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)

```yaml
dataset:
  name: era5
  variables: [t2m, u10, v10]
  resolution: 0.25

model:
  name: graph_transformer
  hidden_dim: 256
  num_layers: 6

training:
  batch_size: 4
  max_epochs: 50
  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

OmegaConf solves this

- represents configuration as Python objects
- preserves hierarchy and structure
- supports interpolation and defaults

The result is a `DictConfig` object.

Properties of `DictConfig`

- hierarchical (mirrors YAML structure)
- accessed via attributes or keys
- supports type checking

Why this matters

- training code stays generic
- behaviour is fully configuration-driven
- experiments become reproducible by construction

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

```python
# cfg provided by Hydra
print(cfg.model.layer_sizes)
print(cfg.training.max_epochs)
```

Resolving and converting

```python
from omegaconf import
    OmegaConf

OmegaConf.resolve(cfg)
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

The experiment is composed, not rewritten.

config.yaml

```yaml
defaults:
  - model: mlp
  - training: simple
  - _self_
```

model/mlp.yaml

```yaml
layer_sizes: [1, 64, 1]
activation: relu
```

model/deep.yaml

```yaml
layer_sizes: [1, 128, 64, 32,
    1]
activation: tanh
```

Hydra in Practice: Same Code, Different Outcomes

What changes between these runs

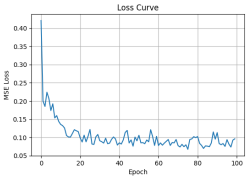
- only the Hydra configuration
- different model definitions (`mlp` vs `deep`)
- same training loop, same data

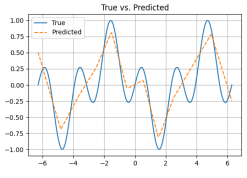
Hydra controls:

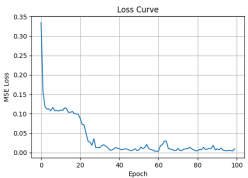
- which YAML files are composed
- which model architecture is instantiated
- which hyperparameters are active

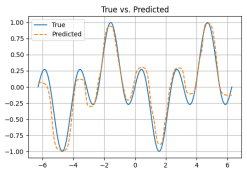
Key observation

- 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

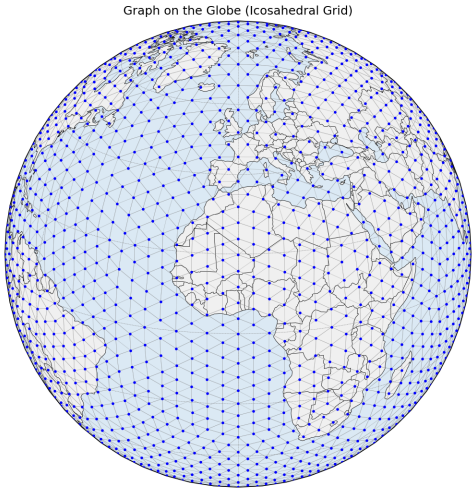
- refined icosahedral grids
- nearly uniform cell areas
- no singularities at the poles

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

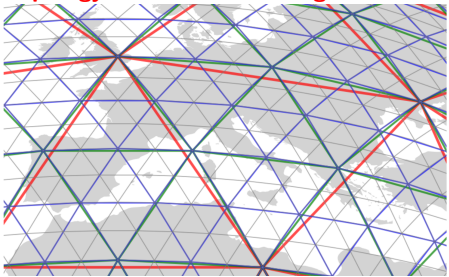
Edge index representation

- 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

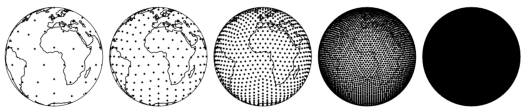
Key components

- `TriNodes` from refined icosahedra
- neighbour definitions via kNN
- edge construction on the sphere

Design principle

- graph built once
- reused for training and inference
- geometry independent of data values

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

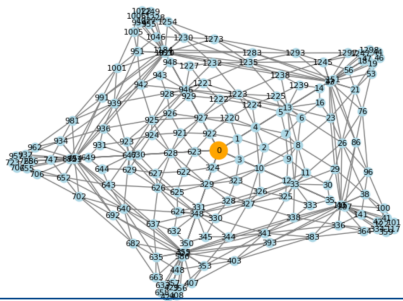
Model families

- 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

Command-line workflow

```
anemoi-datasets create \
  dataset.yaml \
  era5.zarr
```

Result

- validated Zarr dataset
- embedded metadata and statistics
- directly usable by Anemoi training

Data preparation becomes a reproducible pipeline.

https://anemoi.readthedocs.io/projects/datasets/en/latest/

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

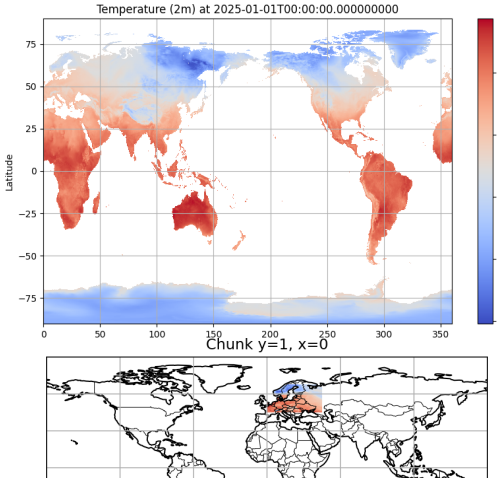
What these images show

- 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.

