



Python and Machine Learning for Weather, Climate and Environment

Tutorial and Guide

by

EUMETNET E-AI Programme,
ECMWF, University of Reading, Deutscher Wetterdienst

Roland Potthast

with contributions by Stefanie Hollborn, Jan Keller, Thomas Deppisch,
Mareike Burba, Matthias Mages, Sarah Heibutzki, Marek Jacob, Florian
Prill, Tobias Göcke, Felix Fundel

January 16, 2026

Table of Contents

Introduction	i
I Programming and working with Data in the Time of AI	i
II General Coding Rules and Strategy	iv
III 5 Days Python and AI	xii
<hr/>	
Day 1: Python Workhorse and AI/ML	1
1 Python Basics	1
1.1 Install, Virtual Environment, Pip und Import	1
1.2 Managing Dependencies with requirements.txt	3
1.3 Introduction to NumPy	7
1.4 Generating Plots based on Matplotlib	10
1.5 Functions	13
1.6 Python Essentials	14
2 Jupyter Notebooks, APIs and Servers	25
2.1 Introduction to Jupyter Notebooks	25
2.2 Introduction to APIs: A Key Principle in Code Development	32
2.3 Making API Requests with requests	34
2.4 Fortran Integration using ctypes as API	42
3 Eccodes for Grib, Opendata, NetCDF, Visualization	44
3.1 Downloading ICON Model GRIB Files from DWD Open Data Server	45
3.2 The Grib Library eccodes	47
3.3 Accessing SYNOP Observation Files from NetCDF	57
3.4 Analysing AIREP Feedback Files in NetCDF Format	60
4 Basics of Artificial Intelligence and Machine Learning (AI/ML)	72
4.1 AI and ML - Basic Ideas	72
4.2 Torch Tensors - Basics and Their Role in Minimization	75
4.3 PyTorch Fundamentals - Model, Loss, and Optimizer	76
4.4 Simple Neural Network Training Example	81
4.5 Gradient Field and Decision Boundary	86
<hr/>	
Day 2: AI/ML Basics, LLMs, Multi-Modality	90
5 Neural Network Architectures	91
5.1 Feed Forward Networks	91

5.2	Graph Neural Networks	96
5.3	Applying Convolutional Neural Networks for Function Classification	101
5.4	LSTM-Based Anomaly Detection in Sensor Data	106
6	Large Language Models	115
6.1	LLM Network as Sequence-to-Sequence Machines via Transformer Models	115
6.2	Implementing and Training a Simple Transformer-Based LLM	119
6.3	Install Your Own LLM, Chat with it and Develop Applications	124
7	LLM with Retrieval-Augmented Generation (RAG)	136
7.1	Preparing Documents	137
7.2	Generating Embeddings for Documents	139
7.3	Using an LLM Locally or with OpenAI for Response Generation	142
7.4	Saving and Reloading the Vector Database, Collecting Search Originals, Chunking long Documents	147
7.5	End-to-End AI-Powered Answer Pipeline	151
8	Multimodal LLMs	156
8.1	Fundamentals of Multimodal Large Language Models	156
8.2	Radar Data Access and AI Interpretation	160
8.3	Cloud Top Height as a Multimodal AI Application	164
Day 3: Diffusion, Agents, Feature Detection, MLflow		168
9	Diffusion and flexible Graph Networks	169
9.1	Diffusion Networks	169
9.2	Flexible Graph Networks for Learning from Sparse Observations	182
9.3	Exploring Graph Structures in Detail	189
9.4	PyTorch Lightning and PyTorch Geometric	194
10	Agents and Coding with LLM	200
10.1	Introduction to Automated Coding with LLMs	200
10.2	Survey of LLM-Based Agent Frameworks	204
10.3	Using LangChain for Code Design and Execution	206
10.4	LangGraph-Based Forecast Assistant	213
11	DAWID, LLMs and Feature Detection	220
11.1	The DAWID Frontend: Upload and Interaction Interface	220
11.2	DAWID Backend Architecture	224
11.3	AI based Feature Detection for Fronts	230
12	MLFlow – An easy way of Managing and Monitoring Training	237
12.1	Introduction to MLFlow	237
12.2	Logging ML Experiments with MLFlow	241
12.3	Running an MLFlow Server	245
12.4	Advanced Features and Model Management	250
Day 4: MLOps, CI/CD, Anemoi, AI Work		253

