UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Master Degree in Physics of Data

Final Dissertation

AI-accelerated solution of Maxwell's equations for the simulation of high-temperature superconductors dedicated to nuclear fusion

Thesis supervisors

Prof. Dr. Mervi Mantsinen

Prof. Lidia Piron

Dr. Eduardo Cabrera Flores

Candidate

Diego Bonato

Academic Year 2024/2025

Contents

Abstract

High-temperature superconductors (HTS) are a key enabler for the next generation of nuclear fusion reactors, thanks to their ability to sustain high magnetic fields while reducing power consumption. However, accurately modeling their electromagnetic behavior requires solving Maxwell's equations through computationally expensive numerical simulations. In this work, we propose an AI-accelerated approach to solving Maxwell's equations for HTS materials, leveraging state-of-the-art deep learning techniques. Specifically, we train an Adaptive Fourier Neural Operator (AFNO) and a Fourier Neural Operator (FNO) on a dataset generated using MAGNET, a finite-element solver developed at the Barcelona Supercomputing Center. We obtain an AI model that can predict the magnetic field evolution on a coarse time grid, in a completely unsupervised way. This approach opens new possibilities for integrating AI-based surrogate models into large-scale fusion simulations, potentially enabling real-time digital twins for superconducting magnet design.

Introduction

Recent years have seen a renewed interest in nuclear fusion as a possible source of clean energy for the upcoming future. Many countries are now directing a lot of funding towards research institutes and big public enterprises like ITER, not to mention the growing landscape of private start-ups that are entering the market of nuclear fusion for energy. The current run for achieving this scientific goal is mainly driven by new technological advancements involving magnetic confinement devices, like tokamaks, which are considered to be the most promising configurations for a future fusion reactor. In particular, the breakthrough brought by type-II high-temperature superconductors (HTS) is veering many efforts towards the study of their implementations for large toroidal coils in these devices.

To characterise the behaviour of these materials, large numerical simulations are required, which are usually very expensive and resource-intensive. This is a situation that is now widespread in all areas of science, where costly simulations are a fundamental tool to explore new phenomena and benchmark models. An increasingly important aid in this respect is being provided by the field of Scientific Machine Learning (SciML). Thanks to the recent advancements in deep learning (DL), literature is now flourishing with new artificial intelligence (AI) tools, as means to integrate traditional numerical solvers with data-driven surrogate models.

Problem statement

Given these premises, the scope of this thesis is to apply a state-of-the-art DL algorithm to speed-up numerical simulations of HTS materials. In particular, we focus on solving Maxwell's equations for a section of a superconducting current-carrying wire, in the infinitely long approximation. We create a dataset of high-fidelity numerical simulations, comprising a variety of materials, initial and boundary conditions. We develop an AI surrogate model that autonomously evolves these systems in a fully unsupervised manner, requiring only the initial solution as a starting point. As will be described more in details in the later sections, we train an Adaptive Fourier Neural Operator (AFNO) on a dataset created with MAGNET, a Maxwell's equations numerical solver developed by the Fusion group at the Barcelona Supercomputing Center. As far as the authors are aware of, this is the first time models based on neural operators are used to speed-up simulations of HTS.

This thesis is laid out as follows. Chapter 1 gives some theoretical basis about HTS, that will be needed to understand the data produced by MAGNET. Chapter 2 presents a small introduction to DL and briefly reviews the literature concerning SciML, focusing on Neural Operators. Chapter 3 follows, where we detail the methodology and techniques used to develop the AI algorithm. Finally, results are presented in chapter 4. Conclusions and future works are displayed in chapter 5.

Chapter 1

High-temperature superconductors for fusion and MAGNET

High-temperature superconductors (HTS) are playing a fundamental role in the advancement of research on fusion reactors. For this reason, the Fusion group at the Barcelona Supercomputing Center has developed MAGNET [1], a Maxwell's equations numerical solver based on finite-element methods, that simulates the behaviour of different HTS materials. The code is integrated into ALYA, a framework focused on achieving high scalability and performance for multi-physics simulations on large-scale supercomputers.
The goal of the present chapter is to give the reader the key tools to understand what MAGNET does and the data that it produces, which will be eventually used to train the AI algorithm. To this end, we first present an overview of the physics behind HTS, then move our focus on the computation performed by MAGNET.

1.1 An overview of Electromagnetism

The macroscopic electromagnetic behaviour of media is described by Maxwell's equations:

$$\begin{aligned} \nabla \cdot \mathbf{D} &= \rho_f, \\ \nabla \cdot \mathbf{B} &= 0, \\ \nabla \times \mathbf{E} &= -\frac{\partial \mathbf{B}}{\partial t}, \\ \nabla \times \mathbf{H} &= \mathbf{J}_f + \frac{\partial \mathbf{D}}{\partial t}. \end{aligned} \tag{1.1}$$

where $\mathbf{H}$ is the external magnetic field and $\mathbf{B}$ is the magnetic flux density. The two are linked through the definition of the magnetization $\mathbf{M}$ via the following equation:

$$\mathbf{B} = \mu_0(\mathbf{H} + \mathbf{M}) \tag{1.2}$$

where μ_0 is the vacuum magnetic permeability, with $\mu_0 = 4\pi \times 10^{-7}\,\mathrm{H/m}$. Moreover, the electric displacement $\mathbf{D}$ is given by the following equation, where $\mathbf{E}$ is the electric field, $\mathbf{P}$ is the polarization vector, and $\epsilon_0 = \frac{1}{\mu_0 \cdot c}$ is the vacuum permittivity ($\epsilon_0 \approx 8.854 \times 10^{-12}\,\mathrm{F \cdot m^{-1}}$, and c is the speed of light in vacuum).

$$\mathbf{D} = \mathbf{E} \cdot \epsilon_0 + \mathbf{P} \tag{1.3}$$

The magnetization $\mathbf{M}$ and the polarization $\mathbf{P}$ can be seen as the response of the material to an external magnetic and electric field, respectively.

Finally, we need to consider the currents and charge densities. In a vacuum, all charges and currents are free, but in a medium, some charges are intrinsic to the material and are bound (ρ_b). Other charges are free and can move while maintaining electrostatic equilibrium; these are denoted as ρ_f in

equation (1.1). The total charge density is the sum of these two components.

We can use these definitions to give a relation between the currents, the magnetic field, and the magnetization:

$$\begin{aligned} \nabla \times \mathbf{H} &= \mathbf{J}_f \\ \nabla \times \mathbf{M} &= \mathbf{J}_b \\ \nabla \times \mathbf{B} &= \mu_0(\mathbf{J}_f + \mathbf{J}_b) \end{aligned}$$

Furthermore, we can introduce the notion of magnetic susceptibility χ_m, defined by

$$\mathbf{M} = \chi_m \mathbf{H}$$

This susceptibility reflects the material's responsiveness to an external magnetic field; the greater the susceptibility, the higher the magnetization induced in the material by the magnetic field.

At this point, we can use the latter definitions to obtain a direct relationship between $\mathbf{B}$ and $\mathbf{H}$,

$$\mathbf{B} = \mu_0(\mathbf{H} + \mathbf{M}) = \mu_0(1 + \chi_m)\mathbf{H} = \mu_0\mu_r\mathbf{H} = \mu\mathbf{H} \tag{1.4}$$

where μ_r is the relative permeability of the material, and $\mu = \mu_0\mu_r$ is the absolute permeability. This relationship shows how the magnetic field $\mathbf{B}$ depends on both the applied field $\mathbf{H}$ and the material's magnetization $\mathbf{M}$, with the material's properties encapsulated in the permeability μ.

A similar approach can be taken to describe the behavior of electric fields in media. In this case, the electric displacement field $\mathbf{D}$ is defined by equation (1.3). Analogously to the relationship between $\mathbf{B}$ and $\mathbf{H}$, the polarization

vector $\mathbf{P}$ can be expressed in terms of the electric susceptibility χ_e:

$$\mathbf{P} = \epsilon_0 \chi_e \mathbf{E}.$$

By substituting this relation into the equation for 1.3 we obtain:

$$\mathbf{D} = \epsilon_0(1 + \chi_e)\mathbf{E} = \epsilon_0 \epsilon_r \mathbf{E} = \epsilon \mathbf{E},$$

where $\epsilon_r = 1 + \chi_e$ is the relative permittivity of the material, and $\epsilon = \epsilon_0 \epsilon_r$ is the absolute permittivity. The latter equation shows how the material modifies the applied electric field, resulting in the electric displacement field $\mathbf{D}$.

Also, a relation between the polarization and the divergence of the field can be given.

$$\begin{aligned} \nabla \cdot \mathbf{D} &= \rho_f \\ \nabla \cdot \mathbf{P} &= -\rho_b \\ \nabla \cdot \mathbf{E} &= \frac{\rho_f + \rho_b}{\epsilon_0} \end{aligned}$$

Finally, given the current density $\mathbf{j}$ and the magnetic field $\mathbf{B}$, the Lorentz force per unit volume can be computed:

$$\mathbf{F} = \mathbf{j} \times \mathbf{B} \tag{1.5}$$

1.2 Superconductivity and MAGNET modelling

Superconductivity refers to a macroscopic electromagnetic phenomenon in which the resistance of materials drops to zero below a certain temperature and all the external magnetic fields are repelled at the same time [2]. Conduction is a fundamental property of materials, governed by their ability to transport electrical current. Conductivity, denoted σ, spans an extensive range, from excellent conductors like copper ($1.77 \times 10^{-8}\Omega \cdot \mathrm{m}$) to insulators such as glass ($\sim 10^{14}\Omega \cdot \mathrm{m}$). Mathematically, conductivity relates the current density $\mathbf{J}$ to the electric field $\mathbf{E}$:

$$\mathbf{J} = \sigma \mathbf{E}. \tag{1.6}$$

In superconductors, this relationship changes drastically, as they exhibit zero resistivity below a critical temperature T_c. This phenomenon enables persistent current without energy dissipation, a remarkable deviation from conventional conductive behaviour.

Superconductivity depends on specific conditions defined by three critical parameters:

- **Critical Temperature (T_c)**: The maximum temperature at which a material remains superconducting.
- **Critical Current Density (J_c)**: The highest current density a superconductor can carry without losing its superconducting state.
- **Critical Magnetic Field (B_c)**: The threshold magnetic field beyond which superconductivity is destroyed.

Above these critical limits, the material reverts to its normal resistive state.

Superconductors can be divided into two main groups according to their Critical Temperature. On one hand, we have Low-Temperature Superconductors

(LTS), which exhibit a superconducting behaviour only at temperatures below 77 K, the boiling point of liquid nitrogen. These are mainly metals and alloys and are cooled with liquid Helium. On the other hand, all material that have superconducting properties at temperature above 77K are called High-Temperature Superconductors (HTS). HTS have many properties that make them more suitable for fusion applications. First of all, they can be cooled down using liquid nitrogen, which is cheaper and easier to obtain than liquid helium [1]. Secondly, as they operate at higher temperatures with respect to LTS, they would require less power to be refrigerated in a cryogenic system used in a fusion plant like ITER. Moreover, HTS materials can achieve significantly higher magnetic fields compared to LTS, with REBCO being the most promising one for fusion applications [3]. For example, HTS coils have enabled magnetic fields up to 45.5 T in laboratory settings, which is critical for compact and high-performance fusion devices [4]. This allows also for more compact magnet designs, enabling smaller and more efficient fusion reactors.

The two most common constitutive models used to link the electric field **E** and current density **J** in superconductors are the critical state model (CSM) and the eddy current model (ECM). In the CSM, a non-zero **E** within the superconductor induces **J** in the same direction and with a magnitude equal to the local critical current density J_c. This model provides a foundation for predicting key characteristics of HTS materials and devices, such as the distributions of current density and magnetic field profiles. On the other hand, a smooth power law dependence is assumed in the ECM, namely

$$\mathbf{E} = \frac{E_c}{J_c} \left(\frac{|J|}{J_c} \right)^{n-1} \mathbf{J} \tag{1.7}$$

The critical electric field E_c, typically around $10^{-4}\,\mathrm{V/m}$, and the parameter n characterize the vortex flux creep, which reduces the penetration of field lines into the material. The n value is determined by the superconductor's nonlinear current-voltage relationship and generally ranges from 5 to

30, depending on the material properties [5]. The **J**-**E** relationship shows good adaptability to different materials, as $n = 1$ corresponds to Ohm's law, while $n \to \infty$ represents the critical state model (CSM). This widely-used framework in the community has been adopted in MAGNET. To compute the resistivity in equation (1.9), MAGNET uses the following expression:

$$\rho = \frac{E_c}{J_c} \left(\frac{|\nabla \times \mathbf{H}|}{J_c} \right)^{n-1} \tag{1.8}$$

Building on this theoretical foundation, MAGNET numerically solves Maxwell's equations in the H-formulation, which is a common choice when dealing with HTS simulations. It can be shown that equations (1.1) are equivalent to

$$\mu_0 \partial_t \mathbf{H} + \nabla \times \rho \nabla \times \mathbf{H} = 0 \tag{1.9}$$

where ρ is the resistivity of the media.

Solving this equation, MAGNET produces as an output the values of the x, y, and z-components of the magnetic field for each time step of the simulation.

1.3 ALYA and MareNostrum 5

MAGNET is a module developed as part of the broader ALYA framework, a software created by the Computer Applications in Science and Engineering department (CASE) at the Barcelona Supercomputing Center. ALYA is based on finite-element methods to solve multiphysics problems and designed to run with high efficiency in an HPC infrastructure [6].

In particular, the Fusion group aims at developing a digital twin of a fusion reactor, and has already developed two modules integrated in ALYA, NEUTRO and MAGNET, that simulate neutron transport and HTS materials, respectively. More modules are being developed to completely characterize the plasma state in a fusion reactor, starting from plasma equilibrium, to wave propagation in hot plasma, while also study breeding blankets and

Component	Details
Node Types	MareNostrum 5 GPP (General Purpose Partition)
Total Nodes	6,192
Processors	2x Intel Xeon Platinum 8480+ 56C 2GHz
Main Memory	16x DIMM 16GB 4800MHz DDR5
Local Storage	960GB NVMe
Network	100Gb/s bandwidth per node)
Total Compute Units	726,880

Table 1.1: MareNostrum 5 general purpose partition specifications

Component	Details
Node Type	MareNostrum 5 ACC (Accelerated Partition)
Total Nodes	1,120 nodes
Processors	2x Intel Xeon Platinum 8460Y+ 40C 2.3GHz (80 cores per node)
GPUs	4x NVIDIA Hopper H100 64GB HBM2
Main Memory	16x DIMM 32GB 4800MHz DDR5 (512GB per node)
Local Storage	480GB NVMe
Network	800Gb/s bandwidth per node
Total Compute Units	680,960 (CPUs + GPUs)

Table 1.2: MareNostrum 5 accelerated partition specifications

plasma disruptions using magneto hydrodynamics.

This work has been developed using MareNostrum 5, a pre-exascale EuroHPC supercomputer that combines Lenovo ThinkSystem SD650 V3 and Eviden BullSequana XH3000 architectures, providing two partitions with different technical characteristics [7]. The first is called General Purpose Partition (GPP), which is used to perform CPU-bounded tasks, like parallel numerical simulations. In total, the GPP counts of 726,880 processor cores and 1.75PB of main memory. The second partition is the Accelerate Partition (ACC), made of 1,120 nodes based on Intel Xeon Sapphire Rapids processors and NVIDIA Hopper GPUs, which amount to a total (CPUs + GPUs) of 680,960 compute units. The properties for each node are summarised in tables 1.1 and 1.2. MareNostrum 5 has a total storage capacity of 650 PB, with each group using the machine allocated a specific quota. The CASE department has been assigned 1171.87 TB, of which 1104.22 TB is currently in use at the time of writing.

Chapter 2

Scientific Machine Learning

The aim of this chapter is to delineate the general theoretical framework needed by a non-expert reader to understand the context in which the present work is set.

First, we present a basic introduction to deep learning, laying out the minimal lexicon and ideas necessary to understand how and why neural networks work. Second, we introduce the topic of Scientific Machine Learning (SciML), giving an overview of some of the approaches and solutions proposed in the literature, applying artificial intelligence to physics and engineering problems. Finally, we enter into the technical details of the models developed for this project, describing in details the theory behind Neural Operators and Vision Transformers. We focus on these models because they showed great results in solving PDEs in literature. More precisely, we use as a reference the work of Pathak et al. [8], where an Adaptive Fourier Neural Network is used to do weather forecasting in an autoregressive manner. We build on this, using data from MAGNET instead of meteorological data. In addition, we compare the Adaptive Fourier Neural Network with a simple Fourier Neural Operator. The former is an adaptation to images of the latter, so we think it is an interesting comparison.

2.1 Introduction to deep learning and neural networks

Artificial Intelligence (AI) is a broad field of computer science that branches into many different areas, which can vary significantly in terms of problem setting and methods.

AI is the study of agents that process information from their environment and take actions accordingly. Each agent operates based on a function that maps sequences of information to specific actions. These functions can be implemented through various approaches, including reactive agents, real-time planning, decision-theoretic models, and deep learning techniques [9]. Learning plays a central role in AI, both as a means of building capable systems and as a strategy for enabling them to adapt to unfamiliar environments beyond the designer's direct specifications. AI algorithms span a wide range of applications, including web search engines, recommendation systems, virtual assistants, autonomous driving, and natural language processing.

Machine learning (ML) is the branch of AI that uses statistical principles to develop algorithms that can learn to perform tasks without being explicitly programmed to do so. The most relevant advancements in this field is related to deep learning, a sub-branch of ML which leverages on artificial neural networks (NNs) to perform specific tasks directly from data. In this section, we describe what neural networks do, what are the tasks that they can perform and what algorithms are used to reach this goal.

