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

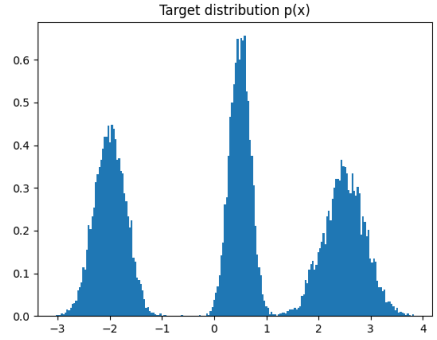
Learning to Sample a Distribution

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



Target distribution

Sampling via a Neural Network

Neural sampler: noise → data

Neural generator mapping noise to samples

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 1) )

    def forward(self, z):
        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)

```
def wasserstein_1d(x_gen, x_data):
  xg, _ = torch.sort(x_gen.view(-1))
  xd, _ = torch.sort(x_data.view(-1))
  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

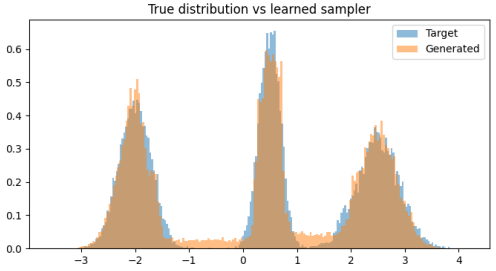
Training result

- 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

This already enables stochastic modeling.



Target distribution and generated samples

Why One-Step Sampling Is Not Enough

What we did so far

- Learn a direct map: noise $\rightarrow$ 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)

```
# x_t : noisy signal at step t
# model learns to predict a less noisy version

for t in reversed(range(1, T)):
    x = model(x, t) # denoise one step
```

Each step is simple and stable.

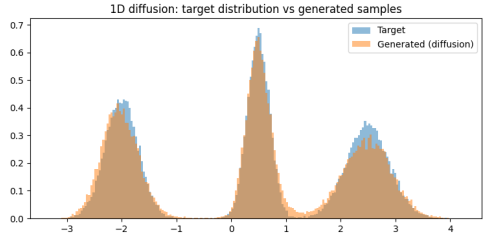
Training principle

- 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

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:

$$q(x_{1:T} \mid x_0) = \prod_{t=1}^{T} q(x_t \mid x_{t-1})$$

with Gaussian transitions

$$q(x_t \mid x_{t-1}) = \mathcal{N}\left(x_t;\ \sqrt{1-\beta_t}\, x_{t-1},\ \beta_t I\right),$$

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:

$$q(x_t \mid x_0) = \mathcal{N}\Big(x_t;\, \sqrt{\bar{\alpha}_t}\, x_0,\, (1-\bar{\alpha}_t) I\Big)$$

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 $t \to T$:

$$\bar{\alpha}_t \to 0 \quad \Rightarrow \quad x_t \sim \mathcal{N}(0, I).$$

This matches the example exactly.

Reverse Diffusion Process

The reverse process aims to invert the forward diffusion:

$$p_\theta(x_{0:T}) = p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1} \mid x_t),$$

with $p(x_T) = \mathcal{N}(0, I)$.

Each reverse step is modeled as a Gaussian:

$$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(x_{t-1};\, \mu_\theta(x_t, t),\, \Sigma_t)\,,$$

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:

$$x_0 = f^*(x_1) + \sqrt{\sigma^2_{\text{post}}}\,\xi, \quad \xi \sim \mathcal{N}(0, 1).$$

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

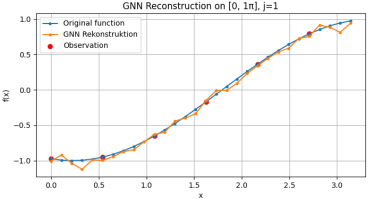
Setup

- 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

The task is to reconstruct $f(x)$ at *all* nodes.



Sparse observations and GNN reconstruction

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)