13 MLOps - Development and Operations Integrated	254
13.1 DevOps and MLOps – Foundations and Motivation	254
13.2 Containerization and Reproducibility	259
13.3 DevOps at DWD – Numerical Weather Prediction as a Forerunner	264
14 CI/CD – Continuous Integration and Deployment	270
14.1 Overview and Motivation	270
14.2 Tools and Frameworks for CI/CD	271
14.3 Testing with Pytest	276
14.4 CI/CD Runners and Cloud Integration	279
14.5 CI/CD for ICON, AICON, and Anemoi	280
15 Anemol – AI-Based Weather Modeling	281
15.1 Yaml, Hydra and OmegaConf	281
15.2 Introduction to Anemol	285
15.3 ZARR, ERA and Datasets for Anemoi	288
15.4 Building an Icosahedral Graph with Anemoi	293
15.5 Hands-On Datasets, Validation, Training With Anemoi	295
15.6 Training Pipeline in Anemoi	299
16 AI Transformation: From Communication to Neural Forecasting	303
16.1 From Communication History to AI Assistants (1440–2022)	303
16.2 What an LLM is: tokens, attention, and next-token training	305
16.3 From Chat to Systems: tools, retrieval (RAG), and agents	307
16.4 Neural Forecasting: learning motion, fronts, and rollouts	308
16.5 CNN Translation Toy World: features, loss, and why it works	311
16.6 Weather Service Perspective: products, services, and operational ecosystems	316
16.7 Outlook: transformation of workflows and service building	319
Day 5: AI Models, Physics, Observation-driven Learning	320
17 Model Emulators, AIFS and AICON	322
17.1 Model emulators based on Anemoi	322
17.2 AIFS	324
17.3 AICON	324
17.4 Running AICON inference at DWD	332
17.5 Further activity: Anemoi vs FRAIM	336
18 AI Data Assimilation	340
18.1 Introduction to AI-based Data Assimilation	340
18.2 Worked Example: 1D Inversion and AI-Var Learning	343
18.3 AI Particle Filter (AIPF): Learning a Posterior Particle Distribution	353
19 AI and Physics and Data	360
19.1 Physics-Informed Neural Networks (PINNs)	360
19.2 Discovering Governing Equations from Data (SINDy)	369
19.3 Learning the Force Term: Neural RHS and Hybrid Dynamics	376
19.4 Physics constraints in neural emulators	384
19.5 Causal Modeling with Neural Networks	392

20 Learning from Observations Only	398
20.1 Obs-to-Obs Learning on a 2D Toy Atmosphere	398
Appendix	425
21 History of Large Language Models	431
21.1 The History of Large Language Models	431
21.2 The Georgetown-IBM Experiment of 1954	433
21.3 ELIZA — The First Chatbot in History	434
21.4 Probabilistic Models in Language Processing in the 1990s	436
21.5 The Rise of Neural Networks from 2000	437
21.6 The Vector Representation of Language and Its Significance	439
21.7 The Transformer Revolution (2010–2020): The Rise of Modern Language Models	440
21.8 The GPT Revolution from 2020: Artificial Intelligence at the Next Level	441
21.9 AI Agents: Autonomous Systems of the Future	443

Introduction

I Programming and working with Data in the Time of AI

I.1 Integrating AI into Forecasting and Modeling: A Strategic Transformation

Artificial Intelligence (AI) is not just an enhancement to traditional forecasting and modeling—it is increasingly becoming a *core component* of next-generation weather and climate observation processing, prediction systems, products and services. Some traditional methods will be replaced by AI-driven approaches, while others will be hybridized with AI for better efficiency and accuracy. Others will stay as clear and targeted methods, but AI will help us to use them, to develop and to understand. AI can even serve as tool of discovery, e.g. AI enables *learning directly from observations*, either by improving data assimilation techniques or solving the data to forecasting task by end-to-end learning.

This section presents a structured transformation strategy for adopting AI-based forecasting, model development, and service automation.

Building AI Expertise with Python and AI Workflows

To effectively integrate AI, we must ensure our teams are skilled in both AI methods and operational workflows. Our approach includes:

- Establishing structured learning paths for key AI techniques relevant to *weather and climate modeling*.
- Using Python libraries such as *numpy*, *eccodes*, *netcdf*, and *xarray* for handling large meteorological datasets.
- Training teams in machine learning frameworks such as *TensorFlow*, *PyTorch*, and *Hugging Face Transformers*.
- Setting up *end-to-end AI workflows* in Jupyter-based environments, covering data ingestion, training, validation, and inference.
- Encouraging collaboration between *meteorologists*, *model developers*, and *AI experts* to foster cross-disciplinary innovation.