The building block of a NN is the *neuron*. A neuron consists in a linear operation applied to a vector, which represents our data, followed by a non-linear function σ, called *activation function*. A NN is an arrangement of neurons that perform operations multiple times. We call an *architecture* a specific arrangement of neurons. The quintessential architecture is the Feed Forward Neural Network (FFNN), which can be represented as a direct acyclic graph, where neurons are arranged in *layers*, and all the neurons of a layer are connected to all the other neurons in the previous and following

layer. Overall, the operation performed by a layer can be written as:

$$f_{\text{layer}}(v) = \sigma(Wv + b) \tag{2.1}$$

where W is a matrix of parameters w_i, which are called *learnable parameters* or *weights*, v is an input vector representing data and b is a scalar value called *bias*. The strength of the connection between neurons is given by the learnable weight w_i. Nowadays, the most used activation function is the *Rectified Linear Unit* (ReLU), which takes the following form:

$$\text{ReLU(x)} = \max(0, x) \tag{2.2}$$

and is applied locally to each element of the vector, that we call x [9]. Another important activation function is the *Gaussian Error Linear Unit* (GELU) [10]:

$$\text{GELU}(x) = x\Phi(x) \tag{2.3}$$

where Φ is the cumulative Gaussian distribution, that is

$$\text{GELU}(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}(x/\sqrt{2})\right] \tag{2.4}$$

Given an architecture, a deep learning algorithm consists in finding the optimal parameters w_i, such that the NN can approximate, or *learn*, a certain function, operator or probability distribution. For example, we could train a NN on a dataset composed of meteorological measurements taken with a certain frequency over a certain period of time, including temperature T, pressure p, and humidity h, each in their appropriate units. Also, each of these comes with a label corresponding to a "sunny", "cloudy" or "rainy" day. In this case, we want the NN to approximate the function $f : \mathcal{A} \to \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$,

where

$$\mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \mathbf{e}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

represent the label "sunny", "cloudy" and "rainy", respectively. $\mathcal{A} \subseteq \mathbb{R}^3$ is the space of vectors representing all possible combinations of the three measurements (T, p, h).

The trained network would be a parametric approximation of f, namely f_w, made of subsequent compositions of functions f_{layer}, such that every input vector v representing a weather measurement gives as an output the corresponding label $\mathbf{e}_1$ if it is a sunny day, $\mathbf{e}_2$ if it is a cloudy one, $\mathbf{e}_3$ if it is rainy.

In order for the training to take place, a task performed by a NN needs to be translated into an optimization problem, thus defining an objective function that the training algorithm will minimize. This objective function is called the *loss function*. For example, to perform the classification task just described, one could define the loss function as:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{3} y_{i,c} \log(p_{i,c}) \tag{2.5}$$

where N is the total number of training samples, $y_{i,c}$ is the one-hot encoded label for the i-th sample and class c, and $p_{i,c}$ is the predicted probability for the i-th sample and class c. This particular loss function $\mathcal{L}$ is called *categorical cross-entropy* and it is the standard for multi-class classification problems.

At this point, it is necessary to define an *optimizer*, which is a specific choice of the algorithm that updates the weights w_i, given the current loss status. The process of updating the weights of the model given the loss is called *backpropagation*. There are many optimizers that perform backpropagation, with two of the most popular being *Stochastic Gradient Descent with mo-*

mentum (SGD) and *Adaptive Moment Estimation* (ADAM). Both of them use the gradients of the loss function to find the parameters w_i that minimize it. The SGD algorithm works as follows.

1: **Input:** Learning rate η, momentum coefficient β, number of iterations T, training data $\{(x_i, y_i)\}_{i=1}^N$
2: **Initialize:** Parameters $w^{(0)}$, velocity $v^{(0)} = 0$
3: **for** $t = 1$ to T **do**
4: Select a random sample (x_t, y_t) from the dataset
5: Compute the gradient $\nabla_w \mathcal{L}(w, x_t, y_t)$ of the loss function with respect to the parameters
6: Update the velocity:

$$v^{(t)} = \beta v^{(t-1)} + (1 - \beta)\nabla_w \mathcal{L}(w^{(t-1)}, x_t, y_t)$$

7: Update the parameters:

$$w^{(t)} = w^{(t-1)} - \eta v^{(t)}$$

8: **end for**
9: **Output:** Optimized parameters $w^{(T)}$

On the other hand, the ADAM algorithm is the following.

1: **Input:** Learning rate η, β_1, β_2, ϵ, number of iterations T, training data $\{(x_i, y_i)\}_{i=1}^N$
2: **Initialize:** Parameters $w^{(0)}$, moment estimates $m^{(0)} = 0$, $v^{(0)} = 0$, time step $t = 0$
3: **for** $t = 1$ to T **do**
4: Select a random sample (x_t, y_t) from the dataset
5: Compute the gradient $\nabla_w \mathcal{L}(w, x_t, y_t)$ of the loss function with respect to the parameters
6: Update the biased first moment estimate:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1 - \beta_1)\nabla_w \mathcal{L}(w^{(t-1)}, x_t, y_t)$$

7: Update the biased second moment estimate:

$$v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2)(\nabla_w \mathcal{L}(w^{(t-1)}, x_t, y_t))^2$$

8: Compute the bias-corrected first moment estimate:

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1^t}$$

9: Compute the bias-corrected second moment estimate:

$$\hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^t}$$

10: Update the parameters:

$$w^{(t)} = w^{(t-1)} - \eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon}$$

11: **end for**

12: **Output:** Optimized parameters $w^{(T)}$

In practice, before starting the training phase, one divides the dataset into training, validation and test set. The training set is used to update the weights of the model, while the validation set is used to assess the performance of the NN on unseen data. The user will change the hyperparameters of the model (such as the learning rate in the previous algorithms) based on the performance of the model on the validation set. Finally, the test set is used only at the end of the training, to evaluate how the model behaves on previously unseen data. The training phase consists in applying iteratively the computations described above on subsets of the training set, called *batches*. An *epoch* consists in one complete iteration of the optimizer over the whole training set, and the training phase consists in multiple epochs.

All recent development in deep learning has been obtained using NNs, all of which rely on the steps just described, even though with much more sophisticated architectures with respect to FFNNs.

Let us make a numerical example related to the aforementioned classification problem. One instance of the dataset is treated as a vector $v \in \mathbb{R}^3$, where each dimension (also called *feature*, in this context) corresponds to a measured quantity. For example,

$$v = \begin{pmatrix} 15 \\ 10001.3 \\ 0.77 \end{pmatrix}$$

corresponds to a measurement of 15°C, 10001.3 mmbar and a humidity of 77%.

An overly simplified architecture to solve this problem can be constituted of a FFNN made of one hidden layer, with 3 neurons. The hidden layer is effectively a 3×3 matrix W of learnable weights w_{ij},

$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}$$

that can be initialized sampling from a normal distribution $\mathcal{N}(\mu = 0, \sigma^2 = 1)$, obtaining something like

$$W = \begin{pmatrix} 0.0447 & 1.9112 & -0.2310 \\ 0.3459 & 1.3180 & 0.3696 \\ 0.3841 & 0.2970 & 0.7473 \end{pmatrix}$$

A *forward pass* consists in the following operations. First of all, consider a proper normalization of the input vectors. For example, if our data is normally distributed, we can subtract the mean value of the dataset and divide by its standard deviation. In this case, the above mentioned vector v

can take the following coordinates,

$$v^* = \begin{pmatrix} 0.4 \\ 0.2 \\ 0.77 \end{pmatrix}$$

Now, the learnable weights are multiplied by the vector, giving

$$Wv = \begin{pmatrix} 0.0447 & 1.9112 & -0.2310 \\ 0.3459 & 1.3180 & 0.3696 \\ 0.3841 & 0.2970 & 0.7473 \end{pmatrix} \begin{pmatrix} 0.4 \\ 0.2 \\ 0.77 \end{pmatrix} = \begin{pmatrix} 0.2222 \\ 0.6866 \\ 0.7885 \end{pmatrix}$$

where we assume a bias $b = 0$. Then, the activation function is applied; in the case of ReLU it gives an output vector $\hat{o}$:

$$\hat{o} = \begin{pmatrix} 0.2222 \\ 0.6866 \\ 0.7885 \end{pmatrix}$$

Moreover, since this is a multi-class classification problem, we design the architecture to have an output vector $o \in \mathbb{R}^3$ such that the first entry gives the probability of having a sunny day, the second a cloudy day and the third a rainy one. To this end, a softmax operation needs to be applied to the output $\hat{o}$,

$$\text{softmax}(\hat{o}) = \frac{\exp(x_i)}{\sum_j (x_j)} \tag{2.6}$$

where x_i is the i-th entry of $\hat{o}$, and softmax guarantees that $\sum_i x_i = 1$. In this case it gives:

$$o = \begin{pmatrix} 0.2297 \\ 0.3655 \\ 0.4047 \end{pmatrix}$$

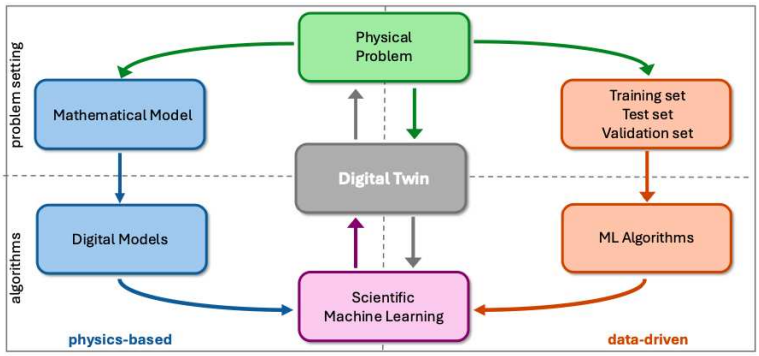

Figure 2.1: SciML is a fundamental tool for the development of digital twins. Image taken from Quarteroni et al. (2025) [11].

that we interpret as a 23% probability of a sunny day, a 37% chance of a cloudy day, and a 40% likelihood of rain. At this point, the loss function (2.5) is computed with respect to the true label corresponding to v. Then, the derivatives with respect to the three input features x_1, x_2, x_3 are computed, and the ADAM algorithm updates the matrix W with the new weights.

In the following section, we review how the deployment of more and more powerful models is affecting the world of natural science, focusing in particular to physics and engineering applications.

2.2 SciML

Scientific Machine Learning (SciML) is an emerging field of research at the intersection between AI and physics. The goal is to blend together existing scientific knowledge with the powerful tools of ML, to create a whole new family of algorithms that can be used to automate, accelerate and enhance traditional workflows. The range of applications is wide, from simulation of complex PDEs, to inverse problems and inference. Physics and mathematics provide us with a long history of powerful mathematical tools and extensive knowledge on a variety of domains of the physical world; the scope of SciML is to find clever ways to inform ML models with this solid prior knowledge of scientific principles.

There are various approaches to achieve this, such as modifying the architecture of a deep NN, designing its loss function, or incorporating symmetries into data processing [12].

This work provides a general overview of two of these techniques, which are central to the context of this thesis: Physics-Informed Neural Networks and Neural Operators.

One crucial application that this work focuses upon is solving forward problems. Simulation in physics and engineering presents numerous long-standing challenges. For most real-world applications, the primary difficulty lies in the exceptionally high computational cost of simulating complex, multi-scale, multi-physics systems. This is where the ALYA framework comes into play (section 1.3), since these kind of problems usually require expensive computations, needing lots of CPUs for many hours. This can become a huge impediment for the deployment of digital twins, that need near real-time feedback between the physical and the digital world to be actually employed in practical implementations. Various general and domain-specific techniques exist to reduce computational costs, including adaptive mesh refinement, subgrid parametrizations, and reduced-order modeling. However, these methods typically involve a trade-off between the accuracy of the physical system representation and the computational efficiency of the simulation [12].

SciML is addressing these challenges by enabling learning from past simulations, offering more effective computational shortcuts while minimizing the impact on simulation fidelity.

For example, SciML is being used to learn more efficient subgrid parametrizations [13], finer-resolution outputs from coarser-resolution simulations [14], and mesh-free methods which do not require elaborate discretization schemes [15].

2.3 Physics Informed Neural Networks

Physics Informed Neural Networks (PINNs) are deep NNs that incorporate physical laws directly in the loss function, translating differential equations and conservation laws into an equivalent optimization problem. Applications are found both in forward and inverse problems, but in this section we will focus on the former since it is of major interest for the scope of this thesis.

PINNs can be seen as a meshless method to solve PDEs by training a NN (forward problem), with the possibility to incorporate noisy data.

Let us start with a simple example [16]. Consider the viscous Burger's equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{2.7}$$

with a suitable initial condition and Dirichlet boundary conditions. $u(t, x)$ is the solution of this PDE. To solve this equation a PINN can be built as follows. In its original formulation [15], define a MLP that takes a two-dimensional input $\mathbf{v} = (t, x)$ and has one neuron as last layer, representing the output of the parametrized solution u_θ. Then, sample N_{physics} *collocation points* from the domain and its boundary $\{(t_i, x_i)\}_{i=1}^{N_{physics}}$ and use them to build a training set adding the available measured data. At this point, the NN can learn a parametrization u_θ of u by solving the associated optimization problem, minimizing the following loss function:

$$\mathcal{L} = w_{\text{data}}\mathcal{L}_{\text{data}} + w_{\text{physics}}\mathcal{L}_{\text{physics}} \tag{2.8}$$

where

$$\mathcal{L}_{\text{data}} = \frac{1}{N_{\text{data}}}\sum_{j=1}^{N_{\text{data}}}(u_\theta(t_j, x_j) - u(t_j, x_j))^2$$

$$\mathcal{L}_{\text{physics}} = \frac{1}{N_{\text{physics}}} \sum_{i=1}^{N_{\text{physics}}} \left(\frac{\partial u_\theta}{\partial t} + u_\theta \frac{\partial u_\theta}{\partial x} - \nu \frac{\partial^2 u_\theta}{\partial x^2} \right)^2 \Big|_{(t_i, x_i)}$$

where $u_\theta(t_i, x_i)$ is the output of the NN corresponding to the input (t_i, x_i) and the derivatives are computed using automatic differentiation. w_{data} and w_{physics} are weights that tune the relative importance of real data with respect to physical principles.

The goal of the training is to learn a parametrization of u such that it minimizes the MSE on both the data measurements and the PDE residuals.

Learning is considered supervised when real, potentially noisy data are provided; otherwise, it is classified as unsupervised. Note that, in the latter case, the training set consists in randomly sampled points from a numerical distribution defined on a specified domain; the training set is potentially infinite (up to numerical precision) and can be generated on the fly during training (up to memory constraints).

PINNs have been used for a variety of problems, from electrodynamics [17], to fluid mechanics [18] and there exist plenty of variants. For an extensive review refer to Quarteroni et al. (2025) [11].

As a quick mention, a first attempt has been made to integrate a PINN into MAGNET, which however lead to poor predictions of the magnetic field. Many different modifications on the standard PINN have been tested, however none of them converged to the true solution of the Maxwell's equations when evolved over time. We believe this is because Maxwell's equations in the H-formulation (1.9) are very stiff and could require a more careful treatment concerning the time domain. These results are non-conclusive and so we are not reporting any of them in this work; future works could attempt to use other formulations of the Maxwell's equations or use recurrent units to model the time evolution of the magnetic field.

2.4 Neural Operators

Neural Operators are a family of models that has gained popularity in the last years in the field of SciML to solve differential equations with NNs. The idea behind Neural Operators finds its base on a well-know theorem for NNs, the "universal function approximator" theorem [19]. It states that, if given enough parameters, NNs converge to whatever function mapping between Euclidean spaces. However, this framework is limiting when dealing with ODEs or PDEs, when maps between function spaces may be involved, for example between initial conditions to the corresponding solutions of a differential equation.

In general, a PDE can be written as

$$(L_a u)(x) = f(x), \quad x \in D, \tag{2.9}$$

with some boundary conditions defined on the border ∂D, where $u : D \to \mathbb{R}^n$ is the solution of the PDE, living in the Banach space $\mathcal{U}$, and $L_a : \mathcal{A} \to \mathcal{L}(\mathcal{U}; \mathcal{U}^*)$ is a mapping from the parameter Banach space $\mathcal{A}$ to the space of linear operators mapping $\mathcal{U}$ to its dual $\mathcal{U}^*$ for some $a \in \mathcal{A}$, $f \in \mathcal{U}^*$ and $D \subset \mathbb{R}^d$ a bounded domain. From this setting, we can define an operator $\mathcal{G}^\dagger := L_a^{-1} f : \mathcal{A} \to \mathcal{U}$, that by definition maps the parameter a to the solution u, $a \mapsto u$, fixing the boundary conditions.

For example, in the case of equation (1.9),

$$\mu_0 \partial_t \mathbf{H} + \nabla \times \rho \nabla \times \mathbf{H} = 0 \tag{2.10}$$

we have:

- The solution u is the magnetic field $\mathbf{H} : \mathbb{R}^3 \to \mathbb{R}^3$.
- The free parameter a is the resistivity ρ.
- The linear operator L_a is $L_\rho = (\mu_0 \partial_t + \nabla \times \rho \nabla \times)$

- $f(x)$ is zero.
- A specific case is the one of an infinitely long current-carrying wire, which is the one we focus our study on. In this case, the domain D is a circle of radius R equal to the radius of the wire, defined on $\mathbb{R}^2$, while ∂D is the boundary of the wire. The boundary conditions are given by the value of the field on the border of the wire for every t, and the value of $\mathbf{H}(t = 0)$ for every (x, y) inside the wire.

A simple approach could be to discretize the space of parameters $\mathcal{A}$ and have a NN train on a subset $\{u_a\}_{a\in\mathcal{A}}$ from that discretized space. In the case of equation (1.9), we would have a NN trained on simulations performed with various values of ρ. This NN would learn specific mappings between different values of ρ and the corresponding solution, however it would perform poorly on samples outside the training set.

In this regard, Neural Operators give a more solid way to tackle the problem. Instead of learning mappings between vector spaces, they approximate operators acting on Banach spaces, mapping from functions to functions. Learning directly the operator gives a more solid way to generalize outside the training set, as shown in the original paper [20].

