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

Same data, different models

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

Conceptual view

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

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

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

Feed Forward Neural Networks (FFNN)

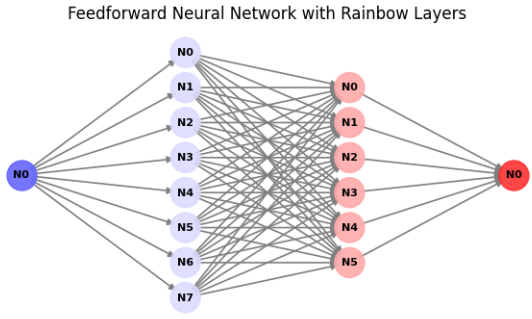
Basic idea

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

Typical use cases

- Regression
- Classification
- Function approximation

Each layer applies a learned transformation.



Input → hidden layers → output

FFNN as a Mathematical Mapping

Layer-wise computation

- Affine transformation
- Nonlinear activation
- Composition of functions

Each layer increases expressiveness.

Purely data-driven mapping.

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

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 $(n_{\text{in}} \to n_{\text{out}})$: $n_{\text{in}} \cdot n_{\text{out}} + n_{\text{out}}$

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

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),
            nn.Tanh(),
            nn.Linear(N1, N1),
            nn.Tanh(),
```

Depth encodes structure, not just capacity.

FFNN as a Function Approximator

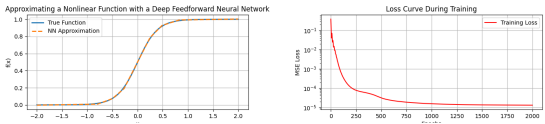
Problem setting

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

Neural-network view

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

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



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

Intuition

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

Computational Graph and Backpropagation

Forward pass

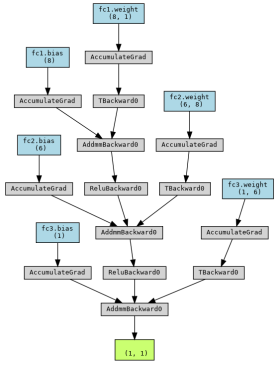
- Input propagated layer by layer
- Linear maps + nonlinear activations
- Produces model output $\hat{y}$

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

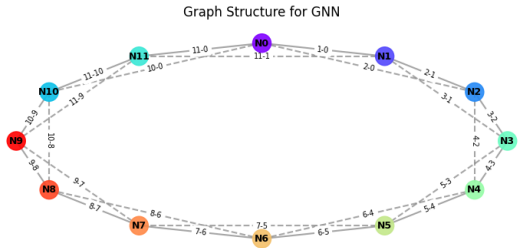
Why Graph Neural Networks?

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

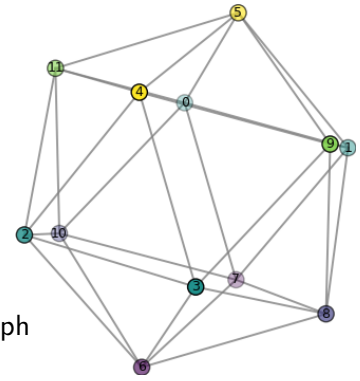
Basic Graph Concepts

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

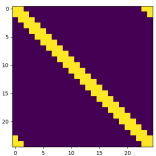
- Data lives on nodes
- Structure lives in edges
- Learning uses both

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

$$\texttt{edge\_index} = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \end{pmatrix}$$

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

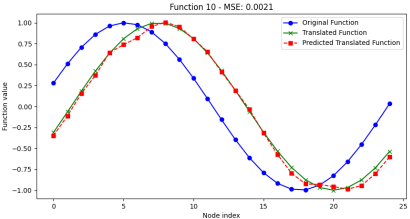
- Nodes arranged on a periodic ring
- Each node holds a scalar field value
- Goal: learn one-step time evolution

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

Interpretation:

message passing $\simeq$ numerical stencil

Training and test loss over epochs

Limitations

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

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:

$$\text{locality} + \text{translation invariance}$$

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

Classes:

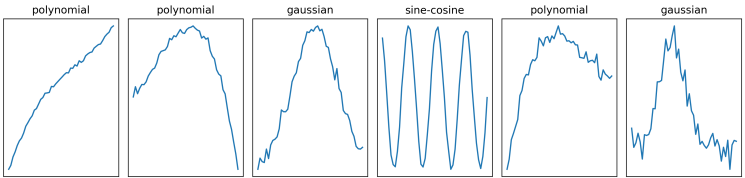
- Sine-like
- Gaussian
- Polynomial

Samples:

- Same grid
- Different shape

CNN sees:

- Local patterns
- Overall structure



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

```python
class FunctionClassifierCNN(nn.Module):
  def __init__(self):
     super().__init__()
     self.conv1 = nn.Conv1d(1, 16,5, padding=2)
     self.conv2 = nn.Conv1d(16,32,5, padding=2)
     self.fc1   = nn.Linear(32*50, 128)
     self.fc2   = nn.Linear(128, 4)

  def forward(self, x):
     x = torch.relu(self.conv1(x))
     x = torch.relu(self.conv2(x))
     x = x.view(x.size(0), -1)
     x = torch.relu(self.fc1(x))
     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

