimental)')
#plt.legend(['experimental', 'prediction'], loc='upper left')
lineStart = min(X_test_N_1D)
lineEnd = max(X_test_N_1D)
plt.plot([lineStart, lineEnd], [lineStart, lineEnd], 'k-', color = 'r',lw=1) #line X_test/X_test
Max_N_xP=np.amax(ARRAY_Y_N)
plt.plot([0, Max_N_xP*1.05], [0, Max_N_xP], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, Max_N_xP*0.95], [0, Max_N_xP], 'k-', color = 'k',lw=1) #-5%
plt.savefig('parity_plot_test_norm.pdf')
#plt.show()
#----train
plt.figure(f'Parity plot (norm)')
plt.scatter(X_train_N_1D, X_train_N_1D, color='r',s=2)
plt.scatter(X_train_N_1D, X_train_pred_N_1D, color='k', marker='X',s=2)
plt.title('parity plot train (norm) ')
plt.ylabel('X (predicted)')
plt.xlabel('X (experimental)')
#plt.legend(['experimental', 'prediction'], loc='upper left')
lineStart = min(X_train_N_1D)
lineEnd = max(X_train_N_1D)
plt.plot([lineStart, lineEnd], [lineStart, lineEnd], 'k-', color = 'r',lw=1) #line X_test/X_test
Max_N_xP=np.amax(ARRAY_Y_N)
plt.plot([0, Max_N_xP*1.05], [0, Max_N_xP], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, Max_N_xP*0.95], [0, Max_N_xP], 'k-', color = 'k',lw=1) #-5%
plt.savefig('parity_plot_train_norm.pdf')
#plt.show()
#---------------------------------------------------------------------------------------------------------
#-------------------------------------------------ENCODER PREDICTION--------------------------------------
#---------------------------------------------------------------------------------------------------------
#---- PREDICTION ENCODER (for all: train and test)
print('encoder result')
X_train_code_N = encoder.predict(X_train_N)
X_test_code_N = encoder.predict(X_test_N)
print('encoder used\n')
print(f'shape X_train_code_N: {X_train_code_N.shape}')
print(f'shape X_test_code_N: {X_test_code_N.shape}')
#de-normalization (forse non serve)
X_train_code=X_train_code_N*STD +MEAN
X_test_code=X_test_code_N*STD +MEAN
#---------------------------------------------------------------------------------------------------------
#-------------------------------------------------SAVE TXT -----------------------------------------------
#---------------------------------------------------------------------------------------------------------
#---SAVE ARCHITECTURE parameters and results in txt
with open('summary.txt','a+') as txt:
txt.write(f'print of CNN5load.py\n\n')
txt.write(f'RESULT\n\n')
txt.write(f'RMSE_test: {RMSE_test}\n')
txt.write(f'NRMSE_test: {NRMSE_test}\n')
txt.write(f'R2_test: {R2_test}\n\n')
txt.write(f'RMSE_train: {RMSE_train}\n')
txt.write(f'NRMSE_train: {NRMSE_train}\n')
txt.write(f'R2_train: {R2_train}\n\n\n')
txt.write(f'RMSE_test_sims{RMSE_test_sims}\n')
txt.write(f'NRMSE_test_sims{NRMSE_test_sims}\n')
txt.write(f'R2_test_sims{R2_test_sims}\n\n')
txt.write(f'RMSE_train_sims{RMSE_train_sims}\n')
txt.write(f'NRMSE_train_sims{NRMSE_train_sims}\n')
txt.write(f'R2_train_sims{R2_train_sims}\n\n\n')
txt.close()
```

B.3.2 CNN5plot.py

```
# use after CNN5.py
# - plot of original field (CNN input)
# - autoencoder prediction and decoded plot for one sim
# - parity plot  and calcul of errors for one sim
# - encoder prediction (code) and print code
#--- pkgs
import os
import numpy as np
from numpy.random import seed
