

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00	Joint Dinner				

AI as Strategic Transformation

Artificial Intelligence is becoming a core component of the next-generation observation–forecast–service chain.

What changes

- ▶ Some components will be replaced by AI-driven methods
- ▶ Many will be hybridized for better efficiency and accuracy
- ▶ Some remain targeted classical methods — but AI supports their use, development and understanding

Key point

AI transforms the full modeling ecosystem, not just single algorithms.

Outcome

- ▶ faster cycles from idea to product
- ▶ stronger automation in workflows
- ▶ new products and interaction modes

Strategic goal: enable scalable AI adoption without losing operational stability .

AI as Tool of Discovery: From Ideas to Tested Systems

Discovery viewpoint

AI is not only about producing forecasts — it increasingly supports scientific discovery and systematic development .

What AI adds to science and engineering

- ▶ accelerate coding and refactoring of research and operational software
- ▶ help to formulate equations, losses, and constraints in a consistent way
- ▶ generate tests , validation scripts, and diagnostic plots automatically
- ▶ explore hypotheses : patterns, regimes, causal links in large datasets

Result

We move from isolated scripts to AI-supported workflows that iterate fast: idea → prototype → test → deploy.

Key takeaway

AI becomes part of the scientific method: hypothesize, implement, test, improve.



Python Everywhere: The Universal AI Workflow Environment

Course principle

We learn AI techniques inside a **Python-based workflow** — because this is the environment that is **adequate today** for research and operations.

Python runs everywhere

- ▶ **laptop** development and rapid prototyping
- ▶ **online + browser** execution (cloud notebooks, collaboration)
- ▶ **GPU** training environments
- ▶ **Linux + macOS** daily engineering platforms
- ▶ **HPC systems** for scaling and operations

Practical consequences

- ▶ transfer skills **across platforms** with minimal friction
- ▶ the same toolchain supports **research and operations**
- ▶ consistent workflows: **data** , **training** , **inference** , **plots**

Key message

Python is mastering data, models, and operations.

We Use Git Heavily: Reproducibility, Collaboration, Traceability

Course principle

We use **Git** as the backbone for structured work — not optional, but essential.

Why Git matters

- ▶ **reproducibility:** exact versions of code + configs
- ▶ **collaboration:** branching, merging, review culture
- ▶ **traceability:** who changed what, why, and when
- ▶ **safe experimentation:** try ideas without breaking the main line

Operational viewpoint

Modern AI systems require **controlled evolution**: models, code, data interfaces and validation must co-evolve.

Key takeaway

Git makes individual work maintainable and save.

In this course

- ▶ clone / pull / commit daily
- ▶ structured repos for lectures + code
- ▶ CI/CD connection later in the week

Setup your own git repos!

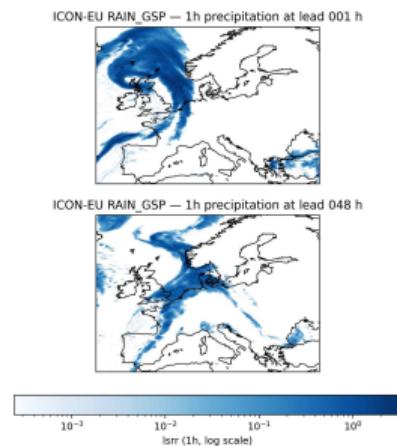
Domain Knowledge Stays Central: AI Needs Science

Message to classical modelers

AI does **not** replace domain expertise. Instead, domain knowledge becomes even more valuable : it determines what is physically meaningful, operationally relevant, and scientifically trustworthy.

Why expertise matters

- ▶ choose variables, scales, and constraints
- ▶ define loss functions and evaluation metrics
- ▶ identify artefacts vs. true signals
- ▶ judge extreme events and non-stationarity



Weather service viewpoint

Forecasting is not only a model: it is an ecosystem of **observation processing, modeling, validation, products and communication**.

Domain knowledge is central in

- ▶ services: impact products, warnings, tailored guidance
- ▶ forecasting: interpretation, uncertainty, plausibility
- ▶ development: model design, physics coupling, DA strategy

AI becomes powerful when grounded in science.

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00	Joint Dinner				

Lecture 1: Python Setup and Basics

Goal of Lecture

- ▶ Setup Python
- ▶ Synchronize with your knowledge of programming
- ▶ Become aware of the importance of proper package management
- ▶ Numpy and Matplotlib
- ▶ Get simple plots working
- ▶ write and import functions
- ▶ Flow Control
- ▶ File management
- ▶ Dictionaries
- ▶ Json Data Handling
- ▶ Classes and Dynamics

JavaScript Object Notation



Python Setup: First Check and Create Virtual Environment

Goal of this step

- ▶ Verify that Python is installed
- ▶ Work across Linux, macOS, Windows, WSL
- ▶ Create isolated environments: aipy
- ▶ No dependency conflicts

Basic Installation

```
1 python --version.    # check version
2 python3 --version.   # check version
3 python3.12 --version
4 cd                  # go to home
5 python -m venv aipy # create venv
6 alias aipy="source ~/aipy/bin/activate"
7 aipy                # activate aipy
8 python              # interactive python
```

Linux/macOS example

Python Package Stack: How Things Build on Each Other

Core idea

- ▶ Python is a layered ecosystem
- ▶ Packages build on top of each other
- ▶ Higher layers assume lower layers exist
- ▶ This structure matters for:
 - ▶ installation
 - ▶ debugging
 - ▶ performance

Conceptual Package Stack (ASCII)

```
1 Python
2 |
3   +-- NumPy          (arrays, numerics)
4   |
5   +-- Matplotlib     (plots, figures)
6   |
7   +-- AI / ML
8     |
9       +-- PyTorch
10      +-- scikit-learn
```

Conceptual view, not an exact dependency graph

Installing Core Scientific Packages

What we need first

- ▶ NumPy for numerical arrays
- ▶ Matplotlib for visualization
- ▶ Foundation for most scientific libraries
- ▶ Same tools across weather, climate, AI

Install basic packages

```
1 # activate virtual environment
2 aipy
3
4 # install core scientific packages
5 pip install numpy
6 pip install matplotlib
7
8 # verify installation
9 python -c "import numpy, matplotlib; print
('OK')"
```

These packages will be reused throughout the course

Installing Python Packages with pip

Why package management matters

- ▶ Python itself is minimal
- ▶ Most functionality comes from packages
- ▶ Packages define your working environment
- ▶ Reproducibility depends on exact versions

Key idea:

- ▶ One project ⇒ one environment

Basic pip workflow

```
1 # activate aipy, list installed
2 aipy
3 pip list
4
5 # install core packages
6 pip install numpy
7 pip install matplotlib
8
9 # verify installation
10 python -c "import numpy, matplotlib;
               print('ok')"
```

Always install packages inside an activated virtual environment

NumPy Arrays: 1D, 2D, and 3D Data - Standard Programming Skills

Core idea

- ▶ One data structure for science
- ▶ Same logic for 1D, 2D, 3D data
- ▶ Used for time series, maps, fields
- ▶ Basis for ML tensors
- ▶ Learn the basics
yourself, its easy.

NumPy array dimensions

```
1 import numpy as np
2
3 x = np.array([1, 2, 4]) # 1D
4 A = np.array([[1, 2], [3, 4]]) # 2D
5 B = np.zeros((10, 50, 100)) # 3D
6
7 print(x.shape, A.shape, B.shape)
8 # prints (3,) (2, 2) (10, 50, 100)
9
10 print(x)
11 # prints [1 2 4]
```

Dimensions encode structure, not meaning

Vectors, Matrices, and Broadcasting

Core concepts

- ▶ Vectors (1D)
- ▶ Matrices (2D)
- ▶ Matrix–vector product
- ▶ Matrix–matrix product
- ▶ Broadcasting

Carry out MATLAB like
Linear Algebra

Loops only Top Level!

Linear algebra in NumPy

```
1 import numpy as np
2
3 x = np.array([1., 2.])
4 A = np.array([[1., 2.], [3., 4.]])
5
6 b = A @ x          # matrix-vector
7 B = A @ A          # matrix-matrix
8 C = A + 1.0         # broadcasting
9
10 print(f "b={b},\nB={B},\nC={C}")
```

Operations follow array shapes

Indexing, Slicing, and Masks - Think Efficiency!

Why this matters

- ▶ Access elements efficiently
- ▶ Select subdomains in space or time
- ▶ Apply conditions to data
- ▶ Basis of filtering and diagnostics

Make sure you use an
efficient coding approach!

Selecting array data

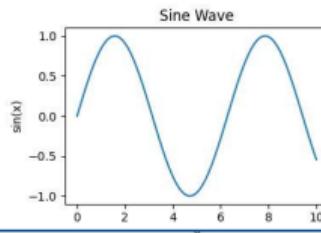
```
1 import numpy as np
2
3 A = np.array([[1., 2.], [3., 4.]])
4
5 B = A[0, 1]                      # single element
6 C = A[:, 0]                       # first column
7 D = A>2                          # boolean mask
8 E = A[A > 2]                     # select elements
9
10 print("A=", A, "\nB=", B, "\nC=", C, "\nD=", D, "\nE=", E)
```

Masks are vectorized and fast

Visualization Engines: A Sine Wave Example

What happens here

- ▶ Generate numerical data with **NumPy**
- ▶ Compute a math function
- ▶ **Visualize** data using Matplotlib
- ▶ **Save** plots for later use



plot-sine-wave.py

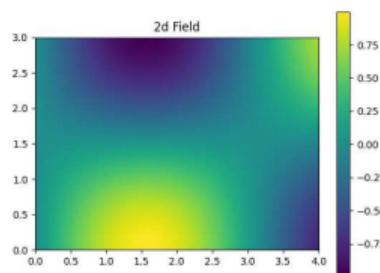
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0, 10, 100)
5 y = np.sin(x)
6
7 plt.figure(figsize=(4,3))
8 plt.plot(x, y)
9 plt.xlabel("x"); plt.ylabel("sin(x)")
10 plt.title('Sine Wave'); plt.tight_layout()
11 plt.savefig("plot-sine-wave.png")
12 plt.close()
```

A minimal but complete scientific plotting workflow

Visualizing 2D Data with Matplotlib

Key idea

- ▶ 2D arrays represent spatial fields
- ▶ Colors encode values
- ▶ Typical for weather and climate data



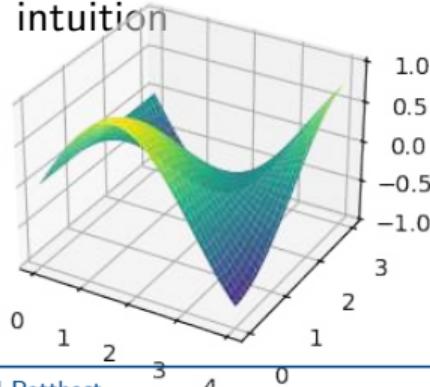
Simple 2D field

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0, 4, 120)
5 y = np.linspace(0, 3, 90)
6 X, Y = np.meshgrid(x, y)
7 Z = np.sin(X) * np.cos(Y)
8
9 plt.imshow(Z, origin="lower",
10             extent=[0,4,0,3], cmap="viridis")
11 plt.colorbar(); plt.title("2d Field");
12 plt.savefig("plot-2d-field.png");
13 plt.close()
```

3D Data: Fields and Surfaces

Why 3D matters

- ▶ Many geophysical fields are spatial
- ▶ Height, depth, or phase space
- ▶ Visualization helps intuition



plot-3d.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0, 4, 80)
5 y = np.linspace(0, 3, 60)
6 X, Y = np.meshgrid(x, y)
7 Z = np.sin(X) * np.cos(Y)
8
9 fig = plt.figure(figsize=(4,3))
10 ax = fig.add_subplot(projection="3d")
11 ax.plot_surface(X, Y, Z, cmap="viridis")
12 plt.savefig("plot-3d.png"); plt.close()
```

Example of a smooth 3D field defined on a 2D grid

Functions: Turn Ideas into Reusable Tools

Why functions matter

- ▶ Turn a workflow step into a **reusable tool**
- ▶ Make intent explicit: inputs → output
- ▶ Easier testing, debugging, and collaboration
- ▶ Foundation for **pipelines** and **ML training loops**

Mini tool: normalize a signal

```
1 import numpy as np
2
3 def normalize(x):
4     x = np.asarray(x)
5     return (x - x.mean()) / x.std()
6
7 x = np.array([2., 3., 5., 9.])
8 print("x =", x)
9 print("z =", normalize(x))
```

A pattern you will reuse everywhere: prepare data in one function

Control Flow: Quality Checks and Simple Automation

Control flow in practice

- ▶ Decision making:
reject bad inputs early
- ▶ Works on full arrays, makes it fast
- ▶ Clearly essential for robust scripts and experiments

Filter missing values

```
1 import numpy as np
2 x = np.array([1.2, 2.0, 1e30, 4.9])
3 thr = 1e20
4 # removing values outside of range:
5 x2 = x[x < thr]; dn = len(x)-len(x2)
6
7 fmt = "{:10.3g}"    # width=10, 3 digits
8 print("".join(fmt.format(v) for v in x))
9 print("".join(fmt.format(v) for v in x2))
10 print("mean=", f"{x2.mean():.3f}")
```

Quality control on observations or model fields

Dictionaries and JSON: Configs You Can Share

Why this is exciting

- ▶ Store model and experiment settings cleanly
- ▶ Pass configs through APIs and workflows
- ▶ **Reproducibility**: one file describes the run

Config for an experiment

```
1 import json
2
3 cfg = {"dt": 0.5, "n": 20, "model": "toy"}
4 s = json.dumps(cfg, indent=2)
5 print(s)
6
7 cfg2 = json.loads(s)
8 print("dt =", cfg2["dt"])
```

This is the bridge to automation + deployment later

Loops in Python: Time Stepping and Accumulation

Typical use case

- ▶ Discrete time stepping
- ▶ Running sums and averages
- ▶ Diagnostics over trajectories
- ▶ Core pattern in models

Mental model

- ▶ Each loop = one time step
- ▶ State evolves sequentially

time_loop.py

```
1 import numpy as np
2
3 dt = 0.1; x = 0.0; traj = [] #  
    initialization
4
5 for n in range(20):
6     x = x + dt * np.sin(x)
7     traj.append(x)
8
9 traj = np.array(traj)
10 print("final x =", x)
11 print("mean x  =", traj.mean())
```

Sequential updates cannot be vectorized away

Files: Log Results, help yourself, Scientist!

File I/O patterns

- ▶ Save key results after every run
- ▶ Keep a simple experiment log
- ▶ The with pattern avoids subtle bugs

Write a tiny log file

```
1 from datetime import datetime
2
3 myerr = 0.23 # example
4 msg = f"{datetime.now()} rmse={myerr}\n"
5 with open("log-run.log", "a") as f:
6     f.write(msg) # a is for append
7
8 with open("log-run.log") as f:
9     last_line = f.readlines()[-1]
10    print(last_line.strip())
```

Minimal logging already gives you insight and transparency

Classes I: Encapsulate State (A Tiny Model Component)

Why classes appear everywhere

- ▶ Components have a state (e.g., temperature)
- ▶ Methods update that state consistently
- ▶ Same structure in NWP, ESMs, and ML modules

A relaxing temperature

```
1 class RelaxTemp:  
2     def __init__(self, T):  
3         self.T = T  
4     def step(self, dt, target):  
5         self.T += 0.1*dt*(target - self.T)  
6  
7 atm = RelaxTemp(288.0)  
8 atm.step(1.0, 290.0)  
9 print("T =", atm.T)
```

A tiny but real modeling pattern: relaxation toward forcing

Classes II: Coupling Two Components (Mini Earth-System Pattern)

The exciting part

- ▶ Two components, each with its own state
- ▶ A controller coordinates information exchange
- ▶ This is the conceptual core of coupled models

Coupling in 10 lines

```
1 class Box:  
2     def __init__(s,x): s.x=x  
3     def step(s,dt,t): s.x+=0.2*dt*(t-s.x)  
4  
5 atm, ocn = Box(288.), Box(290.)  
6 ocn.step(1., atm.x); atm.step(1., ocn.x)  
7 print("atm=", atm.x, "ocn=", ocn.x)
```

Same coupling idea, later: many variables + grids + physics

Lecture 1 — Key Takeaways

Technical Foundations

- ▶ Python as a portable **workhorse** for science
- ▶ **Virtual environments** avoid dependency conflicts
- ▶ pip and requirements.txt ensure reproducibility
- ▶ NumPy arrays as the **core data structure**
- ▶ Vectorization replaces explicit loops
- ▶ Matplotlib for fast diagnostic visualization

Conceptual Lessons

- ▶ Think in terms of **arrays**, not scalars
- ▶ Data selection via slicing and masking
- ▶ Broadcasting enables compact math expressions
- ▶ Visualization supports scientific intuition
- ▶ Clean code beats clever code
- ▶ Python skills will transfer directly to AI/ML development

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Lecture 2: Jupyter Notebooks, APIs and Servers

Goal of this Lecture

- ▶ Work productively with Jupyter Notebooks
- ▶ Understand Notebooks as part of a scientific workflow
- ▶ Prepare the ground for reproducible ML experiments

Focus

- ▶ Not UI details, but how things fit together
- ▶ From exploration to engineering

Topics Overview

- ▶ Jupyter Notebooks: Kernel, Server, Browser
- ▶ Environments and package management
- ▶ Markdown, magic commands, shell integration
- ▶ Data visualization as quality control
- ▶ APIs as a structuring principle
- ▶ Local, library and web APIs
- ▶ Native code integration (Fortran / C++)

Jupyter Notebook

- ▶ Cells with comments
- ▶ Cells with code
- ▶ Cells with output

jupyter Jupyter_Demo1 Last Checkpoint: 20 seconds ago

File Edit View Run Kernel Settings Help Trusted

I + X C JupyterLab ↗ ⌘ (Copy) C

```
[1]: import numpy as np, matplotlib.pyplot as plt
from matplotlib.colors import LightSource

u = np.linspace(0, 2*np.pi, 320); v = np.linspace(-0.6, 0.6, 80); U, V = np.meshgrid(u, v)

# Möbius-like twisted surface with extra complexity
X = (1 + V*np.cos(U/2)) * np.cos(U); Y = (1 + V*np.cos(U/2)) * np.sin(U)
Z = V*np.sin(U/2) + 0.3*np.sin(3*U)

ls = LightSource(azdeg=22, altdeg=65); rgb = ls.shade(Z, cmap=plt.cm.viridis, vert_exag=0.9)
plt.figure(figsize=(12,12)); ax = plt.axes(projection="3d")
ax.plot_surface(X, Y, Z, facecolors=rgb, linewidth=0); ax.set_axis_off(); plt.show()
```

Why Jupyter Notebooks in Science and ML?

Why Notebooks Matter

- ▶ Rapid exploration of data and ideas
- ▶ Immediate feedback via plots and diagnostics
- ▶ Combine code, results and explanation

In Research Contexts

- ▶ Hypothesis testing and prototyping
- ▶ Understanding data and model behavior
- ▶ Bridging theory and implementation

Why They Scale Beyond Prototyping

- ▶ Documentation of decisions and assumptions
- ▶ Reproducible experiments (when done right)
- ▶ Natural interface to libraries and APIs
- ▶ Gateway to larger workflows and services

Key Message

- ▶ Notebooks are tools, not products
- ▶ Value comes from disciplined usage

Jupyter Architecture: Browser – Server – Kernel

Three Core Components

- ▶ **Browser** User interface: notebooks, plots, interaction
- ▶ **Jupyter Server** Manages files, sessions, security
- ▶ **Kernel** Executes Python code, holds state

Key Idea

- ▶ UI and computation are **decoupled**

Why This Matters

- ▶ Code runs in the **kernel**, not in the browser
- ▶ Kernel state persists across cells
- ▶ Server and kernel may run **remotely**
- ▶ Multiple notebooks can share one kernel

Strength and Risk

- ▶ Powerful interactive workflow
- ▶ Risk of hidden state and irreproducibility

Reproducibility in Jupyter Notebooks

Why Reproducibility Matters

- ▶ Results must be repeatable
- ▶ Experiments must be explainable
- ▶ Others (and you later) must trust them

Typical Problems

- ▶ Hidden kernel state
- ▶ Cells executed out of order
- ▶ Undocumented dependencies

Four Simple Rules

- ▶ Restart kernel and **Run All**
- ▶ Clear, explicit import and parameter cells
- ▶ Save outputs (plots, files, artefacts)
- ▶ Document dependencies and environment

Key Message

- ▶ A notebook is an **executable document**

Installing Packages in Jupyter Notebooks

Recommended Workflow

- ▶ Install packages either outside in the right virtual environment or directly in the notebook
- ▶ Use pip install or !pip install
- ▶ Packages are installed into the running kernel

Why This Works

- ▶ Jupyter uses the kernel's Python environment
- ▶ pip is typically bound to this Python

Good Practice: Check Once

Check active environment

```
1 import sys
2 print(f"Python executable: {sys.executable}")
```

Key Message

- ▶ Use pip where you run your code
- ▶ Verify the environment if something looks wrong

Markdown and Narrative Computing

Why Markdown Matters

- ▶ Makes notebooks **readable**
- ▶ **Explains** intent, not just results
- ▶ Turns experiments into documents

Narrative Computing

- ▶ Code, text and results in one place
- ▶ **Reasoning** becomes explicit
- ▶ Supports review and reuse

Minimum You Should Use

- ▶ Headings (#, ##)
- ▶ Bullet lists (-)
- ▶ Inline formulas (x^2)
- ▶ Short explanations

Markdown Cells

- ▶ Change cell type: Esc → M

Key Message

- ▶ A notebook should read like a **lab notebook**

Magic Commands and Shell Access

Why Magic Commands

- ▶ Speed up interactive work
- ▶ Reduce boilerplate code
- ▶ Support exploration and debugging

Line vs Cell Magics

- ▶ % for single-line commands
- ▶ %% for whole cells

The Few You Really Need

- ▶ %timeit (runtime)
- ▶ %%writefile (save code)
- ▶ %%bash (run shell)
- ▶ !pip (install packages)
- ▶ !ls (files)

Key Message

- ▶ Use magic and shell commands
sparingly

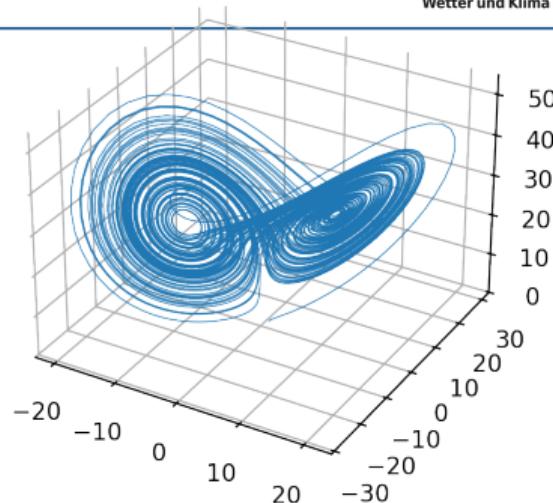
Visualization as Quality Control

Why Visualization Matters

- ▶ See patterns and anomalies
- ▶ Detect bugs early
- ▶ Build intuition about data and models

Typical Questions

- ▶ Does this look reasonable?
- ▶ Are scales and units correct?
- ▶ Are outliers expected?



Static vs Interactive

- ▶ Static: documentation, papers
- ▶ Interactive: exploration, debugging

Key Message

- ▶ Plot early, plot often!

Matplotlib

- ▶ Low-level, explicit control
- ▶ Publication-ready figures
- ▶ Default for scientific Python

Matplotlib (static)

```
1 plt.plot(x, y)
2 plt.xlabel("x");
3 plt.ylabel("y")
4 plt.savefig("plot.png")
```

When to use

- ▶ Final figures
- ▶ Full control over layout

Seaborn

- ▶ Built on Matplotlib
- ▶ Statistical defaults
- ▶ Fast insight into distributions

Seaborn (statistical)

```
1 sns.kdeplot(x=data)
```

Plotly

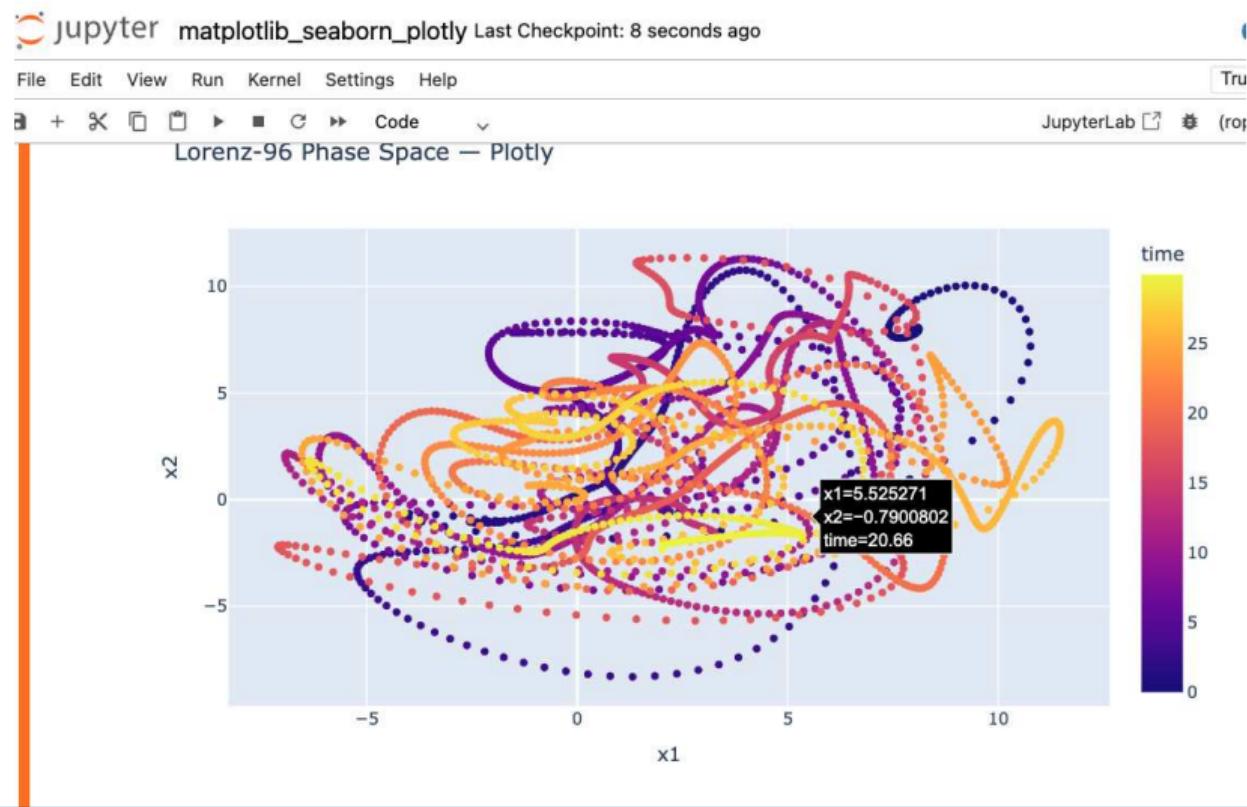
- ▶ Interactive plots
- ▶ Zoom, hover, selection

Plotly (interactive)

```
1 fig = px.scatter(df, x="x",
2                   y="y"); fig.show()
```

Interactive Plot via Plotly

- ▶ Explore your data
- ▶ Learn, check!
- ▶



What Is an API — and What Is Not

What an API Is

- ▶ A **defined interface** to call functionality
- ▶ Clear inputs, outputs and behavior
- ▶ Stable contract between components

Typical API Examples

- ▶ Python functions and classes
- ▶ Library interfaces (e.g. NumPy)
- ▶ Web endpoints (HTTP/REST)

What an API Is *Not*

- ▶ Just installable code
- ▶ A script without defined entry points
- ▶ A collection of loosely coupled functions

Important Distinction

- ▶ Installation is **distribution**
- ▶ An API is **how you call the code**

Key Message

- ▶ Packages may **expose** APIs
- ▶ Installation alone does not define one

APIs as a Scaling Principle

Core Idea

- ▶ APIs defines a contract on how functionality is accessed
- ▶ Clear separation of interface and implementation
- ▶ Same principle from notebooks to services

Why This Matters

- ▶ Logic becomes reusable and testable
- ▶ Notebooks stay thin and readable
- ▶ Systems can grow without rewrites

Example 1: Local API (project code)

Function API usage

```
1 from model import forecast  
2 y = forecast(x, params)
```

Example 2: Web API (service)

HTTP request

```
1 response = requests.get(  
2     "/weather",  
3     params={"city": "Berlin", "  
        date": "2025-01-20"})  
4 data = response.json()
```

Key Insight

- ▶ APIs shift complexity behind a stable boundary

REST Essentials — Tasks as Resources / Agent Framework Elements

Resources

- ▶ A task is a **resource**
- ▶ Identified by a URL
- ▶ Has a state and metadata
- ▶ `/tasks`
- ▶ `/tasks/<id>`

HTTP Methods

- ▶ POST — create a task
- ▶ GET — inspect task or list
- ▶ PUT — change task state
- ▶ DELETE — remove a task

Status Codes (Minimum)

- ▶ 200 OK
- ▶ 201 Created
- ▶ 400 Bad Request
- ▶ 404 Not Found

Task Server Setup — Minimal Flask Example

Goal

- ▶ Tasks as REST resources
- ▶ Explicit task state
- ▶ Minimal server logic

States

- ▶ created
- ▶ checked
- ▶ executing
- ▶ completed / failed

task_server.py

```
1 from flask import Flask, request, jsonify
2 app = Flask(__name__)
3 tasks = {}; i = 1
4
5 @app.post("/tasks")
6 def create():
7     global i
8     if not request.json: abort(400)
9     t = {"id": i, "state": "created", "data":
10         ": request.json}
11     tasks[i] = t; i += 1
12     return jsonify(t), 201
13 app.run()
```

Testing the Task REST API

What We Test

- ▶ Send JSON to the server (POST /tasks)
- ▶ Server creates a new task
- ▶ Server assigns **ID** and initial **state**

Expected Result

- ▶ HTTP status 201 Created
- ▶ JSON response with:
 - ▶ task id
 - ▶ state = created
 - ▶ echoed input data

Test via curl

```
1 curl -X POST http://127.0.0.1:5000/
  tasks -H "Content-Type:
  application/json" -d '{"type":'
  "demo", "params": {"x": 1}}'
```

Minimal Python Task Creation

```
1 import requests
2 r = requests.post(
3     "http://127.0.0.1:5000/tasks",
4     json={"type": "demo", "params": {"x":
5         : 1}})
6 print(r.status_code); print(r.json())
```

```
{'id': 1, 'state': 'created',
'data': {'type': 'demo',
'params': {'x': 1}}}
```

Task API — Server Endpoints

Purpose

- ▶ Expose tasks via REST
- ▶ Read-only inspection
- ▶ Server owns all state

Server Resource Endpoints

- ▶ `/tasks` — task collection
- ▶ `/tasks/<id>` — single task

Semantics

- ▶ Stateless client
- ▶ Explicit URLs
- ▶ JSON responses

```
task_endpoints.py
```

```
1 from flask import Flask, jsonify,
2     abort
3 app = Flask(__name__)
4 tasks = {}
5
6 @app.get("/tasks")
7 def list_tasks():
8     return jsonify(list(tasks.
9         values()))
10
11 @app.get("/tasks/<int:i>")
12 def get_task(i):
13     return jsonify(tasks[i]) if i
14         in tasks else abort(404)
15
16 app.run()
```

Querying Tasks — Inspecting Server State

Goal

- ▶ Inspect existing tasks
- ▶ Read state and metadata
- ▶ No client-side state

REST Principle

- ▶ Tasks are resources
- ▶ Identified by URL
- ▶ Read via GET

Endpoints

- ▶ GET /tasks
- ▶ GET /tasks/<id>

List all Tasks

```
1 import requests
2 r = requests.get("http
                  ://127.0.0.1:5000/tasks")
3 print(r.status_code)
4 print(r.json())
```

Describe one task

```
1 import sys, requests
2 tid = int(sys.argv[1])
3 r = requests.get(f"http
                  ://127.0.0.1:5000/tasks/{tid}")
4 print(r.status_code)
5 print(r.json())
```

Native Code Integration: Fortran and C++ in Python & ML

Why Native Code Matters

- ▶ Decades of validated Fortran in NWP
- ▶ High-performance kernels in C++
- ▶ Tight control over memory and execution
- ▶ Reuse of trusted implementations

Typical Use Cases

- ▶ Physical parameterizations
- ▶ Linear operators, solvers, kernels
- ▶ Observation operators
- ▶ Legacy model components

Python as the Orchestration Layer

- ▶ Python controls the workflow
- ▶ Native code provides compute kernels
- ▶ Clean separation via APIs

Integration Options (Overview)

- ▶ ctypes – explicit C-compatible interfaces
- ▶ f2py – automatic Fortran bindings
- ▶ pybind11 – modern C++ bindings
- ▶ Shared libraries: .so / .dylib

Fortran with C Bindings: A Stable Interface

Why C Bindings?

- ▶ Fortran and Python do **not** talk directly
- ▶ The common denominator is the **C ABI**
- ▶ Stable, explicit, language-independent

Key Concept

- ▶ Fortran exposes functions as **C-compatible symbols**
- ▶ No name mangling
- ▶ Well-defined data types

Minimal Fortran Example

```
1 function f_sin_cos(x) result(f) bind(C)
2   use iso_c_binding
3   real(c_double), intent(in) :: x
4   real(c_double) :: f
5   f = sin(x) * cos(x)
6 end function
```

What This Ensures

- ▶ Symbol name is predictable
- ▶ Argument layout follows C rules
- ▶ Callable from Python, C, C++

Calling Fortran from Python with ctypes

Role of Python

- ▶ Python loads the shared library
- ▶ Defines the function signature
- ▶ Manages data exchange

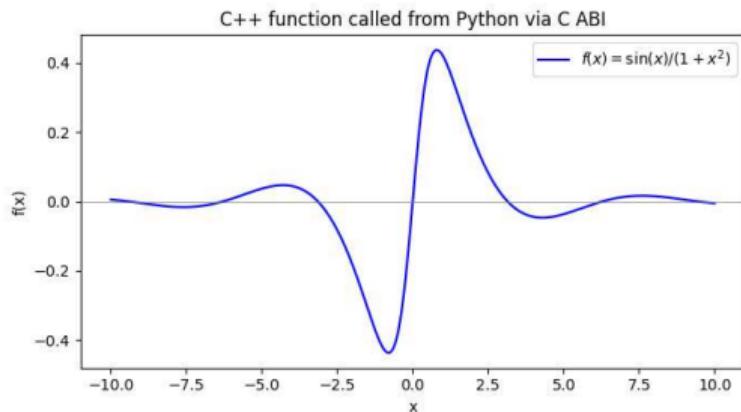
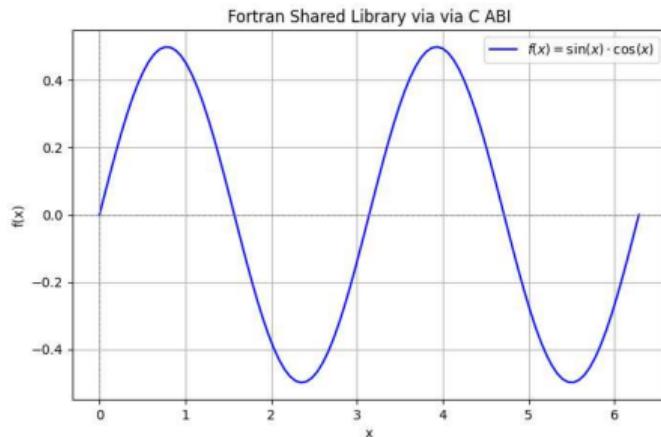
Important Detail

- ▶ Fortran arguments are passed by reference
- ▶ Python must pass pointers
- ▶ Native Fortran computation
- ▶ Controlled Python interface
- ▶ No performance-critical Python loop

Minimal Python Interface

```
1 import ctypes
2
3 lib = ctypes.CDLL("./fortran_interface.so")
4 lib.f_sin_cos.argtypes = [
5     ctypes.POINTER(ctypes.c_double) ]
6 lib.f_sin_cos.restype = ctypes.c_double
7
8 x = ctypes.c_double(1.0)
9 y = lib.f_sin_cos(ctypes.byref(x))
```

Native Code Integration — Fortran vs. C++ from Python



Fortran via ISO_C_BINDING + ctypes
Numerics compiled, called from Python

C++ via C-compatible ABI + ctypes
Explicit interface, manual symbol control

- ▶ Python orchestrates, native code computes
- ▶ Identical workflow: compile → load → call → plot

Exposing a C++ Function via the C ABI

Role of C++

- ▶ Implements numerical logic
- ▶ Compiled into a shared library
- ▶ Exposes a stable C ABI

Key Requirement

- ▶ Use `extern "C"`
- ▶ Avoid C++ name mangling

Minimal C++ Interface

```
1 #include <cmath>
2
3 extern "C" double f_cpp(double x)
4 {
5     return std::sin(x) / (1.0 + x*x);
6 }
```

- ▶ Plain C-compatible symbol
- ▶ No templates, no classes
- ▶ ABI-safe function signature

Calling C++ from Python with ctypes

Role of Python

- ▶ Python loads the shared library
- ▶ Defines the **binary interface**
- ▶ Controls execution and visualization

Important Detail

- ▶ C++ function uses **C ABI**
- ▶ Arguments passed **by value**
- ▶ Native C++ computation
- ▶ Explicit ABI contract
- ▶ Minimal Python overhead

Minimal Python Interface

```
1 import ctypes
2
3 lib = ctypes.CDLL("./cpp_interface.so")
4 lib.f_cpp.argtypes = [ctypes.c_double]
5 lib.f_cpp.restype = ctypes.c_double
6
7 x = 1.0
8 y = lib.f_cpp(x)
```

Remote Jupyter — Principle

Core Idea

- ▶ Compute: **remote** (server, HPC, cloud)
- ▶ Visualization: **local** in the browser
- ▶ Browser setup remains unchanged

Typical Scenarios

- ▶ HPC login or compute nodes
- ▶ Cloud virtual machines
- ▶ Office workstation / bastion host

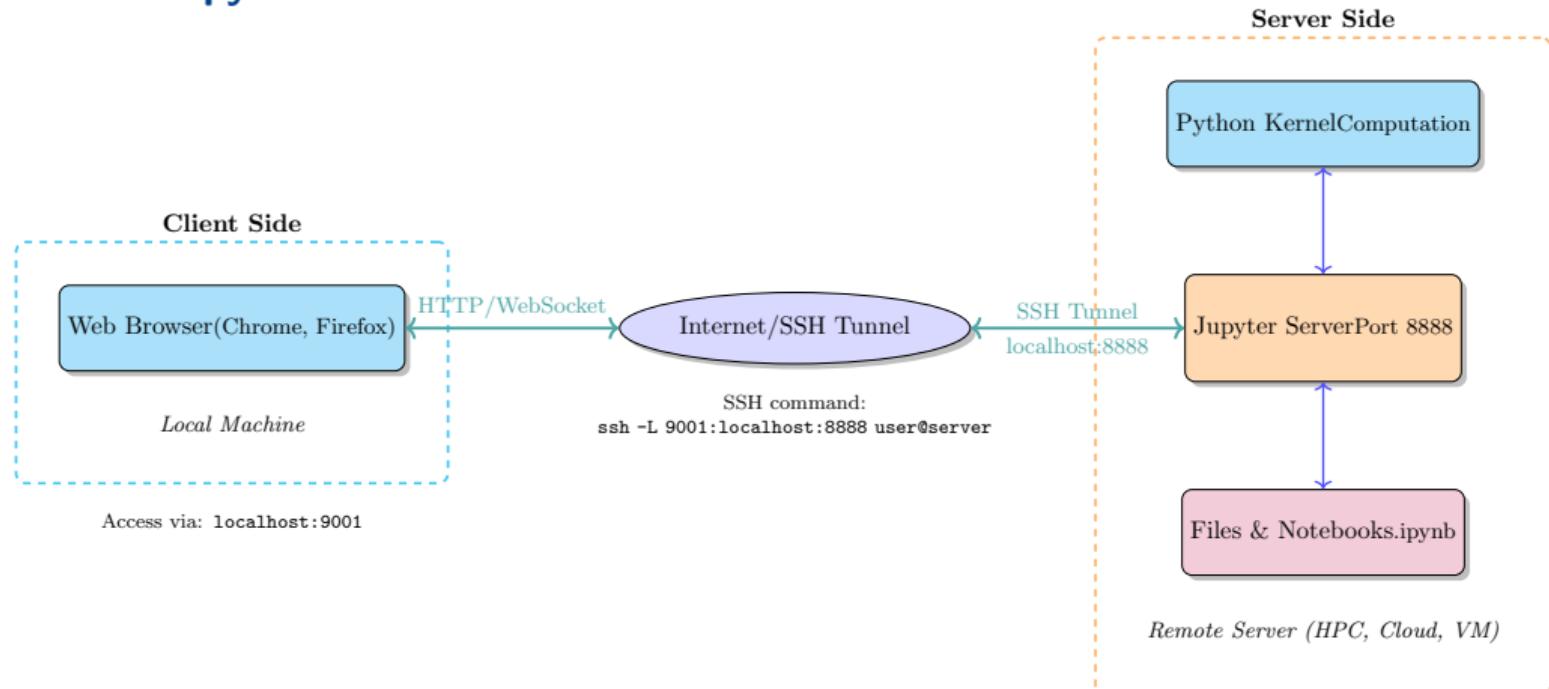
Security Principle

- ▶ **No** open notebook ports to the network
- ▶ Never expose :8888 on a public IP
- ▶ Access exclusively via **SSH tunnels**

Key Takeaway

- ▶ Jupyter only listens **locally**
- ▶ SSH provides secure transport

Remote Jupyter — Network Architecture



Remote Jupyter — Port Forwarding Recipe

Step 1: Start Jupyter Remotely

Remote (Linux)

```
1 jupyter notebook \
2   --no-browser \
3   --port=8888
```

Step 2: Create SSH Tunnel

Local (Bash / PowerShell)

```
1 ssh -N -L 9001:localhost:8888 \
2       user@remote-host
```

Step 3: Open in Browser

- ▶ Access token appears in the terminal
- ▶ Port is **local only** on the remote server

- ▶ <http://localhost:9001>
- ▶ Use token from the remote log
- ▶ **Adjust port** if needed

To access the server, open this file in a browser:

<file:///hpc/uhome/rpotthas/.local/share/jupyter/runtime/jpserver-1662241-open.html>

Or copy and paste one of these URLs:

<http://localhost:8888/tree?token=369b469c83e8eaa369ee02eaa443d994865ca93e4185bb385>

Remote Jupyter — Firewalls & Troubleshooting

Typical Problems

- ▶ Local port already in use
- ▶ Remote port 8888 blocked by firewall
- ▶ Unstable SSH connection
- ▶ Multi-hop access (Jump / Bastion host)

Important Note

- ▶ A blocked 8888 is **not an error**
- ▶ SSH tunnels do not require open inbound ports

Best Practices

- ▶ Remote:
 - ▶ -ip=127.0.0.1
 - ▶ no public binding
- ▶ Local:
 - ▶ choose a free port (9001, 9002, ...)
- ▶ Infrastructure:
 - ▶ Bastion / jump host via -J

Recommendation

- ▶ **Never** expose Jupyter notebook ports publicly

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!



Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Working with Meteorological Data for AI and ML

Context

- ▶ Python-based workflows
- ▶ Meteorological fields for AI and machine learning
- ▶ Real operational datasets

Why this lecture?

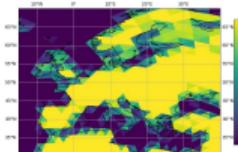
- ▶ ML needs structured input data
- ▶ Models operate on fields, not files
- ▶ Visualization builds physical intuition

Core skills we build

- ▶ Access meteorological fields
- ▶ Understand grids and metadata
- ▶ Convert and preprocess data
- ▶ Visualize fields and observations

Outcome

- ▶ Data ready for ML pipelines
- ▶ Reproducible Python workflows



Installing and Exploring ecCodes

What is ecCodes?

- ▶ ECMWF library for GRIB handling
- ▶ Standard tool in operational NWP
- ▶ Python interface available

Why we need it

- ▶ Forecast data comes as GRIB
- ▶ We need access to fields and metadata
- ▶ Independent of model (IFS, ICON, ...)

Installation and test

```
1 # Linux / WSL
2 sudo apt update
3 sudo apt install eccodes libeccodes-
   tools
4
5 # macOS (Homebrew)
6 brew install eccodes
7
8 # Python bindings
9 pip install eccodes
10
11 # Test installation
12 grib_ls -V
13 python -c "import eccodes; print(
   eccodes.codes_get_api_version())"
```

Using ecCodes

- ▶ See what is inside a GRIB file
- ▶ Identify fields
- ▶ Get size and structure

Key idea

- ▶ A GRIB file contains multiple messages
- ▶ Each message holds one field
- ▶ ecCodes iterates message by message

Inspecting a GRIB file

```
1 import eccodes
2
3 f = open("icon_eu_t2m_latest.grib2", "rb")
4 while True:
5     gid = eccodes.codes_grib_new_from_file(f)
6     if gid is None:
7         break
8
9     short = eccodes.codes_get(gid, "shortName")
10    level = eccodes.codes_get(gid, "level")
11    size = eccodes.codes_get_size(gid, "values")
12    print(short, level, size)
13    eccodes.codes_release(gid)
14 f.close()
```

IFS Open Data — Access via Client Library

- ▶ IFS data is not exposed as simple directories
- ▶ Access is service-based and load-controlled
- ▶ Small subsets can be retrieved efficiently

Key idea

- ▶ Use ECMWF's official open-data client
- ▶ Request fields by parameter, step, level
- ▶ Result is a standard GRIB2 file

Downloading IFS fields with ecmwf-opendata

```
1 from ecmwf.opendata import Client
2
3 client = Client(
4     source="ecmwf",
5     model="ifs",
6 )
7
8 client.retrieve(
9     time=0,
10    type="fc",
11    step=24,
12    param=["2t", "msl"],
13    target="ifs_2t.grib2")
```

Plotting an IFS Field (Regular Grid)

- ▶ IFS open data uses a regular lat–lon grid
- ▶ Field values form a 2-D NumPy array
- ▶ Plotting is straightforward

Important

- ▶ Grid size is fixed by the model
- ▶ figsize controls readability only

Plotting IFS 2 m temperature

```
1 import eccodes
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 f = open("ifs_2t.grib2", "rb")
6 gid = eccodes.codes_grib_new_from_file(f)
7
8 nx = eccodes.codes_get(gid, "Ni")
9 ny = eccodes.codes_get(gid, "Nj")
10 values = eccodes.codes_get_array(gid, "values")
11 field = values.reshape(ny, nx)
12
13 plt.figure(figsize=(7, 3.5))
14 plt.imshow(field, origin="lower")
```

Cartopy — Map Projections for Meteorological Fields

- ▶ Cartopy provides projection-aware plotting
- ▶ Separates data coordinates from map projection
- ▶ Built on top of matplotlib

In practice

- ▶ Define projection, e.g. PlateCarree
- ▶ Coastlines, borders, land masks
- ▶ Define geographic extent

Cartopy-based map plot

```
1 import cartopy.crs as ccrs
2 import cartopy.feature as cfeature
3
4 fig, ax = plt.subplots( figsize=(10,5),
5                         subplot_kw={"projection": ccrs.
6                           PlateCarree()})
7
8 ax.coastlines()
9 ax.add_feature(cfeature.BORDERS)
10
11 ax.pcolormesh(lon, lat, field,
12                 transform=ccrs.PlateCarree(),
13                 cmap="jet" )
```

ICON Open Data — File-Based Access

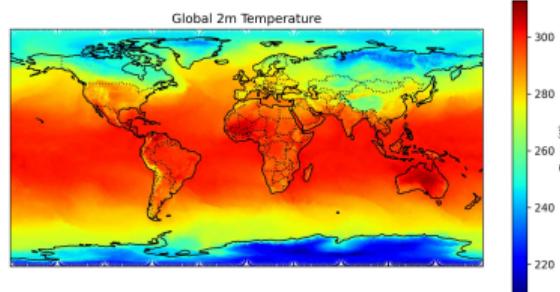
- ▶ ICON data is published as static GRIB files
- ▶ Organised by model, run, and parameter
- ▶ Distributed via DWD Open Data server

Typical workflow

- ▶ Query directory listing
- ▶ Identify latest available timestamp
- ▶ Download and decompress .grib2.bz2

Material provided

- ▶ Python script: icon_download_t2m.py
- ▶ Jupyter notebook: ICON download & inspection



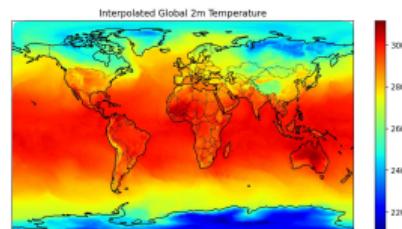
See code and notebooks for a robust, reusable implementation.

ICON Fields — Interpolation and Dateline Handling

- ▶ ICON uses an unstructured triangular grid
 - ▶ Direct plotting requires triangulation
 - ▶ Interpolation needed for:
 - ▶ smooth visualization
 - ▶ map projections
-
- ### Implementation
- ▶ See Python script:
`fix_dateline_triangles.py`
(adapted from ICON tutorial)

Key challenge

- ▶ Triangles crossing the date line
- ▶ Incorrect rendering in global projections
- ▶ Requires explicit geometric handling



Observational Data — Purpose and Characteristics

- ▶ Observations anchor models to reality
- ▶ Essential for:
 - ▶ verification
 - ▶ data assimilation
 - ▶ ML training and validation

Typical properties

- ▶ Irregular spatial distribution
- ▶ Sparse and heterogeneous
- ▶ Strong metadata dependence

Common observation types

- ▶ SYNOP (surface stations)
- ▶ TEMP / radiosondes
- ▶ AIREP / AMDAR (aircraft)
- ▶ Satellite observations

Key difference to models

- ▶ No grid
- ▶ Point-based measurements
- ▶ Representativeness matters

SYNOP Data — Reading Raw Observations

- ▶ SYNOP stored in NetCDF
- ▶ One record per station
- ▶ Variables include:
 - ▶ latitude, longitude
 - ▶ observed temperature

Minimal requirement

- ▶ Read coordinates
- ▶ Read physical quantity
- ▶ Preserve missing values

Python tools

- ▶ netCDF4.Dataset
- ▶ numpy

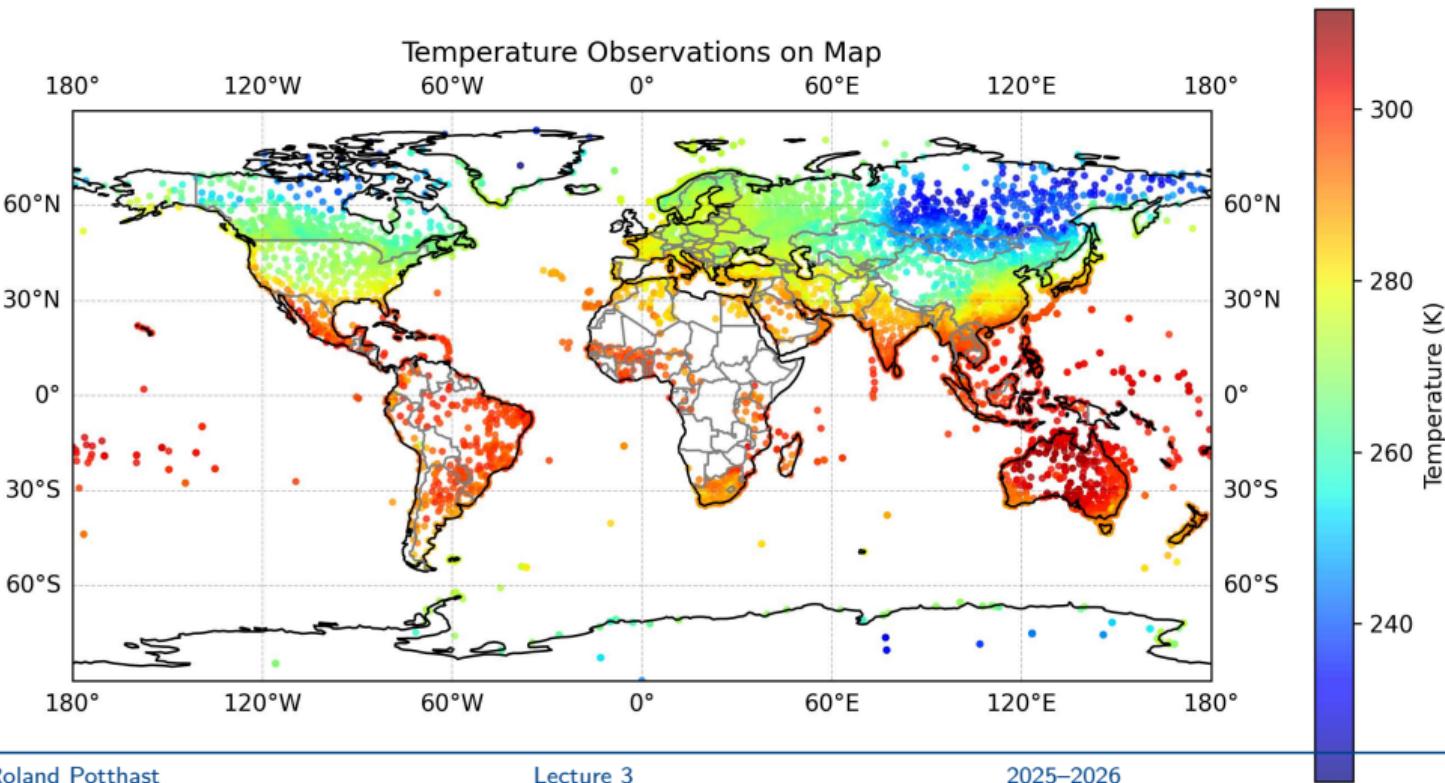
Variables used

- ▶ MLAH — latitude
- ▶ MLOH — longitude
- ▶ MTDBT — temperature

Implementation

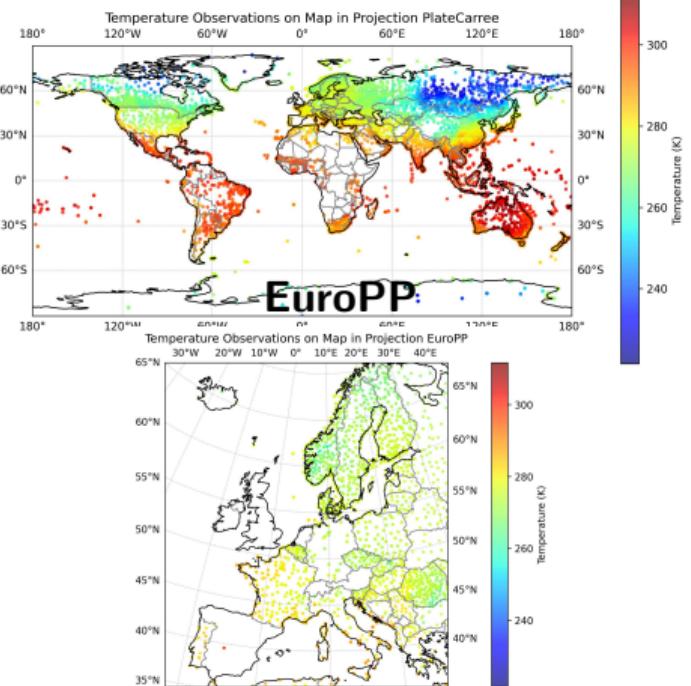
- ▶ See function: `read_synop_data()`

SYNOP Observations — Point-Based Visualization

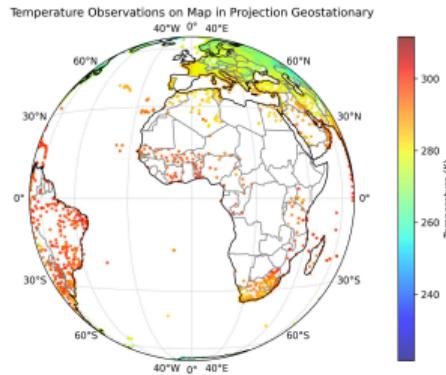


SYNOP Observations — Map Projections

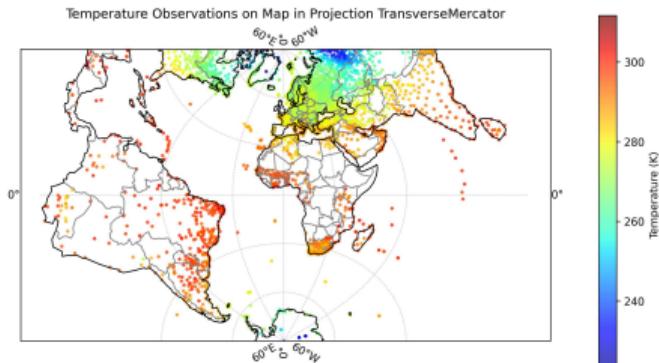
Plate Carree



Geostationary



Transverse Mercator



Feedback Files — Linking Models and Observations

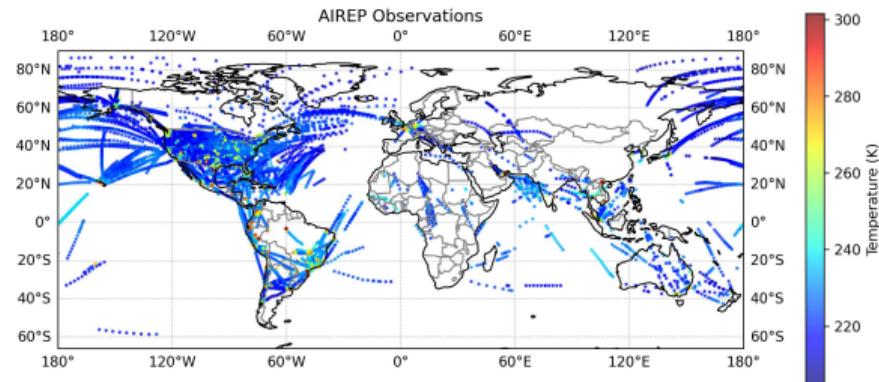
- ▶ Feedback files store model–observation pairs
- ▶ Generated during data assimilation
- ▶ Central diagnostic product

Contained information

- ▶ Observation value
- ▶ Model equivalent ($H(x)$)
- ▶ Quality control flags
- ▶ Metadata (time, location, type)

Why they matter

- ▶ Quantify model bias and error
- ▶ Basis for monitoring systems
- ▶ Ideal input for ML analysis



Feedback Files — Structure and Indexing

- ▶ Stored as structured NetCDF files
- ▶ Separate layers for:
 - ▶ report metadata
 - ▶ individual observations

Key idea

- ▶ One report may contain many observations
- ▶ Explicit indexing links both layers

Core variables (excerpt)

i_body	(nreport)	start index of report
l_body	(nreport)	number in report
obs	(nobs)	observation
veri_data	(nmodel,nobs)	model equivalent $H(x)$

Interpretation

- ▶ `i_body:l_body` maps reports → `obs`
- ▶ `obs` and `veri_data` are aligned
- ▶ Enables direct O-B, O-A diagnostics

AIREP Feedback — Statistical Diagnostics

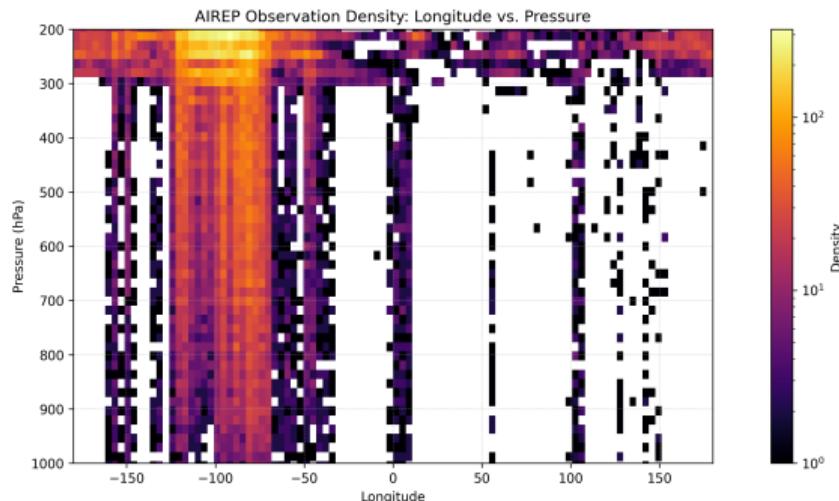
- ▶ AIREP / AMDAR provide aircraft observations
- ▶ Strong vertical and regional sampling biases
- ▶ Feedback files enable systematic evaluation

Typical diagnostics

- ▶ Observation density vs height
- ▶ Mean and spread of innovations (O–B)
- ▶ Model-dependent biases

Why this matters

- ▶ Identifies representativeness errors
- ▶ Reveals flow- and height-dependent biases

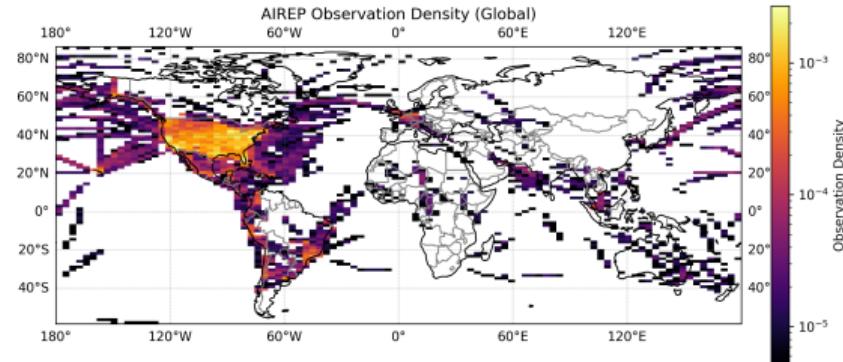


AIREP Feedback — Horizontal Sampling

- ▶ AIREP sampling follows air traffic routes
- ▶ Highly inhomogeneous horizontal coverage
- ▶ Strong land–sea and regional contrasts

Key implications

- ▶ Data density reflects logistics, not physics
- ▶ Large unsampled regions persist
- ▶ Bias risk if learning ignores sampling



Global horizontal density of AIREP observations

Operational relevance

- ▶ Affects verification statistics
- ▶ Affects DA weighting
- ▶ Critical for ML generalisation

Why GRIB? — Operational Requirements

- ▶ GRIB = GRIdded Binary
- ▶ Designed for operational NWP
- ▶ Optimised for:
 - ▶ large data volumes
 - ▶ fast I/O
 - ▶ network distribution

Core design goals

- ▶ Compact encoding
- ▶ Self-describing metadata
- ▶ Bit-level compression

Why it is still used

- ▶ Global models produce terabytes per day
- ▶ Multiple grids and vertical coordinates
- ▶ Strict real-time constraints

Operational reality

- ▶ WMO standard
- ▶ Tooling is mature and stable
- ▶ Backward compatibility matters

Alternatives to GRIB — Strengths and Limits

NetCDF / CF

- ▶ Widely used in research
- ▶ Human-readable metadata
- ▶ Excellent tool support

Zarr / Cloud-native

- ▶ Chunked, parallel access
- ▶ Designed for object storage
- ▶ Attractive for ML workflows

Why GRIB is not easily replaced

- ▶ Operational ecosystems are huge
- ▶ Encoding efficiency still unmatched
- ▶ Standards evolution is slow by design

Current practice

- ▶ GRIB for production and exchange
- ▶ NetCDF / Zarr for analysis and ML
- ▶ Conversion layers in between

Key message

- ▶ Format choice reflects use case

GPU Access in Practice — Why It Depends on the Platform

- ▶ Most laptops *do have a GPU*
- ▶ But GPU **type and backend** differ strongly
- ▶ Python code must match the backend

Common GPU types

- ▶ **NVIDIA** — typical for HPC clusters
- ▶ **Apple GPU** — Apple Silicon
- ▶ **AMD / Radeon** — some laptops, workstations

Key message

- ▶ The **software interface** matters

GPU backends used in Python

- ▶ CUDA — NVIDIA GPUs (Linux, HPC)
- ▶ MPS / Metal — Apple GPUs (macOS)
- ▶ CPU fallback — always available

Implication for ML workflows

- ▶ Development often on laptops
- ▶ Training often on HPC systems
- ▶ Code must be **portable across backends**

GPU Tests in Python — Minimal Backend Checks

NVIDIA GPU (CUDA)

CUDA test

```
1 import torch
2 print(torch.cuda.is_available())
```

Apple GPU (Metal / MPS)

Apple GPU test

```
1 import torch
2 print(torch.backends.mps.
      is_available())
```

AMD / Radeon GPUs

- ▶ No native PyTorch GPU backend

Interpretation

- ▶ True → GPU usable
- ▶ False → CPU fallback
- ▶ Backend determines capability

GPU Execution in Python — Laptop vs. HPC

Apple GPU (Metal / MPS)

Apple GPU test

```
1 import torch, time
2
3 d = torch.device("mps")
4 x = torch.rand((4000,4000),
     device=d)
5
6 t0 = time.time()
7 y = torch.matmul(x, x)
8 torch.mps.synchronize()
9
10 print("Apple GPU time:",
11      round(time.time()-t0,3))
```

NVIDIA A100 (CUDA)

HPC GPU test

```
1 import torch, time
2
3 d = torch.device("cuda")
4 x = torch.rand((8000,8000),
     device=d)
5
6 t0 = time.time()
7 y = torch.matmul(x, x)
8 torch.cuda.synchronize()
9
10 print("A100 time:",
11      round(time.time()-t0,3))
```

Interactive GPU Access and Verification Workflow

Step 1: Interactive GPU login

Shell alias (interactive GPU)

```
1 alias gpu1='qlogin -q
    gp_inter_dgx \
2   --gpnum-lhost=1 \
3   --cpnum-lhost=16 \
4   -l elapstim_req=6:00:00 \
5   -l memsz_job=240gb'
```

Step 2: Select GPU explicitly

Shell

```
1 export CUDA_VISIBLE_DEVICES=7
2 echo $CUDA_VISIBLE_DEVICES
```

Step 3: Minimal verification

Mini GPU check in Python

```
1 import torch, os
2 print("CUDA_VISIBLE_DEVICES ="
      , os.getenv("CUDA_VISIBLE_DEVICES"))
3 print("Visible GPUs =", torch.
      cuda.device_count())
4 print("GPU name =", torch.cuda.
      get_device_name(0))
```

- ▶ Interactive login allocates resources
- ▶ GPU visibility must be set **explicitly**
- ▶ Always verify from inside Python

CPU vs GPU — Execution Model and Performance: 30sec → 3sec

CPU example

```
1 import torch, time
2 n = 30000
3 x0 = torch.rand((n,n))
4 x1 = torch.rand((n,n))
5 t0 = time.time()
6 y0 = torch.matmul(x0, x0)
7 y1 = torch.matmul(x1, x1)
8 print(time.time()-t0)
```

Two-GPU example

```
1 import torch, time
2 d0 = torch.device("cuda:0")
3 d1 = torch.device("cuda:1")
4 n. = 30000
5 x0 = torch.rand((n,n), device=d0)
6 x1 = torch.rand((n,n), device=d1)
7 t0 = time.time()
8 torch.matmul(x0, x0)
9 torch.matmul(x1, x1)
10 torch.cuda.synchronize()
11 print(time.time()-t0)
```

Observed

- ▶ Operations are **blocking**
- ▶ Second call starts after first ends
- ▶ Parallelism only inside BLAS
- ▶ Operations run **concurrently**

Parallel Matrix Multiplication — What Actually Works

Key idea

- ▶ Naive multi-GPU often shows no speed-up
- ▶ Reason: data transfers dominate computation
- ▶ Solution: true model parallelism

What works

- ▶ Build matrices directly on each GPU
- ▶ Split the computation (block-wise)
- ▶ Avoid CPU–GPU transfers
- ▶ Collect results only at the end

True model-parallel matrix multiplication

```
1 import torch, time
2 n = 30000
3 A0 = torch.rand((n//2,n), device=
                 "cuda:0")
4 A1 = torch.rand((n//2,n), device=
                 "cuda:1")
5 B = torch.rand((n,n),           device
                 ="cuda:0")
6 t0 = time.time()
7 C0 = A0 @ B
8 C1 = A1 @ B.to("cuda:1")
9 torch.cuda.synchronize()
10 print("Time:", round(time.time()-
                         t0,3))
```

Mixed Precision Computing — Why FP16 Matters

Why reduced precision?

- ▶ Modern GPUs are optimized for FP16 / BF16
- ▶ Higher throughput, lower memory traffic
- ▶ Essential for large ML models

Typical ML pattern

- ▶ Dense linear layers
- ▶ Nonlinear activation
- ▶ Large batch sizes

FP16 neural-network-style workload

```
1 import torch, time
2 torch.set_default_dtype(torch.float16)
3 d = torch.device("cuda")
4 x = torch.randn((20000, 1024), device=d)
5 W1 = torch.randn((1024, 4096), device=d)
6 W2 = torch.randn((4096, 1024), device=d)
7 t0 = time.time()
8 y = torch.nn.functional.gelu(x @ W1)
9 z = y @ W2
10 torch.cuda.synchronize()
11 print("FP16 time:", round(time.time() - t0, 3))
```

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

AI and ML — A Problem-Solving Perspective

Classical approach

- ▶ Explicit equations
- ▶ Physical laws and dynamics
- ▶ Expert-designed structure
- ▶ Limited by model assumptions

$$\partial_t x = F(x, \theta)$$

The temporal change of x is calculated based on x and parameters θ .

AI / ML approach

- ▶ Learn mappings $x \rightarrow y$ from data
- ▶ High-dimensional, nonlinear relations
- ▶ Neural nets as universal approximators
- ▶ Used in natural sciences as well as language

$$\hat{z} = f_\theta(x)$$

Some quantity z is estimated from input x

AI and ML — A Set of Tools

Core ML frameworks

- ▶ PyTorch, TensorFlow
- ▶ scikit-learn
- ▶ Automatic differentiation
- ▶ GPU acceleration

$$\min_{\theta} L(y, \hat{y}(x; \theta))$$

Training = minimizing a loss function by adjusting parameters θ .

AI as a service

- ▶ **LLM APIs** (OpenAI, Mistral, Anthropic, Google, Meta, ...)
- ▶ Pre-trained foundation models
- ▶ On-premise models (Llama, Mistral)
- ▶ Cloud, local, or hybrid use

API call → model inference

Models are used without training from scratch.

What is my own role in this?

AI and ML — A New Paradigm for Interactivity

Human–AI interaction

- ▶ Code assistants
- ▶ Natural language interfaces
- ▶ Interactive problem solving
- ▶ Rapid prototyping

AI in research workflows

- ▶ Data exploration
- ▶ Hypothesis support
- ▶ Equation generation and review
- ▶ Support Reasoning

Prompt → Response

Iterative dialogue replaces
static interfaces.

Human ↔ AI

Collaboration, not replacement.

AI as Partner for Reasoning.

Critical Evaluation I — Reliability and Limits

Strengths

- ▶ Fast pattern recognition
- ▶ Handles high-dimensional data
- ▶ Automates repetitive tasks
- ▶ Strong empirical performance

Limitations

- ▶ No physical understanding
- ▶ Hallucinations possible
- ▶ Sensitive to data bias
- ▶ Weak extrapolation

$$\hat{y} = f_{\theta}(x)$$

Works well within
the learned data regime.

$$f_{\theta}(x) \neq \text{truth}$$

Prediction is not validation.

Critical Evaluation II — Trust, Oversight, Responsibility

Why human oversight matters

- ▶ AI outputs look convincing
- ▶ Errors are often non-obvious
- ▶ No built-in notion of consequences
- ▶ Responsibility remains human

Key risk dimensions

- ▶ Transparency and explainability
- ▶ Bias and unfairness
- ▶ Reproducibility
- ▶ Accountability

Decision = AI + Human

AI supports, it does not decide.

Confidence \neq Correctness

Trust must be earned, not assumed.

Be careful, AI makes many mistakes!

Torch Tensors — The Core Data Structure

What is a tensor?

- ▶ Similar to NumPy arrays
- ▶ Supports CPU and GPU
- ▶ Tracks operations for gradients
- ▶ Basis of all learning

Key properties

- ▶ Shape and dtype
- ▶ Device awareness
- ▶ `requires_grad=True`

Basic tensor example