Moreover, it can be shown that these models are discretization invariant, contrary to vanilla NNs. A model is defined as discretization invariant if, with a fixed number of parameters, it suffices these three properties:

1. it acts on any discretization of the input function, i.e. accepts any set of points in the input domain,
2. it can be evaluated at any point of the output domain,
3. it converges to a continuum operator as the discretization is refined.

PINNs, presented in section 2.3, take spatial coordinates as input and are inherently discretization-invariant, as they can be applied independently at each location. However, PINNs are designed to approximate the solution of a single instance of a PDE rather than learning a mapping from input

functions to output solution functions.

Convolutional Neural Networks (CNNs), on the other hand, do not naturally converge under grid refinement, as their receptive fields vary with different input discretizations. Nonetheless, when normalized by grid size, CNNs can be applied to uniform grids of varying resolutions. In this case, their behaviour approximates differential operators in a manner similar to the finite difference method.

Furthermore, as we show in the next section, Transformers [21] can be interpreted as a special class of neural operators with structured kernels, making them applicable to input functions defined on arbitrary grids. However, many vision-based Transformer architectures, such as the Vision Transformer (ViT) [22], rely on convolutional tokenization of image patches. As a result, they do not inherently exhibit discretization invariance.

A Neural Operator can be built as a composition of linear operators, followed by a non-linear activation function, just like a FFNN is made by a composition of linear matrix multiplications, also followed by a non-linear activation function.

Mathematically, a Neural Operator as defined by Kovachki et al. (2021) [20] is laid out as follows. Let us call the input functions $a \in \mathcal{A}$, defined on the bounded domain $D \subset \mathbb{R}^d$, and the output functions $u \in \mathcal{U}$, also defined on the bounded domain $D' \subset \mathbb{R}^{d'}$. A Neural Operator $G_\theta : \mathcal{A} \to \mathcal{U}$ has the following overall structure (see Zongyi et al. (2020) [23]):

1. Lifting: Using a pointwise function $R^{d_a} \to R^{d_{v_0}}$, map the input

$$\{a : D \to R^{d_a}\} \mapsto \{v_0 : D \to R^{d_{v_0}}\}$$

to its first hidden representation. The authors of the original paper choose $d_{v_0} > d_a$ and hence this is a lifting operation performed by a fully local operator.

2. Iterative Kernel Integration: For $t = 0, \ldots, T-1$, map each hidden

representation to the next

$$\{v_t : D_t \to R^{d_{v_t}}\} \mapsto \{v_{t+1} : D_{t+1} \to R^{d_{v_{t+1}}}\}$$

via the action of the sum of a local linear operator, a non-local integral kernel operator, and a bias function, composing the sum with a fixed, pointwise non-linearity. Here we set $D_0 = D$ and $D_T = D'$ and impose that $D_t \subset R^{d_t}$ is a bounded domain.

3. Projection: Using a pointwise function $R^{d_{v_T}} \to R^{d_u}$, map the last hidden representation

$$\{v_T : D' \to R^{d_{v_T}}\} \mapsto \{u : D' \to R^{d_u}\}$$

to the output function. Analogously to the first step, the authors usually pick $d_{v_T} > d_u$ and hence this is a projection step performed by a fully local operator.

The outlined structure mimics that of a finite-dimensional neural network where hidden representations are successively mapped to produce the final output. In particular, we have

$$G_\theta := Q \circ \sigma_T(W_{T-1} + K_{T-1} + b_{T-1}) \circ \cdots \circ \sigma_1(W_0 + K_0 + b_0) \circ P.$$

The mappings $P : R^{d_a} \to R^{d_{v_0}}$ and $Q : R^{d_{v_T}} \to R^{d_u}$ represent the local lifting and projection operators, respectively. The matrices $W_t \in R^{d_{v_{t+1}} \times d_{v_t}}$ are local linear operators, while the integral kernel operators K_t map functions $\{v_t : D_t \to R^{d_{v_t}}\}$ to $\{v_{t+1} : D_{t+1} \to R^{d_{v_{t+1}}}\}$. Bias functions are given by $b_t : D_{t+1} \to R^{d_{v_{t+1}}}$, and activation functions σ_t act locally as pointwise maps $R^{d_{v_{t+1}}} \to R^{d_{v_{t+1}}}$.

The dimensions $d_{v_0}, \ldots, d_{v_T}$, input dimensions $d_1, \ldots, d_{T-1}$, and intermediate domains $D_1, \ldots, D_{T-1}$ serve as hyperparameters of the architecture. Local maps act pointwise: for any $x \in D$, the lifting and projection operations satisfy

$$(P(a))(x) = P(a(x)), \quad (Q(v_T))(x) = Q(v_T(x)).$$

Similarly, the activation function operates as

$$(\sigma(v_{t+1}))(x) = \sigma(v_{t+1}(x)) \quad \text{for any } x \in D_{t+1}.$$

See figure 2.2 for a visual scheme of this.

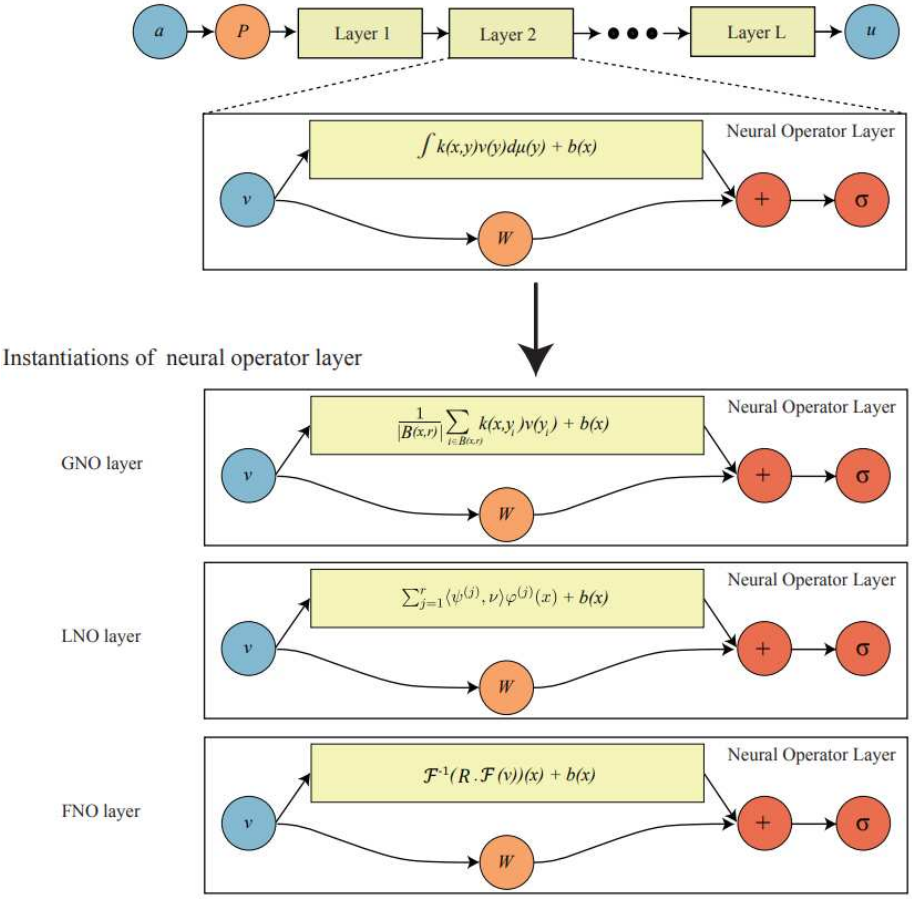

Figure 2.2: Scheme of a general neural operator layer and three possible variants. We are interested in the FNO layer, where the kernel integral is learnt in the Fourier space [20].

Using the notation just defined, and assuming that the input and output spaces $\mathcal{D}$ are the same, a forward pass of one layer can be written as

$$u(x) = \sigma\left(Wv(x) + \int_{\mathcal{D}} k(x,y)\, v(y)\, d\nu(y) + b(x)\right) \qquad \forall x \in \mathcal{D} \qquad (2.11)$$

2.4.1 Fourier Neural Operator

At this point, we have a general framework for building Neural Operators, and we can exploit our prior knowledge coming from physics to design a specific kernel $k(x, y)$ in equation (2.11). Classically, a typical way to solve complex PDEs is to apply a Fourier transform to the original equation, solve it in the Fourier space, and then project it back to the original space by applying the inverse Fourier transform. Thus, we know that moving the original image data representing an instance of the function space into a Fourier embedding could help a Neural Operator learning the solution.

A Fourier Neural Operator (FNO) can be built simply by taking the Fourier transform of the input of each layer, learn the parametrization of the kernel k using a linear layer, and then projecting back using the inverse Fourier transform. Computationally, this is done using the Fast Fourier Transform (FFT) and its inverse (iFFT), truncating the lower modes of the FFT. Moreover, as displayed in fig. 2.2, a residual connection is made between the input of each Fourier layer and the output of the iFFT, allowing for a better propagation of the gradient, as usual in deep networks, and also for learning non-periodic features in the data.

We conclude this section by noting that operator learning can be seen as an image-to-image problem, where instead of using CNNs, we use Fourier layers, which are more suited for solving PDEs. During the development of the present work, we quickly assessed the performance of a classical ResNet architecture, using CNN layers, with a FNO architecture, seeing that CNNs are much harder to train than FNOs in this context.

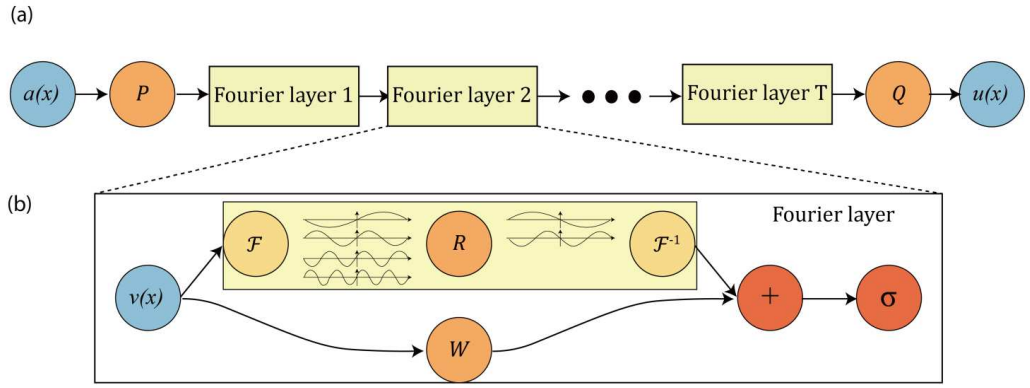

Figure 2.3: (a) The full architecture of neural operator: start from input a. 1. Lift to a higher dimension channel space by a neural network P. 2. Apply T layers of integral operators and activation functions. 3. Project back to the target dimension by a neural network Q. Output u. (b) Fourier layers: Start from input v. On top: apply the Fourier transform $\mathcal{F}$; a linear transform R on the lower Fourier modes and filters out the higher modes; then apply the inverse Fourier transform $\mathcal{F}^{-1}$. On the bottom: apply a local linear transform W. Taken from Zongyi et al. (2020) [23].

2.5 Vision Transformer and Adaptive Fourier Neural Operator

The final architecture we examine is the Adaptive Fourier Neural Operator (AFNO), designed as an optimized Vision Transformer (ViT) that integrates the principles of FNOs. In this section, we summarise the key points regarding the ViT architecture and how the AFNO makes it more efficient.

In recent years, natural language processing has seen significant progress, particularly with the development of Generative Pre-Trained models [24]. These models have achieved notable results, largely due to the Transformer architecture. This architecture supports strong scalability, enabling the training of very large models with billions of parameters. The work of Dosovitskiy et al. (2021) [22] showed that a simple Transformer-like implementation to images can achieve results that outperform classical ResNet architectures based on CNNs.

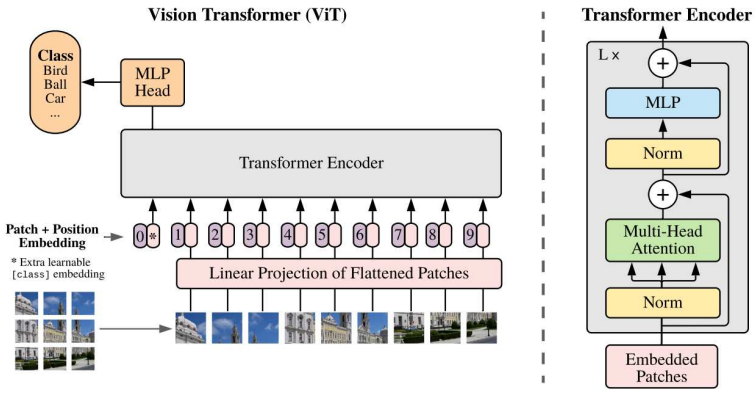

Figure 2.4: ViT overview. An image is split into fixed-size patches, each of which is linearly embedded, position embeddings are added, and the resulting sequence of vectors is fed into a standard Transformer encoder. To perform classification, the standard approach of adding an extra learnable "classification token" to the sequence is employed. This illustration is taken from Dosovitskiy et al. (2021) [22].

We start with a general overview of ViT, followed by a more detailed account of the AFNO in the next paragraph. For more details, please refer to the works of [22] [25].

The input of the ViT is an image of shape (C, H, W), which gets parsed by a CNN layer to create patches of shape (P, P). After the patches are created, they get flattened and linearly projected into an embedding space of dimension d_e. Then, a positional encoding is concatenated to these embeddings, in order not to loose the spatial information of the patches in the original input. The resulting vectors get fed to the Transformer Encoder, which computes a Multi-Head Attention operation on all embedded tokens. At its core, this operation linearly projects each token into three different vectors, called *keys*, *queries* and *values*. Then, a similarity score is computed between the *keys* and the *queries*, which is then used to learn a *context-aware* change of coordinates applied to the *values* vectors. The resulting vectors are then fed to a MLP that learns non-linear relations between them. In this way, the Transformer Encoder creates a meaningful encoded representation of the input data, which can be transferred to downstream tasks, for example classification.

More precisely, a 2D image can be represented as a token tensor $X \in \mathbb{R}^{h\times w\times d}$, made of hw tokens of dimension d. Bare in mind that lowercase h and w are not the height and width of the image in pixels, but rather the number of rows and columns of a matrix, which entries are vectors of dimension d (i.e. X is a tensor). That is, we have $N = hw$ tokens, each of shape d.

By doing so, an image can be treated as a sequence of tokens, from which a Transformer can learn a contextual embedding. ViT does so by computing a similarity score among all pairs of tokens, via *self-attention mixing*. The self-attention mixing operation is defined as $\text{Att} : \mathbb{R}^{N\times d} \to \mathbb{R}^{N\times d}$

$$\text{Att}(X) := \text{softmax}\left(\frac{XW_q(XW_k)^T}{\sqrt{d}}\right)XW_v \tag{2.12}$$

where $W_q, W_k, W_q \in \mathbb{R}^{d\times d}$ are the query, key and value matrices, respectively. X is the input tensor. Softmax is defined as:

$$\text{softmax}(x) = \frac{\exp(x_i)}{\sum_j(x_j)} \tag{2.13}$$

It can easily be shown that the self-attention (2.12) can be viewed as a discrete kernel summation. This allows us to link the theory of neural operators presented in section 2.4 to Transformers. First, we call K the softmax output, which is the $N \times N$ matrix of dot products between the linearly projected keys and the linearly projected queries. Calling $k[s,t] := K[s,t]\cdot W_v$, we have the kernel representation of self-attention:

$$\text{Att}(X)[s] := \sum_{t=1}^{N} X[t]k[s,t] \qquad \forall s \in [N] \tag{2.14}$$

where $s, t \in [hw]$ parametrize the token sequence and the matrix-valued kernel k. Going further, the kernel summation (2.14) can be extended to a continuous kernel formulation, where X is no longer a finite-dimensional vector in the Euclidean space $X \in \mathbb{R}^{h\times w\times d}$, but rather a spatial function

in the function space, $X \in (D, \mathbb{R}^d)$, defined on the domain $D \subset \mathbb{R}^2$ which is the physical (geometrical) space of images. At this point, the theory of operator learning described previously applies, bridging a clear connection between neural operators and attention. In fact, we can take the integral from equation (2.11) and change names to the variables to get

$$K(X)(s) = \int_D k(s,t)X(t)\mathrm{d}t \qquad \forall s \in D \tag{2.15}$$

Again, an appropriate choice for $k(s,t)$ gives us the integral in the FNO layer:

$$K(x)(s) = \mathcal{F}^{-1}(\mathcal{F}(k) \cdot \mathcal{F}(X))(s) \qquad \forall s \in D \tag{2.16}$$

where $\cdot$ denotes a matrix multiplication, $\mathcal{F}$ the continuous Fourier transform, and $\mathcal{F}^{-1}$ its inverse.

Let us see now how the AFNO architecture is built starting from this theoretical common ground. First of all, it should be noted that the attention operation (2.12) is quadratic in the number of tokens, since a dot product is computed between all pairs of tokens. To make this scaling better, AFNO performs the token mixing directly in Fourier space, as a continuous global convolution, using a Fourier layer from FNO. Computationally, this is done by applying the Fast Fourier Transform (FFT) to the input tokens

$$Z = \mathrm{FFT}(X) \tag{2.17}$$

with $Z \in \mathbb{C}^{h \times w \times d \times d}$ the complex-valued tensor of the FFT-transformed input image X. Then, the real and imaginary components of Z are separately multiplied by the kernel k which is parametrized by a matrix $W \in \mathbb{C}^{h \times w \times d \times d}$. An activation function follows, and the operation is repeated again. However, to avoid the aforementioned quadratic scaling of self-attention, a block-diagonal structure is imposed to W. A MLP block follows, that learns the interactions between different tokens, modifying them based on the input adaptively rather than using fixed transformations, as in standard FNO. Inspired

by self-attention, this MLP replaces the static weight transformation with a learnable mapping, allowing different frequency components to interact dynamically. Moreover, to promote sparsity, a soft-thresholding and shrinkage operation is applied:

$$S_\lambda(x) = \text{sign(x)}\ \max\{|x| - \lambda, 0\} \tag{2.18}$$

where $\lambda \in \mathbb{R}$ is a tunable parameter that controls sparsity. Then the inverse Fourier transform is computed. Figure 2.5 shows an illustration of the AFNO architecture.

