

A Hands-On Approach to Scientific Artificial Intelligence Programming

Iciar Alfaro, Carlos Bermejo, Elías Cueto, David González,
Mikel M. Iparraguirre, Lucas Tesán, Alicia Tierz, Pau Urdeitx

*A HANDS-ON APPROACH
TO SCIENTIFIC ARTIFICIAL
INTELLIGENCE PROGRAMMING*

*A HANDS-ON APPROACH
TO SCIENTIFIC ARTIFICIAL
INTELLIGENCE PROGRAMMING*

*Iciar Alfaro, Carlos Bermejo, Elías Cueto, David González,
Mikel M. Iparraguirre, Lucas Tesán, Alicia Tierz, Pau Urdeitx*



Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra solo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

- © Iciar Alfaro, Carlos Bermejo, Elías Cueto, David González, Mikel M. Iparraguirre, Lucas Tesán, Alicia Tierz, Pau Urdeitx
- © De la presente edición, Prensas de la Universidad de Zaragoza
(Vicerrectorado de Cultura y Patrimonio)
1.ª edición, 2026

Colección de Textos Docentes, n.º 337

Prensas de la Universidad de Zaragoza. Edificio de Ciencias Geológicas, c/ Pedro Cerbuna, 12, 50009 Zaragoza, España. Tel.: 976 761 330
puz@unizar.es <http://puz.unizar.es>



Esta editorial es miembro de la UNE, lo que garantiza la difusión y comercialización de sus publicaciones a nivel nacional e internacional.

ISBN 979-13-7014-158-5

Impreso en España

Imprime: Servicio de Publicaciones. Universidad de Zaragoza

D.L.: Z 1161-2026

To our students.

Acknowledgements

We would like to thank Professor Francisco Chinesta for the support he has given us over the years; without his friendship and scientific guidance, the development of a new line of research within our group would not have been possible.

The authors also acknowledge the support of the Spanish Ministry of Science and Innovation, through AEI/10.13039/501100011033 (grant no. PID2023-147373OB-I00). The authors acknowledge the support of the Ministry for Digital Transformation and the Civil Service, through the ENIA 2022 Chairs for the creation of the university-industry chairs in Artificial Intelligence, through grant TSI-100930-2023-1. The authors also acknowledge the support by Keysight-ESI Group through the ENIA Chair, as well as our students for their hard work, both in attending the courses and in the development of this material.

1. Introduction

This book brings together the teaching materials used in the courses on scientific artificial intelligence delivered by the Keysight-UZ Chair of National AI Strategy of the Ministry of Digital Transformation at the University of Zaragoza during the years of its existence, from 2024 onwards.

As will be seen, the material takes a predominantly practical approach and is designed to enable students to experiment with all the modules in the programme.

This course is designed for postgraduate students. Many aspects will be better understood if students have been introduced to basic concepts of physics during their undergraduate studies, particularly mechanics, which is the focus of this book. If they have a background in scientific computing—and particularly in computational engineering—and are familiar with the finite element, finite volume, and finite difference methods, they are sure to find the content both accessible and engaging!

The field of scientific artificial intelligence is fascinating. The possibility that artificial intelligence can make predictions—whether using experimental or synthetic data, or relying solely on the differential equations that govern the phenomena under study—is, quite simply, fascinating. The challenge is for this artificial intelligence to be truly predictive in an engineering sense: that the predictions it makes can be verified and validated, so that industry can ultimately place its trust in the benefits this new discipline can bring.

It is the task of engineers, physicists, and mathematicians to turn this family of techniques into the tool that industry needs right now. The responsibility lies with us, and it begins with training a new generation of researchers in the field who will, no doubt, bring this to a successful conclusion.

2. First steps: Environment installation

This chapter provides a comprehensive guide to setting up a robust Python environment for machine learning development. The necessary instructions for installing the key machine learning libraries—NumPy, Matplotlib, TensorFlow, PyTorch, ...—are included. This chapter is intended for beginners and intermediate users, to equip them with the tools and knowledge needed to work in machine learning.