```
1 import torch
2
3 x = torch.tensor([2.,3.],
4                  requires_grad=True)
5
6 y = x[0]**2 + x[1]**2
7 y.backward()
8 print(x.grad)
```

$$\nabla_x (x_1^2 + x_2^2)$$

Automatic Differentiation (Autograd)

Core idea

- ▶ Gradients computed automatically
- ▶ No manual derivative formulas
- ▶ Works for arbitrary computation graphs
- ▶ Enabled by dynamic graphs

$$\frac{\partial \mathcal{L}}{\partial \theta}$$

Gradients drive parameter updates.

Why this matters

- ▶ Learning = optimization
- ▶ Backpropagation at scale
- ▶ Essential for deep networks
- ▶ Same mechanism on CPU and GPU

$$\theta_{k+1} = \theta_k - \eta \nabla_{\theta} \mathcal{L}$$

Gradient-based learning step.

Data Handling in PyTorch: Dataset and DataLoader

Why data loaders exist

- ▶ Datasets often too large for memory
- ▶ Training uses **mini-batches**
- ▶ Data order matters for optimization
- ▶ Separation of data and model logic

$$(x_i, y_i) \rightarrow (X_B, Y_B)$$

Samples are grouped into batches.

What DataLoader provides

- ▶ Batching
- ▶ Optional shuffling
- ▶ Parallel loading (CPU workers)
- ▶ Consistent interface for training loops

$$(X_B, Y_B) \rightarrow \mathcal{L}(f_\theta(X_B), Y_B)$$

Each batch produces one loss value.

Batches Explained: What Comes Out of the DataLoader

Features and labels

- ▶ Features x : input quantities
- ▶ Labels y : measured target values
- ▶ Learning means fitting $x \rightarrow y$

Structure of the data

- ▶ N = number of samples
- ▶ d = number of features per sample
- ▶ One row = one (x, y) pair

$$X \in \mathbb{R}^{N \times d} \Rightarrow X_B \in \mathbb{R}^{B \times d}$$

Labels and targets

- ▶ Labels collected in Y
 - ▶ One target per input sample
 - ▶ Same batching as for features
- $$Y \in \mathbb{R}^{N \times k} \Rightarrow Y_B \in \mathbb{R}^{B \times k}$$

Why mini-batches help

- ▶ Memory-efficient processing
- ▶ Faster parameter updates
- ▶ Noise improves generalization

$$\nabla_{\theta} \mathcal{L}(X_B, Y_B) \approx \nabla_{\theta} \mathcal{L}(X, Y)$$

Defining a Simple Neural Network

Neural network idea

- ▶ Learn a mapping $f_\theta : x \rightarrow \hat{y}$
- ▶ Parameters θ are trainable
- ▶ Composition of simple operations

Basic building blocks

- ▶ Linear transformation
- ▶ Nonlinear activation
- ▶ Output layer

$$\hat{y} = f_\theta(x)$$

Minimal PyTorch model

```
1 import torch.nn as nn
2
3 class SimpleNN(nn.Module):
4     def __init__(self):
5         super().__init__()
6         self.fc1 = nn.Linear(1,16)
7         self.relu = nn.ReLU()
8         self.fc2 = nn.Linear(16,1)
9
10    def forward(self,x):
11        x = self.fc1(x)
12        x = self.relu(x)
13        return self.fc2(x)
```

What the Model Actually Represents

Model as a function

- ▶ Neural network defines a parametric function
- ▶ Parameters = weights and biases
- ▶ Training adjusts these parameters

$$\hat{y} = W_2 \sigma(W_1 x + b_1) + b_2$$

Nonlinearity σ enables complex mappings.

Trainable parameters

- ▶ Layer 1: $1 \rightarrow 16$
- ▶ Layer 2: $16 \rightarrow 1$

fc1: 16 weights + 16 biases = 32

fc2: 16 weights + 1 bias = 17

Total: $32 + 17 = 49$ parameters

Loss Functions — Measuring Error

What is a loss?

- ▶ Quantifies model error
- ▶ Single scalar value
- ▶ Compares prediction vs label

The loss defines the objective that learning tries to minimize .

Example: Mean Squared Error

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

- ▶ y : true label
- ▶ \hat{y} : model prediction
- ▶ N : number of samples

Lower loss ⇒ better fit .

The Adam Optimizer — Adaptive Gradient Descent

Why optimizers matter

- ▶ Loss defines what to minimize
- ▶ Optimizer defines how
- ▶ Controls stability and speed

Goal: update parameters to reduce loss
efficiently.

Adam combines momentum and scaling:

$$\delta\theta \propto -\frac{\text{average current gradient}}{\text{typical gradient size}}$$

Adam in a nutshell

- ▶ Uses gradients
- ▶ Tracks first moment (mean)
- ▶ Tracks second moment (variance)
- ▶ Adaptive step size per parameter

Parameter update (conceptually):

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

End-to-End Example — Learning a Sine Function

Goal of the example

- ▶ Approximate a known function:
 $\sin(x)$
- ▶ Learn from sampled input–output pairs
- ▶ Demonstrate full ML workflow

Target mapping:

$$x \mapsto \sin(x)$$

Components involved

- ▶ Synthetic dataset (x, y)
- ▶ Neural network model
- ▶ Loss function (error measure)
- ▶ Optimizer updating parameters

Training objective:

$$\min_{\theta} \sum_i \|f_{\theta}(x_i) - y_i\|^2$$

Sine Example — Data and DataLoader

Dataset construction

- ▶ Sample input values x
- ▶ Compute labels $y = \sin(x)$
- ▶ Supervised learning setup

Each sample:

$$x_i \rightarrow y_i$$

Creating dataset and loader

```
1 x = np.linspace(0, 2*np.pi, 1000)
2 y = np.sin(x)
3
4 x_t = torch.tensor(x).float().
    unsqueeze(1)
5 y_t = torch.tensor(y).float().
    unsqueeze(1)
6
7 data = TensorDataset(x_t, y_t)
8 loader = DataLoader(data,
9                         batch_size=32,
10                        shuffle=True)
```

Model and Training Loop

Model idea

- ▶ Input: scalar x
- ▶ Output: scalar \hat{y}
- ▶ Learn nonlinear mapping

Training

- ▶ Compare \hat{y} and y
- ▶ Minimize prediction error
- ▶ Update model parameters

Model and training loop

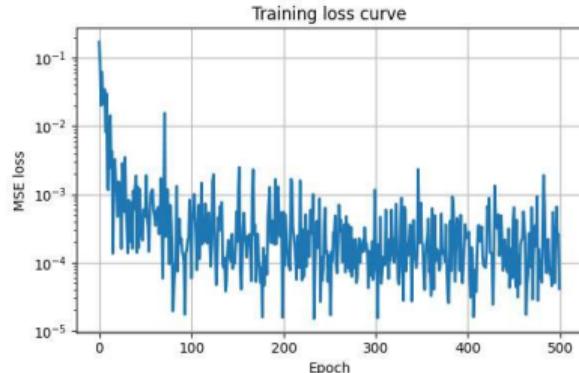
```
1 model = nn.Sequential(  
2     nn.Linear(1,16), nn.ReLU(),  
3     nn.Linear(16,16), nn.ReLU(),  
4     nn.Linear(16,1)  
5 )  
6  
7 loss_fn = nn.MSELoss()  
8 opt = torch.optim.Adam(  
9     model.parameters(), lr=0.01)  
10  
11 for x_b,y_b in loader:  
12     opt.zero_grad()  
13     y_p = model(x_b)  
14     loss = loss_fn(y_p, y_b)  
15     loss.backward()  
16     opt.step()
```

Sine Example — Training Outcome

What happened during training?

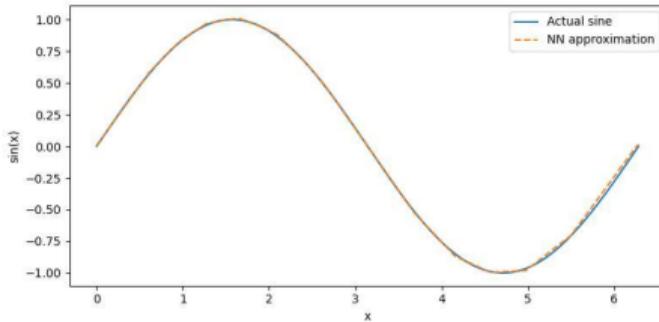
- ▶ Loss decreases over epochs
- ▶ Network parameters adapt
- ▶ Mapping $x \rightarrow \hat{y}$ improves

Loss
during
training



Interpretation

- ▶ Model learned a smooth function
- ▶ No explicit sine formula given
- ▶ Learning via gradient descent



Actual sine (solid) vs. model prediction
(dashed)

DataLoader Without Shuffling

What happens?

- ▶ Samples returned in fixed order
- ▶ Batches follow dataset sequence
- ▶ Same batches every epoch

Why this can be problematic

- ▶ Correlated samples in one batch
- ▶ Biased gradient estimates
- ▶ Slower or unstable learning

Ordered batches

```
1 loader = DataLoader(  
2     dataset,  
3     batch_size=4,  
4     shuffle=False  
5 )  
6  
7 for x_b, y_b in loader:  
8     print(y_b.squeeze())  
9     break
```

First batch always contains the same labels.

DataLoader With Shuffling

What changes?

- ▶ Samples randomly permuted
- ▶ Different batches every epoch
- ▶ Decorrelated gradients

Why this helps

- ▶ More robust optimization
- ▶ Better generalization
- ▶ Standard practice in ML

==== DataLoader WITHOUT shuffling ===

Batch 1: [1, 2, 3, 4, 5, 6]
Batch 2: [7, 8, 9, 10, 11, 12]
Batch 3: [13, 14, 15, 16, 17, 18]

Batch 4: [19, 20]

Shuffled batches

```
1 loader = DataLoader(  
2     dataset,  
3     batch_size=4,  
4     shuffle=True  
5 )  
6  
7 for x_b, y_b in loader:  
8     print(y_b.squeeze())  
9     break
```

First batch changes every run.

==== DataLoader WITH shuffling ===

Batch 1: [3, 4, 15, 7, 18, 6]
Batch 2: [11, 16, 13, 17, 9, 1]
Batch 3: [14, 19, 12, 5, 8, 2]
Batch 4: [20, 10]

From Prediction to Understanding

Prediction is not enough

- ▶ Low error \neq understanding
- ▶ Correct output may hide fragile behavior
- ▶ Especially risky outside training range

Models can be accurate yet **misleading**.

Why gradients matter

- ▶ Sensitivity of output to input
- ▶ Reveal decision boundaries
- ▶ Identify unstable regions

$$\nabla_x f(x)$$

Measures how predictions change locally.

Defining a Classifier — Simple Version

Binary classification setup

- ▶ Input: 2D feature vector
 $x = (x_1, x_2)$
- ▶ Output: probability $\hat{y} \in [0, 1]$
- ▶ Decision via threshold

$$\hat{y} = f_\theta(x)$$

Class label inferred from \hat{y} .

Simple classifier model

```
1 class SimpleClassifier(nn.Module):  
2     def __init__(self):  
3         super().__init__()  
4         self.net = nn.Sequential(  
5             nn.Linear(2,1),  
6             nn.Sigmoid()  
7         )  
8     def forward(self,x):  
9         return self.net(x)
```

Minimal nonlinear decision model.

Improving the Classifier — Nonlinear Model

Why improve the model?

- ▶ Linear boundary often insufficient
- ▶ Real data is nonlinear
- ▶ Need higher expressive power

$$\hat{y} = \sigma(W_2 \phi(W_1 x))$$

Hidden layer enables nonlinear separation.

Better classifier model

```
1 class BetterClassifier(nn.Module):  
2     def __init__(self):  
3         super().__init__()  
4         self.net = nn.Sequential(  
5             nn.Linear(2, 32),  
6             nn.ReLU(),  
7             nn.Linear(32, 1),  
8             nn.Sigmoid()  
9         )  
10    def forward(self, x):  
11        return self.net(x)
```

Nonlinear decision boundary.

Applying the Classifier to Data

Training the classifier

- ▶ Input: feature vectors (x_1, x_2)
- ▶ Output: class probability \hat{y}
- ▶ Supervised binary classification

Goal

- ▶ Separate two classes
- ▶ Learn a decision boundary
- ▶ Minimize classification error

Training loop

```
1 loss_fn = nn.BCELoss()  
2 opt = optim.Adam(model.parameters(),  
                  lr=0.01)  
3  
4 for epoch in range(epochs):  
5     opt.zero_grad()  
6     y_p = model(X)  
7     loss = loss_fn(y_p, y_true)  
8     loss.backward()  
9     opt.step()
```

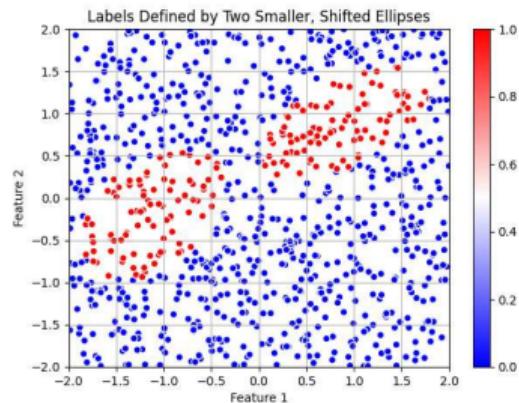
Prediction $\hat{y} \in [0, 1]$ interpreted as probability.

```
grid_points = torch.stack([xx.flatten(), yy.flatten()], dim=1)  
grid_points.requires_grad = True  
grid_grads = grid_points.grad.detach().numpy()
```

Labels vs. Learned Classification

Ground truth labels

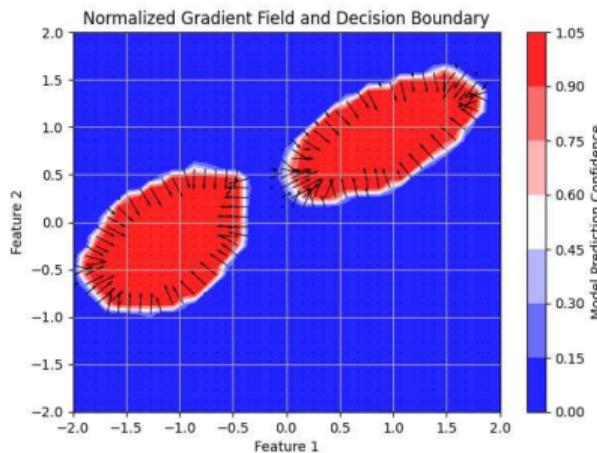
- ▶ Classes defined by geometry
- ▶ Two shifted, rotated ellipses
- ▶ Labels fixed before learning



Blue / red = predefined classes

Model prediction

- ▶ Network output $\hat{y} \in [0, 1]$
- ▶ Nonlinear decision boundary
- ▶ Smooth transition between classes



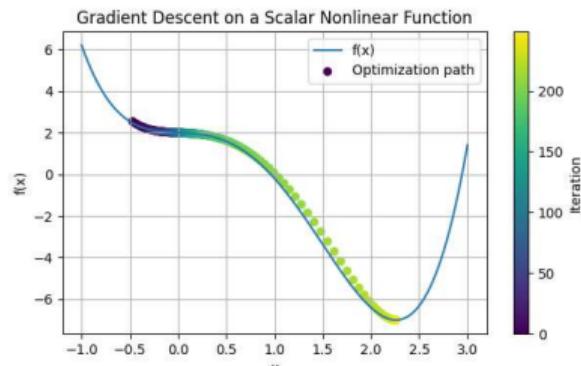
Colors = prediction, arrows = gradients

Chapter 4 — Take-Home Messages

What AI/ML really is

- ▶ Differentiable function approximation
- ▶ Learned from data, not hard-coded
- ▶ Optimized via gradients

Learn f_θ s.t. $f_\theta(x) \approx y$



What really matters

- ▶ Tensors + autograd are the core
- ▶ Loss defines what is learned
- ▶ Data handling controls stability
- ▶ **Domain knowledge remains essential**

AI supports decisions — it does not replace responsibility.

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Neural Network Architectures — Why Structure Matters

Key idea

- ▶ Network structure define the inductive framework
- ▶ Architecture encodes assumptions
- ▶ Learning is constrained by connectivity

Conceptual view

- ▶ Architecture = hypothesis space
- ▶ Optimizer explores this space
- ▶ Data selects a solution

$$\mathcal{H}_{\text{model}} \xrightarrow{\text{training}} \hat{f}$$

Same data, different models

- ▶ Feed Forward Networks
- ▶ Graph Neural Networks
- ▶ Convolutional Networks
- ▶ Recurrent / LSTM models

$$\begin{aligned}\mathcal{H}_{\text{linear}} &= \{ f(x) = w^\top x + b \mid w, b \in \mathbb{R} \} \\ \mathcal{H}_{\text{FFNN}} &= \{ f_\theta(x) = W_2 \sigma(W_1 x) \} \\ \mathcal{H}_{\text{CNN}} &= \{ f_\theta(x) = \sigma(K * x + b) \} \\ \mathcal{H}_{\text{GNN}} &= \{ f_\theta(G) \mid G = (V, E, X) \}\end{aligned}$$

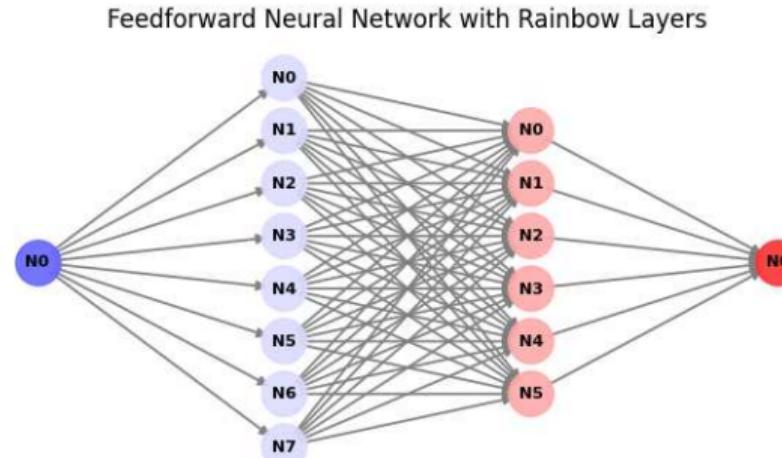
Feed Forward Neural Networks (FFNN)

Basic idea

- ▶ Directed **acyclic** network
- ▶ Information flows strictly forward
- ▶ **No memory , no recurrence**

Typical use cases

- ▶ Regression
- ▶ Classification
- ▶ Function approximation



Input → hidden layers → output

Each layer applies a learned transformation.

FFNN as a Mathematical Mapping

Layer-wise computation

- ▶ Affine transformation
- ▶ Nonlinear activation
- ▶ Composition of functions

$$\begin{aligned}x^{(0)} &= x \\x^{(1)} &= \sigma(W_1 x^{(0)} + b_1) \\x^{(2)} &= \sigma(W_2 x^{(1)} + b_2) \\\hat{y} &= W_3 x^{(2)} + b_3\end{aligned}$$

Each layer increases expressiveness .

Purely data-driven mapping.

σ : ReLU, tanh, sigmoid

Model Capacity and Trainable Parameters

What defines model capacity?

- ▶ Number of parameters
- ▶ Network depth
- ▶ Width of layers

Higher capacity:

- ▶ Fits more complex functions
- ▶ Risk of overfitting

Parameter counting

Each Layer ($n_{\text{in}} \rightarrow n_{\text{out}}$) : $n_{\text{in}} \cdot n_{\text{out}} + n_{\text{out}}$

Total parameters = $\sum_{\ell} (n_{\ell-1} n_{\ell} + n_{\ell})$

Example (1–16–16–1):

$$(1 \cdot 16 + 16) + (16 \cdot 16 + 16) + (16 \cdot 1 + 1) = 337$$

Why Depth Matters

Shallow networks

- ▶ Few layers, many neurons
- ▶ Can approximate any function
- ▶ Often inefficient

Universal Approximation:

- ▶ Existence result
- ▶ Not a statement about efficiency

Deep networks

- ▶ Hierarchical feature extraction
- ▶ Reuse of intermediate representations
- ▶ Fewer parameters for same accuracy

Compositional structure:

$$f(x) = f_L(f_{L-1}(\dots f_1(x)))$$

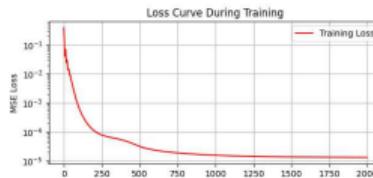
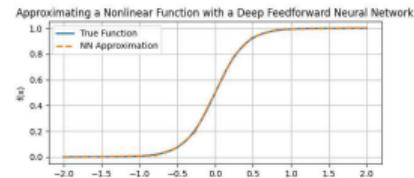
```
class Deep(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.net = nn.Sequential(  
            nn.Linear(1, N1), structure, not  
            nn.Tanh(),  
            nn.Linear(N1, N1), capacity.  
            nn.Tanh(),  
            nn.Linear(N1, N1))
```

FFNN as a Function Approximator

Problem setting

- ▶ Unknown target function $f(x)$
- ▶ Discrete training data available
- ▶ Goal: approximate f using a neural network

$$f(x) \approx f_\theta(x)$$



Ground truth (left) vs. NN approximation and training history (right)

Neural-network view

- ▶ The network defines a family of functions
- ▶ Parameters determine the concrete shape
- ▶ Training = selecting a suitable function

Intuition

- ▶ Each layer further shapes the function
- ▶ Nonlinearities enable complex forms
- ▶ A smooth approximation emerges step by step

Computational Graph and Backpropagation

Forward pass

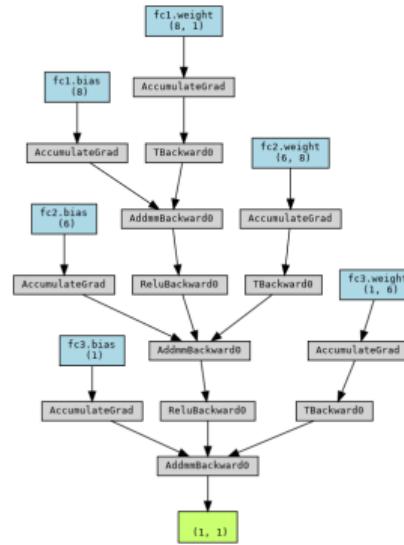
- ▶ Input propagated layer by layer
- ▶ Linear maps + nonlinear activations
- ▶ Produces model output \hat{y}

Backward pass

- ▶ Loss gradient flows backward
- ▶ Chain rule applied automatically
- ▶ Gradients stored in parameters

Training adjusts parameters using these gradients.

Computational graph of a feedforward network



Key idea

- ▶ Graph encodes all operations
- ▶ Enables exact gradient computation
- ▶ Foundation of learning by optimization

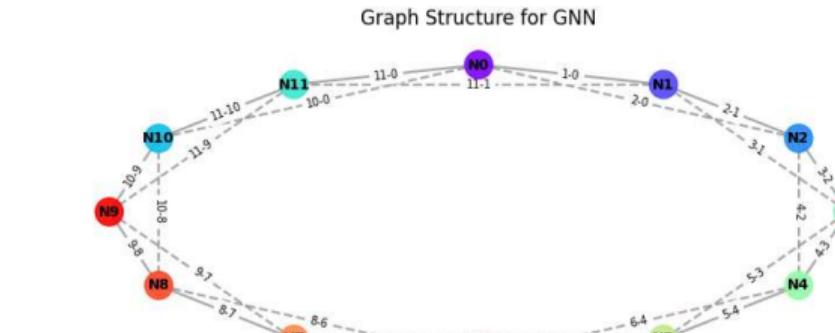
Why Graph Neural Networks?

When FFNNs are not enough

- ▶ Data points are not independent
- ▶ Relations between entities matter
- ▶ Ordering is not fixed or meaningful

Typical examples

- ▶ Physical grids and meshes
- ▶ Sensor networks
- ▶ Molecules, social networks



Graph structure with nodes and edges

Core idea

- ▶ Nodes exchange information
- ▶ Learning respects graph structure
- ▶ Inductive bias for relational data

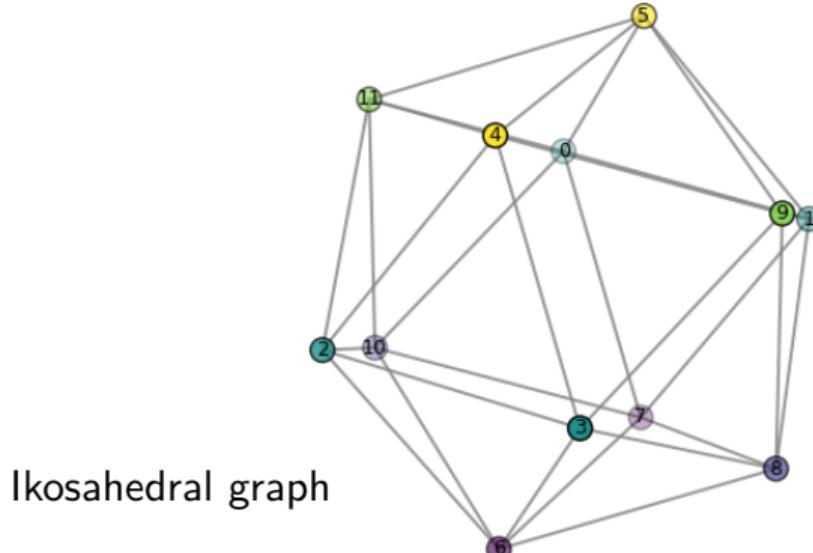
Basic Graph Concepts

Graph definition

- ▶ Graph $G = (V, E)$
- ▶ Nodes V : entities
- ▶ Edges E : relations

Node data

- ▶ Each node has features x_i
- ▶ Labels y_i for supervision
- ▶ Features may be physical states



Ikosahedral graph

Key distinction

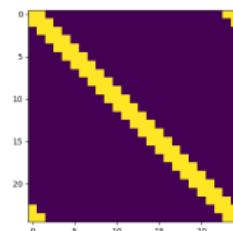
- ▶ Data lives on nodes
- ▶ Structure lives in edges
- ▶ Learning uses both

Adjacency and edge_index

Adjacency matrix

- ▶ Matrix $A \in \{0, 1\}^{N \times N}$
- ▶ $A_{ij} = 1$ if nodes i, j are connected
- ▶ Simple, but memory expensive

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$



edge_index representation

- ▶ Sparse edge list format
- ▶ Two rows: source and target nodes
- ▶ Standard in PyTorch Geometric

$$\text{edge_index} = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \end{pmatrix}$$

Why this matters

- ▶ Scales to large graphs
- ▶ Efficient message passing
- ▶ Natural for irregular structures

GNN Architecture Principle

Key idea

- ▶ Nodes exchange information
- ▶ Messages flow along edges
- ▶ Features are updated iteratively

One GNN layer

- ▶ Collect neighbor features
- ▶ Aggregate (sum / mean / max)
- ▶ Apply learnable transform

Generic update rule

$$h_i^{(k+1)} = \sigma \left(W^{(k)} \sum_{j \in \mathcal{N}(i)} h_j^{(k)} \right)$$

Interpretation

- ▶ $h_i^{(k)} \in \mathbb{R}^{d_k}$: feature vector, node i at layer k
- ▶ $W^{(k)} \in \mathbb{R}^{d_{k+1} \times d_k}$: matrix
- ▶ $\mathcal{N}(i)$: neighbors of node i
- ▶ σ : nonlinearity

Consequence

- ▶ Local interactions build global structure
- ▶ Depth = range of information propagation

GNN Application Example — Advection on a Periodic Grid

Physical setup

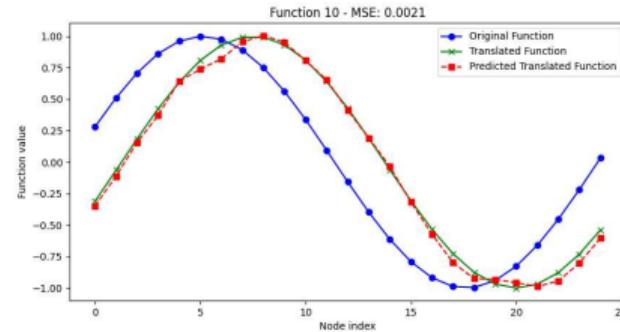
- ▶ Nodes arranged on a periodic ring
- ▶ Each node holds a scalar field value
- ▶ Goal: learn one-step time evolution

Learning task

- ▶ Input: field at time t
- ▶ Output: field at time $t + \Delta t$
- ▶ Advection-like transport process

Mathematically:

$$z^{t+1} \approx f_\theta(G, z^t)$$



Graph structure with periodic connectivity

Why a GNN?

- ▶ Local interactions dominate dynamics
- ▶ Translation invariance on the ring
- ▶ Same update rule for all nodes

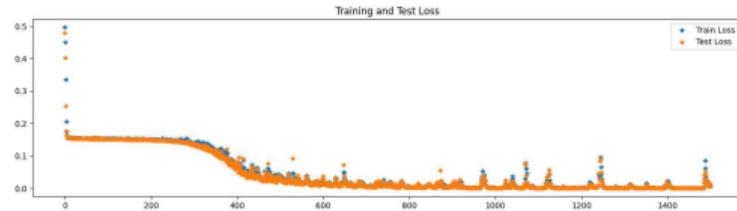
GNN Results — Learning Advection Dynamics

Training behaviour

- ▶ Loss decreases steadily
- ▶ Stable convergence for train and test
- ▶ No explicit physics encoded

What is learned?

- ▶ Local transport along edges
- ▶ Approximate shift of the field
- ▶ Graph-based discretization of dynamics



Training and test loss over epochs

Limitations

- ▶ Short-term prediction only
- ▶ Error accumulation over time
- ▶ Stability not guaranteed

Interpretation:

Why Convolutional Neural Networks?

Key idea

- ▶ Exploit local structure
- ▶ Same operation everywhere
- ▶ Strong inductive bias

Typical data

- ▶ Images (2D grids)
- ▶ Time series (1D signals)
- ▶ Physical fields on grids

CNNs assume:

locality + translation invariance

Convolution instead of full connectivity

- ▶ Small kernel slides over input
- ▶ Shared weights across positions
- ▶ Far fewer parameters than FFNNs

Mathematically (1D):

$$y_i = \sum_{k=-K}^K w_k x_{i+k}$$

Interpretation:

- ▶ Learn local patterns
- ▶ Detect edges, waves, peaks
- ▶ Build hierarchy via depth

Example Setup: Function Classification

Goal

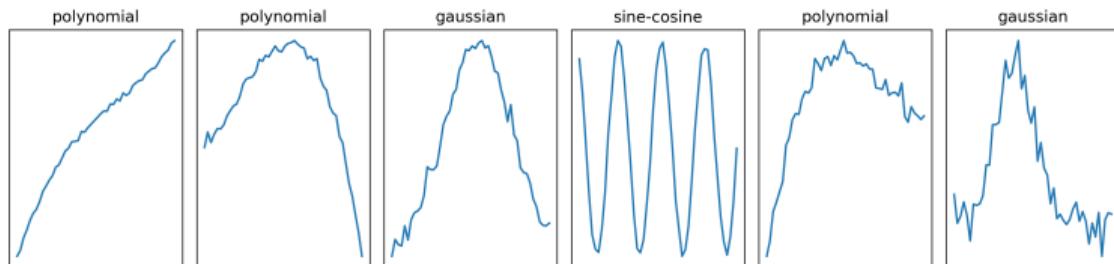
- ▶ Input: sampled 1D function
- ▶ Output: function class

CNN sees:

- ▶ Local patterns
- ▶ Overall structure

Classes:

- ▶ Sine-like
- ▶ Gaussian
- ▶ Polynomial



Samples:

- ▶ Same grid
- ▶ Different shape

Example input functions (with noise)

CNN Structure for Function Classification

Network structure

- ▶ 1D convolutions for local patterns
- ▶ Nonlinear feature extraction
- ▶ Fully connected classifier

Input: (B, 1, 50)
Conv1: (B, 16, 50)
Conv2: (B, 32, 50)
Flatten: (B, 1600)
FC: (B, 4)

CNN model definition

```
1 class FunctionClassifierCNN(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.conv1 = nn.Conv1d(1, 16, 5, padding=2)
5         self.conv2 = nn.Conv1d(16, 32, 5, padding=2)
6         self.fc1   = nn.Linear(32*50, 128)
7         self.fc2   = nn.Linear(128, 4)
8
9     def forward(self, x):
10        x = torch.relu(self.conv1(x))
11        x = torch.relu(self.conv2(x))
12        x = x.view(x.size(0), -1)
13        x = torch.relu(self.fc1(x))
14        return self.fc2(x)
```

Training the CNN

Training step

- ▶ Forward pass
- ▶ Compute classification loss
- ▶ Backpropagate gradients

Loss & optimizer

- ▶ Cross-entropy for classes
- ▶ Adam optimizer

Loss: compares logits vs. labels

Grad: flows through conv + FC

Update: adjusts all weights

CNN training loop

```
1 criterion = nn.CrossEntropyLoss()
2 optimizer = torch.optim.Adam(
3     model.parameters(), lr=1e-3)
4
5 for epoch in range(num_epochs):
6     for xb, yb in train_loader:
7         xb = xb.to(device)
8         yb = yb.to(device)
9
10        optimizer.zero_grad()
11        logits = model(xb)
12        loss = criterion(logits, yb)
13        loss.backward()
14        optimizer.step()
```

Prediction: Function Classification

Inference mode

- ▶ No gradients
- ▶ Fixed trained weights
- ▶ Forward pass only

Classification

- ▶ Output: class logits
- ▶ Argmax selects class

logits → scores per class
argmax → predicted label

CNN prediction

```
1 model.eval()  
2  
3 with torch.no_grad():  
4     logits = model(X_test.to(device))  
5     preds = torch.argmax(logits, dim=1)  
6  
7 accuracy = (preds == y_test.to(device)).  
               float().mean()  
8 print("Accuracy:", accuracy.item())
```

Prediction result:

- ▶ Each function → class
- ▶ Robust to moderate noise

CNN Predictions on Test Data

What is shown?

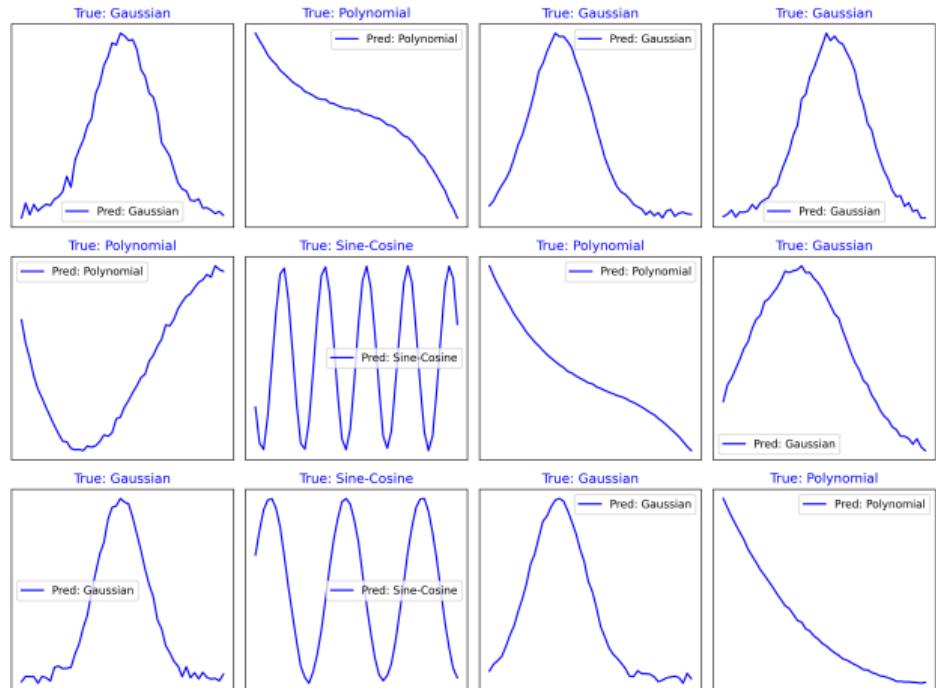
- ▶ Unseen test functions: new samples from random distribution
- ▶ Higher noise than training

Color coding

- ▶ Blue: correct classification
- ▶ Red: misclassification

Interpretation:

- ▶ CNN recognizes shape
- ▶ Errors near ambiguous cases



Why LSTMs for Time Series and Anomalies?

Problem setting

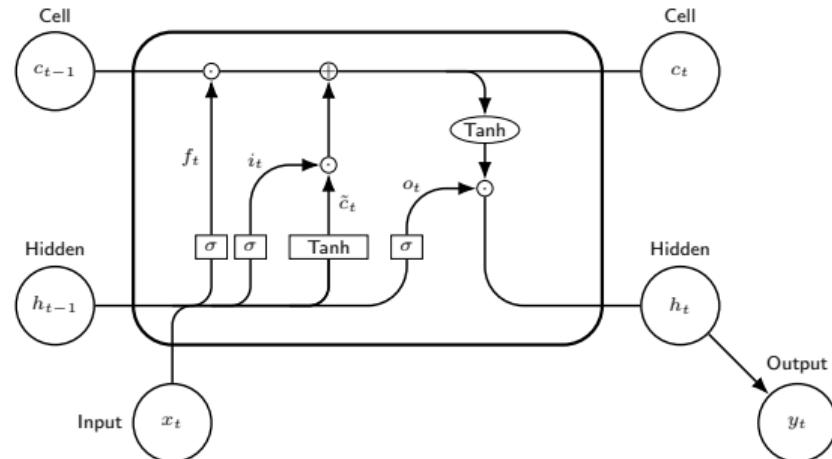
- ▶ Sequential sensor data
- ▶ Temporal correlations
- ▶ Deviations over time

Why not FFNN / CNN?

- ▶ No explicit memory
- ▶ Limited temporal context

Key idea:

anomaly = temporal
inconsistency



LSTM provides:

- ▶ Explicit memory state
- ▶ Controlled information flow
- ▶ Long-term dependency modeling

LSTM Cell: Gated Memory Equations

State variables

- ▶ Cell state c_t : long-term memory
- ▶ Hidden state h_t : exposed state

Gate intuition

- ▶ Forget: scale past memory
- ▶ Input: add new information
- ▶ Output: control exposure

Key property:

- ▶ Additive update of c_t

$$\begin{aligned}f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\h_t &= o_t \odot \tanh(c_t)\end{aligned}$$

- ▶ $x_t \in \mathbb{R}^m$: temporal input at time t
- ▶ $h_t \in \mathbb{R}^n$: hidden state
- ▶ $c_t \in \mathbb{R}^n$: cell state
- ▶ f_t, i_t, o_t : gates
- ▶ \tilde{c}_t : candidate cell update
- ▶ W_*, U_*, b_* : learnable parameters

From LSTM States to Reconstruction Error

What goes in

- ▶ Input sequence $x_{1:T}$
- ▶ One value per time step

Reconstruction step

$$\hat{x}_t = W_y h_t + b_y$$

What the LSTM does

- ▶ Updates (h_t, c_t) sequentially
- ▶ Encodes temporal structure

Training objective

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^T \|x_t - \hat{x}_t\|^2$$

What comes out

- ▶ Hidden states h_t
- ▶ Latent temporal representation

Anomaly detection

- ▶ Low error: normal sequence
- ▶ High error: anomalous sequence

W_y, b_y learned jointly with LSTM weights

LSTM Autoencoder for Anomaly Detection

Autoencoder principle

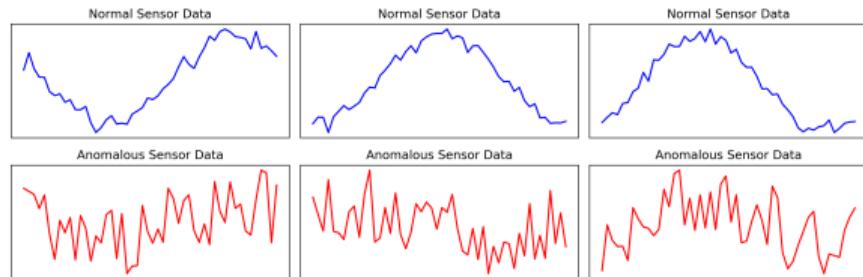
- ▶ Encode normal behaviour
- ▶ Reconstruct input sequence

Anomaly criterion

- ▶ Large reconstruction error
- ▶ Rare under normal dynamics

Key assumption:

normal \Rightarrow low error



Normal vs. anomalous sensor sequences

Model learns:

- ▶ Typical temporal patterns

Coding the LSTM Autoencoder

Model structure

- ▶ Encoder LSTM
- ▶ Decoder LSTM
- ▶ Linear reconstruction

Data flow:

- ▶ Sequence → hidden state
- ▶ Hidden state → sequence

Input: (B, T, 1)

Hidden: (L, B, H)

Output: (B, T, 1)

LSTM autoencoder (PyTorch)

```
1 class LSTMAutoencoder(nn.Module):  
2     def __init__(self, hidden_dim=32, layers=2):  
3         super().__init__()  
4         self.encoder = nn.LSTM(  
5             1, hidden_dim, layers, batch_first=True)  
6         self.decoder = nn.LSTM(  
7             1, hidden_dim, layers, batch_first=True)  
8         self.out = nn.Linear(hidden_dim, 1)  
9  
10    def forward(self, x):  
11        _, (h, c) = self.encoder(x)  
12        z = torch.zeros_like(x)  
13        y, _ = self.decoder(z, (h, c))  
14        return self.out(y)
```

Detected Anomalies in Sensor Data

What is shown?

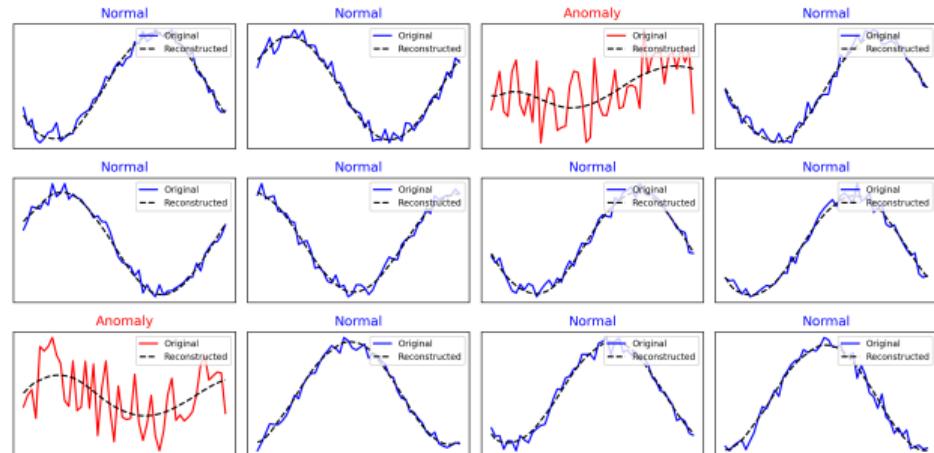
- ▶ Unseen test sequences
- ▶ Reconstruction vs. input

Color coding

- ▶ Blue: normal
- ▶ Red: anomaly

Interpretation:

- ▶ Model flags temporal inconsistency



Original (solid) vs. reconstructed (dashed)

Neural Network Architectures — Chapter Summary

Architectures covered

- ▶ FFNN : global function approximation
- ▶ GNN : interactions on graphs
- ▶ CNN : local pattern extraction
- ▶ LSTM : temporal dependencies

Inductive bias / framework:

- ▶ Structure encoded in connectivity

Tasks demonstrated

- ▶ FFNN: regression , classification
- ▶ GNN: transport and dynamics
- ▶ CNN: function classification
- ▶ LSTM: anomaly detection

Core message:

- ▶ Architecture must match data structure
- ▶ No universally optimal network

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00	Joint Dinner				

Why Large Language Models?

Traditional NLP

- ▶ Separate models for each task
- ▶ Feature engineering required
- ▶ Limited transfer between tasks

Examples

- ▶ Translation
- ▶ Classification
- ▶ Question answering

LLMs **unify** understanding, generation, and reasoning in a single sequence model.

$$\{\text{The, dog, chas, ed, the, cat, .}\} \longrightarrow \{104, 2871, 9123, 214, 201, 4421, 13\}$$

Large Language Models

- ▶ One model, many tasks
- ▶ Learns representations from data
- ▶ Includes Sub-Word **Tokenization**
- ▶ General-purpose language intelligence

Key idea

- ▶ Language as a *sequence prediction problem*
- ▶ Everything reduced to:

$$\text{tokens}_{\text{in}} \longrightarrow \text{tokens}_{\text{out}}$$

LLMs as Sequence-to-Sequence Machines

Core abstraction

- ▶ Input: sequence of tokens
- ▶ Output: sequence of tokens
- ▶ **Tokens** represent words or subwords

Mathematical view

$$(x_1, \dots, x_n) \longrightarrow (y_1, \dots, y_m)$$

- ▶ Variable-length input
- ▶ Variable-length output

Next-token prediction

- ▶ Output generated one token at a time
- ▶ Each step predicts a probability distribution

$$p(y_t \mid x_{1:n}, y_{1:t-1})$$

Consequences

- ▶ Same model for all tasks
- ▶ Tasks differ only by input prompt
- ▶ No task-specific architecture needed

LLMs reduce language understanding and generation to probabilistic sequence modeling.

From Token IDs to Embedding Vectors

What the model actually sees

- ▶ Tokens are integers
- ▶ No words, no strings, no grammar

Example:

{104, 2871, 9123, 214, 201, 4421, 13}

Problem

- ▶ Integers have no semantic meaning
- ▶ Distance between IDs is arbitrary

Solution: embeddings

Each token ID i is mapped to a vector:

$$i \longrightarrow e_i \in \mathbb{R}^{d_{\text{model}}}$$

Embedding matrix

$$E \in \mathbb{R}^{V \times d_{\text{model}}}$$

- ▶ V : vocabulary size (number of distinct tokens)
- ▶ d_{model} : embedding dimension
- ▶ $E_i = e_i$: embedding of token i

Meaning is not stored in tokens, but in their learned vector representations.

Why Sequence Order Matters

Problem with embeddings alone

- ▶ Embeddings encode token meaning
- ▶ But ignore position in the sequence

Example:

dog bites man \neq man bites dog

Yet both contain the same tokens.

Key observation

- ▶ Transformers process tokens in parallel
- ▶ No inherent notion of order

Consequence

- ▶ Sequence order must be added explicitly
- ▶ Otherwise:

$$(x_1, x_2, x_3) \equiv (x_3, x_1, x_2)$$

Order information is not optional.

Transformers require an **explicit mechanism to encode token positions**.

Positional Encoding

Goal

- ▶ Inject **order information** into embeddings
- ▶ Keep full parallelism

Idea

- ▶ Assign a position-dependent vector
- ▶ Add it to the token embedding

$$x_i = e_i + p_i$$

Positional encodings turn a set of tokens into an ordered sequence.

Positional encoding matrix

$$P \in \mathbb{R}^{n \times d_{\text{model}}}$$

- ▶ n : sequence length
- ▶ p_i : encoding of position i

Key properties

- ▶ Same dimension as embeddings
- ▶ Fixed (sinusoidal) or learned
- ▶ Enables attention to use order

Sinusoidal Positional Encoding

Sinusoidal construction

- ▶ Fixed, deterministic encoding
- ▶ Different frequencies per dimension

For position i and dimension j :

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d_{\text{model}}}}\right)$$

$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d_{\text{model}}}}\right)$$

- ▶ Smooth variation with position
- ▶ Relative distances are encoded

Why sinusoids?

- ▶ Works for arbitrary sequence lengths
- ▶ No additional parameters
- ▶ Relative positions can be inferred

Why *add*, not concatenate?

- ▶ Addition keeps dimension at d_{model}
- ▶ Attention projections expect fixed size
- ▶ Concatenation doubles dimension
- ▶ Would require retraining all layers

Addition preserves efficiency
and model structure.

Positional information is injected without changing model dimensionality.

Self-Attention: Core Idea

Key mechanism

- ▶ Each token can look at all other tokens
- ▶ Importance is computed dynamically

Contextual representation

- ▶ Token meaning depends on context
- ▶ Same word, different role

Example:

“The dog chased the cat.”

What attention does

- ▶ Builds a weighted combination of tokens
- ▶ Different focus for each position

For token i :

$$x_i \longrightarrow \sum_j w_{ij} x_j$$

Why this matters

- ▶ Long-range dependencies
- ▶ No fixed context window
- ▶ Fully parallel computation

Self-attention allows each token to decide which other tokens matter.

Queries, Keys, and Values — Intuition

Three roles per token

Each token embedding is projected into:

- ▶ **Query** (Q): what am I looking for?
- ▶ **Key** (K): what do I offer?
- ▶ **Value** (V): what information do I pass on?

All three are learned linear projections.

$$Q = XW^Q, K = XW^K, V = XW^V$$

$$\begin{aligned} X &\in \mathbb{R}^{n \times d_{\text{model}}}, W^Q, W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \\ W^V &\in \mathbb{R}^{d_{\text{model}} \times d_V}, Q, K \in \mathbb{R}^{n \times d_k}, V \in \mathbb{R}^{n \times d_V} \end{aligned}$$

Attention mechanism

- ▶ Compare queries with keys
- ▶ Compute relevance scores
- ▶ Use scores to weight values

Conceptually:

$$\text{attention}(i, j) \sim Q_i \cdot K_j$$

Interpretation

- ▶ Tokens ask questions (queries)
- ▶ Other tokens answer (values)
- ▶ Keys decide relevance

Scaled Dot-Product Attention

Attention scores

- ▶ Compare each query with all keys
- ▶ Dot product measures similarity

$$S = QK^T \quad \in \quad \mathbb{R}^{n \times n}$$

Each row: relevance of one token to all others.

Scaling and normalization

- ▶ Scale by $\sqrt{d_k}$ for numerical stability
- ▶ Apply softmax row-wise

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

Weighted aggregation

$$Z = AV$$

- ▶ $Z \in \mathbb{R}^{n \times d_v}$
- ▶ Each token mixes information from others

Attention computes a context-dependent weighted sum of values.

Why Scale by $\sqrt{d_k}$?

Problem without scaling

- ▶ Dot products grow with dimension
- ▶ Large values enter softmax
- ▶ Gradients become unstable

For random vectors:

$$Q_i \cdot K_j \sim \mathcal{O}(d_k)$$

Effect of scaling

- ▶ Divide by $\sqrt{d_k}$
- ▶ Keeps variance roughly constant

$$\frac{QK^T}{\sqrt{d_k}}$$

Result

- ▶ Softmax stays sensitive
- ▶ Stable gradients during training
- ▶ Faster convergence

Scaling is a numerical necessity, not a heuristic.

The scaling factor prevents softmax saturation in high dimensions.

Attention Matrix Interpretation

Attention matrix

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \in \mathbb{R}^{n \times n}$$

- ▶ Row i : attention of token i
- ▶ Columns: which tokens are attended to

Softmax: Each row sums to 1:

$$\sum_j A_{ij} = 1$$

Interpretation

- ▶ Soft, learned dependency graph
- ▶ Fully connected
- ▶ Directional (row-wise)

Key insight

- ▶ No fixed neighborhood
- ▶ No predefined structure
- ▶ Recomputed at every layer

Attention learns which tokens influence each other.

Self-attention dynamically constructs a weighted interaction graph.

Why Multi-Head Attention?

Limitation of single attention

- ▶ One attention matrix per layer
- ▶ Focuses on one similarity notion
- ▶ Mixes all information at once

Example needs:

- ▶ Syntax (subject–verb)
- ▶ Semantics (actor–object)
- ▶ Position and locality

Multi-head idea

- ▶ Run attention in parallel
- ▶ Different subspaces
- ▶ Different focus patterns

Each head learns:

its own Q_i , K_i , V_i

Benefit

- ▶ Richer representations
- ▶ Multiple relationships at once
- ▶ Improved expressiveness

Multi-head attention lets the model attend to different aspects simultaneously.

Multi-Head Attention: Formulation

Per-head projections

For each head $h = 1, \dots, H$:

$$\begin{aligned} Q^{(h)} &= XW^{Q^{(h)}}, & K^{(h)} &= XW^{K^{(h)}}, \\ V^{(h)} &= XW^{V^{(h)}} \end{aligned}$$

Each head applies scaled dot-product attention :

$$Z^{(h)} = \text{softmax}\left(\frac{Q^{(h)}K^{(h)T}}{\sqrt{d_k}}\right)V^{(h)}$$

Concatenation and projection

$$Z = \text{Concat}(Z^{(1)}, \dots, Z^{(H)})$$

Final linear projection:

$$\text{MultiHead}(X) = ZW^O$$

- ▶ H : number of heads
- ▶ W^O : output projection
- ▶ Output in $\mathbb{R}^{n \times d_{\text{model}}}$

Multiple attention views are merged into one representation.

Multi-head attention preserves model dimension while increasing expressiveness.

Position-Wise Feedforward Network (FFN)

- ▶ Inside each Transformer block
- ▶ After self-attention

What attention does not do

- ▶ Computes weighted averages
- ▶ Linear combination of values
- ▶ No feature-wise nonlinearity

Why the FFN is needed

- ▶ Adds nonlinearity, Recombines features
- ▶ Increases model expressiveness

Feedforward mapping

For one token vector $x \in \mathbb{R}^{d_{\text{model}}}$:

$$\text{FFN}(x) = \sigma(xW_1 + b_1)W_2 + b_2$$

- ▶ $\sigma(\cdot)$: ReLU or GELU
- ▶ $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$
- ▶ $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$

Applied to all tokens:

$$Z_{\text{ff}} \in \mathbb{R}^{n \times d_{\text{model}}}$$

Same FFN parameters are shared across all positions.

Attention mixes tokens; the FFN transforms token features.

Residual Addition, Layer Normalization Layer normalization

Applied after the residual sum, per token i :

Output of multi-head attention

Multi-head attention output (after W^O):

$$Z_{\text{att}} = \text{MultiHead}(X) \in \mathbb{R}^{n \times d_{\text{model}}}$$

$$\mu_i = \frac{1}{d_{\text{model}}} \sum_{k=1}^{d_{\text{model}}} \tilde{X}_{ik}$$

$$\sigma_i^2 = \frac{1}{d_{\text{model}}} \sum_{k=1}^{d_{\text{model}}} (\tilde{X}_{ik} - \mu_i)^2$$

$$\tilde{X} = X + Z_{\text{att}}$$

$$Z_{ik} = \gamma_k \frac{\tilde{X}_{ik} - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}} + \beta_k$$

- ▶ Preserves original information
- ▶ Requires matching dimensions
- ▶ Normalization across feature dimension
- ▶ γ_k, β_k : learned parameters
- ▶ No mixing between tokens

Residual connections preserve information; LayerNorm stabilizes the representation.

Output Projection to Vocabulary

Final Transformer output

After all Transformer blocks:

$$Z \in \mathbb{R}^{n \times d_{\text{model}}}$$

Each row corresponds to:

- ▶ One token position
- ▶ A contextualized representation

Goal

- ▶ Predict the next token
- ▶ From a fixed vocabulary

Logits convert hidden representations into token probabilities.

Linear projection to logits

$$Z_{\text{logits}} = ZW_{\text{out}} + b_{\text{out}}$$

- ▶ $W_{\text{out}} \in \mathbb{R}^{d_{\text{model}} \times V}$
- ▶ $b_{\text{out}} \in \mathbb{R}^V$
- ▶ V : vocabulary size

Softmax probabilities

$$Z_{\text{pred}} = \text{softmax}(Z_{\text{logits}})$$

Each row is a probability distribution over tokens.

Cross-Entropy Loss for Language Modeling

Prediction target

- ▶ One correct token per position
- ▶ Stored as an index in the vocabulary

For position t : $y_t \in \{1, \dots, V\}$ (token index). Equivalent **one-hot** representation:

$$e_{y_t} \in \mathbb{R}^V.$$

Model prediction: $p_t = Z_{\text{pred}, t} \in \mathbb{R}^V$.

Cross-entropy loss

$$\mathcal{L} = - \sum_{t=1}^n \log p_t(y_t)$$

- ▶ $p_t(y_t)$: probability assigned to the correct token
- ▶ Uses index y_t to select one entry

Interpretation

- ▶ High confidence, correct \rightarrow low loss
- ▶ Wrong or uncertain \rightarrow high loss

The model is trained to maximize likelihood of the true sequence.

Cross-entropy measures how well predicted probabilities match the true tokens.

Training the Transformer Model

Training objective

- ▶ Minimize cross-entropy loss
- ▶ Over all tokens in the sequence

$$\mathcal{L} = - \sum_{t=1}^n \log p_t(y_t)$$

Learnable parameters

- ▶ Embeddings
- ▶ Attention projections (W^Q, W^K, W^V, W^O)
- ▶ Feedforward weights
- ▶ Output projection W_{out}

Gradient-based optimization

- ▶ Compute gradients via backpropagation
- ▶ Update parameters to reduce loss

Generic update rule:

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

In practice

- ▶ Adam or AdamW optimizer
- ▶ Mini-batch training
- ▶ Many epochs over large corpora

Training adjusts all parameters to improve
next-token prediction.

Autoregressive Text Generation

Key idea

- ▶ Generate one token at a time
- ▶ Each new token is fed back as input

At position t :

$$p_t = P(x_t \mid x_1, \dots, x_{t-1})$$

Training vs inference

- ▶ Training: true previous tokens known
- ▶ Inference: model uses its own output

Generation loop

1. Start with a prompt
2. Compute logits
3. Apply softmax
4. Select next token
5. Append and repeat

Token selection

- ▶ Argmax (greedy)
- ▶ Sampling, repetition penalty
- ▶ Top- k , top- p (nucleus)

LLMs are trained once, then generate text step by step.

A Tiny Language Dataset

Vocabulary

- ▶ Small, fixed word set
- ▶ Each word mapped to an ID

Example:

“I am hungry” → [1, 2, 43]

Padding

- ▶ Fixed sequence length
- ▶ Padding token = 0

Training sentences

- ▶ I am hungry
- ▶ you are tired
- ▶ we are happy
- ▶ they are sad
- ▶ the weather is nice

Training pairs

Input: (x_1, \dots, x_{n-1}) → Target: (x_1, \dots, x_n)

Teacher forcing: true previous tokens are known.

The model learns language from very simple sequences.

Self-Attention Layer: Class Setup

What is defined here

- ▶ Learnable weight matrices
- ▶ Head configuration
- ▶ No computation yet

Key parameters

- ▶ d_{model} : embedding dimension
- ▶ H : number of heads
- ▶ $d_k = d_{\text{model}}/H$

Self-attention layer (initialization)

```
1 class SelfAttention(nn.Module):  
2     def __init__(self, d_m, num_heads):  
3         super().__init__()  
4         assert d_model % num_heads == 0  
5         self.num_heads = num_heads  
6         self.d_k = d_m // num_heads  
7  
8         self.q_linear = nn.Linear(d_m, d_m)  
9         self.k_linear = nn.Linear(d_m, d_m)  
10        self.v_linear = nn.Linear(d_m, d_m)  
11        self.out_linear = nn.Linear(d_m, d_m)
```

This sets up the attention mechanism structurally.

The class defines *what can be learned*, not how it is used.

Self-Attention — Forward Pass in Code

Core computation

- ▶ Linear projections to Q, K, V
- ▶ Scaled dot-product attention
- ▶ Weighted value aggregation
- ▶ Merge heads + output projection

$$Z = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$

Self-attention (PyTorch, condensed)

```
1 def forward(self, x):
2     B, T, _ = x.shape
3     q = self.q(x).view(B, T, H, D).transpose(1, 2)
4     k = self.k(x).view(B, T, H, D).transpose(1, 2)
5     v = self.v(x).view(B, T, H, D).transpose(1, 2)
6
7     a = (q @ k.transpose(-2, -1)) / sqrt(D)
8     a = softmax(a, dim=-1)
9
10    z = (a @ v).transpose(1, 2).reshape(B, T, H*D)
11    return self.out(z)
```

This is the attention equation executed in code.

Training the Transformer

Training objective

- ▶ Predict the next token
- ▶ At every sequence position

$$\mathcal{L} = - \sum_{t=1}^n \log p_t(y_t)$$

Learned parameters

- ▶ Embeddings
- ▶ Attention projections
- ▶ FFN weights
- ▶ Output projection

Training loop (PyTorch)

```
1 criterion = CrossEntropyLoss(  
2     ignore_index=0)  
3 optimizer = Adam(  
4     model.parameters(), lr=1e-3)  
5  
6 for x, y in dataloader:  
7     optimizer.zero_grad()  
8     logits = model(x)  
9     loss = criterion(  
10         logits.view(-1, V), y.view(-1))  
11     loss.backward()  
12     optimizer.step()
```

Gradients flow through the entire Transformer.

Training is standard gradient-based optimization.

What the Model Predicts

Given input

“I am” → [1, 2]

Model output

- ▶ Probability over vocabulary
- ▶ One distribution per position

Prediction

$\arg \max p(\text{token}) \Rightarrow \text{“hungry”}$

Inference (greedy decoding)

```
1 model.eval()  
2 x = [I, am]  
3  
4 logits = model(x)  
5 next = argmax( logits[-1] )  
6  
7 x = append(x, next)
```

Key point

- ▶ Same model as during training
- ▶ No architecture change
- ▶ Only token selection differs

Generation is just repeated next-token prediction.

Why Run a Language Model Locally?

Cloud-based LLMs

- ▶ High performance
- ▶ No setup required
- ▶ External infrastructure

Typical examples:

- ▶ ChatGPT
- ▶ Claude
- ▶ Gemini

Local LLMs

- ▶ Full data privacy
- ▶ Offline capability
- ▶ Direct system integration

Use cases:

- ▶ Prototyping
- ▶ Teaching & learning
- ▶ Domain-specific tools

Modern LLMs are small and fast enough to run on a laptop.

Installing Ollama

What is Ollama?

- ▶ Local LLM runtime
- ▶ Optimized inference
- ▶ Simple CLI and API

Supported platforms

- ▶ Linux
- ▶ macOS
- ▶ Windows (WSL)

Install Ollama (Linux / macOS)

```
1 curl -fsSL https://ollama.com/  
      install.sh | sh
```

Verify installation

```
1 ollama --version
```

Ollama runs as a local service on localhost.

No Python, no GPU setup, no environment management.

Running a Local LLM with Ollama

Download a model

- ▶ Models are pulled on demand
- ▶ Stored locally

Examples:

- ▶ mistral
- ▶ llama3
- ▶ deepseek-r1

Run a model

```
1 ollama pull mistral
2 ollama run mistral
```

Example prompt

```
1 > Explain self-attention in
one paragraph.
```

Responses are generated locally.

```
(ropy) rpotthas@ofmws253 ~/all/e-ai_ml2/course/lectures/lec06 % ollama run mistral
>>> explain self attention in one paragraph
Self-attention, also known as scaled dot-product attention or multi-head attention, is a mechanism in deep learning that allows models to focus on different parts of the input sequence when processing information. It works by weighting the importance of each input element based on relationships with other elements. In essence, self-attention measures the similarity between every pair of input elements and uses these scores to compute a weighted sum. This mechanism is particularly useful in tasks such as natural language processing where understanding the relationships between words can greatly enhance performance. It helps models to selectively focus on relevant information while ignoring irrelevant details, thus improving their ability to capture complex patterns within the data.

>>> [end a message (/? for help)]
```

This is a full LLM running on your machine.

Streaming Responses from an LLM

Standard inference

- ▶ Request sent
- ▶ Model computes full response
- ▶ Response returned at once

Drawback

- ▶ No intermediate output
- ▶ Latency feels high

Streaming inference

- ▶ Tokens generated sequentially
- ▶ Output arrives chunk by chunk
- ▶ Immediate user feedback

Key idea

$\text{tokens}_1, \text{tokens}_2, \text{tokens}_3, \dots$

Streaming exposes the autoregressive nature of

Streaming from Ollama using curl

Ollama REST API

- ▶ Local HTTP service
- ▶ JSON-based requests
- ▶ Supports streaming

Key option

"stream": true

Streaming request with curl

```
1 curl -X POST http://localhost:11434/api/
      generate \
2 -H "Content-Type: application/json" \
3 -d '{
4   "model": "mistral",
5   "prompt": "Explain self-attention.",
6   "stream": true }'
```

The response arrives as a sequence of JSON objects.

Each chunk corresponds to newly generated tokens.