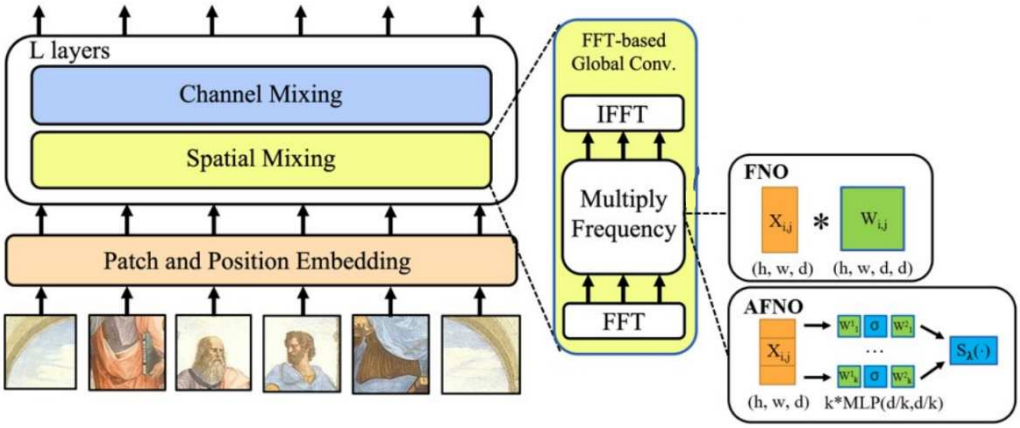

Figure 2.5: The multi-layer Transformer network with FNO and AFNO mixers. FNO performs full matrix multiplication that mixes all the channels. AFNO performs block-wise channel mixing using MLP along with soft-thresholding. The symbols h, w, d, and k refer to the height, width, channel size, and block count, respectively. Image taken from Guibas et al. (2022) [25].

Chapter 3

Methodology

In this chapter, we explain the methodology used in this project. We start by outlining the goal of this thesis, giving a general overview of how this work is laid out. We continue by describing the dataset generation process and its statistical properties, followed by a detailed discussion of the approach used to train and improve the NNs. Finally, we examine the processing pipeline and training strategy.

3.1 Problem statement

Our objective is to develop an AI surrogate model that can autonomously simulate the evolution of the magnetic field in a cross-section of current-carrying wire, in the infinitely long approximation. Ideally, the model should work for any superconducting material (material-agnostic) and across a wide range of boundary conditions, requiring only the initial conditions as input.

In order to train such a model, we need to create a dataset made of numerical simulations from which the model can learn the solutions of Maxwell's equations for a variety of different materials, boundary and initial conditions. Then, we need to define a learning framework, choosing the specific architectures of the NNs and determine a training strategy to accomplish the

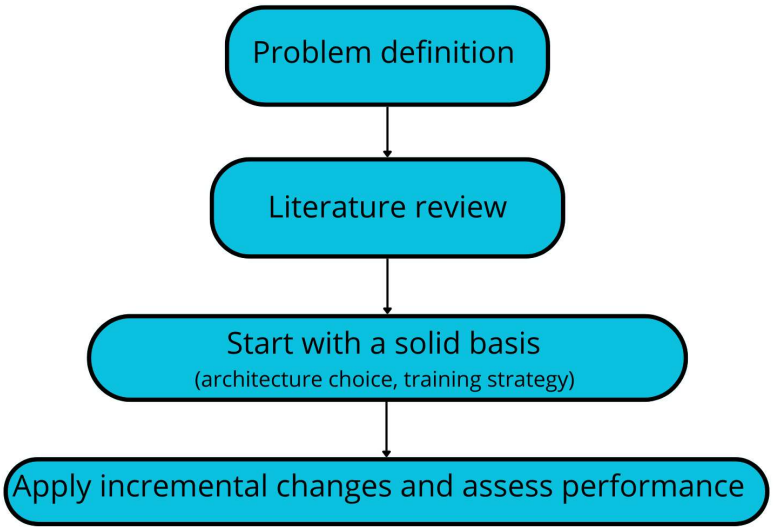

Figure 3.1: Scheme of the general methodology used for this work.

task. Last, the model needs to be evaluated both in terms of precision and performance, highlighting its strengths and weaknesses.

3.2 Dataset creation, management and analysis

In this section we describe the dataset used to train the AI models, starting from the data creation, to the data management and processing.

3.2.1 Dataset creation

We create a dataset using the MAGNET finite element code, described in chapter 1. We run simulations on a simple physical scenario, that is of a circular section of an infinitely long current-carrying superconducting wire. The wire has radius 0.001m and the boundary conditions are given by a sinusoidal current flowing on the border of the wire, having equation:

$$i(t) = i_0 \ \sin(2\pi f \cdot t + \phi) \tag{3.1}$$

where $i(t)$ is the instantaneous value of the current, i_0 is the current amplitude, f is the frequency, t is the time and ϕ is the phase. The value of f is fixed at 50Hz, $\phi = 0$ rad, and t ranges from 0s to 0.05s with an adaptive timestep. The maximum number of timesteps is fixed at $10,000$. Maxwell's equations (1.9) are solved on an unstructured mesh made of 4525 node elements. The outputs of one simulation consists in binary files, containing the solution of Maxwell's equations at each node of the mesh, which are then converted to text files. In order to avoid too large outputs, we save every 10th timestep. Each simulation comes also with a text file containing the (x, y, z) coordinates for each node. In this case z is always zero, since this is a two-dimensional problem, defined on the (x, y) plane. Beyond computing the solution of the magnetic field $\mathbf{H}$, MAGNET also computes other relevant physical quantities at every timestep; the total output is formed by 8 quantities:

MAGNET output	Description
H_x	The x component of the magnetic field
H_y	The y component of the magnetic field
H_z	The z component of the magnetic field
F_x	The x component of the Lorentz force
F_y	The y component of the Lorentz force
F_z	The z component of the Lorentz force
j_z	The z component of the induced current
Q	The dissipated power per unit volume

Table 3.1: List of physical quantities produced by MAGNET

The magnetic field is given in Tesla, the Lorentz force in Newton, the induced current in Ampere, and the dissipated energy in Watts per cube meters, which is computed as the integral of the induced current by the electric field, over the domain. In image 3.2 some snapshots of these quantities are shown.

Regarding storage, MAGNET uses double floating points numerical repre-

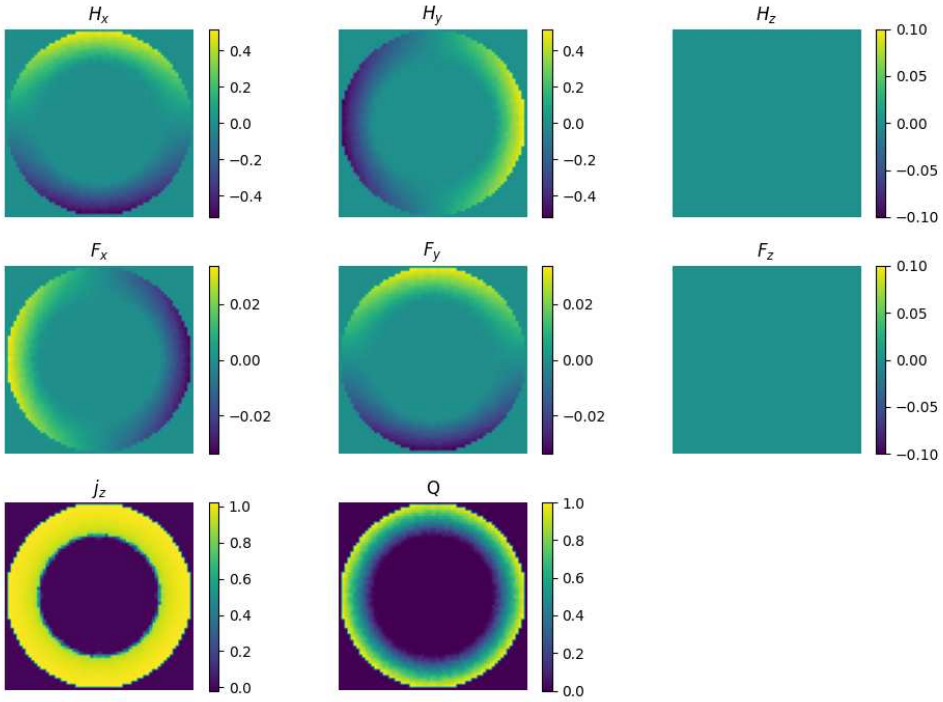

Figure 3.2: An example of the output quantities from a simulation performed with MAGNET. Units are normalized by the value on the border.

sentation, so each timestep requires a total of:

$$\text{number of nodes} \times \text{number of physical quantities} \times 8\text{B}$$

which translates into $4525 \times 8 \times 8\text{B} \approx 290\text{KB}$ per timestep. This means that if one simulation is made of 1000 timesteps, it will occupy almost 290 MB in storage, which in turns translates to almost 3TB of data on disk for $10,000$ simulations. We should keep this figure in mind, knowing that the storage available is limited in MareNostrum 5.

Creating a dataset from scratch requires careful design, since the choices done in this phase will inevitably have consequences on the outcome of the final model. Also, launching multiple simulations requires a lot of time, so it is fundamental to envisage possible problems beforehand. Moreover, to build a robust model that performs well in real-world scenarios, it is crucial to create a dataset that adequately represents the distribution it will encounter in production. However, the higher the variance of the dataset, the harder it is to be learned by a NN. It is also important to understand that creating a high-variance dataset means to save multiple simulations, which come with a big storage consumption, and that is the main constraint in MareNostrum 5. We have to strike a balance between these considerations and decide to simulate 9100 different simulations, each running with different physical parameters. The dataset occupies almost 3 TB in storage and is made of $27,662,245$ total files. The physical parameters that we choose to change are:

- The value of the exponent n in equation (1.7). 10 linearly spaced values are taken ranging from 5 to 30, comprehending almost all HTS materials [1].
- The value of the critical magnetic field E_c in equation (1.7). 10 logarithmically spaced values are taken, ranging from 10^{-5} to 10^{-1}.
- The value of the current amplitude i_0 in equation (3.1), ranging from 100 A to 1090 A, with a step size of 10 A.

The range of these values have been chosen by looking at the standard values used in literature (cfr. [26] [27] [28] [29]).

This results in a total of 10,000 different configurations. However, due to time constraints, we discontinued simulations that required more than 24 hours to converge, discarding 900 simulations in the process.

Each simulation runs with 12 cores and each run takes a variable time span to complete. The whole process of the data creation, starting from running the simulations, to converting the binary files into text files, followed by the compression in tar archives takes approximately 10 days. This figure is mainly driven by the time spent for a job to get out of the Slurm queue before being executed.

3.2.2 Data management

After creating the dataset, it is important to understand the best way to manage it, in order to optimize the processing pipeline during training and inference. Even though we are not dealing with a huge dataset, it is still big enough to cause some challenges. In this section, a first approach to data handling is described which resulted in issues that hindered concurrent training of the AI models. Afterwards, we describe how we tackled and solved these problems.

Let's start by considering the easiest, yet faulty, way of dealing with big data on a distributed computing environment. As soon as the MAGNET simulations are run and the text files are created, the dataset consists of a folder called *dataset*, containing 9100 folders inside, each of these comprising a variable number of files. As will be explained more in detail in section 3.4, data needs to be loaded from disk, pre-processed on CPU and then sent to GPU for training. In this approach, the process of loading and processing of data is managed by a Dataset class that we define in Python. During training, this class randomly reads a csv file containing the path to the necessary files; these are then opened in read-only mode from disk to CPU, processed one by one and then sent to GPU in batches. Also, the Dataset class is

wrapped by Pytorch Dataloader, which spawns a tunable number of workers to perform these computations. One training is run on one node, with 160 logical cores and 4 Nvidia H100 GPUs. The number of workers is set to 39, in order to have 1 logical core available for each GPU and to exploit the others to load data in parallel (160 total cores - 4 GPU = 156 free cores for computation; to distribute evenly the cores on the 4 GPU, 156/4 = 39 workers for each GPU). The strategy used for parallelizing the training is Distributed Data Parallel (DDP). DDP works by launching a separate process for each GPU, where each process independently holds a copy of the model and processes a different mini-batch of data. Each process has its own DataLoader, meaning data loading is done in parallel for each GPU. After computing local gradients, DDP synchronizes the gradients across GPUs and then updates the model weights.

Problems rise when launching more than 20 independent jobs concurrently, the General Parallel File System (GPFS) of MareNostrum 5 suffers from serious delays, affecting all users of the supercomputer. So, we had to investigate the reasons behind this to find a way to exploit the resource available at best.

First of all, we realize that this problem is linked to I/O operations from disk to CPU. It is not related to communication between the CPU and GPU, nor to communication between nodes, as it specifically affects operations involving GPFS on the login node, and each node runs an independent job. This happens because GPFS is a (distributed) networked file system, which can suffer from delays if the storage nodes become overloaded by a high volume of small, concurrent requests in the same directory. The ideal case would be to load all the data at once on memory, which is of course not possible (each node has a memory of 512 GB and our dataset is 3 TB in storage). To get over this situation, it is necessary to read large chunks of data instead. To this aim, the dataset is compressed into tar archives, each comprising 100 folders. Then, the sharded data is loaded sequentially from these archives using the webdataset library in Python. Sequential reading is faster than random indexing, and also loading shards instead of single little files exploits the bandwidth capabilities of MareNostrum 5, which are

800GB/s in the accelerated partition [7]. Moreover, a deeper look at the process spawned by Pytorch Dataloader showed that the optimal number of workers is 6, see figures 3.3 and 3.4. This changes allow concurrent trainings on multiple nodes, without overloading the GPFS. In section 3.4 we explain in more detail how data is processed before sending it to GPU.

3.2.3 Dataset statistics - exploratory analysis

Before applying whatever ML algorithm to a dataset, it is mandatory to perform an exploratory analysis first, in order to gather statistical information that will be used for pre-processing. In this section we report the insights found and how these relate to the type of normalization applied to data, before training the AI algorithm.

Our data is made of 3TB of txt files, which makes it impossible for us to load it all in memory and perform directly an analysis on it.

We begin our analysis qualitatively by examining the distribution of values at a single timestep, than a more quantitative analysis is performed. Immediately, we observe very skewed distributions in most timesteps, for every physical quantity, and in every simulation. The high variance observed is an inherent part of the simulation and therefore, rather than removing the extreme values, they must be handled carefully. However, as we will describe in chapter 4, if we subsample the dataset and take only one timestep every 50 for every simulation, the distribution of these values becomes Gaussian.

With these considerations in mind, it is clear that a transformation of this kind should consist in one single function that is invertible and uniquely applied to all the instances of the dataset. For example, it would be wrong to compute a normalization for each timestep or for each batch of timesteps, since it would remove the information about the time evolution of these values; it would also be wrong to apply a different transformation to each simulation for two reasons. The first one is that each simulation runs with different absolute values of the currents, that would be flattened from a normalization of this kind. The second reason is that computing a quantity

Number of workers	Iterations per second	D state vs R state	One minute average CPU load
0	~4.7 it/s	Many processes in R state, many in D state. This number does not change much over time. Many processes spawned even if num_workers=0	160
6	4.7 it/s	Many processes in R state, many in D state. This number does not change much over time.	165
39	~4 it/s	Many processes in R state, many in D state. This number does not change much over time.	170
100	~4.7 it/s	Many processes in R state, many in D state. This number does not change much over time.	160

Figure 3.3: Qualitative assessment of I/O load on the GPFS when performing random access operations on the dataset. All values are obtained heuristically using the Linux `top` command. They are not statistically rigorous but serve as an indicative measure of the I/O load on the file system. The first column refers to the number of threads used by each PyTorch dataloader. The second column gives a figure of the number of batches processed by the neural networks each second. The third column refers to processes actively working (R state) and idle processes waiting for I/O operations (D state). Many processes in R state correspond to a higher load on the file system. The last column shows, indicatively, the five-minute average load on the CPU. Randomly accessing the dataset leads to very high CPU load and numerous D-state processes that negatively impact the GPFS. Additionally, changing the number of workers in the dataloader does not affect the number of iterations per second.

Number of workers	Iterations per second	D state vs R state	One minute average CPU load
0	1.7 it/s	One process in R state. No processes in D state.	~ 20
5	< 8 it/s	All processes in R state. No process in D state.	~ 20
6	> 8 it/s	Most processes in R state. Once every 30 seconds some process in D state appears for a couple of seconds, then go back to R.	~ 25
from 7 to 10	from 7 it/s to 6 it/s	Gradually more processes in D state. When using 10 workers almost all processes in D state and stay in D state for long.	from ~30 to ~40
39	> 7it/s	All processes in D state. Two/three proccesses in R state.	~140
100	< 7it/s	All processes in D state. Two/three proccesses in R state.	407

Figure 3.4: Qualitative assessment of I/O load on the GPFS when accessing the dataset in shards, using webdataset. All values are obtained heuristically using the Linux `top` command. They are not statistically rigorous but serve as an indicative measure of the I/O load on the file system. The first column refers to the number of threads used by each PyTorch dataloader. The second column gives a figure of the number of batches processed by the neural networks each second. The third column refers to processes actively working (R state) and idle processes waiting for I/O operations (D state). Many processes in R state correspond to a higher load on the file system. The last column shows, indicatively, the five-minute average load on the CPU. This approach leads to a higher number of iterations per second and a lower load both on the file system and on the CPU. The optimal number of workers is around six.