Replacing and Hybridizing Forecasting Systems with AI

Some forecasting components will be fully *replaced by AI*, while others will integrate AI as a *hybrid solution*. Key shifts include:

- *AI-Based Nowcasting*: AI-driven short-term weather predictions using real-time observational data (e.g., satellite, radar, sensors), enhancing or replacing conventional nowcasting techniques.
- *Neural Weather Models*: Deep learning models trained on historical data can generate competitive forecasts at lower computational cost.
- *Hybrid AI-NWP Models*: AI enhances physics-based forecasting through bias correction, uncertainty quantification, and ensemble optimization.
- *Machine Learning for Subgrid Processes*: AI improves or replaces empirical parameterizations in turbulence, cloud physics, and convection models.
- *Automated Impact Forecasting*: AI-driven models provide direct risk assessments for extreme weather events, minimizing reliance on manual interpretation.

AI Forecasting and AI Data Assimilation: New Core Components in the Model Chain

Traditional numerical weather prediction (NWP) relies on physics-based models, but AI is rapidly becoming an integral part of the *full model chain*, improving both forecasting efficiency and data assimilation.

AI-Based Forecasting AI-driven forecasting models are evolving as viable alternatives and enhancements to traditional numerical methods:

- *AI-Based Nowcasting*: Rapid, high-resolution short-term forecasting from observational data, improving local prediction accuracy.
- *Neural Weather Models*: Machine learning models that approximate NWP output with lower computational requirements.
- *Hybrid AI-NWP Models*: AI refining traditional numerical forecasts, enhancing post-processing and uncertainty quantification.

AI in Data Assimilation and Learning Directly from Observations AI is transforming data assimilation, which is essential for initializing forecasts:

- *Machine Learning for Observation Processing*: AI-driven quality control of observational data, filling data gaps and detecting sensor anomalies.
- *AI-Based Data Assimilation*: AI improving assimilation processes by optimizing observation ingestion.
- *Deep Learning for Data Assimilation*: AI learning complex relationships between observations and model states, accelerating assimilation workflows.
- *End-to-End AI Data Ingestion*: Future AI models trained directly on observational datasets, potentially reducing reliance on classical assimilation techniques.

- *Self-Learning Systems*: AI dynamically adjusting to new data, improving continuously without manual recalibration.

Using AI for Code Refactoring and Model Development

AI also modernizes modeling workflows, improving efficiency in research and development:

- *Refactoring Legacy Code*: AI-assisted tools improving *Fortran, C++, and Python* models for better maintainability and performance.
- *Automated Model Optimization*: AI tuning hyperparameters and optimizing computational performance.
- *AI-Assisted Scientific Discovery*: AI identifying new climate and weather patterns in large datasets.
- *AI-Generated Documentation and Testing*: Automating documentation and generating validation tests for numerical models.

Transforming Services and User Interaction with AI

AI enables new ways to deliver weather and climate services, improving *automation, personalization, and accessibility*:

- *AI-Generated Weather Reports*: Natural language generation models translating raw data into meaningful insights for different user groups.
- *Conversational Forecasting Assistants*: AI chatbots and voice assistants allowing users to interactively query weather and climate predictions.
- *Real-Time Impact Forecasting*: AI models directly linking weather forecasts to risks in agriculture, energy, transportation, and disaster management.
- *AI-Powered Data Visualization*: Interactive AI tools allowing users to explore, manipulate, and interpret complex weather datasets.

A Clear Migration Strategy for AI Transformation

To successfully integrate AI while maintaining operational stability, we adopt a *structured migration strategy*:

1. *AI Readiness Assessment*: Identify areas where AI provides the highest impact while ensuring compatibility with existing workflows.
2. *Pilot AI Replacements*: Test AI-based forecasting models in parallel with traditional methods before full adoption.
3. *Hybrid Deployment Strategy*: Introduce AI-driven improvements in *stages*, ensuring fallback options are in place.
4. *AI Validation and Trust Building*: Develop transparent evaluation metrics for AI models to ensure trust and reliability.

5. *Workforce Training and Knowledge Transfer*: Enable teams to transition smoothly from traditional methods to AI-driven solutions.
6. *Continuous AI Governance*: Establish guidelines for AI model retraining, performance monitoring, and ethical considerations.