```
{"model":"mistral","created_at":"2025-12-29T13:38:01.545932Z","response":" Self","done":false}
{"model":"mistral","created_at":"2025-12-29T13:38:01.570159Z","response":"-","done":false}
{"model":"mistral","created_at":"2025-12-29T13:38:01.592394Z","response":"att","done":false}
{"model":"mistral","created_at":"2025-12-29T13:38:01.614036Z","response":"ention","done":false}
{"model":"mistral","created_at":"2025-12-29T13:38:01.635585Z","response":".","done":false}
```

Streaming from Ollama using Python

Why Python streaming?

- ▶ Interactive applications
- ▶ Live UIs
- ▶ Custom text processing

Mechanism

- ▶ HTTP response stream
- ▶ Incremental JSON decoding
- ▶ Append text chunks

Streaming response (Python)

```
1 import requests, json
2 url = "http://localhost:11434/api/generate"
3 data = {
4     "model": "mistral",
5     "prompt": "Explain self-attention.",
6     "stream": True }
7 r = requests.post(url, json=data, stream=True)
8 for line in r.iter_lines():
9     if line:
10         msg = json.loads(line)
11         print(msg["response"], end="")
```

Text is printed as soon as tokens arrive.

Streaming enables responsive, real-time LLM applications.

Accessing Ollama via a Local Streaming Server

Architecture overview

- ▶ Ollama runs the LLM locally
- ▶ Flask provides a lightweight API
- ▶ HTML/JS frontend streams responses

Starting the streaming server

Run the Flask server

```
1 cd code06  
2 python 5_flask_streaming.py
```

Open the UI in your browser

Access the interface

```
1 http://127.0.0.1:5000
```

Responses are streamed token by token via text/event-stream.

Browser → Flask server → Ollama

No cloud, no external services, full local control.

This setup turns a local LLM into an interactive application platform.

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Why Retrieval-Augmented Generation (RAG)?

Limitations of standalone LLMs

- ▶ Fixed knowledge at training time
- ▶ No access to private or local data
- ▶ Risk of hallucinated answers

Who is president of Deutscher Wetterdienst?

The President of the German Weather Service (Deutscher Wetterdienst, DWD) is actually a position that has not existed since 2005. Prior to then, the head of the organization was titled "President" or "Director General". However, in 2005, the role was redefined as a "Chairperson" or "Managing Director".

As of my knowledge cutoff, the current Chairperson (Präsident) of Deutscher Wetterdienst is Prof. Dr. Gerhard Adrian.

Please note that this information might be subject to change, and I'd be happy to update you if new developments occur!

LLMs answer from *learned parameters only*. **This can be completely off reality!**

Typical real-world needs

- ▶ Large code bases
- ▶ Technical documentation
- ▶ Evolving project knowledge

Key idea:

Bring the knowledge to the model.

Core Idea of Retrieval-Augmented Generation

Standard LLM workflow

Prompt → LLM → Answer

- ▶ No external knowledge
- ▶ No verification

Pure LLMs work well for general reasoning, language understanding, summarization, and creative text generation when the required knowledge is common and static. They perform poorly when accurate, up-to-date, proprietary, or highly technical domain knowledge is required, because they cannot verify facts or access external sources. In such cases, LLMs tend to hallucinate plausible but incorrect answers.

RAG workflow

Query → Retrieve → LLM → Answer

- ▶ Search relevant documents
- ▶ Inject context into prompt
- ▶ Grounded answers

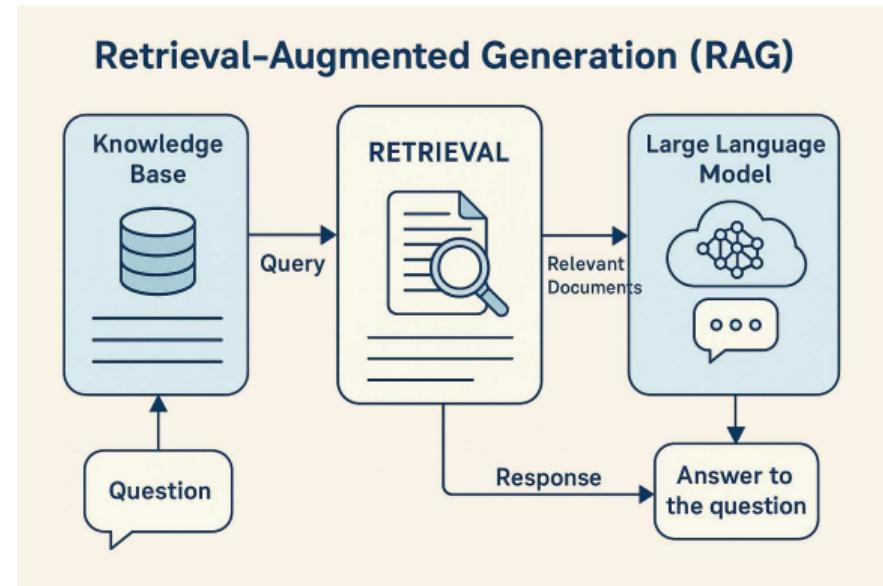
RAG Architecture Overview

RAG is more than just text search!

Main components

- ▶ Document collection
- ▶ Vector database
- ▶ Retriever
- ▶ Language model

Each component has a *clear role*.



Documents provide context for generation

Vector Databases and Transformer Embeddings

Text to vector mapping

Given a tokenized input sequence

$$(w_1, w_2, \dots, w_n)$$

a Transformer maps tokens to embeddings:

$$x_i = E(w_i) \in \mathbb{R}^d$$

After contextualization (self-attention):

$$h_i = \text{Transformer}(x_1, \dots, x_n)$$

A sentence or paragraph embedding is typically:

$$z = \frac{1}{n} \sum_{i=1}^n h_i$$

Vector database view

Each paragraph is stored as:

$$z_j \in \mathbb{R}^d$$

Similarity search uses a distance measure, e.g.

$$\text{sim}(z_q, z_j) = \frac{z_q \cdot z_j}{\|z_q\| \|z_j\|}$$

or equivalently:

$$\|z_q - z_j\|_2$$

Key point: The same embedding space is reused — now *outside* the LLM.

Elementary Vector Database — Setup

Toy example

- ▶ Small set of short sentences
- ▶ Different semantic topics
- ▶ Some paraphrases, some unrelated

Example sentences

```
1 sentences = [  
2     "The sun is shining brightly today.",  
3     "Heavy rain is falling over the city.",  
4     "Neural networks can learn patterns.",  
5     "Transformers use self-attention."  
6 ]
```

Each sentence will become one vector in a shared space.

We use this to build intuition before scaling to real documents.

$$\text{Sentence} \rightarrow z \in \mathbb{R}^d$$

Text to Vector Space

Sentence embedding

A sentence s_i is mapped to a vector

$$z_i \in \mathbb{R}^d$$

using a pretrained Transformer:

$$z_i = f_\theta(s_i)$$

In our example:

$$d = 384$$

Embedding matrix

For N sentences, we obtain:

$$Z = \begin{bmatrix} z_1^\top \\ z_2^\top \\ \vdots \\ z_N^\top \end{bmatrix} \in \mathbb{R}^{N \times d}$$

Each row corresponds to *one sentence*.

Similarity Search

Query embedding

A query sentence is mapped to:

$$z_q \in \mathbb{R}^d$$

Similarity to stored sentences is measured using cosine similarity:

$$\text{sim}(z_q, z_i) = \frac{z_q \cdot z_i}{\|z_q\| \|z_i\|}$$

See example `1_vector_db_elementary.ipynb` in the `code07/` directory carrying out transforms explicitly. The command `SentenceTransformer("all-MiniLM-L6-v2")` loads a pretrained Transformer-based sentence embedding model that maps each input sentence to a fixed-size vector $z \in \mathbb{R}^{384}$ capturing its semantic meaning. The model is publicly available at huggingface.co/sentence-transformers/all-MiniLM-L6-v2.

Similarity search (conceptual)

```
1 scores = cosine_similarity(  
2     query_embedding,  
3     embeddings )  
4  
5 top_k = argsort(scores)[-k:]
```

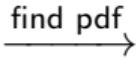
Result: Sentences with closest meaning are retrieved — no keywords needed.

Document Search via Embeddings

Goal

- ▶ Search *inside* a document
- ▶ Retrieve relevant pages or sections
- ▶ No keyword matching required

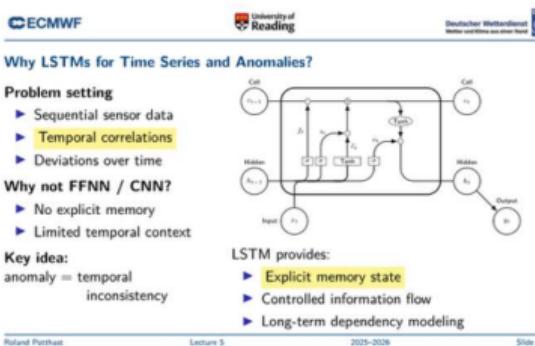
We use the same vector-space idea as in the elementary example.

"temporal pattern"  find pdf

Input

- ▶ One lecture PDF (Lecture 05)
- ▶ Pages as semantic units
- ▶ Natural-language queries

Output: Relevant pages from the PDF



From PDF Pages to Vectors

Decomposition

The PDF is split into pages:

$$\{\text{page}_1, \dots, \text{page}_N\}$$

Each page is embedded as:

$$z_i = f_\theta(\text{page}_i) \in \mathbb{R}^d$$

In our example:

$$d = 384$$

Embedding matrix

All pages form:

$$Z \in \mathbb{R}^{N \times d}$$

Each row corresponds to one PDF page.

Key point: Documents are represented as collections of vectors.

Extract text from PDF pages

```
1 for i, page in enumerate(reader.pages):
2     text = page.extract_text()
3     if text and len(text.strip()) > 50:
4         pages.append(text.strip())
5         page_ids.append(i + 1)
```

Embedding

```
1 page_embeddings = model.encode(
2     pages,
3     convert_to_numpy=True,
4     show_progress_bar=False)
```

Semantic Search inside a PDF

Query embedding

A user query is mapped to:

$$z_q \in \mathbb{R}^d$$

Similarity to page embeddings is computed via:

$$\text{sim}(z_q, z_i) = \frac{z_q \cdot z_i}{\|z_q\| \|z_i\|}$$

Top- k most relevant pages are retrieved.

Semantic page search

```
1 def search_pdf(query, model, pages, embeddings
2                 , page_ids, top_k=3):
3     q_emb = model.encode(
4         query, convert_to_numpy=True)
5     scores = []
6     for i, emb in enumerate(embeddings):
7         score = cosine_similarity(q_emb, emb)
8         scores.append((page_ids[i], score, pages
9                         [i]))
10    scores.sort(key=lambda x: x[1], reverse=
11                  True)
12    return scores[:top_k]
```

From Retrieval to Evidence

What we display

- ▶ Retrieved page number
 - ▶ Extracted text snippet
 - ▶ Original PDF page

This allows direct verification of the search result.

Why this matters

- ▶ Transparent retrieval
 - ▶ No hidden reasoning
 - ▶ Trust through inspectable sources

Query: What is a graph network

Result 1: Page 9 (similarity = 0.623)

Extracted text (snippet):

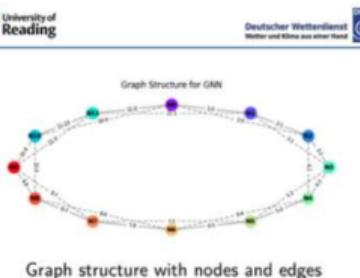
Why Graph Neural Networks? When FFNNs are not enough ►Data points are not independent ►Relations between entities matter ►Ordering is not fixed or meaningful
Typical examples ►Physical grids and meshes ►Sensor networks ►Molecules, social networks
Graph structure with nodes and edges Core idea ►Nodes exchange information ►Learning respects graph structure ►Inductive bias for relational data
Roland Potthast Lecture 5 2025–2026 Slide 8



Why Graph Neural Networks?

When FFNNs are not enough

- ▶ Data points are not independent
 - ▶ Relations between entities matter
 - ▶ Ordering is not fixed or meaningful



Next: **Retrieval Augmented Generation:** Retrieve first — generate later.

Why Do We Need FAISS?

Above approach

- ▶ Loop over all embeddings
- ▶ Compute similarity one by one
- ▶ Exact but very (!) slow

Cost of one query:

$$\mathcal{O}(N \cdot d)$$

Problem size

- ▶ N : number of stored vectors (pages, paragraphs, documents)
- ▶ d : embedding dimension (e.g. $d = 384$)

Each query compares against *all* vectors.

Scaling issue: Large N makes brute-force search infeasible.

Typical RAG library sizes

- ▶ **Personal projects:** 10^2 – 10^3 documents → $N \sim 10^4$ chunks
- ▶ **Team / institutional data:** 10^4 – 10^5 documents → $N \sim 10^6$ – 10^7 chunks
- ▶ **Enterprise-scale systems:** 10^6 + documents → $N \gg 10^7$ chunks

Chunking multiplies the number of stored vectors N .

FAISS — Vector Search at Scale

- ▶ Facebook AI Similarity Search
- ▶ Optimized nearest-neighbor search
- ▶ Designed for large vector collections

Stores vectors:

$$z_i \in \mathbb{R}^d$$

Key idea

- ▶ Build an index once
- ▶ Query many times
- ▶ Fast top- k retrieval

Search replaces explicit loops.

Hierarchical search in FAISS

FAISS accelerates nearest-neighbor search by introducing a *coarse-to-fine hierarchy* in vector space.

Step 1: Coarse partitioning

The vector space is partitioned into M regions using representative centroids:

$$\{c_1, \dots, c_M\}, \quad c_j \in \mathbb{R}^d.$$

Each stored vector z_i is assigned to its nearest centroid:

$$z_i \mapsto c(z_i).$$

FAISS in Practice

Index construction

- ▶ Choose distance metric
- ▶ Add all embeddings
- ▶ Index lives in memory or on disk

Common choice: IndexFlatL2

Step 2: Query routing

For a query vector z_q , FAISS finds the closest centroids:

$$\mathcal{C}_q = \arg \underset{L}{\text{top}} \|z_q - c_j\|_2, \quad L \ll M.$$

Build and query FAISS index

```
1 import faiss
2 d = page_embeddings.shape[1]
3 index = faiss.IndexFlatL2(d)
4
5 index.add(page_embeddings)
6
7 distances, indices = index.search(
8     query_embedding, k)
```

Step 3: Local search

Exact distances are computed only for vectors stored in the selected regions:

$$z_i \text{ with } c(z_i) \in \mathcal{C}_q.$$

Streaming LLM Responses

Standard LLM interaction

prompt → full response

User waits until generation is finished.

Streaming interaction

prompt → $(\delta_1, \delta_2, \dots)$

Partial tokens are delivered incrementally.

Recall that

LLMs generate tokens incrementally!

What streaming changes

- ▶ Faster perceived response
- ▶ Progressive rendering
- ▶ Interactive user experience

What streaming does not change

- ▶ Model architecture
- ▶ Reasoning capability
- ▶ Final content

OpenAI Streaming

Streaming API call

- ▶ Same prompt structure
- ▶ `stream=True`
- ▶ Tokens arrive as **deltas**

Each chunk contains new text:

$$\delta_k \subset \text{response}$$

There are several older OpenAI interface versions that language models may still suggest. Be careful: to avoid deprecated APIs, it is often necessary to explicitly provide a current code template.

OpenAI streaming in Jupyter

```
1 stream = client.chat.completions.create(  
2     model="gpt-4o-mini", messages=[...],  
3     stream=True )  
4  
5 accumulated = ""  
6 handle = display(  
7     Markdown(""), display_id=True)  
8  
9 for chunk in stream:  
10    delta = chunk.choices[0].delta  
11    if delta.content:  
12        accumulated += delta.content  
13        handle.update(  
14            Markdown(accumulated))
```

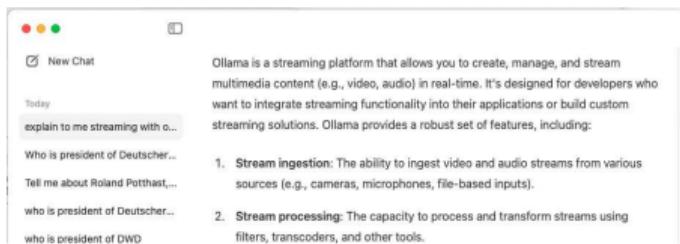
Streaming with Local LLMs (Ollama)

Local execution

- ▶ Models run on local hardware
- ▶ No external API calls
- ▶ Full data control

Streaming follows the same principle:

$$(\delta_1, \delta_2, \dots)$$



Advantages

- ▶ Privacy and compliance
- ▶ Offline usage - train or plane
- ▶ No usage-based cost

Trade-offs

- ▶ Smaller models
- ▶ Hardware dependent speed

List of all available models:
<https://ollama.ai/library>

Ollama Streaming in Python

Ollama streaming call

- ▶ Local HTTP interface
- ▶ Same message format
- ▶ Streaming enabled

Works with: `llama3`,
`mistral`, `mixtral`,
`deepseek-r1`

Ollama streaming in Jupyter

```
1 stream = ollama.chat( model="llama3",
2                         messages=[...], stream=True )
3
4 accumulated = ""
5 handle = display(
6     Markdown(""), display_id=True)
7
8 for chunk in stream:
9     if "content" in chunk["message"]:
10         accumulated += chunk["message"]\
11             ["content"]
12         handle.update(Markdown(accumulated))
```

RAG Implementation: Concrete Steps

Implemented components

- ▶ text chunking (files → chunks) ✓
 - ▶ sentence embeddings ✓
 - ▶ FAISS index ✓
 - ▶ metadata tracking ✓
 - ▶ LLM query with context ✓

Execution order

1. build vector database (offline)
 2. embed user query
 3. retrieve top- k chunks
 4. assemble prompt context
 5. generate answer (optional streaming)

```
stream = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "system",
            "content": (
                "You are a technical assistant. "
                "Answer only using the provided context."
            ),
        },
        {
            "role": "user",
            "content": (
                "Context:\n"
                f"{context}\n"
                "Question:\n"
                f"{query}"
            ),
        },
    ],
)
```

Top- k Retrieval: What Do We Actually Get?

FAISS output

For a query embedding z_q , FAISS returns:

$$\{(i_1, d_1), (i_2, d_2), \dots, (i_k, d_k)\}$$

- ▶ i_j : index of a stored chunk
- ▶ d_j : distance or similarity score

These indices refer to:

$$z_{i_j} \leftrightarrow \text{chunk}_{i_j}$$

Important clarification

- ▶ FAISS returns *vectors*, not text
- ▶ Text is recovered via metadata lookup
- ▶ Ordering is by similarity score

At this stage:

- ▶ no LLM involved
- ▶ no generation
- ▶ no reasoning

From FAISS Indices to Text Chunks

Stored metadata per chunk

Each vector z_i is linked to:

- ▶ file path
- ▶ line range or section
- ▶ raw text content

This mapping is stored outside FAISS
(e.g. Python lists, JSON, databases).

Lookup step

For each returned index i_j :

$$i_j \rightarrow (\text{file, lines, text})$$

This produces a ranked list:

$$(\text{chunk}_1, \dots, \text{chunk}_k)$$

Result: Concrete, inspectable evidence.

Composing the Retrieval Context

Context construction

Retrieved chunks are:

- ▶ ordered by similarity
- ▶ concatenated into one context block

Typical structure:

- ▶ file reference
- ▶ optional score
- ▶ text snippet

Context size is explicitly limited .

Context assembly (ICON example)

```
1 context = ""
2 for r in retrieved_chunks:
3     context += (
4         f"[{r.file}], lines {r.start}-{r.end}]\n"
5         + r.text + "\n\n")
```

OpenSearch

What is OpenSearch?

OpenSearch is an open-source, distributed search and analytics engine.

It is designed to index and query large collections of documents with low latency.

Originally derived from Elasticsearch, OpenSearch is developed under the Apache 2.0 license.

Core capabilities

- ▶ full-text search (inverted index)
- ▶ metadata filtering and aggregation
- ▶ persistent, disk-based indices
- ▶ distributed execution across nodes
- ▶ **vector similarity search**

OpenSearch combines text-based and vector-based retrieval in a single system.

Google Search as Retrieval-Augmented Generation

Core idea

Retrieval-Augmented Generation (RAG) combines an LLM with an external information source.

In classical RAG, retrieval uses:

- ▶ vector databases (FAISS)
- ▶ local document collections

Google-based RAG

Here, retrieval is delegated to:

- ▶ Google Search
- ▶ live web documents

Same logical pipeline

Query → Retrieve →
Context → LLM → Answer

Key difference

- ▶ Local RAG: curated, controlled, static
- ▶ Google RAG: open, dynamic, up-to-date

Interpretation

The LLM does *not* search —
it *summarizes retrieved evidence*.

Search to LLM Pipeline

Pipeline steps

1. Issue search query
2. Retrieve top- k URLs
3. Scrape page text
4. Build context block
5. Ask LLM to answer

Answer =

LLM (Query, Web Evidence)

Important

No embeddings are required —
ranking by Search Engine .

Google–LLM pipeline (schematic)

```
1 urls = search_google(query)
2 texts = [ scrape_website(url),
3   for url in urls ]
4
5 prompt = f """
6 Answer the question using
7 the sources below.
8 Question: {query}
9 Sources: {texts}
10 """
11
12 response = client.chat.completions.
13   create( model="gpt-4o-mini",
14     messages=[{"role": "user", "content
15       :prompt}] )
```

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00				Joint Dinner	

What is Multimodal Artificial Intelligence?

Single-modality AI

- ▶ Classical ML processes one modality at a time
- ▶ Text-only LLMs operate on sequences of tokens
- ▶ Images, audio, or fields are handled separately

Limitation

- ▶ No explicit link between different data modalities
- ▶ Limited representation of real-world information

Multimodal AI

- ▶ Jointly processes multiple modalities
- ▶ Learns **relationships across modalities**
- ▶ Typical inputs:
 - ▶ text
 - ▶ images
 - ▶ audio
 - ▶ spatial or physical fields

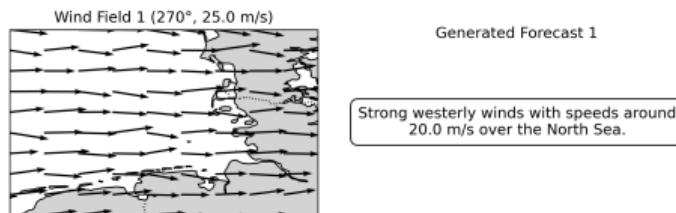
Key idea: Meaning emerges from the interaction of modalities .

Why Multimodal AI in Weather and Climate?

Nature of meteorological data

- ▶ Numerical model fields
- ▶ Maps and visualisations
- ▶ Radar and satellite images
- ▶ Textual forecasts and warnings

Weather information is inherently multimodal.



Human forecasters

- ▶ Integrate multiple information sources
- ▶ Combine perception and physical reasoning
- ▶ Translate observations into language

Key question: How can AI systems learn this integration?

Example A: Symbolic Multimodality

Basic idea

- ▶ Start from a physical wind field
- ▶ Extract a small set of **symbolic descriptors**
- ▶ Use a language model to generate text

The model does *not* see images or full fields.

Representation pipeline

- ▶ Numerical wind field
- ▶ → symbolic spatial summary
- ▶ → text prompt
- ▶ → language model

Multimodality is achieved through **representation design**.

From Wind Fields to Symbols

Physical input

- ▶ Two-dimensional wind field
- ▶ Zonal and meridional components (u, v)
- ▶ Defined on a spatial grid

This information is high-dimensional and continuous.

Symbolic reduction

- ▶ Mean wind speed
- ▶ Dominant wind direction
- ▶ Categorical intensity classes

Use a small number of *interpretable* features are retained.

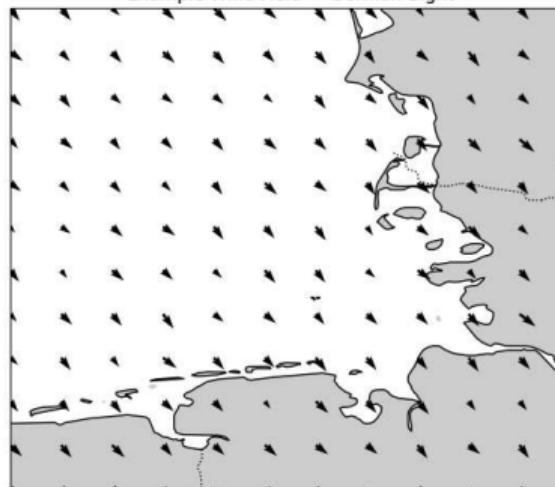
==== SYMBOLIC INPUT ===

Wind field summary:
Mean speed: 9.1 m/s
Speed variability: low
Dominant direction: northwesterly (316°)
Directional spread: ±5.8°
Stronger winds in the northern sector.
Overall intensity: moderate

==== TARGET TEXT ===

Northwesterly winds over the German Bight with moderate intensity and speeds around 10 m/s.

Example Wind Field — German Bight



Text Generation from Symbolic Descriptors

LLM input

- ▶ Short symbolic prompt
- ▶ Encodes physical properties
- ▶ Passed directly to the language model

Mean wind speed: 15.2 m/s

Dominant direction: northwesterly

Region: North Sea

LLM output

- ▶ Natural-language forecast text
- ▶ Generated solely from symbolic input

Strong northwesterly winds with speeds around 15 m/s over the North Sea.

Example A: Core Implementation Idea

Step 1: Symbolic preprocessing

- ▶ Reduce the wind field to a few key descriptors
- ▶ Physics-informed, human-designed
- ▶ Low-dimensional and interpretable

This step encodes *what matters* for the language model.

Step 2: Prompt-based text generation

Symbolic prompt and text generation

```
1 summary = f"Mean speed: {vmean:.1f} m/s, "
2 summary += f"Direction: {direction}"
3
4 inputs = tokenizer(
5     summary,
6     return_tensors="pt"
7 )
8
9 outputs = model.generate(**inputs)
```

In this example the LLM sees only text, not physics .

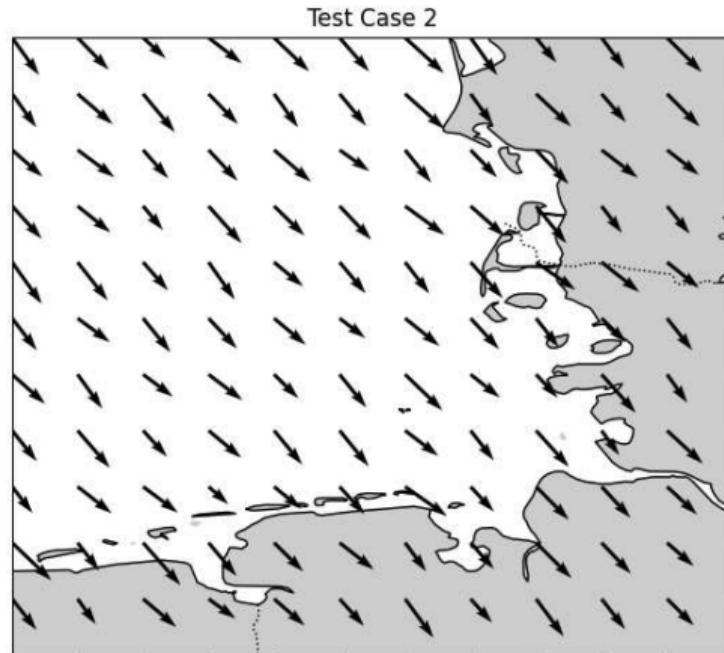
Example A: Result on a Concrete Wind Field

Physical input

- ▶ Synthetic wind field over the North Sea
- ▶ Used to compute symbolic descriptors
- ▶ Image shown here for illustration

Generated text (Example A)

- ▶ Language model sees only **symbolic input**
- ▶ No image is passed to the model



Example B: Visual Multimodality

Key difference to Example A

- ▶ No symbolic preprocessing
- ▶ No explicit speed or direction given
- ▶ The model receives an **image only**
- ▶ Converted to RGB pixel values

All physical structure must be inferred visually.

```
# Draw the canvas
fig.canvas.draw()

buf = np.asarray(fig.canvas.buffer_rgba())
image = buf[..., :3]    # drop alpha channel
```

Normalize to [0, 1]

Multimodal processing pipeline

- ▶ Wind field rendered as image
- ▶ Vision encoder extracts features
- ▶ Features are mapped to language space
- ▶ Language model generates text

This is *true* multimodal learning.

```
# -----
# Render the wind field as an image
# This image is the ONLY input seen by
# the multimodal model.
# -----
img = render_wind_image(LON, LAT, U, V)
```

From Image Tensor to Vision Embedding

Vision encoder (ViT)

- ▶ Input: image tensor (1, 3, 224, 224)
- ▶ Image is split into patches
- ▶ Self-attention models spatial relations

The vision model extracts a high-level representation of the entire image.

Forward pass and feature extraction

Vision Transformer

```
1 with torch.no_grad():
2     vision_out = vit(
3         pixel_values=pixel_values )
4
5 # Global image representation
6 vision_feat = (
7     vision_out.last_hidden_state[:,0,:]
8 ) # CLS token
```

The CLS token summarizes the full image content.

$$\text{img} \in \mathbb{R}^{224 \times 224 \times 3} \rightarrow \text{vision_feat} \in \mathbb{R}^{1 \times 768}$$

CLS token $\in \mathbb{R}^{768}$ — a learned global image representation

From Vision Embedding to Language Model Input

The modality gap

- ▶ Vision encoder outputs visual embeddings
- ▶ Language models expect text embeddings
- ▶ Both live in different representation spaces

A learned projection is required to connect vision and language.

Vision–language bridge

Project vision features into language space

```
1 # vision features -> embedding
2 encoder_embed = bridge( vision_feat
3     ).unsqueeze(1)
4
5 # Pass embeddings to the T5 encoder
6 outputs = t5(
7     inputs_embeds=encoder_embed,
8     labels=labels )
```

The language model receives embeddings, not tokens.

$$\text{CLS} \in \mathbb{R}^{768} \xrightarrow{\text{Linear projection}} \text{encoder_embed} \in \mathbb{R}^{1 \times 1 \times 512}$$

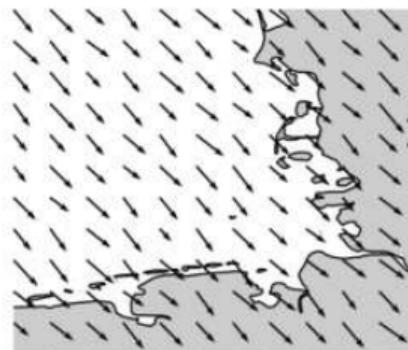
Example B: Results on Unseen Wind Fields

Model input

- ▶ Wind field rendered as an image
- ▶ No symbolic or numerical input
- ▶ Image only, unseen during training

The model infers direction and strength **from visual structure alone.**

Input wind field
Base direction: 315°, mean speed: 25.2 m/s



Northwesterly winds over the German Bight with strong intensity and speeds around 20 m/s.

Vision Transformer: Patch Embedding

Image representation

An **input image** is represented as

$$\mathbf{x} \in \mathbb{R}^{H \times W \times 3}$$

- ▶ H, W : image height and width (pixels)
- ▶ 3: RGB color channels

The image is split into non-overlapping patches of size $P \times P$.

Number of patches:

$$N = \frac{H \cdot W}{P^2}$$

Patch embedding

Each image patch is flattened:

$$\mathbf{x}_p \in \mathbb{R}^{P^2 \cdot 3}$$

and projected into an embedding space:

$$\mathbf{x}_p \xrightarrow{E} \mathbf{z}_p \in \mathbb{R}^D$$

The full input sequence is:

$$\mathbf{Z}_0 = [\mathbf{z}_{\text{CLS}}, \mathbf{z}_1, \dots, \mathbf{z}_N] \in \mathbb{R}^{(N+1) \times D}$$

D is the **transformer embedding dimension**.

Vision Transformer: Self-Attention

Transformer processing

The patch embedding sequence is processed by L stacked transformer layers:

$$\mathbf{Z}_{\ell+1} = \text{Transformer}(\mathbf{Z}_\ell), \quad \ell = 0, \dots, L-1$$

Each layer consists of:

- ▶ Multi-head self-attention
- ▶ Feed-forward networks
- ▶ Residual connections $x + \mathcal{F}(x)$
- ▶ Layer normalization

CLS token as global image embedding

Self-attention enables each patch to attend to all other patches.

After the final layer, the CLS token is:

$$\mathbf{z}_{\text{CLS}}^{(L)} \in \mathbb{R}^D$$

This vector represents the entire image and can be used for:

- ▶ Image classification
- ▶ Regression tasks
- ▶ Multimodal conditioning of language models

Radar Reflectivity as Multimodal Input

Weather radar data

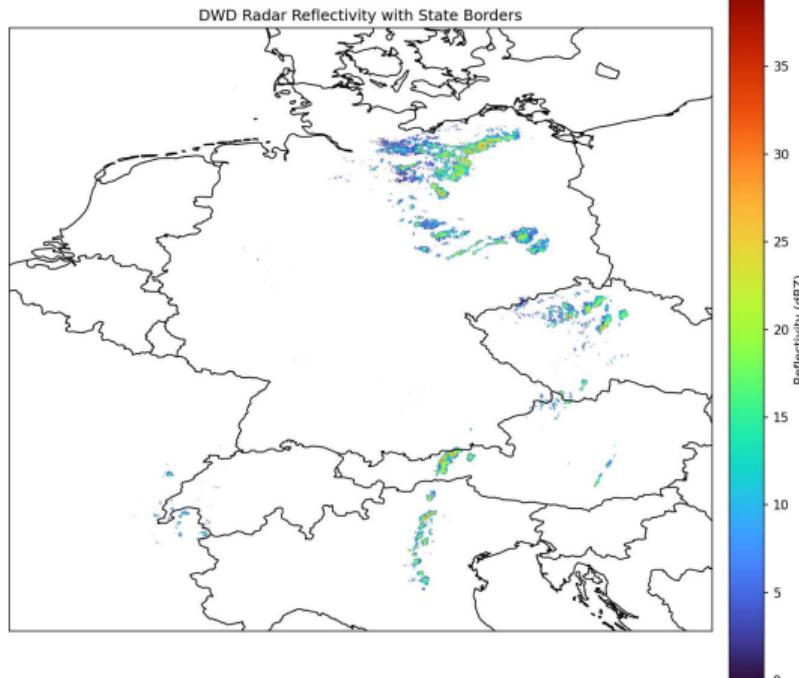
- ▶ Active remote sensing (microwave)
- ▶ Measures backscattered power
- ▶ Proxy for precipitation intensity

Radar reflectivity is expressed in

$$\text{dBZ} = 10 \log_{10}(Z)$$

with Z the radar reflectivity factor.

Radar images are routinely interpreted visually by forecasters.



DWD radar composite with state borders. Colors indicate reflectivity intensity (dBZ).

From Radar Product to AI-Readable Image

Operational radar data

- ▶ Provided as HDF5 files
- ▶ Contains data and metadata
- ▶ Scaling and masking required

Physical decoding:

- ▶ apply gain and offset
- ▶ mask undetectable values
- ▶ remove non-physical signals

Result: a georeferenced reflectivity field.

Decode radar reflectivity

```
1 raw = f["dataset1/data1/data"][:]
2
3 gain    = what.attrs["gain"]
4 offset  = what.attrs["offset"]
5
6 data = raw.astype(np.float32) *
         gain + offset
7 data[(raw == 0) | (raw == 65535) |
         (data < 0)] = np.nan
```

This step preserves physical meaning before visualization.

Radar as Image Modality

Why visualization matters

- ▶ Radar fields are spatial data
- ▶ Interpretation relies on patterns
- ▶ Humans read radar as images

For multimodal AI, the radar field is therefore converted into a

visual representation :

- ▶ color encodes intensity (dBZ)
- ▶ white indicates missing / no signal
- ▶ borders provide spatial context

The **image** becomes the

AI input modality.

Render radar reflectivity map

```
1 masked = np.ma.masked_invalid(data)
2 plt.figure(figsize=(12, 10))
3 ax = plt.axes(projection=ccrs.
               PlateCarree())
4 ax.set_extent([3, 17, 44, 56])
5 im = ax.imshow(
6     masked, origin="lower",
7     cmap=cmap, vmin=0, vmax=40,
8     transform=ccrs.PlateCarree() )
9
10 ax.add_feature(cfeature.BORDERS)
11 ax.coastlines(resolution="10m")
12 plt.colorbar(im, ax=ax, label=
               "Reflectivity (dBZ)")
13 plt.savefig("radar_map_germany.png",
              dpi=150)
```

Multimodal AI Interpretation of Radar Images

Multimodal model input

The AI model receives:

- ▶ a radar image (visual modality)
- ▶ a textual instruction (language modality)

The model jointly reasons over:

- ▶ spatial precipitation patterns
- ▶ reflectivity intensity gradients
- ▶ mesoscale structure

The **output** is a natural-language meteorological interpretation.

Multimodal AI request

```
1 query = "Please interpret this
          radar reflectivity image ..."
2 response = client.chat.completions.
            create(
3   model="gpt-4-turbo",
4   messages=[{ "role": "user",
5     "content": [
6       { "type": "text", "text": query
7     },
8     { "type": "image_url",
9       "image_url": { "url": f"data:
          image/png;base64,{encoded}"
10      } } ] } ],
10   max_tokens=800 )
```

Example: AI-Generated Radar Interpretation

Input to the model

- ▶ Radar reflectivity image (dBZ)
- ▶ Geographical context (Germany)
- ▶ Task-oriented meteorological prompt

The AI has no direct access to:

- ▶ numerical radar grids
- ▶ timestamps or motion
- ▶ physical radar equations

This radar reflectivity image shows precipitation distribution and intensity across Germany and parts of its neighboring countries, indicated by different colors reflecting different reflectivity values measured in dBZ (decibels of Z). Here's a detailed analysis:

[...] Stratiform Rainfall: The majority of the precipitation patterns, especially the widespread areas with uniform green shades in the northern and southwestern parts, suggest stratiform precipitation structures. These are typically smooth and consistent patterns associated with frontal systems or broad-scale upward motions that produce persistent and uniform rainfall.

Capabilities and Limitations of Multimodal Radar Interpretation

What the model can do well

- ▶ Identify precipitation regions
- ▶ Distinguish weak vs. strong reflectivity
- ▶ Recognize spatial patterns
- ▶ Describe convective vs. stratiform structures

This closely mirrors human visual interpretation.

Key limitations

- ▶ No access to raw radar measurements
- ▶ No temporal information (single snapshot)
- ▶ No physical thresholds or calibration
- ▶ Interpretation depends on visualization choices

The model provides *descriptive insight*, not quantitative meteorological analysis.

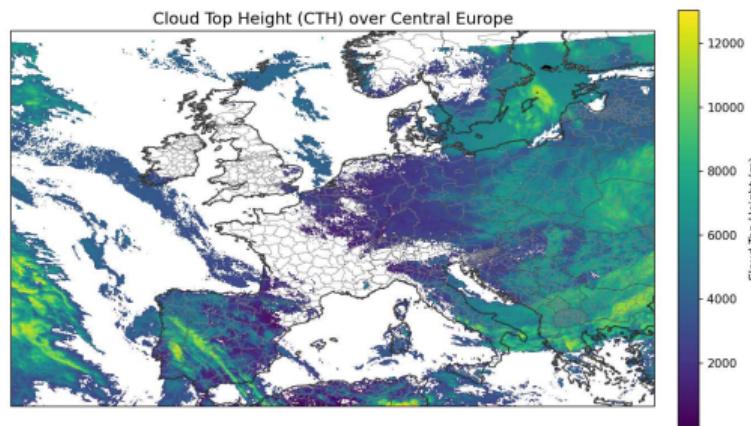
Cloud Top Height as Multimodal Input

What is Cloud Top Height (CTH)?

- ▶ Satellite-derived cloud product
- ▶ Estimates height of the upper cloud boundary
- ▶ Derived from thermal infrared observations

CTH is typically expressed in meters above sea level and provides information on:

- ▶ vertical cloud structure
- ▶ convective depth
- ▶ storm intensity



Satellite-derived Cloud Top Height (CTH) over Central Europe. Higher values correspond to deeper cloud systems.

High CTH values often indicate **deep convection**.

Accessing Cloud Top Height (CTH) Data

Operational satellite product

- ▶ Provided by DWD Open Data
- ▶ Derived from geostationary satellites
- ▶ Updated at regular intervals

CTH data is distributed as:

- ▶ compressed NetCDF files (.nc.bz2)
- ▶ regular latitude-longitude grids
- ▶ physical units: meters

The latest available file is selected automatically.

Download latest CTH product

```
1 base_url = (
2     "https://opendata.dwd.de/weather/"
3     "satellite/clouds/CTH/")
4 response = requests.get(base_url)
5 soup = BeautifulSoup(response.text, "
6         html.parser")
7 cth_files = sorted([ link.get("href")
8     for link in soup.find_all("a")
9     if link.get("href", "").endswith("."
10        nc.bz2") ])
11 my_f = cth_files[-1]
12 url = urljoin(base_url, my_f)
13 with open("cth.nc.bz2", "wb") as f:
14     f.write(requests.get(url).content)
```

Multimodal AI Interpretation of Cloud Top Height

Model input

The AI model receives:

- ▶ a Cloud Top Height image
- ▶ a task-oriented text prompt

The task focuses on:

- ▶ identifying regions of high cloud tops
- ▶ distinguishing deep convection from stratiform clouds
- ▶ describing large-scale cloud structures

Submit CTH image to multimodal LLM

```
1 query = "This is a satellite-derived
Cloud Top Height image [...]"
2 response = client.chat.completions.
    create(
3     model="gpt-4-turbo",
4     messages=[{"role": "user",
5         "content": [
6             {"type": "text", "text": query},
7             {"type": "image_url", "image_url": {
8                 "url": f"data:image/png;base64,{encoded}" }} ] }],
9     max_tokens=800,)
```

Multimodal AI in Weather: Comparison and Takeaways

Four multimodal examples

- ▶ **Synthetic wind fields (A)**
 - ▶ physical feature extraction
 - ▶ numerical fields → text
- ▶ **Synthetic wind fields (B)**
 - ▶ image-based representation
 - ▶ learned visual perception
- ▶ **Radar reflectivity**
 - ▶ precipitation structure
 - ▶ surface-reaching hydrometeors
- ▶ **Cloud Top Height (CTH)**
 - ▶ vertical cloud structure
 - ▶ deep convection indicators

Key insights

- ▶ Multimodal AI reasons over images and text
- ▶ Physical meaning enters via representation
- ▶ Visualization choices matter
- ▶ Models provide *interpretation*, not physics
- ▶ How Vision Transformers (ViT) work

Outlook

- ▶ combining multiple modalities
- ▶ adding temporal context
- ▶ integrating AI with NWP systems

Multimodal Models Available at OpenAI

GPT-4 family

- ▶ First widely deployed multimodal models
- ▶ Text and image input
- ▶ Text output
- ▶ Strong vision-language reasoning

Typical use cases:

- ▶ image interpretation
- ▶ document understanding
- ▶ visual question answering

Used extensively in early multimodal weather and geoscience applications.

GPT-5 family

- ▶ Current OpenAI foundation model generation
- ▶ Native multimodal reasoning
- ▶ Text and image input
- ▶ Text output with improved reasoning depth

Key characteristics:

- ▶ stronger cross-modal alignment
- ▶ improved robustness and consistency
- ▶ better handling of complex visual scenes

Multimodal AI on IONOS

IONOS AI Model Hub

Provides a curated set of foundation models via an OpenAI-compatible API endpoint.

Available model categories

- ▶ Text-only language models
- ▶ Embedding models
- ▶ Image generation models
- ▶ **One vision–language model**

Most models remain text-only.

Multimodal image understanding

Mistral Small 24B

- ▶ Input: text + image
- ▶ Output: text
- ▶ Tool calling supported
- ▶ OpenAI-compatible chat schema

Successfully used to interpret:

- ▶ radar reflectivity images
- ▶ cloud top height (CTH) maps

Selective but production-grade multimodality.

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Lecture 9 — Diffusion and Flexible Graph Networks

From prediction to distributions

- ▶ Classical ML: predict a single output
- ▶ Reality: many plausible outcomes
- ▶ We need models that *sample*

Weather and climate are inherently:

- ▶ stochastic
- ▶ high-dimensional
- ▶ uncertain

Two complementary ideas

- ▶ Diffusion models learn how to turn noise into structure
- ▶ Graph neural networks learn from sparse, irregular observations

Key message

- ▶ Learn *distributions*, not just functions
- ▶ Learn *structure*, not just grids

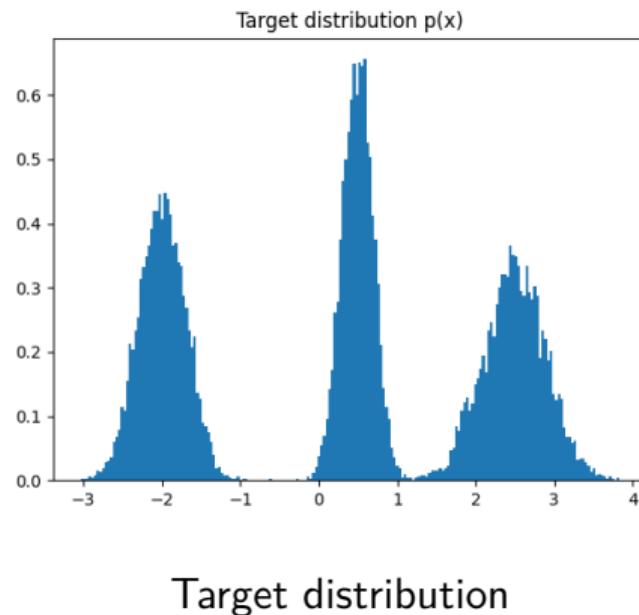
Learning to Sample a Distribution

Regression is not enough

- ▶ Regression predicts a single value
- ▶ Many systems have multiple valid outcomes
- ▶ We need to model and sample the uncertainty !
- ▶ We want samples: $x \sim p(x)$

Key idea

- ▶ Learn a *mapping from noise to data*
- ▶ Noise represents uncertainty



Sampling via a Neural Network

Neural sampler: noise → data

Neural generator mapping noise to samples

```
1 class Generator(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.net = nn.Sequential(
5             nn.Linear(1, 64),
6             nn.ReLU(),
7             nn.Linear(64, 64),
8             nn.ReLU(),
9             nn.Linear(64, 1) )
10
11     def forward(self, z):
12         return self.net(z)
```

The network transforms white noise into samples.

What is different from regression?

- ▶ No target output for a given input
- ▶ Input is random noise
- ▶ Only the *distribution* matters

Interpretation

- ▶ $z \sim \mathcal{N}(0, 1)$
- ▶ $x = f_\theta(z) \sim p_\theta(x)$

```
z = torch.randn(N, 1) # noise
x = G(z)           # samples
```

Training without Targets: Distribution Matching

Key problem

There is no correct output x for a given noise input z .

Solution: compare distributions

Differentiable distribution loss (1D Wasserstein)

```
1 def wasserstein_1d(x_gen, x_data):  
2     xg, _ = torch.sort(x_gen.view(-1))  
3     xd, _ = torch.sort(x_data.view(-1))  
4     return torch.mean((xg - xd)**2)
```

What happens here?

- ▶ Draw samples from both distributions
- ▶ Sort them by value
- ▶ Compare *quantiles*

Interpretation

- ▶ We do not match samples
- ▶ We match ranks / transport

This turns sampling into a learnable problem.

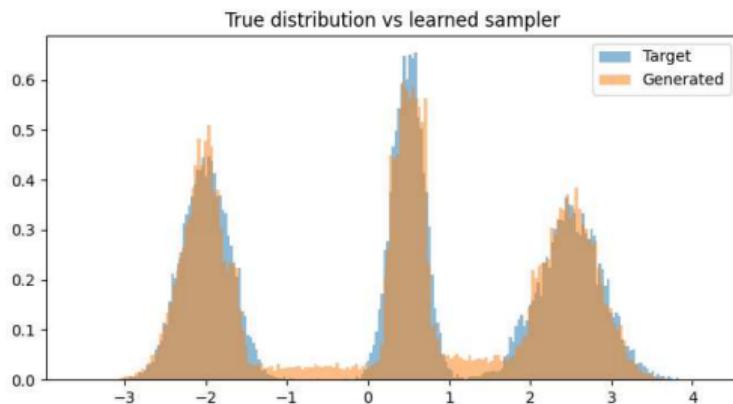
Learning to Sample a Distribution

Training result

- ▶ Network maps noise to samples
- ▶ Generated samples follow $p(x)$
- ▶ Multimodal structure is recovered

Key point

- ▶ No regression target was used
- ▶ Only a distribution loss



Target distribution and generated samples

This already enables stochastic modeling.

Why One-Step Sampling Is Not Enough

What we did so far

- ▶ Learn a direct map: noise → data
- ▶ Works well in low dimensions
- ▶ Works for simple distributions

But this has limits

- ▶ High-dimensional distributions are complex
- ▶ Direct mapping becomes unstable
- ▶ Hard to represent fine structure

Key idea

- ▶ Do not sample in one step
- ▶
- ▶ Use many small, simple steps

Intuition

- ▶ Large transport is hard
- ▶ Small corrections are easy

This leads to diffusion models .

Forward Diffusion: Adding Noise Step by Step

Forward process

We gradually destroy structure by adding noise:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1)$$

- ▶ $t = 0$: clean data
- ▶ $t \uparrow$: increasing noise
- ▶ $t = T$: pure noise

Important properties

- ▶ Process is fixed
- ▶ No learning involved
- ▶ Fully known statistics

Why do this?

- ▶ Generates training data
- ▶ Connects data to noise

This defines the learning problem.

Reverse Diffusion: Learning to Denoise

Idea: Train a network to

remove a *small amount of noise*:

$$x_t \longrightarrow x_{t-1}$$

Denoising step (learned reverse process)

```
1 # x_t : noisy signal at step t
2 # model learns to predict a less
   noisy version
3
4 for t in reversed(range(1, T)):
5     x = model(x, t) # denoise one step
```

Training principle

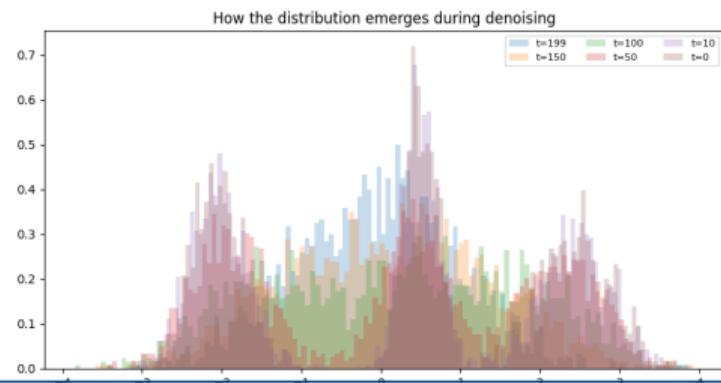
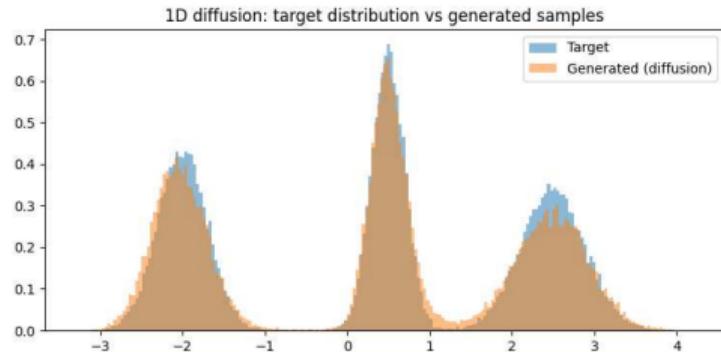
- ▶ Start from clean signal x_0
- ▶ Add noise step by step
- ▶ Train the model to undo each step

Key point

- ▶ Same network for all steps
- ▶ Only local corrections

Each step is simple and stable.

Diffusion in Action: Example Result



What we see

- ▶ Start from pure noise
- ▶ Apply learned denoising steps
- ▶ End up with valid samples

Key observation

- ▶ Samples follow the target distribution
- ▶ Stochastic but structured

This is sampling by refinement , not regression.

Forward Diffusion Process

Let $x_0 \sim p_{\text{data}}(x)$ be a data sample.

The forward diffusion process is a Markov chain:

$$q(x_{1:T} | x_0) = \prod_{t=1}^T q(x_t | x_{t-1})$$

with Gaussian transitions

$$q(x_t | x_{t-1}) = \mathcal{N}\left(x_t; \sqrt{1 - \beta_t} x_{t-1}, \beta_t I\right),$$

with $\beta_t \in (0, 1)$.

Equivalently, each step can be written as

$$x_t = \sqrt{1 - \beta_t} x_{t-1} + \sqrt{\beta_t} \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, I).$$

Consequences

- ▶ Mean is damped at each step
- ▶ Variance increases monotonically
- ▶ Signal-to-noise ratio decreases

After many steps:

$$x_T \approx \mathcal{N}(0, I)$$

This connects data to pure noise .

Closed-Form Forward Diffusion

Because all forward steps are Gaussian, the marginal distribution of x_t given x_0 has a closed form:

$$q(x_t | x_0) = \mathcal{N}\left(x_t; \sqrt{\bar{\alpha}_t} x_0, (1 - \bar{\alpha}_t)I\right)$$

with cumulative noise factor

$$\bar{\alpha}_t = \prod_{s=1}^t (1 - \beta_s).$$

Equivalently, sampling can be written as

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, \quad \epsilon \sim \mathcal{N}(0, I).$$

Interpretation

- ▶ Signal amplitude decays with $\sqrt{\bar{\alpha}_t}$
- ▶ Noise amplitude grows with $\sqrt{1 - \bar{\alpha}_t}$
- ▶ t directly controls noise level

As $t \rightarrow T$:

$$\bar{\alpha}_t \rightarrow 0 \quad \Rightarrow \quad x_t \sim \mathcal{N}(0, I).$$

This matches the example exactly.

Reverse Diffusion Process

The reverse process aims to invert the forward diffusion:

$$p_{\theta}(x_{0:T}) = p(x_T) \prod_{t=1}^T p_{\theta}(x_{t-1} | x_t),$$

with $p(x_T) = \mathcal{N}(0, I)$.

Each reverse step is modeled as a Gaussian:

$$p_{\theta}(x_{t-1} | x_t) = \mathcal{N}(x_{t-1}; \mu_{\theta}(x_t, t), \Sigma_t),$$

where μ_{θ} is predicted by a neural network.

The variance Σ_t is usually fixed or predefined.

What is learned?

- ▶ Only the conditional mean
- ▶ One network for all t
- ▶ Local denoising steps

Key idea

Each reverse step removes
a small amount of noise .

This turns sampling into a
sequence of simple corrections.

Why Diffusion Sampling Is Correct

Assume a scalar Gaussian data distribution:

$$x_0 \sim \mathcal{N}(\mu_0, \sigma_0^2), \quad x_1 = x_0 + \epsilon,$$

with $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

The optimal denoiser (MSE sense) is:

$$f^*(x_1) = \mathbb{E}[x_0 \mid x_1] = \frac{\sigma^2 \mu_0 + \sigma_0^2 x_1}{\sigma_0^2 + \sigma^2}.$$

This pulls the noisy sample toward μ_0 , but reduces variance.

Key observation

- ▶ Denoising alone shrinks variance
- ▶ Mean is biased toward μ_0

Therefore

To recover the full distribution, we must sample:

$$x_0 = f^*(x_1) + \sqrt{\sigma_{\text{post}}^2} \xi, \quad \xi \sim \mathcal{N}(0, 1).$$

This is exactly what diffusion does at every step.

Why Graph Networks?

The problem

- ▶ Data is sparse, irregular, incomplete
- ▶ Classical grids assume full coverage
- ▶ Interpolation rules are often fixed

In many applications, we only know values at

- ▶ a few locations,
- ▶ irregular positions,
- ▶ changing resolutions.

The graph perspective

- ▶ Nodes represent locations
- ▶ Edges represent influence / neighborhood
- ▶ Features store values and metadata

Key idea

- ▶ Geometry is encoded in the graph
- ▶ Learning happens locally
- ▶ Same model works on different domains

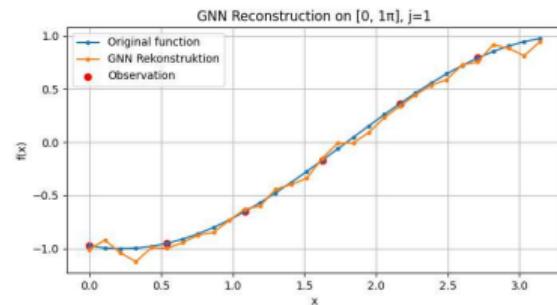
Graph Example: Reconstructing a Function from Sparse Data

Setup

- ▶ 1D spatial domain discretized into nodes
- ▶ True function $f(x)$ defined on all nodes
- ▶ Observations available only at few locations

Each node carries:

- ▶ observed value (or zero if missing)
- ▶ a binary observation mask



Sparse observations and GNN reconstruction

The task is to reconstruct $f(x)$ at *all* nodes.

Message Passing: How Information Flows

Local updates

Each node updates its state by combining:

- ▶ its own current value,
- ▶ information from neighboring nodes.

This update is *learned*, not prescribed.

Key mechanism

- ▶ Same update rule for all nodes
- ▶ Shared parameters across the graph
- ▶ Repeated over multiple layers

Interpretation

- ▶ Observations act as sources
- ▶ Information spreads gradually
- ▶ Missing values are inferred

Why this matters

- ▶ No fixed interpolation stencil
- ▶ Flexible neighborhood influence
- ▶ Scales to new domains

Message Passing: Minimal Code Example

One GNN message-passing layer (conceptual)

```
1 class GNNLayer(nn.Module):
2     def __init__(self, dim):
3         super().__init__()
4         self.self_lin = nn.Linear(dim, dim)
5         self.neigh_lin = nn.Linear(dim, dim)
6
7     def forward(self, x, edge_index):
8         row, col = edge_index
9         agg = torch.zeros_like(x)
10        agg.index_add_(0, row, x[col])
11        return F.relu(
12            self.self_lin(x) +
13            self.neigh_lin(agg) )
```

What this shows

- ▶ Neighbor features are summed
- ▶ Same weights used everywhere
- ▶ Graph defines information flow

Key insight

- ▶ No coordinates needed
- ▶ No stencil prescribed
- ▶ Structure comes from edges

Generalization: Same Network, Different Domain

What changes

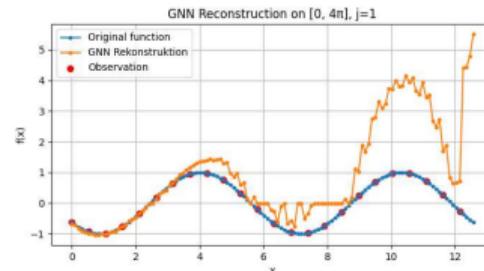
- ▶ Number of nodes N
- ▶ Physical domain length
- ▶ Resolution of the grid

What stays the same

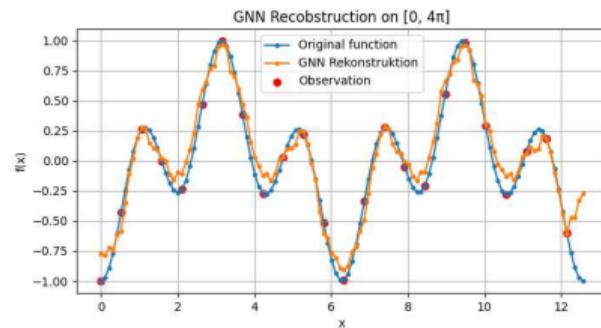
- ▶ Network architecture
- ▶ Trainable parameters
- ▶ Message-passing rule

Trained on $[0, \pi]$, applied on $[0, 4\pi]$.

Only the *graph* is rebuilt.



Failed reconstruction (absolute coordinates)



Successful (coordinate-free, graph-based)

Why PyTorch Lightning?

Problem with raw PyTorch

- ▶ Training loops get long and repetitive
- ▶ Device handling (CPU/GPU) is manual
- ▶ Logging and checkpoints are ad hoc

Code quickly becomes hard to read and maintain.

```
from torch_geometric.loader import DataLoader
```

```
# Create dataset
dataset = [generate_sample_graph(n_nodes=64, dn=8, k=3) for _ in range(200)]
loader = DataLoader(dataset, batch_size=16, shuffle=True)
```

```
# Train
```

```
model = GNNInterpolator()
trainer = pl.Trainer(max_epochs=200, logger=False, enable_checkpointing=False)
```

LightningModule: Clean Structure

Minimal Lightning module structure

```
1 class Model(pl.LightningModule):
2     def forward(self, x):
3         ...
4
5     def training_step(self, batch,
batch_idx):
6         ...
7         return loss
8
9     def configure_optimizers(self):
10        return torch.optim.Adam(self.
parameters())
```

Key idea

- ▶ No explicit training loop
- ▶ No device checks
- ▶ No boilerplate code

Focus stays on the model and the loss.

What Lightning Handles for You

Automatically

- ▶ CPU / GPU / multi-GPU
- ▶ Epoch and batch loops
- ▶ Gradient handling
- ▶ Logging

```
class GNNInterpolator(pl.LightningModule):  
    def __init__(self):  
        super().__init__()  
        self.gcn1 = GCNConv(2, n1) # 2 input features  
        self.gcn2 = GCNConv(n1, n2)  
        self.out = GCNConv(n2, 1)  
  
    def forward(self, data):  
        x = data.x_feat  
        x = F.relu(self.gcn1(x, data.edge_index))  
        x = self.gcn2(x, data.edge_index)  
        return self.out(x)
```

You still control

- ▶ Model architecture
- ▶ Loss functions
- ▶ Optimizers
- ▶ Data handling

Lightning enforces structure — not constraints.

```
def training_step(self, batch, batch_idx):  
    pred = self(batch)  
    mask = ~torch.isnan(batch.y_obs)  
    loss = F.mse_loss(pred[~mask], batch.y[~mask])  
    self.log("train_loss", loss)  
    return loss
```

Why PyTorch Geometric?

Graph-specific challenges

- ▶ Variable graph sizes
- ▶ Sparse connectivity
- ▶ Efficient message passing

Helps implementing this efficiently.

PyTorch Geometric

- ▶ Native graph data structures
- ▶ Optimized message passing
- ▶ Many standard GNN layers

Built directly on PyTorch tensors.

Graphs in PyTorch Geometric

Basic PyG data object

```
1 from torch_geometric.data import Data
2
3 data = Data(
4     x = node_features,
5     edge_index = edge_index,
6     y = targets
7 )
```

Key concept

- ▶ Nodes = rows of x
- ▶ Edges define message flow
- ▶ Graph size is flexible

No grid assumptions required.

PyTorch Geometric + Lightning

Why combine them?

- ▶ PyG: graph operations
- ▶ Lightning: clean training

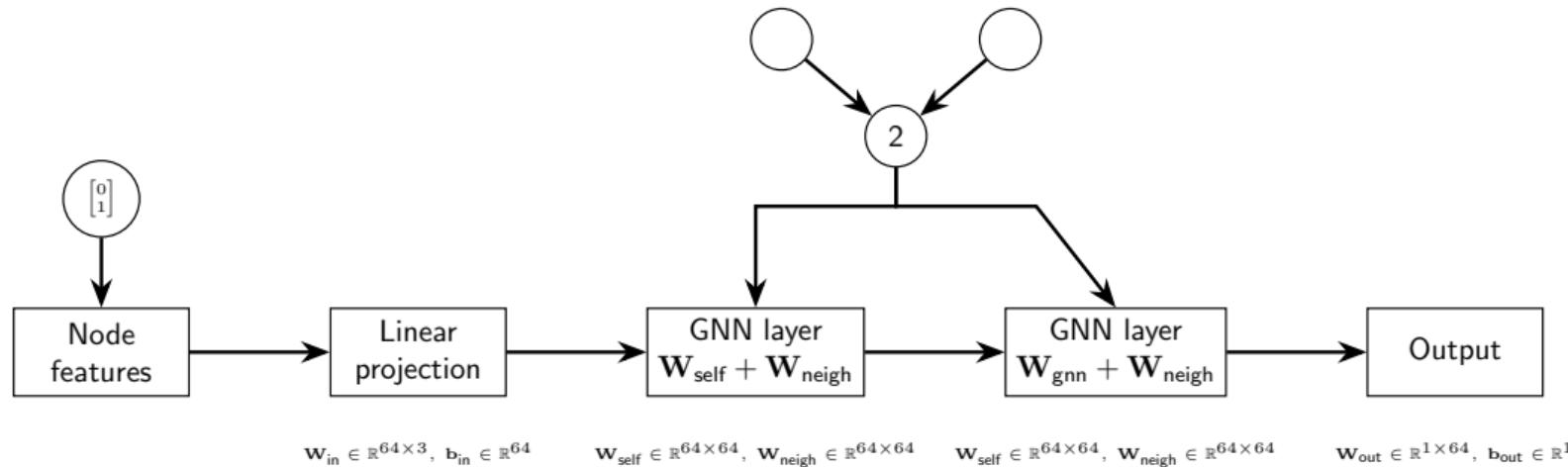
Each library does one thing well.

Result

- ▶ Clean, readable code
- ▶ Scales to large graphs
- ▶ Easy experimentation

Ideal for research and teaching.

Graph Neural Network Structure



Example structure with latent dimension 64 times number of nodes.

Graph Representation: Data vs Structure

Node data

$$X \in \mathbb{R}^{N \times d}$$

$$\mathbf{x}_i \in \mathbb{R}^d \quad (i = 1, \dots, N)$$

- ▶ N : number of nodes
- ▶ d : features per node
- ▶ One feature vector per node

Interpretation

- ▶ Rows correspond to nodes
- ▶ Columns correspond to features
- ▶ Node order is arbitrary

Graph structure

$$\text{edge_index} \in \mathbb{N}^{2 \times E}$$

Each column defines one directed edge:

$$(j \rightarrow i)$$

- ▶ Encodes connectivity only
- ▶ No numerical weights
- ▶ No trainable parameters

Separation of roles

- ▶ Data: X
- ▶ Structure: edge_index
- ▶ Learning: weights (later)

Node-wise Feature Transformation

Input

$$X \in \mathbb{R}^{N \times d}, \quad \mathbf{x}_i \in \mathbb{R}^d$$

Learned feature map

$$\mathbf{h}_i = \sigma(W\mathbf{x}_i + \mathbf{b})$$

$$W \in \mathbb{R}^{h \times d}, \quad \mathbf{b} \in \mathbb{R}^h$$

$$H \in \mathbb{R}^{N \times h}$$

Key property

Same transformation for every node.

Node-wise linear layer

```
1 self.lin = nn.Linear(d, h)
2 h = F.relu(self.lin(x))
```

What happens here

- ▶ Feature extraction
- ▶ Dimensionality change
- ▶ No neighbor interaction

What does *not* happen

- ▶ No graph usage
- ▶ No message passing
- ▶ No dependence on N

Message Passing: Injecting Graph Structure

Latent node features

$$H \in \mathbb{R}^{N \times h},$$

$$\mathbf{h}_i \in \mathbb{R}^h$$

Neighborhood aggregation

$$\mathbf{a}_i = \sum_{j \in \mathcal{N}(i)} \mathbf{h}_j,$$

$$A \in \mathbb{R}^{N \times h}$$

Graph-aware update

$$\mathbf{h}'_i = \sigma(W_{\text{self}}\mathbf{h}_i + W_{\text{neigh}}\mathbf{a}_i)$$

Graph affects *which terms are summed*, not the number of parameters.

Edge-based aggregation

```
1 row, col = edge_index
2 agg = torch.zeros_like(h)
3 agg.index_add_(0, row, h[col])
```

- ▶ Uses `edge_index` only
- ▶ No learnable graph parameters
- ▶ Permutation invariant

Effect

- ▶ Information exchange
- ▶ Local propagation
- ▶ One-hop interaction

Stacking Message Passing and Output Projection

Layer stacking

$$H^{(0)} \in \mathbb{R}^{N \times d},$$

$$H^{(1)} = \Phi(H^{(0)}, \text{edge_index}),$$

$$H^{(2)} = \Phi(H^{(1)}, \text{edge_index})$$

Each layer expands the receptive field:

- ▶ 1 layer: 1-hop neighbors
- ▶ 2 layers: 2-hop neighbors

Final node representation

$$H^{(L)} \in \mathbb{R}^{N \times h}$$

Output projection

$$\hat{y}_i = w^\top \mathbf{h}_i^{(L)} + b,$$

$$w \in \mathbb{R}^h, \quad b \in \mathbb{R}$$

$$\hat{y} \in \mathbb{R}^N$$

Final linear layer

```
1 self.out = nn.Linear(h, 1)
2 y_hat = self.out(h).squeeze(-1)
```

Key properties

- ▶ Same output map for all nodes
- ▶ Independent of graph size
- ▶ Fully permutation invariant

Lecture 9 — Big Picture

What we learned

- ▶ **Sampling** instead of regression
- ▶ Explicit modeling of **uncertainty**
- ▶ Generating distributions, not point estimates

Diffusion networks

- ▶ Noise → data via iterative refinement
- ▶ Correct sampling through stochastic reverse steps
- ▶ Strong theoretical foundation

Graph networks

- ▶ Flexible representation of structure
- ▶ Learning from sparse, irregular data
- ▶ Strong generalization across domains

Order, structure, efficiency

- ▶ Inductive bias through graphs and locality
- ▶ Separation of geometry and learning
- ▶ Scalable implementations with
 - ▶ PyTorch Lightning
 - ▶ PyTorch Geometric

Modern ML combines **sampling**, **structure**, and **efficient tooling** to build flexible, generalizable models.

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Why Function Calling Exists

- ▶ Pure text generation is insufficient
- ▶ Real systems need **actions**, not prose
- ▶ **To make the LLM productive,
it needs a link to reality!**
- ▶ Let it call functions!

Key idea

*The model should decide what to do,
the system should decide how to do it.*

Goal:

- ▶ separate **reasoning** from **execution**

```
{  
  "tool_call": {  
    "name": "<function name>",  
    "arguments": { ... }  
  }  
}
```

Typical problems:

- ▶ fragile JSON parsing
- ▶ ambiguous intent
- ▶ mixed explanation and action

Function calling introduces:

- ▶ explicit actions
- ▶ typed arguments
- ▶ machine-readable decisions

Tool Contracts and Schemas

A tool is defined by:

- ▶ name
- ▶ description
- ▶ input schema

Schemas specify:

- ▶ required arguments
- ▶ data types
- ▶ allowed structure

Important:

- ▶ tools are defined by the system
- ▶ never invented by the model

Tool schema

```
1 name: get_temperature
2 arguments:
3   location: string
4   leadtime: integer
```

The model learns:

- ▶ when this tool applies
- ▶ how to fill arguments

```
{
  "tool_call": {
    "name": "<function name>",
    "arguments": { ... }
  }
}
```

Tool Calls in Model Output

Modern LLMs can output:

- ▶ normal assistant text
- ▶ structured tool calls

A tool call contains:

- ▶ tool name
- ▶ arguments
- ▶ no natural language

This decision is:

- ▶ made by the model
- ▶ enforced by the API

LLM output

```
1 tool_call:  
2   name: get_temperature  
3   arguments:  
4     location: "Berlin"  
5     leadtime: 24
```

No parsing of prose required.

```
try:  
    tool_call = json.loads(raw_output)["tool_call"]  
    print("Parsed tool call:")  
    print("Tool name:", tool_call["name"])  
    print("Arguments:", tool_call["arguments"])  
except Exception as e:  
    print("Failed to parse JSON output")  
    print("Error:", e)
```

Streaming and Open-Source Models

In streaming APIs:

- ▶ **tool calls** appear **inside the stream**
- ▶ mixed with text tokens

Open-source models (e.g. LLaMA):

- ▶ emit structured patterns
- ▶ still require system-side handling

Common fallback:

- ▶ detect JSON blocks
- ▶ interpret intent manually

Historical context

- ▶ Older Framework: JSON scanning
- ▶ Claude UI: similar hybrid behavior

Native tool calling reduces:

- ▶ ambiguity
- ▶ parsing complexity

From JSON Parsing to Native Tool Calling

Old approach:

- ▶ prompt for JSON
- ▶ parse model output
- ▶ recover from errors

Modern approach:

- ▶ tools defined explicitly
- ▶ model selects tool
- ▶ system executes tool

Key shift:

- ▶ from text parsing to action selection

Architectural consequence

- ▶ cleaner agent design
- ▶ safer execution
- ▶ better testability

This enables:

- ▶ LangChain tools
- ▶ LangGraph nodes
- ▶ reliable agents

From Scripts to Agents: Why AI Agents Exist

Classical scripts and pipelines

- ▶ Deterministic execution
- ▶ Fixed control flow
- ▶ Explicit inputs and outputs

They work well if:

- ▶ tasks are fully specified
- ▶ all cases are known in advance

Limitations

- ▶ No interpretation of intent
- ▶ Poor handling of ambiguity
- ▶ Fragile when requirements change

Where this moves today

- ▶ Natural-language problem descriptions
- ▶ Large, evolving code bases
- ▶ Underspecified or incomplete tasks



From rigid pipelines to **adaptive systems**

What Is an AI Agent? — Core Mental Model

Agent as a closed loop

- perception
- reasoning (LLM)
- action
- environment / state

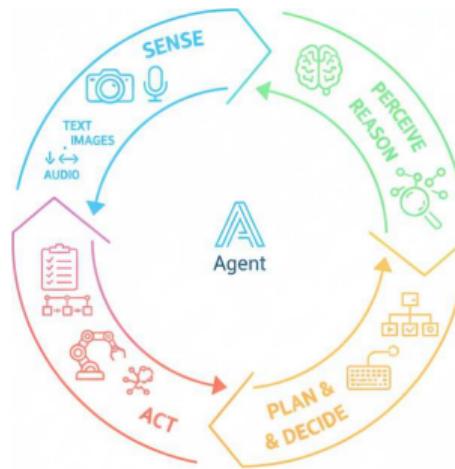
Key ingredients:

- ▶ internal state
- ▶ access to tools
- ▶ explicit control flow

An agent is *active*, not passive.

LLM vs tool vs agent

- ▶ **LLM:** maps text to text
- ▶ **Tool:** executes a fixed function
- ▶ **Agent:** decides *what to do next*



Decision-making loop
with state
and actions

LLM Capabilities and Limits in Practice

What LLMs are good at

- ▶ Understanding natural language
- ▶ Generating plausible code and text
- ▶ Pattern completion and refactoring
- ▶ Explaining existing code

LLMs approximate

$$p(\text{next token} \mid \text{context})$$

from large training corpora.

Systematic limitations

- ▶ No ground truth or verification
- ▶ Hallucinated but plausible outputs
- ▶ Overconfidence in incorrect answers
- ▶ Sensitivity to prompt phrasing

Sampling matters

temperature $T \uparrow \Rightarrow$ variance of outputs \uparrow

Low T : reproducible but rigid
High T : creative but unstable

Prompting for Code Generation

- ▶ Unambiguous task description
- ▶ Explicit constraints
- ▶ Predictable output format

Critical constraints

- ▶ Code-only output
- ▶ No explanations or markdown
- ▶ Explicit library choices

Poor prompts lead to:

- ▶ mixed prose and code
- ▶ missing imports
- ▶ implicit assumptions

Example: strict system prompt

System message

- 1 You are an AI coder.
- 2 Output ONLY valid Python code.
- 3 No markdown. No explanations.
- 4 Assume Python 3.10.

User prompt pattern

User message

- 1 Write a Python function `f(x)`
- 2 using numpy that returns a
- 3 polynomial with $|f(x)| < 10$
- 4 `for x in [-10, 10].`

Key point: Prompting is part of the program.

Manual Coding with LLMs (Human in the Loop)

Typical workflow

- ▶ Prompt LLM for code
- ▶ Inspect generated output
- ▶ Edit or correct manually
- ▶ Execute and test

Why humans stay involved

- ▶ Detect logical errors
- ▶ Spot missing assumptions
- ▶ Judge correctness, not plausibility

LLMs assist, they do not decide.

Trust boundaries

- ▶ Never execute blindly
- ▶ Always read generated code
- ▶ Treat LLM output as a draft

Productive collaboration

- ▶ LLM: speed, syntax, structure
- ▶ Human: intent, validation, responsibility

Key point:

Responsibility stays with the human.

Manual Coding with LLMs — Example UI Interaction

Human in the loop

- ▶ Natural-language prompt
- ▶ Code suggestion by LLM
- ▶ Human inspection
- ▶ Manual correction

This is the **baseline** mode of working with LLMs.

The screenshot shows a web-based interface for interacting with a large language model (LLM) to generate Python code for plotting a hypotrochoid curve. The interface includes a text input field for the user's prompt, a code editor window showing the generated code, and a text area for the LLM's response.

Prompt: Plotting an interesting 1D curve in Python

I'll create a beautiful parametric curve for you - a hypotrochoid, which creates mesmerizing spirograph-like patterns.

Code Suggestion: Hypotrochoid Curve Plot

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters for the hypotrochoid
R = 5 # radius of fixed circle
r = 3 # radius of rolling circle
d = 5 # distance from center of rolling circle

# Generate parameter t
t = np.linspace(0, 2 * np.pi * r / np.gcd(R, r), 1000)

# Parametric equations for hypotrochoid
x = (R - r) * np.cos(t) + d * np.cos((R - r) / r * t)
y = (R - r) * np.sin(t) - d * np.sin((R - r) / r * t)

# Create the plot
plt.figure(figsize=(10, 10))
plt.plot(x, y, linewidth=1.5, color='darkblue')
plt.axis('equal')
plt.grid(True, alpha=0.3)
plt.title('Hypotrochoid Curve (R=5, r=3, d=5)', fontsize=14, fontweight='bold')
plt.xlabel('x')
plt.ylabel('y')
plt.tight_layout()
plt.show()
```

Answer: Antwort...
Sonnet 4.5

Typical web-based LLM interface used for interactive coding

Executing Generated Code Safely

Execution options

- ▶ exec inside current process
- ▶ Separate process via subprocess
- ▶ File-based execution

Trade-offs

- ▶ exec: fast, **unsafe**
- ▶ subprocess: isolated, slower
- ▶ Files: traceable, debuggable

Rule of thumb

Never execute LLM code
without isolation or inspection.

Minimal example

File-based execution

```
1 with open("gen.py", "w") as f:  
2     f.write(code)  
3  
4 import subprocess  
5 subprocess.run(  
6     ["python", "gen.py"],  
7     check=True)
```

Security risks

- ▶ File system access
- ▶ Network calls
- ▶ Infinite loops

Mitigation requires **process isolation**.

Error Handling and Feedback Loops

Why errors are central

- ▶ LLM-generated code is often incomplete
- ▶ Small syntax or logic errors are common
- ▶ First attempt rarely works

Errors provide structured feedback :

- ▶ missing imports
- ▶ wrong assumptions
- ▶ invalid API usage

Typical feedback loop

Error capture and retry

```
1 try:  
2     exec(code)  
3     success = True  
4 except Exception as e:  
5     error = traceback.format_exc()  
6     success = False
```

Key design choices

- ▶ feed back full traceback
- ▶ limit number of retries
- ▶ detect repeating failures

Key point: Errors drive self-correction .

The First Coding Agent: Self-Correcting Loops

Core idea

- ▶ LLM generates code
- ▶ Code is executed
- ▶ Errors are captured
- ▶ LLM is prompted to fix them

This creates an
autonomous correction loop .