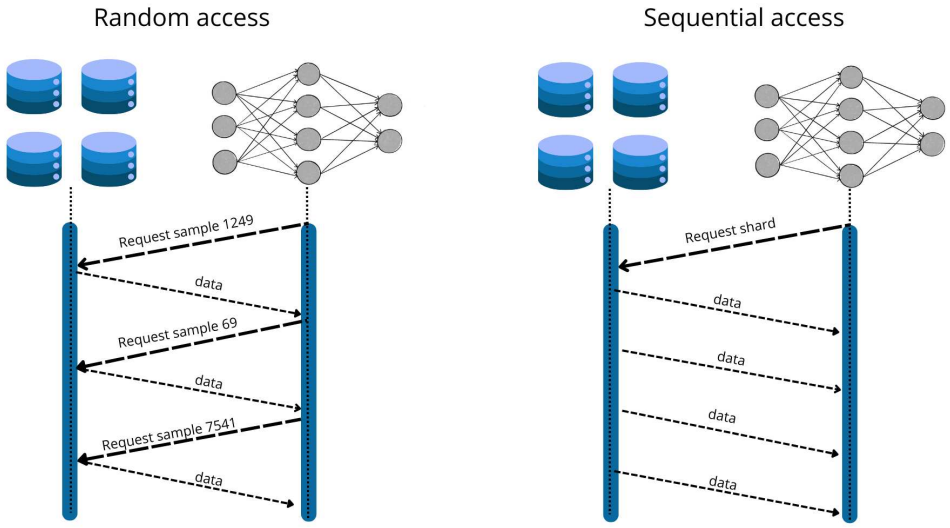

Figure 3.5: Random dataset access versus sequential access. Making many small requests from the dataloader to the storage causes the GPFS to overload. Image inspired by "Introduction to Large Scale Deep Learning" - Thomas M. Breuel [30].

simulation-wise, that is over all the timesteps of a given simulation, requires to know in advance how the simulation will evolve already at time zero, which is not possible during inference, where the model will be given only the first timestep.

To this end, we decide to use Robust Scaler, which consists in the following transformation [31]:

$$\text{RobustScaler}(x_i) = \frac{x_i - \mu_{1/2}}{\text{IQR}} \tag{3.2}$$

where x_i is a node value, $\mu_{1/2}$ is the median of the distribution of values of $\{x_i\}$, and IQR is the interquartile range which is the difference between the third quartile and the first quartile computed on the distribution of the values of $\{x_i\}$.

The values found are:

Quantity	Value
Median Values	
H_x, H_y, H_z, j_z, F_x, F_y, F_z	0
Q	7422.14
Interquartile Range (IQR) Values	
H_x	6.3468×10^3
H_y	6.3468×10^3
H_z	0
j_z	1.5579×10^8
F_x	6.6735×10^5
F_y	6.6735×10^5
F_z	0
Q	2.2810×10^5

Table 3.2: Median and IQR of the training set. The units are reported in the previous section.

3.3 General methodology for model selection

In this section, we discuss the modus operandi followed during the development of this project. Besides general deep learning good practices, we followed the tips contained in Google's Deep-Learning Tuning playbook [32]. The idea is to start by getting a solid baseline, and then fine-tune it by introducing incremental changes to either the model architecture or data processing. A modification is accepted only if it leads to better performance in the adopted metrics. As said, the process is incremental, in the sense that when a change is found to be beneficial, it is kept fixed and never tuned again, even when further changes are made on other hyperparameters. For example, if we find that a certain number of layers is optimal, we treat that number as a fixed parameter when assessing the performance of weight decay. This is done to avoid trying all possible combinations of the hyperparameters, which can be very expensive. An exception is made for the learning rate, which is tuned each time a new change is made to the network. Hyperparameters tuning has been performed using a grid search, launching multiple jobs with Slurm.

To create our AI surrogate model, we tested two different NNs. The first

is a FNO, which has shown good results in learning continuous mappings in infinite-dimensional spaces of functions, and we hope that it can perform better than standard CNN-based models. The second model is an AFNO, which uses an attention mechanism to learn long-range dependencies between patches. This architecture showed great potential in solving complex PDEs autoregressively [8], and we hope to replicate the results found in literature by adapting it to our use case. For the theoretical details behind these, please refer to Chapter 2.

In the Learning framework section 3.5.2 we explain the final architectures in detail, while in the section 3.5.1 we describe thoroughly how these models are trained to obtain a generative model for the simulation of HTS wires.

Since we deal with a generative task, a natural choice for the loss function is the mean-squared error (MSE), which is computed pixel-wise between the output of the network and the true solution of the PDE,

$$\mathcal{L} = \frac{1}{N} \sum_{i=0}^{N} (x_i - \hat{x_i})^2 \tag{3.3}$$

where N is the total number of pixels in an image, x_i is the true value of the pixel i, while $\hat{x_i}$ is the corresponding value from the generated image. In other words, we compute the error as the squared difference between the true and the generated value of the PDE solution for every node in the mesh and for a given timestep, then we take the average over all the nodes and use that value to compute the gradients for backpropagation.

The best model is chosen considering the total loss (3.3) as the main metric, followed by the convergence speed.

We do not treat the batch size as a tunable parameter [33], but rather we choose the maximum value that decreases the computing time and makes the training more stable, given the GPU resources constraints. We choose Adam as optimizer, fine tuning its hyperparameters based on its behaviour on training and validation loss curves. Moreover, we tried different processing of

data. Advancing with the training pipeline, we tested our nets with different number of layers, channels of the convolutional layers in the FNO, embedding dimension of the AFNO, and resolution of the input. We stress the fact that these changes have been studied incrementally.

3.4 Processing pipeline

In this section we report the steps followed to process a data instance before feeding it to the NN. First of all, data is stored in tar archives in a .txt file format, as explained in section 3.2.2. At this point, we would like to shuffle the data before splitting it into train, validation and test set. However, webdataset requires data that belong to the same training sample to stay in the same shard, since data is read sequentially. That means that shuffling all data is not possible, and instead we shuffle the indexes related to the 911 tar archives. Within each tar archive, data is not shuffled. Shuffling is done by fixing a random seed for reproducibility. As said, data is read sequentially in shards by the webdataset class, which also performs the following pre-processing steps on each txt file:

1. Converts the txt file to a Numpy array. The array has shape (1,4525) if it represents a scalar field, i.e. induced current and dissipated energy, while it has shape (3,4525) if it is a vector field, i.e. magnetic field and Lorentz force.

2. A Euclidean mesh of shape (R, R) is created and the field is interpolated on it using a nearest-neighbor function. The resulting array is of shape $(1, R, R)$ if it's a scalar field, or $(3, R, R)$ if it's a vector field. We do not use other kinds of interpolations, to avoid artifacts in the model that are not present in the original MAGNET simulation.

3. Data is then sent to the GPU in batches. The batch contains (batch size, 5) elements; in order these are: the key of the timesteps and physical quantities loaded, current array, magnetic field array, Lorentz force array, dissipated energy array.

4. Data is normalized with Robust Scaler, using the values reported at the end of section 3.2.3.

5. Each array contains both input and targets of the network, so we create the input and target arrays from the batch array. Before doing this, we have to be sure that the inputs match correctly the targets. Given the way the dataset is stored and read, this is done by checking that two contiguous arrays belong to the same simulation, using the key mentioned in point 3 to find the index where this change occurs. The batch is discarded if it contains only one instance.

6. Finally, the input array is passed to the network, that performs the forward pass, computes the loss and backpropagates the gradients. Loss is MSE (eq. 3.3).

At this point, some reasoning is needed regarding the interpolation phase. As reported in section 3.2, MAGNET simulations are performed on an unstructured grid, while all the nets that we tested require image-like inputs. These are obtained by interpolating the unstructured arrays into a Euclidean grid of shape (R, R), where R defines the resolution of the input data (point 2 of the list above). We immediately recognise a clear tradeoff between resource consumption on one hand, and data quality one the other. In fact, low-resolution input data would require a low memory footprint for two reasons. The first is evidently linked to data itself, since the less bytes are used to represent it, the less memory it uses on the GPU. The second is instead linked to the model complexity. In fact, it would require less parameters to capture the relevant features of low-resolution data, with respect to high-resolution, which fine-details can be captured by increasing the number of neurons. In other words, high-quality input data calls for more complex models, with more parameters and, thus, a bigger memory footprint. Adding more degrees of freedom allows to extrapolate more information from data while, on the contrary, the same number of parameters could be excessive with low-quality data. The tradeoff consists in finding a low enough value of R, such that we can still retrieve the relevant information from data without running out of memory,

given the computational resources at our disposal. To make this reasoning more quantitative, consider for example the memory usage of a low-resolution input instance, with $R = 64$. As already explained in section 3.2, each input data is made of 8 channels stacked together, which means that the actual input fed to the net is an array of shape $(\text{batch size}, 8, R, R)$, for example $(1, 8, 64, 64)$. Given these numbers, a single data instance represented by single precision floating points (float32), would occupy (32bits=4B):

$$(1 \times 8 \times 64 \times 64) \times 32\text{bits} \approx 131\text{KB}$$

that is, $8 \times 64 \times 64 = 32,768$ data points to be evaluated. For an input like this, a reasonable value for the total number of parameters of a NN is of order of some millions, which would corresponds to an order of some MB on the GPU.

On the other hand, a high resolution input with $R = 1024$, using the same float32 precision, would consist in an array of

$$(1 \times 8 \times 1024 \times 1024) \times 32\text{bits} \approx 34\text{MB}$$

with a quadratic dependency of the memory footprint on the resolution. An input like this consists in $\approx 8 \times 10^6$ data points, which would require billions of parameters to fit properly, occupying some GB in the GPU.

Having said this, we must recall that the original unstructured mesh on which the numerical simulation is performed consists in 4025 nodes, implying that using $R = 64$ would create more interpolated points than the ones where the PDE was numerically evaluated ($64 \times 64 = 4096$, which is already greater than the number of nodes used by MAGNET, 4025). However, we think that a slightly bigger number of interpolated point can benefit the learning process of the network, even though the interpolation will inevitably create artifacts not present in the original data. This is a limitation of our approach, that will be discussed in the last chapter.

3.5 Training pipeline

In this section we describe in detail the steps followed to obtain an AI surrogate model. The final goal is to obtain a deep learning model that produces the same outputs of the MAGNET code on a section of an infinitely long current-carrying HTS wire. The outputs produced are listed in section 3.2.1. Once the initial conditions are given to it, the model should be able to generate the next output autoregressively, ideally as long as we want. In section 3.5.1 the steps followed to obtain such a model are described, while in section 3.5.2 the architectures of the best model found after hyperparameters tuning are shown. This approach draws heavily from the work of Pathak et al. [8], which applied the same techniques in meteorology.

3.5.1 Training strategy

First of all, the dataset is split into train, validation and test set. This is done by first shuffling the indexes of the tar archives where the dataset is stored, and then considering the first 70% of the indexes to be in the training set, the second 15% to be in the validation set, and the third 15% to be in the test set. We split on the tar archives because of the nature of how data is loaded by the webdataset class. As said in section 3.2.2, data is read in shards and each of these contain the timesteps data in order. Mixing inside the tar archive would impede this kind of loading. As common practice in machine learning, the training set is used to update the weights of the model, the validation set is used to assess the impact of different hyperparameters. The test set is never seen until the very end of the project, to assess the performance of the model on previously unseen data.

Following the prescription of Pathak et al. (2022) [8], we divide our training phase in three steps:

1. Pre-training phase
2. Fine-tuning phase
3. Autoregressive evaluation

The pre-training phase consists in making the net learn to predict one timestep ahead. In this phase, the input of the net is one timestep x_t and the loss is computed as the MSE between the output of the net $\hat{x}_{t+1}$ and the true timestep x_{t+1}. However, differently from Pathak et al. (2022) [8], we use this phase mainly to assess how the number of parameters of each model impacts the performance on the loss. In order to obtain results quickly, we stop the training as soon as the validation curves show a clear trend, giving the best model. Most times one epoch is enough to assess this. The resulting models are fine-tuned in the second phase. Now the model has to learn how to predict two timesteps ahead of the input. This is done by performing the same computation as during the pre-training, taking one timestep x_t and generating $\hat{x}_{t+1}$, computing a loss $\mathcal{L}_1$. However, in this phase the generated $\hat{x}_{t+1}$ is fed again to the net, which predicts $\hat{x}_{t+2}$. Another loss $\mathcal{L}_2$ is computed between the second prediction and the true value x_{t+2}. The total loss is the sum of the two, $\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2$, which is used for backpropagation. In this way, the model is forced to learn how to deal with its own predictions, which should aid the autoregressive predictions. All $\{x_i\}$ are taken from the training set. Before the pre-training phase all the networks are initialized using He normalization, while for the fine-tuning phase the nets are initialized with the best weights found in the pre-training phase.

The last step is to use the trained models to generate sequences of meaningful timesteps, and estimate the overall performance. During this phase, gradients are detached from the computational graph and the weights of the model are not updated. We start by choosing one timestep from the validation set, feeding it to the network and predict the next timestep. Then, the output is fed again into the net to predict the following step.

3.5.2 Learning framework

All the runs are performed using Lightning Pytorch [34] on the supercomputer MareNostrum 5. Each run uses 4 GPU NVIDIA H100, and one training epoch lasts approximately 2 hours.

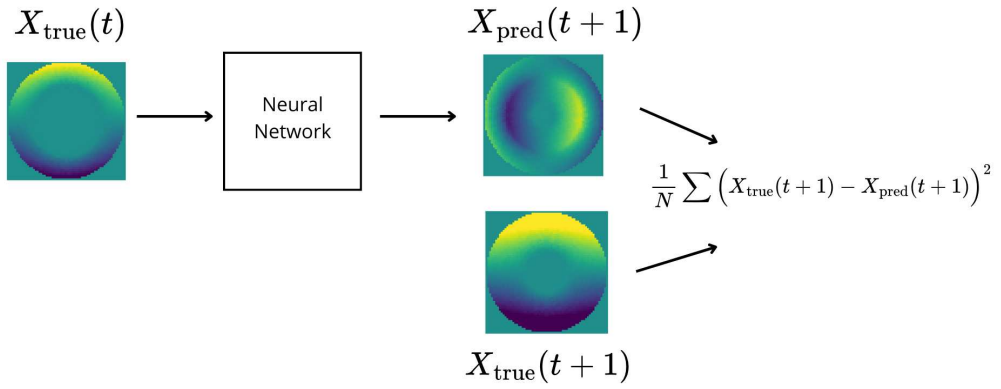

Figure 3.6: Illustration of the pre-training phase. The input of the NN is a numerical solution from MAGNET, for some time step t and a configuration of (E_c, n, i_0). The net predicts the following timestep, which is compared to the target numerical solution via MSE. Then the weights of the model are updated with backpropagation.

We train two different architectures, a Fourier Neural Operator (FNO) and an Adaptive Fourier Neural Operator (AFNO). We choose these because they both showed good results in solving PDEs [23, 25] and we hope to replicate them for our use-case. We also try a ResNet architecture, because its a natural benchmark that can be interesting to compare against the other two. However, we immediately see that ResNet is harder to train with respect to FNO and AFNO, which show promising results without much finetuning. So, we decide to discard ResNet from future experiments, given our time and resources budget.

As already explained in section 3.5.1, we start with a grid search during the pre-training phase. The grid search is performed by running N independent concurrent jobs on N nodes in the MareNostrum 5 cluster. All training use Adam as optimizer. The first parameter that we assess is the number of layers. Once found the optimal number of layers, we perform a second grid search, aiming to see if the nets could benefit from adding more parameters.

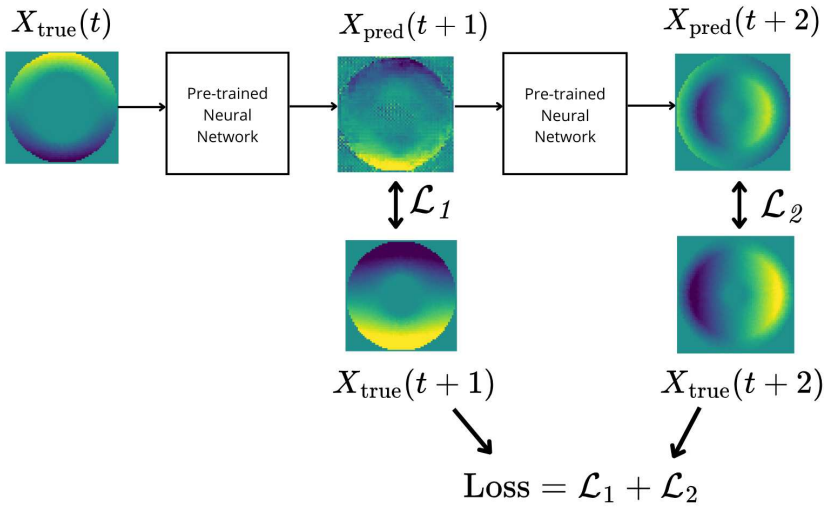

Figure 3.7: Illustration of the fine tuning phase. The NN is loaded with the weights found in the pre-training phase. Similarly to the pre-training phase, a true sample $X_{\text{true}}(t)$ is given as an input of the NN, which predicts the following timestep $X_{\text{pred}}(t+1)$. This is then fed again to the model, which predicts $X_{\text{pred}}(t+2)$. The final loss used for backpropagation is the sum of the two losses $\mathcal{L}_1$,$\mathcal{L}_2$ computed in the intermediate steps.

To do so, we vary the dimensionality of the embedding space of the AFNO and the number of latent channels in the spectral layers of FNO. Finally, we run a full training with the best parameters found. In this phase we also test some variations to the optimizer, changing β_1, applying weight decay, and assessing Cosine Annealing scheduler for the learning rate.

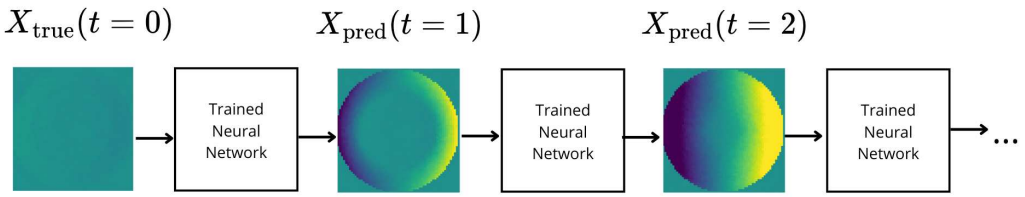

Figure 3.8: Illustration of the autoregressive inference. During this phase gradients are detached from the computational graph. The model is initialized with $X_{\text{true}}(t = 0)$, then the model predicts the following timestep, which is used as input for the successive prediction. In this way, the model is able to evolve the system in a completely unsupervised way.