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
    model.parameters(), lr=1e-3)

for epoch in range(num_epochs):
    for xb, yb in train_loader:
        xb = xb.to(device)
        yb = yb.to(device)

        optimizer.zero_grad()
        logits = model(xb)
        loss = criterion(logits, yb)
        loss.backward()
        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

```
model.eval()

with torch.no_grad():
    logits = model(X_test.to(device))
    preds  = torch.argmax(logits, dim=1)

accuracy = (preds == y_test.to(device)).
    float().mean()
print("Accuracy:", accuracy.item())
```

Prediction result:

- Each function → class
- Robust to moderate noise

CNN Predictions on Test Data

What is shown?

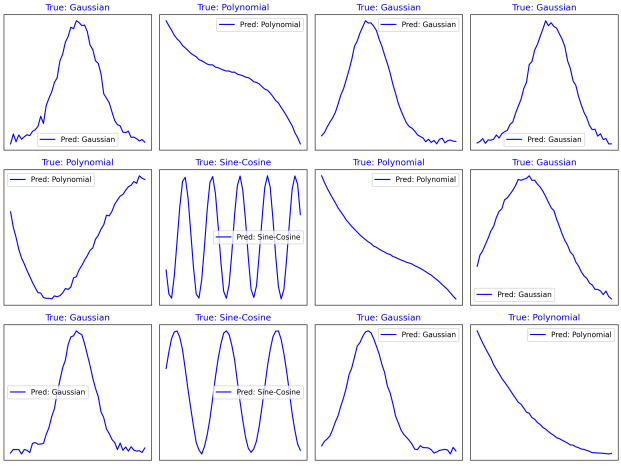
- Unseen test functions: new samples from random distribution
- Higher noise than training

Color coding

- Blue: correct classification
- Red: misclassification

Interpretation:

- CNN recognizes shape
- Errors near ambiguous cases



Predicted vs. true function classes

Why LSTMs for Time Series and Anomalies?

Problem setting

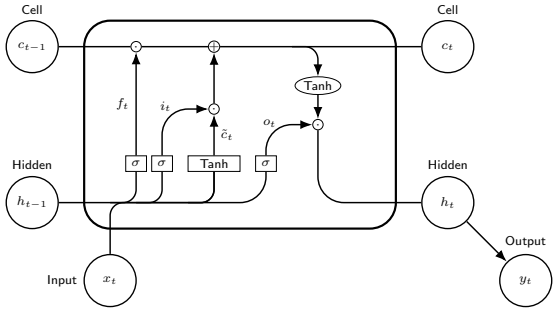
- Sequential sensor data
- Temporal correlations
- Deviations over time

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

What the LSTM does

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

What comes out

- Hidden states h_t
- Latent temporal representation

Reconstruction step

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

Training objective

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

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

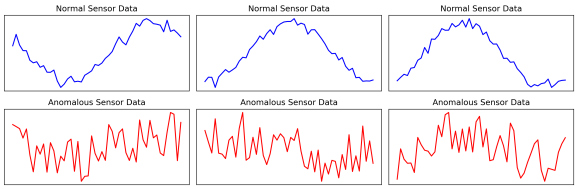
- Encode normal behaviour
- Reconstruct input sequence

Anomaly criterion

- Large reconstruction error
- Rare under normal dynamics

Key assumption:

$$\text{normal} \Rightarrow \text{low error}$$



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)

```python
class LSTMAutoencoder(nn.Module):
  def __init__(self, hidden_dim=32, layers=2):
    super().__init__()
    self.encoder = nn.LSTM(
      1, hidden_dim, layers, batch_first=True)
    self.decoder = nn.LSTM(
      1, hidden_dim, layers, batch_first=True)
    self.out = nn.Linear(hidden_dim, 1)

  def forward(self, x):
    _, (h, c) = self.encoder(x)
    z = torch.zeros_like(x)
    y, _ = self.decoder(z, (h, c))
    return self.out(y)
```

Detected Anomalies in Sensor Data

What is shown?

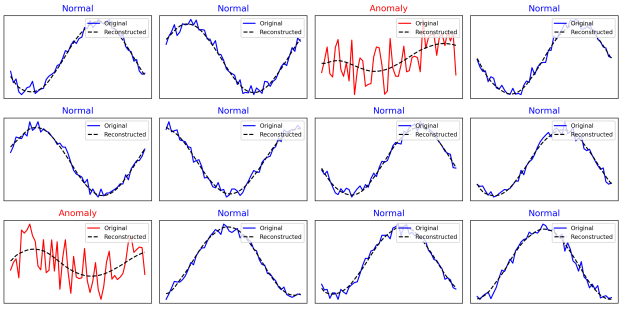
- Unseen test sequences
- Reconstruction vs. input

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