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 $x \to y$ 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 $\rightarrow$ 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

Prompt $\rightarrow$ Response

Iterative dialogue replaces static interfaces.

AI in research workflows

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

Human $\leftrightarrow$ 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

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

Works well within the learned data regime.

Limitations

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

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

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

$$\text{Decision} = \text{AI} + \text{Human}$$

AI supports, it does not decide.

Key risk dimensions

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

$$\text{Confidence} \neq \text{Correctness}$$

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

```python
import torch

x = torch.tensor([2.,3.],
    requires_grad=True)

y = x[0]**2 + x[1]**2
y.backward()

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 $x \to y$

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 $f_\theta : x \to \hat{y}$
- 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

```python
import torch.nn as nn

class SimpleNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1,16)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(16,1)

    def forward(self,x):
        x = self.fc1(x)
        x = self.relu(x)
        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: $1 \to 16$
- Layer 2: $16 \to 1$

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 $\Rightarrow$ 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:

$$\min_{\theta} \sum_{i} \left\| f_\theta(x_i) - y_i \right\|^2$$

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

```
model = nn.Sequential(
    nn.Linear(1,16), nn.ReLU(),
    nn.Linear(16,16), nn.ReLU(),
    nn.Linear(16,1)
)

loss_fn = nn.MSELoss()
opt = torch.optim.Adam(
      model.parameters(), lr=0.01)

for x_b,y_b in loader:
    opt.zero_grad()
    y_p = model(x_b)
    loss = loss_fn(y_p, y_b)
    loss.backward()
    opt.step()
```

Sine Example — Training Outcome

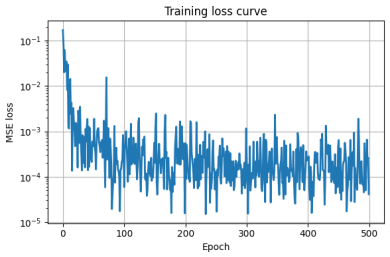
What happened during training?

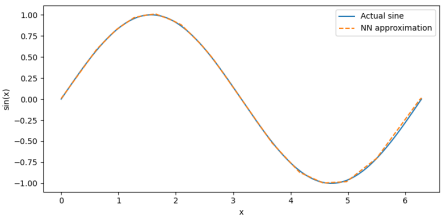
- Loss decreases over epochs
- Network parameters adapt
- Mapping $x \to \hat{y}$ improves

Interpretation

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

Loss during training





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

```
loader = DataLoader(
    dataset,
    batch_size=4,
    shuffle=False
)

for x_b, y_b in loader:
    print(y_b.squeeze())
    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

Shuffled batches

```
loader = DataLoader(
    dataset,
    batch_size=4,
    shuffle=True
)

for x_b, y_b in loader:
    print(y_b.squeeze())
    break
```

First batch changes every run.

```
=== 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]
```

```
=== 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

```
class SimpleClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2,1),
            nn.Sigmoid()
        )
    def forward(self,x):
        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

```
class BetterClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2,32),
            nn.ReLU(),
            nn.Linear(32,1),
            nn.Sigmoid()
        )
    def forward(self,x):
        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()
```

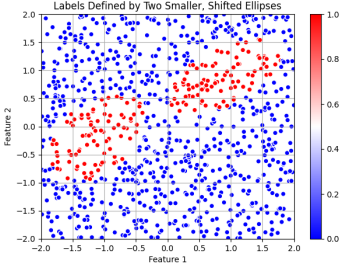
Prediction $\hat{y} \in [0, 1]$ interpreted as probability.

```
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

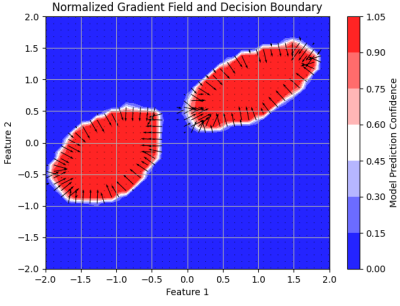
- Classes defined by geometry
- Two shifted, rotated ellipses
- Labels fixed before learning



Blue / red = predefined classes

Model prediction

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



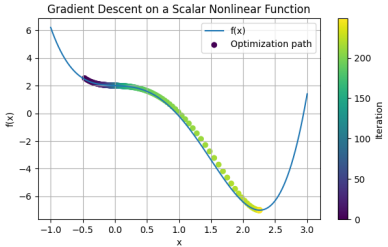
Colors = prediction, arrows = gradients

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.