Chapter 4

Results

In this chapter we outline the results obtained during the development of our AI-surrogate model aimed at accelerating HTS simulations. The models presented vary on their complexity and on the difficulty of the task that they are trained on, starting from a simple, light model that serve as starting point and proof of concept, going to highly parameterized networks trained on the whole dataset and many physical quantities. We end up with a model that can evolve the magnetic fields in a HTS wire autoregressively for around 4 to 5 steps in a coarse grid, corresponding to $200 - 250$ steps in the MAGNET, but only for a subset of the test data.

Proof of concept

First of all, we want to assess if our approach is feasible. To do so, we select a subset of the dataset and train an AFNO on that. The idea is that if the AFNO shows promising results on a low-variance dataset, it means that a more thorough study can be performed.

The subset is selected such that only the current i_0 changes, while the material-related parameters stay fixed. No hyper parameters fine tuning is performed, since we are interested in assessing the base performance of the model.

Fig. 4.1 shows the performance of the proof-of-concept experiment. We saw empirically that, for many simulations, a characteristic time scale of the evolution is $\tau = 50$ MAGNET timesteps. So, we believe that training will be much easier if the net is trained on a coarse grid, learning to predict 50 timesteps ahead, instead of just 1. We start following this intuition, even though, in the next section we try training on all the timesteps, to see if the time resolution can be improved. The reconstructions on the validation set are promising, however, when evaluated autoregressively, the model prediction diverge quite quickly from the ground truth, as shown in fig. 4.2. This suggests that model fine-tuning is necessary. We are satisfied with these preliminary results, since they show that the task is feasible and it is worth extending to the whole dataset.

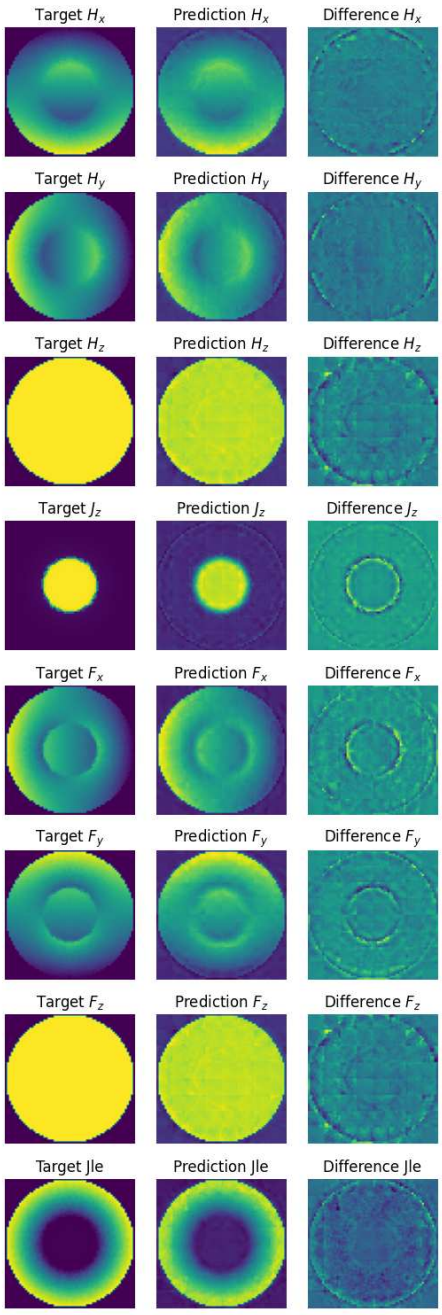

Figure 4.1: Pre-training results obtained with the AFNO model during the explorative proof-of-concept phase. The model is fed with a solution $X_{\text{true}}(t)$ and the predicted $X_{\text{pred}}(t+1)$ is shown in the central column. The left column shows the true solution as computed by MAGNET, while on the right the pointwise difference is shown. All images share the same scale of colors.

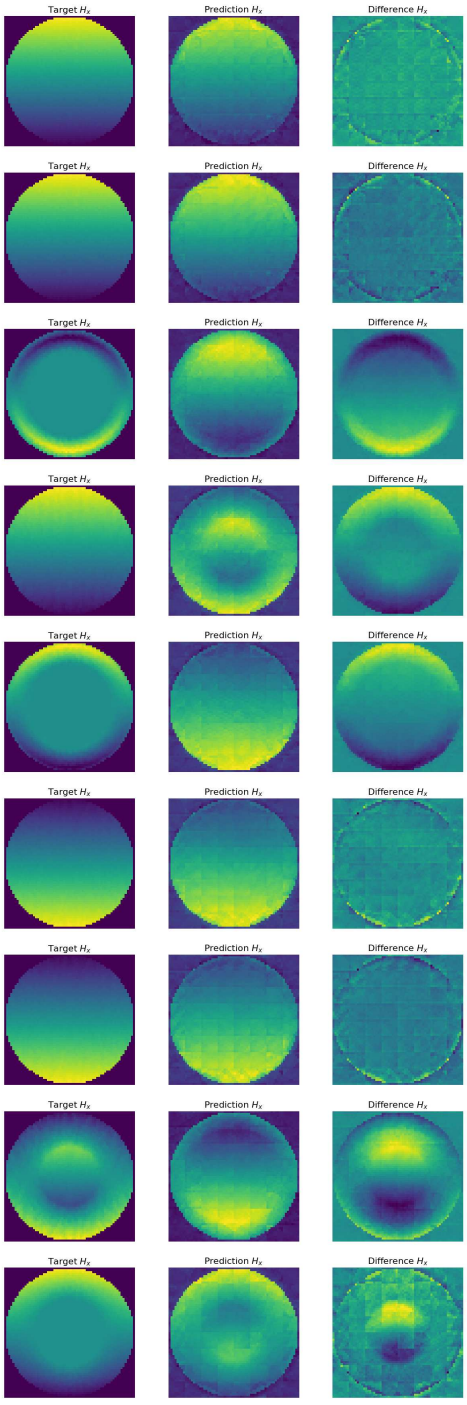

Figure 4.2: AFNO autoregressive evolution of H_x as found during the explorative proof-of-concept phase, using a coarse grid. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last one shows the pointwise difference between the two. We see that the AI predictions quickly diverge from the ground truth.

Training on a fine-grained timescale leads to learning the identity function

Given the promising results obtained with the toy model and data, we train two NNs to evolve all 8 physical quantities (cfr. sec 3.2), with a time resolution of 1 timestep, on the whole dataset. We consider a FNO and an AFNO architecture.

During the pre-training phase both FNO and AFNO seem to perform well, however when evaluated autoregressively neither of the two are able to evolve the system. Our intuition is that the networks learned the identity function, and not the differential operator. In fact, we already know that the time scale of the problem is $\tau \approx 50$, making two subsequent timesteps practically identical. A NN can reach very low MSE losses just by giving as an output the very same input, that is, learning the identity function. To solve this issue, we train the net on a much coarser time scale, with the aforementioned timestep.

Coarser timescale leads to less data

To this end, we filter the dataset, keeping only every 50th timestep. Also, since we want the network to learn the evolution of the system, we select only the simulations that contain more than 500 timesteps, in order to have at least 10 training instances for each configuration. The downside is that we end up with a very limited dataset of 6554 simulations, accounting for $12,877$ total training instances, 1746 timesteps in the validation and 2336 timesteps in the test set. The training is very fast, both because the dataset is small and because lighter models are enough, however these converge to very high losses, which correspond to bad reconstructions.

Training on H_x only is an easier task

To counteract this issue, we decided to simplify the task and train the NNs to evolve only H_x, assuming that an easier task could be accomplished even with less data. Indeed, we find that our best FNO and AFNO obtain good

reconstructions in the pre-training phase and also are able to evolve autoregressively some instances of the training set. However, both models struggle to generalize due to the limited amount of data. At this point, two possible paths can be followed: creating more data from simulations, or using data augmentation techniques. The first would be ideal, but due to the limited amount of time at our disposal, we decide to go for the second option.

Using all data on a coarse grid requires bigger models

Instead of using standard data augmentation techniques, like applying random noise or masking the data, we start by exploiting the existing data that we have. The former techniques are used to effectively create new artificial data instances by modifying the true original ones. However, since we already have a large amount of true high fidelity data, we decide to use that first. Moreover, these data is perfectly suited for our goal, since close timesteps are very similar to each other (and that is precisely the reason why we discarded them in the first place, moving from a 1 timestep prediction to a coarser time resolution of 50 timesteps).

During this stage we consider all the data in our original dataset that contain at least 50 timesteps, and train the NNs to predict 50 timesteps ahead. Even though this is not a data augmentation technique *stricto sensu*, it could be interpreted as such, where we transform both the input and output data, hoping that this will make the NN both generalize better and also more stable during autoregressive inference.

However, this method comes with some drawbacks linked to time and hardware resources. Training with 50 times as much data requires bigger models, that need more GPUs and more time to train. We immediately see that the networks used require more parameters then the ones trained on the fine grid, consolidating our intuition that the previously mentioned models learned the identity function and not a proper evolution operator, which is a more difficult task that requires more training parameters. Also, for the way webdataset deals with data, we need to load big batches during training, be-

cause data is read sequentially. For example, a batch size of 50 consists in all timesteps ranging from $t = 1$ to $t = 50$, but since we train the model to predict 50 timesteps ahead, this effectively translates in just one $(input, target)$ pair (there is no target for $t = 2$ and all the others). Thus, we decide to train with a batch size of 200. However, given this batch size and the necessity for bigger models, 4 GPUs become necessary. This is not a problem *per se*, even though the queuing time to access these resources on MareNostrum 5 becomes very large and impedes agile experimenting, taking more than 40 hours to start. So, we perform experiments with the largest possible model that fits in 2 GPUs, which can be accessed with much lower queuing times (almost immediately). These models are suboptimal, since we see empirically that heavier models perform better during grid search.

Good results by learning H_x and H_y

The best results are obtained by training with two channels, corresponding to the two non-zero components of the magnetic field H_x and H_y. These models are trained on a coarse time grid, predicting one step every 50 MAGNET steps. No data augmentation is applied. We think that, in this case, training is easier compared to using one channel only, probably because H_y can be seen as a 90 degree rotation of H_x, effectively behaving like a data augmentation technique. Also, using two channels instead of eight makes the training easier, since lighter models can be used. Moreover, having a model that predicts the magnetic field is enough to compute all the other quantities analytically. The configuration of the best two models are reported in tables 4.1 and 4.2. In figures 4.3, 4.4, 4.8 and 4.9 you can see the performance of these models in the test set. From figure 4.7 you can see that lower losses are obtained with FNO.

The best AFNO model we obtained has nearly 100 times fewer parameters than the best FNO, leading to reduced performance in our experiments. While we tested larger AFNO models, increasing the parameter count did not yield substantial improvements. We attribute this to AFNO's lower inductive bias compared to FNO, meaning it relies more on learning from

data rather than leveraging built-in structural assumptions. This suggests that AFNO could outperform FNO given sufficient training data, but under limited data conditions, FNO's structured approach remains advantageous.

We also find that both model perform well only on some data, while in others they almost completely mistake the evolution of the system, see figures 4.3, 4.4, 4.8 and 4.9. We tried to find correlations between the simulation parameters and the model performance, however we have not found conclusive results, which are left as future works. Lastly, in Figure 4.12, we quantitatively assess how much the model diverges from the ground truth during autoregressive evolution. We observe that the median MSE of AFNO over time is best approximated by a second-order polynomial, whereas for FNO, a linear fit provides the best approximation. On one hand, this suggests that FNO maintains better stability during autoregressive inference. On the other hand, we find that FNO exhibits a much higher variance in its predictions, indicating greater inconsistency across different runs.

Architecture	**AFNO**
`Patch size`	[2, 2]
`Embedding dimension`	512
`Depth`	8
`Number of blocks`	2
`MLP ratio`	4.0
`Learning rate`	0.001
`ADAM` β_1	0.9
`Weight decay`	0.0
`Cosine annealing`	True
`Total number of parameters`	21.6 M
Test loss	0.68
`Mean time for one forward pass`	$0.515 \pm 0.001s$

Table 4.1: Hyper parameters of the best AFNO model, obtained with the methods described in chapter 3

In tables 4.1 and 4.2 we report also the test loss and the average time to perform one forward pass. As expected, both models are very fast during inference. As a figure of reference, on average one MAGNET timestep required

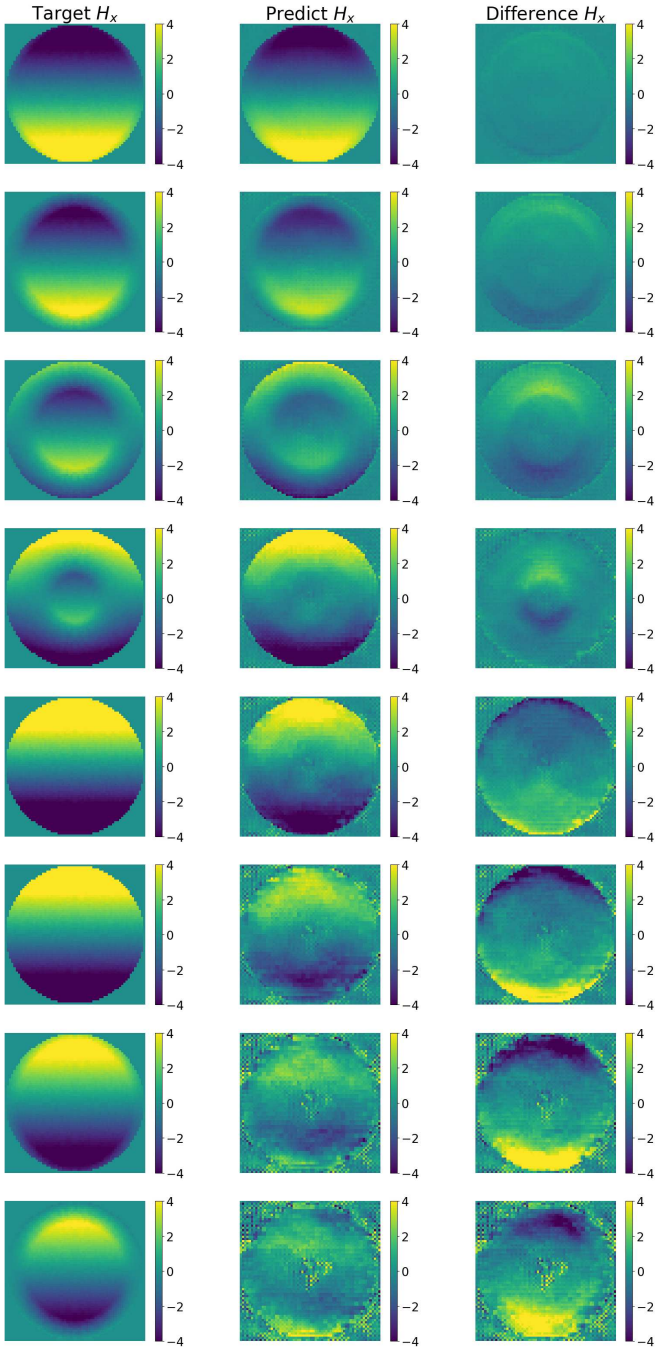

Figure 4.3: AFNO autoregressive evolution of the H_x component. Sample taken from the test set. Simulation parameters are: $E_c = 0.0359$, $n = 27.2$, $i_0 = 330A$. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last column shows the point wise difference between the two. The net is able to evolve the system for 4 timesteps, before degrading the output.

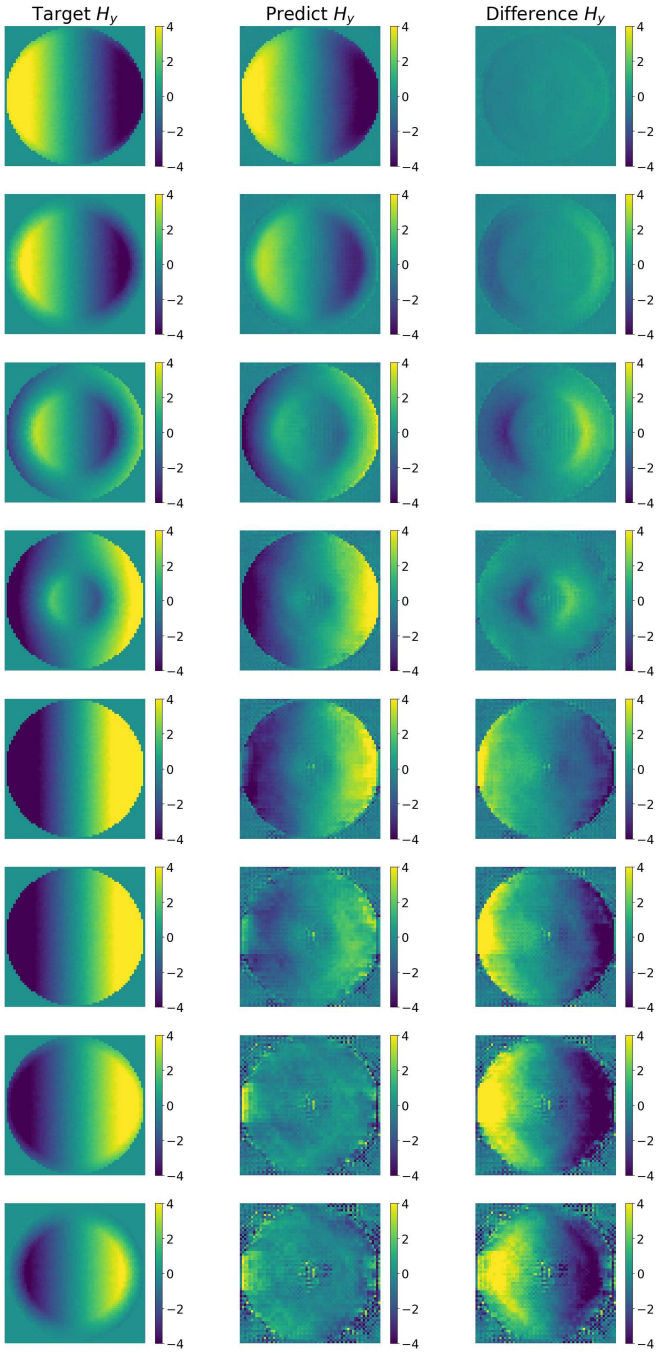

Figure 4.4: AFNO autoregressive evolution of the H_y component. Sample taken from the test set. Simulation parameters are: $E_c = 0.0359$, $n = 27.2$, $i_0 = 330A$. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last one shows the point wise difference between the two. The net is able to evolve the system for 4 timesteps, before degrading the output.