Limitations and Responsible Use of AI

While AI offers transformative opportunities in forecasting, modeling, and service delivery, it is crucial to acknowledge its current limitations and apply it with scientific caution:

- *Data Requirements*: Most AI models rely on large, high-quality datasets and perform poorly in data-sparse or non-stationary environments.
- *Lack of Physical Consistency*: AI predictions may violate conservation laws or produce unrealistic results in rapidly evolving scenarios.
- *Limited Interpretability*: Unlike traditional models, many AI systems operate as black boxes, making it difficult to understand or trace their internal reasoning.
- *Bias and Overfitting*: Biased or unbalanced training data can lead to flawed predictions, while overfitting to historical data may reduce adaptability to changing climate conditions.
- *Need for Rigorous Validation*: AI models must be continuously monitored, validated, and benchmarked to ensure stability, fairness, and scientific reliability. Validation needs metrics and scores beyond traditional forecasting metrics.
- *Complementary Role*: AI should be seen as an enhancement to—not a replacement for—physics-based modeling, supporting a hybrid approach for trustworthy innovation.

By following this transformation roadmap, we ensure that AI adoption is *structured, scalable, and scientifically sound*, positioning our forecasting and modeling systems for the future.

II General Coding Rules and Strategy

Our coding principles focus on *Maintainability, Testability, and Automation*. Code should be structured, tested, and versioned properly, ensuring long-term reliability and ease of collaboration.

II.1 Code Management with Git

All code is managed in Git, following a structured workflow:

- *Repository Structure*: Organize code into well-defined modules, using a clear folder structure (src/, tests/, docs/).
- *Branching Strategy*: Use a master/dev/feature branching model:
 - master: Production-ready, thoroughly tested.
 - dev: Integration branch for new features.
 - feature/*: Short-lived branches for individual tasks, merged via pull requests.

- *Commits and Documentation:*
 - Each commit should contain *atomic changes* with clear commit messages (`git commit -m "Refactored data pipeline for efficiency"`).
 - You might use Git hooks for enforcing style checks (e.g., `pre-commit` for black and `flake8`).

Essential Git Commands and Best Practices

Git is a distributed version control system, enabling efficient collaboration. The following commands cover the most common workflows. We usually employ git via gitlab or github, but you can use it yourself on any linux system, letting your own repo work as a server for you, and do git add/commit/push as with some gitlab installation!

Initializing and Cloning Repositories

```
git init # Initialize a new Git repository
git clone <repo-url> # Clone an existing repository
```

Working with Branches

```
git branch feature-xyz          # Create a new branch
git checkout feature-xyz        # Switch to a branch
git switch -c feature-xyz      # Create and switch to a branch (newer Git versions)
git checkout -b mylocalname origin/reponame # Track a remote branch with a local name
git merge feature-xyz          # Merge changes into the current branch
git rebase main                 # Reapply commits on top of the main branch
```

Committing and Pushing Changes

```
git status # Show modified files
git add . # Stage all changes
git commit -m "Describe your change" # Commit changes
git push origin feature-xyz # Push changes to the remote repository
```

Syncing and Undoing Changes

```
git pull origin main # Update the local branch with remote changes
git reset --hard HEAD~1 # Undo the last commit
git checkout -- <file> # Revert changes to a file before commit
```

Tracking and Reviewing History

```
git log --oneline --graph --decorate # View commit history
git diff # Show uncommitted changes
git blame <file> # Show line-by-line history of changes
```

Best Practices for Git Usage

- *Commit frequently*: Avoid large, monolithic commits.
- *Write meaningful commit messages*: Summarize what and why, not just how.
- *Rebase instead of merge (when appropriate)*: Keeps history linear.
- *Use .gitignore*: Prevent unnecessary files from being tracked.
- *Pull before pushing*: Avoid conflicts by updating from the remote branch.
- *Tag important versions*: Use git tag v1.0 for release milestones.

Adding a .gitignore File for LaTeX Projects

Before committing files to a Git repository, it's important to add a .gitignore file to prevent cluttering the version history for example with automatically generated LaTeX files. These include temporary files, auxiliary logs, and build artifacts that should not be tracked. Here's a recommended .gitignore for LaTeX projects:

```
# LaTeX intermediate and output files
*.aux
*.bb1
*.blg
*.brf
*.fdb_latexmk
*.fls
*.idx
*.ilg
*.ind
*.lof
*.log
*.lot
*.nav
*.out
*.pdf
*.snm
*.synctex.gz
*.toc
*.vrb
*.xdv

# Editor backup files
*~
*.swp
.DS_Store
```

This ensures that only the actual source files (e.g., .tex, .bib, .sty, images, and configuration files) are tracked in your repository.