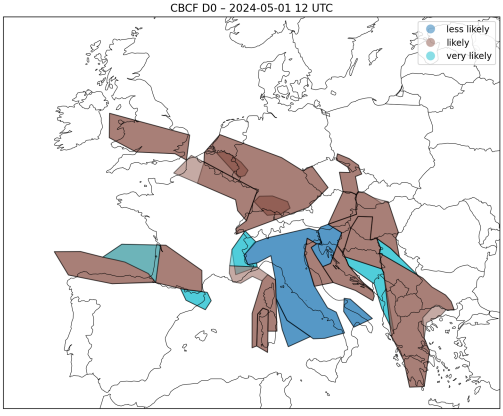
Training data

- 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

We train on masks — and recover polygons afterwards.



Example CBCF polygons (likelihood categories).

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 $m(x) \in [0, 1]$

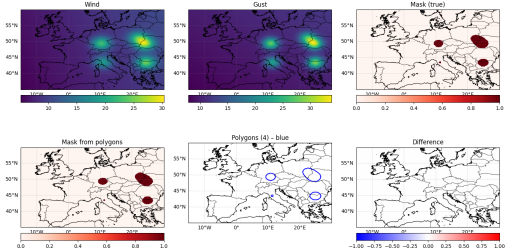
Two-way conversion

- **Polygon → mask:** rasterize onto model grid
- **Mask → polygon:** contour / region extraction

Quality control

- check mask(polygons) against reference labels
- difference highlights discretization artefacts

- predictors: wind + gust fields
- target: warning mask derived from polygons



Polygon-to-mask conversion and consistency check.

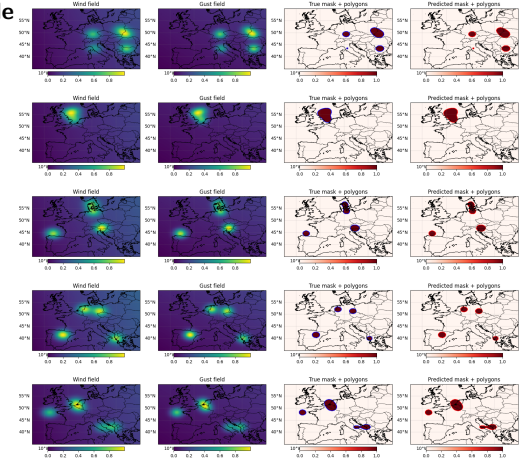
Learning Demo: Inputs (Wind/Gust) → True vs Predicted Mask & Polygons

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

Layout (4 columns)

1. **Input 1:** wind field
2. **Input 2:** gust field
3. **Target:** true mask + true polygons
4. **Prediction:** predicted mask + recovered 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

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.

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.)

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

Take-away

- same pipeline works for long archives
- basis for operational verification and monitoring

Code sketch (from `main.py`)

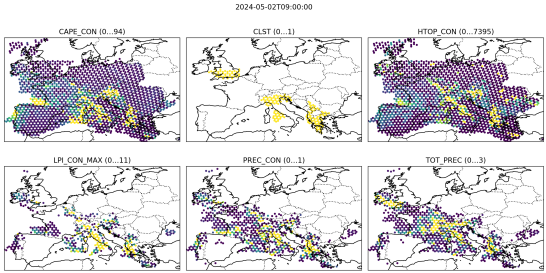
```
model = torch.load(ckpt)
model.eval()

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.

CBCF Results: Anemoi Training Demo (Examples)

What we show

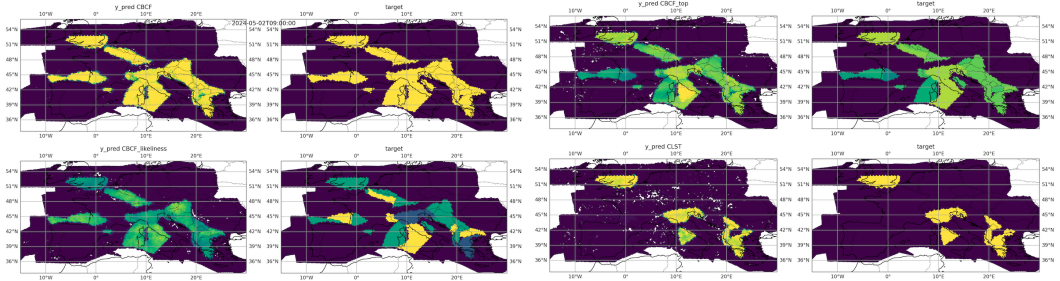
- results from the Anemoi-based training workflow
- prediction of CBCF target fields / masks from forecast inputs
- comparison against the corresponding targets



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_likeliness`: 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)