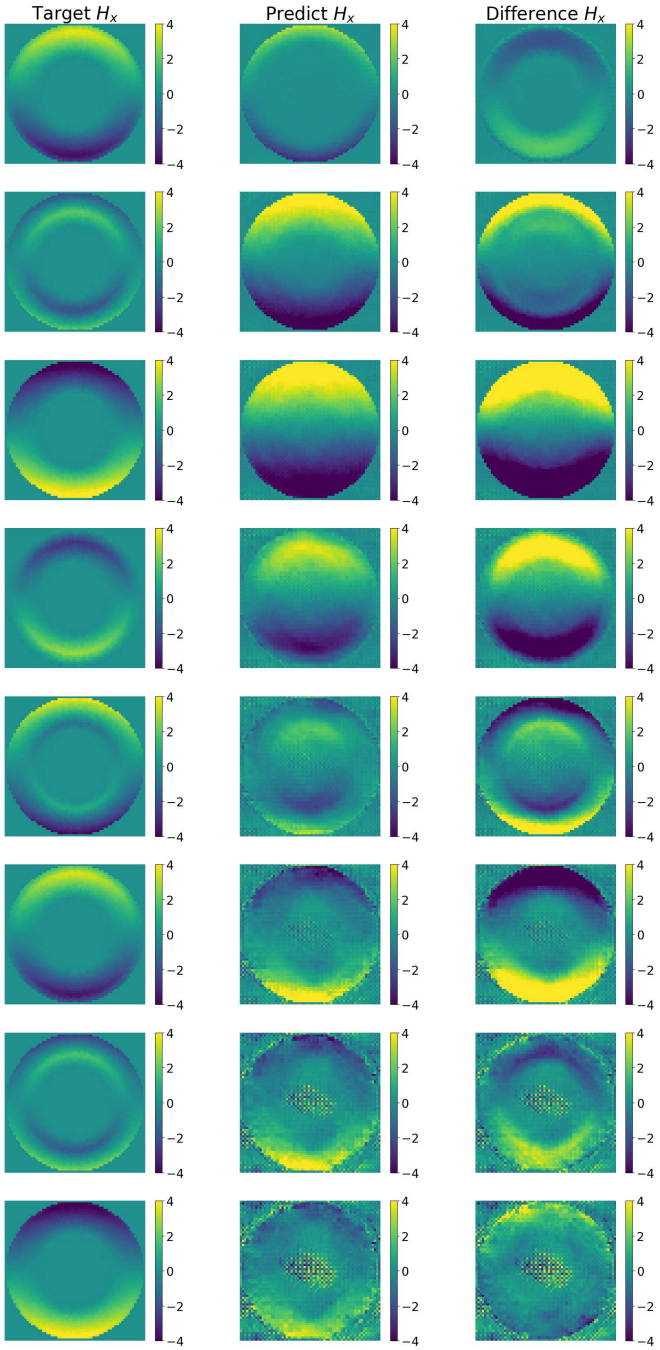

Figure 4.5: Example of failure in AFNO autoregressive evolution, H_x component. Sample taken from the test set. Simulation parameters are: $E_c = 0.0006$, $n = 21.7$, $i_0 = 200A$. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last one shows the pointwise difference between the two. The net hallucinates immediately and predicts wrong solutions.

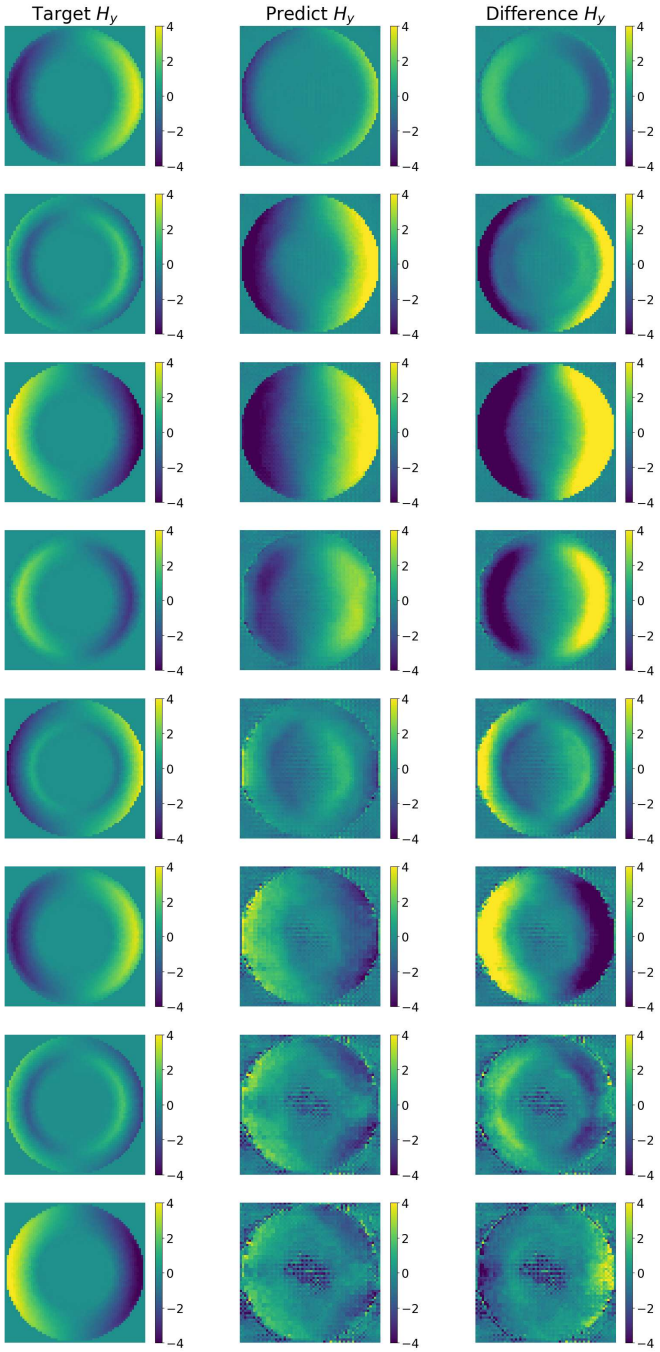

Figure 4.6: Example of failure in AFNO autoregressive evolution, H_y component. Sample taken from the test set. Simulation parameters are: $E_c = 0.0006$, $n = 21.7$, $i_0 = 200A$. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last one shows the pointwise difference between the two. The net hallucinates immediately and predicts wrong solutions.

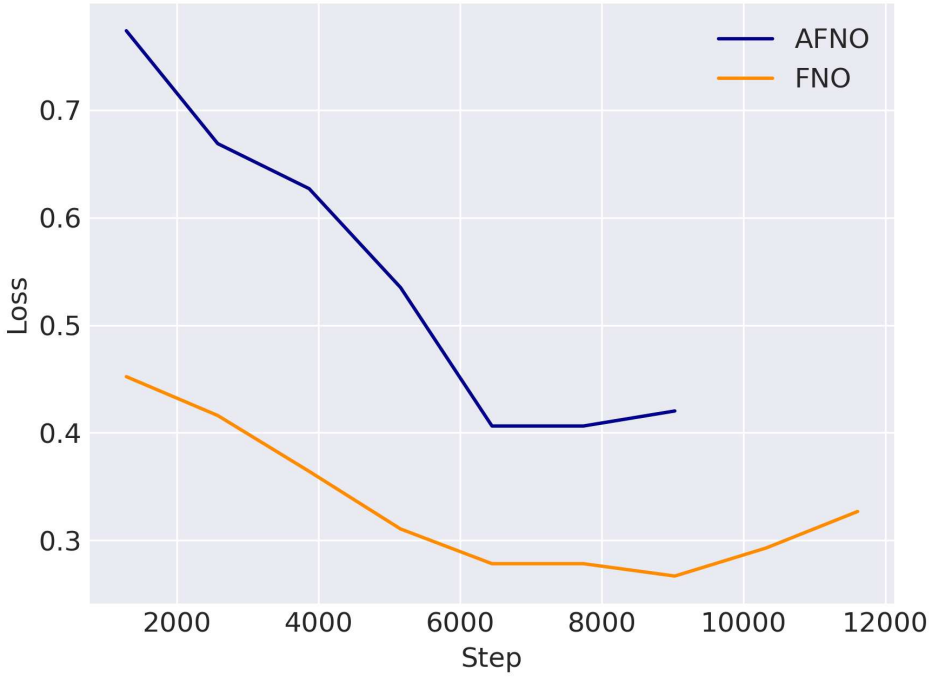

Figure 4.7: Comparison of validation loss curves between the best two models. On the x axis, the number of processed batches are shown; on the y axis the relative MSE of the models. Despite having almost 100 times less parameters, AFNO has slightly worse losses than FNO. Training is stopped when the validation loss does not improve of 0.01 after 2 epochs.

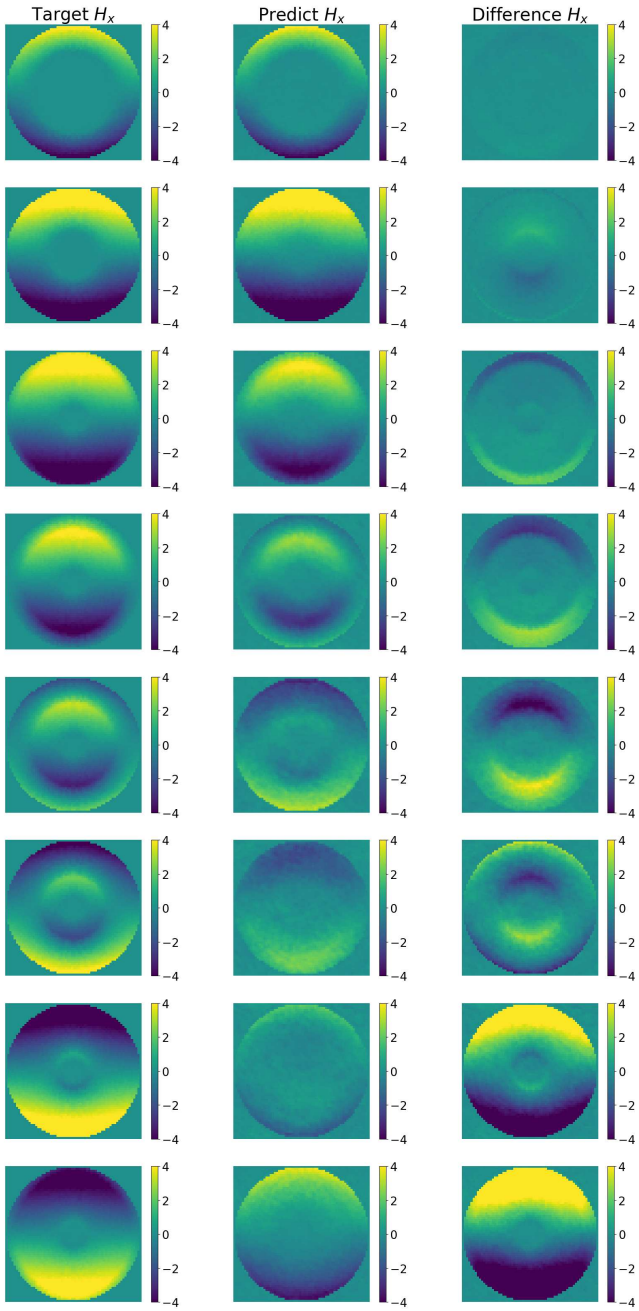

Figure 4.8: FNO autoregressive evolution of the H_x component. Sample taken from the test set. Simulation parameters are: $E_c = 0.0359$, $n = 27.2$, $i_0 = 330A$. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last one shows the pointwise difference between the two. The net is able to evolve the system for almost 5 timesteps, before diverging significantly from the ground truth. Compared to AFNO, the output does not present noisy features.

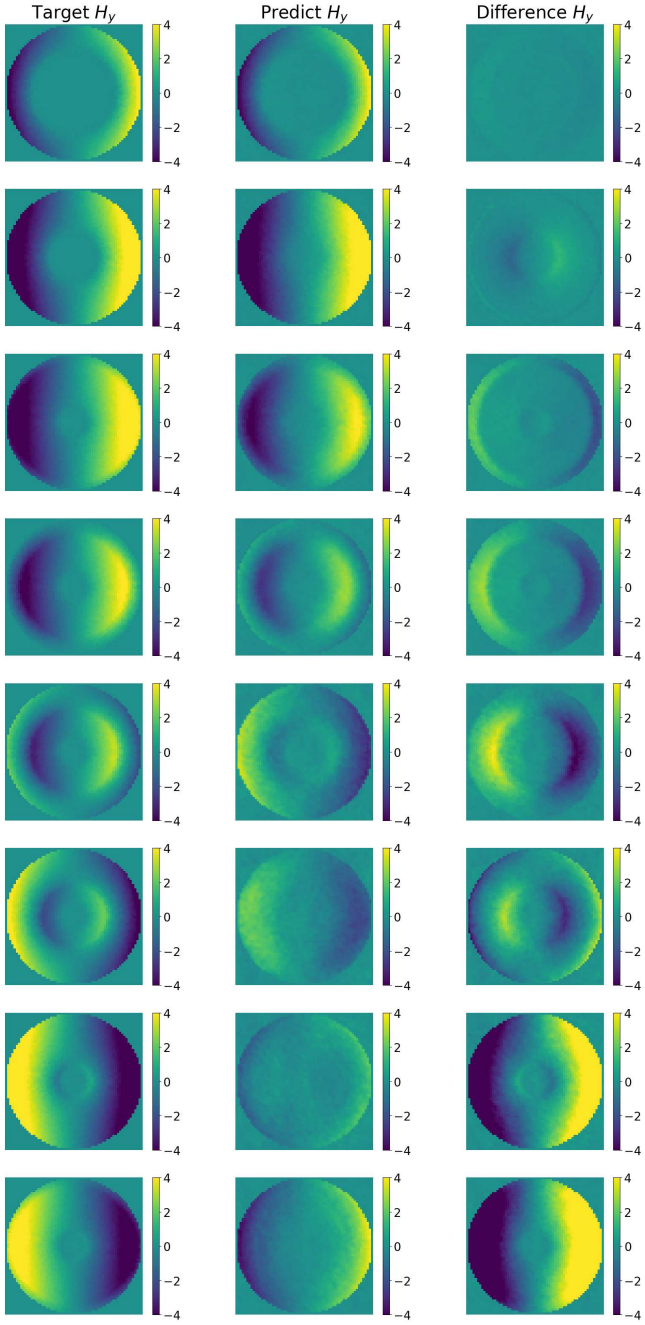

Figure 4.9: FNO autoregressive evolution of the H_y component. Sample taken from the test set. Simulation parameters are: $E_c = 0.0359$, $n = 27.2$, $i_0 = 330A$. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last one shows the pointwise difference between the two. The net is able to evolve the system for almost 5 timesteps, before diverging significantly from the ground truth. Compared to AFNO, the output does not present noisy features.

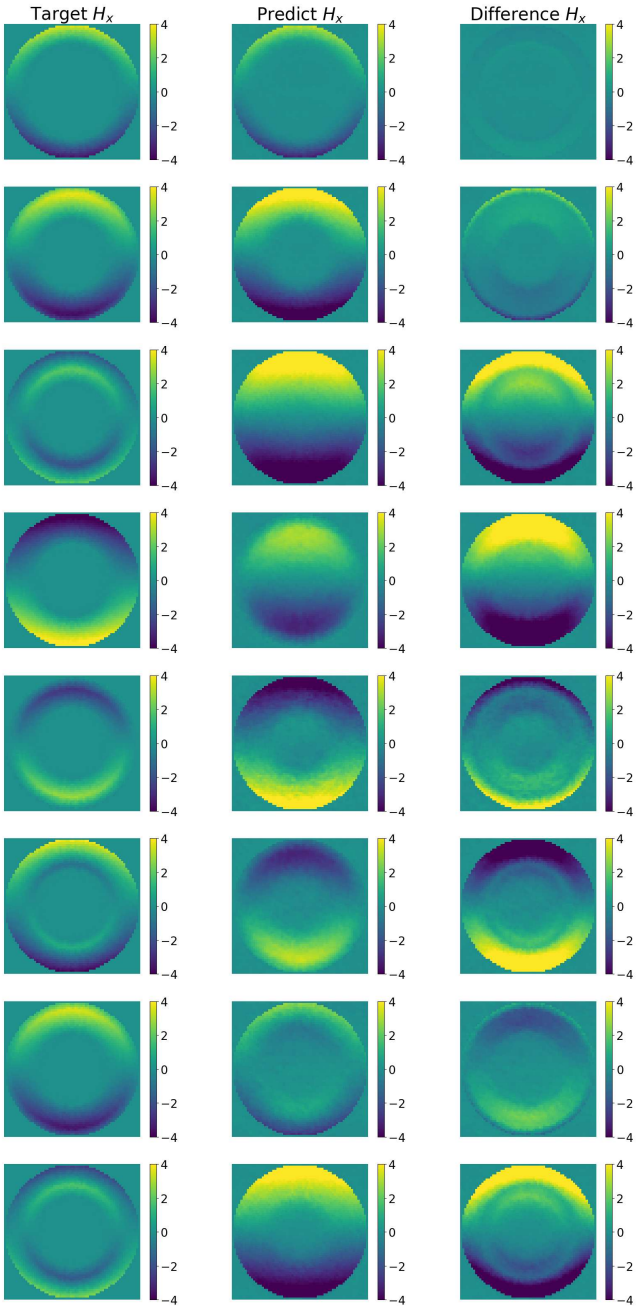

Figure 4.10: Example of failure in FNO autoregressive evolution, H_x component. Sample taken from the test set. Simulation parameters are: $E_c = 0.0006$, $n = 21.7$, $i_0 = 200A$. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last one shows the pointwise difference between the two. As in AFNO, the net hallucinates immediately and predicts wrong solutions.