Best Practices for Repository Management

- *Do not commit large binaries or large images into a GitLab or any other code repository!*
- *Keep project-related materials (e.g., PowerPoint presentations) in separate repositories from code development!*

Usually, it is a good idea to have your project repo with branches in a place where storage limitations are not important. You can use a local git repo to make sure your own versions on different computers are well synchronized, and clone and push into a central space on some linux server. Use gitlab or github to manage code repos.

- *Do not store measurement or model data in a Git repository!*
- Manage data instead in accessible folders with a clear and documented structure, or store it in a database environment.

II.2 Managing Multiple Git Repositories for AI Development

AI-based applications for weather, climate, and environmental forecasting often involve multiple interconnected repositories, such as core models, data pipelines, and frontend applications. Efficiently managing and synchronizing these repositories is essential, especially in large-scale initiatives like the *EUMETNET E-AI Programme on Artificial Intelligence and Machine Learning for Weather, Climate, and Environmental Applications* or the *DWD AI Center*. When we want to work both with e.g. DWD repos, MeteoFrance repos, Anemol repos and your local repos, a careful repo management is in order.

To facilitate multi-repo workflows, we either use lean and elementary scripts such as `gitall` or we employ a combination of `meta` (for managing multiple repositories as a single unit) and `git worktree` (for handling multiple branches efficiently).

Elementary Git Repository Management Script To streamline the handling of multiple Git repositories in a single parent folder, we provide the script `gitall.sh` (located in `./scripts`). Its key features and usage recommendations are:

- *Location:* The script is located in the `./scripts` directory.
- *Function:* It checks the status of all Git repositories or updates them with `git pull` commands.
- *Usage examples:*
 - `./gitall.sh pull` – Pulls the latest changes in all repositories.
 - `./gitall.sh s` – Shows the Git status for all repositories.
 - `./gitall.sh pull s` – Pulls and then shows the status.
- *Recommendation:* Create a symbolic link (e.g., `ln -s ./scripts/gitall.sh /bin/gitall`) to call it globally.
- *Documentation:* Detailed usage information is included as comments at the top of the script.

The script is easily extensible to your particular needs.

Organizing Repositories with Meta The package `meta` enables structured management of multiple repositories by defining them in a single `meta.json` file. This allows users to clone, update, and execute commands across all repositories in a unified manner.

```
{
  "projects": {
    "eai-tutorials": "https://github.com/eumetnet-e-ai/tutorials.git",
    "eai-toolbox-explore": "https://gitlab.dkrz.de/dwd-ki-zentrum/\\
      infrastructure/e-ai_toolbox_explore"
  }
}
```

With this setup, all repositories can be cloned simultaneously using:

```
meta git clone
```

Efficient Branch Management with Git Worktree

For AI research and development, multiple experiments and feature branches often need to be managed simultaneously. Instead of constantly switching branches, `git worktree` allows separate working directories for each branch.

To create worktrees for feature branches:

```
meta exec "git worktree add ../feature-dwd-ai feature-branch"
meta exec "git worktree add ../feature-eai-pipeline feature-branch"
```

This approach provides a clean way to work on different tasks in parallel without redundant repository clones.

Automation and Best Practices To ensure consistency in AI workflows:

- Pull updates across repositories with:

```
meta exec "git pull origin main"
```

- Regularly clean up unused worktrees:

```
meta exec "git worktree prune"
```

- Store `meta.json` in version control to ensure reproducibility across teams.
- Automate workflows via CI/CD pipelines to test and deploy AI models across repositories.

By integrating `meta` and `git worktree`, the DWD AI Center and EUMETNET E-AI Programme can maintain a structured and efficient workflow, allowing researchers and engineers to focus on AI model development rather than repository management.

II.3 Testing and Continuous Integration

To maintain quality, every function and module should have explicit *unit tests*, written with pytest. We encourage a *test-driven development (TDD)* approach where applicable.

- *Unit Testing*: Every function should be covered by a test to catch potential bugs early.
- *Integration Testing*: Ensure different components interact correctly, particularly in model training and evaluation pipelines.
- *Automated Testing*:
 - Git push should be able to trigger a *CI/CD pipeline* that runs all tests.
 - Developers should be able to run pytest locally before committing changes.
 - Centralized CI testing (e.g., GitHub Actions, GitLab CI/CD, Jenkins) should validate code before deployment.

II.4 Code Quality and Documentation

Maintaining high code readability and quality is essential. The following standards apply:

- *Code Formatting*: Enforce coding standards using black (formatting), flake8 (linting), and mypy (type checking).
- *Pre-Commit Hooks*: Automate checks to prevent incorrect code from being committed.
- *Documentation*:
 - Every function and class should include *docstrings* in Google or NumPy format.
 - API documentation should be maintained using mkdocs or Sphinx.