Minimal success criteria

- ▶ Code executes without error
- ▶ Output matches basic expectations

Minimal agent loop

Self-correcting loop

```
1 for attempt in range(max_tries):  
2     code = llm(prompt)  
3     try:  
4         exec(code)  
5     break  
6 except Exception as e:  
7     prompt += traceback.format_exc()
```

Why this is already an agent

- ▶ autonomous retries
- ▶ internal state (prompt history)
- ▶ decision: retry vs stop

No framework required.

From Prompts to Programs: Abstraction Boundaries

The core problem

- ▶ Prompts mix intent and execution
- ▶ Small wording changes alter behavior
- ▶ Logic is implicit and fragile

This leads to prompt brittleness :

- ▶ hard to debug
- ▶ hard to reuse
- ▶ hard to test

Separating responsibilities

- ▶ Prompt: what should be done
- ▶ Code: how it is executed
- ▶ Control flow: when to retry or stop

Design principle

- ▶ Prompts describe intent
- ▶ Programs enforce structure

Key point: LLMs belong inside programs, not around them.

Why Agent Frameworks Exist

Scaling problems of ad-hoc agents

- ▶ Prompts grow uncontrollably
- ▶ Control logic becomes implicit
- ▶ Error handling is duplicated

As systems grow:

- ▶ code becomes unstructured
- ▶ behavior is hard to reproduce
- ▶ debugging is expensive

What frameworks provide

- ▶ Explicit control flow
- ▶ Reusable abstractions
- ▶ Tool and memory interfaces
- ▶ Observability and logging

What they do *not* provide

- ▶ Correct reasoning
- ▶ Ground truth
- ▶ Guaranteed success

Key point: Frameworks add **structure**, not intelligence.

Survey of Agent Frameworks (Critical View)

Why so many frameworks?

- ▶ **No standard agent abstraction**
- ▶ **Rapidly evolving LLM APIs**
- ▶ Different design philosophies

Main categories

- ▶ Loop-based agents
- ▶ Chain-based frameworks
- ▶ Graph-based workflows
- ▶ Multi-agent orchestration

Typical examples

- ▶ LangChain : chaining + tools
- ▶ LangGraph : explicit control flow
- ▶ CrewAI : role-based agents
- ▶ AutoGPT-style: autonomous loops

Practical assessment

- ▶ Most are experimental
- ▶ APIs change quickly
- ▶ Production use needs caution

Rule: Use frameworks to clarify , not to hide logic.

LangChain: Motivation and Architecture

What LangChain targets

- ▶ Reusable prompt templates
- ▶ Standardized tool access
- ▶ Simple memory abstractions

LangChain focuses on:

- ▶ composition
- ▶ integration
- ▶ rapid prototyping

Core building blocks

- ▶ LLM interface
- ▶ PromptTemplate
- ▶ Chain
- ▶ Tool
- ▶ Memory

Limitation

- ▶ Linear execution model
- ▶ Limited explicit control flow

Key point: LangChain **glues components**, it does not control logic.

Prompt Templates and Tool Integration (LangChain)

- ▶ Separates instructions and variables **Tool integration**
- ▶ Enforces a fixed structure
- ▶ Improves reuse and testing

PromptTemplate example

```
1 from langchain.prompts
    import PromptTemplate
2
3 prompt = PromptTemplate(
4     input_variables=["x"],
5     template=
6         "Write Python code
7             computing f(x)
8             such that |f(x)| < 10. )
```

Tool definition

```
1 from langchain.tools import tool
2
3 @tool
4 def square(x: float) -> float:
5     return x * x
```

Critical limitations

- ▶ LLM may select wrong tool
- ▶ Arguments may be malformed

Rule: Tool calls must be **validated outside** the LLM.

Autonomous Coding Agent — Self-Correcting Loop

Agent task

- ▶ Natural-language goal
- ▶ LLM generates full script
- ▶ Script is executed
- ▶ Errors are fed back automatically

This is already a **true agent**:

- ▶ autonomous retries
- ▶ internal state (prompt history)
- ▶ stop criterion

Autonomous code agent (excerpt)

```
1 def autonomous_code_agent(task):  
2     for attempt in range(5):  
3         code = llm.invoke(task).content  
4         try:  
5             exec(code)  
6             return True  
7         except Exception as e:  
8             task += traceback.format_exc()  
9     return False
```

Key point: No framework required — this is pure control logic around an LLM.

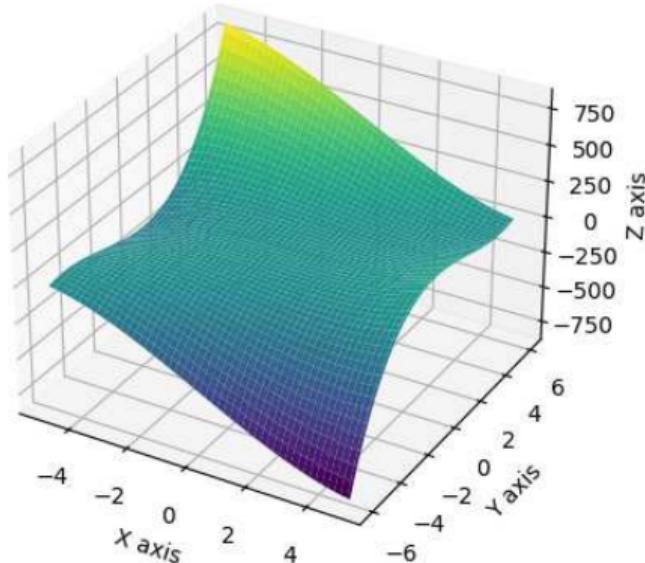
Autonomous Agent Output — 3D Polynomial Visualization

Task executed

- ▶ Generate 2D polynomial
- ▶ Evaluate on grid
- ▶ Create 3D visualization
- ▶ Save result as image

Important

- ▶ Code was generated
- ▶ Code was executed
- ▶ Output was validated



3D surface plot generated by an autonomous LLM coding agent

Memory and Context Management in Agents

Why agents need memory

- ▶ Remember past attempts
- ▶ Accumulate errors and feedback
- ▶ **Maintain task continuity**

Without memory:

- ▶ repeated failures
- ▶ no learning across steps
- ▶ brittle behavior

Types of memory

- ▶ Prompt memory (growing instruction and error history)
- ▶ File-based memory (saved code, plots, logs)
- ▶ Explicit state (variables passed between steps)

Key distinction

- ▶ Chat history \neq agent state
- ▶ State must be **explicit and inspectable**

Memory Management in LangGraph

- ▶ No hidden chat history
- ▶ No implicit conversation memory
- ▶ No prompt accumulation

Instead:

- ▶ All memory lives in a **typed state object**
- ▶ Each node reads and updates this state
- ▶ State is passed explicitly between nodes

Consequence:

- ▶ Deterministic execution
- ▶ Fully inspectable memory
- ▶ Reproducible agent behavior

LangGraph state

```
1 class MyState(TypedDict):  
2     query: str  
3     fc_datetime: str  
4     fc_reference_datetime: str  
5     fc_leadtime: str  
6     fc_location_of_interest: str  
7     fc_variable: str  
8     temperature_data: Any  
9     output: str
```

In LangGraph, chat history, tool memory, and agent scratchpads are replaced by a single typed state object. Memory is **explicit data**, not implicit text.

Control Flow in LangGraph

- ▶ No hidden agent loops
- ▶ No implicit retries
- ▶ No LLM-driven control decisions

Instead:

- ▶ Execution follows a directed graph
- ▶ Nodes are pure Python functions
- ▶ Edges define allowed transitions

LangGraph control flow

```
1 builder.set_entry_point("extract_forecast_datetime")
2 builder.add_edge(
3     "extract_forecast_datetime",
4     "get_latest_forecast_reference_time"
5 )
6 builder.add_edge(
7     "get_latest_forecast_reference_time",
8     "calculate_lead_time" )
9 [...]
```

builder.set_finish_point("plot_temperature")

The graph itself defines what happens next. There is no hidden agent controller.

LLMs as Nodes in LangGraph

- ▶ LLMs do *not* control execution
- ▶ LLMs do *not* manage memory
- ▶ LLMs do *not* decide termination

Instead:

- ▶ LLMs are used for local reasoning tasks
- ▶ Each call has a well-defined input
- ▶ Each call produces a bounded output

Typical LLM roles:

- ▶ information extraction
- ▶ classification
- ▶ summarization

LangGraph node

```
1 def extr_loc_node(state: MyState) ->
   MyState:
2     location = extr_loc(state["query"])
3     state["fc_location"] = location
4     return state
```

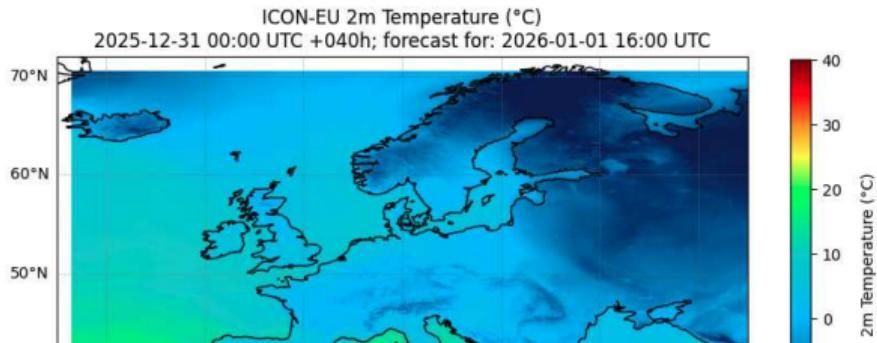
The LLM acts as a pure transformation from input fields to output fields.

LangGraph Weather Forecast Assistant

```
[3]: dawid.ai("Please give me the temperature forecast for Monday 4pm.")

extract_forecast_datetime: 2026-01-01 16
get_latest_forecast_reference_time: 2025-12-31 00
calculate_forecast_lead_time: 40.0
extract_location: Location: Not specified
extract_variable: Temperature
Downloading: https://opendata.dwd.de/weather/nwp/icon-eu/grib/00/t_2m/icon-eu_europe_regular-lat-lon_single-level_2025123100_040_T_2M.grib2.bz2
Plot saved as temperature_forecast.png
```

I've created and displayed the temperature forecast for Monday at 4 PM. If you have a specific location in mind, feel free to let me know for more detailed information!



End-to-end agent pipeline: natural language → structured state → tools → visualization

Failure Handling and Robustness

- ▶ LLM extraction errors
- ▶ Missing or delayed data
- ▶ Tool execution failures
- ▶ Invalid intermediate state

LangGraph strategy

Each node handles local failure

State records partial results

Graph execution remains controlled

Key principle

- ▶ Failures are data, not crashes

Failure-aware node

```
1 def plot_temperature_node(state: MyState)
2     -> MyState:
3     if state.get("temperature_data") is None:
4         state["output"] = "No data available"
5     return state
6
7     plot_t2m_EU(state["temperature_data"],
8                 save_plot=True)
9     state["output"] = "Plot created"
10    return state
```

Failures do not break the agent. They update the state and allow the graph to terminate safely.

Why This Scales: From Prototype to System

Problems with classic agent loops

- ▶ hidden prompt growth
- ▶ non-reproducible behavior
- ▶ difficult debugging
- ▶ unclear failure causes

LangGraph advantages

- ▶ explicit state evolution
- ▶ deterministic control flow
- ▶ inspectable intermediate results
- ▶ testable nodes

Engineering outcome

- ▶ agents become **systems**
- ▶ not demos or experiments

What becomes possible

- ▶ unit tests per node
- ▶ regression tests on state
- ▶ logging and metrics
- ▶ CI/CD integration

Key insight

Agents scale when they obey the same rules as software.

When Multi-Agent Systems Make Sense

One agent is sufficient when

- ▶ tasks are sequential
- ▶ state is compact
- ▶ logic is well-defined
- ▶ tools dominate execution

Multiple agents are useful when

- ▶ responsibilities are clearly separable
- ▶ different reasoning styles are needed
- ▶ tasks can proceed independently
- ▶ software components can be done independently

Key warning

- ▶ multi-agent systems are not better by default

Typical multi-agent roles

- ▶ planner / coordinator
- ▶ domain expert
- ▶ tool executor
- ▶ verifier or critic

Design principle

Add agents only when you can explain their responsibility.

CrewAI: Role-Based Agent Collaboration

What CrewAI provides

- ▶ explicit agent roles
- ▶ task delegation
- ▶ simple coordination logic
- ▶ readable high-level structure

Typical use cases

- ▶ document analysis
- ▶ research workflows
- ▶ report generation
- ▶ exploratory automation

Strength

- ▶ fast prototyping of multi-agent ideas

Limitations

- ▶ implicit memory handling
- ▶ limited state visibility
- ▶ weak failure control
- ▶ hard to test systematically

Key takeaway

*CrewAI is useful for
coordination demos, not for
operating critical systems.*

CrewAI Demo: Internal Achievements Report

Scenario

- ▶ 3–4 months of AI-related achievements
- ▶ Research and software development
- ▶ Target: internal newsletter or ministry

Agent roles

- ▶ Planner: report structure
- ▶ Scientific writer: technical accuracy
- ▶ Impact translator: non-expert framing
- ▶ Editor: clarity and consistency

Notebook flow

- ▶ raw bullet-point inputs
- ▶ sequential task execution
- ▶ agent-to-agent refinement
- ▶ final Markdown output

What this demonstrates

- ▶ realistic knowledge work
- ▶ role separation
- ▶ human-facing automation

AI Agent Landscape — January 2026

What has stabilized

- ▶ **LLMs as reasoning engines**
- ▶ Tool calling as standard interface
- ▶ Explicit state and control flow
- ▶ Strong separation of roles

What is fading

- ▶ prompt-only agents
- ▶ hidden scratchpads
- ▶ uncontrolled self-loops
- ▶ purely conversational systems

Dominant design patterns

- ▶ Graph-based agents (LangGraph)
- ▶ Tool-driven execution
- ▶ **Typed, inspectable state**
- ▶ Human-in-the-loop checkpoints

Reality check

- ▶ Agents are software systems
- ▶ Not autonomous intelligence
- ▶ Require engineering discipline

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30				Lunch Break	
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00				Joint Dinner	

DAWID in Action

User: rpotthast | Logout New Session 

Session ID: 63e08dff07f356340512ac4b94f39b8 Sessions

DAWID beta

You asked:

what is the weather tomorrow at 3pm in Frankfurt?

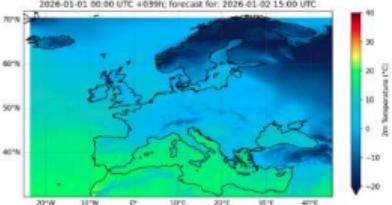
DAWID:

- Model
- Function get_weather_forecast Results
- ▼ Model

gpt-5-mini

Here is the ICON-EU forecast plot for Frankfurt tomorrow (2026-01-02) at 15:00 (03:00 PM) — see the image below:

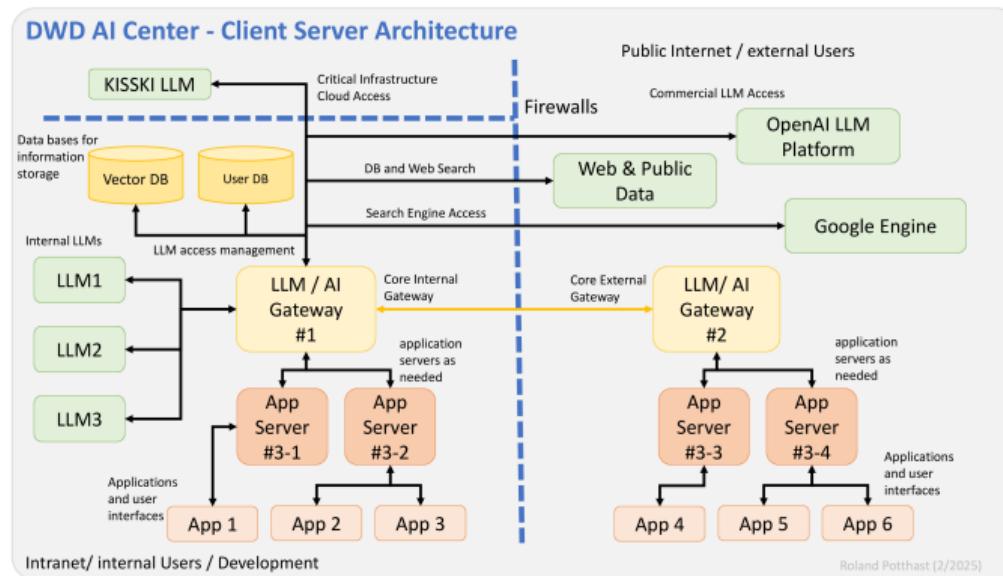
ICON-EU 2m Temperature (°C)
2026-01-01 00:00 UTC +099h; forecast for: 2026-01-02 15:00 UTC



Would you like a numeric summary (temperature, precipitation chance, wind speed/direction) or a short text forecast? I can extract those values for you.

Interactive DAWID assistant session with live streaming responses and tool integration

AI Centre Client–Server Architecture



Frontend UI, backend orchestration, language models, tools, and data sources

DAWID System Architecture — Frontend Stack

User Interface Layer

- ▶ **HTML5** Page structure, forms, UI elements
- ▶ **CSS** Layout, styling, responsive design
- ▶ **JavaScript** Interaction logic and event handling
- ▶ **Fetch API** Asynchronous requests and streaming

Dynamic Content

- ▶ **Markdown rendering** Incremental formatting of streamed output

Browser Capabilities

- ▶ **Audio** recording via Web APIs
- ▶ Clipboard access
- ▶ Local UI state and session handling

Design Principle

- ▶ No heavy frontend frameworks
- ▶ No client-side AI logic
- ▶ Frontend as thin interaction layer
- ▶ Works on **Laptop**, Workstation, **Mobile Device**

DAWID System Architecture — Server and Backend

Web and Gateway Layer

- ▶ **Web Server** (Apache / Nginx)
TLS termination, static content, routing
- ▶ **PHP Gateway** Sessions, uploads, request forwarding
- ▶ **Security Boundary** No direct access to backend services

Role

- ▶ Separates public UI from compute backend
- ▶ Enforces access control and isolation

Backend and Compute Layer

- ▶ **Python Backend (FastAPI)**
Central orchestration logic
- ▶ **Unicorn Server** Async execution, streaming responses
- ▶ **LLM Backends** Local and cloud-based models

Key Principle

- ▶ Backend owns state, context, and tools
- ▶ Frontend remains stateless and simple

File Upload Workflow in the DAWID Frontend

User Interaction

- ▶ File selected via HTML form
- ▶ Automatic submission on selection
- ▶ No explicit upload button required

Gateway Interaction

- ▶ Upload sent to PHP gateway
- ▶ File stored in session-specific folder
- ▶ Backend notified of new resource

Frontend Logic

- ▶ JavaScript listens to file change events
- ▶ Uses **Fetch API** for upload
- ▶ Asynchronous, non-blocking UI

Minimal JavaScript Logic

Automatic upload on file selection

```
1 fileInput.addEventListener("  
  change", () => {  
  2   uploadForm.requestSubmit();  
  3 });
```

Streaming LLM Responses in the Browser

Why Streaming?

- ▶ Immediate user feedback
- ▶ Reduced perceived latency
- ▶ Long answers remain usable

Frontend Mechanism

- ▶ **Fetch API** with streamed response
- ▶ **ReadableStream** reader
- ▶ Chunk-by-chunk text processing

Key UX Effect

- ▶ User sees **partial results** instantly
- ▶ No blocking on full completion

Incremental Rendering

- ▶ Accumulate streamed text
- ▶ Re-render on each new chunk
- ▶ Markdown parsed continuously

```
const reader = response.body.getReader();
while (true) {
    const { done, value } = await reader.read();
    if (done) break;
    partial += decoder.decode(value);
    responseDiv.innerHTML = marked.parse(partial)
}
```

Design Choice

- ▶ Simple robust rendering

Frontend Capabilities: Math Rendering and Sessions

Streaming Mathematical Content

1. Proof by similar triangles

Let triangle ABC be right-angled at C . Put $AB = c$ (hypotenuse), $BC = a$, $AC = b$ the foot be D . Let $AD = d$, $DB = e$ so $d + e = c$. From similarity of triangles:

$$\frac{a}{c} = \frac{c}{e} \Rightarrow a^2 = c \cdot e$$

$$\frac{b}{c} = \frac{c}{d} \Rightarrow b^2 = c \cdot d$$

Adding:

$$a^2 + b^2 = c(e + d) = c \cdot c = c^2.$$

2. Area / rearrangement proof (square with four triangles)

Mathematical expressions rendered live during streamed responses using **Markdown** and **MathJax**.

Session Display and Management



Multiple interaction sessions with selectable history and clear **context separation** at the frontend level.

DAWID Backend: Central Unicorn Server

Role of the Backend Server

- ▶ Single central entry point for all requests
- ▶ Receives queries from the frontend gateway
- ▶ Handles sessions and user context
- ▶ Streams responses back to the client

Key Responsibilities

- ▶ Model selection and routing
- ▶ Context assembly (history, RAG)
- ▶ Tool and workflow execution

FastAPI + Unicorn

- ▶ **FastAPI**: API definition and validation
- ▶ **Unicorn**: async, high-performance server
- ▶ Multi-worker capable deployment

Design Principle

- ▶ Backend owns state, logic, and control
- ▶ LLMs are subsystems, not the system

Backend Route Overview

Main Route Categories

- ▶ **LLM Routes** Prompt handling, streaming responses
- ▶ **Upload Routes** File transfer, storage, metadata
- ▶ **Audio Routes** Speech upload and transcription

Supporting Routes

- ▶ Dataspace and document access
- ▶ Session and user management

API Routes

- ▶ OpenAI-compatible chat API
- ▶ Streaming and non-streaming endpoints
- ▶ Programmatic access for external tools

Design Principles

- ▶ Clear separation of concerns
- ▶ Explicit routing and ownership
- ▶ Routes reflect system capabilities

LLM Streaming Routes

Purpose of LLM Routes

- ▶ Accept user queries from the gateway
- ▶ Build prompts from session history
- ▶ Integrate retrieved context (RAG)
- ▶ Return responses as streams

Streaming Logic

- ▶ Token-wise or chunk-wise output
- ▶ Immediate forwarding to frontend
- ▶ Non-blocking async execution

Unified Interface

- ▶ Same route for local and cloud models
- ▶ OpenAI, Claude, Gemini, LLaMA, etc.
- ▶ Model chosen at runtime

Interception Points

- ▶ Function call detection
- ▶ Tool execution triggers
- ▶ Session state updates

Key Insight

- ▶ LLM output is **not final**
- ▶ It is an intermediate signal

Upload and File Management Routes

Purpose of Upload Routes

- ▶ Receive files from the frontend gateway
- ▶ Store data in session- or user-specific folders
- ▶ Register files for later access by tools

File Types

- ▶ Documents (PDF, text, Markdown)
- ▶ Data files (NetCDF, images)
- ▶ Audio recordings

Backend Handling

- ▶ Controlled directory structure
- ▶ Metadata stored with session context
- ▶ Optional indexing for retrieval

Integration Points

- ▶ Retrieval-Augmented Generation (RAG)
- ▶ Tool execution (plots, analysis)
- ▶ Download and reuse in workflows

Key Principle

- ▶ Uploaded files become first-class resources

Audio Routes: Speech-to-Text

Motivation

- ▶ Natural user interaction via speech
- ▶ Hands-free input for complex queries
- ▶ Multimodal access without UI complexity

Frontend Interaction

- ▶ Audio recorded in the browser
- ▶ File uploaded via gateway
- ▶ Treated like any other uploaded resource

Backend Processing

- ▶ Local or remote speech-to-text engine
- ▶ Conversion to plain text
- ▶ Injected into standard LLM pipeline

Design Principle

- ▶ Audio is a preprocessing step
- ▶ All downstream logic remains text-based
- ▶ No special cases for speech afterwards

User Management and Dataspaces

User and Session Model

- ▶ Each interaction tied to a session ID
- ▶ Optional persistent user identity
- ▶ Separation of short-term and long-term context

Access Control

- ▶ User-specific private areas
- ▶ Controlled access to shared resources

Dataspaces

- ▶ Private document folders per user
- ▶ Group-shared knowledge spaces
- ▶ Clear ownership and responsibility

Purpose

- ▶ Organize documents and data
- ▶ Provide structured context to the LLM
- ▶ Enable collaborative workflows

Retrieval-Augmented Generation (RAG)

Motivation

- ▶ LLMs do not know user-specific documents
- ▶ Uploaded data must influence responses
- ▶ Context must be selected , not dumped

Local Retrieval

- ▶ Document chunks embedded locally
- ▶ Similarity search via **FAISS**
- ▶ No external data transfer

Context Assembly

- ▶ Detect when documents are relevant
- ▶ Retrieve top-matching chunks
- ▶ Inject into prompt before generation

Key Principle

- ▶ RAG augments knowledge , not reasoning
- ▶ The LLM remains the decision engine

Available LLM Models and Capability Tiers

Capability Tiers

- ▶ **FAST** Lightweight, fast, cost-efficient
- ▶ **CORE** Strong general-purpose models
- ▶ **PRO** Heavy reasoning, coding, long context
- ▶ **ULTRA** Highest available model capability

Design Choice

- ▶ One **best model** per tier and supplier
- ▶ No artificial or redundant model options

Supported Model Families

- ▶ OpenAI (Gpt5.2, GPT5.1, GPT-4o, GPT-5-mini)
- ▶ Claude (Anthropic)
- ▶ Gemini (Google)
- ▶ LLaMA (local and remote)
- ▶ Mistral / Mixtral
- ▶ GPT-OSS

Key Principle

- ▶ Users select **capability**, not internals
- ▶ Backend resolves tier to concrete model

DAWID Model Aliases and Routing

Internal Model Aliases

- ▶ Abstract names used by the frontend
- ▶ Examples:
 - ▶ openai-fast, openai-core
 - ▶ claude-pro
 - ▶ llama-fast
- ▶ Aliases encode **capability tier**

Why Aliases?

- ▶ Stable interface for users
- ▶ Models can change transparently

Backend Resolution

- ▶ Alias mapped to concrete model name
- ▶ Mapping defined centrally in settings
- ▶ Supplier-specific resolution

Routing Logic

- ▶ Backend checks model ownership
- ▶ Selects local or remote backend
- ▶ Routes request to correct engine

Key Principle

- ▶ Frontend never sees real model IDs
- ▶ Backend retains **full control**

Function Calling in DAWID: Weather Service Productivity

DAWID as a Productivity Platform

- ▶ DAWID supports internal and external weather services
- ▶ Focus on operational and scientific workflows
- ▶ Integration of data, models, and tools

Role of Function Calling

- ▶ LLM suggests domain-relevant actions
- ▶ DAWID executes approved functions
- ▶ Actions are embedded in workflows

Typical Weather-Service Tasks

- ▶ Access and process NWP data
- ▶ Generate plots, maps, and diagnostics
- ▶ Perform domain-specific analyses
- ▶ Support decision-making and reporting

Key Principle

- ▶ Function calls increase efficiency, not autonomy
- ▶ **DAWID augments expert work, it does not replace it**

Classical Function Calling in DAWID (JSON-Based)

What “Classical” Means in DAWID

- ▶ Function calls encoded explicitly in text
- ▶ Usually formatted as **JSON blocks**
- ▶ Independent of specific LLM providers

Why DAWID Still Supports This

- ▶ Works with **all models** (local and remote)
- ▶ Robust fallback mechanism
- ▶ Easy to debug and inspect

Example: Function Proposal by the LLM

JSON-style function call

```
1 {"function_calls": [{}  
2   "name": "get_icon_forecast  
3   ",  
4   "arguments": {  
5     "variable": "t2m",  
6     "region": "Germany" }]}]
```

DAWID Interpretation

- ▶ JSON is **parsed and validated**
- ▶ Function is checked against allow-list
- ▶ Execution only happens in backend

Native Function Calling in DAWID

What “Native” Means

- ▶ Functions declared via explicit schemas
- ▶ Typed arguments and clear signatures
- ▶ Direct support by modern LLM APIs

Advantages in DAWID

- ▶ No JSON parsing from free text
- ▶ Fewer hallucinated calls
- ▶ Safer execution path

DAWID Integration

- ▶ Functions registered centrally
- ▶ Only allowed tools are exposed
- ▶ Backend controls execution order

Typical Use Cases

- ▶ Plot generation
- ▶ Data download and extraction
- ▶ Model-based diagnostics

Design Principle

- ▶ Native calls are preferred
- ▶ Classical calls remain a fallback

Function Calling inside DAWID: Execution Paths

Internal Execution: LangGraph

- ▶ Functions organized as LangGraph nodes
- ▶ Explicit shared state passed between nodes
- ▶ Deterministic execution order
- ▶ Suitable for multi-step workflows

Typical Internal Tasks

- ▶ Data preprocessing and analysis
- ▶ Plot and image generation
- ▶ Chained scientific workflows

External Execution: HTTP Endpoints

- ▶ Functions exposed via REST endpoints
- ▶ Called from DAWID via HTTP requests
- ▶ External services remain isolated

Typical External Tasks

- ▶ Access to targeted data services
- ▶ Specialized model inference
- ▶ **enables modular service development**

Key Principle

- ▶ function can run on various computers
- ▶ Execution remains controlled and auditable, flexible, cloud ready

External Function Calls inside DAWID

```
# === External Client API forwarding ===
external_funcs = [f for f in functions_requested if f["name"] in EXTERNAL_FUNCTIONS]

print(f"external_funcs={external_funcs}")

if external_funcs:
    fc = external_funcs[0]

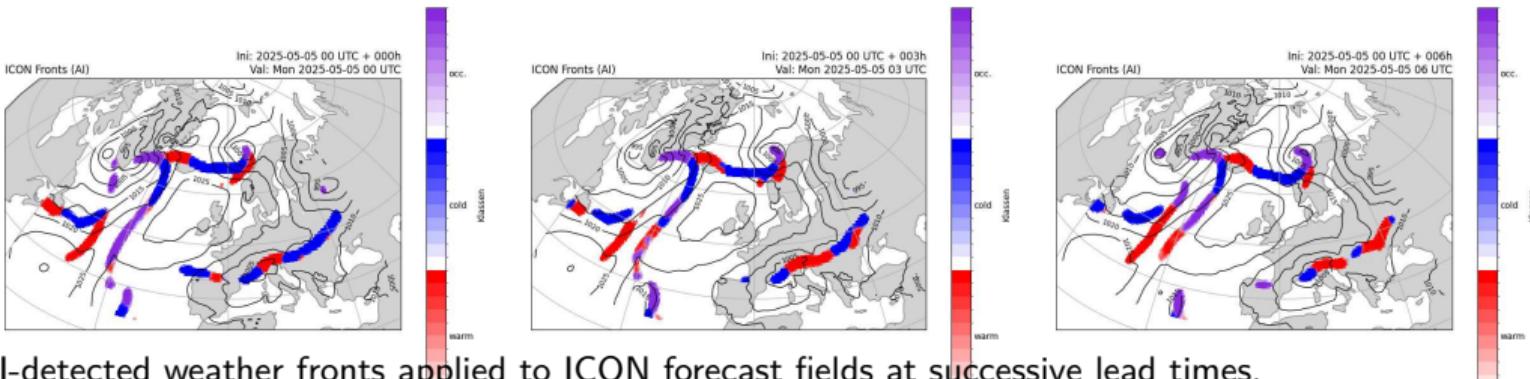
    payload = {
        "session_id": session_id,
        "username": username,
        "function_call": fc
    }

EXTERNAL_URL = os.getenv("EXTERNAL_CLIENT_API", "http://localhost:5000/api/forward")

try:
    import requests
    r = requests.post(EXTERNAL_URL, json=payload, timeout=20)
    r.raise_for_status()
    return f"/ External client forwarded: {r.text}"
except Exception as e:
    return f"✗ External client call failed: {e}"
```

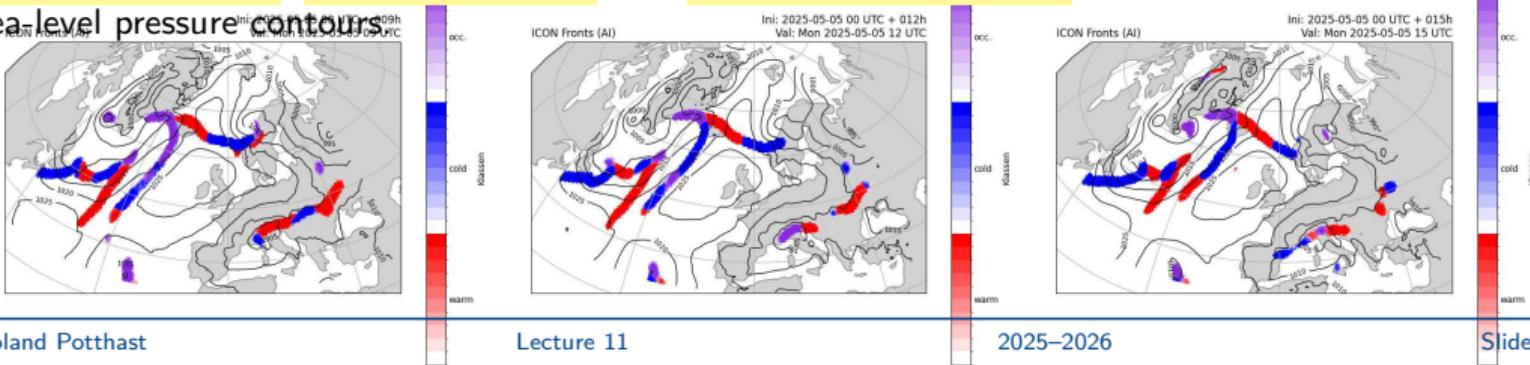
Integration of **external function calls** within the DAWID backend through REQUESTS.

AI-Based Feature Detection: Weather Fronts



AI-detected weather fronts applied to ICON forecast fields at successive lead times.

Warm fronts (red), cold fronts (blue), and occluded fronts (violet) are overlaid with mean sea-level pressure contours.



Input Data and Preprocessing

Meteorological Input Fields

- ▶ Mean sea-level pressure (**PMSL**)
- ▶ 2 m temperature (**T2M**)
- ▶ 2 m relative humidity (**RH2M**)
- ▶ 10 m wind components (**U10M**,
V10M)
- ▶ Land-sea mask (**FRLAND**)

Data Source

- ▶ ICON analysis and forecast fields
- ▶ Regular latitude-longitude grid

Preprocessing Steps

- ▶ Channel-wise normalization
- ▶ Scaling to comparable numerical ranges
- ▶ Binary encoding of land-sea mask

Resulting Tensor

- ▶ Shape: [6, lat, lon]
- ▶ Stored in NetCDF format
- ▶ One file per analysis or forecast time

Design Principle

- ▶ Preserve physical meaning
- ▶ Avoid unnecessary feature engineering

Model Architecture and Training

Model Architecture

- ▶ U-Net for semantic segmentation
- ▶ Encoder–decoder with skip connections
- ▶ Preserves spatial detail

Input / Output

- ▶ Input channels: 6
- ▶ Output classes: 4
- ▶ Pixel-wise classification

PyTorch Model Setup

```
model = UNet(  
    in_channels=6,  
    out_channels=4,  
    init_features=64  
).to(device)  
  
loss_fn = nn.CrossEntropyLoss()  
optimizer = torch.optim.AdamW(  
    model.parameters(),  
    lr=1e-4 )
```

Training Setup

- ▶ Supervised learning
- ▶ Labeled frontal maps
- ▶ Early stopping and validation

Operational Front Detection in Forecast Practice

From Research to Operations

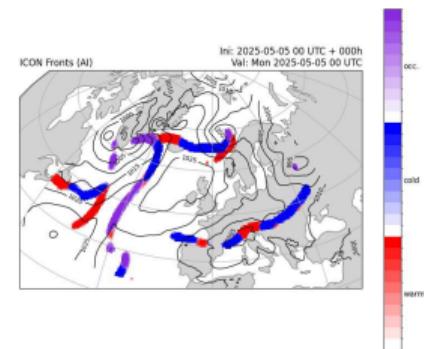
- ▶ AI-based front detection is operationally deployed
- ▶ Integrated into NinJo service stations
- ▶ Available directly in the forecasters' workflow

Operational Value

- ▶ Fast, consistent front analysis
- ▶ Supports situational awareness
- ▶ Reduces manual drawing effort

User Acceptance

- ▶ Widely used by weather forecasters
- ▶ Particularly helpful in complex synoptic situations
- ▶ Acts as decision support, not automation



Lecture 11 — Summary

DAWID Platform

- ▶ **Productivity platform** for NHMS
- ▶ Supports internal and external users
- ▶ Lightweight web frontend
- ▶ Streaming interaction and session management

Backend Architecture

- ▶ Modular FastAPI backend
- ▶ Unified access to multiple LLM families
- ▶ Controlled function execution
- ▶ Local and external services integrated

AI Integration

- ▶ Function calling via LangGraph workflows
- ▶ External tools accessed through validated APIs
- ▶ Retrieval-Augmented Generation RAG using FAISS

Operational Feature Detection

- ▶ AI-based feature detection in operations
- ▶ Front detection integrated into NinJo
- ▶ High acceptance among forecasters

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00				Joint Dinner	

Live Experiment Tracking — Minimal Example

Goal

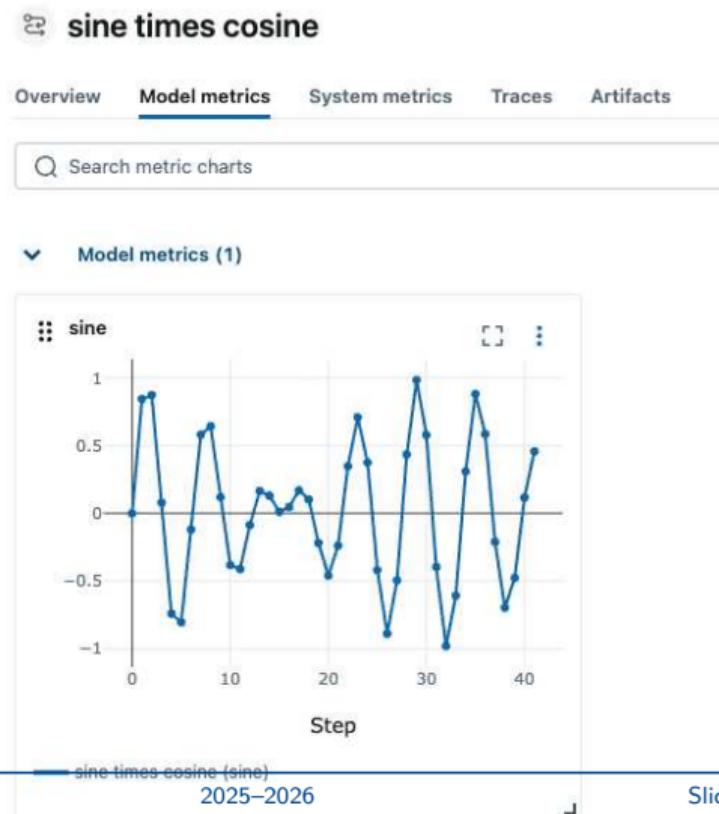
- ▶ Run a simple Python script
- ▶ Log values every second
- ▶ Observe results *live* in MLflow

Key idea

- ▶ No model
- ▶ No training
- ▶ Just time-dependent metrics

This example uses the notebook:

- ▶ `1_live_tracking.ipynb`



What Is Logged — And What Is Not

Logged quantities

- ▶ Time step (logical step index)
- ▶ $\sin(t) \cdot \cos(t/10)$

Not logged

- ▶ No loss
- ▶ No gradients
- ▶ No model parameters

Interpretation

- ▶ MLflow tracks *numbers over time*
- ▶ The meaning is entirely user-defined

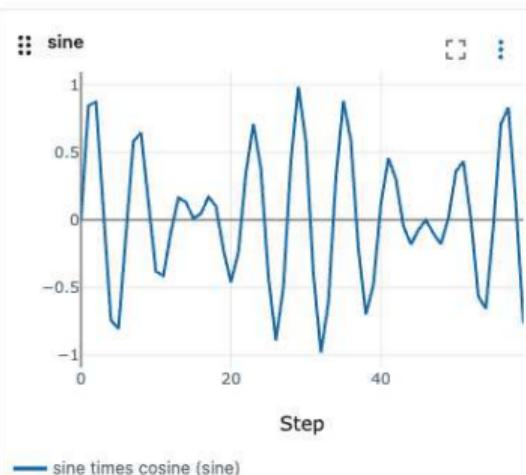
Live Sine Demo > Runs >

 sine times cosine

Overview Model metrics System metrics Traces Artifacts

Search metric charts

▼ Model metrics (1)



What This Example Demonstrates

MLflow is passive

- ▶ Python code runs independently
- ▶ MLflow only records what it is told

Live monitoring

- ▶ Metrics appear while code is running
- ▶ UI polls the backend periodically

Key separation

- ▶ Execution ≠ visualization

Mental model



No direct coupling between:

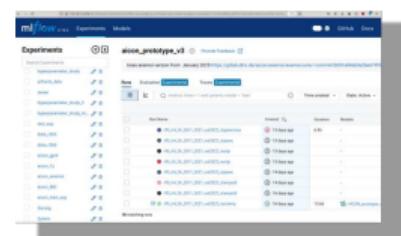
- ▶ execution
- ▶ user interface

Why Experiment Tracking?

- ▶ Repeated training runs
- ▶ Changing hyperparameters
- ▶ Evolving code versions
- ▶ Different datasets

Core requirement

- ▶ reproducibility
- ▶ transparency



Typical problems without tracking

- ▶ Best parameter settings get lost
- ▶ Results cannot be reproduced
- ▶ Missing or incomplete metadata
- ▶ Knowledge is locked in scripts

Common symptoms:

- ▶ many scripts with unclear differences
- ▶ results that cannot be explained later

Artifacts: Persisting Results Beyond Numbers

What are artifacts?

- ▶ Files produced during or after an experiment
- ▶ Stored together with parameters and metrics
- ▶ Provide **context** and **interpretability**

Typical artifacts in ML applications

- ▶ Trained model files (.pt, .onnx, .pkl)
- ▶ Diagnostic plots (loss curves, skill scores)
- ▶ Evaluation outputs (tables, reports)
- ▶ Configuration snapshots (YAML, JSON)
- ▶ Logs and summaries

Why artifacts matter

- ▶ Results are **explainable**
- ▶ Decisions become **auditable**
- ▶ Knowledge is not **lost in scripts**

How artifacts fit into the workflow

- ▶ Research and development
- ▶ Repeated experimentation
- ▶ Comparison and selection of models
- ▶ Preparing models for later use

Key idea

- ▶ Metrics answer **how well**
- ▶ Artifacts answer **why**

Core Objects: Experiments and Runs

Experiment

- ▶ Logical container for related executions
- ▶ Identified by name
- ▶ Groups comparable runs

Key properties

- ▶ Named
- ▶ Timestamped
- ▶ Reproducible

Mental model

Run

- ▶ One concrete execution of code
- ▶ Has a unique ID
- ▶ Records parameters, metrics, files

Experiment

▷ Run₁, Run₂, ...

Runs differ by:

- ▶ parameters
- ▶ code state
- ▶ data

Comparing Parallel Runs

Setup

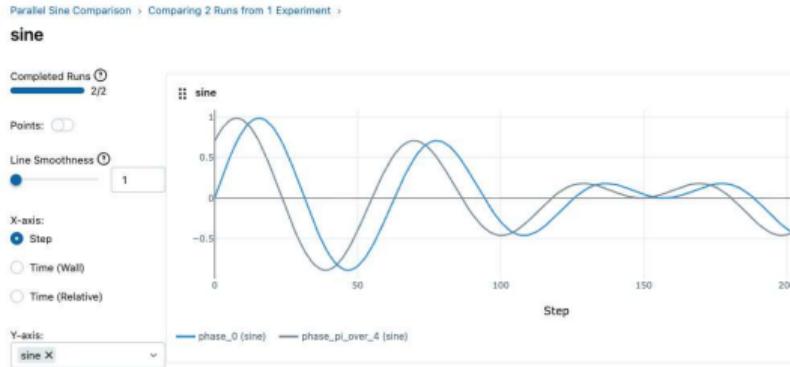
- ▶ One experiment
- ▶ Two independent runs
- ▶ Same metric, same step axis

Difference between runs

- ▶ Single parameter: **phase**
- ▶ Everything else identical

Key observation

- ▶ MLflow overlays both curves automatically
- ▶ No plotting code required



```
def run_sine(phase, name):  
    mlflow.set_tracking_uri("http://localhost:5000")  
    mlflow.set_experiment("Parallel Sine Comparison")  
  
    with mlflow.start_run(run_name=name):  
        mlflow.log_param("phase", phase)  
        for step in range(240):  
            t = step * 0.1  
            value = math.sin(t + phase)*math.cos((t+phase)/10)  
            mlflow.log_metric("sine", value, step=step)  
            time.sleep(1)
```

Tracking URI

Definition

- ▶ URI = Uniform Resource Identifier
- ▶ address of the MLflow tracking backend
- ▶ Decides where runs are written
- ▶ Destination of
 - ▶ experiments
 - ▶ runs
 - ▶ metrics
 - ▶ artifacts

Keep untouched:

- ▶ Training code
- ▶ Logging calls

How it is set in code

```
1 import mlflow
2
3 mlflow.set_tracking_uri(
4     "http://localhost:5000")
```

Typical values

- ▶ file:./mlruns for local storage only
- ▶ http://localhost:5000 for using the mlflow server

The MLflow tracking behavior is fully determined by the tracking URI. If the URI is set to `sqlite:///mlflow.db`, the Python process writes experiment metadata directly into a local SQLite database, without using any HTTP communication or server process. In contrast, if the URI is set to `http://localhost:5000`, all tracking data is sent via HTTP to a running MLflow server, which then stores the results using its configured backend store (e.g. SQLite).

Minimal Logging Example

Essential steps

- ▶ Select an experiment
- ▶ Start a run
- ▶ Log parameters
- ▶ Log metrics

Key idea

- ▶ One run = one execution
- ▶ Everything else is optional

Upload Artefact

```
1 mlflow.log_artifact("a.png")
```

Minimal MLflow code

```
1 import mlflow
2
3 mlflow.set_experiment("Demo")
4
5 with mlflow.start_run():
6     mlflow.log_param("lr", 1e-3)
7     mlflow.log_metric("loss", 0.42)
```

What this code does

- ▶ Creates or selects the experiment Demo
- ▶ Opens a new run with a unique ID
- ▶ Stores one parameter and one metric
- ▶ Makes the run visible in the MLflow UI

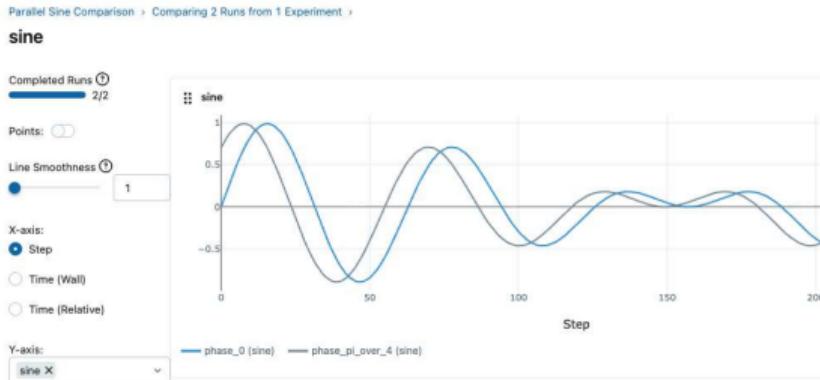
Inspecting Results in the MLflow UI

Grouping principle

- ▶ Runs are grouped by experiment
- ▶ Experiment name defines the comparison scope

Within one experiment

- ▶ Select one or more runs
- ▶ Compare metrics
- ▶ Inspect parameters
- ▶ Browse artifacts



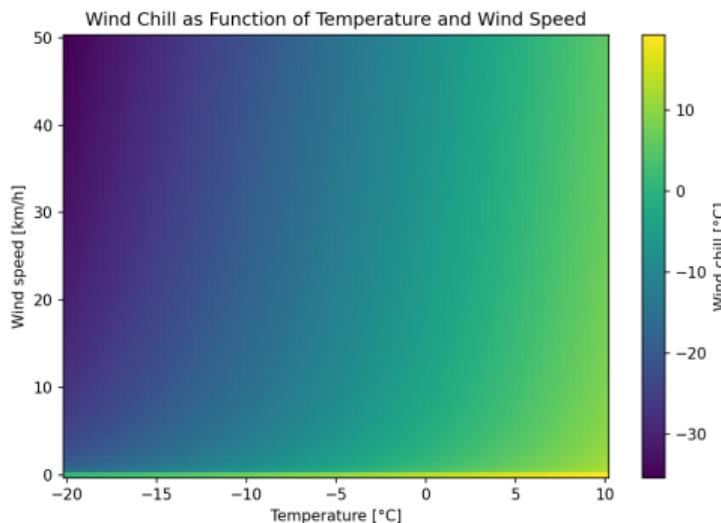
Example Problem: Wind Chill Regression

Problem setup

- ▶ Supervised regression task
- ▶ Two inputs: temperature, wind speed
- ▶ One target: wind chill

Why this example

- ▶ Simple but non-trivial
- ▶ Continuous target variable
- ▶ Suitable for long training runs



Wind Chill Surface

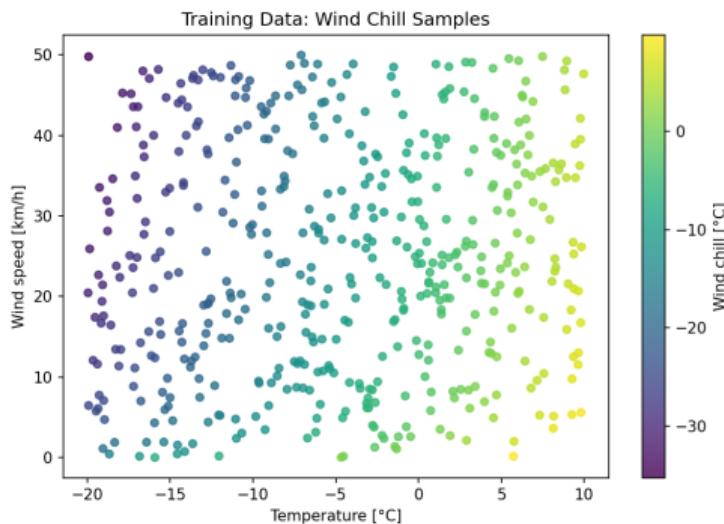
Data Generation and Preprocessing

Synthetic dataset

- ▶ Random temperature values
- ▶ Random wind speed values
- ▶ Wind chill computed from physical formula

Preparation

- ▶ Stack inputs into feature vectors
- ▶ Convert to tensors
- ▶ No normalization tricks required



Input–output relationship

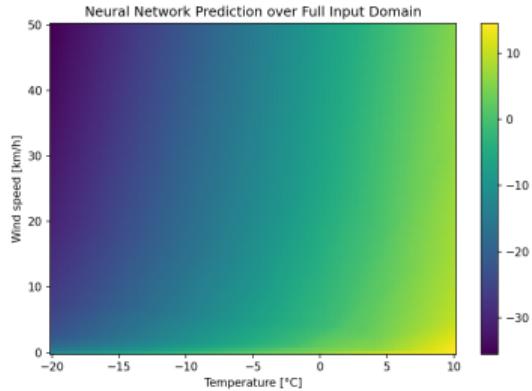
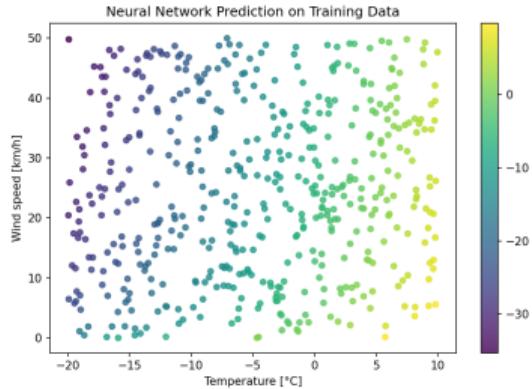
Model Definition

Neural network

- ▶ Feedforward architecture
- ▶ Two hidden layers
- ▶ Scalar regression output

Design choice

- ▶ Model kept intentionally simple
- ▶ Focus is on tracking, not architecture



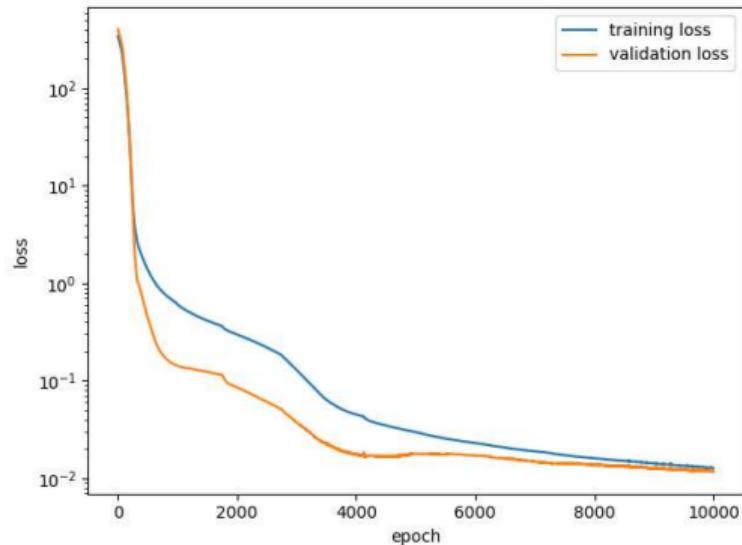
Training with MLflow Logging

What is logged

- ▶ Model hyperparameters
- ▶ Training loss per epoch
- ▶ Continuous progress information

Key benefit

- ▶ Training is observable while running
- ▶ Long runs become transparent

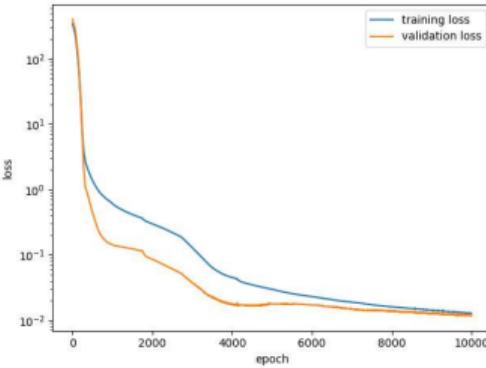


Live loss tracking in MLflow UI

Persisting Results: Artifacts and Models

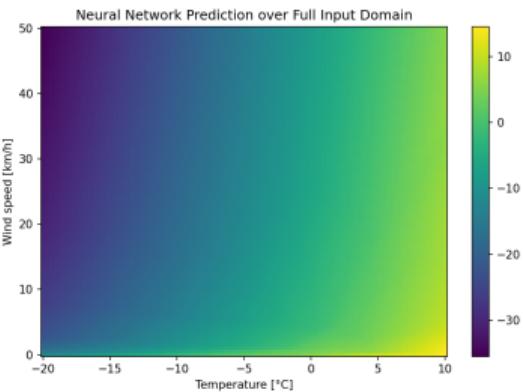
Artifacts

- ▶ Loss curves
- ▶ Diagnostic plots
- ▶ Configuration snapshots



Models

- ▶ Serialized network weights
- ▶ Input–output signature
- ▶ Reusable for later deployment



Local UI Mode

Purpose

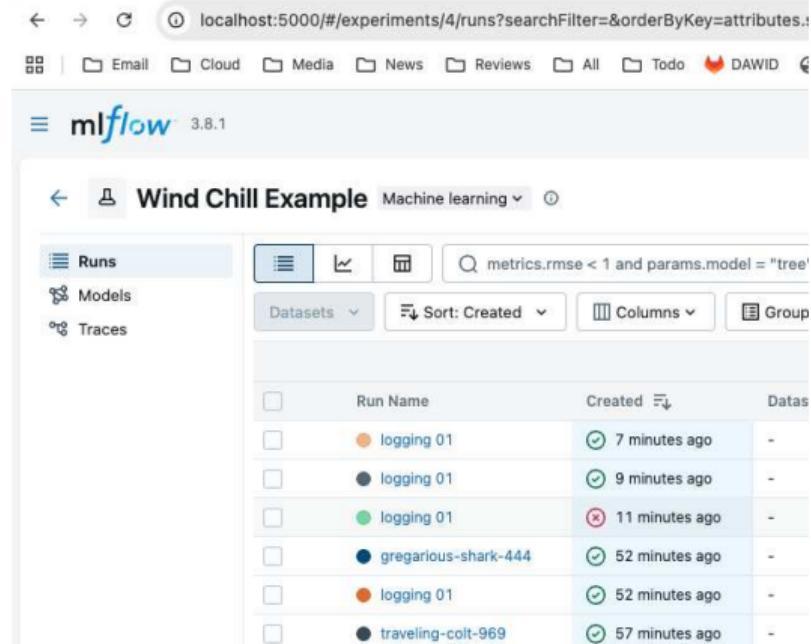
- ▶ Inspect completed or running experiments
- ▶ Visualize metrics and parameters
- ▶ Single-user exploration

Command

- ▶ `mlflow ui`

Key property

- ▶ Read-only visualization
- ▶ No training happens here



Run Name	Created	Data
logging 01	7 minutes ago	-
logging 01	9 minutes ago	-
logging 01	11 minutes ago	-
gregarious-shark-444	52 minutes ago	-
logging 01	52 minutes ago	-
traveling-colt-969	57 minutes ago	-

By default, the UI reads from the locally configured tracking backend (e.g. `mlruns/` or a local SQLite database).

Server Mode for Collaboration

Why server mode?

- ▶ Multiple users
- ▶ Multiple machines
- ▶ Shared experiment history

Command

- ▶ `mlflow server`

Important point

- ▶ Training code stays unchanged
- ▶ Only the tracking URI changes

Server startup

```
1 mlflow server \\  
2   --host 0.0.0.0 \\  
3   --port 5000
```

Clients connect via `http://<server>:5000` and log runs remotely.

Server Setup

An MLflow server can run on any host reachable by domain name or IP address; clients only need the tracking URI to connect.

Background and Persistent Execution

Why this matters

- ▶ Long-running experiments
- ▶ Remote machines
- ▶ Logout-safe operation

Common patterns

```
1 mlflow server ... &
2
3 nohup mlflow server ... &
4
5 screen -S mlflow
```

Servers must survive terminals.

These mechanisms are generic system tools for running long-lived services; MLflow behaves like any other server process.

Use screen or tmux for interactive re-attachment and monitoring.

```
tmux new -s mlflow          # start a persistent session
mlflow server --host 0.0.0.0 --port 5000
Ctrl+B D                   # detach from session
tmux attach -t mlflow       # reconnect later
```

Authentication and Credentials

Why authentication?

- ▶ Shared infrastructure
- ▶ Access control
- ▶ Separation of users

Credential handling

- ▶ Server: auth configuration
- ▶ Client: credential file

In `auth_config.ini`:

```
[mlflow]
auth_enabled = true
database_uri = sqlite:///mlflow_auth.db
admin_username = admin
admin_password = ChangeThisPassword888!
```

MLflow support

- ▶ Basic authentication
- ▶ Server-side user management

In `auth_config.ini`:

```
export MLFLOW_AUTH_CONFIG_PATH=auth_config.ini
export MLFLOW_FLASK_SERVER_SECRET_KEY='8346918649864986498'
mlflow server --host 0.0.0.0 --port 5000 --app-name basic-auth
```

Using a Trained Model from MLflow

Key idea

- ▶ Trained models are stored as MLflow artifacts
- ▶ A model can be loaded in any notebook or script
- ▶ No access to training code is required

Workflow

- ▶ Identify the model via its run ID
- ▶ Load the model using an MLflow URI
- ▶ Apply forward inference on new data
- ▶ Optionally log new artifacts back to the same run

```
[3]: import mlflow.pytorch
import torch

# Point to the model via run_id
run_id = "50848f8f09f1471bb4070ea8de9076e3"

model = mlflow.pytorch.load_model(
    f"runs:{run_id}/model"
)

model.eval()

print(model)
```

Downloading artifacts: 0% 0/1 [00:00<?, ?it/s]

Downloading artifacts: 100% 8/8 [00:00<00:00, 920.79it/s]

wind_chill_model
(fc1): Linear(in_features=2, out_features=20, bias=True)
(fc2): Linear(in_features=20, out_features=20, bias=True)
(fc3): Linear(in_features=20, out_features=1, bias=True)
(relu): ReLU()

Loading a model directly from MLflow

Reproducibility

- ▶ Architecture, weights, environment stored
- ▶ Input/output signature enforced
- ▶ Full traceability to the training run

Model Registry: What Is Stored

Core concept

- ▶ A registered model version references exactly **one run**
- ▶ Model versions are immutable
- ▶ Full traceability to data and code

Stored per model version

- ▶ Trained parameters (weights)
- ▶ Model architecture
- ▶ MLFlow flavor metadata
- ▶ Environment specification
- ▶ Optional input/output signature

How the architecture is stored

- ▶ Serialized via the **MLFlow flavor**
- ▶ For PyTorch:
 - ▶ Python class structure
 - ▶ State dictionary (parameter tensors)
 - ▶ Loader reference (`mlflow.pytorch`)
- ▶ Architecture is reconstructed at load time

When a model is logged from PyTorch, MLFlow stores all information needed to reconstruct the complete neural network at the PyTorch level, including the network structure, trained parameters, and the code required to load the model.

Model Versions and Parallel Development

Versioning principle

- ▶ Each model version corresponds to one run
- ▶ Versions are ordered but independent
- ▶ Older versions remain accessible

Why this matters

- ▶ Multiple ideas explored in parallel
- ▶ No overwriting of previous results
- ▶ Safe comparison and rollback

What changes between versions

- ▶ Training data
- ▶ Hyperparameters
- ▶ Network architecture
- ▶ Optimization settings

Model evolution is additive, not destructive.

What “safe rollback” means

- ▶ Older model versions remain unchanged
- ▶ Switching back requires no retraining
- ▶ Deployment can point to any previous version

Model Lineage and Training Continuation

What MLFlow tracks automatically

- ▶ Each run has a unique identifier
- ▶ Each model version points to exactly one run
- ▶ Parameters, metrics, artifacts are immutable

What MLFlow does not infer

- ▶ No automatic parent-child model tree
- ▶ No implicit notion of fine-tuning or continuation

How lineage is expressed explicitly

- ▶ Log the parent run ID as a parameter or tag
- ▶ Log the source model URI when continuing training
- ▶ Use naming and tagging conventions

Example intent

- ▶ “run B continues training from run A”
- ▶ “model v3 fine-tuned from model v1”

Lineage is recorded by metadata, not guessed.

```
import mlflow
with mlflow.start_run():
    mlflow.set_tag("parent_run_id", "50848f8f09f1471bb4070ea8de9076e3")
    mlflow.set_tag("source_model", "runs:/50848f8f09f1471bb4070ea8de9076e3/model")
    mlflow.set_tag("training_type", "fine_tuning")
```

What MLFlow Stores (and What It Does Not)

Stored explicitly

- ▶ Parameters and metrics
- ▶ Artifacts (plots, files, models)
- ▶ Model binaries (framework-specific)
- ▶ Environment YAML files
(only when a model is logged)

YAML handling

- ▶ Generated during `log_model`
- ▶ Stored as part of the model artifact
- ▶ Enables reproducible loading

Not stored automatically

- ▶ Full runtime environment
- ▶ OS-level dependencies
- ▶ GPUs, drivers, system libraries

Important boundary

- ▶ MLFlow does **not infer** environments
- ▶ Reproducibility requires explicit logging

MLFlow records what you declare, not what it guesses.

From Tracking to Operations

What MLFlow enables

- ▶ Versioned model artifacts
- ▶ Stable model identifiers
- ▶ Promotion via registry stages
- ▶ Rollback to known-good models

Operational contract

- ▶ Training produces runs
- ▶ Registry exposes deployable models
- ▶ Inference consumes model URLs

What MLFlow is not

- ▶ Not a deployment platform
- ▶ Not a monitoring system
- ▶ Not a CI/CD engine

Typical integration

- ▶ MLFlow for model lifecycle
- ▶ External systems for serving
- ▶ External monitoring and alerts

MLFlow provides the training state of a model,
not its operational execution.

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

MLOps: From Machine Learning to Operations

Why this lecture

- ▶ Machine learning models are **no longer research prototypes**
- ▶ AI systems increasingly enter **operational environments**
- ▶ Reliability, reproducibility and traceability become critical

Traditional ML focuses on *training*.
Operations focus on *stability*.

MLOps connects both worlds.

Key question

- ▶ How do we turn ML code into an operational system?

This includes

- ▶ controlled software environments
- ▶ automated build and deployment
- ▶ clear separation of roles and responsibilities
- ▶ safe execution on HPC and production systems

This lecture focuses on *processes*, not algorithms.
But also lots of **continuous integration (CI)**.

Why Machine Learning Needs Operations

Classical software assumptions

- ▶ Code fully defines behavior
- ▶ Same input → same output
- ▶ Changes are explicit and infrequent

Once deployed, behavior is largely predictable and stable.



Data Improvement

More & better training data



Bug Fixes

Fix errors in code/pipeline

Machine learning reality

- ▶ Behavior emerges from **data**
- ▶ Outputs are **statistical, not deterministic**
- ▶ Models **degrade as data distributions change**

Operational consequences

- ▶ Retraining becomes part of operations
- ▶ Monitoring must include **model performance**
- ▶ Reproducibility extends **beyond source code**



Feature Engineering
New features & transformations



Architecture
Improve model architecture



Hyperparameters
Optimize training parameters

Machine learning systems require operational control from the very beginning.

What Is MLOps?

Basic idea

- ▶ MLOps extends DevOps principles to machine learning
- ▶ Focus shifts from code alone to data and models
- ▶ The full model lifecycle becomes operational

MLOps treats ML systems as *long-lived services*, not one-off experiments.

What MLOps manages

- ▶ Data pipelines and preprocessing
- ▶ Training and validation workflows
- ▶ Model versions and metadata
- ▶ Deployment, monitoring and rollback

Why this is necessary

- ▶ ML behavior changes even when code does not
- ▶ Manual processes do not scale
- ▶ Missing control leads to silent failures

MLOps is not a tool — it is an engineering discipline.

DevOps vs MLOps: What Really Changes

DevOps perspective

- ▶ Software behavior defined by **source code**
- ▶ Releases are **explicit and controlled**
- ▶ Bugs are fixed by changing code

How do we monitor systems, and control changes, fixes, improvements?



MLOps perspective

- ▶ Behavior emerges from **data and training**
- ▶ Models evolve through **retraining**
- ▶ Performance can change **without code changes**

Operational impact

- ▶ Data becomes a **first-class artifact**
- ▶ Validation must be **continuous**
- ▶ Rollback applies to **models**, not just code

MLOps extends DevOps — it does not replace it.

The Classical MLOps Cycle

From experiment to operation

- ▶ Data ingestion and preprocessing
- ▶ Model training and validation
- ▶ Deployment into an operational environment

Operational feedback loop

- ▶ Monitoring of system and model performance
- ▶ Detection of drift and degradation
- ▶ Triggering retraining or rollback

This cycle extends the classical DevOps loop by explicitly including data and models.

Key property

- ▶ The cycle is continuous, not linear
- ▶ Operations actively influence development

Modern MLOps has many elements of typical NWP development cycles. NWP also integrates data, e.g. orography, canopy layers, and real data through observations.

MLOps is a closed loop, not a one-time deployment.

How Automation Is Achieved in MLOps

Automation principles

- ▶ Each step is explicitly defined and scripted
- ▶ Execution is trigger-based, not manual
- ▶ Results are logged and versioned automatically

Automation replaces informal procedures by repeatable execution rules.

Technical mechanisms

- ▶ CI/CD pipelines trigger builds and checks
- ▶ Version control tracks code, configs, and metadata
- ▶ Registries store models and runtime artifacts
- ▶ Containers freeze execution environments

Key effect

- ▶ Humans decide what should happen
- ▶ Systems enforce how it happens

Automation makes MLOps scalable, auditable, and safe.

CI and CD in MLOps

Continuous Integration (CI)

CI = Every change is automatically built, checked, and validated

- ▶ Triggered by commits to code or configuration
- ▶ Executes tests, linters, and consistency checks
- ▶ Detects problems early and reproducibly

Continuous Delivery / Deployment (CD)

CD = Validated artifacts are automatically prepared for operation

- ▶ Packages software, models, and environments
- ▶ Produces versioned, deployable artifacts
- ▶ Enables controlled rollout or rollback

CI/CD in MLOps

- ▶ CI validates logic and structure
- ▶ CD delivers operational artifacts

Important distinction

- ▶ CI/CD does not decide when models are used
- ▶ Operations retain execution control

Automation without loss of control.

MLOps as a Role-Based Pipeline

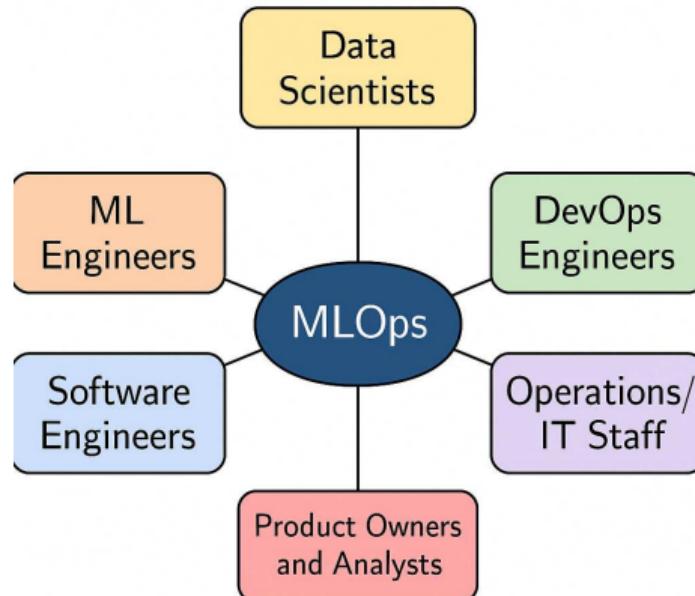
It is a **sequence of responsibilities** that together form a **complete lifecycle**:

- ▶ data preparation and feature design
- ▶ model development and validation
- ▶ packaging and deployment
- ▶ monitoring, feedback and retraining

Each of these steps:

- ▶ requires **different skills**,
- ▶ has **different risks**,
- ▶ and implies **different ownership**.

A useful way to understand MLOps is therefore as a **role game**, where different actors take responsibility for different parts of the pipeline.



Roles in MLOps Environments

Who Defines What “Good” Means?

In MLOps, quality is not a technical concept .

It is defined by:

- ▶ physical constraints,
- ▶ domain knowledge,
- ▶ operational requirements,
- ▶ user expectations.

These aspects cannot be learned from data alone.

Therefore:

- ▶ Domain experts define what “good” means ,
- ▶ ML systems optimize *towards* this definition,
- ▶ Operations validate it against reality.

Without a domain expertise, metrics are meaningless.

Domain Expert

- ▶ Defines quality and success criteria
- ▶ Specifies relevant metrics and constraints
- ▶ Interprets model behaviour in context
- ▶ Does not implement models or pipelines

Roles in the MLOps Pipeline

Data Scientist

- ▶ Develops models to optimize domain criteria
- ▶ Explores data and hypotheses
- ▶ Translates questions into ML problems
- ▶ **Does not decide operational relevance**

ML Engineer

- ▶ Turns models into software artifacts
- ▶ Builds training and inference pipelines
- ▶ **Ensures reproducibility and versioning**
- ▶ Bridges research and operations

DevOps / Platform Engineer

- ▶ Provides CI/CD and build infrastructure
- ▶ Manages containers and registries
- ▶ Ensures security and scalability
- ▶ **Does not tune models**

Domain Expert / Operations / Users

- ▶ Run models in production
- ▶ Monitor behaviour and failures
- ▶ Validate outputs against reality
- ▶ **Do not modify code or models**

Roles should be clear, people may be the same.