seed(9)
import os
os.environ ['KMP_DUPLICATE_LIB_OK'] = 'True'
import keras
from keras.metrics import MeanSquaredError
import matplotlib.pyplot as plt
#-----------------------------------------------------------------------------------------------------
#----------------------------------------------INPUT--------------------------------------------------
#-----------------------------------------------------------------------------------------------------
FIELD=1
sim=22
cell= 0.0067 # dim cell in meters
hx=0.1
hy=0.1
hz= 0.72
Norm='C'
name_autoencoder_folder= '/AUTOENCODER_1_1_17_08_2020_11_35_11'
#don't modify
#---
cell_folder=f'/Users/Fabi/Desktop/my_script_Fabi/gitLab/ulbatmtool/programs/CNN5_folder/Scell{cell}'
autoencoder_path_folder= cell_folder+f'/AUTOENCODER/AUTOENCODER_{FIELD}' + name_autoencoder_folder
autoencoder_name='autoencoder'
encoder_name='encoder'
ARRAY_Y_npy= cell_folder+'/ARRAY_Y'+f'/ARRAY_FIELD{FIELD}.npy'
#---
norm_folder= cell_folder+'/ARRAY_Y'+f'/Norm{Norm}'
Mean_npy= norm_folder+f'/mean/mean_{FIELD}.npy'
Std_npy= norm_folder+f'/std/std_{FIELD}.npy'
#-----------------------------------------------------------------------------------------------------
#----------------------------------------------START--------------------------------------------------
#-----------------------------------------------------------------------------------------------------
os.chdir(autoencoder_path_folder)
print(f'os get dir: {os.getcwd()}')
print(f'field= {FIELD} where 1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO')
#-----------------------------------------------------------------------------------------------------
#-----------------------------------------------ORIGINAL----------------------------------------------
#-----------------------------------------------------------------------------------------------------
#-----------------------------------------------------------------------------------------------------
#----------------------------------------------input simulation -------------------------------------
#-----------------------------------------------------------------------------------------------------
print('load array...')
#  1: T | 2: CH4 | 3: O2 | 4: CO2 | 5: H2O | 6: OH | 7: CO | 8: NO
```

```
if FIELD==1:
ARRAY_T = np.load(ARRAY_Y_npy)
print(f'shape ARRAY_T {ARRAY_T.shape}')
ARRAY_Y=ARRAY_T
if FIELD==2:
ARRAY_CH4=np.load(ARRAY_Y_npy)
print(f'ARRAY_CH4 shape {ARRAY_CH4.shape}')
ARRAY_Y=ARRAY_CH4
if FIELD==3:
ARRAY_O2=np.load(ARRAY_Y_npy)
print(f'shape ARRAY_O2 {ARRAY_O2.shape}')
ARRAY_Y=ARRAY_O2
if FIELD==4:
ARRAY_CO2=np.load(ARRAY_Y_npy)
print(f'ARRAY_CO2 shape {ARRAY_CO2.shape}')
ARRAY_Y=ARRAY_CO2
if FIELD==5:
ARRAY_H2O=np.load(ARRAY_Y_npy)
print(f'ARRAY_H2O shape {ARRAY_H2O.shape}')
ARRAY_Y=ARRAY_H2O
if FIELD==6:
ARRAY_OH=np.load(ARRAY_Y_npy)
print(f'ARRAY_CH4 shape {ARRAY_CH4.shape}')
ARRAY_Y=ARRAY_OH
if FIELD==7:
ARRAY_CO=np.load(ARRAY_Y_npy)
print(f'ARRAY_CO shape {ARRAY_CO.shape}')
ARRAY_Y=ARRAY_CO
if FIELD==8:
ARRAY_NO=np.load(ARRAY_Y_npy)
print(f'ARRAY_NO shape {ARRAY_NO.shape}')
ARRAY_Y=ARRAY_NO
print(f'shape ARRAY_Y {ARRAY_Y.shape}')
print(f'min value of ARRAY_Y (so of all simulations) is {np.amin(ARRAY_Y)}')