It is important to establish a reproducible Python environment using Conda to manage project-specific dependencies. In particular, set up essential tools, including PyTorch for deep learning development. There is a framework that abstracts repetitive code to build more scalable, readable, and modular deep learning pipelines, called `PyTorch Lightning`. It will be useful to generate a requirements file to document all dependencies and achieve consistent configuration across all systems.

2.1 Set-up of the working environment

For first steps, begin by setting up a battle-tested development environment—this same configuration will power all of this book’s projects, ensuring consistent behavior across every application.

At this point, a question may arise: are Conda environments indispensable? Working with isolated Conda environments is crucial for three key reasons:

- **Dependency management:** different projects often require conflicting package versions. Conda environments prevent the “dependency hell” problem by keeping each project’s requirements separate.
- **Conda allows you to specify and install a particular version of the Python interpreter for each environment.** This is essential for maintaining compatibility with legacy code or testing your projects against the latest releases without affecting the rest of your system.
- **Reproducibility:** an environment file (`environment.yml`) lets exactly recreate the same setup across different machines and collaborators — critical for reliable results.

- Project Isolation Experiments will not interfere with each other or break your base Python installation. The user can safely test new packages without system-wide consequences.

Pro tip

Always create a new environment per project, and document dependencies using:

```
conda env export > environment.yml
```

```
# Install versions with known parameter mismatch
!pip install -q seaborn==0.10.1
# Install a compatible matplotlib version
!pip install -q matplotlib==3.2.2 --no-cache-dir

import seaborn as sns
import matplotlib.pyplot as plt

print("seaborn version:", sns.__version__)
print("matplotlib version:", plt.matplotlib.__version__)

try:
    # 'sns.histplot' didn't exist in seaborn 0.10.1, so this will raise
    # an AttributeError
    sns.histplot(data=[1, 2, 2, 3, 3, 3])
    plt.title("Histogram")
    plt.show()
except Exception as e:
    print("Unexpected error:")
    print(e)
```

2.1.1 The nontrivial consequences of ignoring environments

Let's see a real example of version conflicts. In Table 2.1, we can check the compatibility summary table for Torch, Lightning, NumPy, and Matplotlib.

To add some context to the table:

- TensorFlow and Numba require NumPy ≤ 2.1 , so NumPy 2.x conflicts with them.
- Thinc (used by spaCy) requires NumPy ≥ 2.0 , conflicting with PyTorch ecosystems needing NumPy 1.x.

Library	Recommended Version	Key Dependencies / Requirements	Common Incompatibilities	Notes / Recommendations
PyTorch (torch)	2.0.x, 2.1.x	Compatible with NumPy ≥ 1.20 and ≤ 2.0	NumPy ≥ 2.0 may cause internal errors or binding issues	Keep NumPy ≤ 2.0 to avoid C/C++ binding problems with PyTorch
PyTorch Lightning	2.0.x	Depends on compatible PyTorch	Same as PyTorch, relies on NumPy ≤ 2.0	Generally no issues with Matplotlib; no strict Matplotlib version restrictions
NumPy	1.26.x	Numerical base, library compatibility	NumPy 2.x causes incompatibility with TensorFlow, Numba, SciPy, Thinc, others	Version 1.26.x is stable for most frameworks; NumPy 2.x is very new and can cause conflicts
Matplotlib	3.8.x	Compatible with NumPy ≥ 1.17	Some functions like <code>stem(use_line_collection=True)</code> changed after 3.1	Matplotlib 3.8.x works fine with NumPy 1.26.x; watch for API changes in specific functions

Table 2.1: Library compatibility and recommendations

- Google Colab preinstalls some dependencies with fixed versions (pandas, packaging, tensorflow), which may cause version conflicts.
- Matplotlib function signatures may have changed between versions; check documentation when using arguments like `use_line_collection`.