Starting Point: A Baseline ML Application

We start with a minimal working ML example.

The goal at this stage is simple:

- ▶ load a pretrained model,
- ▶ apply it to input data,
- ▶ inspect the output.

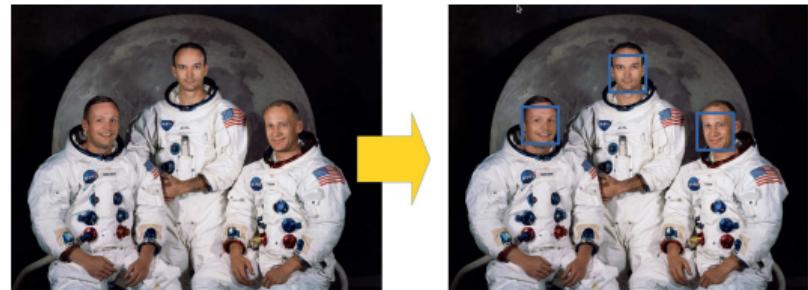
This corresponds to running the notebook:

- ▶ 00_face-detection-onnx.py

At this point:

- ▶ the method works,
- ▶ results can be inspected visually.

Nothing here is operational yet.



The notebook instantiates a pretrained face-detection model and applies it directly to an input image.

```
def mark_faces(image_filename):  
    """Mark all faces recognized in the image"""\n    image = PIL.Image.open(image_filename)  
  
    faces = detect_faces(image)  
  
    render_data = detections_to_render_data(  
        faces, bounds_color=Colors.GREEN, line_w
```



Input image

Raw input data, loaded via PIL. No preprocessing, no metadata, no assumptions beyond file availability.



Model output

Bounding boxes produced by a pretrained face-detection model and rendered directly onto the image.

This is pure inference: load → detect → render.

Docker: Installation and Input Files

If you have **admin rights** on your computer, installation of docker is easy.

```
brew install --cask docker
```

Step 1: Install Docker

- ▶ Install **Docker Desktop** (Mac)
- ▶ Start Docker service
- ▶ Verify installation:

Check Docker

```
1 docker --version
2 docker run hello-world
```

If this works, Docker is **ready to use**.

Step 2: Python code to run in Docker

02_code_to_execute_in_docker.py

```
1 from pathlib import Path
2
3 print("Hello from Docker")
4
5 out = Path("docker_output.txt")
6 out.write_text("Generated inside
    Docker\n")
7
8 print("Wrote:", out)
```

This is **ordinary Python**. **No Docker logic** inside the code.

Dockerfile, Image Build, and Execution

Step 4: Build and run

Step 3: Define the container

```
FROM python:3.11-slim
WORKDIR /app
COPY 02_code_to_execute_in_docker.py /app/
CMD ["python", "02_code_to_execute_in_docker.py"]
```

Docker **freezes the runtime environment**:

- ▶ operating system (Debian slim)
- ▶ Python version (3.11.x)
- ▶ system libraries and pip

A Docker image already acts as a virtual environment.

No venv activation is required; packages installed in the image are isolated by default.

Build image

```
1 docker build -t \
2   python-hello-docker \
3   -f 02_dockerfile.txt .
```

Run container

```
1 docker run --rm \
2   -v "$(pwd):/app" \
3   python-hello-docker
```

Volume mount exposes results. Without **-v**, files disappear.

Building Docker Containers on GitHub

General principle

- ▶ Docker containers do **not need** to be built locally
- ▶ GitHub provides **CI runners** with Docker installed
- ▶ A container can be built **directly on the platform**

The build is moved from the laptop to the platform.

Concrete example (this lecture)

- ▶ Python script generates a plot
- ▶ Dockerfile defines the runtime
- ▶ GitHub Actions builds the image
- ▶ Image is pushed to a **container registry**

Result

- ▶ Reproducible container image
- ▶ Independent of local setup
- ▶ Ready for download and execution

Code lives in Git.

Images live in the registry.

What is required

- ▶ Source code
- ▶ A **Dockerfile**
- ▶ A CI workflow definition

CI builds containers as first-class artifacts.

Downloading and Running a Container Image

Pulling a container image

- ▶ Images are downloaded via Docker CLI
- ▶ Private images require authentication
- ▶ Access is granted via registry tokens

Download image

```
1 docker pull ghcr.io/eumetnet-e  
    -ai/docker-plot-03:latest
```

Running the container

```
docker run --rm \  
-v "$(pwd)/03_docker_plot:/app" \  
ghcr.io/eumetnet-e-ai/docker-plot-03:latest
```

Why the volume mount matters

- ▶ Container files are ephemeral
- ▶ Volume mounts expose results
- ▶ Without -v, outputs disappear

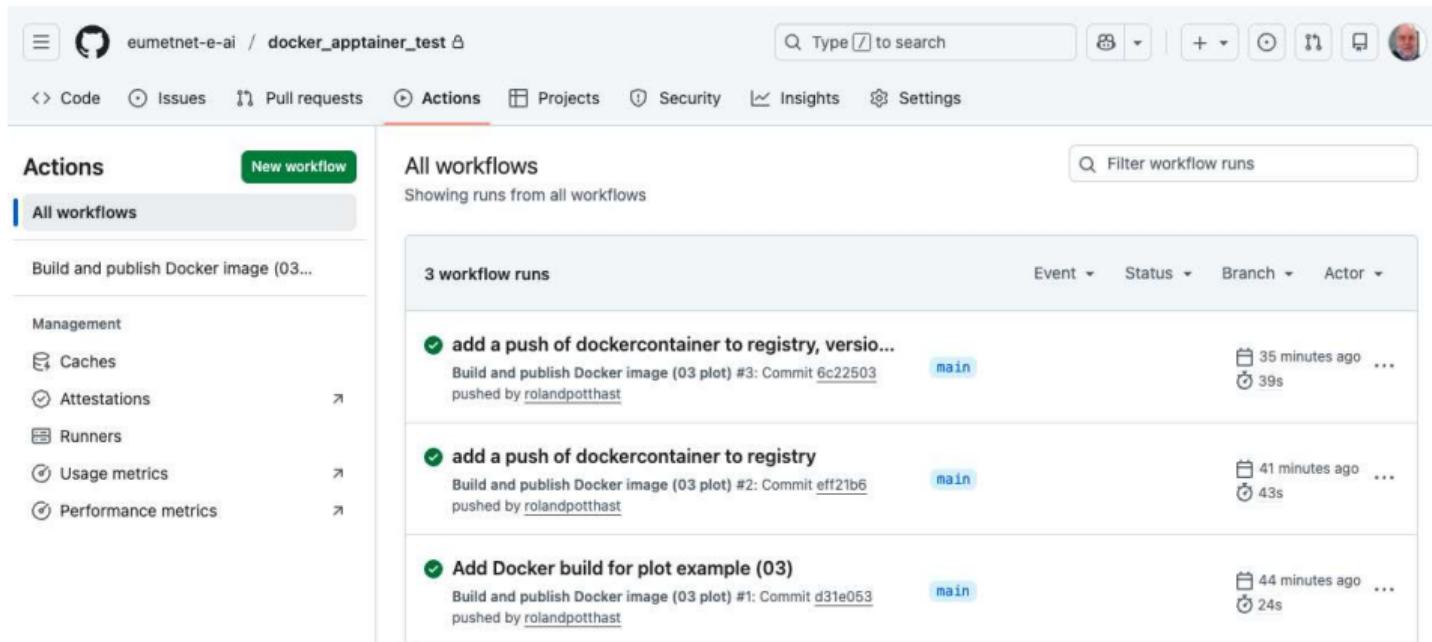
Execution is decoupled from development.

Important distinction

- ▶ Git access \neq registry access
- ▶ Separate permissions for code and containers

Build once, run anywhere — with controlled environments.

Docker Containers Built on GitHub



The screenshot shows a GitHub repository page for 'eumetnet-e-ai / docker_apptainer_test'. The 'Actions' tab is selected, displaying three workflow runs:

Event	Status	Branch	Actor	Time	Actions
add a push of dockercontainer to registry, version 0.1.0	Success	main	rolandpotthast	35 minutes ago	... 39s
add a push of dockercontainer to registry	Success	main	rolandpotthast	41 minutes ago	... 43s
Add Docker build for plot example (03)	Success	main	rolandpotthast	44 minutes ago	... 24s

The sidebar on the left includes sections for Actions, New workflow, All workflows, Build and publish Docker image (03...), Management, Caches, Attestations, Runners, Usage metrics, and Performance metrics.

Source code in Git → CI build on GitHub → Container registry → Pull and run anywhere

GitHub Container Registry (ghcr.io)

What is ghcr.io?

- ▶ GitHub's container registry service
- ▶ Stores Docker / OCI container images
- ▶ Integrated with GitHub repositories and CI

How it works

- ▶ CI builds an image from a Dockerfile
- ▶ Image is pushed to ghcr.io
- ▶ Image is identified by name, tag, and SHA

Containers are artifacts, not source code.

Access and usage

- ▶ Images are pulled via Docker or Apptainer
- ▶ Private images require authentication
- ▶ Access control is separate from Git

Browser access

- ▶ Containers can be inspected in the browser
- ▶ Tags, SHAs, and metadata are visible
- ▶ No direct download button

Click to inspect. Pull to execute.

ghcr.io stores images; runtimes execute them.

Apptainer on HPC: Build vs Run

What happens when pulling Docker images

- ▶ Apptainer converts Docker images into local containers
- ▶ This involves SquashFS compression
- ▶ Compression is memory- and thread-intensive

Typical failure on login nodes

- ▶ Limited memory and process counts
- ▶ SquashFS build cannot create threads

Out of memory (frag_thrd)

Failed to create thread

Important distinction

- ▶ Running containers is lightweight
- ▶ Building containers is a compute task

HPC policy (typical)

- ▶ Login nodes: editing, submitting jobs
- ▶ Compute nodes: builds, compression, heavy work

Apptainer pull = build step, not runtime.

Disk quota is irrelevant — memory and threads decide.

Correct AppTainer Workflow on HPC

Recommended solution

- ▶ Run container pulls as **batch jobs**
- ▶ Request sufficient **memory**
- ▶ Avoid compressed SIF builds on login nodes

Use sandbox format

- ▶ Uncompressed directory container
- ▶ Much lower memory pressure
- ▶ Fully usable with AppTainer

Batch job (concept)

```
1 #PBS -q gp_norm_all
2 #PBS -l memsz_job=128gb
3 #PBS -l cpunum_job=2
4
5 apptainer build --sandbox \
6 my-container \
7 docker://ghcr.io/ORG/IMAGE:TAG
```

Running the container

- ▶ No root privileges
- ▶ Current directory mounted by default
- ▶ Same performance as SIF

HPC rule: build elsewhere, run everywhere.

For production: build SIFs in CI, not on the cluster.

Building Apptainer Containers on GitHub

Motivation

- ▶ HPC systems often **cannot build containers locally**
- ▶ Docker images must be converted to **Apptainer (SIF)**
- ▶ Conversion is **memory-intensive**

Key idea

- ▶ Move the **build step off the HPC**
- ▶ Use **GitHub Actions** as build infrastructure
- ▶ Produce ready-to-run **SIF artifacts**

What GitHub provides

- ▶ Linux build nodes with Docker
- ▶ Sufficient memory for SquashFS
- ▶ Integrated access to **ghcr.io**

Required inputs

- ▶ Docker image in a registry
- ▶ Apptainer installed in CI
- ▶ Authentication token (`read:packages`)

Result: A portable **single-file container**.

CI builds containers, HPC only runs them.

CI Workflow: Docker Image to Apptainer SIF

Automated workflow

- ▶ Triggered after successful Docker build
- ▶ Runs on GitHub Linux runner
- ▶ Converts image to **SIF** format

Main steps within the Github environment

1. Install Apptainer
2. Authenticate to ghcr.io
3. Pull Docker image
4. Convert to **SIF**
5. Upload as artifact

This is **CI/CD** for runtime environments.

Core CI command on GitHub

```
1 apptainer pull \
2   docker-plot-03.sif \
3   docker://ghcr.io/ORG/IMAGE:TAG
```

Distribution

- ▶ SIF stored as CI artifact
- ▶ Downloadable via browser
- ▶ Transfer to HPC via scp

On the cluster

- ▶ **No build step**
- ▶ No Docker required
- ▶ Immediate execution

Triggering the SIF Build via GitHub Actions, YAML

What this workflow does

- ▶ Listens for a successful Docker build
- ▶ Runs only if the image exists
- ▶ Converts Docker → Apptainer

Why this design

- ▶ Avoids duplicate builds
- ▶ Enforces a clean dependency chain
- ▶ CI controls the runtime environment

Docker build is a prerequisite.

This makes the SIF build:

- ▶ deterministic, reproducible
- ▶ traceable to a commit

```
name: Export Apptainer SIF (Example 03)
on:
  workflow_run:
    workflows: ["Build and publish Docker image (03)"]
    types: [completed]
jobs:
  build-sif:
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: read
    steps:
      - name: Install Apptainer
        run: |
          sudo apt-get update
          sudo apt-get install -y \
```

Running the Apptainer Container on the HPC

On the HPC system

- ▶ No Docker available
- ▶ Apptainer pre-installed
- ▶ No build permissions required

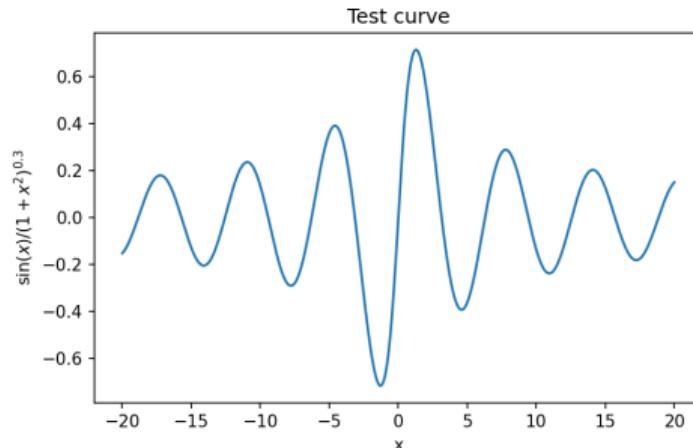
Run the container

HPC execution

```
1 apptainer exec \
2 docker-plot-03.sif python \
3 -B 03_plot_curve.py
```

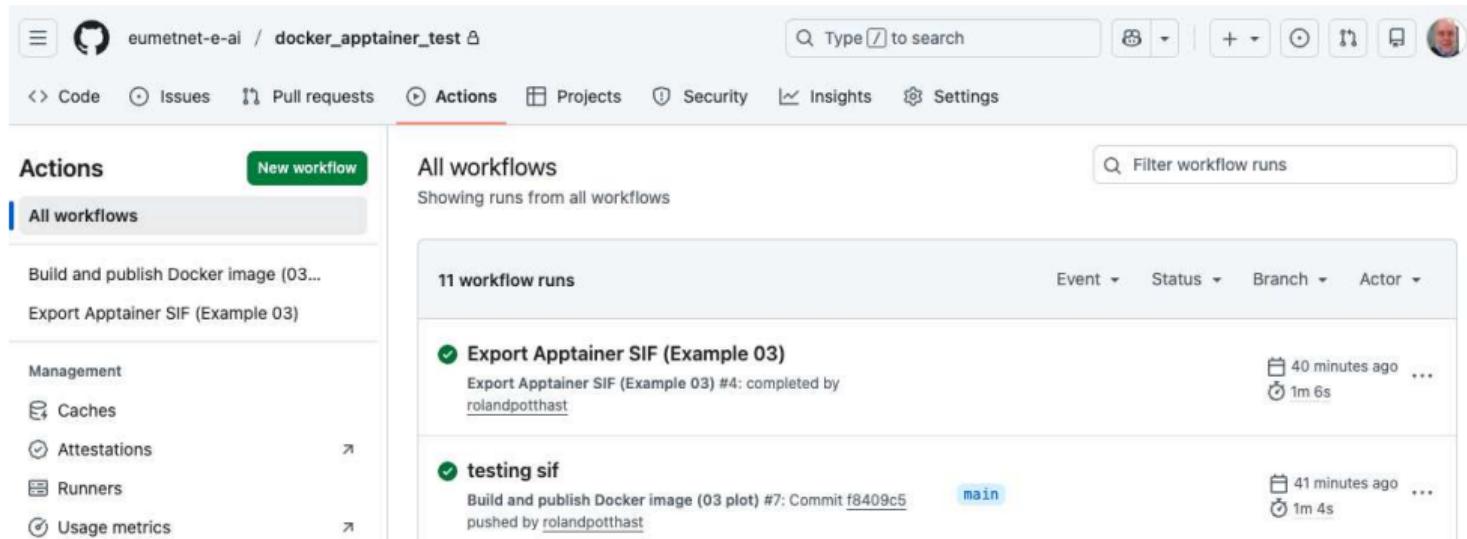
What happens

- ▶ Python code runs inside container
- ▶ Output file written to working directory
- ▶ No external dependencies needed



Same container, different system.

From Docker Image to Apptainer SIF on GitHub



The screenshot shows a GitHub Actions workflow page for the repository `eumetnet-e-ai / docker_apptainer_test`. The left sidebar lists management options like Caches, Attestations, Runners, and Usage metrics. The main area displays the 'Actions' tab, which is currently selected. It shows two workflow runs:

- Export Apptainer SIF (Example 03)**: Completed by `rolandpotthast` 40 minutes ago, 1m 6s ago.
- testing sif**: Completed by `rolandpotthast` 41 minutes ago, 1m 4s ago. This run was triggered by a build and publish Docker image (03 plot) #7: Commit `f8409c5` pushed to the `main` branch.

Why Containers are Essential in ML Frameworks

The core problem in ML

- ▶ ML code depends on:
 - ▶ specific library versions
 - ▶ CUDA / CPU features
 - ▶ system-level dependencies
- ▶ These dependencies **change over time**
- ▶ Results become hard to reproduce

ML models are not standalone artifacts.

What containers provide

- ▶ Encapsulation of:
 - ▶ code
 - ▶ libraries
 - ▶ runtime environment
- ▶ Identical execution on:
 - ▶ laptops
 - ▶ CI systems
 - ▶ HPC clusters

Containers turn models into executable artifacts.

Reproducibility, portability, and controlled execution are **non-negotiable** in ML.

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Why CI/CD Is Not Optional for AI/ML

What continuously changes in AI/ML

- ▶ Model parameters through retraining
- ▶ Training and validation data
- ▶ Feature engineering and preprocessing
- ▶ Hyperparameters and runtime configuration

As a consequence:

- ▶ System behavior is **not fixed**
- ▶ Outputs depend on
code, data, and environment
- ▶ Small changes can have **large effects**

Why manual workflows break down

- ▶ Experiments cannot be reproduced reliably
- ▶ Results depend on undocumented environments or training data
- ▶ Errors surface **late or not at all**
- ▶ Deployment decisions become guesswork

Without automation:

- ▶ Models cannot be trusted operationally
- ▶ Debugging becomes **forensic work**

CI/CD is the mechanism that makes AI/ML systems controllable and trustworthy.

AI/ML Is Not Special — But Updating Becomes Complex

From a software engineering perspective

- ▶ AI/ML systems are still software systems
- ▶ They use standard languages, libraries and toolchains
- ▶ The same DevOps principles apply

There is no special engineering magic in AI/ML:

- ▶ version control
- ▶ testing
- ▶ packaging
- ▶ deployment

CI/CD is needed to manage updates, not to handle “AI magic”.

Where the real complexity comes from

- ▶ Frequent retraining with new data
- ▶ Multiple preprocessing and feature pipelines
- ▶ Changing model architectures
- ▶ Different loss functions and objectives
- ▶ Many runtime and training configurations

As a result:

- ▶ Updates are continuous
- ▶ Reproducibility becomes non-trivial
- ▶ Manual tracking no longer works

AI/ML CI/CD Pipeline

Key components for building robust MLOps workflows

Training Data

- Data Versioning
- Quality Checks
- Labeling Pipeline
- Data Drift Detection

Model Architecture

- Model Registry
- Architecture Versioning
- Hyperparameters
- Code Repository

MLflow

- Experiment Tracking
- Model Registry
- Artifact Storage
- Metrics Logging

CI/CD Pipeline

- Automated Testing
- Model Training
- Build & Package
- Deployment Automation

Verification

- Model Validation
- Performance Tests
- A/B Testing
- Shadow Deployment

Applications

- API Endpoints
- Batch Inference
- Real-time Serving
- Model Integration

Monitoring

- Performance Metrics
- Model Drift
- System Health
- Alert Management

Starting Locally: A Concrete Git Hook Example

Black is an automatic code formatter for Python.

It rewrites Python source code into a single, consistent style, without asking questions.

Minimal pre-commit hook

.git/hooks/pre-commit

```
1 #!/bin/sh
2 pytest || exit 1
3 black .
```

This hook is executed automatically when running:

- ▶ git commit

If tests fail, the commit is blocked.

Git hooks enforce local discipline — not global policy.

What this enforces locally

- ▶ Code must be syntactically correct
- ▶ Tests must pass
- ▶ Code formatting is consistent

Key properties:

- ▶ Runs before code leaves the laptop
- ▶ No CI server involved
- ▶ Immediate feedback to the developer

When Are Git Hooks Executed?

Git commands trigger hooks

Git automatically executes hooks at well-defined points in its workflow.

Common examples:

- ▶ `pre-commit` — before a commit is created
- ▶ `commit-msg` — to validate commit messages
- ▶ `pre-push` — before pushing to a remote

Hooks run **before** Git completes the command.

Effect on the workflow

- ▶ Hook succeeds → Git continues
- ▶ Hook fails → Git aborts the command

This means:

- ▶ Invalid code never enters the repository
- ▶ Errors are caught **immediately**
- ▶ No manual checks are required

Hooks are **deterministic**: same input → same outcome.

Hooks turn Git commands into enforced quality gates.

A Git Hook in Action: What Actually Happens

Before committing

- ▶ Python test file is **poorly formatted**
- ▶ Code is syntactically valid
- ▶ Tests pass when run manually

After running black:

- ▶ Formatting is rewritten automatically
- ▶ No code logic is changed

During git commit

- ▶ pytest is executed automatically
- ▶ black is executed automatically
- ▶ Both run via the **pre-commit hook**

The hook turns a manual checklist into an automatic guarantee.

Outcome:

- ▶ Tests pass
- ▶ Formatting is consistent
- ▶ Commit is **accepted**

```
def add( a ,b ):  
    return a+b  
def test_answer( ):  
    print("Testing add(1, 3) == 4")  
    assert add(1,3)==4  
  
def add(a, b):  
    return a + b  
def test_answer():  
    print("Testing add(1, 3) == 4")  
    assert add(1, 3) == 4
```

Strengths and Limits of Git Hooks

Strengths

- ▶ Extremely fast feedback . Helps you!!
- ▶ No external infrastructure required
- ▶ Works offline
- ▶ Integrated directly into Git commands

Limitations

- ▶ Run only on the developer machine
- ▶ Not enforced across a team
- ▶ Can be bypassed or disabled
- ▶ No neutral execution environment

Git hooks are ideal for:

- ▶ formatting checks
- ▶ unit tests
- ▶ catching trivial mistakes early

As a consequence:

- ▶ Hooks improve discipline
- ▶ But they do **not guarantee quality**

Git hooks accelerate development — CI platforms enforce standards.

Formatting vs Behavior: Two Different Checks

Code formatting (Black)

- ▶ Enforces a consistent style
- ▶ Removes whitespace and layout differences
- ▶ Does **not** change program logic

Example:

black reformats code

```
1 def add( a ,b ):  
2     return a+b
```

Behavior testing (pytest)

- ▶ Checks whether code does the right thing
- ▶ Executes functions and validates results
- ▶ Fails if behavior changes unexpectedly

Formatting makes code readable. Testing makes code **correct**.

becomes:

```
1 def add(a, b):  
2     return a + b
```

Style and correctness are independent concerns.

Code and Test: Defining and Enforcing Behavior

Formatted Code

```
test_example.py
```

```
1 def add(a, b):  
2     return a + b
```

This code:

- ▶ is syntactically correct
- ▶ is well formatted
- ▶ may still be **logically wrong**

Formatting alone cannot guarantee correctness.

**Correct behavior is defined by function tests,
not by appearance.**

Test that enforces behavior

```
test_example.py
```

```
1 def test_add():  
2     assert add(2, 3) == 5
```

This test:

- ▶ executes the function
- ▶ checks the expected result
- ▶ fails automatically if behavior changes

The test can run:

- ▶ locally
- ▶ in Git hooks
- ▶ in CI pipelines

CI/CD Tools: Local Discipline and Correctness

Version control

- ▶ **Git**

Tracks changes to source code and configuration files, enabling branching, merging, and full reconstruction of development history.

- ▶ **GitHub / GitLab**

Hosting platforms for Git repositories, providing collaboration, access control, and CI integration.

Local enforcement

- ▶ **Git hooks**

Local scripts executed automatically on Git events (e.g. pre-commit) to enforce rules before code is committed.

- ▶ **pre-commit framework**

Version-controlled hook manager that ensures identical checks across developers and CI environments.

Code quality and correctness

- ▶ **Black**

Automatic Python formatter that enforces a single, deterministic code style without configuration decisions.

- ▶ **pytest**

Python testing framework that executes test functions and fails automatically when expected behavior is violated.

These tools:

- ▶ run fast

- ▶ provide immediate feedback

- ▶ prevent trivial errors from propagating

Local discipline reduces error rates before CI even starts.

CI/CD Tools: Automation and Execution

CI orchestration

▶ GitHub Actions

Event-driven CI system integrated into GitHub, executing workflows defined in YAML on managed runners.

▶ GitLab CI

Pipeline-based CI system configured via `.gitlab-ci.yml`, supporting self-hosted and specialized runners (e.g. HPC, GPU).

▶ Jenkins

Standalone automation server using scripted pipelines, common in legacy and enterprise environments.

Execution environments

▶ Python virtual environments

Isolate Python dependencies to ensure reproducible runtime behavior.

▶ Containers (Docker, Apptainer)

Package applications and dependencies into portable, reproducible execution units across systems.

Artifacts

▶ CI artifacts and registries

Store outputs such as logs, test reports, trained models, and container images produced during pipelines.

CI platforms enforce rules; environments make them reproducible.

Why Data Formats Matter in AI/ML

Characteristics of training data

- ▶ Very large (GB–TB scale)
- ▶ Multi-dimensional (time, space, channels)
- ▶ Often produced continuously

Typical ML access patterns

- ▶ Repeated sampling of small subsets
- ▶ Random or structured access
- ▶ Parallel reading by many workers

In contrast to classical workflows:

- ▶ Data rarely fits into memory
- ▶ Full sequential reads are uncommon

As a result, the **data format** directly affects:

- ▶ I/O performance
- ▶ Scalability of training
- ▶ Feasibility of distributed ML

Data formats are part of the ML infrastructure, not just storage.

Zarr: Why It Is Attractive for AI/ML

Design principles

- ▶ Chunked, array-based storage
- ▶ Designed for cloud and HPC
- ▶ Partial reads without full downloads

Why this helps ML training

- ▶ Efficient random sampling
- ▶ Scales to distributed workers
- ▶ Reduces I/O bottlenecks

Performance features

- ▶ Parallel reads of independent chunks
- ▶ Flexible chunk layout
- ▶ Compression per chunk

Typical use cases

- ▶ Images and video
- ▶ Satellite and geospatial data
- ▶ Scientific simulation output

Zarr optimizes data access patterns — not the ML logic itself.

Zarr: Limitations and When to Use Alternatives

Limitations

- ▶ Limited native integration with ML frameworks
- ▶ Metadata overhead for many small arrays
- ▶ Not well suited for tabular data
- ▶ Care needed for concurrent writes

Zarr adds complexity that is not always justified.

When to choose alternatives

- ▶ Tabular ML data: Parquet , Arrow
- ▶ TensorFlow pipelines: TFRecord
- ▶ PyTorch streaming: WebDataset
- ▶ Small datasets: NetCDF, HDF5, NPY

Guiding principle

- ▶ Choose formats based on data structure
- ▶ Optimize for access pattern, not fashion

Zarr is powerful when the problem matches the format.

Demo Forecast Dataset (Europe, 5 variables)

Dataset design

We generate a small **synthetic forecast archive** over Europe:

- ▶ grid: lat × lon (regular lat/lon)
- ▶ time axis: `valid_time` for lead times (e.g. 0–120h)
- ▶ variables:
 - ▶ `t2m` (2m temperature)
 - ▶ `u10, v10` (10m wind)
 - ▶ `mslp` (mean sea-level pressure)
 - ▶ `tp` (precipitation)

We mimic the **access patterns** of real NWP:

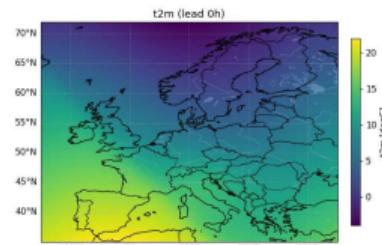
- ▶ map snapshot for one lead time
- ▶ town / station time series extraction
- ▶ local patches for ML training

Notebook outputs

- ▶ Zarr store:
`data/demo_eu_forecast.zarr`
- ▶ first plots: Europe maps
- ▶ metadata: coordinates + attrs

Next

- ▶ generate fields with moving patterns
- ▶ write to Zarr with chunking



Lead Time: 0h

Synthetic Forecast Generation (moving patterns)

Idea

We generate physically plausible
spatio-temporal structure :

- ▶ coherent “weather patterns” move eastward with time
- ▶ correlated variables:
 - ▶ pressure wave → wind via spatial gradients
 - ▶ temperature advected + noise
 - ▶ precipitation linked to fronts (thresholded patterns)

```
# grid (Europe)
lon2d, lat2d = np.meshgrid(lon, lat)
# moving phase (eastward shift with lead time)
phase = kx * lon2d + ky * lat2d - omega * lead_hours[t]

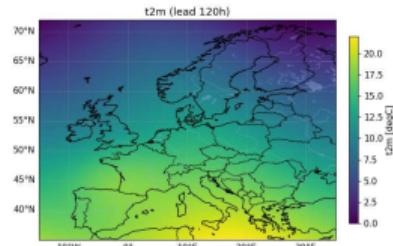
# temperature: advected pattern
t2m[t] = 10.0 + 6.0*np.cos(phase + 0.7) + 0.8*rng.normal(size=lon2d.shape)
```

Result

Fields are realistic enough to demonstrate:

- ▶ map snapshots
- ▶ point extraction
- ▶ patch-based ML sampling

Example snapshot



Lead Time: 120h

Writing the Zarr Archive (chunking for point extraction)

Zarr write: key decisions

We write the dataset as a **chunked archive**:

- ▶ **time chunk = 1:** fast town time series extraction
- ▶ **lat/lon chunks = 60:** moderate patch size for ML sampling

Chunk layout

- ▶ chunks stored as independent files
- ▶ town extraction reads **few chunks only**
- ▶ patches match ML mini-batches

We generate chunks that contain exactly **one lead time** and a **60×60 spatial patch**.

Example:

`(valid_time, lat, lon) = (1, 60, 60)`

```
# -----
# Zarr write
# -----
OUTDIR = "data"
os.makedirs(OUTDIR, exist_ok=True)

ZARR_PATH = os.path.join(OUTDIR, "demo_eu_forecast.zarr")

# best practice: chunk so point extraction is fast
# - valid_time chunk = 1 (fast timeseries reading)
# - lat/lon chunks ~ moderate
chunked = ds.chunk({"valid_time": 1, "lat": 60, "lon": 60})

# overwrite if exists
if os.path.exists(ZARR_PATH):
    import shutil
    shutil.rmtree(ZARR_PATH)

chunked.to_zarr(ZARR_PATH, mode="w", consolidated=True)
print("Wrote:", ZARR_PATH)
```

Open Zarr Archive and Inspect Dataset

Open archive

We open the Zarr store as an `xarray.Dataset`:

- ▶ lazy loading via `dask.array`
- ▶ chunk layout is preserved:
`(valid_time, lat, lon) = (1, 60, 60)`

Dataset structure

- ▶ dimensions: `valid_time=41, lat=121, lon=181`
- ▶ variables: `t2m, u10, v10, mslp, tp`
- ▶ coordinate: `lead_time` as `timedelta`

Key observation

Even though the archive is chunked, `xarray` provides a `single logical dataset` interface.

Example:

```
ZARR_PATH = "data/demo_eu_forecast.zarr"  
ds = xr.open_zarr(ZARR_PATH, consolidated=True)
```

```
xarray.Dataset  
-> Dimensions: (valid_time: 41, lat: 121, lon: 181)  
▼ Coordinates:  
  valid_time (valid_time) datetime64[ns] 2026-01-10 ... 2026-01-15  
  lat (lat) float64 35.0 35.31 35.62 ... 71.69 72.0  
  lon (lon) float64 -15.0 -14.72 -14.44 ... 34.72 35.0  
  init_time () datetime64[ns] ...  
  lead_time (valid_time) timedelta64[s] dask.array<chunksize={1}, meta=np...  
▼ Data variables:  
  msdp (valid_time, lat, lon) float32 dask.array<chunksize={1, 60, 60}, m...  
  t2m (valid_time, lat, lon) float32 dask.array<chunksize={1, 60, 60}, m...  
  tp (valid_time, lat, lon) float32 dask.array<chunksize={1, 60, 60}, m...  
  u10 (valid_time, lat, lon) float32 dask.array<chunksize={1, 60, 60}, m...  
  v10 (valid_time, lat, lon) float32 dask.array<chunksize={1, 60, 60}, m...  
▼ Attributes:  
  title : Synthetic European Forecast Demo  
  institution : Demo dataset for Zarr tutorial  
  history : Created in Notebook 04
```

Dataset preview: dimensions, coords, variables, and Dask chunks.

Town Forecast Extraction (nearest grid point)

Offline town database

We use a small built-in town table:

- ▶ no internet required
- ▶ town → (lat,lon)

Point extraction

From the full dataset we extract a single point forecast :

- ▶ use nearest neighbor selection:
`ds.sel(..., method="nearest")`
- ▶ result: dataset with only dimension `valid_time`

Example:

```
TOWNS = {  
    "Berlin": (52.5200, 13.4050), ... }  
  
pt = ds.sel(lat=lat0, lon=lon0, method="nearest")
```

```
xarray.Dataset  
  
Dimensions: (valid_time: 41)  
Coordinates:  
  valid_time (valid_time) datetime64[ns] 2026-01-10 ... 2026-01-15  
  init_time () datetime64[ns] ...  
  lat () float64 52.58  
  lead_time (valid_time) timedelta64[1s] dask.array<chunksize=(1,), meta=np.ndarray>  
  lon () float64 13.33  
  
Data variables:  
  mslp (valid_time) float32 dask.array<chunksize=(1,), meta=np.ndarray>  
  t2m (valid_time) float32 dask.array<chunksize=(1,), meta=np.ndarray>  
  tp (valid_time) float32 dask.array<chunksize=(1,), meta=np.ndarray>  
  u10 (valid_time) float32 dask.array<chunksize=(1,), meta=np.ndarray>  
  v10 (valid_time) float32 dask.array<chunksize=(1,), meta=np.ndarray>  
  
Attributes:  
  title : Synthetic European Forecast Demo  
  institution : Demo dataset for Zarr tutorial  
  history : Created In Notebook 04
```

Output

A compact dataset with time series for: `t2m`, `u10`, `v10`, `mslp`, `tp`

Point forecast: only dimension `valid_time` remains.

Town Forecast Dynamics (build time series table)

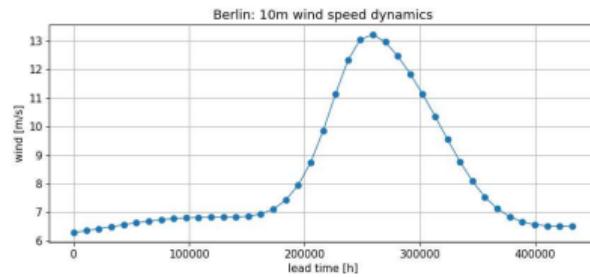
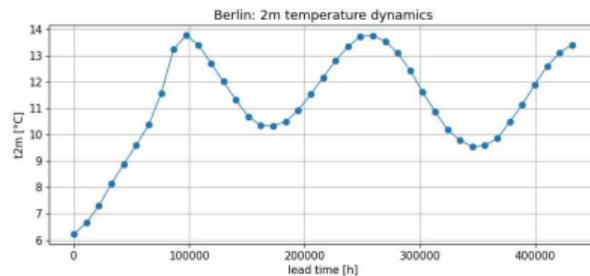
Prepare time series data

We convert the extracted point dataset into a clean table:

- ▶ convert times to datetime
- ▶ convert lead time to hours
- ▶ compute derived variables (e.g. wind speed)

```
# -----
# Create a clean timeseries table
#
lead_h = (pt["lead_time"].values / np.timedelta64(1, "h")).astype(int)

df = pd.DataFrame({
    "valid_time": pd.to_datetime(pt["valid_time"].values),
    "lead_h": lead_h,
    "t2m_C": pt["t2m"].values,
    "wind_ms": np.sqrt(pt["u10"].values**2 + pt["v10"].values**2),
    "mslp_hPa": pt["mslp"].values,
    "tp_mm": pt["tp"].values,
})
```



Local Map Plot (OSM basemap + 100 km temperature overlay)

Visualize a local patch around the town:

- ▶ **OpenStreetMap (OSM)** background tiles
- ▶ overlay t_{2m} for one lead time
- ▶ show only a 100 km box for clarity and speed

Key steps

- ▶ select lead time index (e.g. 24h)
- ▶ slice t_{2m} in a local lon/lat window
- ▶ reproject to WebMercator (EPSG:3857)

```
t2m_local = ds["t2m"].isel(valid_time=i).sel(  
    lat=slice(latc-dlat, latc+dlat),  
    lon=slice(lonc-dlon, lonc+dlon)  
)  
  
# plot on OSM background (contextily)  
ctx.add_basemap(ax, source=ctx.providers.OpenStreetMap.Mapnik, zoom=10)
```



Example: Berlin, t_{2m} at lead 24h
(local 100 km window).

Training a CNN: field(t) -> field(t+1)

Learning task

We train a CNN to predict next-step temperature:

- ▶ input at time t: t_{2m} , $u10$, $v10$, $mslp$
- ▶ target: $t_{2m}(t+1)$

Patch training

Instead of full fields, we train on random patches:

- ▶ sample 64x64 patches from random positions
- ▶ efficient mini-batches on laptop/GPU
- ▶ same idea as Zarr chunks: local spatial tiles

Normalization

Channel-wise mean/std over time and space:

- ▶ stabilize training
- ▶ same scaling for training and evaluation

Example:

```
# inputs and target in RAM
t2m = ds["t2m"].astype("float32").values
u10 = ds["u10"].astype("float32").values
v10 = ds["v10"].astype("float32").values
mslp = ds["mslp"].astype("float32").values

# normalization
Xin = np.stack([t2m, u10, v10, mslp], axis=1)
X_mean = Xin.mean(axis=(0,2,3), keepdims=True)
X_std = Xin.std(axis=(0,2,3), keepdims=True) + 1e-6

y_mean = t2m.mean()
y_std = t2m.std() + 1e-6

# patch sampler: X(t) -> y(t+1)
Xt, yt = sample_patch_batch(batch_size=8, patch=64)

# small CNN: 4ch -> 1ch
model = SmallCNN(in_ch=4, out_ch=1, hidden=32)
loss_fn = nn.MSELoss()
opt = optim.Adam(model.parameters(), lr=2e-3)
```

Training Loop and Outcome (loss curve)

Example:

Training loop

We train on randomly sampled spatial patches:

- ▶ each step draws a new batch: location + time
- ▶ objective: MSE between predicted and true $t2m(t+1)$
- ▶ optimizer: Adam

Expected behavior

- ▶ loss decreases quickly in the first iterations
- ▶ later it stabilizes (noise floor of the synthetic data)
- ▶ model learns local transport and correlations

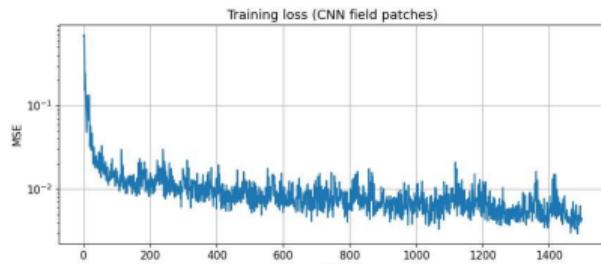
```
rng = np.random.default_rng(123)
n_steps = 800
batch_size = 8

loss_hist = []

model.train()
for step in range(1, n_steps + 1):
    Xt, yt = sample_patch_batch(batch_size=batch_size, rng=rng)
    pred = model(Xt)
    loss = loss_fn(pred, yt)

    opt.zero_grad()
    loss.backward()
    opt.step()

    loss_hist.append(float(loss.item()))
```



Evaluation: rollout forecast from initial time

One-step skill

We test the CNN on random samples:

- ▶ predict $t2m(t+1)$ from $X(t)$
- ▶ compare to truth with RMSE
- ▶ compare to persistence baseline

Rollout forecast

Starting from the initial forecast field:

- ▶ use CNN repeatedly: $\hat{x}_{t+1} = f_\theta(\hat{x}_t)$
- ▶ build a full forecast trajectory
- ▶ then extract town series for comparison

This gives a realistic "mini NWP" demonstration:

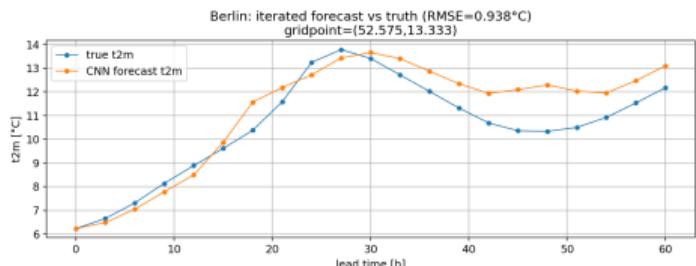
Zarr archive -> ML training -> forecast rollout .

Example (rollout):

```
# rollout: start from true initial field at t0
t2m_hat = t2m[0].copy()

pred_series = [t2m_hat]
for k in range(0, nt-1):
    Xk = build_features(t2m_hat, u10[k], v10[k], mslp[k])
    t2m_hat = predict_next_field(model, Xk)
    pred_series.append(t2m_hat)

pred_series = np.stack(pred_series, axis=0) # (time, lat, lon)
```



Outcome: Town Forecasts from ML Rollout

Town-level evaluation

After rollout, we extract time series at town locations:

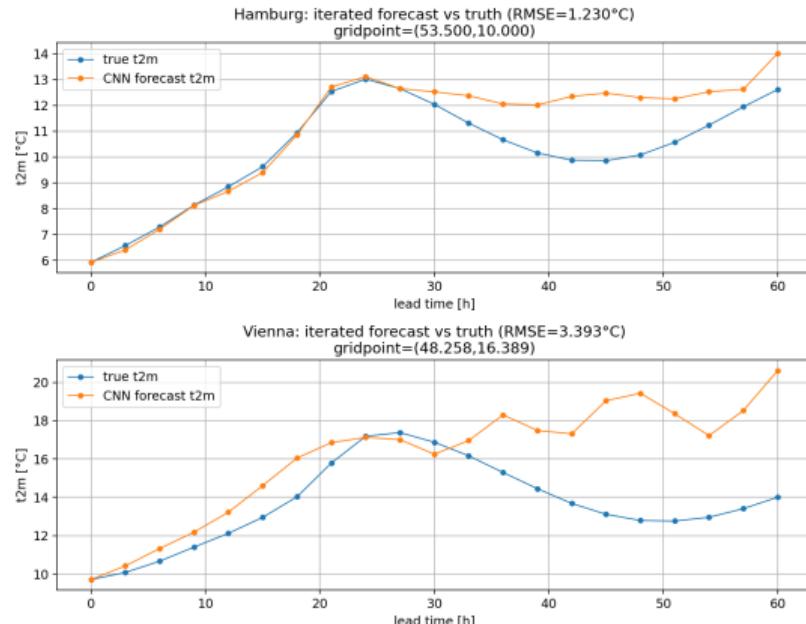
- ▶ nearest neighbor on the lat/lon grid
- ▶ compare truth vs ML forecast
- ▶ generate one plot per town

Saved outputs

- ▶ figures: `fc_<town>.png`
- ▶ fast way to compare skill across cities

```
for town, (lat0, lon0) in TOWNS.items():
    # nearest grid point
    pt_true = ds.sel(lat=lat0, lon=lon0, method="nearest")

    # predicted series at same grid point
    y_true = pt_true["t2m"].values[:n_forecast+1]
    y_pred = pred_series[:, iy, ix]  # from rollout
```



Zarr Chunk Decomposition (patch-only visualization)

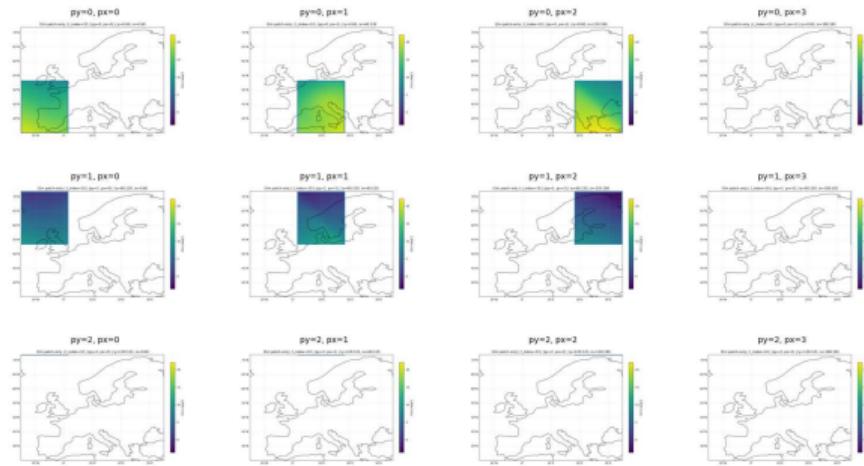
Goal

Visualize the Zarr chunk structure :

- ▶ each chunk corresponds to a spatial patch (60x60)
- ▶ plot patch values only (no full-field replot)
- ▶ use one global color scale (vmin/vmax of full field)

Procedure

- ▶ load full field once (for vmin/vmax)
- ▶ loop over patch indices (py, px)
- ▶ save one image per patch



Patch-only plots: $py=0..2$, $px=0..3$ (Europe extent, global color scale).

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!



Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Why Anemoi?

Weather forecasting is a large-scale problem

- ▶ modern forecasts rely on very large state vectors
- ▶ global and regional fields with millions of grid points
- ▶ high temporal resolution and long time series

Training data is massive

- ▶ global reanalyses such as ERA5 , ICON-DREAM
- ▶ convection-permitting simulations (e.g. Arome, ICON-Force)
- ▶ multi-variable, multi-level, multi-year datasets

These datasets quickly reach terabyte scale.

What this implies for ML

- ▶ training must be parallel and distributed
- ▶ data access must be chunked and efficient
- ▶ geometry and grid structure must be respected

Role of Anemoi

- ▶ parallelizes fields and samples
- ▶ integrates data, graphs, models, and training
- ▶ supports the full ML lifecycle for weather models

Anemoi is not a model — it is a framework.

Design Philosophy of Anemoi

Core principles

- ▶ configuration over hard-coded logic
- ▶ clear separation of responsibilities
- ▶ scalable by construction

Anemoi is designed such that:

- ▶ data handling,
- ▶ graph construction,
- ▶ model definition,
- ▶ training orchestration

are **independent but composable** components.

Declarative workflows

- ▶ experiments are defined in **YAML**
- ▶ components are selected, not programmed
- ▶ changes are traceable and reproducible

Operational mindset

- ▶ same setup works on laptop, HPC, and cloud
- ▶ parallelism is explicit, not accidental
- ▶ full provenance of data and models

Anemoi treats ML experiments as engineering systems.

YAML: The Configuration Backbone

Why configuration matters at scale

- ▶ experiments involve many interacting components
- ▶ parameters change more often than code
- ▶ reproducibility depends on explicit configuration

In Anemoi, YAML files define :

- ▶ datasets and preprocessing
- ▶ model architectures and hyperparameters
- ▶ training strategies and resources

Why YAML?

- ▶ human-readable and versionable
- ▶ hierarchical and structured
- ▶ easy to override and compose

Design choice

- ▶ no Python code for experiment logic
- ▶ no hidden defaults in scripts
- ▶ configuration becomes a **first-class artifact**

In Anemoi, changing the experiment means changing YAML.

YAML in Practice: Declaring an Experiment

A single YAML file describes an experiment

- ▶ what data is used
- ▶ which model is trained
- ▶ how training is executed

No experiment logic is hidden in Python code.

The configuration is the experiment.

This YAML file fully specifies one training run.

Example: training configuration (simplified)

```
1 dataset:  
2   name: era5  
3   variables: [t2m, u10, v10]  
4   resolution: 0.25  
5  
6 model:  
7   name: graph_transformer  
8   hidden_dim: 256  
9   num_layers: 6  
10  
11 training:  
12   batch_size: 4  
13   max_epochs: 50  
14   accelerator: gpu
```

OmegaConf: From YAML to Runtime Objects

The problem

- ▶ YAML files are static text
- ▶ training code needs structured objects
- ▶ configuration must be inspectable at runtime

Properties of DictConfig

- ▶ hierarchical (mirrors YAML structure)
- ▶ accessed via attributes or keys
- ▶ supports type checking

OmegaConf solves this

- ▶ represents configuration as Python objects
- ▶ preserves hierarchy and structure
- ▶ supports interpolation and defaults

Why this matters

- ▶ training code stays generic
- ▶ behaviour is fully configuration-driven
- ▶ experiments become reproducible by construction

The result is a `DictConfig` object.

Configuration becomes part of the program state.

OmegaConf in Practice and in Anemoi

Using OmegaConf in Python

- ▶ configuration is passed into the training code
- ▶ parameters are accessed programmatically
- ▶ no hard-coded values are needed

Accessing the configuration

```
1 # cfg provided by Hydra
2 print(cfg.model.layer_sizes)
3 print(cfg.training.max_epochs)
```

Resolving and converting

```
1 from omegaconf import
     OmegaConf
2
3 OmegaConf.resolve(cfg)
4 cfg_dict = OmegaConf.
     to_object(cfg)
```

Role in Anemoi

- ▶ schemas validate configs before training
- ▶ resolved configs are logged with results
- ▶ the full configuration defines the experiment

OmegaConf links configuration and execution.

Hydra: Managing Variants of the Same Experiment

Motivation

- ▶ same training code
- ▶ same task (e.g. learning $\sin(x)$)
- ▶ different model architectures

Hydra allows:

- ▶ selecting model variants via YAML
- ▶ switching architectures without code changes
- ▶ keeping experiments comparable

Base configuration

- ▶ defines which components are active
- ▶ serves as the experiment entry point

config.yaml

```
1 defaults:  
2   - model: mlp  
3   - training: simple  
4   - _self_
```

model/mlp.yaml

```
1 layer_sizes: [1, 64, 1]  
2 activation: relu
```

model/deep.yaml

```
1 layer_sizes: [1, 128, 64, 32,  
               1]  
2 activation: tanh
```

The experiment is composed, not rewritten.

Hydra in Practice: Same Code, Different Outcomes

What changes between these runs

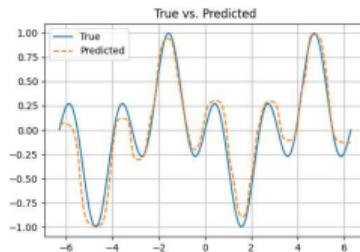
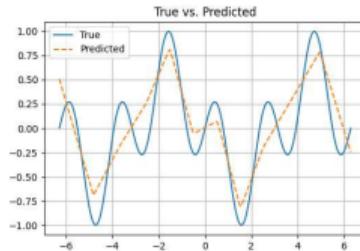
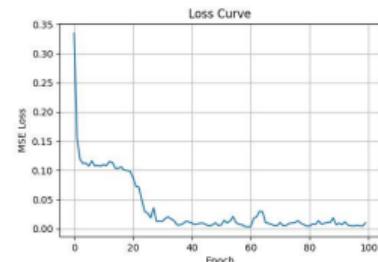
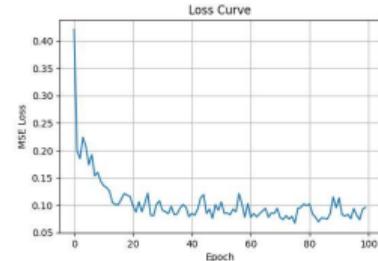
- ▶ only the **Hydra configuration**
- ▶ different model definitions (`mlp` vs `deep`)
- ▶ same training loop, same data

Hydra controls:

- ▶ which YAML files are composed
- ▶ which model architecture is instantiated
- ▶ which hyperparameters are active

Key observation

- ▶ shallow model converges smoothly
- ▶ deeper model shows slower, less stable training



Hydra makes architectural choices explicit, comparable, and reproducible.

Why Graphs in Weather Machine Learning?

Weather data is not Cartesian

- ▶ global models use spherical geometry
- ▶ grids are often **non-uniform and unstructured**
- ▶ classical CNN assumptions break down

Examples:

- ▶ ICON triangular grid
- ▶ reduced Gaussian grids
- ▶ observation networks

Why graphs are a natural abstraction

- ▶ nodes represent spatial locations
- ▶ edges represent physical neighbourhoods
- ▶ locality is explicit, not implicit

Key idea

- ▶ move from **array indices** to **connectivity**
- ▶ geometry becomes part of the model

Graphs decouple geometry from resolution.

Icosahedral and Geodesic Graphs

ICON as a guiding principle

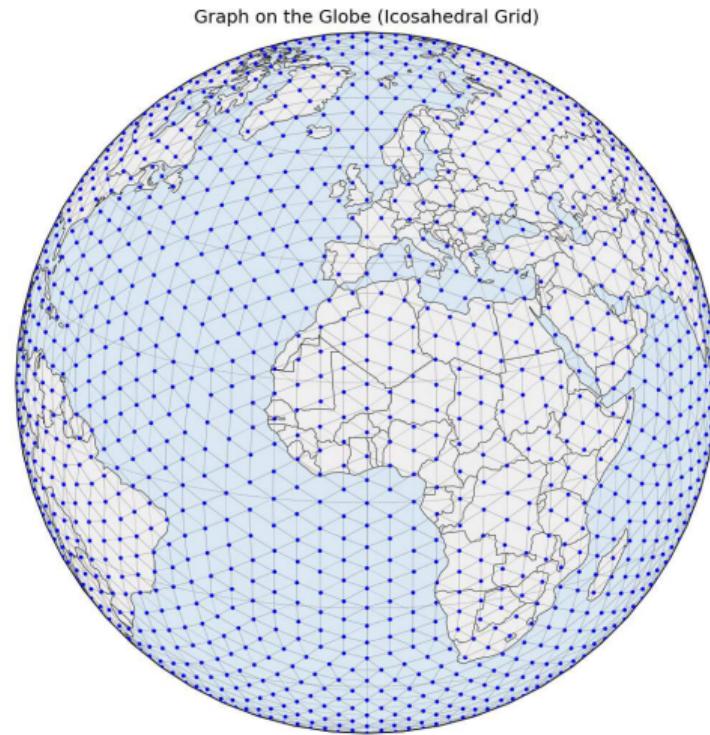
- ▶ refined icosahedral grids
- ▶ nearly uniform cell areas
- ▶ no singularities at the poles

Anemoi adopts the same idea:

- ▶ nodes placed on a refined icosahedron
- ▶ spherical geometry is explicit
- ▶ resolution controlled by refinement level

Key trade-off

- ▶ higher resolution \Rightarrow more nodes
- ▶ increased cost, but better spatial fidelity



Icosahedral graph on the sphere (orthographic projection). Edges follow great-circle distances.

Edges, Neighbours, and Graph Topology

Defining neighbourhoods

- ▶ edges encode spatial interaction
- ▶ most common choice:
k-nearest neighbours (kNN)
- ▶ distance measured on the sphere

Each node exchanges information only with its neighbours:

- ▶ local interactions
- ▶ scalable message passing

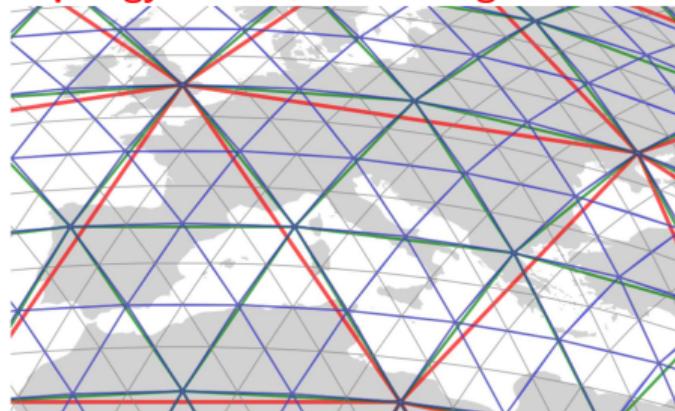
Edge index representation

- ▶ graph stored as `edge_index`
- ▶ integer pairs (i, j) define connections
- ▶ independent of data values

Reuse across experiments

- ▶ graph built once
- ▶ reused for training, validation, inference
- ▶ ensures geometric consistency

Topology is fixed; data changes.



Anemoi: A Modular ML Framework for Weather

Design philosophy

- ▶ separate concerns cleanly
- ▶ make each component replaceable
- ▶ scale from experiments to operations

Anemoi is not a single package, but a coordinated ecosystem .

Each package addresses one layer:

- ▶ data handling
- ▶ graph construction
- ▶ model definition
- ▶ training orchestration

Core Anemoi packages

- ▶ `anemoi-datasets`
- ▶ `anemoi-graphs` in `anemoi-core` repo
- ▶ `anemoi-models` in `anemoi-core` repo
- ▶ `anemoi-training` in `anemoi-core` repo

All packages:

- ▶ use YAML + Hydra + OmegaConf
- ▶ integrate via clearly defined interfaces
- ▶ are developed in a shared monorepo

Graphs connect geometry; packages connect workflows.

anemoi-datasets: Data and Metadata

Purpose

- ▶ prepare large meteorological datasets for ML
- ▶ unify metadata, statistics, and structure
- ▶ decouple raw data formats from training

Key features

- ▶ ingestion from GRIB / NetCDF
- ▶ conversion to Zarr
- ▶ automatic statistics (mean, std, min, max)
- ▶ validation and consistency checks

<https://anemoi.readthedocs.io/projects/datasets/en/latest/>

Why this matters

- ▶ scalable I/O for HPC and cloud
- ▶ reproducible data pipelines
- ▶ identical datasets across experiments

Data becomes a first-class, validated object.

	shortName	Name	Levels
0	P	Pressure	[101, 108, 112, 119, 120, 49, 57, 64, 70, 75, 79, 86, 91, 96]
1	QV	Specific Humidity	[101, 108, 112, 119, 120, 49, 57, 64, 70, 75, 79, 86, 91, 96]
2	T	Temperature	[101, 108, 112, 119, 120, 49, 57, 64, 70, 75, 79, 86, 91, 96]
3	U	U-Component of Wind	[101, 108, 112, 119, 120, 49, 57, 64, 70, 75, 79, 86, 91, 96]
4	V	V-Component of Wind	[101, 108, 112, 119, 120, 49, 57, 64, 70, 75, 79, 86, 91, 96]
5	W	Vertical Velocity (Geometric) (w)	[101, 108, 112, 119, 120, 49, 57, 64, 70, 75, 79, 86, 91, 96]

anemoi-graphs: Geometry and Connectivity

Purpose

- ▶ represent spatial geometry explicitly
- ▶ construct reusable graph topologies
- ▶ support non-Cartesian grids

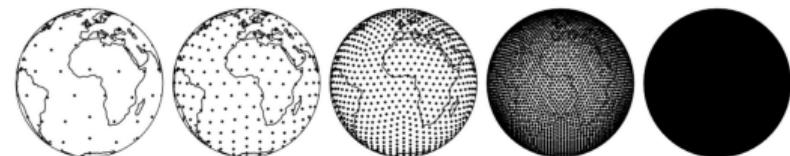
Design principle

- ▶ graph built once
- ▶ reused for training and inference
- ▶ geometry independent of data values

Key components

- ▶ TriNodes from refined icosahedra
- ▶ neighbour definitions via kNN
- ▶ edge construction on the sphere

Topology is fixed; learning happens on it.



Docs: <https://anemoi.readthedocs.io/projects/graphs/en/latest/>

anemoi-models: Neural Architectures

Purpose

- ▶ define ML models for weather prediction
 - ▶ separate architecture from training logic
 - ▶ support graph-based learning

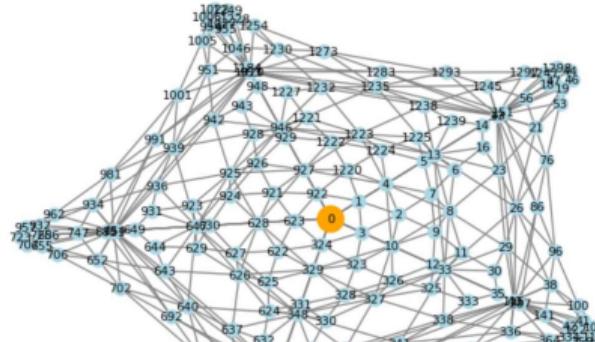
Model families

- ▶ graph neural networks
 - ▶ transformer-based architectures
 - ▶ hybrid models for spatio-temporal data

Interfaces

- ▶ standardized input/output conventions
 - ▶ integration with PyTorch Lightning
 - ▶ instantiated via Hydra configs

Models are components, not scripts.



anemoi-training: Orchestrating Experiments

Purpose

- ▶ orchestrate the full training lifecycle
- ▶ integrate data, graphs, and models
- ▶ scale to HPC environments

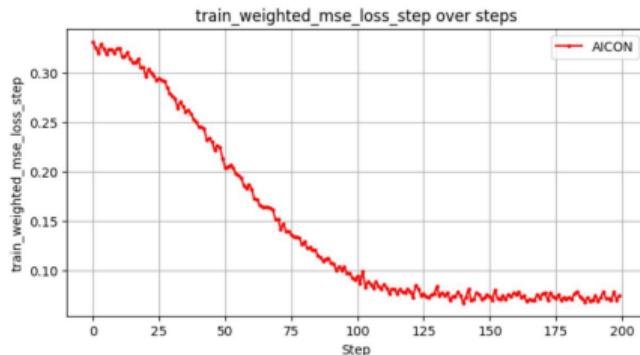
Key elements

- ▶ AnemoiTrainer as central controller
- ▶ PyTorch Lightning backend
- ▶ distributed and multi-node training

Experiment control

- ▶ driven entirely by configuration
- ▶ automatic logging and checkpointing
- ▶ reproducible, restartable runs

Training becomes declarative and scalable.



Docs: <https://anemoi.readthedocs.io/projects/training/en/latest/>

Creating Zarr Datasets with anemoi-datasets

- ▶ convert raw meteorological data into ML-ready format
- ▶ standardize structure, metadata, and statistics
- ▶ enable scalable training and validation

Key idea

- ▶ dataset creation is fully configuration-driven
- ▶ no preprocessing logic is hard-coded
- ▶ every dataset is reproducible

Typical processing steps:

- ▶ read GRIB or NetCDF input
- ▶ reshape and flatten grids if required
- ▶ compute global statistics
- ▶ write chunked Zarr archive

<https://anemoi.readthedocs.io/projects/datasets/en/latest/>

Command-line workflow

```
1 anemoi-datasets create \
2   dataset.yaml \
3   era5.zarr
```

Result

- ▶ validated Zarr dataset
- ▶ embedded metadata and statistics
- ▶ directly usable by Anemoi training

Data preparation becomes a reproducible pipeline.

Validating and Inspecting Zarr Datasets

Why validation matters

- ▶ training assumes consistent metadata
- ▶ missing values or shapes cause silent errors
- ▶ errors should be caught **before** training

Anemoi provides built-in tools to:

- ▶ verify temporal coverage
- ▶ check spatial resolution and shapes
- ▶ ensure statistics are present

Validation is part of the dataset lifecycle, not an afterthought.

Docs: [anemoi-datasets](#)

Inspecting a dataset

```
1 anemoi-datasets inspect  
      era5.zarr
```

Typical output includes:

- ▶ time range and frequency
- ▶ variables and dimensions
- ▶ min / max / mean / std
- ▶ total size and chunking

Only validated datasets enter training.

ERA Data Stored in Zarr

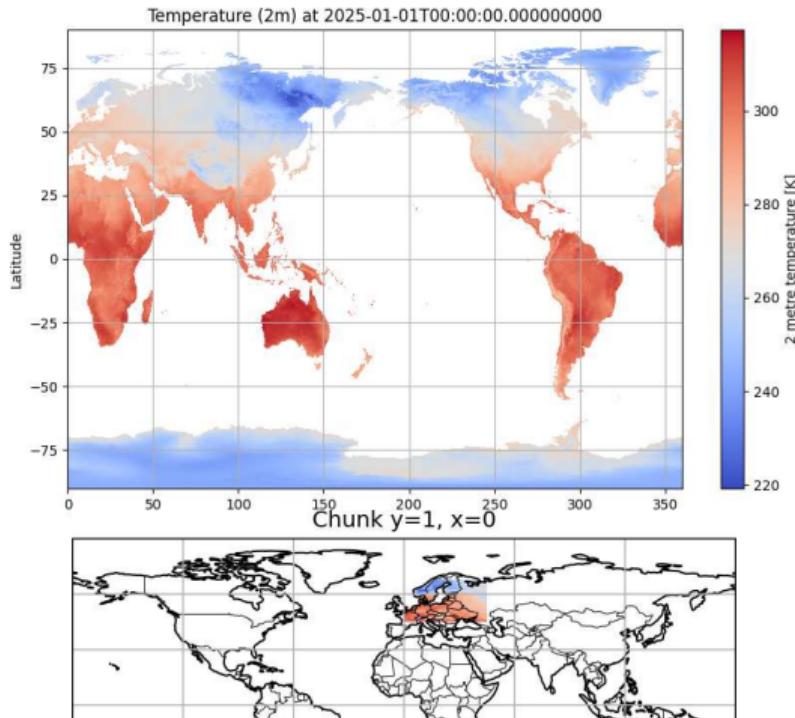
What these images show

- ▶ ERA reanalysis data after Zarr conversion
- ▶ accessed chunk-wise, not as monolithic files
- ▶ identical physical content, different storage logic

Zarr enables:

- ▶ parallel access to spatial subdomains
- ▶ efficient mini-batching for ML
- ▶ scalable training on HPC systems

**Zarr is an ML-optimized view
of reanalysis data.**



Top: Global ERA5 2 m temperature field.
Bottom: Spatial chunk extracted from Zarr archive.

Application Example: Learning CBCF Warning Polygons from Forecast Fields

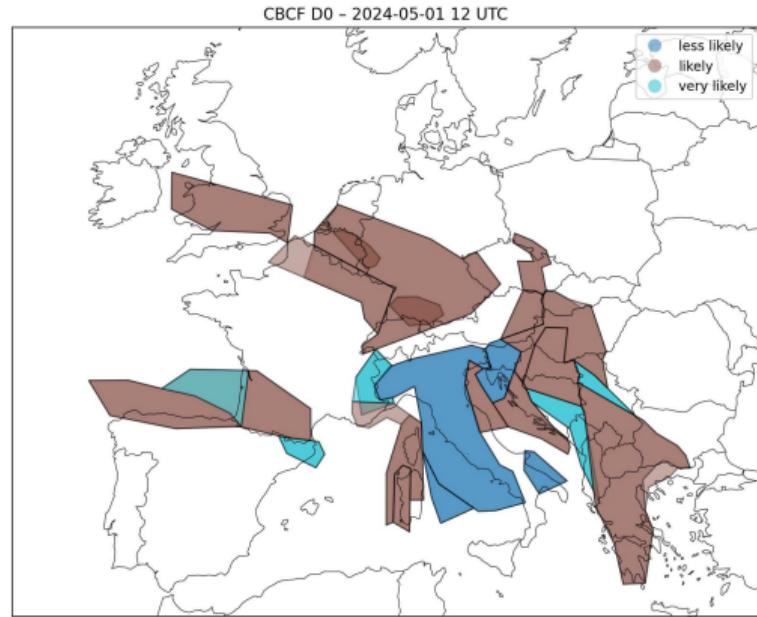
Learn an AI model that predicts operational warning regions (CBCF polygons) from model forecast fields.

Training data

- ▶ multi-year archive of forecast fields (e.g. wind/gust)
- ▶ forecaster-issued warning polygons

Core idea

- ▶ polygons are a human product representation
- ▶ neural nets learn on grids / tensors
- ▶ therefore we convert polygons → mask labels



Example CBCF polygons (likelihood categories).

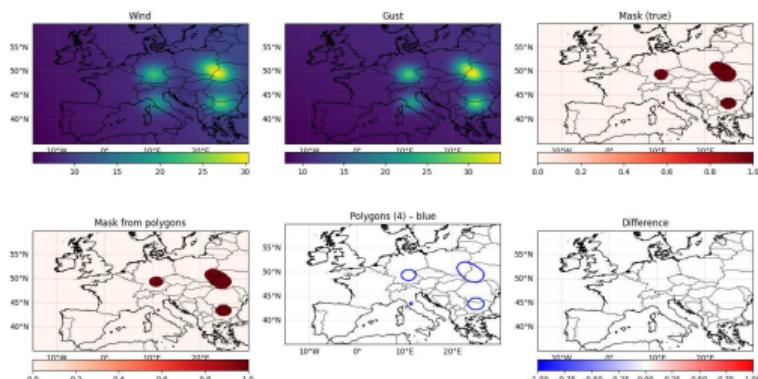
We train on masks — and recover polygons afterwards.

From Polygons to Masks: Training Labels on the Model Grid

Why convert polygons to masks?

- ▶ operational warnings are defined as **polygons**
- ▶ neural nets require **grid-aligned targets**
- ▶ rasterization yields a binary/soft **mask**
 $m(x) \in [0, 1]$

- ▶ predictors: **wind** + **gust** fields
- ▶ target: warning mask derived from polygons



Two-way conversion

- ▶ **Polygon → mask:** rasterize onto model grid
- ▶ **Mask → polygon:** contour / region extraction

Quality control

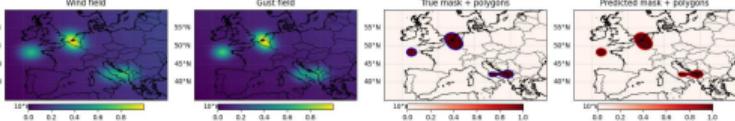
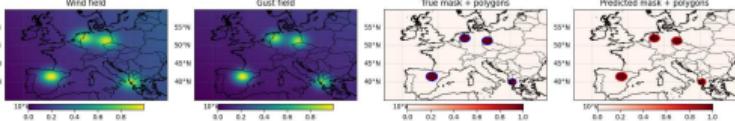
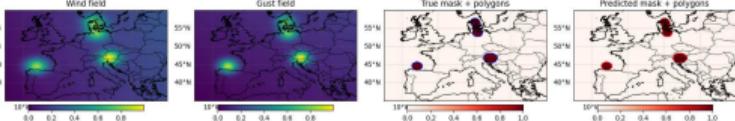
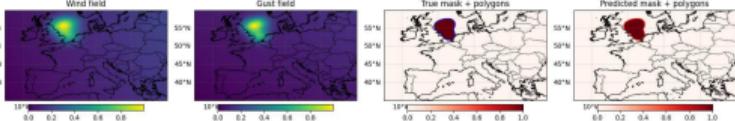
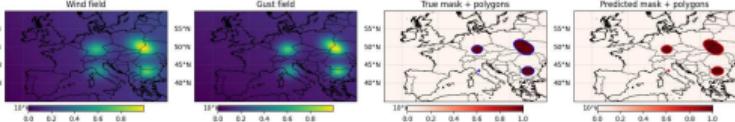
- ▶ check **mask(polygons)** against reference labels
- ▶ difference highlights discretization artefacts

Polygon-to-mask conversion and consistency check.