print(f'max value of ARRAY_Y (so of all simulations) is {np.amax(ARRAY_Y)}\n')
#-- extrapolation of my sim
ARRAY_SIM=ARRAY_Y[sim-1]
print(f'min value of ARRAY_SIM (so of my simulation) is {np.amin(ARRAY_SIM)}')
print(f'max value of ARRAY_SIM (so of my simulations)is {np.amax(ARRAY_SIM)}\n')
#-- normalization
MEAN=np.load(Mean_npy)
STD=np.load(Std_npy)
print(f'mean= {MEAN}')
print(f'std={STD}')
ARRAY_SIM_N=(ARRAY_SIM-MEAN)/STD
print(f'shape ARRAY_SIM_N {ARRAY_SIM_N.shape}')
print(f'min value of ARRAY_SIM_N (so of my simulation) normalized is {np.amin(ARRAY_SIM_N)}')
print(f'max value of ARRAY_SIM_N (so of my simulations) normalized is {np.amax(ARRAY_SIM_N)}\n')
#--reshape
```

```
ARRAY_SIM_N = np.reshape(ARRAY_SIM_N, (1, ARRAY_SIM_N.shape[0], ARRAY_SIM_N.shape[1], ARRAY_SIM_N.shape[2], 1))
#--flatten for postprocessing
ARRAY_SIM_1D=ARRAY_SIM.flatten()
ARRAY_SIM_N_1D=ARRAY_SIM_N.flatten()
print(f'shape ARRAY_SIM_N_1D: {ARRAY_SIM_N_1D.shape}\n')
#-----------------------------------------------------------------------------------------------
#-----------------------------------------------PLOT - ARRAY_SIM original---------------------------------
#-----------------------------------------------------------------------------------------------
#---- MESHGRID PP
x = np.arange(0, hx, cell)
y = np.arange(0, hy, cell)
z = np.arange(0, hz, cell)
xx, yy, zz = np.meshgrid(x,y,z)
fig = plt.figure('plot of mesh PP')
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xx, yy, zz,s=1.5)
ax.set_xlabel('X ')
ax.set_ylabel('Y ')
ax.set_zlabel('Z ')
#plt.show()
#---plot of original
print('plot of original ')
print(f'number of grid points (input value)= {ARRAY_SIM.shape[0]*ARRAY_SIM.shape[1]*ARRAY_SIM.shape[2]}')
from scipy.interpolate import griddata as gd
fig = plt.figure('plot of original ')
ax = fig.add_subplot(111, projection='3d')
im=ax.scatter(xx, yy, zz, c=ARRAY_SIM, cmap='jet', vmin=np.amin(ARRAY_SIM), vmax=np.amax(ARRAY_SIM),s=1.5)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
cax = fig.add_axes([0.05, 0.1, 0.02, 0.8])
fig.colorbar(im, orientation='vertical', cax=cax)
#plt.show()
#---plot of original (norm)
print('plot of original (norm)')
print(f'number of grid points (input value)= {ARRAY_SIM_N.shape[0]*ARRAY_SIM_N.shape[1]*ARRAY_SIM_N.shape[2]}')
from scipy.interpolate import griddata as gd
fig = plt.figure('plot of original (norm)')
ax =fig.add_subplot(111, projection='3d')
im=ax.scatter(xx, yy, zz, c=ARRAY_SIM_N, cmap='jet', vmin=np.amin(ARRAY_SIM_N), vmax=np.amax(ARRAY_SIM_N),s=1.5)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
cax = fig.add_axes([0.05, 0.1, 0.02, 0.8])
fig.colorbar(im, orientation='vertical', cax=cax)
#plt.show()
#-----------------------------------------------------------------------------------------------
#-----------------------------------------------AUTOENCODER PREDICTION----------------------------------
#-----------------------------------------------------------------------------------------------
```

```
#---LOAD AUTOENCODER AND ENCODER
from keras.models import load_model
print('autoencoder loading..')
autoencoder = load_model(autoencoder_name, custom_objects={'mean_squared_error': 'mean_squared_error'})
autoencoder.summary()
print('encoder loading..')
