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.

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:

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

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

LLMs as Sequence-to-Sequence Machines

Core abstraction

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

Mathematical view

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

- 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:

$$\text{dog bites man} \quad \neq \quad \text{man bites dog}$$

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 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

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

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.

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

$$X \in \mathbb{R}^{n \times d_{\text{model}}}, W^Q, W^K \in \mathbb{R}^{d_{\text{model}} \times d_k},$$
$$W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}, Q, K \in \mathbb{R}^{n \times d_k}, V \in \mathbb{R}^{n \times d_v}$$

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

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

- 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:

$$\text{its own } Q_i,\ K_i,\ V_i$$

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 $h = 1, \ldots, H$:

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

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

$$Z = \text{Concat}\left(Z^{(1)}, \ldots, Z^{(H)}\right)$$

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

Output of multi-head attention

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

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

Residual connection

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

- Preserves original information
- Requires matching dimensions

Layer normalization

Applied after the residual sum, per token i:

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

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

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

- 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

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.

Logits convert hidden representations into token probabilities.

Cross-Entropy Loss for Language Modeling

Prediction target

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

For position t: $y_t \in \{1, \ldots, V\}$ (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:

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

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" $\longrightarrow$ [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: $(x_1, \ldots, x_{n-1})$ $\rightarrow$ Target: $(x_1, \ldots, x_n)$

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)

```
class SelfAttention(nn.Module):
    def __init__(self, d_m, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.num_heads = num_heads
        self.d_k = d_m // num_heads

        self.q_linear = nn.Linear(d_m, d_m)
        self.k_linear = nn.Linear(d_m, d_m)
        self.v_linear = nn.Linear(d_m, d_m)
        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)

```
criterion = CrossEntropyLoss(
    ignore_index=0)
optimizer = Adam(
    model.parameters(), lr=1e-3)

for x, y in dataloader:
    optimizer.zero_grad()
    logits = model(x)
    loss = criterion(
        logits.view(-1, V), y.view(-1))
    loss.backward()
    optimizer.step()
```

Gradients flow through the entire Transformer.

Training is standard gradient-based optimization.

What the Model Predicts

Given input

"I am" $\longrightarrow$ [1, 2]

Model output

- Probability over vocabulary
- One distribution per position

Prediction

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

Inference (greedy decoding)

```
model.eval()
x = [I, am]

logits = model(x)
next = argmax( logits[-1] )

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)

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

Verify installation

```
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

```
ollama pull mistral
ollama run mistral
```

Example prompt

```
> 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.

>>> Send 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

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

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

```
curl -X POST http://localhost:11434/api/
    generate \
-H "Content-Type: application/json" \
-d '{
  "model": "mistral",
  "prompt": "Explain self-attention.",
  "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":fal
{"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":fa
{"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)

```python
import requests, json
url = "http://localhost:11434/api/generate"
data = {
  "model": "mistral",
  "prompt": "Explain self-attention.",
  "stream": True }
r = requests.post(url, json=data, stream=True)
for line in r.iter_lines():
    if line:
        msg = json.loads(line)
        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

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

Open the UI in your browser

Access the interface

```
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.