Learning Demo: Inputs (Wind/Gust) → True vs Predicted Mask & Polygons

This figure shows one training/evaluation example

- ▶ predictors are gridded forecast fields
- ▶ target is a warning mask derived from forecaster polygons
- ▶ predicted mask is transformed back into polygons



4-column result: inputs, target, prediction.

Real Demo: CBCF Training with Anemoi (Repository Setup)

We ran the CBCF demo using an Anemoi-based training stack.

Local repo structure

- ▶ anemoi-datasets (data handling, zarr)
- ▶ anemoi-transform (preprocessing)
- ▶ anemoi-core (graphs, models, training)
- ▶ training_config/ (yaml configs)
- ▶ batch_train.sh (run script)

Key point

- ▶ this is not a standalone notebook hack
- ▶ it is a reproducible training workflow (configs + checkpoints)

Dependencies (uv workspace)

- ▶ anemoi-training,
- anemoi-models, anemoi-graphs,
- ...
- ▶ aicon-catalog[create]

Commands (as executed)

```
cd cbcf
git pull
ls training_config
# run: batch_train.sh
```

Outcome

- ▶ trained checkpoints
- ▶ inference + plots

Configuration-Driven Workflow: Training & Diagnosis YAML

- ▶ training is **fully reproducible** and versionable
- ▶ separation of concerns:
 - ▶ dataset / resolution (R3B5, R3B8, ...)
 - ▶ variables (inputs/targets)
 - ▶ model architecture choice
 - ▶ training schedule / precision / hardware

Conceptual config blocks

```
dataset:  
    resolution: R03B05  
model:  
    type: graph/diagnoser  
training:  
    precision: 16-mixed  
outputs:  
    checkpoint: inference-last.ckpt
```

(Shown here only conceptually — actual files live in `training_config/` and include many more details.)

Example config files

- ▶ `training_config/cbcf_graph_R3B5.yaml`
- ▶ `training_config/cbcf_diagnose_R3B5.yaml`

Demo message

We can train CBCF models for
different ICON grids and resolutions without changing
code — only by switching configs.

Inference Pipeline: Load Checkpoint → Predict → Plot

What happens after training?

- ▶ load **checkpoint** (`inference-last.ckpt`)
- ▶ load forecast data from **zarr archive**
- ▶ build input tensor (multi-step input supported)
- ▶ run `model.predict_step(...)`

Outputs

- ▶ predicted CBCF fields / masks
- ▶ comparison against targets
- ▶ saved plots per date + variable

Code sketch (from `main.py`)

```
model = torch.load(ckpt)
model.eval()

ds = zarr.open("...CBCF.zarr")

input = build_tensor(...)
y_pred = model.predict_step(input)

plot(y_pred); plot(target)
```

The demo uses multi-step inputs and produces predicted/target plots.

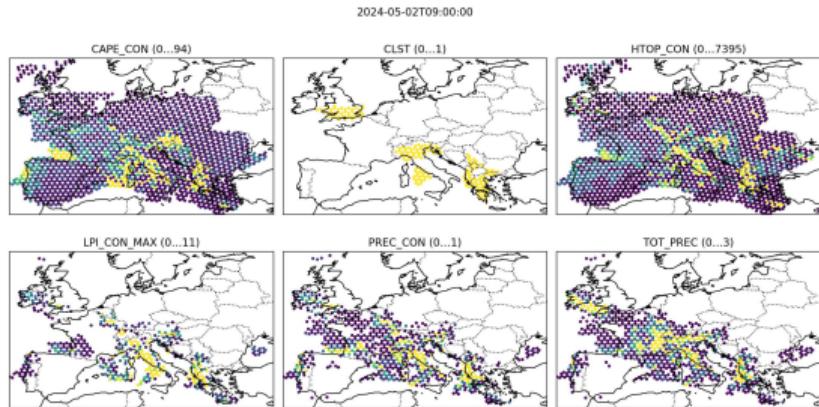
Take-away

- ▶ same pipeline works for long archives
- ▶ basis for **operational verification and monitoring**

CBCF Results: Anemoi Training Demo (Examples)

What we show

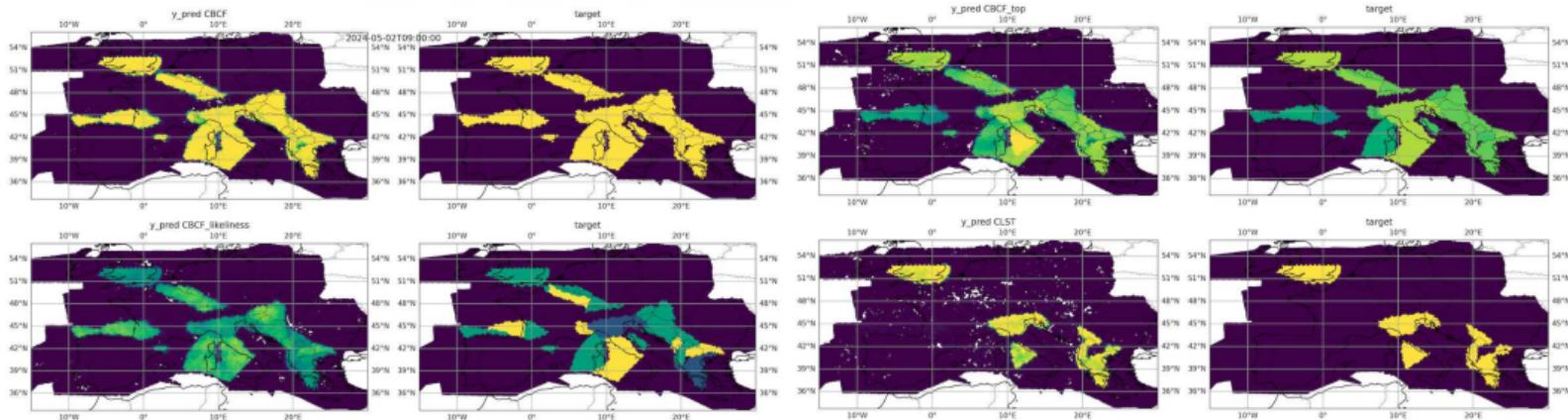
- ▶ results from the Anemoi-based training workflow
- ▶ prediction of CBCF target fields / masks from forecast inputs
- ▶ comparison against the corresponding targets



Interpretation

- ▶ spatial patterns are learned coherently
- ▶ overall structure matches operational products
- ▶ remaining differences motivate the next steps:
 - ▶ improved architectures
 - ▶ longer training / larger datasets
 - ▶ calibration and polygon-space verification

CBCF Results: Predicted vs Target Fields (Selected Outputs)



Fields shown (left = prediction, right = target):

- ▶ CBCF: main warning mask/product
- ▶ CBCF_likelihood: continuous likelihood / confidence field

Fields shown (left = prediction, right = target):

- ▶ CBCF_top: top-level / dominant CBCF intensity field
- ▶ CLST: additional label channel (sparser warning type)

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00	Joint Dinner				

The AI Transformation: Communication History in 8 Steps

From printed knowledge to mass media

- ▶ **Book printing**
(Gutenberg press, scalable knowledge)
- ▶ **Electricity / telegraph**
(Morse telegraph, instant signals)
- ▶ **Radio**
(wireless transmission, mass broadcasting)
- ▶ **Television**
(moving images, global events)

From computation to language interfaces

- ▶ **Computers**
(Konrad Zuse Z3, programmable digital computer)
- ▶ **Internet**
(TCP/IP, global connectivity)
- ▶ **WWW**
(Internet for all!)
- ▶ **Social media**
(Facebook era, interactive networks)
- ▶ **LLMs & AI systems**
(language becomes an interface , interaction at scale)

Transformation:

Communication turns into interaction.

Why Now? The Next Step Has Arrived

A new communication revolution

- ▶ Book printing scaled **knowledge**
- ▶ Internet scaled **connectivity**
- ▶ **LLMs scale interaction:**
language becomes an interface



What changes:

- ▶ from **information access**
- ▶ to **interactive assistance**

Progress in Communication.

What does it mean?

Where are we as humans?

This affects communication, documentation, coding, workflows

— across the entire organisation.

Why Now? Language Becomes an Interface

What changed in 2022?

- ▶ For the first time, computers can handle natural language **appropriately**
- ▶ This enables **interaction** instead of programming
- ▶ Not only answers — but **actions**:
(writing, summarizing, translating, coding, planning)

Examples in daily work

- ▶ Draft emails, minutes, reports
- ▶ Summarize long documents
- ▶ Create slides from bullet points
- ▶ Helpdesk: FAQ, ticket triage
- ▶ Code assistance: refactor + docs

New capability: conversational work

- ▶ Ask → draft → revise → finalize
- ▶ Explain → implement → test → debug

Key message:

AI affects everyone — not only scientists or IT staff.

Slow Down: 1993: The Internet Becomes Public

1993: a global information layer

- ▶ Web browsers make the Internet usable for everyone
- ▶ Information access becomes **self-service**
- ▶ Knowledge shifts from **local** to **global**

Key change:

Search and access replace distribution.

Many workflows become web-based:
documentation, support, collaboration,
publishing.

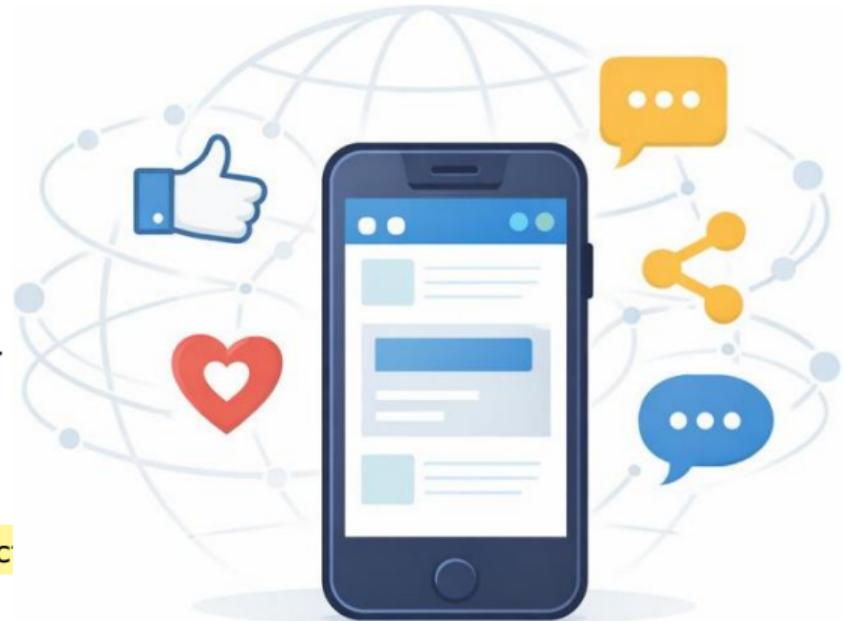


Internet era: global connectivity and instant access

2004 / 2008: Social Media + Smartphones

From websites to interactive networks

- ▶ 2004: social networks go mainstream
- ▶ 2007/2008: smartphones put the Internet in every pocket
- ▶ Everyone becomes: **consumer + producer + distributor**



Key change:

Communication becomes continuous and interactive

Attention becomes a resource; content flows in real time.

The always-on era: interaction at global scale

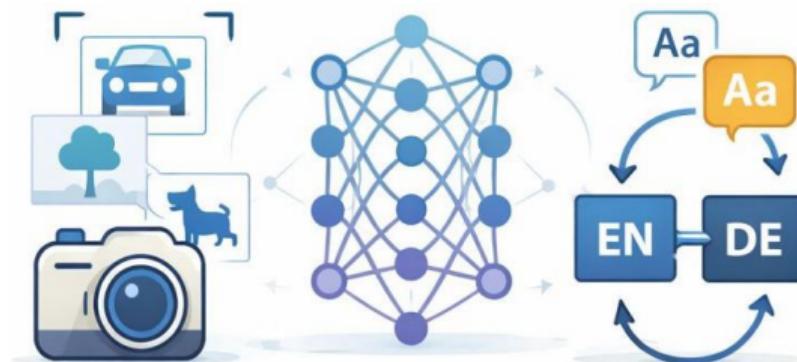
2012–2019: The ML Revolution in Vision and Translation

Deep learning becomes practical

- ▶ **Image recognition** jumps to new accuracy levels
- ▶ **Machine translation** improves dramatically
- ▶ Pattern learning outperforms hand-crafted rules

Key change:

Perception tasks become learnable at scale.



Modern deep learning architectures enable scaling

This is the foundation for today's multi-modal AI (text, images, audio).

2022: The Chatbot Threshold

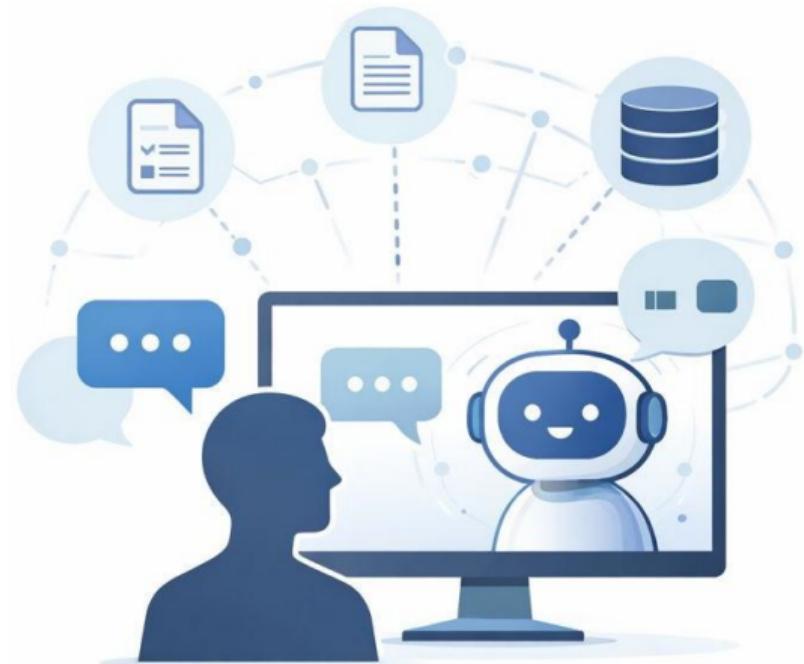
The breakthrough: language interaction

- ▶ Chatbots become useful for real work
- ▶ People can **ask, refine, and iterate** in dialogue
- ▶ The interface is now **natural language**

Key change:

From communication to interaction.

LLMs + tools + knowledge access ⇒ AI assistants and agents.



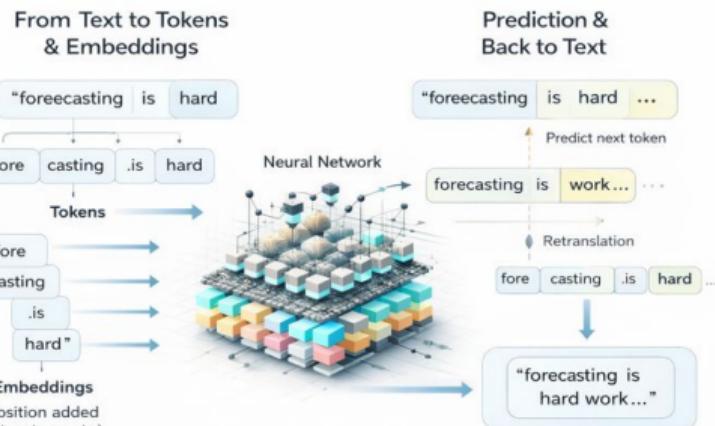
Chatbots evolve into systems: retrieval + tools + guardrails

LLMs in One Sentence

A large language model is a prediction machine

- ▶ It reads text and predicts **the next token**
- ▶ It repeats this step many times to produce an answer
- ▶ Training: learn statistical structure from huge corpora

Key point: It does not store facts like an encyclopedia. It learns patterns of language and reasoning .



Transformer models: sequence in → sequence out

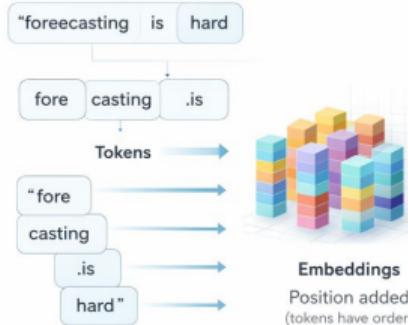
Step 1: From Text to Tokens

Computers do not read words

- ▶ Text is split into **tokens**
- ▶ Tokens are pieces of words, punctuation, spaces
- ▶ Example:

```
"forecasting is hard"  
⇒ ["fore", "casting", " is", " hard"]  
⇒ [1523, 9182, 318, 6732] (token IDs)
```

Why tokens matter: they define what the model can represent efficiently.



Tokens → vectors

- ▶ each token becomes a vector (**embedding**)
- ▶ vectors capture similarity:
("rain" closer to "cloud" than to "banana")
- ▶ position is added:
(word order matters)

This is how text enters a neural network: as numbers in a high-dimensional space.

Step 2: Attention — How the Model Understands Context

Attention is selective reading

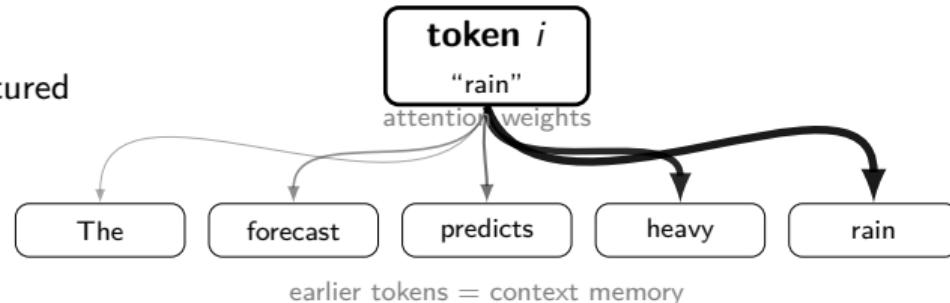
- ▶ for each word, the model decides: **what earlier words matter**
- ▶ it forms weighted links between tokens
- ▶ repeated many times in layers

A simple intuition

- ▶ Like humans: **scan + focus + connect**
- ▶ Not one focus, but many:
(multi-head attention)

Transformer = many attention layers + feed-forward layers.

Key point: Attention allows long-range dependencies and structured reasoning.



Step 3: Training — Next Token Prediction

Training objective

- ▶ show text to the model
- ▶ hide the next token
- ▶ train it to predict the hidden token

Why it can fail

- ▶ it predicts **plausible text**
- ▶ not guaranteed truth
- ▶ can **hallucinate** details

Result: the model learns grammar, style, facts, and reasoning patterns **as an emergent capability**.

Therefore: verification + grounding are crucial in professional use.

Important: training is expensive (compute + data).

Using a model is cheap (inference).



Inference: Prompting Is Steering

A prompt creates the context

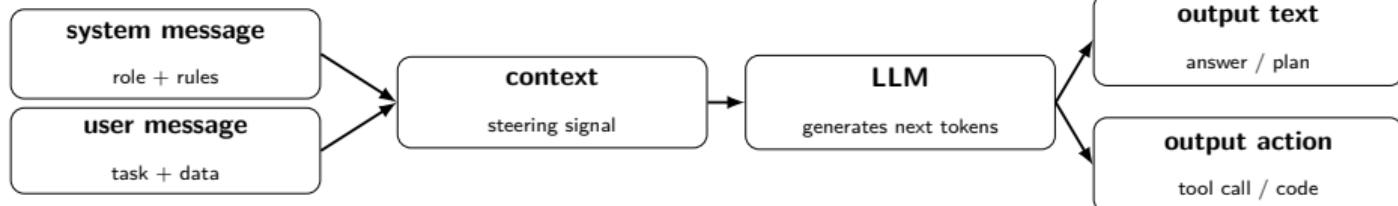
- ▶ system message: role + rules
- ▶ user message: task + data
- ▶ the model continues from there

What improves results

- ▶ clear goal + constraints
- ▶ examples of desired style
- ▶ provide trusted information
- ▶ ask for uncertainties / checks

Key point: The model is **not stable by itself**.
It is shaped by the context we provide.

Good prompting is a communication skill: precise intent, good context, clear quality criteria.



Prompting = setting context that steers the model's behavior and output.

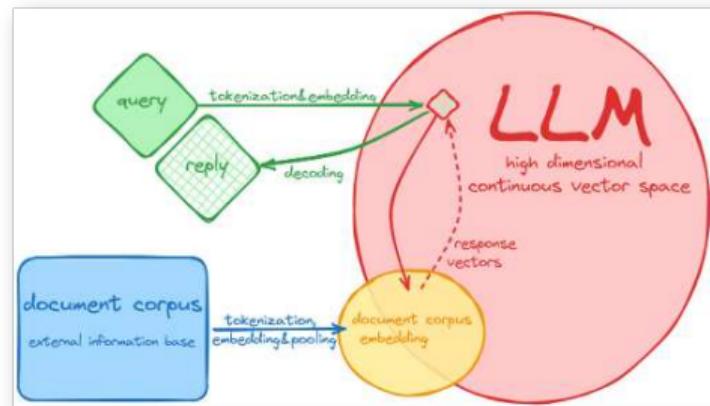
From Language to Interaction: LLMs + Tools

LLMs become useful at work when connected

- ▶ retrieval (RAG): trusted documents
 - ▶ policies, manuals, guidance, internal knowledge bases
 - ▶ reduces hallucinations by grounding on sources
- ▶ tools: APIs, code execution, search
 - ▶ from “talking” to **doing** (compute, query, automate)
 - ▶ reproducible outputs: scripts, plots, tables

Key message: Real systems are **LLM-based assistants**, not just chat.

- ▶ **memory:** project and user context
 - ▶ remembers assumptions, preferences, ongoing tasks
 - ▶ avoids repeating the same onboarding every time



From Chat to Action: Why Function Calling Matters

Example: tool call proposed by the LLM

Chat is not the goal.

LLMs become truly useful when they can trigger real actions :

- ▶ download and inspect forecast data
- ▶ compute diagnostics and key numbers
- ▶ generate plots and reports
- ▶ call services (archives, NWP, HPC tools)

Function calling turns language into structured decisions — executed by a controlled backend (not by the LLM).

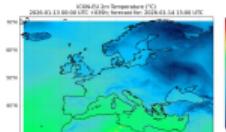
```
{  
  "tool_name": "get_weather_forecast",  
  "arguments": {  
    "model": "ICON-EU",  
    "variable": "t2m",  
    "location": "Berlin",  
    "lead_time_h": 24  
  }  
}
```

what is the weather tomorrow at 3pm in Reading, UK?

DAWID:

* Model:
 + Function-get_weather_forecast
 + Forecast

Here is the ICON-EU forecast plot for Reading (UK) for tomorrow at 15:00 (local time):



DAWID executes the tool call and returns the result.

Tool Awareness: How the Model Knows What It Can Do

The key idea

The LLM cannot “invent” actions. It can only call tools that we **explicitly provide**.

Why this matters

- ▶ clear capabilities: **what is possible**
- ▶ safety boundary: **what is not allowed**
- ▶ reproducible results: same tool ⇒ same behavior

Think of it as a menu: the model chooses an item, the system cooks it.

The tool list becomes the model's action space.

```
Tool registry (provided to the model)
TOOLS = [
  {
    "name": "get_weather_forecast",
    "description": "Retrieve ICON-EU forecast values",
    "inputs": {
      "location": "string",
      "variable": "t2m|wind|precip",
      "lead_time_h": "integer"
    }
  },
  {
    "name": "google_search",
    "description": "Search the web for up-to-date information",
    "inputs": { "query": "string" }
  }
]
```



Tools can be Functions or Agents

Next step beyond tool calling

With function calling, a “tool” does not need to be a single function. It can also be an **agent** — a specialist that performs multi-step actions.

Example specialist agents

- ▶ **Search agent:** finds reliable sources
- ▶ **Data agent:** downloads + preprocesses datasets
- ▶ **Code agent:** runs scripts and produces figures
- ▶ **Report agent:** writes a structured summary

Key idea: the LLM becomes a **coordinator** that delegates tasks and combines results into one coherent answer.

Example: delegate to a specialist agent

```
{  
    "tool_name": "search_agent",  
    "arguments": {  
        "task": "Find the latest DWD warnings for Be-",  
        "sources": "official",  
        "return_format": "short_summary + links"  
    }  
}
```

Delivers as return:

```
{  
    "agent_result": {  
        "summary": "...",  
        "sources": [..., ...]  
    }  
}
```

LLM Orchestration: Decisions & Guardrails

LLM = Decision Engine

For each user request, the model decides what happens next :

- ▶ answer directly (when trivial)
- ▶ call a function tool (fast action)
- ▶ delegate to an agent (multi-step work)
- ▶ ask a clarifying question (missing info)

Key idea: The model routes the task to the best capability.

(Like a dispatcher: choose the right action at the right time.)

Guardrails = System Control

The system stays in charge of execution :

- ▶ only approved tools exist (tool registry)
- ▶ arguments are validated (schemas)
- ▶ execution runs outside the LLM
- ▶ logging & reproducibility by design

Result: Reliable actions, not hallucinated actions.

The LLM proposes — the backend disposes.

The Race for Function & Agent Standards

OpenAI — Tool Calling + JSON Schema

- ▶ tools defined via **JSON Schema**
- ▶ **Structured Outputs / strict schema** for reliable calls

Anthropic — MCP (Model Context Protocol)

- ▶ open connector:
model ↔ MCP server ↔ tools/data
- ▶ goal: universal plug-in interface for enterprise tools

Google — Gemini Function Calling

- ▶ built-in function calling in the Gemini API ecosystem
- ▶ tight integration with the Google stack

Linux Foundation / AAIF — Interoperability

- ▶ push for neutral/open **agent tool interfaces**
- ▶ avoid vendor lock-in and “tool Babel”



OpenAI — Tool Calling + JSON Schema

- tools defined via **JSON Schema**
- **Structured Outputs / strict schema** for reliable calls



Google — Gemini Function Calling

- built-in function calling in the Gemini API ecosystem
- tight integration with the **Google stack**



Anthropic — MCP (Model Context Protocol)

- open connector: **model ↔ MCP server ↔ tools/data**
- goal: universal plug-in interface for enterprise tools



- #### Linux Foundation / AAIF — Interoperability
- push for neutral/open **agent tool interfaces**
 - avoid vendor lock-in and “tool Babel”

Our Opportunity: DAWID Puts Trusted Functions at Our Fingertips

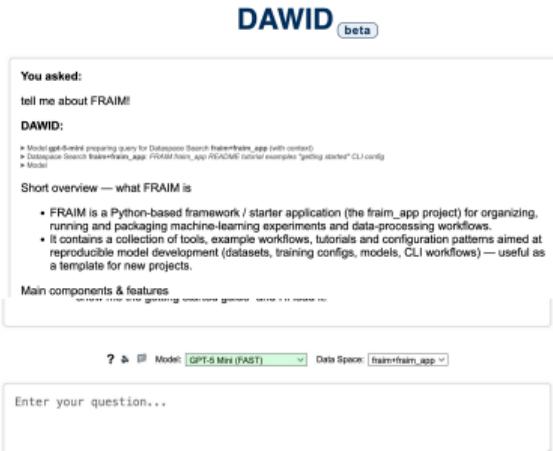
What we can do immediately

DAWID gives us a practical advantage: we can expose our own operational capabilities as tools.

- ▶ tools = trusted ECMWF / DWD / NHMS functions (not generic chat)
- ▶ each tool encodes a workflow step we already know
- ▶ the LLM becomes the orchestrator and UI

Result:

- ▶ faster exploration and analysis
- ▶ reproducible results (same tools, same outputs)
- ▶ knowledge transfer: tools carry expertise



The screenshot shows the DAWID beta user interface. At the top, it says "DAWID beta". Below that, a text input field contains "You asked: tell me about FRAIM!". Underneath, it says "DAWID:" followed by a list of model configurations. A section titled "Short overview — what FRAIM is" lists the following points:

- FRAIM is a Python-based framework / starter application (the `fraim_app` project) for organizing, running and packaging machine-learning experiments and data-processing workflows.
- It contains a collection of tools, example workflows, tutorials and configuration patterns aimed at reproducible model development (datasets, training configs, models, CLI workflows) — useful as a template for new projects.

At the bottom, there's a "Main components & features" section, a "Model" dropdown set to "GPT-3 Mini (FAST)", a "Data Space" dropdown set to "faim:faim_app", and a large text input field labeled "Enter your question...".

DAWID: User Interface

— OUR functions!!

AI Forecasting: Learning to Predict the Next Weather State

Key idea

- ▶ The atmosphere has a **state** (a snapshot)
- ▶ Forecasting means predicting a **future state**
- ▶ A neural network learns a mapping:

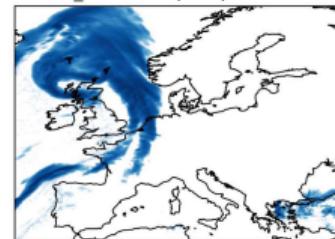
$$x(t) \Rightarrow x(t + 48h)$$

where x contains fields like temperature, wind, pressure, humidity.

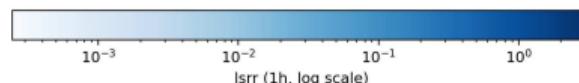
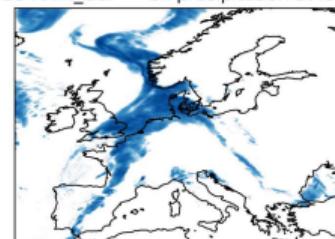
Training: learn from millions of examples
(reanalysis + observations).

After training, the model can produce forecasts extremely fast.

ICON-EU RAIN_GSP — 1h precipitation at lead 001 h



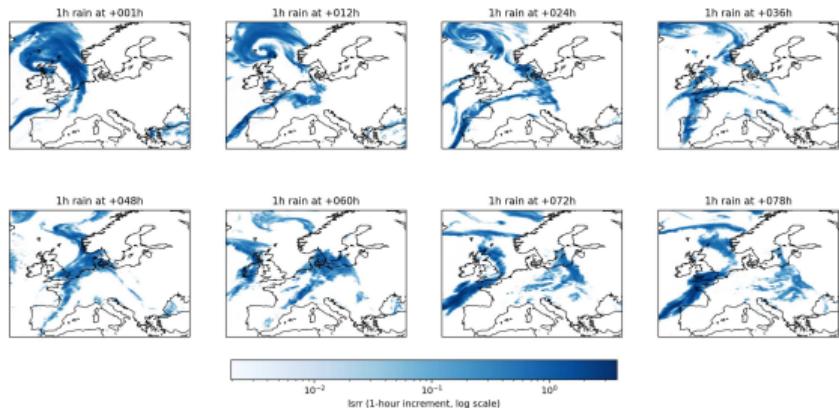
ICON-EU RAIN_GSP — 1h precipitation at lead 048 h



Neural Network Forecasting: One Step → Many Steps

How an AI forecast is produced

- ▶ The model learns: state now → state later
- ▶ Longer lead times come from **repeating the prediction step**
 $x(t) \rightarrow x(t + \Delta t) \rightarrow x(t + 2\Delta t) \rightarrow \dots$



Physics detail: accumulated variables

- ▶ precipitation is often stored as an **accumulated sum**
- ▶ to get **1-hour rain** at lead L :

$$\text{RAIN}(L) - \text{RAIN}(L - 1)$$

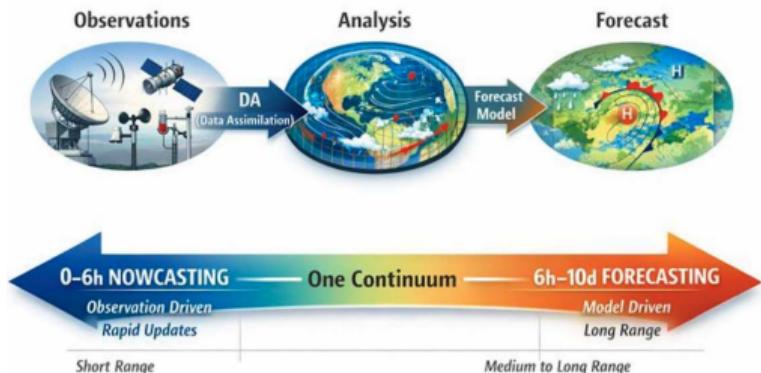
Example: ICON-EU 1-hour precipitation for multiple lead times.

Here: Nowcasting or NWP - depending on input, time scales and variables!

Nowcasting and NWP: One Continuum (Observations → DA → Forecast)

NWP with Data Assimilation (DA) is the full framework

- ▶ **observations** enter continuously:
radar, satellite, aircraft, surface stations, etc.
- ▶ **DA** combines obs + model:
best estimate of the 3D atmosphere at "now"
- ▶ then **forecast propagation** produces future weather



Nowcasting is the short-range regime

- ▶ lead times: **minutes to a few hours**
- ▶ very high weight of recent observations
- ▶ focus on rapidly evolving phenomena:
convective storms, precipitation cells

AI can support different parts: observations, DA, model emulation, and products.

There is no strict boundary: it is one continuum.

Forecasting in One Sentence

Weather forecasting = state estimation + prediction

- ▶ We estimate the **current atmospheric state**
- ▶ Then we predict how it changes in time

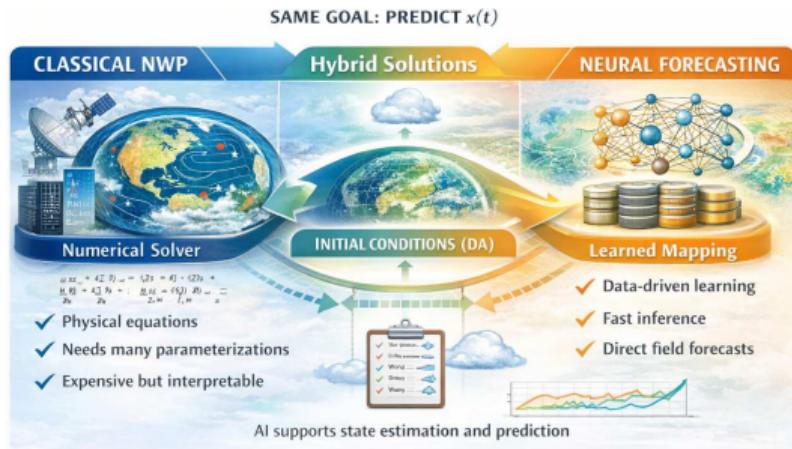
$$\underbrace{x(t_0)}_{\text{best estimate now}} \rightarrow \underbrace{x(t_0 + \Delta t)}_{\text{future state}}$$

Why it is hard

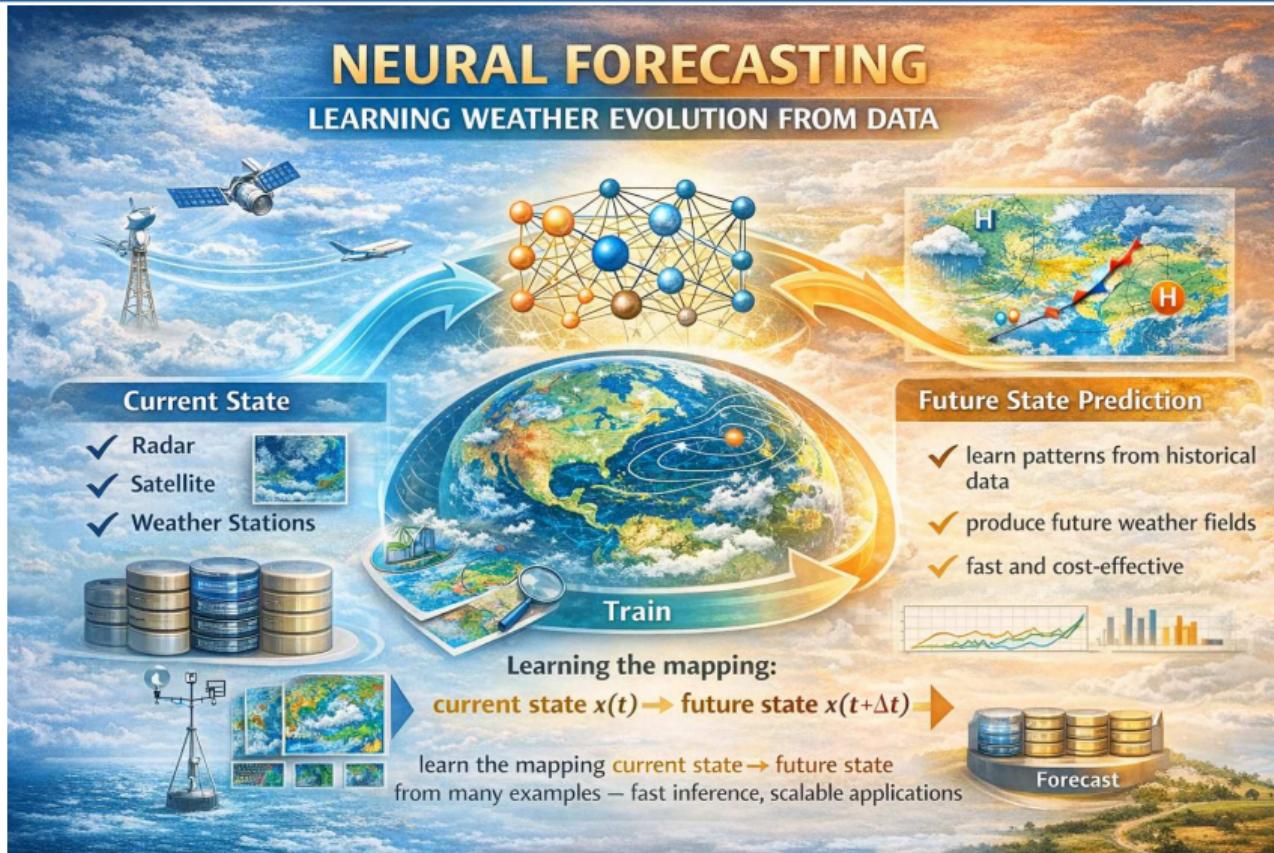
- ▶ the atmosphere is **chaotic**
- ▶ small errors grow with time
- ▶ observations are incomplete and noisy

So we quantify uncertainty

- ▶ ensembles: many forecasts instead of one



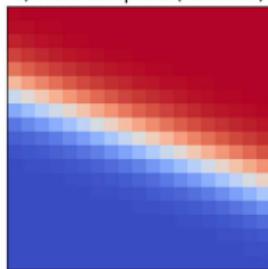
Forecasting: observation → analysis → prediction



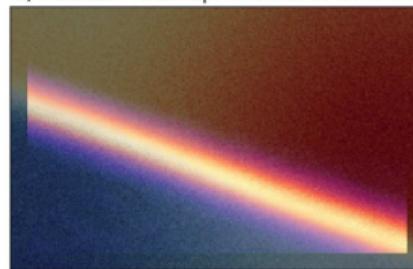
How a Neural Network Learns Forecasting: Front Template + Motion

Neural forecasting intuition: template → heatmap → shift → next field

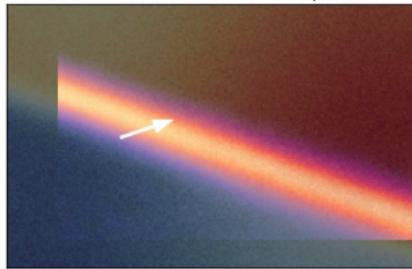
1) Front template ("stencil")



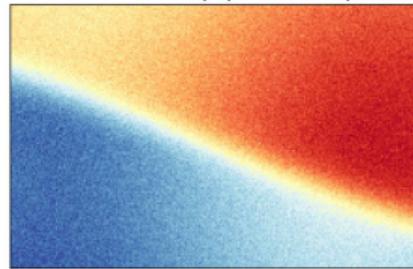
2) Detection heatmap: where does it match?



3) Motion direction: shift the pattern



4) Next forecast map (pattern transported)



Visual argument: the NN learns **templates** (front patterns) and how they **move / evolve** over time.

One Step, Many Steps: Templates + Transport + Accumulations

Neural forecasting learns reusable building blocks

- ▶ learn **templates** for structures:
fronts, rain bands, vortices, jets
- ▶ learn **how they move and change** from context
(e.g. wind, humidity, stability)

Then: one step → many steps

$$x(t) \rightarrow x(t + \Delta t) \rightarrow x(t + 2\Delta t) \rightarrow \dots$$

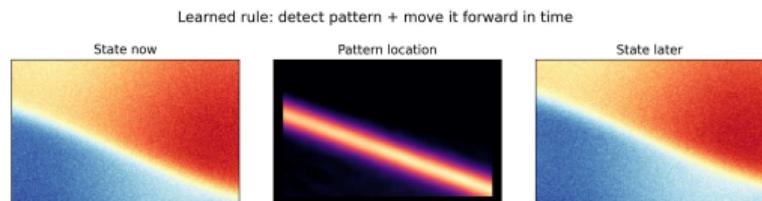
Important for users: accumulated products

- ▶ many outputs are stored as **accumulations**
- ▶ 1-hour precipitation at lead L is:

$$\text{RAIN}(L) - \text{RAIN}(L - 1)$$

Take-away:

Forecasting = detect patterns +
transport them forward in time.



Templates → heatmaps → shifted patterns → next map (repeated for longer lead times).

A Minimal Forecasting World: A Blob Moving on a Circle

We create a simple 1D world on $[0, 10]$:

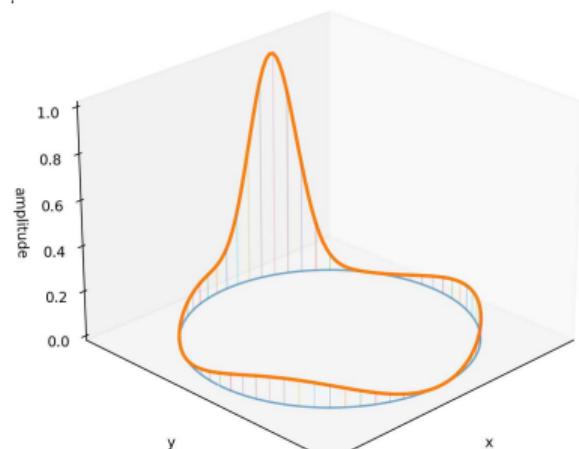
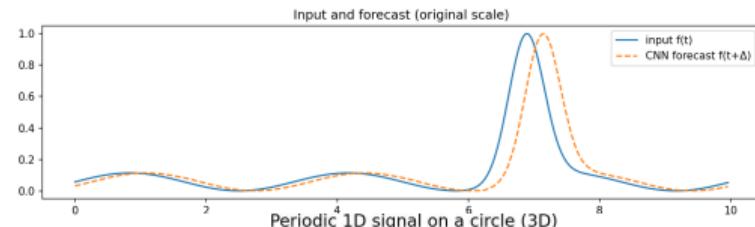
- ▶ a function $f(t, x)$ (a “blob”)
- ▶ periodic domain: $x = 0$ connects to $x = 10$
- ▶ dynamics: pure translation

$$f(t + \Delta, x) = f(t, x - \delta)$$

Forecasting task: given $f(t, \cdot)$, predict $f(t + \Delta, \cdot)$.

This captures the essence of advection:

structures move .



What the Neural Network Sees

Important: the input is the **full signal**, not single points.

The network gets all grid values of $f(t, x)$ at once:

$$f(t, \cdot) \in \mathbb{R}^N \quad (N = 256)$$

Stacking layers increases context

After 3 conv layers (all $k = 7$), each output point depends on roughly:

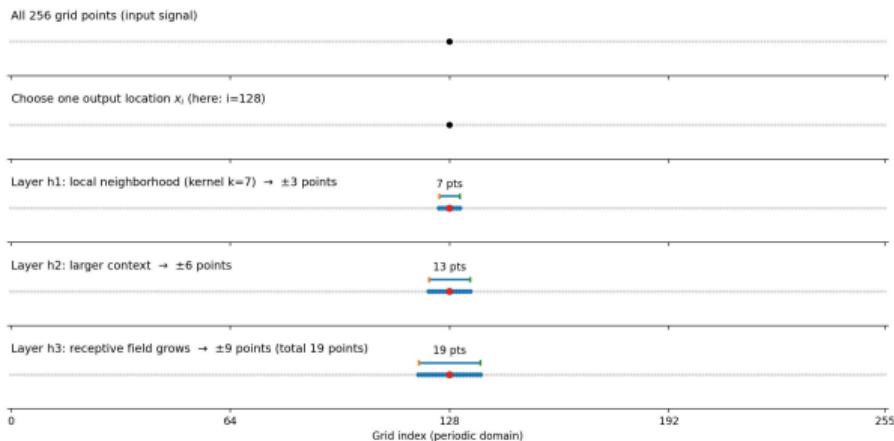
$$1 + 3 \cdot (k - 1) = 19 \text{ grid points}$$

But the computation is local: each output point is computed from a **neighborhood**.

Kernel size k = width of the filter

With $k = 7$, the network looks at:

$$[x_{i-3}, \dots, x_i, \dots, x_{i+3}]$$

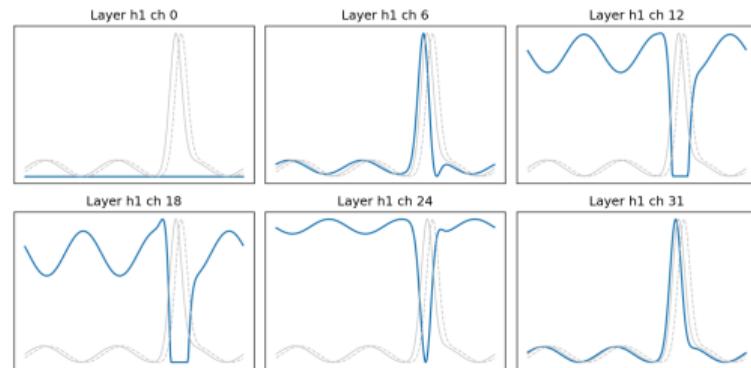


Inside the CNN: Many Feature Variables

The CNN transforms the input into many internal variables:

$$1 \rightarrow 32 \rightarrow 32 \rightarrow 32 \rightarrow 1$$

- ▶ input: one function $f(t, x)$
- ▶ hidden layers: 32 feature channels each
- ▶ output: predicted function $\hat{f}(t + \Delta, x)$



Example: 6 of 32 feature channels in layer h1.

Interpretation: each channel is a learned detector for local shapes (edges, slopes, curvature, ...).

The variables h_1, h_2, h_3 are **hidden feature maps** inside the CNN. They form a **latent feature space**: not observed, not physical, but learned. Each layer contains many channels (e.g. 32), extracting different local aspects of the signal. The forecast emerges by transforming $f(t, \cdot) \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow \hat{f}(t + \Delta, \cdot)$.

Why This Is *Not* Explicit Pattern Matching

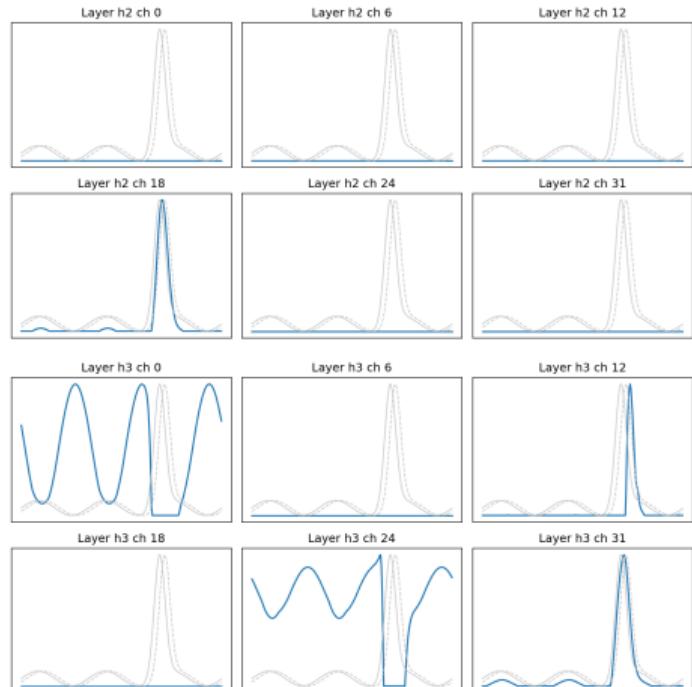
A common misconception is:

"The network finds the blob pattern and just copies it forward."

What actually happens:

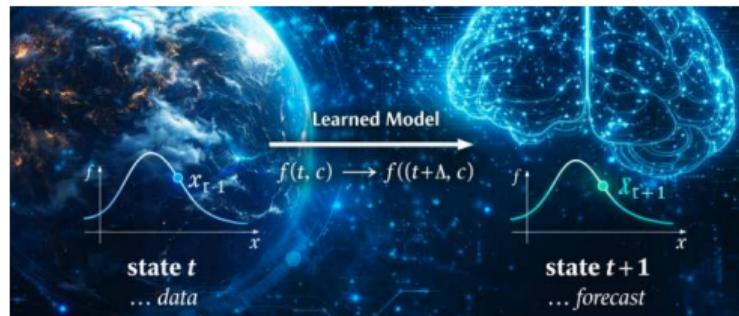
- ▶ the CNN computes many internal feature variables (32 channels)
- ▶ each channel responds to different local aspects
- ▶ the forecast is produced by recombining these features

Key point: the NN does not store templates — it learns a continuous mapping from input to forecast.



Example: 6 of 32 feature channels in layer h3.

What We Can Do Already



We already have strong building blocks in place:

- ▶ We have created a framework for AI-based weather forecasting with **Anemoi**

- ▶ We have developed libraries (**mfa**, **FRAIM**) for:
 - ▶ downscaling and high-resolution products
 - ▶ road weather services
 - ▶ high-impact weather feature extraction
 - ▶ weather interpretation and explainability
 - ▶ nowcasting of observation fields (radiation, precipitation, ...)

- ▶ We have gained expertise across the full AI/ML value chain in weather services and international organizations.
- ▶ We have created networking and structures for joint development and sustainable research — connected to leading global AI/ML developments.

What We Can Do Already (Operational AI Forecasting)

AI forecasting is already operational.

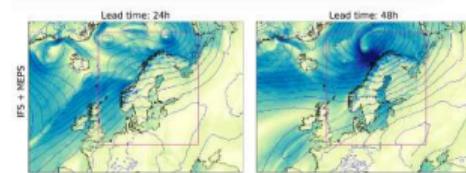
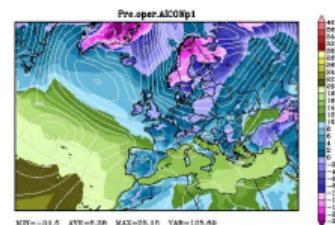
In Europe, several AI-based forecasting systems are now in production use (e.g. AIFS, AICON, BRIS). They complement classical NWP by providing:

- ▶ fast state-to-state forecasts (minutes instead of hours)
- ▶ competitive large-scale skill for key variables
- ▶ robust baselines and rapid experimentation

Key message: AI is no longer “research only” — it is part of the operational toolbox .

This opens a clear opportunity:

- ▶ combine NWP + AI hybrids
- ▶ integrate new observation-driven products
- ▶ accelerate development cycles via ML pipelines



Library Ecosystem: Complementary Capabilities

We are building a strong ecosystem of AI libraries

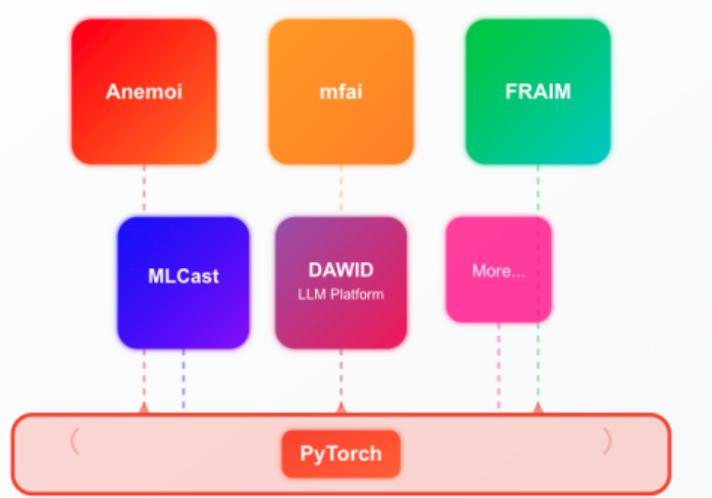
Different libraries contribute complementary capabilities:

- ▶ **Anemoi**: end-to-end AI weather models
- ▶ **mfaι**: vision transformers and many applications
- ▶ **MLCast**: nowcasting library, observation based
- ▶ **FRAIM**: products and services, full value chain

Positive outlook

- ▶ no single library needs to do everything
- ▶ modular building blocks enable rapid innovation
- ▶ shared standards ⇒ interoperability and reuse

Key message: Together, these components form a platform for scalable AI in weather services .



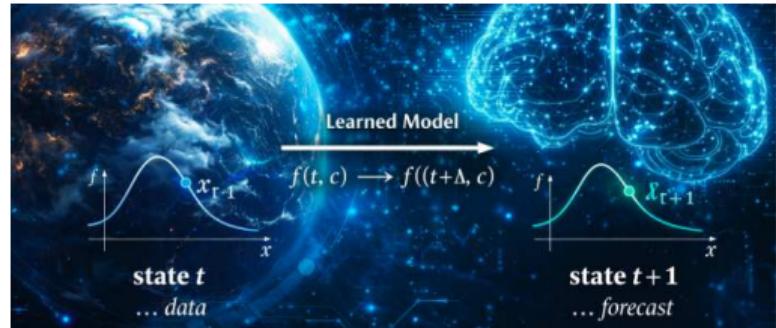
Ecosystem view: different strengths, one common mission.

Why Physics Matters: Future AI Forecast Systems Will Be Physics + AI

Weather is not just a pattern problem

Forecasting is about **consistent evolution** of a physical state :

- ▶ conservation laws (mass, energy, water)
- ▶ balances and constraints (geostrophy, stability)
- ▶ multiscale interactions (local storms \leftrightarrow global flow)



AI gives speed and learning.

Physics gives truth and trust.

Why “pure AI” can fail

- ▶ excellent short-term skill can still drift long-term
- ▶ small violations accumulate (mass / moisture / energy)
- ▶ rare extremes need physics consistency, not just averages

- ▶ AI learns **corrections / closures**
- ▶ **constraints and invariants built into the model**
- ▶ physics-based evaluation & reliability

What You Can Do (A): Use LLMs as a Tool — You Stay the Master

Use AI for: understanding, exploring, developing

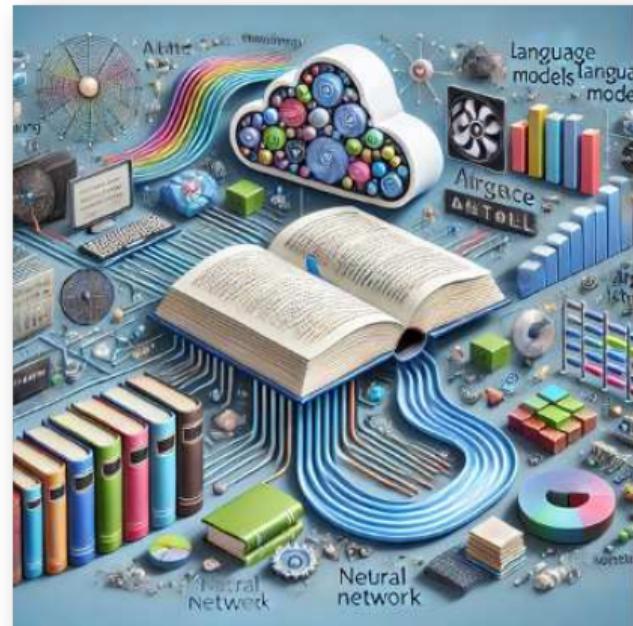
Think of LLMs as a power tool for knowledge work:

- ▶ explain unfamiliar concepts in your context
- ▶ explore alternatives and trade-offs quickly
- ▶ generate drafts, code sketches, slide structure
- ▶ summarize documents, meetings, research threads

The right mindset

- ▶ you are the pilot, the AI is the assistant
- ▶ ask for options, then decide yourself
- ▶ validate important facts (sources, experiments)

Key message: AI needs guidance — quality comes from your questions and your checks .



Human judgment stays in control.

What LLMs Do Today — and What “World Models” Add Next

LLMs today (what they actually do)

- ▶ generate text/code by next-token prediction
- ▶ strong at language tasks: explain, summarize, draft, transform
- ▶ can appear to reason, but truth not guaranteed
- ▶ without grounding: can be confidently wrong

World-model approaches (the next step)

- ▶ learn explicit state representations of a system (world state)
- ▶ learn dynamics : $\text{state}(t) + \text{action} \rightarrow \text{state}(t+1)$
- ▶ enable planning, long-horizon consistency, and controllable actions
- ▶ usually needs grounding (data, sensors, simulators, tools)

Message: LLMs are powerful interfaces — world models aim at reliable dynamics .

LLMs for knowledge work

Be careful:

- ▶ verify facts, numbers, and operational details
- ▶ demand sources or reproduce with scripts
- ▶ watch for missing assumptions and edge cases

Take full advantage:

- ▶ accelerate understanding and documentation
- ▶ structure tasks into steps, checklists, and options
- ▶ generate first drafts and improve clarity
- ▶ connect to tools/agents for grounded actions

**Fast thinking + human validation
= real productivity.**

What You Can Do (B): Rebuild Services with Tools, Agents, and Clear APIs

Use the AI revolution to modernize services

LLMs work best when they can call clear interfaces. This is an opportunity to rebuild our services in a clean way:

- ▶ define small, robust functions (tools)
- ▶ combine them into agent workflows
- ▶ expose everything as API-based services

Why this is powerful

- ▶ clarity: inputs/outputs become explicit
- ▶ reuse: one tool serves many applications
- ▶ automation: agents connect tools: end-to-end
- ▶ acceleration: faster prototyping and delivery



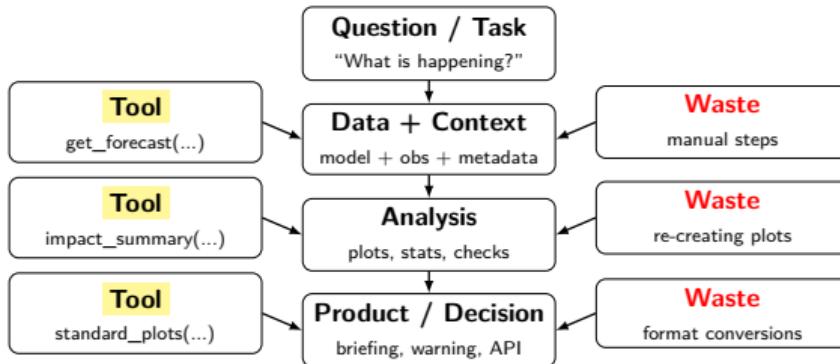
Tool calling turns services into reusable building blocks.

Key message: LLMs need good tool definitions — we can use this to create better services for ourselves.

Slide 39: Rebuild Services — Analyze Work, Find Waste, Build Tools

Step 1: Look at our daily workflow

- ▶ Where do we **repeat** the same steps?
- ▶ Where do we **copy/paste** between systems?
- ▶ Where do we **wait** for data, plots, approvals?
- ▶ Where do we **lose context** (emails, chats, files)?



Replace repeated manual work by reusable tools with clear interfaces.

Step 2: Turn waste into tools

- ▶ define a clear **input/output interface**
- ▶ make it callable (API / function / agent tool)
- ▶ reuse it everywhere: DAWID, scripts, services, pipelines

Key message: If a task repeats, it should become a **tool**.

What You Can Do (C): Bring Your Domain Expertise — Make AI Your Own

AI becomes powerful when it is grounded in meteorology

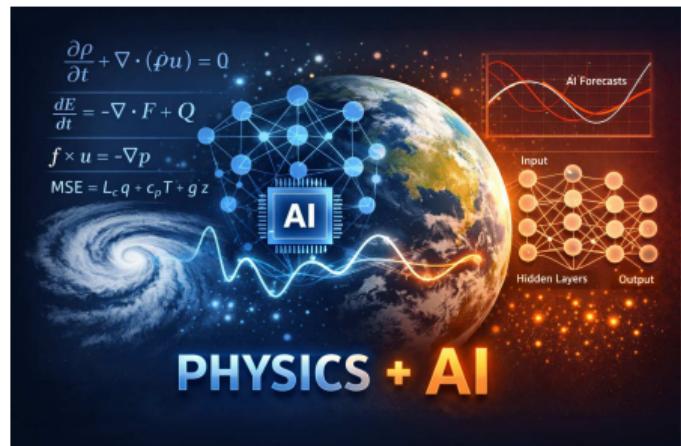
LLMs and neural networks do not automatically understand:

- ▶ processes, states, and transitions
- ▶ limits and plausibility
- ▶ physical constraints and balances
- ▶ weather regimes and rare extremes

Your expertise is the missing ingredient

- ▶ define what matters: variables, events, diagnostics
- ▶ define what is allowed: limits, physics, consistency
- ▶ define what is useful: products, warnings, explanations

Key message: Do not outsource AI — embed your knowledge into tools, workflows, and constraints.



Domain expertise turns AI into trust.

Python and AI/ML for Weather, Climate and Environmental Applications

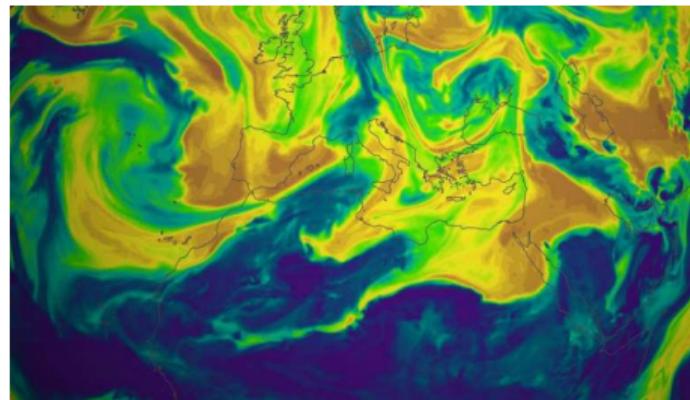


Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Why Emulators in Numerical Weather Prediction?



- ▶ High-resolution NWP is computationally expensive
- ▶ Many applications do not require full physical fidelity
- ▶ Fast forecasts enable new applications and scale

Emulator concept

- ▶ Learns the forecast step: state → state
- ▶ Operates within an existing NWP ecosystem
- ▶ Preserves grids, variables, and semantics

Goal of this lecture

- ▶ Understand emulator-based AI systems
- ▶ Compare AIFS and AICON
- ▶ Walk through AICON step by step

Emulators complement numerical models — they do not replace physical understanding.

Big-Tech AI Weather Models: Architectural Landscape

GraphCast (Google DeepMind)

- ▶ Graph Neural Network (GNN)
- ▶ Icosahedral grid, $\sim 0.25^\circ$
- ▶ Multi-mesh message passing
- ▶ Deterministic medium-range forecasts

Pangu-Weather (Huawei)

- ▶ 3D Transformer architecture
- ▶ Regular lat-lon grid, $\sim 0.25^\circ$
- ▶ Attention over space and vertical levels
- ▶ Image-like representation of atmosphere

GenCast (Google DeepMind)

- ▶ Diffusion model on graphs
- ▶ Icosahedral grid, $\sim 1^\circ - 0.25^\circ$
- ▶ Probabilistic forecasting
- ▶ Explicit uncertainty representation

FourCastNet (NVIDIA)

- ▶ Fourier Neural Operator (FNO)
- ▶ Regular lat-lon grid, $\sim 0.25^\circ$
- ▶ Spectral convolution in Fourier space
- ▶ Extremely fast inference

All models replace the forecast step — none are embedded in classical NWP systems.

Research and Operational AI Weather Systems

Aurora (Microsoft)

- ▶ Transformer-based foundation model
- ▶ Regular lat–lon grids
- ▶ Multi-task Earth-system scope
- ▶ Research-driven, not NWP-native

AIFS (ECMWF)

- ▶ Graph Neural Network (Anemoi)
- ▶ ECMWF Flagship ML Model
- ▶ Trained on ECMWF reanalysis ERA
- ▶ Operational global forecast system

BRIS (Met Norway)

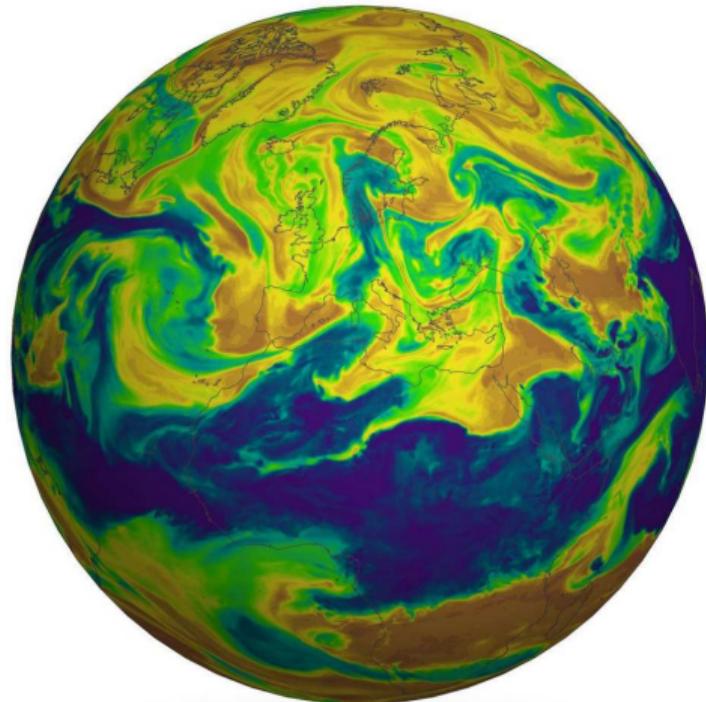
- ▶ Anemoi-based national system
- ▶ Nested Grid Approach, high resolution
- ▶ Independent training, AIFS-aligned
- ▶ Operational national usage

AICON (DWD)

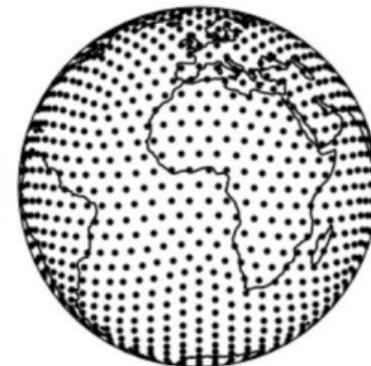
- ▶ Graph Neural Network (Anemoi)
- ▶ Native ICON triangular grid
- ▶ Trained on ICON-DREAM reanalysis
- ▶ Operational ICON emulator

European Meteorological Infrastructure is carrying out exciting development.

Anemoi based Emulators: Fields and Grids



Icosahedral Grid Representation



Triangular ICON mesh used as the computational graph for message passing and inference.

AICON learns a mapping on the ICON mesh, not on a Cartesian grid.

ICON-DREAM Reanalysis: High-quality training basis for AI emulators

Motivation

- ▶ reanalyses are key for **climate services** and many applications
- ▶ now also essential as **training basis** for AI-based NWP emulators

ICON reanalysis framework at DWD

- ▶ **ICON-DREAM:** global-to-regional ensemble reanalysis
- ▶ **ICON-FORCE:** 2.1 km ensemble reanalysis for Central Europe
- ▶ focus on **Europe** and on **continuous updates** (2010–present)

ICON-DREAM delivers **consistent, high-resolution data**

ICON-DREAM: key characteristics

- ▶ global ICON at **13 km**, 120 levels
- ▶ two-way nest over Europe at **6.5 km**
- ▶ DA: **LETKF** with **20-member ensemble** for *B*-covariances
- ▶ ensemble at 40 km (20 km over Europe)

ICON-FORCE (2 km):

- ▶ hourly LETKF (KENDA), incl. radar + SEVIRI
- ▶ 2-moment microphysics, snow / SST analyses

Outlook: backward extension to the 1980s using rescued obs + CDR products.

AICON Walkthrough: Environment Setup

Thanks to Florian Prill for a great Notebook!

Core ML stack

- ▶ Python ≥ 3.10
- ▶ PyTorch (CPU or CUDA)
- ▶ PyTorch Lightning
- ▶ PyTorch Geometric (PyG)

Anemoi framework

- ▶ anemoi-core (graphs, models, training)
- ▶ anemoi-datasets (Zarr + YAML I/O)
- ▶ anemoi-inference (generic inference)
- ▶ Supporting packages (e.g. transforms)

Data and configuration

- ▶ Zarr (chunked training data)
- ▶ Xarray (inspection and analysis)
- ▶ eccodes + earthkit (GRIB2 handling)
- ▶ Hydra / OmegaConf (YAML configuration)

Recommended setup

- ▶ Pinned requirements.txt
- ▶ Virtual environment or container
- ▶ Consistent library versions

Reproducibility depends on a stable, fully controlled software environment.

AICON Walkthrough: Manual Setup (Packages and ecCodes)

Install required Python packages

Pinned Python environment

```
1 # Install all required
   packages (version!)
2 %pip install --no-cache-
   dir -U -r ../
   requirements.txt
3
4 # Inspect installed
   versions
5 !pip freeze
```

Exact package versions are critical for PyTorch, PyG, Lightning, and Anemoi compatibility.

Configure ecCodes definitions

```
import os
from pathlib import Path

# Path to ICON/DWD ecCodes definitions
# (provided with the repo)
edzw_defs = Path("eccodes/definitions.edzw").resolve()

# Standard ecCodes definitions (system install)
std_defs = "/path/to/eccodes/definitions"

os.environ["ECCODES_DEFINITION_PATH"] = \
    f"{edzw_defs}:{std_defs}"
```

ICON GRIB decoding requires extended definition tables beyond standard ecCodes.

AICON Walkthrough: Notebook Initialization

Jupyter and runtime setup

Notebook basics

```
1 % Enable autoreload for
    iterative development
2 %load_ext autoreload
3 %autoreload 2
4
5 # Standard imports
6 import os
7 import numpy as np
8 import torch
```

Ensures code changes are picked up and the runtime environment is visible.

Initialization is not boilerplate — it defines the experimental contract.

Determinism and sanity checks

Reproducibility checks

```
1 # Fix random seeds
2 torch.manual_seed(42)
3 np.random.seed(42)
4
5 # Check PyTorch device
6 device = torch.device("cuda" if
    torch.cuda.is_available()
    else "cpu")
7 print("Using device:", device)
```

Deterministic behavior is essential for debugging and scientific comparison.