encoder = load_model(encoder_name, custom_objects={'mean_squared_error': 'mean_squared_error'})
encoder.summary()
#---------------------------------------------------------------------------
#---- PREDICTION AUTOENCODER
print('now i will predict')
ARRAY_SIM_pred_N = autoencoder.predict(ARRAY_SIM_N)
print('autoencoder used\n')
print(f'shape ARRAY_SIM_pred_N: {ARRAY_SIM_pred_N.shape}')
print(f'min value of ARRAY_SIM_pred_N (so of my pred sim) normalized is {np.amin(ARRAY_SIM_pred_N)}')
print(f'max value of ARRAY_SIM_pred_N (so of my pred sim) normalized is {np.amax(ARRAY_SIM_pred_N)}\n')
#--- de-normalization
ARRAY_SIM_pred= ARRAY_SIM_pred_N*STD +MEAN
print(f'shape ARRAY_SIM_pred: {ARRAY_SIM_pred.shape}')
print(f'min value of ARRAY_SIM_pred (so of my pred sim) is {np.amin(ARRAY_SIM_pred)}')
print(f'max value of ARRAY_SIM_pred (so of my pred sim) is {np.amax(ARRAY_SIM_pred)}\n')
#flatten for postprocessing
ARRAY_SIM_pred_1D=ARRAY_SIM_pred.flatten()
ARRAY_SIM_pred_N_1D=ARRAY_SIM_pred_N.flatten()
print(f'shape ARRAY_SIM_pred_N_1D: {ARRAY_SIM_pred_N_1D.shape}')
#---------------------------------------------------------------------------------------------------------
#-----------------------------------------------PLOT - prediction -----------------------------------------
#---------------------------------------------------------------------------------------------------------
#---normalized value plot
from scipy.interpolate import griddata as gd
fig = plt.figure('prediction with normalization')
ax = fig.add_subplot(111, projection='3d')
im=ax.scatter(xx, yy, zz, c=ARRAY_SIM_pred_N, cmap='jet', vmin=np.amin(ARRAY_SIM_pred_N),
 vmax=np.amax(ARRAY_SIM_pred_N),s=1.5)
ax.set_xlabel('X ')
ax.set_ylabel('Y ')
ax.set_zlabel('Z ')
cax = fig.add_axes([0.05, 0.1, 0.02, 0.8])
fig.colorbar(im, orientation='vertical', cax=cax)
#plt.show()
#---plot (without normalization)
fig = plt.figure('prediction')
ax = fig.add_subplot(111, projection='3d')
im=ax.scatter(xx, yy, zz, c=ARRAY_SIM_pred, cmap='jet', vmin=np.amin(ARRAY_SIM_pred),
 vmax=np.amax(ARRAY_SIM_pred),s=1.5)
ax.set_xlabel('X ')
ax.set_ylabel('Y ')
ax.set_zlabel('Z ')
cax = fig.add_axes([0.05, 0.1, 0.02, 0.8])
```

```
fig.colorbar(im, orientation='vertical', cax=cax)
#plt.show()
#----------------------------------------------------------------------------------------------------------
#-----------------------------------POST PREDICTION for 1 simulation -----------------------------------
#----------------------------------------------------------------------------------------------------------
#--------------------------parity plot: 1 sim-------------------------------------
#----parity plot with normalization
plt.figure(f'Parity plot with normalized value')
plt.scatter(ARRAY_SIM_N_1D, ARRAY_SIM_N_1D, color='r',s=2)
plt.scatter(ARRAY_SIM_N_1D, ARRAY_SIM_pred_N_1D, color='b',s=2)
plt.title('parity plot with normalized value')
plt.ylabel('X (predicted)')
plt.xlabel('X (experimental)')
lineStart = min(ARRAY_SIM_N_1D)
lineEnd = max(ARRAY_SIM_N_1D)
Max_N_xP= max(ARRAY_SIM_N_1D)
plt.plot([lineStart, lineEnd], [lineStart, lineEnd], 'k-', color = 'r',lw=1) #line X_test/X_test
plt.plot([0, Max_N_xP*1.05], [0, Max_N_xP], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, Max_N_xP*0.95], [0, Max_N_xP], 'k-', color = 'k',lw=1) #-5%
plt.savefig(f'parity_plot_sim{sim}_norm.pdf')