```
class GNNLayer(nn.Module):
  def __init__(self, dim):
    super().__init__()
    self.self_lin  = nn.Linear(dim, dim)
    self.neigh_lin = nn.Linear(dim, dim)

  def forward(self, x, edge_index):
    row, col = edge_index
    agg = torch.zeros_like(x)
    agg.index_add_(0, row, x[col])
    return F.relu(
        self.self_lin(x) +
        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

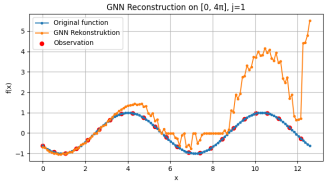
What changes

- Number of nodes N
- Physical domain length
- Resolution of the grid

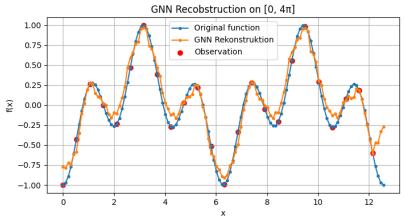
What stays the same

- 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.

Lightning idea

- Separate *what* from *how*
- Model logic vs. training mechanics
- Convention over configuration

Lightning is *PyTorch*, not a replacement.

```
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

```
class Model(pl.LightningModule):
    def forward(self, x):
        ...

    def training_step(self, batch, batch_idx):
        ...
        return loss

    def configure_optimizers(self):
        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

You still control

- Model architecture
- Loss functions
- Optimizers
- Data handling

Lightning enforces structure — not constraints.

```
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))
```

```
def training_step(self, batch, batch_idx):
    pred = self(batch)
    mask = ~torch.isnan(batch.y_obs)
    loss = F.mse_loss(pred[~mask], batch.y[~
    self.log("train_loss", loss)
    return loss

def configure_optimizers(self):
```

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

```
from torch_geometric.data import Data

data = Data(
    x = node_features,
    edge_index = edge_index,
    y = targets
)
```

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

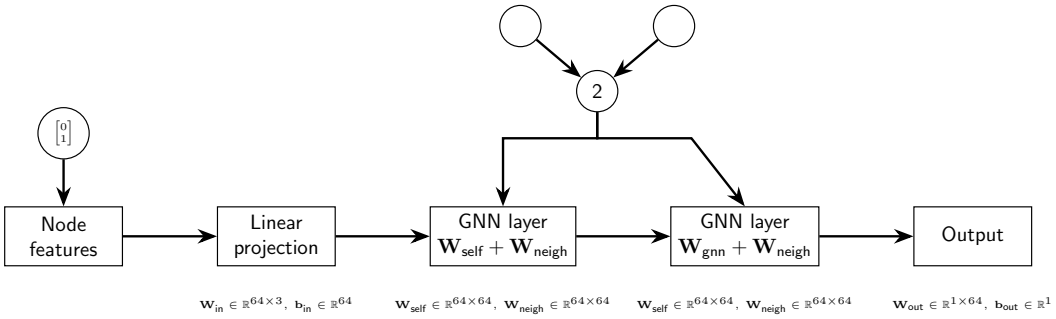
Each library does one thing well.

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

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

$$\mathbf{x}_i \in \mathbb{R}^d \quad (i = 1, \ldots, N)$$

- 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

$$\texttt{edge\_index} \in \mathbb{N}^{2 \times E}$$

Each column defines one directed edge:

$$(j \to i)$$

- 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

```
self.lin = nn.Linear(d, h)
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

```
row, col = edge_index
agg = torch.zeros_like(h)
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

$$H^{(0)} \in \mathbb{R}^{N \times d},$$
$$H^{(1)} = \Phi(H^{(0)}, \texttt{edge\_index}),$$
$$H^{(2)} = \Phi(H^{(1)}, \texttt{edge\_index})$$

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

```
self.out = nn.Linear(h, 1)
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.