AICON training setup (Anemoi): config-driven workflow

Main idea

- ▶ training is
fully controlled by YAML configs
- ▶ Hydra composes config blocks:
 - ▶ data (Zarr), variables,
normalization
 - ▶ model architecture (AICON /
GNN)
 - ▶ trainer settings (epochs, batch,
strategy)

Workflow

- ▶ choose config (start from integration test)
- ▶ instantiate trainer
- ▶ run training (checkpoints + logger)

Notebook pattern (core lines)

```
config_filename = "test_aicon_01.yaml"  
  
# Hydra loads and resolves config  
config = load_config(config_filename)  
  
# Anemoi trainer wraps Lightning  
trainer = AnemoiTrainer(config)  
  
# run training  
trainer.train()
```

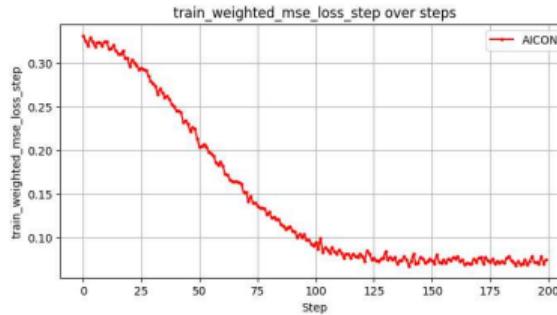
Outputs

- ▶ checkpoints: last.ckpt,
inference-last.ckpt
- ▶ logs: TensorBoard / MLflow-style metrics

Training diagnostics: loss curve and experiment logger

What we monitor

- ▶ training loss vs epoch/step
- ▶ validation loss (generalization)
- ▶ learning rate schedule
- ▶ walltime + throughput



Why it matters

- ▶ early detection of instability
- ▶ reproducible experiments
- ▶ compare model variants

Take-away: training is observable and traceable .

```
class GraphicalLiveLogger(Logger):
    def __init__(self, interval=10):
        super().__init__()
        self.metric_name = "train_weighted_mse_loss_step"
        self.metrics_history = []

    @property
    def name(self): return "LiveLogger"
    @property
    def version(self): return "0.1"

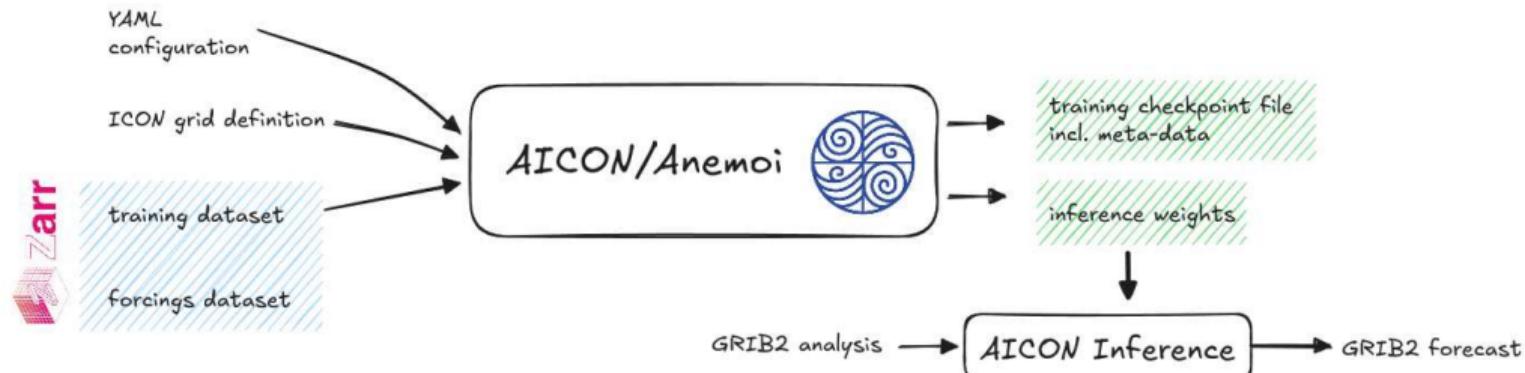
    def log_hyperparams(self, *args, **kwargs): pass

    def log_metrics(self, metrics, step):
        self.metrics_history.append((step, metrics))

    def finalize(self, status):
        steps, values = zip(*[s,m[self.metric_name]] for s,m in self.metrics_history if self.metric_name in m)
        plt.figure(figsize=(8,4))
        plt.plot(steps, values, 'o-', ms=2, label="AICON", color="red")
        plt.title(f"{self.metric_name} over steps"); plt.xlabel("Step"); plt.ylabel(self.metric_name)
        plt.grid(True); plt.legend(); plt.show()

trainer.model.logger_enabled = True
trainer.loggers = GraphicalLiveLogger()
```

AICON training workflow (overview)



Pipeline: ICON data → graph dataset → training config → trainer → checkpoints + logs

Training configuration: input and outputs

Input data (from config)

```
text
1 ICON mesh (NetCDF):
2   <.../icon_mesh.nc>
3
4 Training interval:
5   <YYYY-MM-DD HH> ... <YYYY-MM-
6   DD HH>
7 Data files:
8   <...> (joined list)
```

Output directories

```
text
1 Checkpoints:
2   <.../checkpoints/...>
3
4 Logs (incl. MLFlow):
5   <.../logs/...>
```

Take-away: config \Rightarrow reproducible paths and artifacts.

Key point: a single config fixes both data geometry
and time range .

Training dataset (Zarr) inspected as Xarray

Key idea

- ▶ dataset is a **4D tensor** :
(time, variable, ensemble, cell)
- ▶ efficient I/O via **chunking**
(Dask)
- ▶ includes metadata:
 - ▶ latitudes / longitudes
 - ▶ variable-wise statistics
(mean, std, min/max)

**Take-away: this is ML-ready
data with rich metadata.**

▶ Dimensions: (variable: 84, time: 724, ensemble: 1, cell: 11520)

▶ Coordinates: (0)

▼ Data variables:

count	(variable)	float64	dask.array<chunksize=(84,), me...  
data	(time, variable, ensemble, cell)	float32	dask.array<chunksize=(1, 84, 1,...  
dates	(time)	datetime64[s]	dask.array<chunksize=(724,), m...  
has_nans	(variable)	object	dask.array<chunksize=(84,), me...  
latitudes	(cell)	float64	dask.array<chunksize=(11520,),...  
longitudes	(cell)	float64	dask.array<chunksize=(11520,),...  
maximum	(variable)	float64	dask.array<chunksize=(84,), me...  
mean	(variable)	float64	dask.array<chunksize=(84,), me...  
minimum	(variable)	float64	dask.array<chunksize=(84,), me...  
squares	(variable)	float64	dask.array<chunksize=(84,), me...  
stdev	(variable)	float64	dask.array<chunksize=(84,), me...  
sums	(variable)	float64	dask.array<chunksize=(84,), me...  

Vertical levels: selected ICON layers in the dataset

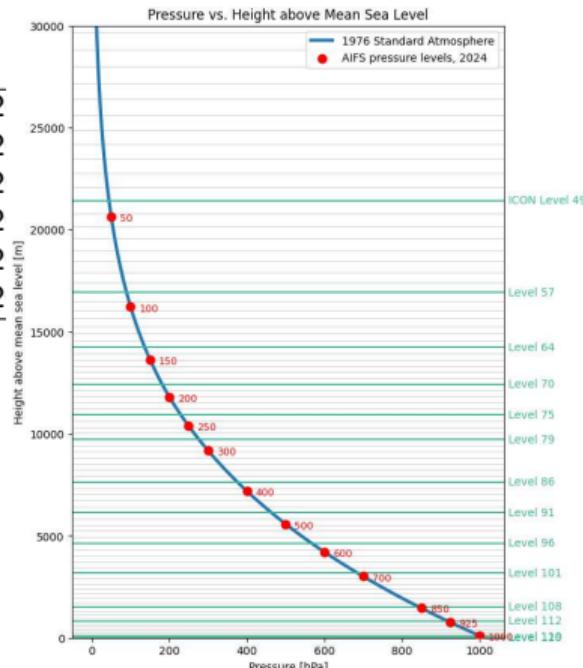
Variables on model levels

shortName	Name	Levels
P	Pressure	49,57,64,70,75,79,86,...,12
QV	Specific humidity	49,57,64,70,75,79,86,...,12
T	Temperature	49,57,64,70,75,79,86,...,12
U	U-component of wind	49,57,64,70,75,79,86,...,12
V	V-component of wind	49,57,64,70,75,79,86,...,12
W	Vertical velocity (geom.)	49,57,64,70,75,79,86,...,12

Forcings / static features (examples):

EMIS_RAD, FR_LAND, FR_LAKE, HSURF, Z0,
 insolation, SSO_*, sin/cos(latitude, longitude, local_time,
 julian_day)

Idea: multi-level state + forcings \Rightarrow complete ML
 input for AICON.



ICON level structure: full model vertical grid
 (selection highlighted).

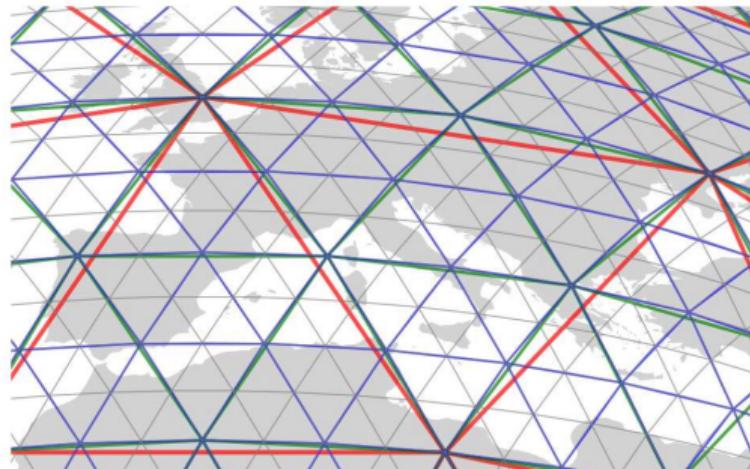
ICON multimesh: the hidden graph behind AICON

Key idea

- ▶ AICON does not run on a regular lat–lon grid
- ▶ it uses ICON's hierarchical triangular mesh
- ▶ the GNN operates on a hidden multi-mesh

Multi-mesh principle

- ▶ union of coarse-to-fine subgraphs
 $RnB0 \cup RnB1 \cup \dots \cup RnBk$
- ▶ mixes short-range + long-range edges
- ▶ similar spirit as GraphCast multi-mesh



Hidden mesh edges shown level-wise: coarse levels enable long-range interaction.

Take-away: the multi-mesh gives the GNN both local physics and global context .

AICON model: encoder – graph processor – decoder

Graph-to-graph forecasting

- ▶ input: state + forcings on ICON cells (nodes)
- ▶ output: next-step state (multi-var, multi-level)

Architecture blocks

- ▶ **Encoder:** maps raw variables to latent node/edge features
- ▶ **Processor:** message passing on the mesh graph (several layers)
- ▶ **Decoder:** projects latent features to physical output variables

Take-away: encoder/decoder makes the network variable-agnostic in latent space.

```
encoder:  
    _target_: anemoi.models.layers.mapper.GraphTransformerForwardMapper  
    _convert_: all  
    trainable_size: ${model.trainable_parameters.data2hidden}  
    sub_graph_edge_attributes: ${model.attributes.edges}  
    num_chunks: 2  
    cpu_offload: ${model.cpu_offload}  
    mlp_hidden_ratio: 4  
    num_heads: 16  
    qk_norm: false  
    layer_kernels:  
        Linear:  
            _target_: torch.nn.Linear  
            _partial_: true  
        Activation:  
            _target_: torch.nn.GELU  
decoder:  
    _target_: anemoi.models.layers.mapper.GraphTransformerBackwardMapper  
    _convert_: all  
    trainable_size: ${model.trainable_parameters.hidden2data}  
    sub_graph_edge_attributes: ${model.attributes.edges}  
    num_chunks: 1  
    cpu_offload: ${model.cpu_offload}  
    mlp_hidden_ratio: 4  
    num_heads: 16  
    qk_norm: false  
    initialise_data_extractor_zero: false  
    layer_kernels:  
        Linear:  
            _target_: torch.nn.Linear  
            _partial_: true
```

Encoder maps physics variables → latent graph; decoder maps latent → output fields.

Graph structure: local neighborhoods on the ICON mesh

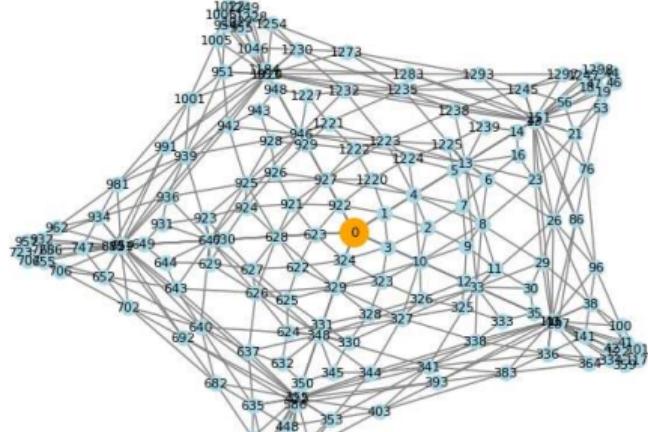
Why this matters

- ▶ AICON runs on the ICON grid as graph neural network (GNN)
 - ▶ each ICON cell \Rightarrow one graph node
 - ▶ edges connect mesh neighbors

Message passing view

- ▶ prediction at one node uses information from:
 - ▶ **1st neighbors** (directly connected)
 - ▶ **2nd neighbors**
(neighbors-of-neighbors)
 - ▶ this defines the local receptive field of the GNN

Take-away: GNNs learn transport + interaction patterns through local connectivity .



Example: subgraph around one node (orange) with 1st and 2nd neighbors.

Graph Transformer attention: which neighbors matter?

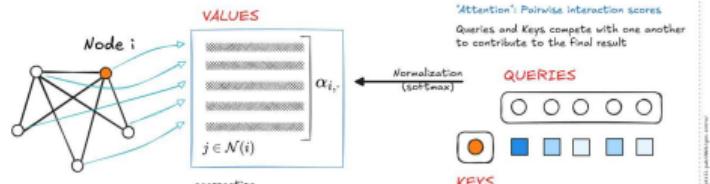
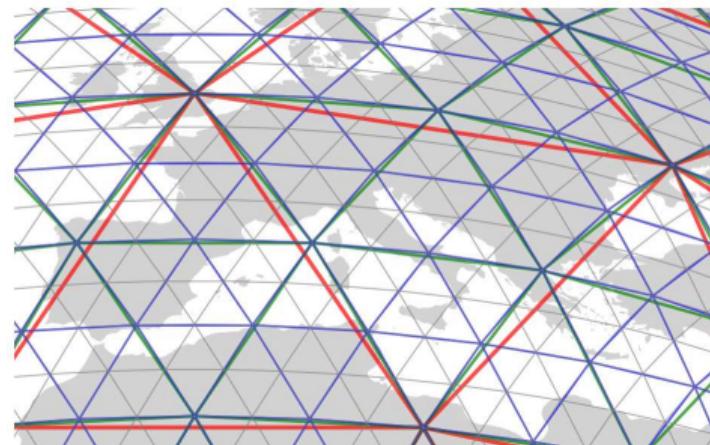
Graph Transformer vs. plain message passing

- ▶ classic GNN: neighbors contribute via fixed averaging / sum
- ▶ GraphTransformer: each edge gets a learned attention weight

Interpretation

- ▶ for one receiver node i , all incoming edges ($j \rightarrow i$)
- ▶ attention weights $\alpha_{i,j}$ sum to 1 (per head)
- ▶ thick edges \Rightarrow high influence of neighbor j

Take-away: the model learns where to look on the graph depending on flow regime and structure.



Example: (top) ICON multi-mesh graph (coarse + fine connections); (bottom) attention setup

Transfer learning: reuse weights on a finer hidden mesh

Motivation

- ▶ training on the finest multi-mesh is expensive
- ▶ idea: **train on coarse mesh** and transfer to finer mesh

What transfers well?

- ▶ most encoder/processor/decoder weights
- ▶ learned graph attention patterns are largely mesh-agnostic

What must be adapted?

- ▶ graph-dependent parameters:
 - ▶ hidden node coordinates / embeddings
 - ▶ extra trainable node/edge attributes
- ▶ these are re-initialized and fine-tuned on the new mesh

Example (AICON walkthrough)

- ▶ change hidden mesh resolution:
 - ▶ `max_level_multimesh: 3 → 4`
- ▶ load weights only + enable transfer learning

Config idea (YAML)

```
training:  
  load_weights_only: true  
  transfer_learning: true  
  run_id: <pretrained_run_id>
```

```
graph:  
  nodes:  
    icon_mesh:  
      node_builder:
```

Operational inference at DWD: why a dedicated tool?

Goal

- ▶ run AICON forecasts **operationally** and **reproducibly**
- ▶ integrate into existing NWP infrastructure (GRIB2, workflows, monitoring)

Why not only anemoi-inference?

- ▶ DWD-specific I/O and preprocessing (GRIB2 conventions)
- ▶ additional input engineering (e.g. soil moisture index etc.)
- ▶ packaging as stable runtime for ops (container)
- ▶ hooks for monitoring and verification pipelines

Key ingredients

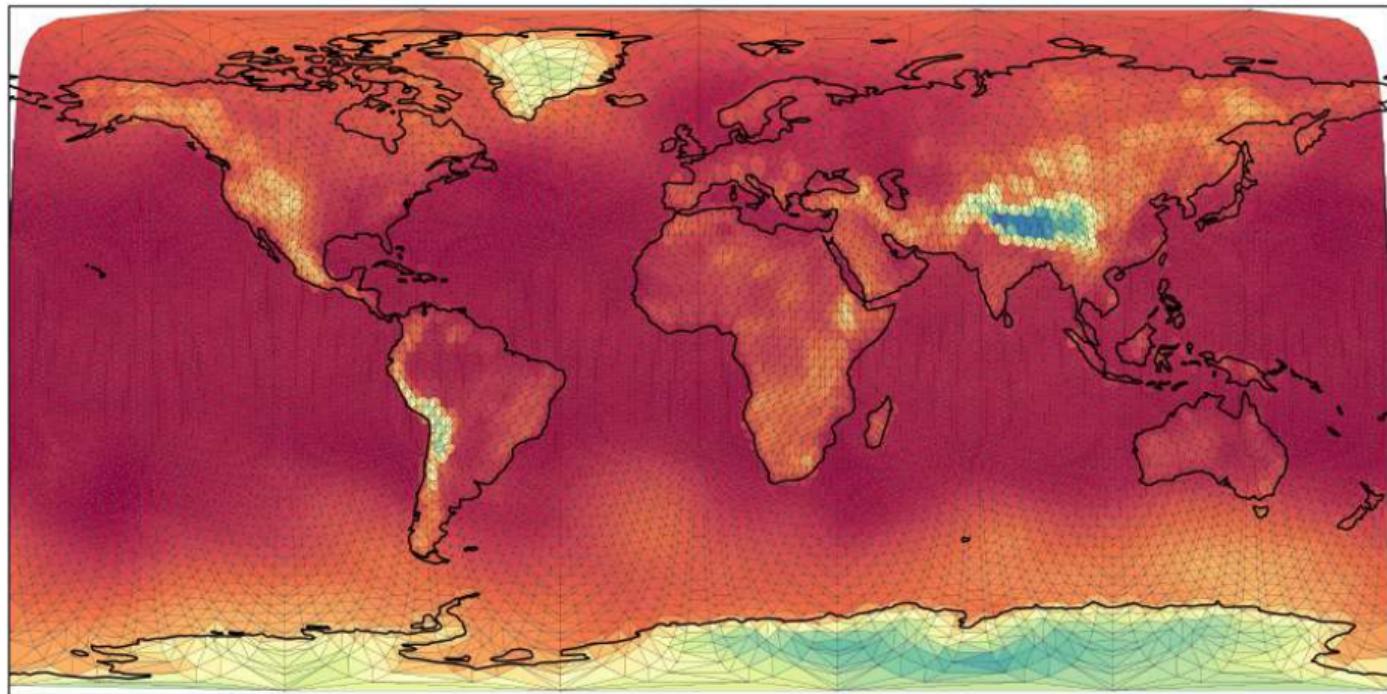
- ▶ **checkpoint:** inference-last.ckpt
- ▶ **input:** ICON / reanalysis data streams
- ▶ **output:** GRIB2 forecast products

Operational packaging

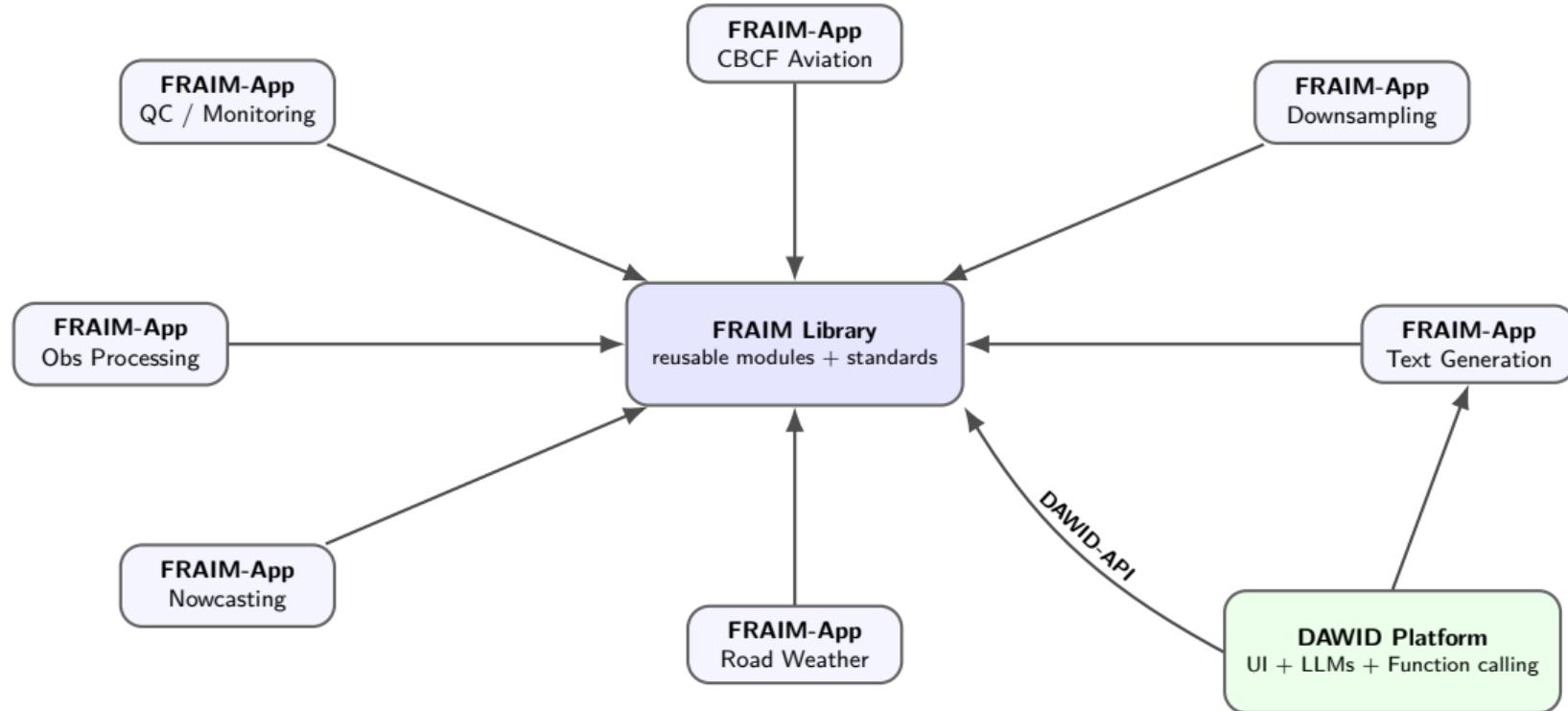
- ▶ Apptainer / Singularity container
- ▶ fixed paths to constants + weights
- ▶ versioning: container + config + checkpoint

**Take-away: DWD inference tool =
bridge from ML model to ops .**

Demo: AICON forecast from operational inference



FRAIM Ecosystem: Central Library, FRAIM-Apps, and DAWID Platform



Anemoi vs. FRAIM — What are they about?

Anemoi (ECMWF & Partners)

- ▶ international collaboration to build ML-based forecast models
- ▶ end-to-end framework for training & inference
- ▶ well-defined pipeline:
datasets → graphs/models → training → inference/deploy
- ▶ **Open Source**, pan-European community

Typical question:

How do we train and operate an ML forecasting model operationally?

FRAIM (E-AI / DWD / Partners)

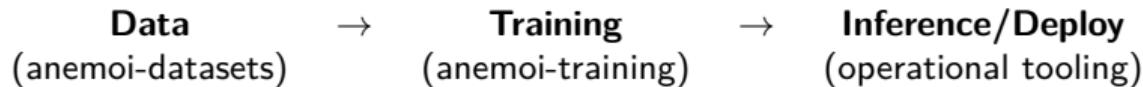
- ▶ international collaboration focused on many AI applications
- ▶ **method toolbox / modular framework** across a broad portfolio:
products, services, and smaller AI components
- ▶ platform/integration view:
standards, building blocks, reuse
- ▶ **umbrella framework** (forecasting is only one use case)

Typical question:

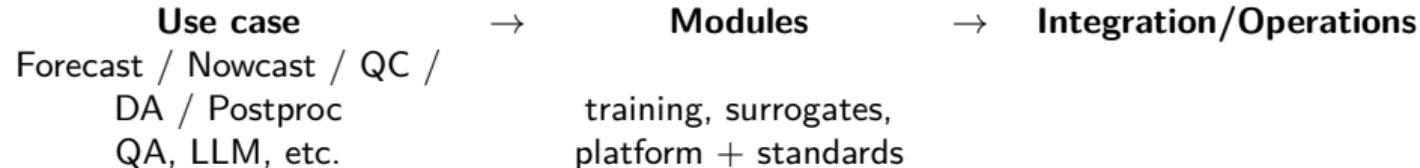
How do we build a modular AI ecosystem for many meteorological products and services?

Pipeline view: where does each framework sit?

Anemoi: lifecycle of an ML forecast model



FRAIM: system / toolbox view across multiple AI components



Take-away:

Anemoi is **forecast-model-centric**, FRAIM is **architecture- & reuse-centric**.

Concrete differences and a good integration story

Differences (short & concrete)

- ▶ **Scope:** Anemoi = end-to-end ML forecasting
FRAIM = modular AI toolbox for many meteorological products
- ▶ **Core artifacts:**
 - ▶ **Anemoi:** datasets, graphs, model weights, training pipelines
 - ▶ **FRAIM:** central library + reusable modules + standards
- ▶ **Organization principle:**
 - ▶ Anemoi: one coherent forecasting stack
 - ▶ FRAIM: many use-case driven FRAIM-Apps (each in its own repo)

How does this combine well?

- ▶ FRAIM can integrate mfa or Anemoi as a forecasting or processing engine
- ▶ FRAIM then provides:
 - ▶ product-specific pipelines (QC, DA, verification, monitoring, etc.)
 - ▶ deployment patterns and operational standards

One-liner:

Anemoi = forecast-model factory;
FRAIM = modular product ecosystem.

DAWID — LLM Platform, Tools, and FRAIM Integration

DAWID capabilities

- ▶ **LLM user interface**
 - ▶ chat + document context (RAG)
 - ▶ role-based workflows (science / dev / ops)
- ▶ **Function calling**
 - ▶ controlled execution of domain functions
 - ▶ structured I/O and provenance
- ▶ **Agents (multi-step)**
 - ▶ plan → call tools → validate

DAWID-API integration

- ▶ **Unified API to LLMs and tools**
 - ▶ multi-LLM backends
 - ▶ tool/function endpoints
- ▶ **Link to FRAIM-Apps**
 - ▶ FRAIM-App → DAWID-API: reasoning + tooling
 - ▶ consistent interface across products

**Take-away: DAWID connects
LLMs + tools + FRAIM-Apps .**

Key idea: DAWID is the **interactive AI cockpit** .

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

AI Data Assimilation — Why Does It Matter?

Numerical Weather Prediction (NWP)

Weather forecasts are produced by integrating high-dimensional dynamical models forward in time.

However:

- ▶ The atmosphere is **chaotic**
- ▶ Small initial errors grow rapidly
- ▶ Forecast skill is **dominated by initial conditions**

Core problem

At any analysis time, we have:

- ▶ an **imperfect model forecast** (background)
- ▶ **sparse, noisy observations**

These must be combined optimally.

Data assimilation

Data assimilation provides a statistically consistent framework to merge:

model information + observations

The result is the **analysis state**:

- ▶ best estimate of the atmospheric state
- ▶ starting point for forecasts

Data assimilation is the information bottleneck of NWP.

Everything that follows — forecasts, warnings, applications — depends on it.

Classical Data Assimilation: The Analysis Cycle

The classical DA cycle

Operational NWP systems run a repeating assimilation cycle :

1. Start from a background state x_b
2. Assimilate observations y
3. Compute an analysis x_a
4. Run the numerical model forward
5. Use the forecast as next background

This cycle is repeated every few hours as new observations become available.

The analysis step is the only place where observations enter.

Established DA methods

Several algorithmic families are used in practice:

- ▶ **3D-Var / 4D-Var**
 - ▶ variational optimization
 - ▶ adjoint-based
- ▶ **Ensemble Kalman Filters (EnKF)**
 - ▶ flow-dependent uncertainty
 - ▶ ensemble statistics
- ▶ **Particle Filters**
 - ▶ fully Bayesian
 - ▶ now also high-dimensional!

All methods aim at the same goal : a statistically optimal analysis.

Two AI Paths for Using Observations

Path 1: AI-based forecasting

Observations are used directly inside neural networks that produce forecasts.

Typical characteristics:

- ▶ observations as additional inputs
- ▶ sometimes no explicit model state
- ▶ learning focuses on **prediction skill**

This approach bypasses the classical analysis concept.

Observations → forecast directly

Path 2: AI-based data assimilation

Observations are used to compute an analysis state, not a forecast.

Key properties:

- ▶ preserves the analysis–forecast separation
- ▶ consistent with Bayesian DA theory
- ▶ AI replaces the analysis algorithm, not the model

Observations → analysis → forecast

This is the conceptual space of **AI-Var**.

Why Bring AI into Data Assimilation?

Limits of classical DA

State-of-the-art data assimilation systems are:

- ▶ computationally expensive
- ▶ difficult to scale to higher resolution
- ▶ reliant on adjoint models

Operational challenges include:

- ▶ complex model development and maintenance
- ▶ long wall-clock times
- ▶ limited flexibility for new observation types

DA is often the most expensive component of NWP.

What AI can offer

Modern neural networks provide:

- ▶ fast inference once trained
- ▶ automatic differentiation
- ▶ flexible nonlinear mappings

Potential benefits for DA:

- ▶ orders-of-magnitude speedup
- ▶ end-to-end differentiability
- ▶ easier adaptation to new data streams

Goal: replace the solver, not the statistics.

From Variational Data Assimilation to AI-Var

Variational DA: main Idea

The analysis is obtained by minimizing:

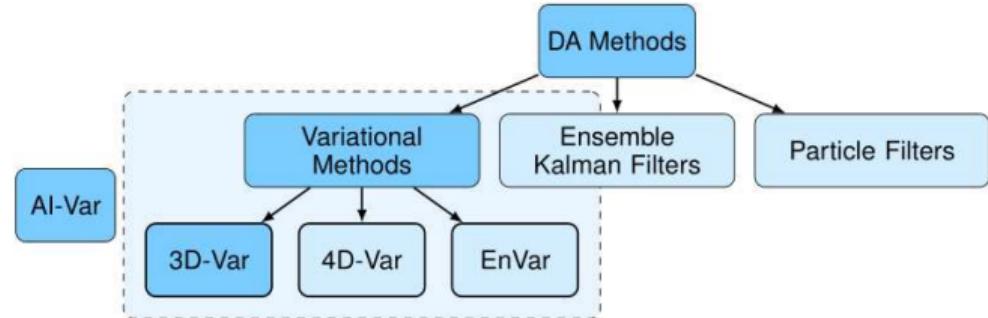
$$J(x) = \frac{1}{2}(x - x_b)^T B^{-1}(x - x_b) + \frac{1}{2}(y - H(x))^T$$

Interpretation

- ▶ first term: background constraint
- ▶ second term: observation constraint

Classical DA = iterative numerical minimization.

AI-VAR is the **AI verion** of **3D-Var, 4D-Var or En-Var** depending on how exactly the minimizer is formulated.



AI-Var sits inside the **variational DA family**, alongside 3D-Var, 4D-Var, and EnVar.

Core Idea of AI-Var

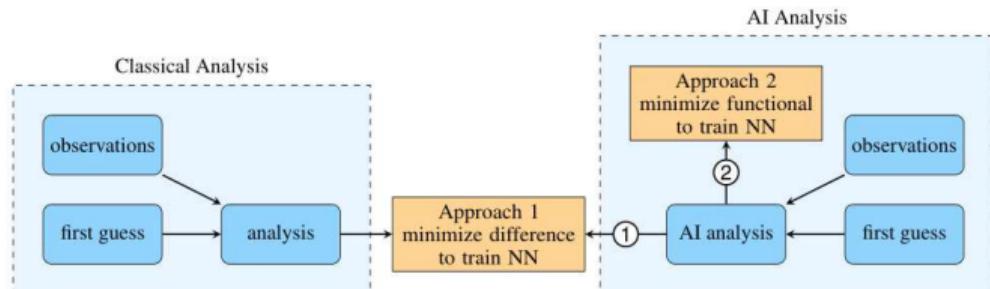
Classical variational DA

- ▶ iterative minimization of $J(x)$
- ▶ requires adjoints and solvers
- ▶ expensive and sequential

Replace the minimization algorithm.

AI-Var

- ▶ neural network approximates the minimizer
- ▶ outputs analysis x_a directly
- ▶ trained using the same cost function



Classical DA workflow (left) versus AI-based inference of the analysis (right).

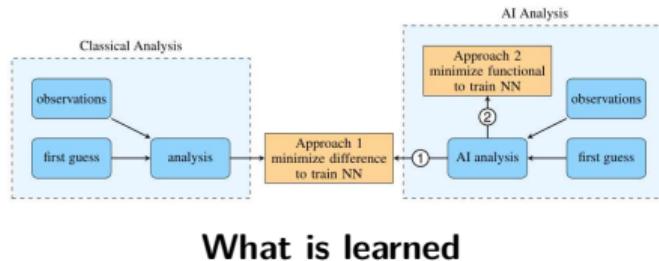
Paradigm shift

- ▶ DA becomes inference, not optimization
- ▶ milliseconds instead of iterations
- ▶ same statistics, different machinery
- ▶ flexible inflow of further information

AI-Var: Architecture, Loss, and Learning

Architecture

- ▶ inputs: background x_b and observations y
- ▶ neural network outputs analysis \hat{x}_a



What is learned

- ▶ mapping $(x_b, y) \rightarrow x_a$

$$L = (\hat{x}_a - x_b)^T B^{-1} (\hat{x}_a - x_b) + (H(\hat{x}_a) - y)^T R^{-1} (H(\hat{x}_a) - y)$$

What is not learned

- ▶ B^{-1} and R^{-1} weight uncertainties
- ▶ observation operator H is inside the loss

- ▶ no reanalysis targets
- ▶ no iterative solvers
- ▶ no adjoint code

Statistics and physics are embedded in the loss.

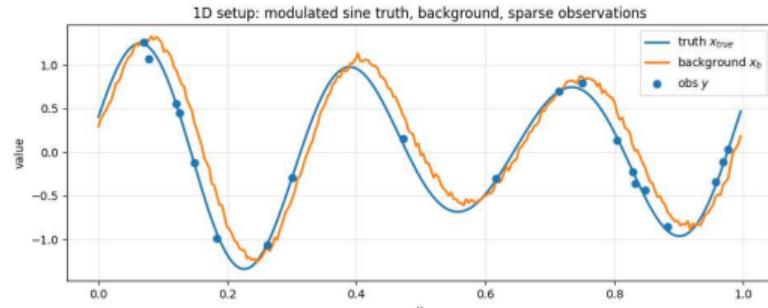
Optimization is replaced by inference.

1D Data Assimilation Setup

Toy problem

We consider a 1D state on a fixed grid, periodic (!).

- ▶ truth: modulated sine function
- ▶ background x_b : shifted, smoothed, biased
- ▶ observations y : sparse, noisy point samples



Truth x_{true} , background x_b , and sparse observations y .

All ingredients are fully controlled :

- ▶ known truth
- ▶ known error statistics
- ▶ explicit observation locations

Ideal testbed to explore data assimilation.

Classical 3D-Var in 1D: Mathematics Made Explicit

3D-Var cost function

The analysis x_a minimizes $J(x)$ from above.

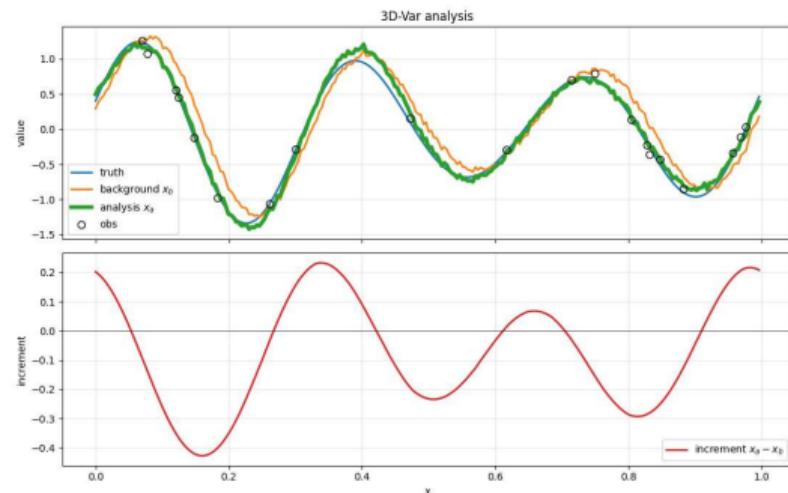
Linear case (this setup)

- ▶ H : point-sampling operator
- ▶ B : Gaussian covariance
- ▶ $R = \sigma_o^2 I$

Closed-form solution

$$x_a = x_b + BH^T(HBH^T + R)^{-1}(y - Hx_b)$$

This is the optimal Bayesian estimate.



Top: state space — truth, background, analysis.

Bottom: analysis increment

$$\delta x = x_a - x_b$$

Information from sparse observations is spread by B .

AI-Var in 1D: From Mathematics to Code

Neural increment model

The network predicts the analysis increment :

$$\delta x_\theta = \mathcal{N}_\theta(x_b, y)$$

PyTorch implementation (kept simple)

```
class IncrementMLP(nn.Module):
    def __init__(self, n):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3*n, 256),
            nn.Tanh(),
            nn.Linear(256, 256),
            nn.Tanh(),
            nn.Linear(256, n))

    def forward(self, inp):
        return self.net(inp)
```

Input vector:

$$\text{inp} = [x_b, y_{\text{grid}}, \text{mask}]$$

Variational loss in code

The 3D-Var cost is used directly :

```
def J_3dvar(delta_x):
    x = xb + delta_x
    innov = y - H @ x

    Jb = 0.5 * (delta_x @ B_inv @ delta_x)
    Jo = 0.5 * (innov @ R_inv @ innov)
    return Jb + Jo
```

No analysis data. No solver. Only the variational objective.

AI-Var in 1D: Training Loop with Background & Observations

Neural input construction

The network receives both background and observations:

```
# observations mapped to grid
y_grid = torch.zeros(n)
mask   = torch.zeros(n)

y_grid[obs_idx] = y
mask[obs_idx]   = 1.0

# NN input
inp = torch.cat([xb_t, y_grid, mask])

 $\delta x_\theta = \mathcal{N}_\theta(x_b, y)$ 
```

The solver is replaced by learning.

Training loop

```
for ep in range(n_epochs):
    optimizer.zero_grad()

    delta_x = model(inp)

    loss, Jb, Jo = J_3dvar(delta_x)

    loss.backward()
    optimizer.step()
```

Key points

- ▶ loss = classical 3D-Var functional
- ▶ no analysis targets
- ▶ gradients pass through B^{-1} and H

1D Result: AI-Var Analysis vs 3D-Var

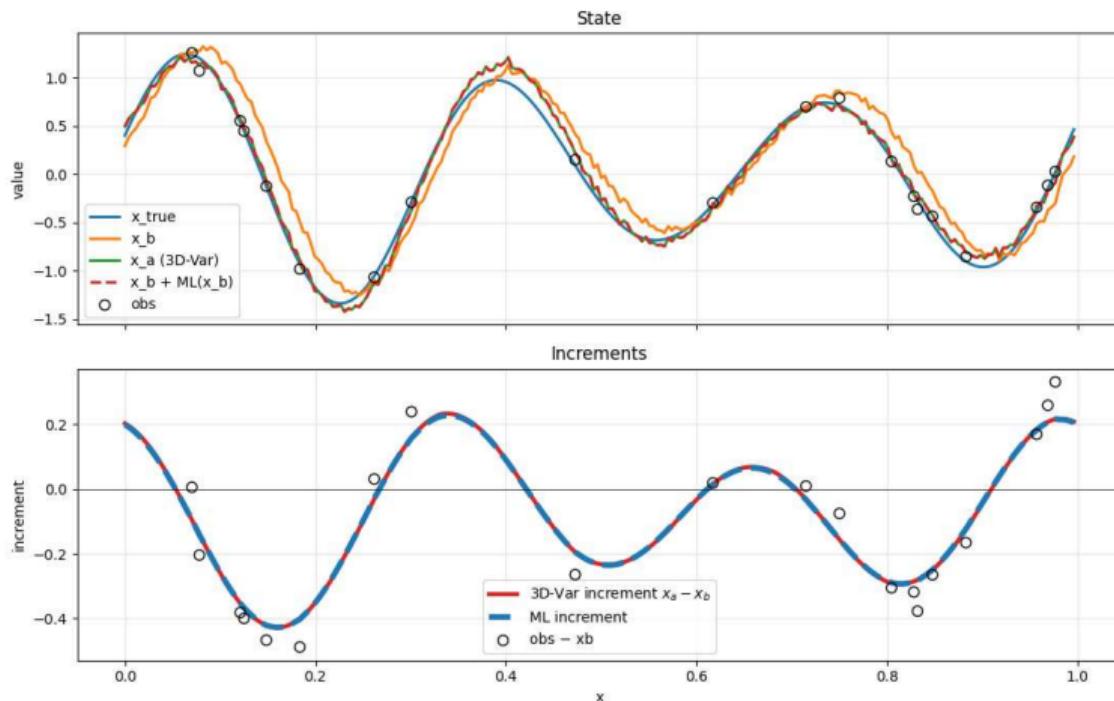
What is shown

- ▶ true state x_{true}
- ▶ background x_b
- ▶ 3D-Var analysis x_a
- ▶ AI-Var analysis $x_b + \delta x_{ML}$

Bottom panel

- ▶ classical 3D-Var increment
- ▶ learned AI-Var increment
- ▶ observation increments at obs points

AI-Var reproduces the
variational update.



AI-Var in 1D: Training on Many Cases

From single case to ensemble training

Instead of one fixed (x_b, y) pair, we train on many randomly generated cases .

Each training sample contains:

- ▶ a new truth x_{true}
- ▶ a new background x_b
- ▶ new observation locations and values y

What stays fixed

- ▶ background covariance B
- ▶ observation error R
- ▶ variational cost J

The network learns a general DA operator.

Training logic (conceptual)

```
for sample in training_set:  
    xb, y = sample  
    inp = build_input(xb, y)  
  
    delta_x = model(inp)  
    loss = J_3dvar(delta_x)  
  
    loss.backward()  
    optimizer.step()
```

Same loss, many realizations.

1D Generalization: Many Unseen Test Cases

Test phase

The trained AI-Var network is applied to previously unseen cases.

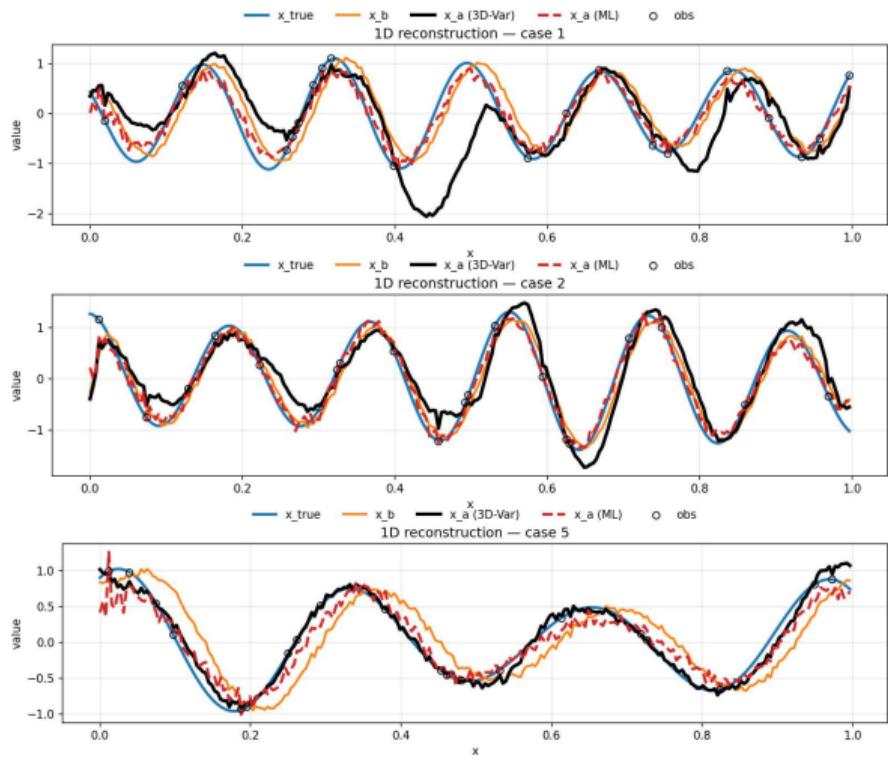
For each case:

- ▶ different truth
- ▶ different background
- ▶ different observations

Observation

- ▶ consistent increments
- ▶ smooth, physical updates
- ▶ no retraining required

One network, many analyses.



2D Data Assimilation Setup

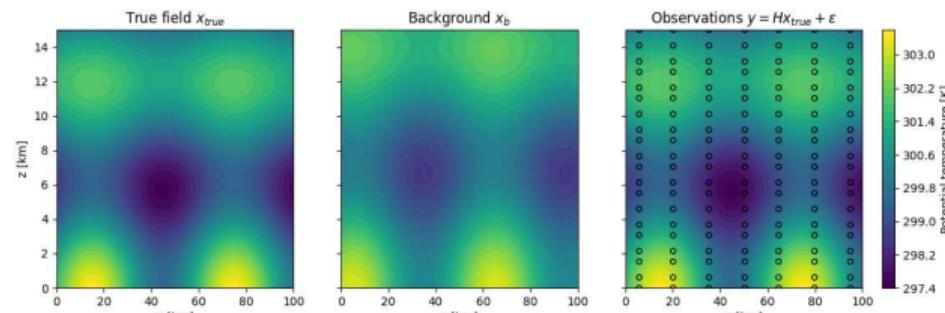
2D atmospheric toy problem

- ▶ horizontal–vertical grid
- ▶ structured background errors
- ▶ sparse column observations

State variables

- ▶ truth $x_{true}(x, z)$
- ▶ background $x_b(x, z)$
- ▶ analysis $x_a(x, z)$

This already resembles NWP geometry.

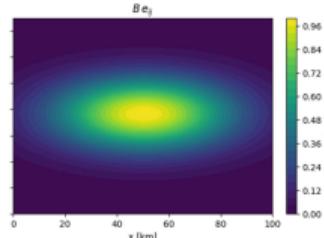


2D setup in code

```
# 2D grid (x,z)
nx, nz = 120, 40
x = np.linspace(0, Lx, nx)
z = np.linspace(0, Lz, nz)

# background covariance (separable)
B = Bx * Bz

# sparse vertical profiles
H : (n_obs x n_state)
```



B matrix in 2d, Be_j

Same DA ingredients as in 1D, now on a 2D grid.

2D AI-Var: Explicit Spectral B^{-1} in Flattened Control Space

2D control vector

The 2D field is represented as a 1D control vector :

$$x \in \mathbb{R}^n, \quad n = n_x n_z$$

```
nz, nx = xb.shape
n = nz * nx

X_flat = X.reshape(-1)
Z_flat = Z.reshape(-1)
```

Full Gaussian background covariance

```
dx2 = (X_flat[:,None]-X_flat[None,:])**2
dz2 = (Z_flat[:,None]-Z_flat[None,:])**2

B = sigma_b**2 * np.exp(
    -0.5*(dx2/Lx**2 + dz2/Lz**2)
)
B = 0.5*(B + B.T)
```

Spectral regularization

```
lam, U = np.linalg.eigh(B)

lam_floor = alpha * lam.max()
lam_reg   = np.maximum(lam, lam_floor)

Breg_inv = (U * (1.0/lam_reg)) @ U.T
Breg_inv = 0.5*(Breg_inv + Breg_inv.T)
```

Background term in the loss

$$J_b(\delta x) = \frac{1}{2} \delta x^T B^{-1} \delta x$$

```
Jb = 0.5 * dx @ Breg_inv @ dx
```

Key point: B^{-1} is built explicitly in this tutorial.

2D AI-Var (Tutorial Code): Flattened Control + Obs Encoding + 3D-Var Loss

Inputs: background + obs on grid + mask

```
# flatten 2D -> 1D control
nz, nx = xb.shape
n = nz * nx
xb_t = torch.tensor(xb.reshape(-1), dtype=dtype, device=device)

# obs indices (iz-major, ix-minor)
obs_indices = iz_idx * nx + ix_idx
y_vec = y_field.reshape(-1)[obs_indices]
y_t    = torch.tensor(y_vec, dtype=dtype, device=device)

# obs on grid + mask (length n)
y_grid = np.zeros(n); mask = np.zeros(n)
y_grid[obs_indices] = y_vec
mask[obs_indices]   = 1.0

inp_t = torch.cat([xb_t,
                  torch.tensor(y_grid, dtype=dtype, device=device),
                  torch.tensor(mask, dtype=dtype, device=device)], dim=0)
```

Input is $[x_b, y_{\text{grid}}, \text{mask}]$ in control space.

Increment MLP + variational loss

```
class IncrementMLP(nn.Module):
    def __init__(self, n):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3*n, 256), nn.Tanh(),
            nn.Linear(256, 256), nn.Tanh(),
            nn.Linear(256, n),
        )

    def quadform(A, v): return torch.dot(v, A @ v)

    def J_3dvar(dx):
        x = xb_t + dx
        innov = y_t - (H_t @ x)
        Jb = 0.5 * quadform(B_inv_t, dx)
        Jo = 0.5 * quadform(R_inv_t, innov)
        return Jb + Jo
```

Exactly the classical 3D-Var objective,
now differentiable.

2D Result: Variational Reference vs Learned Analysis

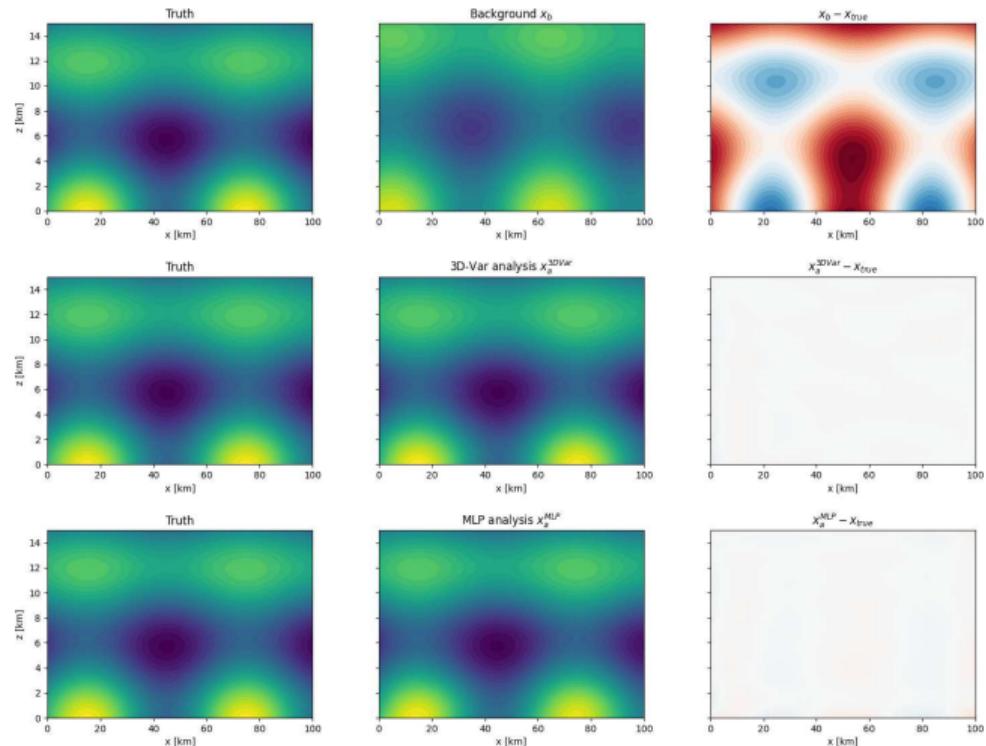
What is compared

- ▶ background x_b
- ▶ 3D-Var analysis x_a
- ▶ AI-Var analysis \hat{x}_a

Key observation

- ▶ smooth spatial increments
- ▶ correct information spreading
- ▶ close to variational solution

Learned analysis = inference of the minimizer.



AI Particle Filter (Gaussian Mixture): Core Idea

Filtering problem

Sequential posterior:

$$p(x_n | y_{1:n})$$

Represent the distribution by particles

$$X_n = \{x_n^{(i)}\}_{i=1}^N$$

AI Particle Filter (this tutorial)

Forecast ensemble X_n^b is transformed into analysis ensemble:

$$X_n^a = \mathcal{N}_\theta(X_n^b, y_n)$$

Gaussian mixture view

Both prior and posterior are approximated by mixtures:

$$q^b(x) \approx \frac{1}{N} \sum_i \mathcal{N}(x|x^{b,(i)}, \Sigma), \quad q^a(x) \approx \frac{1}{N} \sum_i \mathcal{N}(x|x^{a,(i)}, \Sigma)$$

Model problem: Lorenz-63

- ▶ state $x = (x_1, x_2, x_3) \in \mathbb{R}^3$
- ▶ partial observations $y = (x_1, x_2) +$
noise
- ▶ wrong forecast model (intentional)

Key message

Instead of resampling / MCMC moves, we learn a distribution transform that fits the ensemble to the posterior.

Network output is a posterior particle cloud.

Neural Particle Update: DeepSets (Permutation Invariance)

Particle set input

Forecast ensemble is an unordered set:

$$X^b = \{x_i^b\}_{i=1}^N$$

Update must be **permutation invariant**:

$$\mathcal{N}_\theta(\pi X^b, y) = \pi \mathcal{N}_\theta(X^b, y)$$

```
class ParticleUpdateNN(nn.Module):
    def forward(self, Xb, y):
        N = Xb.shape[0]
        y_rep = y.expand(N, -1)

        emb = phi(torch.cat([Xb, y_rep], 1))
        pooled = emb.mean(0, keepdim=True)
        ctx = rho(pooled).expand(N, -1)

        dX = psi(torch.cat([Xb, ctx, y_rep], 1))
        return Xb + dX
```

DeepSets structure

Output: updated ensemble X^a .

$$\phi(x_i, y) \rightarrow \text{mean pool} \rightarrow \rho(\cdot) \rightarrow \psi(x_i, \text{context}, y)$$

**Each particle sees local state + global context
+ obs.**

Training Loss: Fit a Gaussian-Mixture Posterior

Posterior fitting objective

Train the particle update network such that the analysis ensemble represents the posterior:

$$q^a(x) \approx p(x | y)$$

Gaussian mixture model

Particles define a mixture density:

$$q(x | X) = \frac{1}{N} \sum_{i=1}^N \mathcal{N}(x | x^{(i)}, \Sigma)$$

Loss = likelihood + KL fit

$$L = \underbrace{-\log \left(\frac{1}{N} \sum_i p(y | x_i^a) \right)}_{L_{\text{obs}}} + \lambda_{\text{bg}} \underbrace{\text{KL}\left(q_{\text{target}}(\cdot) \| q^a(\cdot) \right)}_{L_{\text{GM}}}$$

```
# evaluation points (single Kalman update)
Z = kalman_eval_points(Xb, y)

# target posterior weights on Z
log_t = log_mix(Z|Xb) + log_like(y|Z)
w_t = softmax(log_t)

# model mixture log-density on Z
log_m = log_mix(Z|Xa)

# mixture KL term
L_GM = sum_z w_t(z) * (log w_t(z))
- log softmax(log_m(z)))
loss = L_obs + lambda_bg * L_GM
```

Target uses prior mixture + obs likelihood.

Learn the posterior distribution, not only the mean.

Gaussian-Mixture PF: Comparing Two Distributions (Math)

1) Ensembles as Gaussian mixtures

Forecast particles $X^b = \{x_i^b\}_{i=1}^N$ and analysis particles $X^a = \{x_i^a\}_{i=1}^N$ define mixture densities, e.g.:

$$q_\theta^a(x) = \frac{1}{N} \sum_{i=1}^N \mathcal{N}(x | x_i^a, \Sigma)$$

2) Discretize comparison

Choose evaluation points $Z = \{z_k\}_{k=1}^K$ (from a Kalman-type proposal step in the notebook).

Target posterior on Z

$$w_k^* = \frac{q^b(z_k) p(y | z_k)}{\sum_{\ell=1}^K q^b(z_\ell) p(y | z_\ell)}$$

Prior mixture + obs likelihood \Rightarrow posterior weights.

3) Model distribution on Z

Evaluate the analysis mixture on the same points and normalize:

$$\pi_{\theta,k} = \frac{q_\theta^a(z_k)}{\sum_{\ell=1}^K q_\theta^a(z_\ell)} = \text{softmax}(\log q_\theta^a(z_k))$$

4) Fit posterior mass distribution

Training minimizes the discrete KL divergence:

$$L_{\text{GM}} = \text{KL}(w^* \parallel \pi_\theta) = \sum_{k=1}^K w_k^* \log \frac{w_k^*}{\pi_{\theta,k}}$$

Network learns to move particles toward posterior mass.

Ensemble Geometry: Prior → Analysis in a 2D Slice

What the plot shows

2D slice of the state space
(e.g. x_1-x_2 plane).

Prior (forecast ensemble)

- ▶ particles X^b (cloud)
- ▶ prior mean \bar{x}^b

Observation / truth

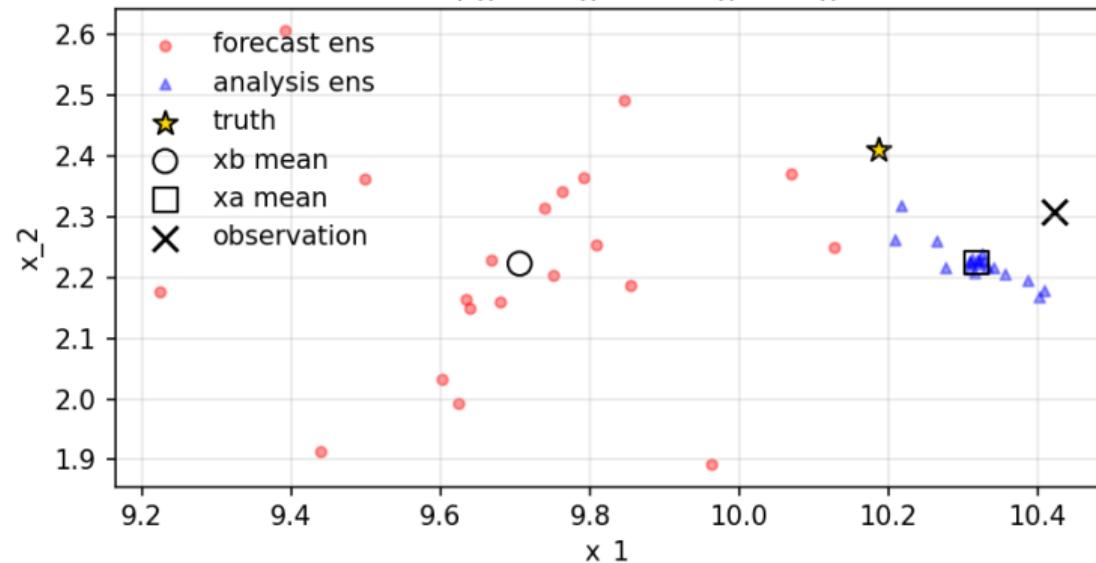
- ▶ observation y (marker)
- ▶ truth x_{true} (marker)

Analysis ensemble

- ▶ updated particles X^a
- ▶ analysis mean \bar{x}^a

Ensemble scatter in x_1-x_2 plane

step n=22 | $||x_b-x_t||=0.59$, $||x_a-x_t||=0.35$



Prior ensemble (forecast) and analysis ensemble after the learned update.

Ablation: Why the Background / Posterior-Fit Term Matters

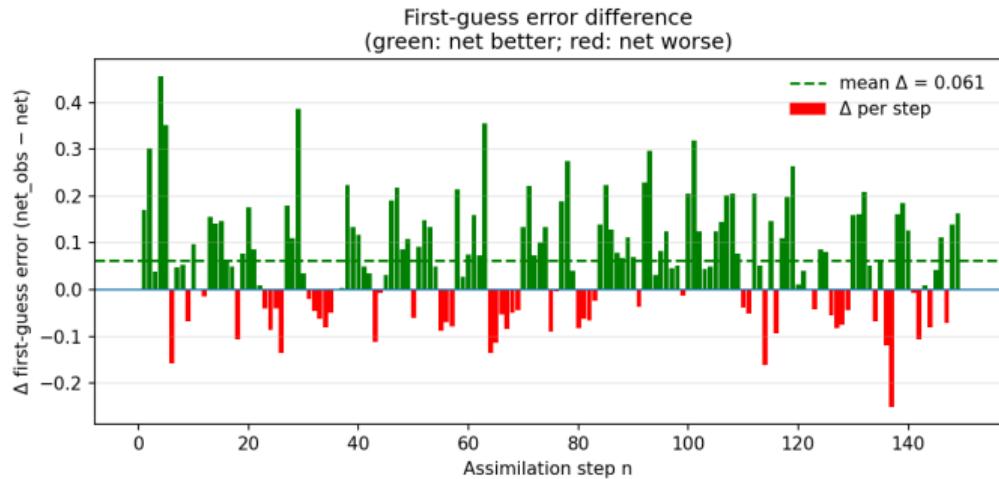
Two trained networks

- ▶ **net**: full AI-PF loss
- ▶ **net_obs**: obs term only

Metric shown

Difference of first-guess error:

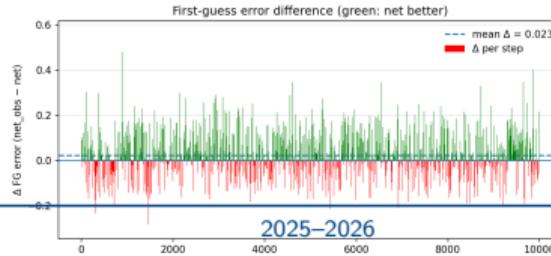
$$\Delta = \text{FG}_{\text{err}}(\text{net_obs}) - \text{FG}_{\text{err}}(\text{net})$$



Interpretation

- ▶ $\Delta > 0$ (green) : net is better
- ▶ $\Delta < 0$ (red): net_{obs} is better

Evaluation over 10 000 assimilation cycles after short training.



Summary: Two AI Paths for Data Assimilation

AI-Var (Keller & Potthast)

What it learns:

$$(x_b, y) \mapsto x_a$$

- ▶ neural network approximates the minimizer
- ▶ trained by variational cost $J(x)$
- ▶ output = one deterministic analysis field

Strengths

- ▶ stable, structured increments
- ▶ scalable in dimension (1D → 2D demo)
- ▶ direct link to operational 3D/4D-Var

AI Particle Filter (Gaussian mixture)

What it learns:

$$(X^b, y) \mapsto X^a$$

- ▶ neural update transforms particle cloud
- ▶ trained to fit posterior mass distribution
- ▶ output = analysis distribution (ensemble)

Strengths

- ▶ non-Gaussian posteriors (multi-modal)
- ▶ distribution-aware filtering
- ▶ background-term ablation shows skill gain

Take-home: AI-Var learns the analysis; AI-PF learns a posterior distribution.

Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 
playing  with
Python  and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00					Joint Dinner

Lecture 19: AI and Physics and Data

Core question

Given a dynamical system, what can machine learning do?

- ▶ Solve known equations (ODE/PDE)
- ▶ Discover unknown governing laws from observations
- ▶ Emulate complex dynamics as a surrogate model

Unifying viewpoint

All methods impose (explicitly or implicitly) a constraint:

state evolution must be consistent with $\dot{x} = f(x)$.

How can different ML approaches enforce this consistency?

Three routes in this lecture

1. PINNs: physics drives training
2. SINDy: sparse laws from data
3. Neural RHS learning:
black-box emulation

Key trade-offs

- ▶ Accuracy vs. interpretability
- ▶ Data-efficiency vs. flexibility
- ▶ Stability / extrapolation vs. expressiveness

Message:

same goal (dynamics), different framework.

Lecture Roadmap

Part I — Physics-Informed Neural Networks

- ▶ learn **solutions** of known equations
- ▶ training uses **ODE/PDE residuals** + anchor conditions
- ▶ representation matters for extrapolation

Part III — Learning the Force Term

- ▶ learn the unknown RHS / forcing from data:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) + \mathbf{g}_\theta(\mathbf{x})$$

- ▶ **hybrid modeling:** known physics + learned closure
- ▶ stable rollout by integrating the learned system

Part II — Discovering equations from data

- ▶ **SINDy:** sparse regression on a function library
- ▶ **Neural SINDy:** smooth NN trajectory \Rightarrow stable derivatives

Part IV — Causal Modeling with Neural Networks

- ▶ distinguish **correlation** vs. **cause**
- ▶ learn structural relations (SCMs) as NN modules
- ▶ intervene and test counterfactual predictions:

$$do(X = x) \quad \Rightarrow \quad p(Y | do(X = x))$$

A Minimal Physics-Informed Neural Network (PINN)

Problem setup

We consider a simple second-order ODE:

$$y''(x) + y(x) = 0$$

with boundary conditions

$$y(0) = 0, \quad y'(0) = 1.$$

The unique solution is

$$y(x) = \sin(x).$$

This example is deliberately **simple**, but already captures all essential PINN ingredients.

PINN idea

- ▶ Approximate $y(x)$ by a neural network $y_\theta(x)$
- ▶ **No training data** $y(x)$ are used
- ▶ Training is driven by **physics constraints**

What is enforced

- ▶ Differential equation via automatic differentiation
- ▶ Boundary conditions via penalty terms

The network learns the solution by minimizing violations of physics.

PINN Loss: Physics Instead of Data

ODE residual

Using automatic differentiation, we compute

$$y'_\theta(x), \quad y''_\theta(x).$$

The differential equation is enforced by minimizing

$$r(x) = y''_\theta(x) + y_\theta(x).$$

The corresponding loss term is

$$\mathcal{L}_{\text{ODE}} = \frac{1}{N} \sum_{i=1}^N (y''_\theta(x_i) + y_\theta(x_i))^2.$$

Collocation points x_i are sampled in the domain.

Boundary conditions

Boundary (anchor) constraints enforce uniqueness:

$$y_\theta(0) = 0, \quad y'_\theta(0) = 1.$$

This yields the boundary loss

$$\mathcal{L}_{\text{BC}} = (y_\theta(0))^2 + (y'_\theta(0) - 1)^2.$$

Total loss

The parameter λ controls the strength of the boundary conditions.

$$\mathcal{L} = \mathcal{L}_{\text{ODE}} + \lambda \mathcal{L}_{\text{BC}}.$$

No data term appears anywhere in the loss.

PINN Result: Naive Training, Extended Evaluation

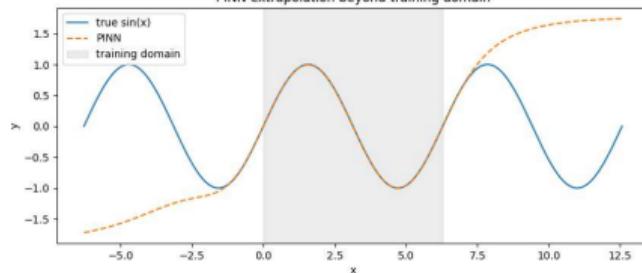
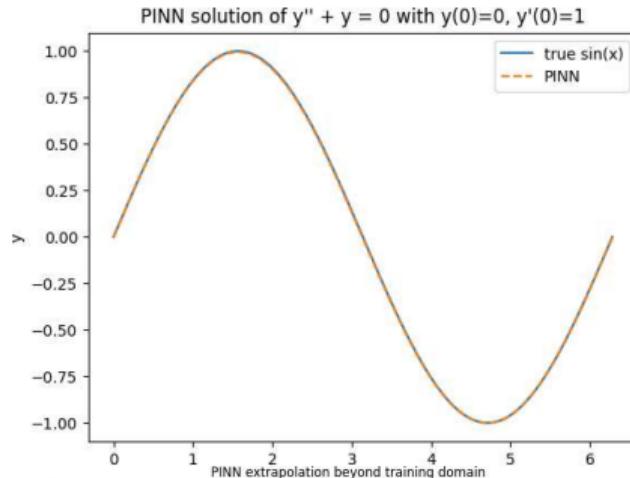
Training setup

- ▶ ODE residual enforced on $[0, 2\pi]$
- ▶ Boundary conditions at $x = 0$
- ▶ No data, no periodic constraints
- ▶ Standard MLP representation

Evaluation

- ▶ Solution evaluated on a larger domain
- ▶ Outside the region where physics was enforced

This tests extrapolation, not interpolation.



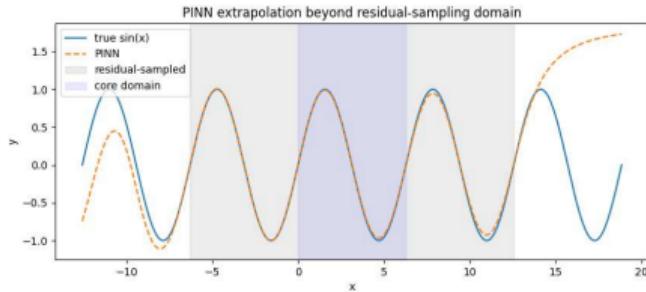
Improving Extrapolation by Wider Residual Sampling

Key modification

The PINN formulation is unchanged, but the ODE residual is enforced on a wider domain .

Training setup

- ▶ ODE residual sampled beyond $[0, 2\pi]$
- ▶ Boundary conditions still imposed at $x = 0$
- ▶ Same network architecture and loss terms



This already fixes many extrapolation problems for simple ODEs.

Effect

- ▶ Physics is enforced more globally
- ▶ Extrapolation becomes significantly more stable
- ▶ No change in representation or constraints

Improving Representation with Fourier Features

Motivation

Standard MLPs learn functions of x that are biased toward smooth, slowly varying behavior.

Oscillatory solutions, such as

$$y(x) = \sin(x),$$

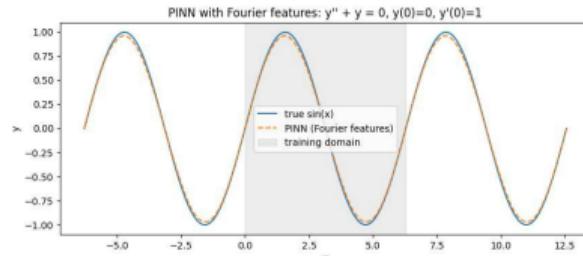
are therefore harder to represent and extrapolate.

Fourier feature idea

Instead of learning directly from x , we apply a fixed feature map:

$$x \mapsto (\sin(\omega_k x), \cos(\omega_k x))_{k=1}^m.$$

The neural network then learns a function of these periodic features.



Effect on the PINN

- ▶ Periodicity is easy to represent
- ▶ Long-range extrapolation improves
- ▶ Fewer parameters are needed

Interpretation

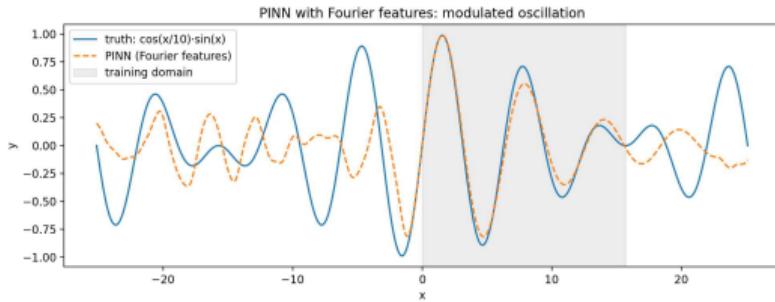
- ▶ Linear models become Fourier series fits
- ▶ Nonlinear MLPs allow mode interactions

This changes the representation, not the physics.

Result: Fourier-Feature PINN for a Modulated Oscillation

What changed compared to previous examples

- ▶ Governing ODE and anchor conditions unchanged
- ▶ Same PINN loss formulation
- ▶ Only the input representation is modified



The network uses a Fourier feature embedding.