II.5 Reproducibility and Environment Management

To ensure consistent execution across different systems:

- *Dependency Management*:
 - Use virtual environments such as venv, Poetry, or Pipenv for managing dependencies.
 - Store dependencies explicitly in pyproject.toml (Poetry) or requirements.txt (pip).
- *Containerization*:
 - Use Docker to provide isolated, reproducible environments.
 - Maintain *pre-configured environments* for development and production.

II.6 Automated Workflows and Continuous Deployment

Automation is key to ensuring reliability and scalability:

- *CI/CD Pipelines*: Tests, builds, and deployments should at least in principle be fully automated, meaning that necessary human based evaluation and reviews are built into an automated process which is thoroughly tested.
- *Model Retraining and Monitoring*: Machine learning models will probably be retrained on a regular basis, with performance monitoring to track drift.
- *Version Control for Models*: Every trained model should be versioned for reproducibility.

We remark that most of these steps are standard in the framework of *Numerical Weather Prediction*, where code management for models and data assimilation code has a long tradition. Also, running a very organized process including

- **code development**,
- small and large-scale **testing**,
- **evaluation** and **verification**,
- **decision making** based on the results and
- deployment through an operational system with **parallel routine** and
- **routine** runs based on ecflow schedulers

is best-practice in NWP centres. At DWD, we run a weekly *routine meeting* where evaluation results and decisions are made for the NWP forecasting system. Scripting systems for full-scale NWP experiments are available both for fast development and routine-type testing and deployment. AI applications can seamlessly be integrated into this approach, which guarantees quality assurance and flexibility at the same time.

II.7 Basic Coding Principles

- Use meaningful variable names and keep functions concise.
- Follow PEP8 for consistent Python code style.
- Avoid hardcoded values; instead, use configuration files.
- Ensure all functions include a well-defined docstring.

II.8 Git and Collaboration Best Practices

To maintain a clean and organized codebase:

- Follow a structured Git workflow using `main`, `dev`, and `feature/*` branches.
- Ensure commits are atomic, with a clear description of changes.
- Use Git hooks to enforce style checks automatically before committing.

II.9 Testing and Quality Assurance

Before merging code:

- Every function should have a corresponding unit test.
- Tests should be run locally before pushing changes to Git.
- CI/CD pipelines must validate all code changes with unit and integration tests.

II.10 Deployment and Reproducibility

To ensure stable production releases:

- No code is merged into master unless all tests pass.
- Machine learning models should be retrained periodically with performance monitoring.
- Environments should be reproducible using virtual environments with requirements files or Docker containers.

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				

Figure 1: Five days Python and AI.

Structure of E-AI Activity and the DWD AI Centre

We note that the Structure of E-AI Activity as well as the DWD AI Centre as shown in Figure 2 is highly dynamic, with projects and repos being in a process of setup and consolidation. The following

graphics provides a snapshot and example layout as currently pursued.

- The DWD AI Centre connects internal development units, contributors, and public users to shared infrastructure and applications.
- The repository structure is grouped into *Apps*, *Infrastructure*, some including *Externals* such as Anemoi or mfa1 or MLCast, each containing specific projects for modeling, processing, and integration.
- The central node acts as the coordination hub, linking various AI-driven tools with operational workflows and collaborative partners.

III 5 Days Python and AI

III.1 Schedule

The following table provides an overview of the tutorial structure, covering key topics in Python programming, artificial intelligence, and machine learning for applications in weather, climate, and environmental sciences. The tutorial is designed as a structured five-day course, with each day focusing on specific themes. The content progresses from fundamental Python concepts and data handling to advanced AI techniques such as large language models (LLMs), retrieval-augmented generation (RAG), and AI-driven forecasting. We introduce many practical applications, and advanced topics including AI data assimilation, model emulation, and AI-enhanced operational workflows. Each day consists of multiple modules, ensuring a comprehensive and hands-on learning experience.

III.2 Training Codes

To ensure a structured and reproducible learning experience, all training codes are provided *chapter by chapter*. This should allow participants to *easily locate, reference, and execute* the relevant scripts corresponding to specific tutorial sections.

We have tested the scripts as far as possible for the following computing environments and give specific advice when things were difficult in a particular framework.