To build a small script to automatically check installed library versions and suggest compatible upgrades/downgrades. Take a look to key missing or changed features across versions:

- PyTorch Lightning: trainer module/class.
 - NOT available in very early versions (≤ 1.0).
 - Present and stable in versions 1.0+.
- PyTorch (torch): Complex CUDA support.
 - Enhanced in recent versions; older versions lack features for newer GPU architectures.
- PyTorch Geometric (PyG): Version-specific compatibility for Graph Neural Networks.
 - Essential for the advanced GNN architectures discussed in subsequent chapters.

- PyG often lags behind the latest PyTorch releases. It requires specific binaries that must match both the PyTorch version and the CUDA toolkit, making version pinning indispensable to avoid installation failures.
- NumPy: Version 2.x.
 - Backward-incompatible changes in internal C API, causing binding errors with some libraries.
 - New functions and deprecations compared to 1.x (e.g., changes in random API).
- Matplotlib: `plt.hist()` vs `sns.histplot()` (Seaborn)
 - `sns.histplot` introduced in seaborn 0.11.0, absent in older versions.

So, for a brief summary:

- Check the official changelogs or documentation for breaking changes.
- When backporting code to older versions, expect missing modules/classes or different method signatures.
- For new features, upgrade cautiously and verify dependency compatibility.

The next code install seaborn and matplotlib compatible with pandas 2.x in Colab:

```
# Installing specific, compatible versions of seaborn and matplotlib
!pip install -q seaborn==0.13.1

import seaborn as sns
import matplotlib.pyplot as plt

print("seaborn version:", sns.__version__)
print("matplotlib version:", plt.matplotlib.__version__)

try:
    sns.histplot(data=[1, 2, 2, 3, 3, 3])
    plt.title("Histogram (compatible)")
    plt.show()
except Exception as e:
    print("Unexpected error:")
    print(e)
```

2.2 Conda environment

Let us look at the few steps to follow for installing the Conda environment. Although the instructions below are specifically tailored for Google Colab to ensure a smooth start for everyone, we strongly recommend following a similar setup on your local machine. The logic, commands, and environment management workflow remain nearly identical, making it easy to transition your projects from the cloud to a professional local development workstation.

2.2.1 Step 1: Install Conda

Miniconda is a lightweight Conda distribution that allows us to manage isolated Python environments and packages [Anac16]. In this step, how to install Miniconda in the Colab system and configure the necessary environment variables so the system recognizes Conda commands, are described. This is essential because Colab doesn't come with Conda pre-installed like other environments.

In this step it is necessary:

- Download the Miniconda installer.
- Make the installer executable.
- Run the installation to /usr/local.
- Configure the system PATH to include Conda.
- Verify the Conda installation.

This first step is done by running the next code.

```
# Download and install Miniconda silently in /usr/local
!wget -nc https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
!chmod +x Miniconda3-latest-Linux-x86_64.sh
!bash ./Miniconda3-latest-Linux-x86_64.sh -b -f -p /usr/local

# Add conda to the current Python environment's PATH
import sys
sys.path.append('/usr/local/lib/python3.8/site-packages')
import os
os.environ['PATH'] = "/usr/local/bin:" + os.environ['PATH']
# Verify installation
!conda --version
```

2.2.2 Step 2: Create new Conda environment.

Creating isolated environments is a best practice for managing different projects with their own dependencies. Here, an environment called “Zaragoza” with Python 3.10 and basic scientific packages pre-installed, are established. The version checks ensure everything was set up correctly. Conda environments prevent conflicts between package versions across different projects.

In this step, it is necessary:

- Create a new environment named “Zaragoza” with Python 3.10.
- Pre-install core scientific packages (NumPy, SciPy, Matplotlib).
- Check the Python and NumPy versions in our new environment.
- List all available environments to verify creation.