#plt.show()
# ----parity plot
plt.figure(f'Parity plot')
plt.scatter(ARRAY_SIM_1D, ARRAY_SIM_1D, color='r',s=2)
plt.scatter(ARRAY_SIM_1D, ARRAY_SIM_pred_1D, color='b',s=2)
plt.title('parity plot ')
plt.ylabel('X (predicted)')
plt.xlabel('X (experimental)')
lineStart = min(ARRAY_SIM_1D)
lineEnd = max(ARRAY_SIM_1D)
Max_xP=max(ARRAY_SIM_1D)
plt.plot([lineStart, lineEnd], [lineStart, lineEnd], 'k-', color = 'r',lw=1) #line X_test/X_test
plt.plot([0, Max_xP*1.05], [0, Max_xP], 'k-', color = 'k',lw=1) #+5%
plt.plot([0, Max_xP*0.95], [0, Max_xP], 'k-', color = 'k',lw=1) #-5%
plt.savefig(f'parity_plot_sim{sim}.pdf')
plt.show()
#--------------------------------------------RMSE, NRMSE , R2 ------------------------------------
#----RMSE, NRMSE , R2
mse_sim= sum((ARRAY_SIM_pred_1D-ARRAY_SIM_1D)**2)/len(ARRAY_SIM_pred_1D)
RMSE_sim=(mse_sim)**(1/2)
print(f'RMSE_sim {RMSE_sim}')
NRMSE_sim=RMSE_sim/np.mean(ARRAY_SIM_1D)
print(f'NRMSE_sim: {NRMSE_sim}')
R2_sim= np.corrcoef(ARRAY_SIM_1D, ARRAY_SIM_pred_1D)[0,1]**2
print(f'R2_sim {R2_sim}')
#------------------------------------------------------------------------------------------------------------
#----------------------------------------ENCODER PREDICTION--------------------------------------------
#------------------------------------------------------------------------------------------------------------
#---- PREDICTION ENCODER
```

```
print('now i will code')
ARRAY_SIM_code_pred_N = encoder.predict(ARRAY_SIM_N)
print('encoder used\n')
print(f'shape ARRAY_SIM_code_pred_N: {ARRAY_SIM_code_pred_N.shape}')
print(ARRAY_SIM_code_pred_N)
#--flatten for postprocessing
ARRAY_SIM_code_pred_N_1D=ARRAY_SIM_code_pred_N.flatten()
print(f'shape ARRAY_SIM_code_pred_N_1D: {ARRAY_SIM_code_pred_N_1D.shape}')
print(ARRAY_SIM_code_pred_N_1D)
```

Bibliography

[1] Aversano G., Bellemans aA., Li Z., Coussement A., Gicquel O., and Parente A. Application of reduced-order models based on PCA, Kriging for the development of digital twins of reacting flow applications. Computers and Chemical Engineering, 121:422?441, 2019.

[2] Aversano G., Ferrarotti M., Parente A. Digital twin of a MILD combustion furnace: reduced-order model development from CFD simulations

[3] Cavaliere A. and de Joannon M. Mild combustion. Prog. Energy Combust. Sci., 30:329 366, 2004.

[4] Chollet Francois, Deep learning with python

[5] Ferrarotti M., Furst M., Cresci E., De Paepe W., and Parente A. Key Modeling Aspects in the Simulation of a Quasi-industrial 20 kW Moderate or Intense Low-oxygen Dilution Combustion Chamber. Energy Fuels, 32(10):10228 10241, 2018.

[6] Ferrarotti M., Li Z., and Parente A. On the role of mixing models in the simulation of mild combustion using finite rate chemistry combustion models. Proceedings of the Combustion Institute, 37(4):4531 4538, 2019

[7] Ferrarotti M. Lupant D., Parente A. Analysis of a 20 kW flameless furnace fired with natural gas

[8] Kajishima, Takeo, Taira, Kunihiko. (2017). Reynolds-Averaged Navier Stokes Equations. 10.1007/978-3-319-45304-07

[9] Li, Z.; Ferrarotti, M.; Cuoci, A.; Parente, A. Finite-rate chemistry modelling of non conventional combustion regimes using a partially stirred reactor closure: Combustion model formulation and implementation details. Appl. Energy 2018, 225, 637 655.

[10] Nielsen Michael, Neural Networks and Deep Learning.