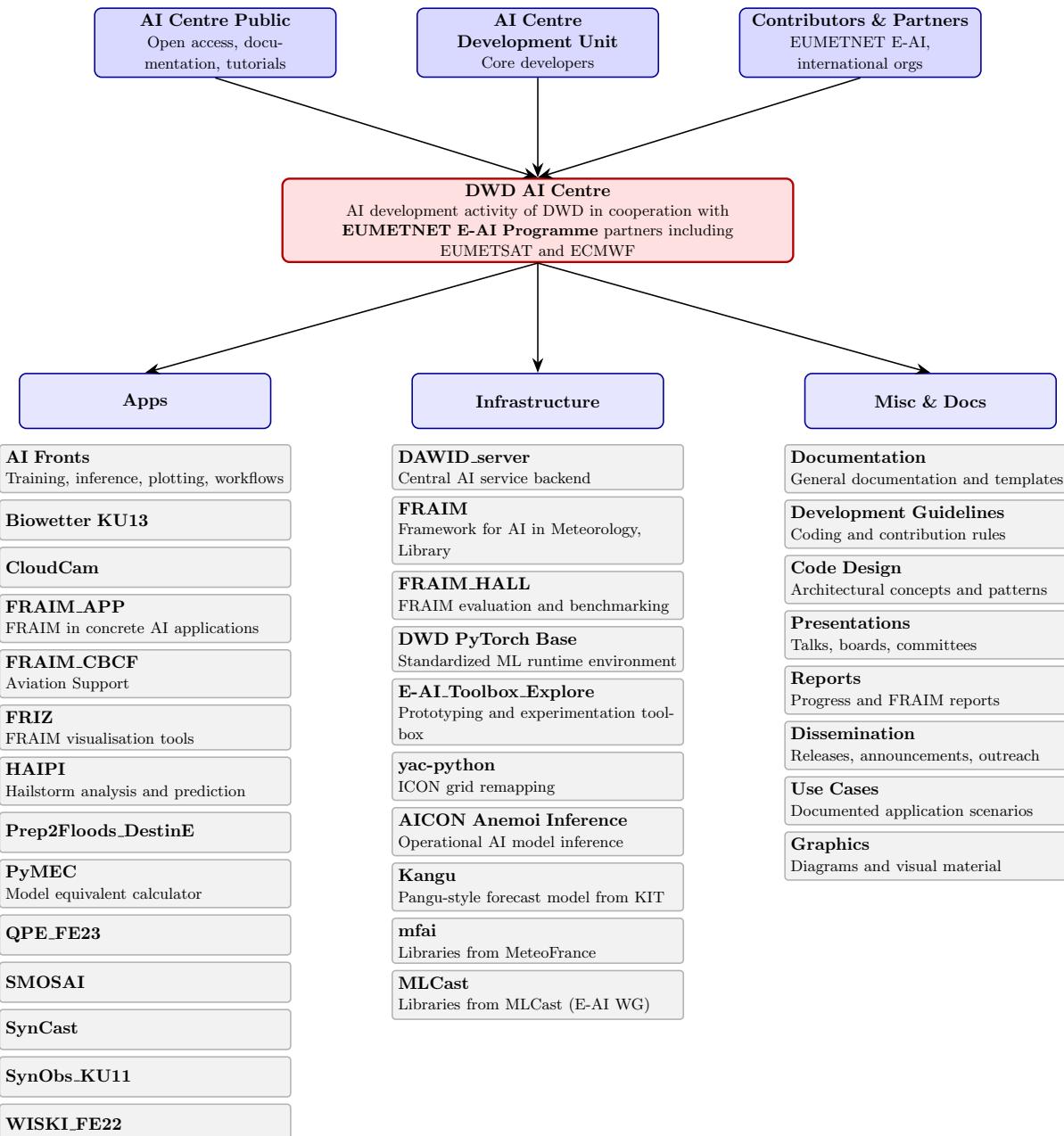


Figure 2: Organizational and technical structure of the DWD AI Centre repositories.