The second step is done by running the next code.

```
# Create a Conda environment named "Zaragoza" with Python 3.10 and
# common packages
!pip install tabulate
!conda create -n Zaragoza python=3.10 numpy scipy pandas matplotlib
    tqdm scikit-learn ipywidgets -y --quiet
# Run Python and check version using the Conda environment
!conda run -n Zaragoza python -c "import sys; print('Python version:',
    sys.version)"
!conda run -n Zaragoza python -c "import numpy as np; print('NumPy
    version:', np.__version__)"

import subprocess
from tabulate import tabulate
# Run 'conda env list' and capture the output
output = subprocess.run(["conda", "env", "list"], capture_output=True,
    text=True)
# Split the output into lines and remove empty lines
lines = output.stdout.strip().split("\n")
# First line is the header, skip the second (divider), process the rest
envs = []
for line in lines[2:]:
    if line.strip(): # ignore blank lines
        parts = line.split()
        name = parts[0]
        path = parts[-1]
        active = "*" if "*" in parts else ""
        envs.append([name, path, active])
# Display results as a compact markdown-style table
print(tabulate(envs, headers=["Environment", "Path", "Active"],
    tablefmt="github"))
```

2.2.3 Step 3: Install additional packages

In this step, we expand the environment's capabilities by installing PyTorch and PyTorch Lightning, essential frameworks for deep learning. We specify the CPU-only version for compatibility with Colab's environment. Installing in the isolated environment ensures these dependencies won't affect other potential environments or the base system.

In this step, it is necessary:

- Install PyTorch suite (torch, torchvision, torchaudio).
- Add PyTorch Lightning from conda-forge.
- Activate the environment to verify installation.

This third step is done by running the next code.

```
# Install PyTorch and PyTorch Lightning in the Conda environment
"Zaragoza"
!conda install -n Zaragoza pytorch torchvision torchaudio cpuonly -c
pytorch -y --quiet
!conda install -n Zaragoza pytorch-lightning -c conda-forge -y --quiet

# Activate the environment (alternative method)
!source activate Zaragoza

# For Colab, you might need to use:
# !conda activate Zaragoza
```

2.2.4 Step 4: Basic environment usage

Now that we have our environment set up, it's important to understand how to properly use it. This step shows the basic commands for activating the environment and verifying you're working within it. Remember to always activate your environment before working on your project.

In this step, it is necessary:

- Properly activate our Conda environment.
- Verify you're working in the correct environment.
- List installed packages.
- Deactivate when finished.

Pro tip

Validate your environment matches ours with:

```
conda list --export > environment_audit.txt
```

This final step is done by running the next code.

```
# Activate the environment (Colab-specific method)
!conda activate Zaragoza
# Verify you're in the correct environment
!conda info
# Check installed packages in the active environment
!conda list -n Zaragoza
# To deactivate when finished (though not strictly necessary in Colab)
!conda deactivate
```

2.2.5 Exporting and recreating Conda environment

Let us export the environment to:

- An environment file: `environment.yml`.
- A requirements file: `requirements.txt`.

```
## Copy the environment with YAML
# List of Packages in Zaragoza environment
print('Packages in Zaragoza:')
!conda list -n Zaragoza | grep -E
    '^(python|numpy|matplotlib|scipy|torch)'

# Export environment to YAML
!conda env export -n Zaragoza > environment.yml
# Create new environment (copy of Zaragoza) from exported YAML
!conda env create -n conda_cpy -f environment.yml
# List of Packages in conda_cpy environment
print('Packages in conda_cpy:')
!conda list -n conda_cpy | grep -E
    '^(python|numpy|matplotlib|scipy|torch)'
```

And recreate the environment on the new system:

- Copy environment with `environment.yml`.
- Create new environment and install packages based on `requirements.txt`.