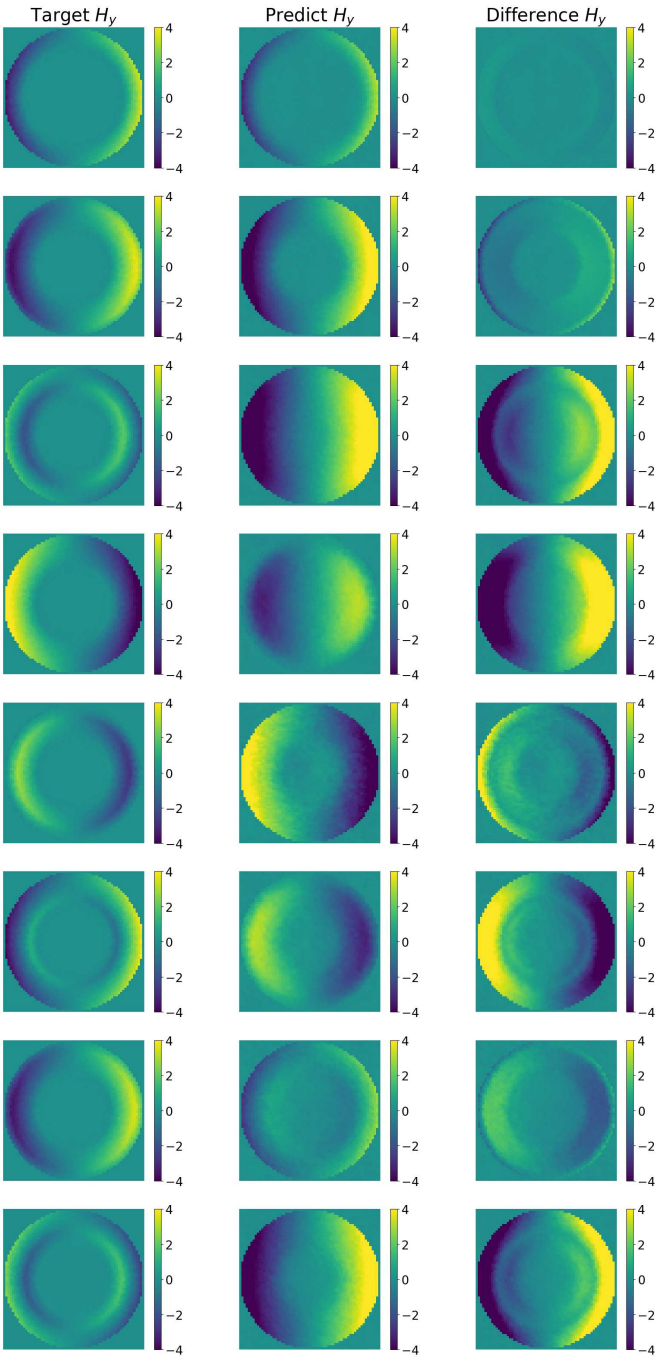

Figure 4.11: Example of failure in FNO autoregressive evolution, H_y component. Sample taken from the test set. Simulation parameters are: $E_c = 0.0006$, $n = 21.7$, $i_0 = 200A$. Each row is a different time step of the evolution. The first column shows the target simulation as performed by MAGNET; the central column shows the predictions made by the NN; the last one shows the pointwise difference between the two. The net hallucinates immediately and predicts wrong solutions.

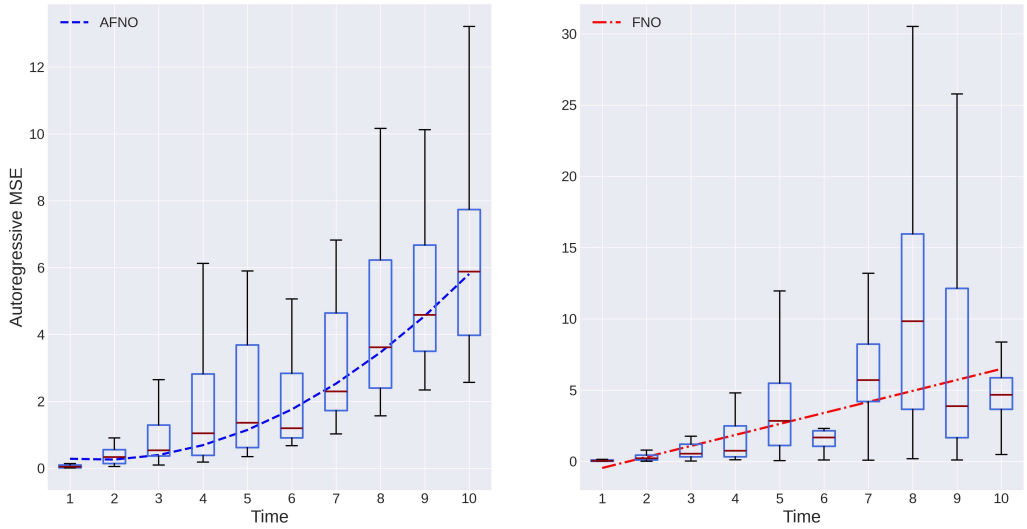

Figure 4.12: Time dependency of MSE during autoregressive inference. Comparison between AFNO and FNO. The losses are evaluated on simulations of the test set. The boxes extend from the first quartile to the third quartile of the data. The red line is the median. The whiskers extend from the box to the farthest data point lying within $1.5x$ the inter-quartile range (IQR) from the box. AFNO median values are best fit by a second-order polynomial, with coefficients $0.08x^2 - 0.26x + 0.47$. The FNO median values are best fit from a straight line, with slope $m = 0.77$ and intercept $q = -1.24$. MSE grows faster in AFNO than in FNO, however FNO presents greater outliers.

Architecture	**FNO**
`Decoder layers`	2
`Decoder layer size`	128
`Latent channels`	512
`Number of FNO layers`	8
`Learning rate`	0.0001
`ADAM` β_1	0.9
`Weight decay`	0.0
`Cosine annealing`	True
`Total number of parameters`	2.1 B
`Test loss`	0.60
`Mean time for one forward pass`	$0.256 \pm 0.034s$

Table 4.2: Hyper parameters of the best FNO model, obtained with the methods described in chapter 3

$2.652 \pm 0.677s$ to compute. All these timings refer to the simulations in the test set. However, these figures have to be taken with a grain of salt, avoiding to strike a direct comparison between the AI and the numerical solver. In fact, the NNs and MAGNET have a very different degree of accuracy and comparing the time performance of the two, without taking it into account, would be unfair. Moreover, the AI timing performance is measured from a plain python script, which can be made faster using compiled code. Nevertheless, as will be discussed further in the next section, our results show that an AI-accelerated approach to numerical simulations is possible, with AI models showing faster performance, sacrificing accuracy in the long term.

Chapter 5

Conclusions and future works

In conclusion, informed by the recent advancements in deep learning and, in particular, in its applications to science (SciML), we developed an AI surrogate model with the goal of accelerating numerical simulations of HTS. To this end, we created a dataset composed of 9100 high-fidelity simulations with MAGNET, a Maxwell's equations numerical solver. We exploited the hardware resources of MareNostrum 5, the supercomputer at the Barcelona Supercomputing Center, to perform various tests on different NN architectures, assessing the performance of a neural operator (FNO) and its transformer-like variant (AFNO). We saw that training struggles if the time resolution is too fine, and the NNs benefit from coarser time scales that capture the global time evolution of the system more easily. The final models obtained are able to evolve the magnetic field of the system for about four to five steps in this coarse grid, corresponding to 200-250 timesteps of the MAGNET simulations. However, our best networks are able to evolve only some of the configurations in the dataset, struggling to generalise to all the physical set-ups. Future work should investigate more deeply why this happens. In any case, we showed that the approach is feasible and that further works can be done to improve the AI models. In the methodology section we have already outlined possible strategies to tackle the problem, for example by using data augmentation techniques and heavier models. Also, a possible

tentative can be done by implementing all the physical quantities produced by MAGNET, in order to have specialised models that can find correlations between the dissipated energy and the induced current, with the magnetic field values.

Even though our model is not precise enough to completely substitute a numerical solver, it could be used as an integration to it, to make it converge faster. In fact, given a solution at whatever timestep t, our model is able to predict with high accuracy the corresponding solution at time $t + 1$ in the coarse grid. Such a model could be used effectively as initial guess for time-parallelizing algorithms, as already attempted in literature [35]. Moreover, our model can be effectively chosen for time-constrained applications, for example in the context of real-time simulation-based inference, as shown in the original FNO paper [23]. Future work should focus on integrating traditional physics-based methods with data-driven approaches to leverage the strengths of both worlds. Additionally, engineers must weigh the benefits and limitations of numerical and AI-based methods, selecting the most appropriate approach based on the specific requirements of their applications.

It is also important to highlight the limitations of the methodology followed in this work. First of all, it is clear that a very big amount of data is necessary to train these kinds of models, even on the very simple scenario that we considered. As discussed in chapter 3.2.1, the creation of a high fidelity dataset requires not only a long time to create and plenty of computing resources, but also a lot of storage, the latter being the biggest limitation in the MareNostrum supercomputing facility. However, this should not be considered when comparing the speed of a trained NN to that of a numerical solver, as these are sunk costs.

Another limitation consists in the choice of the architectures. Both FNO and AFNO take structured mesh as an input, while MAGNET solves the Maxwell's equations on an unstructured grid. To use the two combined, data had to be interpolated, causing possible artifacts in the input data and losing the high-fidelity information coming from numerical simulations. To

overcome this issue, future works shall focus on graph-based networks.

Moreover, in our approach we treated time as an additional independent variable with space. However, models that take into account the sequential nature of time evolution could be used, for example Long Short-Term Memory networks (LSTMs) [36] or Neural ODEs [37], which could improve the performance during autoregressive evolution.

Overall, this thesis provides a solid starting point for the integration of numerical solvers with AI surrogate models, in the context of HTS for nuclear fusion reactors. We assessed the challenges and limitations of this approach, while also showing how future works can build on it to make these models more precise and useful in practical applications.

Bibliography

[1] A. Soba et al. "A high-performance electromagnetic code to simulate high-temperature superconductors". In: *Fusion Engineering and Design* 201 (2024), p. 114282. ISSN: 0920-3796. DOI: https://doi.org/10.1016/j.fusengdes.2024.114282. URL: https://www.sciencedirect.com/science/article/pii/S0920379624001352.

[2] Shan Liu and Gui Yu. "Fabrication, energy band engineering, and strong correlations of two-dimensional van der Waals moiré superlattices". In: *Nano Today* 50 (2023), p. 101829. ISSN: 1748-0132. DOI: https://doi.org/10.1016/j.nantod.2023.101829. URL: https://www.sciencedirect.com/science/article/pii/S1748013223000786.

[3] Zachary S. Hartwig et al. "The SPARC Toroidal Field Model Coil Program". In: *IEEE Transactions on Applied Superconductivity* 34.2 (2024), pp. 1–16. DOI: 10.1109/TASC.2023.3332613.

[4] Michael J. Wolf et al. "High temperature superconductors for fusion applications and new developments for the HTS CroCo conductor design". In: *Fusion Engineering and Design* 172 (2021), p. 112739. ISSN: 0920-3796. DOI: https://doi.org/10.1016/j.fusengdes.2021.112739. URL: https://www.sciencedirect.com/science/article/pii/S0920379621005159.

[5] A. Soba et al. "A high-performance electromagnetic code to simulate high-temperature superconductors". In: *Fusion Engineering and Design* 201 (Apr. 2024), p. 114282. ISSN: 0920-3796. DOI: 10.1016/j.fusengdes.2024.114282. URL: https://www.sciencedirect.com/science/article/pii/S0920379624001352 (visited on 08/03/2024).

[6] M.Vázquez et al. "Alya: Multiphysics engineering simulation toward exascale". In: *Journal of Computational Science*. The Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills 14 (May 2016), pp. 15–27. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2015.12.007. URL: https://www.sciencedirect.com/science/article/pii/S1877750315300521 (visited on 08/02/2024).

[7] *Mare Nostrum 5 System Overview*. URL: https://www.bsc.es/supportkc/docs/MareNostrum5/overview/.

[8] Jaideep Pathak et al. *FourCastNet: A Global Data-driven High-resolution Weather Model using Adaptive Fourier Neural Operators*. 2022. arXiv: 2202.11214 [physics.ao-ph]. URL: https://arxiv.org/abs/2202.11214.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[10] Dan Hendrycks and Kevin Gimpel. "Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units". In: *CoRR* abs/1606.08415 (2016). arXiv: 1606.08415. URL: http://arxiv.org/abs/1606.08415.

[11] Alfio Quarteroni, Paola Gervasio, and Francesco Regazzoni. *Combining physics-based and data-driven models: advancing the frontiers of research with Scientific Machine Learning*. 2025. arXiv: 2501.18708 [math.NA]. URL: https://arxiv.org/abs/2501.18708.

[12] B. Moseley. "Physics-informed machine learning: from concepts to real-world applications [PhD thesis]". PhD thesis. University of Oxford., 2022.

[13] Christopher Rackauckas et al. *Universal Differential Equations for Scientific Machine Learning*. 2021. arXiv: 2001.04385 [cs.LG]. URL: https://arxiv.org/abs/2001.04385.

[14] João Pedro Souza de Oliveira et al. "Coupling a neural network technique with CFD simulations for predicting 2-D atmospheric dispersion analyzing wind and composition effects". In: *Journal of Loss Prevention in the Process Industries* 80 (2022), p. 104930. ISSN: 0950-4230. DOI:

https://doi.org/10.1016/j.jlp.2022.104930. URL: https://www.sciencedirect.com/science/article/pii/S0950423022002066.

[15] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. URL: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[16] George Em Karniadakis et al. "Physics-informed machine learning". In: *Nature Reviews Physics* 3 (June 2021), pp. 422–440. DOI: 10.1038/s42254-021-00314-5. URL: https://doi.org/10.1038/s42254-021-00314-5.

[17] Chengping Rao, Hao Sun, and Yang Liu. *Physics informed deep learning for computational elastodynamics without labeled data*. 2020. arXiv: 2006.08472 [math.NA]. URL: https://arxiv.org/abs/2006.08472.

[18] Shengze Cai et al. "Physics-informed neural networks (PINNs) for fluid mechanics: a review". In: *Acta Mechanica Sinica* 37 (2021), pp. 1727–1738. DOI: 10.1007/s10409-021-01148-1. URL: https://doi.org/10.1007/s10409-021-01148-1.

[19] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[20] Nikola B. Kovachki et al. "Neural Operator: Learning Maps Between Function Spaces". In: *CoRR* abs/2108.08481 (2021). arXiv: 2108.08481. URL: https://arxiv.org/abs/2108.08481.

[21] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[22] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *CoRR* abs/2010.11929 (2020). arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

[23] Zongyi Li et al. “Fourier Neural Operator for Parametric Partial Differential Equations”. In: *CoRR* abs/2010.08895 (2020). arXiv: 2010.08895. URL: https://arxiv.org/abs/2010.08895.

[24] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[25] John Guibas et al. “Adaptive Fourier Neural Operators: Efficient Token Mixers for Transformers”. In: *CoRR* abs/2111.13587 (2021). arXiv: 2111.13587. URL: https://arxiv.org/abs/2111.13587.

[26] HJ. Wu Y. et al. Han P. Liu. “Design and Fabrication of a Hybrid HTS Magnet for 150 kJ SMES”. In: *J Fusion Energ 33, 759–764* (2014).

[27] Bachmann et al. “Influence of a high magnetic field to the design of EU DEMO”. In: *Fusion Engineering and Design 197* (2023).

[28] Shikov et al. “Development of the superconductors for ITER magnet system”. In: *Journal of Nuclear Materials 258-263* (1998).

[29] Wei-bin Xi et al. “Study on the Operation Safety of HTS Current Leads in EAST Device”. In: *J Fusion Energ* (2015).

[30] *Introduction to Large Scale Deep Learning - Thomas M. Breuel*. URL: https://www.youtube.com/watch?v=kNuA2wflygM.

[31] Scikit learn - Preprocessing data.

[32] Varun Godbole et al. *Deep Learning Tuning Playbook*. Version 1.0. 2023. URL: http://github.com/google-research/tuning_playbook.

[33] Christopher J. Shallue et al. “Measuring the Effects of Data Parallelism on Neural Network Training”. In: *CoRR* abs/1811.03600 (2018). arXiv: 1811.03600. URL: http://arxiv.org/abs/1811.03600.

[34] William Falcon and The PyTorch Lightning team. *PyTorch Lightning*. Version 1.4. Mar. 2019. DOI: 10.5281/zenodo.3828935. URL: https://www.pytorchlightning.ai.

[35] S.J.P. Pamela et al. “Neural-Parareal: Self-improving acceleration of fusion MHD simulations using time-parallelisation and neural operators”.

In: *Computer Physics Communications* 307 (2025), p. 109391. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2024.109391. URL: https://www.sciencedirect.com/science/article/pii/S001046552400314X.

[36] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[37] Tian Qi Chen et al. "Neural Ordinary Differential Equations". In: *CoRR* abs/1806.07366 (2018). arXiv: 1806.07366. URL: http://arxiv.org/abs/1806.07366.