[11] Parente A, Malik MR, Contino F., Cuoci A, Dally BB. Extension of the eddy dissipation concept for turbulence/chemistry interactions to MILD combustion. Fuel 2015;163:98 111

Sitography

1. https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/ (august 2020)
2. https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/ (august 2020)
3. https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c (august 2020)
4. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798 (august 2020)
5. https://towardsdatascience.com/autoencoders-vs-pca-when-to-use-which-73de063f5d7 (august 2020)
6. https://towardsdatascience.com/classification-using-neural-networks-b8e98f3a904f (august 2020)

7. https://towardsdatascience.com/coding-deep-learning-for-beginners-linear-regression-part-2-cost-function-49545303d29f (august 2020)
8. https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23 (august 2020)
9. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (august 2020)
10. https://missinglink.ai/guides/neural-network-concepts/hyperparameters-optimization-methods-and-real-world-model-management/ (august 2020)
11. https://towardsdatascience.com/introduction-to-artificial-neural-networks-ann-1aea15775ef9 (august 2020)

Acknowledgements

Caro lettore,

è arrivato il momento di scriverti una seconda volta. Ora però ti scrivo con la consapevolezza che si chiude un percorso durato cinque anni che mi ha regalato esperienze uniche.

Come già ti ho raccontato nella nostra prima chiacchierata, i miei genitori sono stati molto preziosi per me in questo viaggio, ti dirò di più, mi sono stati vicino e hanno creduto in me quando tracciavo la rotta.. tanto pianificata quanto improvvisata. Grazie perché mi avete dato la grinta nei momenti più stanchi e avete ristabilito l'equilibrio in quelli più burrascosi. Grazie perché con me siete arrivati in questo porto, consapevoli che presto ripartiremo.

Sai com'è.. i viaggi più belli non si fanno mai da soli... grazie ragazzi di aver condiviso con me questi anni, ricorderò per sempre le nostre risate, le foto in laboratorio, i dibattiti sui cappelletti o tortellini, i pranzi appollaiati sulle scale o quelli "illegali" nella mensa, le partite a briscola e le curiosità su San Severo, le bonarie litigate per capire chi fosse l'amico del giaguaro, le news in tempo reale sul fuggitivo a Cento, le corse al posto per evitare la chiamata alla lavagna, le indicazioni per la TA- AMOIL e le (chiamiamole) riflessioni sulla "probabilità di morteee!!!" Ah ah ah . Ad ognuno di loro tengo in modo unico ma in questa sede ti racconterò qualcosa di più su una ragazza.. l'altro membro, oltre a me, della coppia scoppiata! E' stato fin da subito un bel rapporto che si è distinto con tutta naturalezza da ciò che ci veniva presentato mentre ancora sondavamo il terreno universitario. Un rapporto che è cresciuto grazie alle esperienze affrontate assieme, districandoci tra integrali e calcolatrici ;) una ragazza dalle sembianze docili che nasconde una personalità determinata e grintosa, una ingegnerAAAA di tutto rispetto della quale avrò sempre un dolce ricordo...

Come non parlare poi dei ragazzi e le ragazze extra-uni! Ognuno a modo suo ha colorato le mie giornate, portando aria nuova, creando uno scambio di pensieri a dir poco frizzante. Un grazie lo dedico alle due ragazze con cui spesso ho condiviso una bella coppa di gelato da Babbi guarnita di molte chiacchiere e risate, capaci di fare luce su altri due mondi tanto diversi quanto interessanti (vi ho stupito con l'inverno eh) ah ah ah. Grazie perché avete sempre dimostrato di esserci. ....Un altro di cui ti vorrei parlare è il ragazzo con la bici senza freni, si tratta di una persona tutt'altro che scontata, capace di ascoltarti e stupirti ma anche di farti ridere con i suoi, ormai celebri, interventi: "Lo denuncio!" e di farti preoccupare decidendo di andare da solo in mezzo ai monti per provare l'ebbrezza di dormire con gli orsi! (sa vut fe! L'è iccè) grazie perché hai condiviso con me pensieri e parole... Poi c'è il suo socio in affari.. l'eterno innamorato che passa dal citare versi da Nero Bali alle massime del suo zio giapponese, dall'essere il tipo perfetto per fare "balotta" al pisolino pomeridiano con la sua copertina, dall'essere Cracco 2.0 al "trituro i findus perché non li so cucinare" ah ah ah . Ti ringrazio per essere stato la persona con cui piacevolmente poter condividere il racconto della giornata ad ogni cena passata assieme...

In ultimo devi sapere che c'è stata un'altra persona che ha lasciato un segno... Sono sempre bastate poche parole perché lei capisse come mi sentivo. E la ringrazio per averne invece usate tante.. per essersi messo nei miei panni e per avermi preso per mano a ritrovare le fila dei miei pensieri. Grazie.