Contents

ACKNOWLEDGEMENTS	9
1 Introduction	11
2 First steps: Environment Installation	13
2.1 Set-up of the working environment	13
2.1.1 The nontrivial consequences of ignoring environments	14
2.2 Conda environment	17
2.2.1 Step 1: Install Conda	17
2.2.2 Step 2: Create new Conda environment.	18
2.2.3 Step 3: Install additional packages	19
2.2.4 Step 4: Basic environment usage	19
2.2.5 Exporting and recreating Conda environment	20
2.3 Python libraries	21
2.4 PyTorch: Deep Learning framework	25
2.5 Random seed for reproducibility	26
2.5.1 Key sources of randomness	27
2.6 Discussion and conclusions for set-up of the working environment	28
2.7 Instructions to set up the Conda environment locally	29
2.8 Setting up Python environment	30
2.8.1 Testing the final setup: Running a simple machine learning example	33
3 M.O.R.	35
3.1 Dataset and techniques to compare	35
3.2 Reduced Order Modelling (ROM)	36
3.2.1 Swiss-Roll dataset generation	38
3.3 Principal Component Analysis (PCA)	41
3.3.1 Singular Value Decomposition (SVD) and PCA	42
3.4 Kernel PCA (k-PCA)	44
3.4.1 The kernel trick	46
3.4.2 Kernel comparison on Swiss-Roll	47
3.4.3 Kernel PCA with scikit-learn	49
3.5 Proper Generalized Decomposition (PGD)	50

3.5.1	Core idea	50
3.5.2	PGD for Poisson problems	51
3.5.3	Example: 2D Poisson equation with homogeneous Dirichlet conditions	52
3.5.4	Plotting PGD solution functions	54
3.5.5	PGD examples	56
3.6	PGD for parametric problems: Moving load on a cantilever beam	58
3.6.1	Key differences from the Poisson example	58
3.6.2	Algorithm workflow	59
3.6.3	PGD: Key insights and limitations	64
3.7	Autoencoders for latent space construction in scientific machine learning	65
3.8	What is an Autoencoder?	66
3.8.1	Key architectures of an Autoencoder:	66
3.8.2	Key advantages of Autoencoders:	67
3.8.3	Latent space interpolation with autoencoders	67
3.8.4	Autoencoder application: Swiss-Roll	67
3.8.5	Representation of the latent space and reconstruction	71
3.8.6	Why does the latent space look disordered?	72
3.9	Discussion and conclusions for MOR techniques	75
4	From MLPs to CNNs and Autoencoders	77
4.1	Setup	77
4.1.1	Exploring the MNIST dataset	79
4.2	Data normalization and data loaders	79
4.3	MLP classifier (with plain PyTorch)	80
4.3.1	Manual training loop (basic PyTorch)	81
4.3.2	Evaluate accuracy and performance	82
4.4	Convolutional Neural Network (CNN)	83
4.4.1	Evaluate accuracy and performance	86
4.5	EMNIST dataset	86
4.6	Introducing PyTorch Lightning	88
4.6.1	Why use PyTorch Lightning?	88
4.7	Brief reminder of Autoencoders	89
4.8	Variational Autoencoder (VAE)	92

5	Hyperparameter Optimization	97
5.1	Sweep configuration	97
5.2	Training and evaluating sampled hyperparameter sets	99
5.2.1	Launch hyperparameter optimization	100
5.2.2	Final training with best hyperparameters and early stopping	100
5.2.3	Final test with best hyperparameters	102
6	Introduction to Physics Informed Neural Networks	103
6.1	Simple ODE: Sinusoidal velocity	103
6.1.1	Input data: Control points to train the models	104
6.2	Supervised learning: Learning from data	106
6.2.1	Modelling from available data when unknown ODE	106
6.2.2	Models theta	108
6.2.3	Data splitting D_{train} and D_{valid} to avoid overfitting	108
6.2.4	Training process	110
6.3	Self-supervised learning: Deep learning as an ODE solver	111
6.3.1	Modelling from known ODE when unavailable data	112
6.3.2	Loss based on residuals ODE	113
6.3.3	Models theta	115
6.3.4	Training process	115
6.3.5	Generalization: Out of training range	116
6.4	Under-damped harmonic oscillator	117
6.5	Example 1: Data-driven approach without physics knowledge	119
6.6	Example 2: Self-supervised approach with physics only	121
6.6.1	Loss based on residuals ODE	121
6.7	Example 3: Hybrid approach with physics bias	126
6.8	Solving inverse problems with PINNs	128
6.8.1	Example: inverse problem	128
7	Introduction to thermodynamics-informed neural networks	137
7.1	Generalized thermodynamics	138
7.1.1	Poisson bracket: reversible dynamics	138
7.1.2	Dissipative bracket: dissipative dynamics	139
7.1.3	Physics with neural networks: models to compare	141
7.2	Example 1: Elastic pendulum	141
7.2.1	Initialitation	143
7.2.2	Synthetic data	144