Observed behavior

- ▶ Accurate solution on the training domain $[0, 5\pi]$
- ▶ Stable extrapolation far beyond the training region
- ▶ Correct phase and amplitude over many oscillations

Gray shading indicates the training domain. The solution is evaluated well beyond the region where the ODE residual was enforced.

Representation choice alone can control extrapolation quality.

SINDy: Sparse Identification of Nonlinear Dynamics

Problem setting

We observe a dynamical system

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t)), \quad \mathbf{x}(t) \in \mathbb{R}^n,$$

from time series data $\mathbf{x}(t_i)$.

Key assumption (sparsity)

The vector field can be written as a sparse combination of candidate functions:

$$\dot{\mathbf{x}}(t) \approx \Theta(\mathbf{x}(t)) \Xi,$$

where

- ▶ $\Theta(\mathbf{x})$ is a library of functions (e.g. $1, x, y, z, xy, xz, yz, \dots$),
- ▶ Ξ is a sparse coefficient matrix.

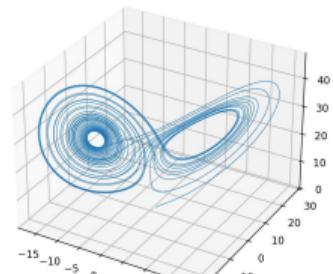
Identification

SINDy solves a sequence of

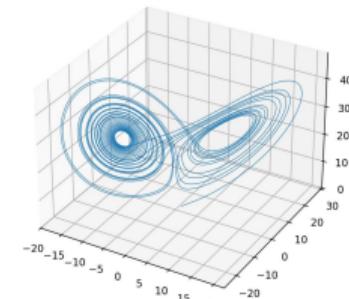
$$\min_{\Xi} \|\Theta(\mathbf{x})\Xi - \dot{\mathbf{x}}\|_2^2$$

with thresholding to eliminate small coefficients.

True Lorenz-63



SINDy-discovered Lorenz-63



Lorenz-63 example (noise-free).

Left: true trajectory. Right: trajectory from *SINDy*.

SINDy Sparse Regression of Nonlinear Dynamics: Lorenz–63 System

True governing equations

The Lorenz–63 system is defined by

$$\begin{aligned}\dot{x} &= \sigma(y - x), & \sigma &= 10, \\ \dot{y} &= x(\rho - z) - y, & \rho &= 28, \\ \dot{z} &= xy - \beta z, & \beta &= \frac{8}{3}.\end{aligned}$$

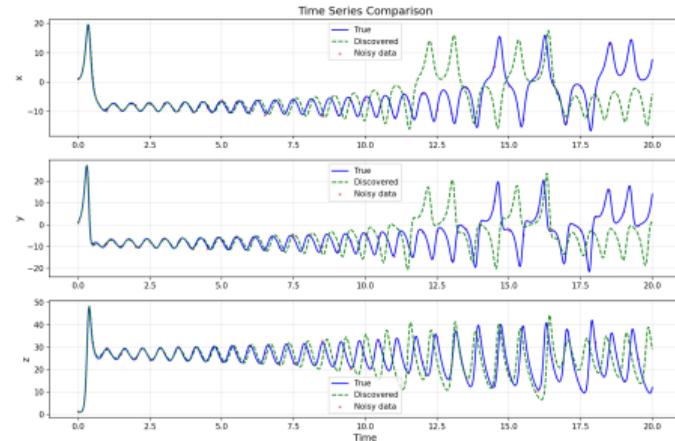
SINDy function library

SINDy assumes the dynamics can be written as a sparse linear combination of candidate functions:

$$\text{Theta}(x, y, z) = [1, x, y, z, x*y, x*z, y*z]$$

Only a few of these terms are retained in each equation after sparse regression.

Goal: recover the correct active terms and coefficients directly from time series data.



Time series of the Lorenz–63 state variables used as input for SINDy (blue). Numerical derivatives are estimated and matched against the candidate library during sparse regression. Discovered dynamics evolution in Green.

Neural SINDy: Using Neural Networks as Smooth Surrogates

Neural SINDy augments classical SINDy by introducing a neural network as a smooth surrogate for the observed trajectory:

$$\mathbf{x}(t) \longrightarrow \mathbf{x}_\theta(t).$$

The neural network is trained on noisy observations $\mathbf{x}_{\text{obs}}(t_i)$, but constrained to produce a *smooth time-continuous representation*.

Neural trajectory fitting

The network parameters θ are obtained by minimizing

$$\min_{\theta} \sum_i \|\mathbf{x}_\theta(t_i) - \mathbf{x}_{\text{obs}}(t_i)\|^2 + \alpha \int \left\| \frac{d}{dt} \mathbf{x}_\theta(t) \right\|^2 dt.$$

A regularization term penalizes rapid temporal variations and suppresses noise amplification.

Derivative estimation via autograd

Once trained, time derivatives are computed analytically:

$$\dot{\mathbf{x}}_\theta(t) = \frac{d}{dt} \mathbf{x}_\theta(t),$$

using automatic differentiation.

Sparse discovery step

The smoothed trajectory and its derivatives are then passed to the *unchanged* SINDy pipeline:

$$\dot{\mathbf{x}}_\theta(t) \approx \Theta(\mathbf{x}_\theta(t)) \Xi, \quad \Xi \text{ sparse.}$$

Here:

Neural networks are *not* used to represent the dynamics, but only to stabilize derivative estimation prior to sparse regression.

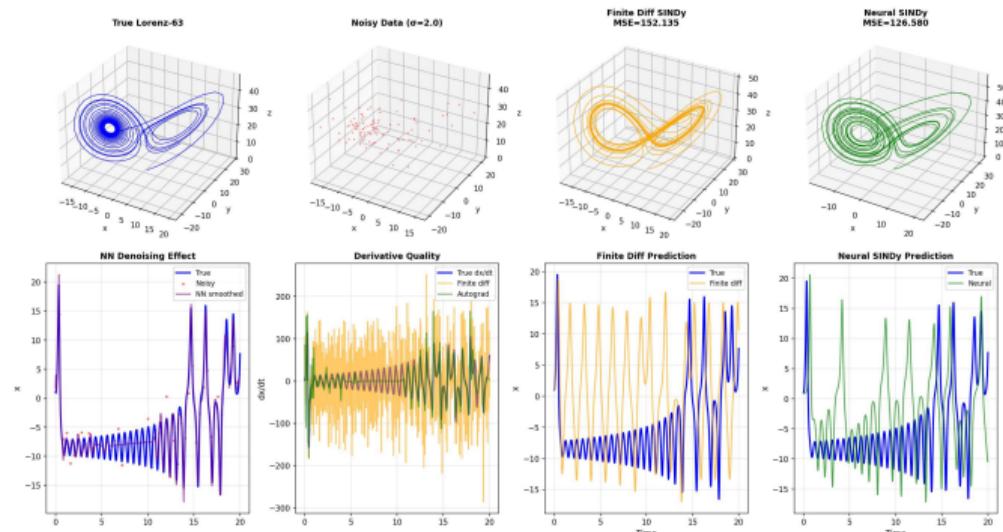
Neural SINDy: Results on Noisy Lorenz–63 Data

Experimental setup

- ▶ Lorenz–63 system
- ▶ Strong additive noise on observations
- ▶ Identical SINDy library and sparsity settings

Comparison

- ▶ Classical SINDy:
finite-difference derivatives
- ▶ Neural SINDy: NN-smoothed
trajectory + autograd



Comparison of trajectories, derivatives, and predictions for classical vs. Neural SINDy on noisy data.

Outcome

- ▶ Neural SINDy recovers an improved equation structure

Alternative: Learning the Full RHS with Neural Networks

Black-box RHS learning

Instead of discovering equations, one may directly learn the vector field

$$\dot{\mathbf{x}} = \mathbf{f}_\theta(\mathbf{x}),$$

where \mathbf{f}_θ is a neural network.

The network is trained to match observed time derivatives or trajectories.

- ▶ Very flexible function class
- ▶ Can approximate complex, unknown dynamics
- ▶ Naturally handles noise with regularization

Neural SINDy: a different goal

Neural SINDy uses neural networks only to stabilize intermediate steps (denoising and differentiation), but still identifies

$$\dot{\mathbf{x}} \approx \Theta(\mathbf{x}) \Xi,$$

with a sparse, explicit structure.

Key distinction:

- ▶ Neural RHS learning → predictive black box
- ▶ Neural SINDy → *interpretable equations*

This distinction is critical in scientific modeling, where the goal is understanding, not only prediction.

Neural RHS Learning : Results on Lorenz-63

Experimental setup

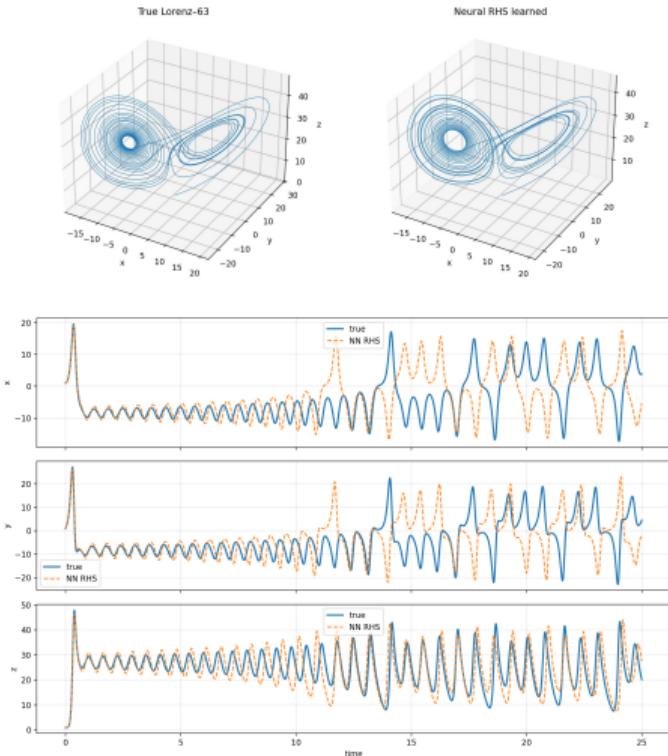
- ▶ Lorenz-63 system
- ▶ Neural network trained on $(\mathbf{x}, \dot{\mathbf{x}})$ pairs
- ▶ No sparsity or physics constraints

Observations

- ▶ Learned vector field \mathbf{f}_θ reproduces the chaotic attractor
- ▶ Short- to medium-term trajectories remain accurate, **climatology ok!**
- ▶ Long-term divergence is unavoidable due to chaos

Interpretation:

The neural network successfully emulates the dynamics, but the governing equations remain hidden.



Neural RHS Learning in State Space

Experimental setup

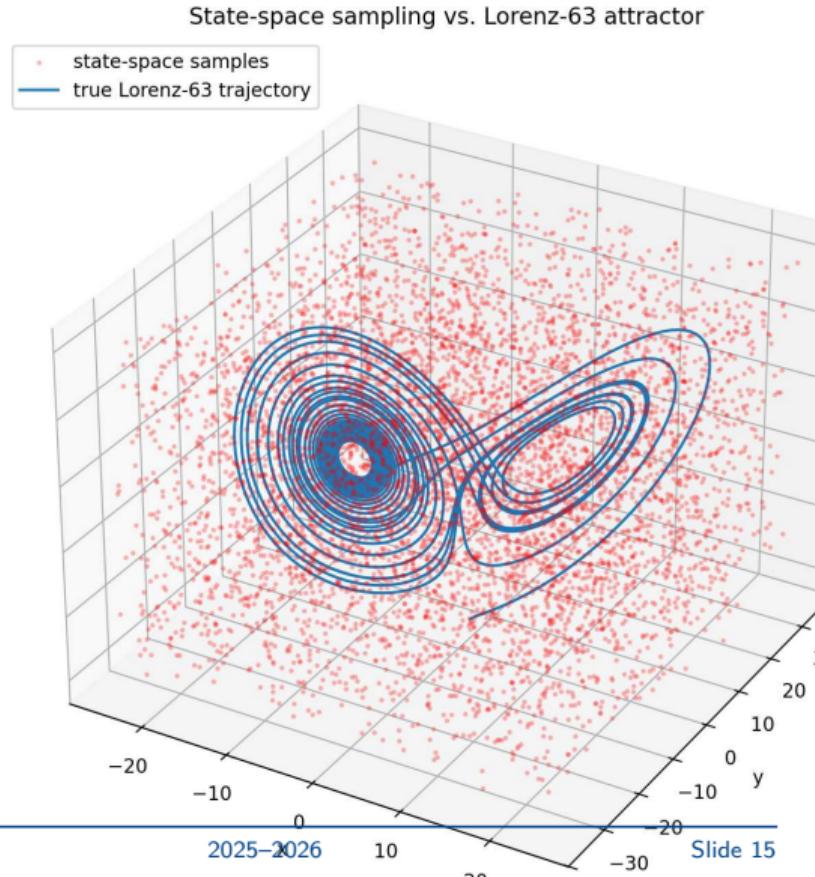
- ▶ Lorenz-63 vector field sampled in state space
- ▶ Random training points covering a 3D domain
- ▶ Neural network trained on $(\mathbf{x}, \dot{\mathbf{x}})$ pairs

Key difference to trajectory-based learning

- ▶ Training data no longer restricted to the attractor
- ▶ Vector field is constrained in a full region of state space
- ▶ Dynamics are learned as a global mapping

Outcome:

The learned neural vector field reproduces the Lorenz dynamics consistently from unseen initial conditions, not only along the original trajectory.



Learning Dynamical Systems: Goals and Methods

Three fundamentally different goals

▶ **Solve known equations**

- ▶ Given: governing ODE/PDE
- ▶ Task: compute the solution
- ▶ Method: PINNs

▶ **Discover equations from data**

- ▶ Given: time series observations
- ▶ Task: identify governing laws
- ▶ Method: SINDy / Neural SINDy

▶ **Emulate dynamics**

- ▶ Given: state–derivative pairs
- ▶ Task: reproduce system behavior
- ▶ Method: Neural RHS learning

What is learned in each case?

▶ PINNs

- ▶ learn the *solution* $x(t)$
- ▶ equations are assumed known

▶ SINDy / Neural SINDy

- ▶ learn *explicit equations*
- ▶ sparse, interpretable models

▶ Neural RHS learning

- ▶ learn a *vector field*
- ▶ accurate dynamics, but opaque

Key trade-off:

Interpretability \leftrightarrow Flexibility

Physics-Constrained Neural Emulators: 1D Periodic Advection

We consider 1D linear advection on a periodic domain:

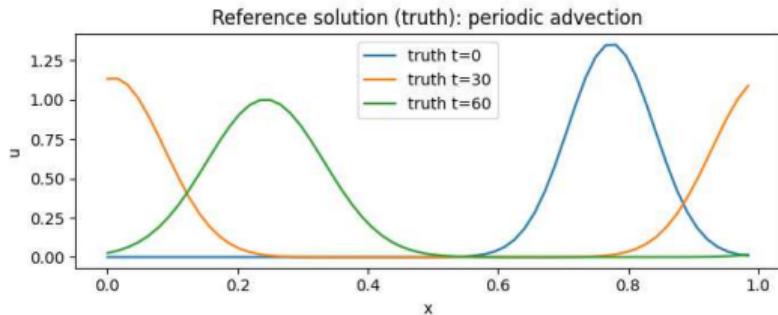
$$\partial_t u + c \partial_x u = 0, \quad x \in [0, 1], \quad u(0) = u(1).$$

Key physical properties:

- ▶ Pure translation on a ring
- ▶ Exact mass conservation
- ▶ Smooth initial conditions remain smooth

Learning task

- ▶ Learn a one-step map $u^n \mapsto u^{n+1}$
- ▶ Training data from a conservative upwind solver
- ▶ Compare different neural inductive biases



Reference solution (truth).

Gaussian bump advected periodically. Snapshots at $t = 0, 30, 60$ clearly show wrap-around and mass preservation.

Physics is simple — but violations are immediately visible.

CNN Emulator without Mass Conservation

Neural model

- ▶ Local 1D CNN with **circular padding**
- ▶ Learns one-step map $u^n \mapsto u^{n+1}$
- ▶ Trained by minimizing one-step MSE

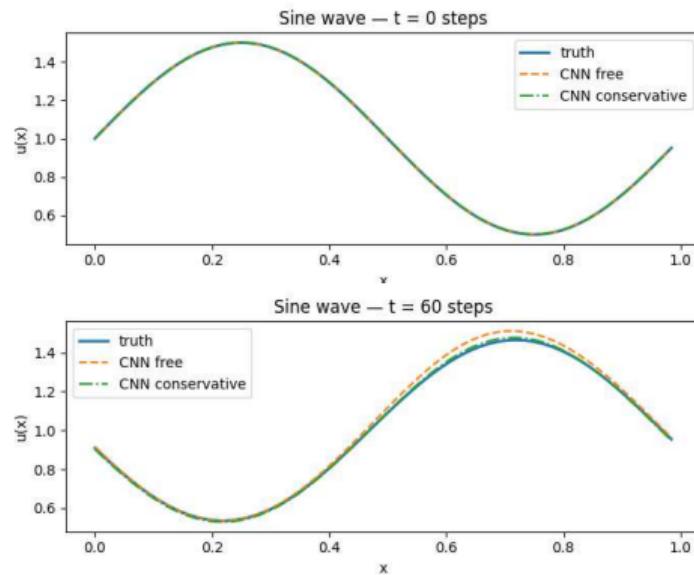
What is **not** enforced

- ▶ No mass conservation constraint
- ▶ No global invariant control

Observed behavior

- ▶ Low training loss
- ▶ Smooth short-term evolution
- ▶ **Gradual drift in total mass**

Locality alone is not enough to guarantee physical correctness.



CNN without conservation. Small amplitude and mass errors accumulate despite visually plausible transport.

CNN Emulator with Exact Mass Conservation

Conservative update

The CNN predicts a residual update

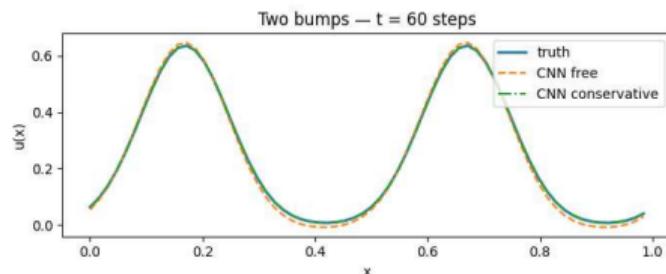
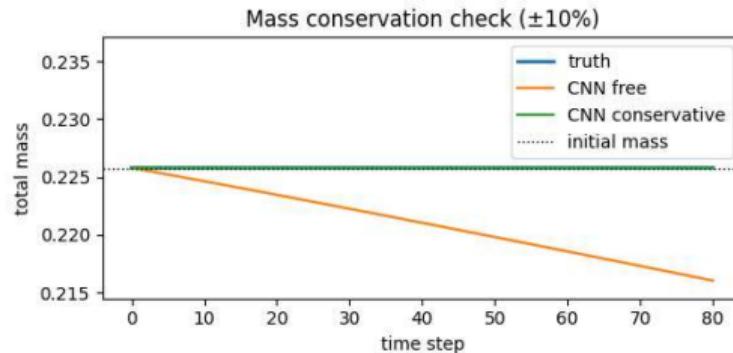
$$u^{n+1} = u^n + \Delta u,$$

with the constraint $\sum_i \Delta u_i = 0$.

- ▶ Mass conservation enforced by construction
- ▶ Same architecture and data as unconstrained CNN
- ▶ No penalty tuning required

Effect

- ▶ Slightly higher one-step loss
- ▶ Exact preservation of the global invariant
- ▶ Significantly improved long-term behavior



CNN with mass conservation. The global invariant is preserved exactly.

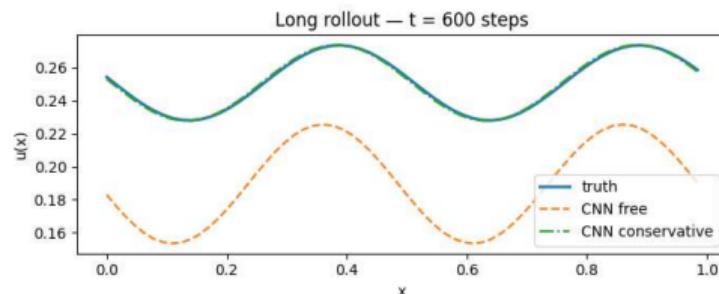
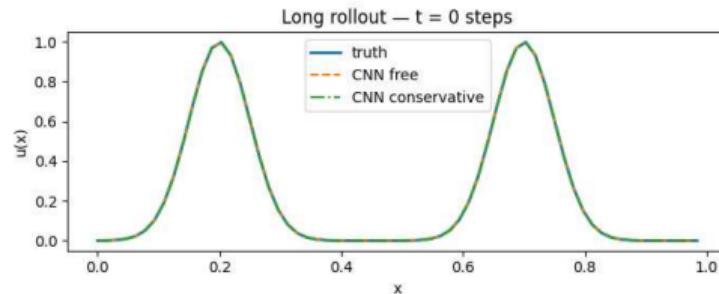
Long-Time Rollout: Accuracy vs Physical Validity

Long-time test

- ▶ Rollout over hundreds of time steps
- ▶ Same initial condition
- ▶ Compare truth, CNN-free, CNN-conservative

Key observation

- ▶ CNN-free: errors accumulate steadily
- ▶ CNN-conservative: structure remains coherent
- ▶ Physics constraints matter most far beyond training horizon



Short-term accuracy is not a proxy for long-term validity.

Only the conservative model maintains physically consistent transport.

Generalization: Advection of Unseen Shapes

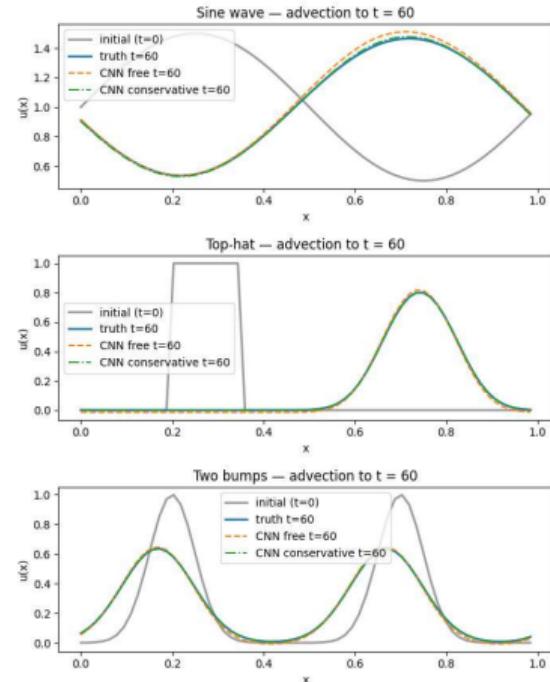
What was trained

- ▶ CNN emulator trained only on Gaussian initial conditions
- ▶ Local architecture with periodic padding
- ▶ Conservative variant enforces exact mass preservation

We test: Sine waves, Top-hat functions (discontinuous), Multiple separated bumps.

What we observe

- ▶ Correct translation on the periodic domain
- ▶ Shapes are transported without spurious creation or loss of mass
- ▶ Superpositions (multiple bumps) are handled consistently



Examples of **unseen initial conditions** advected by the CNN emulator.

Correlation vs Causality in Dynamical Systems

The trap

- ▶ Many variables in geosciences are strongly correlated
- ▶ But correlation alone cannot identify directionality
- ▶ Hidden drivers (confounders) can create spurious links

Causal question

$$P(T | P) \neq P(T | do(P))$$

Why time series are special

- ▶ Dynamics introduce time ordering
- ▶ Causal effects typically appear with lags
- ▶ Strong autocorrelation can mask cross-effects

Key idea

- ▶ Use lagged dependencies to separate:
 - ▶ self-dynamics (memory)
 - ▶ cross-variable influences

In Earth-system applications this matters because

- ▶ interventions (what-if) require causal structure
- ▶ attribution needs confounding control
- ▶ robust extrapolation benefits from causal mechanisms

Take-home message:

similar correlations do not imply same physics

Two Physical Processes Behind the Same Correlation

Scenario 1: direct causal coupling

- ▶ Pressure evolves under synoptic forcing
- ▶ Temperature responds via adiabatic processes

$$\frac{dP}{dt} = -\gamma_P(P - P_{eq}) + \beta_P F(t)$$

$$\frac{dT}{dt} = -\gamma_T(T - T_{eq}) + \alpha(P - P_{eq})$$

True causal link: $P \rightarrow T$

Scenario 2: common external forcing

- ▶ Air-mass advection drives both variables
- ▶ No direct physical coupling between P and T

$$\frac{dP}{dt} = -\gamma_P(P - P_{eq}) + \beta_P F(t)$$

$$\frac{dT}{dt} = -\gamma_T(T - T_{eq}) + \beta_T F(t)$$

No direct causal link: $P \not\rightarrow T$

Both scenarios can show similar $P-T$ correlation — but they are physically different.

Baseline: Classical Statistical Causal Analysis

Naive statistical analysis

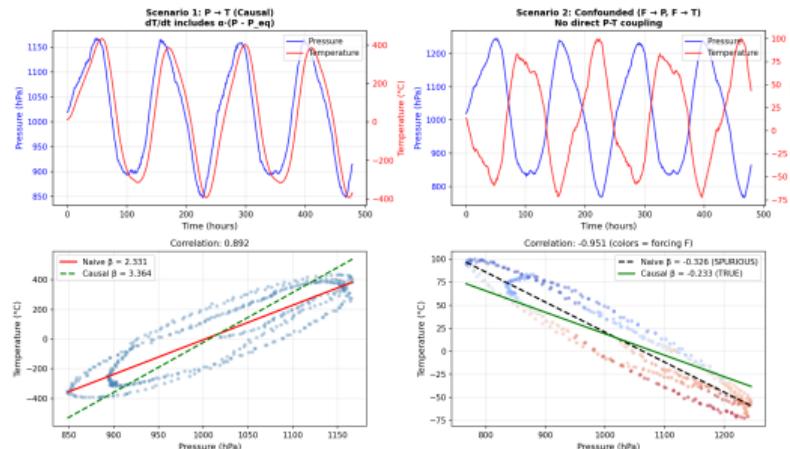
- ▶ correlation: $\text{corr}(P, T)$
- ▶ regression: $T \sim P$

Problem

- ▶ in Scenario 2, regression suggests $P \rightarrow T$
- ▶ but correlation is **spurious** (confounded by F)

Improvement (if forcing known)

- ▶ control for confounder:
$$T \sim P + F$$
- ▶ can recover causal effect if model is correct



Simulation diagnostics. Time series + scatter:
naive regression can be misleading when $F(t)$ induces common variability.

Limitation: Confounders must be known and included (strong prior assumptions).

Causal Discovery from Time Series: PCMCI (Classical Method)

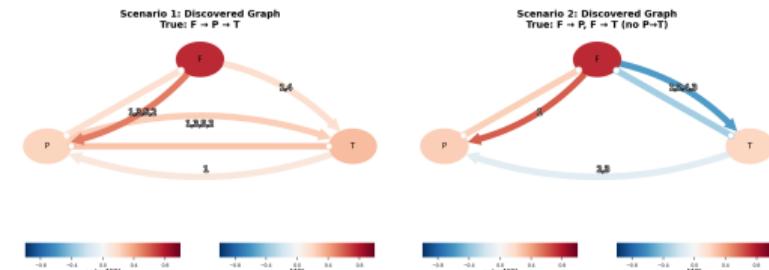
Goal

- ▶ learn the directed graph from multivariate time series
- ▶ distinguish:
 - ▶ direct causal links
 - ▶ common forcing and confounding

PCMCI in one line

- ▶ PC algorithm + conditional independence tests
- ▶ works with lagged dependencies
- ▶ reduces false links by conditioning on relevant parents

Important: PCMCI is statistical causal discovery, not neural.



Discovered graphs (PCMCI). Left: Scenario 1 matches the chain $F \rightarrow P \rightarrow T$. Right: Scenario 2 reveals no direct $P \rightarrow T$ link.

Message: same correlation \neq same causal structure.

Key capability

- ▶ Scenario 1: recover $F \rightarrow P \rightarrow T$
- ▶ Scenario 2: recover $F \rightarrow P$ and $F \rightarrow T$ only

Neural Causal Discovery: Scaling Ideas to Complex Systems

Motivation

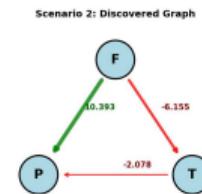
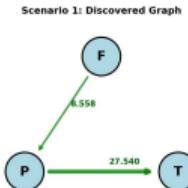
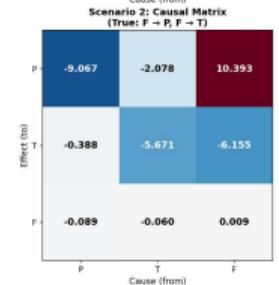
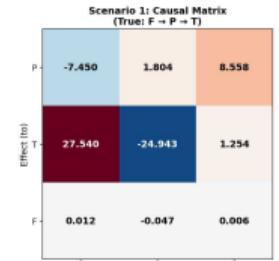
- ▶ Earth-system variables are high-dimensional
- ▶ dependencies are nonlinear and state-dependent
- ▶ autocorrelation dominates raw signals

Neural causal discovery idea

- ▶ first remove self-dependence (autocorrelation)
- ▶ learn cross-effects from innovations
- ▶ enforce sparsity \Rightarrow interpretable directed links

Innovation form: $x(t) - ax(t-1)$

Key message: AI helps to extend classical causal ideas to nonlinear, high-dimensional dynamics.



Python and AI/ML for Weather, Climate and Environmental Applications



Let us enjoy 🚀
playing 🤖 with
Python 🐍 and AI/ML!
 

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	Lab: Python Startup: Basics	Lab: Feed-forward and Graph NNs	Lab: Graph Learning with PyTorch	Lab: Containers and Reproducibility	Lab: Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	Lab: Work environments, Python everywhere	Lab: Simple Transformer and LLM Use	Lab: Agent Frameworks	Lab: CI/CD Pipelines	Lab: Graph-based Assimilation
12:45–13:30	Lunch Break				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	Lab: GRIB, NetCDF and Obs Visualisation	Lab: RAG Pipeline	Lab: DAWID Exploration	Lab: Anemoi Training Pipeline	Lab: Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	The AI Transformation	Learning from Observations Only
16:15–17:00	Lab: Torch Tensors and First Neural Net	Lab: Radar, SAT and Multimodal Data	Lab: MLflow Hands-on	Lab: How work style could change	Lab: ORIGEN and Open Discussion
17:00–20:00				Joint Dinner	

Lecture 20 — Obs-to-Obs Learning on a 2D Toy Atmosphere

Goal of this lecture

- ▶ Build a simple 2D dynamical system $\phi(x, z, t)$ with transport + diffusion + source
- ▶ Generate two observation types
 - ▶ **Radiosondes (RS):** sparse vertical profiles at a few columns
 - ▶ **Satellite (SAT):** integrated vertical weighted observations for every column
- ▶ Train a neural network to predict next-step observations

$$(y_t^{sat}, y_t^{rs}) \mapsto (y_{t+1}^{sat}, y_{t+1}^{rs})$$

- ▶ Reconstruct the full field at $t + 1$ by querying RS predictions everywhere

Key message: learn a forecast step in observation space, and still recover a state-like field.

Toy Dynamics: Advection–Diffusion with Source and Damping

We simulate a tracer/heating field $\phi(x, z, t)$ on a 2D domain:

$$(x, z) \in [0, L_x] \times [0, L_z].$$

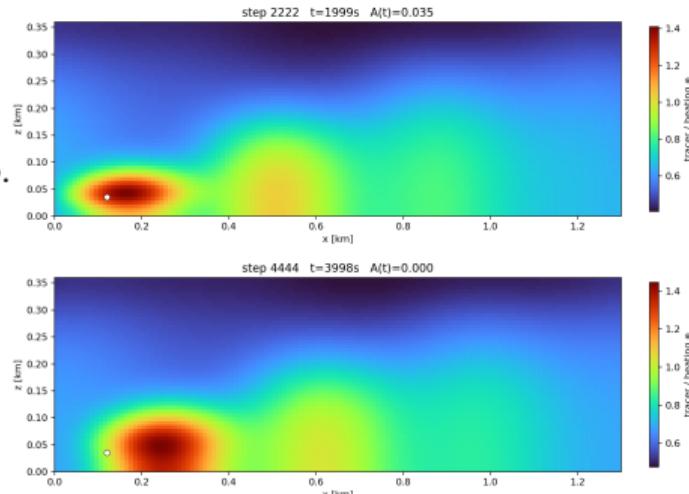
Dynamics (PDE)

$$\frac{\partial \phi}{\partial t} = -u \frac{\partial \phi}{\partial x} - w \frac{\partial \phi}{\partial z} + K \nabla^2 \phi + A(t) S(x, z) - \lambda \phi - \lambda_{\text{top}}(z) \phi.$$

Key design choices

- ▶ Mean wind: (u, w) right + upward \Rightarrow visible transport
- ▶ Source $S(x, z)$: localized bottom-left heating region
- ▶ Pulsed forcing $A(t)$ \Rightarrow visible trace stripes
- ▶ Damping λ + top sponge λ_{top} \Rightarrow equilibrium

Time stepping: RK4.



Boundary Conditions: Wrap-Around Transport (Periodic in x)

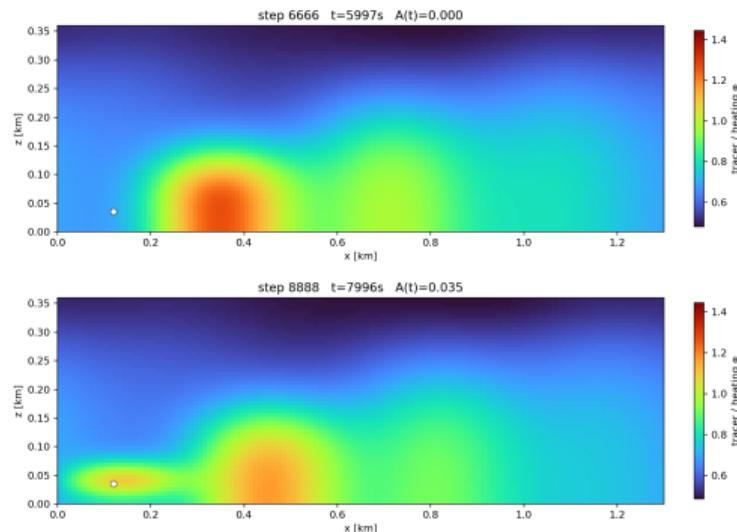
We want a flow where structures move out to the right and re-enter from the left.

Boundary conditions

- ▶ Periodic in x : outflow wraps around
- ▶ Reflective/top treatment in z (plus sponge)

Effect

- ▶ A persistent tracer train forms
- ▶ The system reaches a statistical steady state
- ▶ Ideal for learning time transitions



This creates a clean, cyclic “atmosphere” for ML demos.

RK4 Implementation (Core Loop)

Right-hand side

$$\text{RHS}(\phi, t) = -u \partial_x \phi - w \partial_z \phi + K \nabla^2 \phi + A(t)S(x, z) - (\lambda + \lambda_{\text{top}})\phi.$$

One RK4 step (schematic)

```
k1 = rhs(phi, t)
k2 = rhs(phi + 0.5*dt*k1, t + 0.5*dt)
k3 = rhs(phi + 0.5*dt*k2, t + 0.5*dt)
k4 = rhs(phi + dt*k3,      t + dt)
phi_next = phi + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
```

Snapshot logic

- ▶ Choose n_{vis} time indices between 0 and n_{steps}
- ▶ Save numbered PNGs: 1_dyn_XX.png

This gives clean training snapshots + nice lecture figures.

Energy / Mass Budget Monitoring (Sanity Checks)

We monitor injected heat content and losses:

$$C(t) = \int \phi \, dx \, dz, \quad I(t) = \int A(t)S \, dx \, dz, \quad L(t) = \int (\lambda + \lambda_{top})\phi \, dx \, dz.$$

Cumulative budget (discrete)

$$\text{acc_in} = \sum_n I(t_n)\Delta t, \quad \text{acc_loss} = \sum_n L(t_n)\Delta t.$$

- ▶ $\int \phi$ stabilizes near equilibrium
- ▶ injected vs lost energy becomes balanced
- ▶ confirms numerical stability and correct forcing/damping design

Always build monitoring into the toy model: it prevents silent nonsense.

Observations: Radiosondes (Sparse Vertical Profiles)

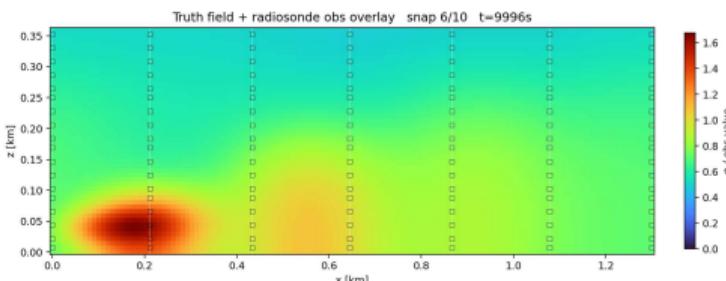
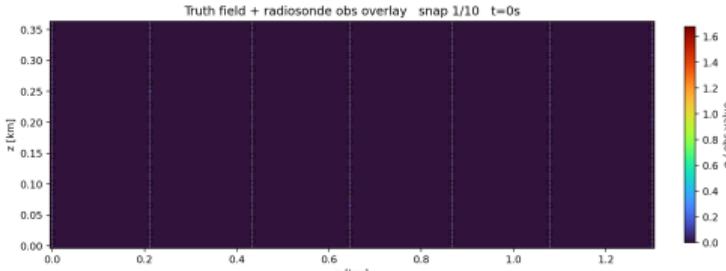
Radiosonde observation operator

At station column x_s and selected heights z_k :

$$y_t^{rs}(s, k) = \phi(x_s, z_k, t) + \epsilon.$$

Geometry is discrete

- ▶ n_{rs} station columns: `rs_ix`
- ▶ n_{vert} vertical levels: `rs_iz`
- ▶ output array: `yrs[time, station, vert]`



RS gives a sparse but physically intuitive reference view.

Observations: Satellite (Vertical Weighting Integrals)

Satellite observation operator

For each channel c (Gaussian vertical weights $w_c(z)$):

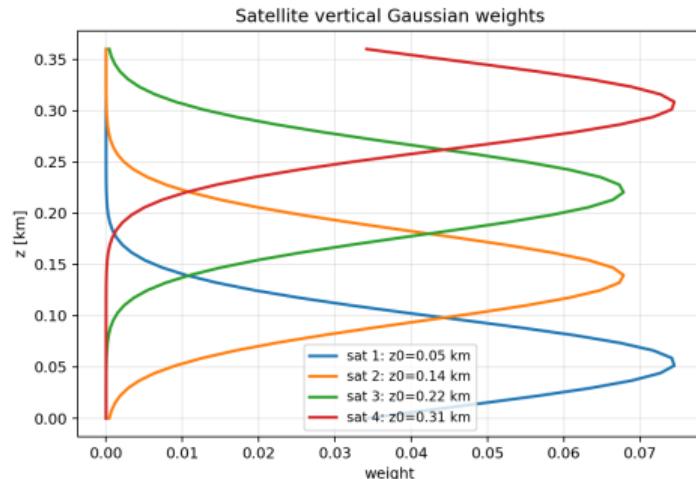
$$y_t^{\text{sat}}(c, x_i) = \sum_{j=1}^{n_z} w_c(z_j) \phi(x_i, z_j, t) \Delta z.$$

Properties

- ▶ available for all columns x_i
- ▶ integrated information \Rightarrow ill-posed inverse problem
- ▶ multiple channels \Rightarrow multi-layer sensitivity

Array

- ▶ `ysat[time, channel, x]`



Gaussian vertical weighting functions
used for the SAT channels.

Saved Dataset: Truth + Two Observation Types

We store everything into `dyn_truth_obs.npz`:

- ▶ **Truth fields**

$$x_{true} \in \mathbb{R}^{T_{snap} \times n_z \times n_x}$$

- ▶ **Radiosondes**

$$yrs \in \mathbb{R}^{T_{snap} \times n_{rs} \times n_{vert}}$$

- ▶ **Satellite**

$$ysat \in \mathbb{R}^{T_{snap} \times n_{sat} \times n_x}$$

- ▶ Snapshot times: `t_snap`
- ▶ Geometry: `rs_ix`, `rs_iz`
- ▶ SAT vertical weights: `sat_w`

This allows a clean separation: Notebook 1 = generate dataset, Notebook 2 = learn transitions.

Learning Task: Next-Step Obs Forecasting

We train an ML model that predicts the next observation state:

$$(y_t^{sat}, y_t^{rs}) \mapsto (\hat{y}_{t+1}^{sat}, \hat{y}_{t+1}^{rs}(\cdot)).$$

Inputs at time t

- ▶ Satellite curtain: $y_t^{sat}(c, x)$ for all columns
- ▶ RS set of points: $\{(x_m, z_m, y_m)\}_{m=1}^{N_{in}}$

Outputs at time $t + 1$

- ▶ Predicted satellite curtain $\hat{y}_{t+1}^{sat}(c, x)$
- ▶ Predicted RS values at query points $\hat{y}_{t+1}^{rs}(x_q, z_q)$

Important: query points can be anywhere \Rightarrow generalization to new RS placements .

Normalization: The Key Practical Ingredient

We normalize observations before training:

SAT: per-channel statistics

$$y_n^{sat} = \frac{y^{sat} - \mu_{sat}}{\sigma_{sat}}, \quad \mu_{sat} = \mathbb{E}_{t,x}[y^{sat}], \quad \sigma_{sat} = \text{std}_{t,x}[y^{sat}].$$

RS / truth: global

$$\phi_n = \frac{\phi - \mu_\phi}{\sigma_\phi}, \quad y_n^{rs} = \frac{y^{rs} - \mu_\phi}{\sigma_\phi}.$$

Without normalization the training becomes unstable and “learns the wrong scale”.

Flexible RS Input: Two Dataset Modes (A/B Switch)

We implement a dataset switch (important for interpretability):

Mode A (synthetic RS from truth)

- ▶ sample RS input points randomly from $xtrue[t]$
- ▶ add realistic noise σ_{rs}
- ▶ excellent generalization for arbitrary RS geometry

Mode B (use only stored RS observations)

- ▶ RS input points are exactly $yrs[t,:,:]$ at fixed rs_ix , rs_iz
- ▶ strict statement: “**input uses only observations at time t** ”
- ▶ weaker geometric diversity, but realistic RS network

Reality: We have only fixed radiosondes, but we have airplanes !

Saved:

data_dyn_obs/dyn_truth_obs.npz

Content:

`xtrue` shape=(10,50,130)
`yrs` shape=(10,7,18)
`ysat` shape=(10,4,130)

`t_snap` shape=(10,)
`rs_ix` shape=(7,)
`rs_iz` shape=(18,)
`sat_w` shape=(4,50)

`x` shape=(130,)
`z` shape=(50,)

Architecture: CNN on SAT + Set Encoder for RS + Query Head

Why a CNN?

SAT is a **curtain** $y^{sat}(c, x)$ along x .

Spatial patterns advect \Rightarrow translation-like structure.

Components

- ▶ **SAT encoder:** 1D-CNN
 - ▶ extracts local features along x
- ▶ **RS encoder:** DeepSets / pooling
 - ▶ handles variable-size RS point sets
- ▶ **Query decoder:** predicts $\hat{y}_{t+1}^{rs}(x_q, z_q)$

This enforces coupling: RS inference is driven by SAT structure .

Query-style output

$$\hat{y}_{t+1}^{rs}(x_q, z_q) = g_\theta(\text{CNN}(y_t^{sat}), \text{Set}(y_t^{rs}), x_q, z_q).$$

Practical win

- ▶ RS can be placed anywhere
- ▶ same model predicts:
 - ▶ sparse profiles
 - ▶ full fields (query everywhere)

DeepSets / pooling: why order does not matter

RS inputs form a set $\mathcal{S} = \{(x_m, z_m, y_m)\}_{m=1}^N$. A set has no order, so the encoding must satisfy $F(\mathcal{S}) = F(\pi(\mathcal{S}))$ for any permutation π . We enforce this with a permutation-invariant encoder:

$$e_m = \psi_\theta(x_m, z_m, y_m), \quad E = \text{pool}(e_1, \dots, e_N)$$

where pooling is sum/mean/max (commutative \Rightarrow order-invariant). Locations matter via (x_m, z_m) inside ψ_θ .

Training Loss: Joint SAT + RS Query Targets

We train on snapshot transitions $t \rightarrow t + 1$ using a joint loss:

SAT loss (next curtain)

$$\mathcal{L}_{sat} = \|\hat{y}_{t+1}^{sat} - y_{t+1}^{sat}\|_2^2.$$

RS query loss (next profile at query points)

$$\mathcal{L}_{rs} = \frac{1}{N_q} \sum_{q=1}^{N_q} (\hat{y}_{t+1}^{rs}(x_q, z_q) - y_{t+1}^{rs}(x_q, z_q))^2.$$

Total

$$\mathcal{L} = \mathcal{L}_{sat} + \alpha \mathcal{L}_{rs}.$$

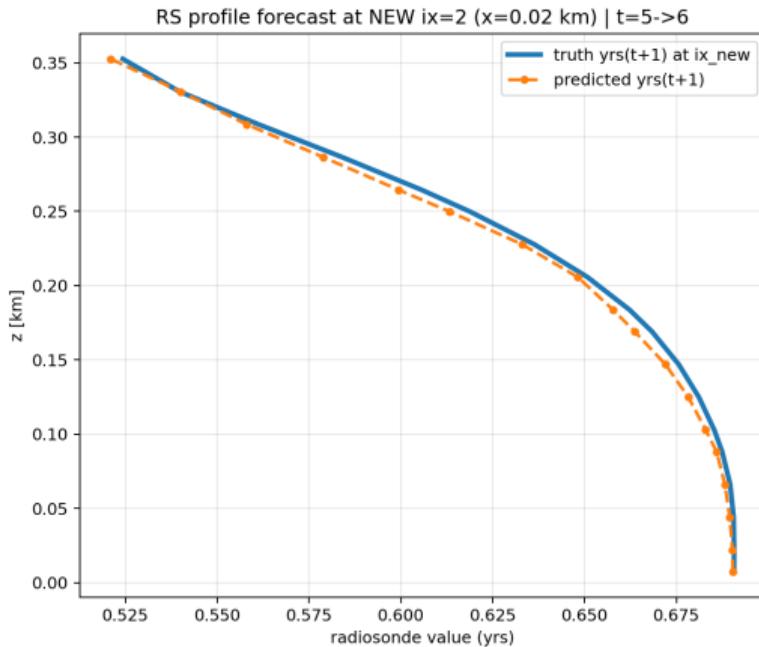
Note: RS targets for arbitrary (x_q, z_q) are taken from truth $xtrue[t + 1]$.

Evaluation 1: Predict a New RS Profile at an Unseen Location

We test generalization :

- ▶ Input RS: taken from stored network $yrs[t, :, :]$
- ▶ Choose a new x column not used by RS
- ▶ Query all heights $\{z_q\}$ at that x
- ▶ Compare predicted profile \hat{y}_{t+1}^{rs} vs truth $xtrue[t + 1]$

This checks whether the model learned physics-like transport information from SAT.



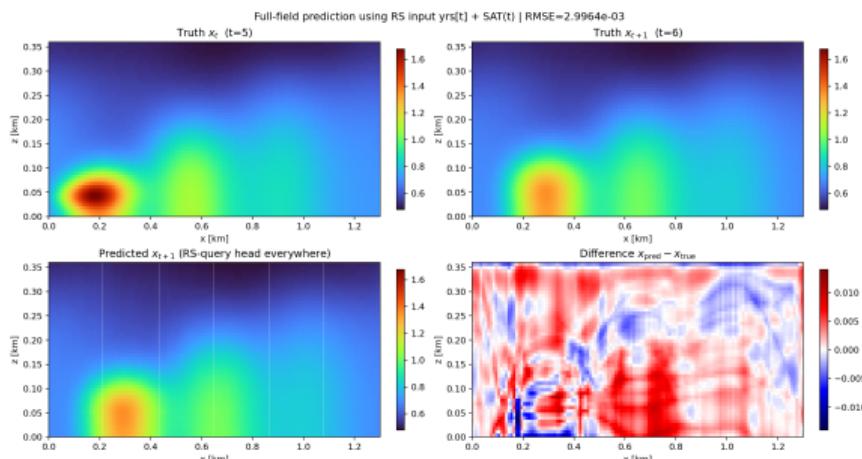
Evaluation 2: Full-Field Reconstruction at $t + 1$ (Query Everywhere)

The RS query head can be evaluated on the whole grid:

$$x_{\text{pred}}(x_i, z_j, t + 1) := \hat{y}_{t+1}^{rs}(x_i, z_j).$$

We visualize

- ▶ $x_{\text{true}}[t]$ (reference)
- ▶ $x_{\text{true}}[t + 1]$ (truth)
- ▶ $x_{\text{pred}}[t + 1]$ (prediction)
- ▶ difference $x_{\text{pred}} - x_{\text{true}}$



This turns obs-to-obs learning into a state-like field prediction.

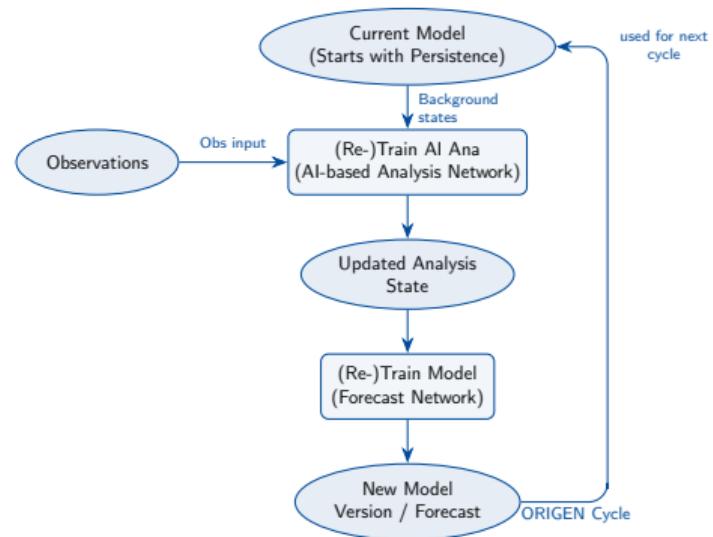
Observation-based Reconstruction & Inversion for Generative Emulation of Nonlinear Systems: ORIGEN

ORIGEN cycle schematic

- ▶ A closed learning cycle linking observations , analysis reconstruction , and forecast models .

Conceptual loop

- ▶ Start from a current model (here: persistence / baseline)
- ▶ Use observations to reconstruct state:
AI analysis / inversion
- ▶ Update the model/forecast emulator from reconstructed states
- ▶ Iterate: next cycle uses the new model



ORIGEN on a Minimal Example: Simple Oscillator

Minimal testbed (2D circle)

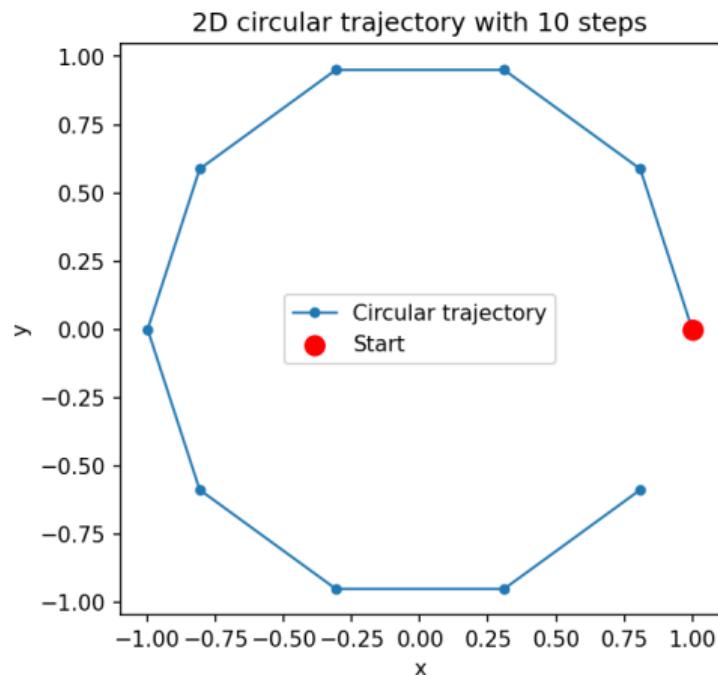
- ▶ Truth trajectory $x_k = (x_{1,k}, x_{2,k})$ on a circle

$$x_k = \begin{bmatrix} \cos \theta_k \\ \sin \theta_k \end{bmatrix}, \quad \theta_k = \frac{2\pi k}{n}$$

- ▶ Observations are scalar and partial :

$$y_k = H_k x_k + \epsilon_k, \quad H_k \in \{[1, 0], [0, 1]\}$$

Aim: Explain ORIGEN mechanics without complex dynamics.



Observations: Partial Measurements of x_1 or x_2

Time-dependent observation operator

- ▶ At each step k we observe only one component

▶ Selector:

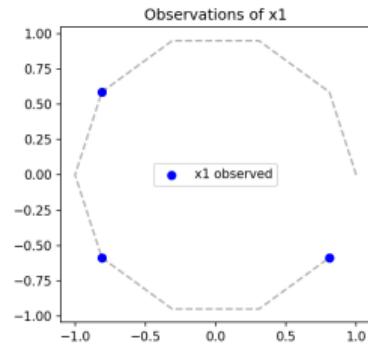
$$s_k \in \{1, 2\}$$

- ▶ Observation operator:

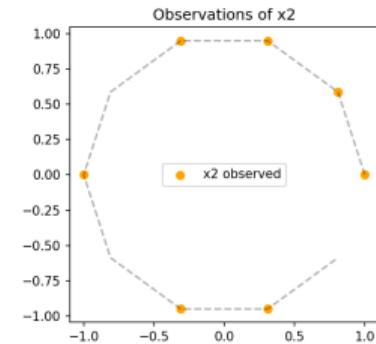
$$H_k = \begin{cases} [1, 0] & s_k = 1 \text{ (observe } x_1) \\ [0, 1] & s_k = 2 \text{ (observe } x_2) \end{cases}$$

- ▶ Observation equation:

$$y_k = H_k x_k + \epsilon_k, \quad \epsilon_k \sim \mathcal{N}(0, R)$$



Blue: x_1 observed



Orange: x_2 observed

Interpretation: This mimics
heterogeneous sensors and missing data .

3D-Var Step: Background → Analysis (Single Observation)

We combine background + one scalar obs

- ▶ Background state: $x_k^b \in \mathbb{R}^2$
- ▶ Obs: $y_k \in \mathbb{R}$, operator $H_k \in \mathbb{R}^{1 \times 2}$

Innovation

$$d_k = y_k - H_k x_k^b$$

Gain (scalar obs)

$$K_k = B H_k^\top \left(H_k B H_k^\top + R \right)^{-1}$$

Analysis update

$$x_k^a = x_k^b + K_k d_k$$

Effect: update only in observed direction, but shaped by B .

Initial choice of covariances

$$B = \begin{bmatrix} \sigma_b^2(x_1) & 0 \\ 0 & \sigma_b^2(x_2) \end{bmatrix}, \quad R = \sigma_o^2$$

Cycling (persistence model)

$$x_{k+1}^b = M(x_k^a), \quad M(x) = x$$

- ▶ This produces a full sequence:

$$x_0^b \rightarrow x_0^a \rightarrow x_1^b \rightarrow \dots$$

- ▶ We visualize results next: truth vs background vs analysis

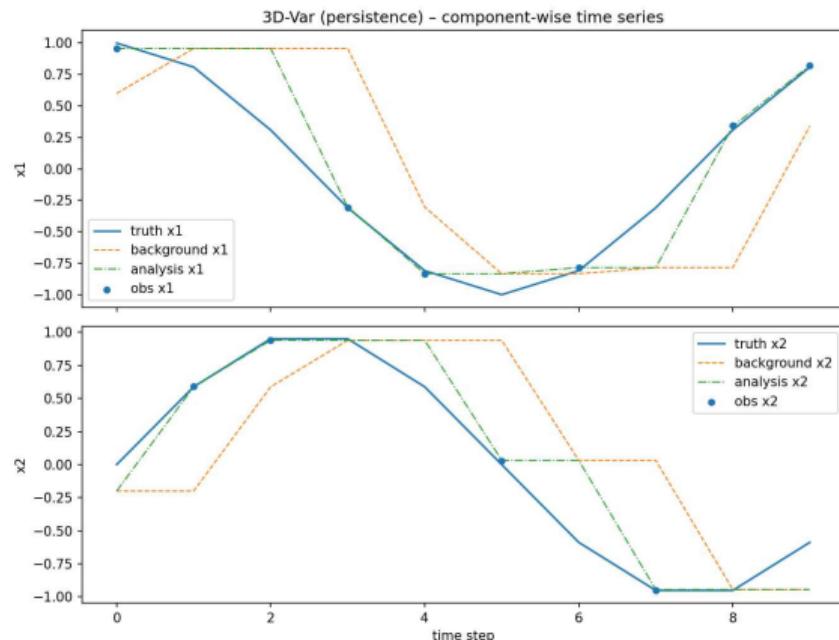
3D-Var Cycle Result: Time Series of x_1 and x_2

What we compare

- ▶ Truth x_k
- ▶ Background x_k^b
- ▶ Analysis x_k^a
- ▶ and the scalar observations y_k

Key behavior

- ▶ analyses are pulled towards the obs
- ▶ cycling propagates improvements forward
- ▶ unobserved component still benefits indirectly (via B and cycling)



Idea: Information is collected iteratively.

Sequential Rounds: With All Observations We Converge in Two Cycles

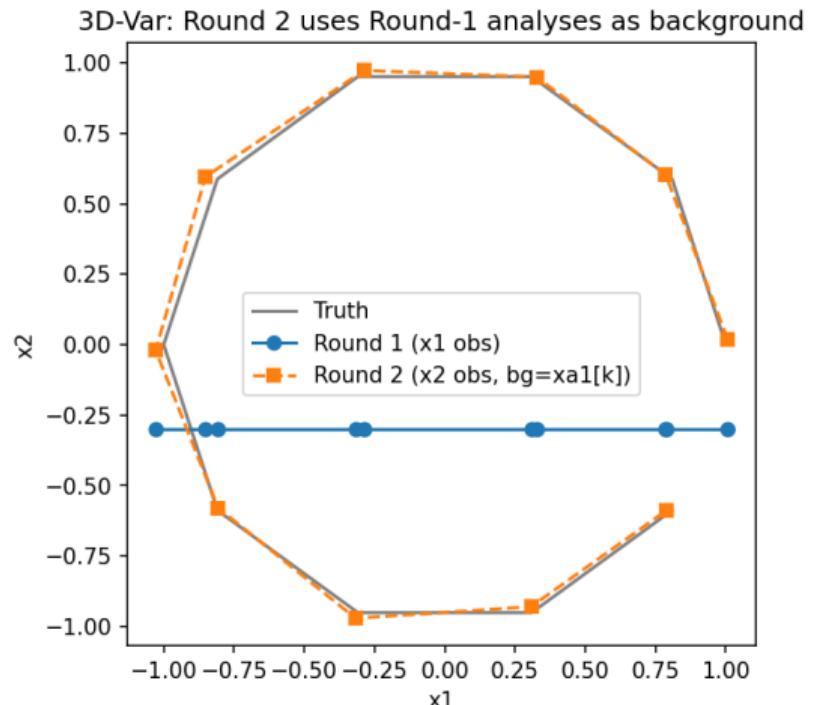
Idea: assimilate in two rounds

- ▶ Round 1: observe x_1 only
$$y_k^{(1)} = x_{1,k} + \epsilon_k$$
- ▶ Round 2: observe x_2 only, using
$$x_k^{b,(2)} := x_k^{a,(1)}$$

If obs error is small

- ▶ after Round 1: x_1 aligns with truth
- ▶ after Round 2: x_2 aligns with truth
- ▶ both components observed ⇒ fast convergence

Message: With complete information,
ORIGEN-style cycling converges quickly.



Iterative 3D-Var: Reconstruction Improves over Cycles (Demo)

What is iterated here?

- We reconstruct an entire trajectory

$$\{x_k^a\}_{k=0}^{n-1}$$

from noisy, partial observations

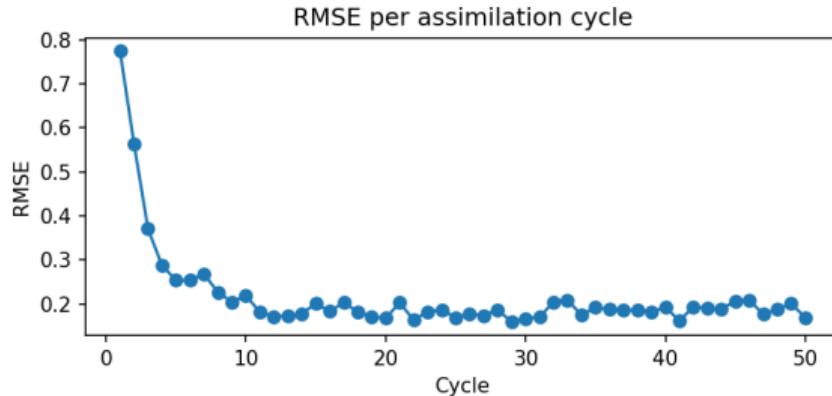
$$y_k = H_k x_k + \epsilon_k$$

- Each cycle produces a refined estimate of the full sequence:

$$\{x_k^{a,(c)}\} \Rightarrow \{x_k^{a,(c+1)}\}$$

Crucial point

- The “dynamics” is implicit in the reconstructed series
- We are not propagating with a separate model; we repeatedly improve the sequence itself



RMSE per assimilation cycle: iterative 3D-Var improves the reconstructed trajectory

ORIGEN: from the reconstructed $\{x_k^a\}$ we learn a forecast map

$$x^a(t) \mapsto x^a(t + \Delta t),$$

in the notebook we focus on trajectory reconstruction.

Reconstruction: 2D Trajectory over Iterative 3D-Var Cycles

Iterative 3D-Var reconstruction

- ▶ Each cycle reconstructs the full sequence:

$$\{x_k^a\}_{k=0}^{n-1}$$

- ▶ Observations remain **noisy and partial**:

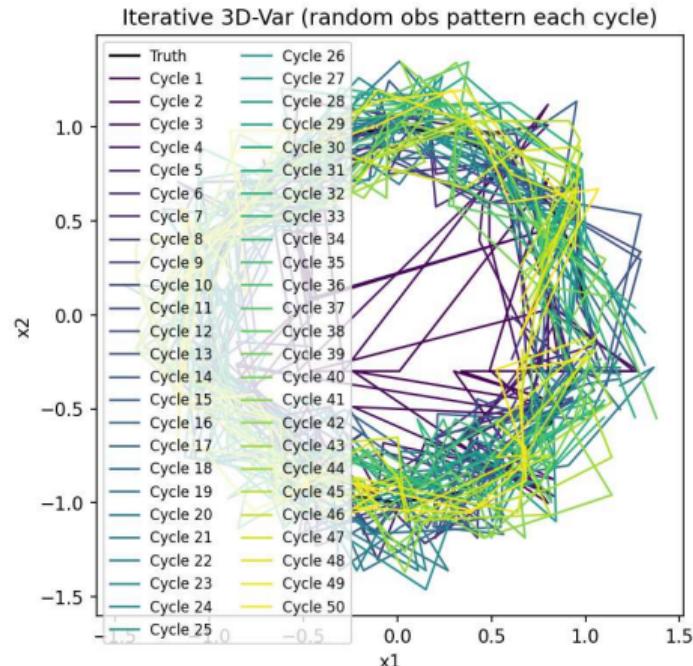
$$y_k = H_k x_k + \epsilon_k$$

- ▶ Repeated cycles reduce reconstruction error

What you see on the right

- ▶ Truth trajectory (black)
- ▶ Selected analysis cycles (colored)
- ▶ Later cycles are closer to truth

Message: 3D-Var keeps error coming in from the observation error



Reconstruction: Cycle Time Series with Noisy Observations

Final-cycle diagnostics

- ▶ Compare truth vs final reconstructed trajectory
- ▶ Show observations for the last random pattern:

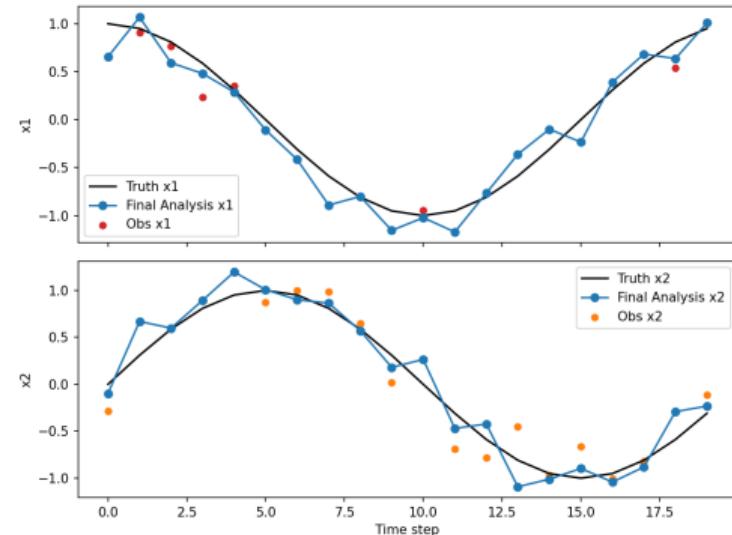
$$H_k \in \{[1, 0], [0, 1]\}$$

- ▶ Observations are noisy \Rightarrow analysis does not overfit

3D-Var property

- ▶ analysis is a weighted compromise
- ▶ controlled by B and R

Message: 3D-VAR is not enough



Final cycle: time series for x_1 and x_2 with noisy partial observations

Adding a Covariance Update: Kalman Filter–Type Uncertainty Reduction

Upgrade: update B during cycling

- ▶ Until now: fixed background covariance B
- ▶ Now: after each analysis step we also update uncertainty

Kalman filter covariance update

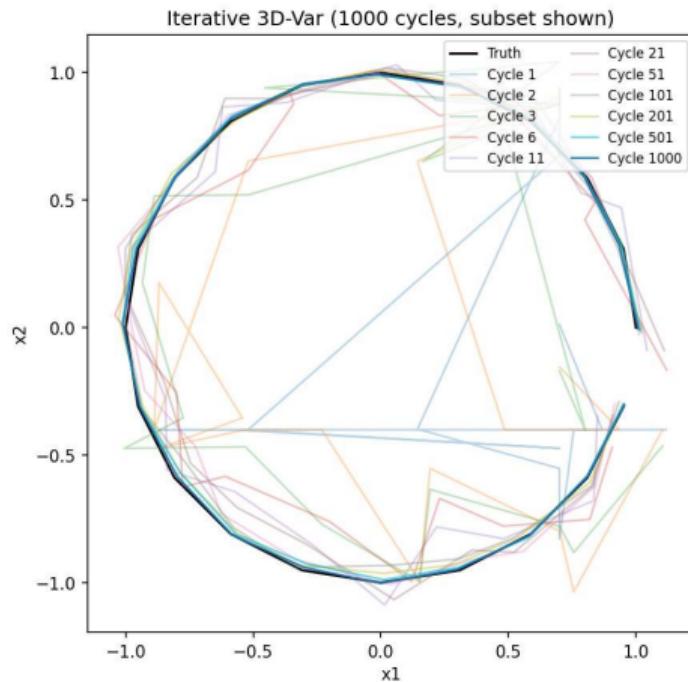
$$A_k = (I - K_k H_k) B_k \quad \Rightarrow \quad B_k \leftarrow A_k$$

Effect:

- ▶ uncertainty shrinks in observed directions
- ▶ gain adapts across cycles
- ▶ analysis trajectory converges strongly to truth

Message:

With adaptive B , iterative DA converges.



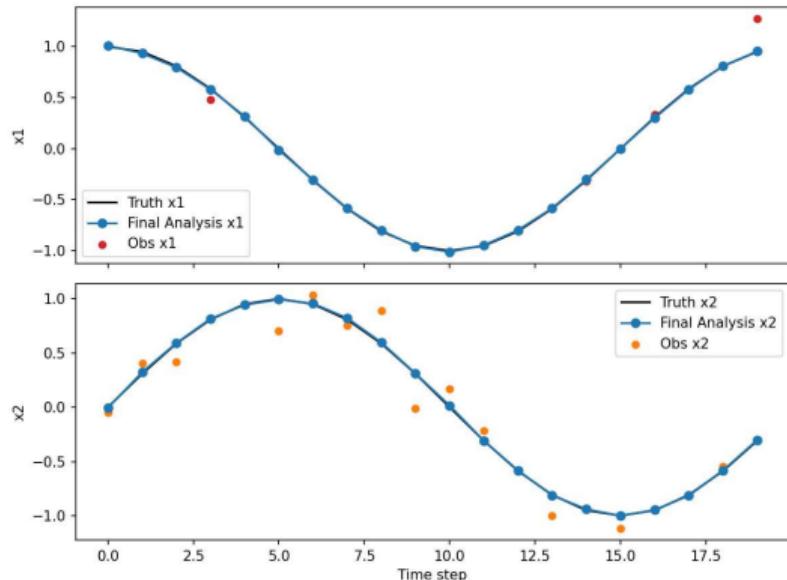
Full Convergence: Final Trajectory with Covariance Update

Final-cycle reconstruction

- ▶ With B update, repeated cycling yields near-perfect reconstruction
- ▶ Even with noisy and partial observations

Why it converges

- ▶ analysis reduces both error and uncertainty
- ▶ smaller uncertainty \Rightarrow more consistent updates
- ▶ loop stabilizes around the truth trajectory



Message: ORIGEN principle: reconstruction + uncertainty adaptation \Rightarrow convergence.

Final cycle: time series x_1, x_2 converged

Lorenz-63: ORIGEN Reconstruction & Learned Forecast Model

ORIGEN loop (concept)

1. Assimilate random observed subsets (x, y, z, xy, xz, yz)
2. Update background $\mathbf{x}^b \leftarrow \mathbf{x}^a$
3. Update covariance $B \leftarrow P^a$ (Kalman-style)

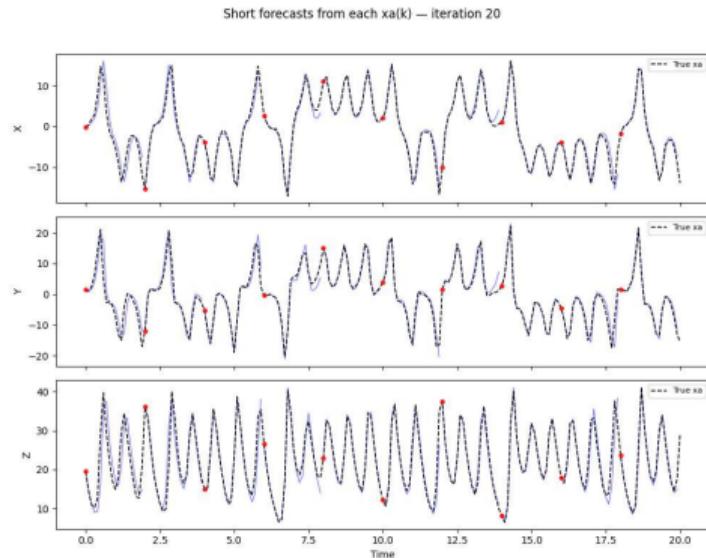
Model learning

- ▶ Train NN to learn one-step map

$$\mathbf{x}^a(k) \mapsto \mathbf{x}^a(k+1)$$

- ▶ **Fine-Tuning with Rollout**

Rollout: many subsequent short forecasts



Forecast rollouts (blue) starting from analysis states (red dots), compared to the reconstructed reference trajectory (black dashed).