7.2.3	Model 1: Black-box approach	145
7.2.4	Model 2: Thermodynamics-Informed Neural Network — Ideal System — Known L Matrix	154
7.2.5	Model 3: Physics-Informed Neural Network — Ideal System — Unknown L Operator	157
7.3	Example 2: Damped elastic pendulum	160
7.3.1	Model 4: Thermodynamics-Informed Neural Network — Complete System — Known L Matrix	162
7.3.2	Model 5: Thermodynamics-informed neural network - Complete system	167
7.4	Example 3: Double thermoelastic pendulum	171
7.5	Additional and related work	175
7.5.1	Tunning hyperparameters	177
7.5.2	Known L Operator in the Double Pendulum	179
8	Graph Neural Networks	181
8.1	Motivation	181
8.2	GNN methodology	182
8.2.1	Geometric structure: Graph neural networks	184
8.3	GNNs-Lightning Setup: Environment Configuration & Reposi- tory Installation	188
8.4	Solving the Poisson equation with graph neural networks	188
8.5	Solving the Wave equation with graph neural networks	191
8.5.1	Hyperbolic problem dataset	192
A	Appendix	195
A.1	Classes for environment installation	195
A.1.1	Metrics Plotter Class	195
A.2	Classes for MOR	198
A.2.1	PGD Class	198
A.2.2	PGD Multiparametric Class	204
A.2.3	Autoencoder Swiss-Roll Class	215
A.3	MLP, CNN, and Autoencoder classes	216
A.3.1	MLP Class	216
A.3.2	CNN Class	216

A.3.3	Lithting Module: LitClassifier Class	217
A.3.4	Convolutional Autoencoder Class	218
A.3.5	VAE Class	219
A.4	Classes for PINNs	221
A.4.1	Solver Class	221
A.5	Classes for TINNs	222
A.5.1	MLP Class (TINN)	222
A.5.2	BlackBox Class	223
A.5.3	Solver Class for TINNs	223
A.5.4	Loss History Callback Class	225
A.5.5	Plot Every N Epochs Class	225
A.5.6	TINN_01 Class	226
A.5.7	TINN_02 Class	227
A.5.8	TINN_03 Class	228
A.5.9	Solver_TINNS Class	230
A.5.10	TINN Class	231
A.6	Classes and functions for GNNs	232
A.6.1	Graph Data Module Class	233
A.6.2	MPNN Graph Dataset Class	236
A.6.3	Poisson Graph Dataset Class	239
A.6.4	Simulator GNN Class	243
A.6.5	Normalizer Class	254
A.6.6	Model Save Class	256
A.6.7	Model Save Top K Class	256
A.6.8	Face To Edge Tethra Class	258
A.6.9	Radius Graph Mesh Class	259
A.6.10	Contact Distance Class	261
A.6.11	Mesh Distance Class	262
A.6.12	Add Steady Step Class	263
A.6.13	Element To Face Class	264
A.6.14	Encoder-Processor-Decoder Class	265
A.6.15	Encoder Class	268
A.6.16	Processor Class	269
A.6.17	Decoder Class	270
A.6.18	Node Processor Module Class	271
A.6.19	Node Type DP Class	293

ISBN 979-13-7014-158-5



TECNOLOGÍAS

PROYECTO FINANCIADO POR SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL



Financiado por
la Unión Europea
NextGenerationEU



GOBIERNO
DE ESPAÑA

MINISTERIO
PARA LA TRANSFORMACIÓN DIGITAL
Y DE LA FUNCIÓN PÚBLICA

SECRETARÍA DE ESTADO
DE DIGITALIZACIÓN
E INTELIGENCIA ARTIFICIAL



Plan de Recuperación,
Transformación
y Resiliencia

España | digital