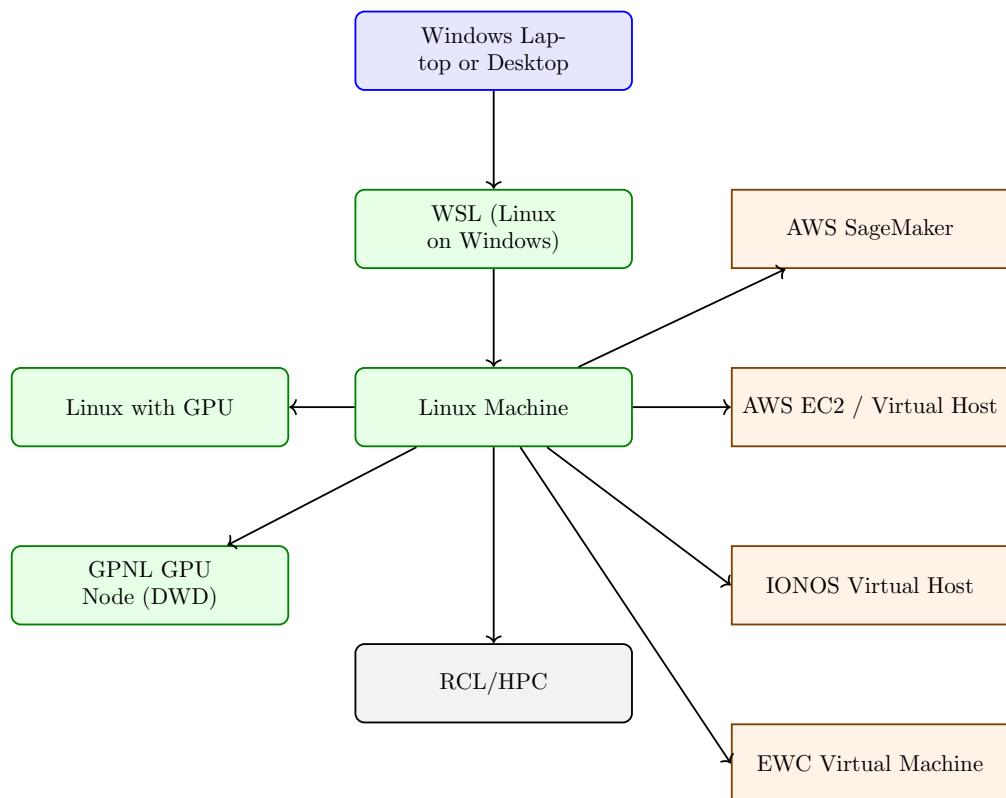


Figure 3: We want to enable choices and independence of particular solutions or infrastructures.

Chapter 1

Python Basics

We do not want to provide a full python tutorial, but rather formulate a guide through main steps and a setup which is very flexible to work with python and machine learning for weather, climate and environment.

Our goal is to enable our scientists and developers to use python and machine learning in a flexible, modular and portable way for their development, for science as well as for products and services of various types. On the basis of python in combination with large language models we will touch the full workflow for science, development, product design and deployment.

1.1 Install, Virtual Environment, Pip und Import

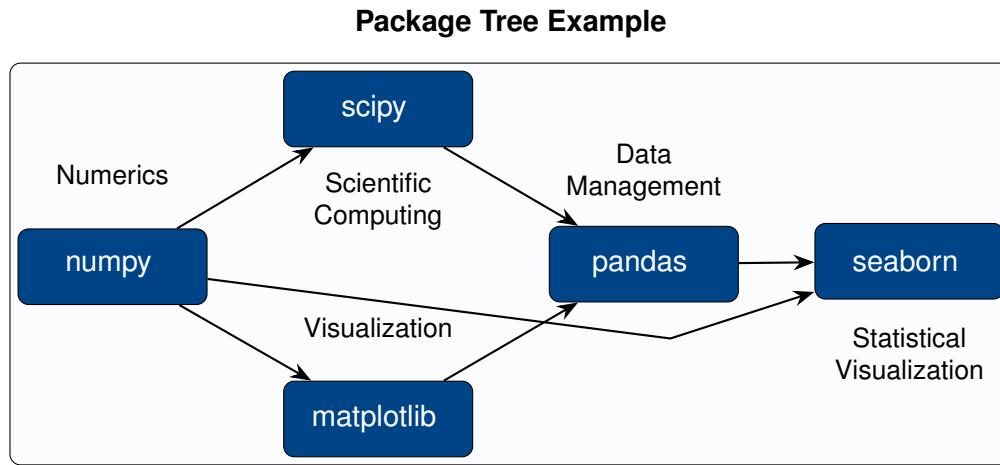
1.1.1 Install and Virtual Environment

Before running Python commands, ensure you have Python installed. It is very easy to have Python installed on your laptop by simply downloading it - this is often possible without administration rights. You will then need to set the path variable properly. On Linux, depending on the version installed by your system administrator, you may find the executable under `python`, `python3`, `python3.11` or `python3.12`. We recommend not working with versions earlier than Python 3.10 because you may run into compatibility issues; instead, make sure you have an up-to-date Python version installed. Test the installed version by:

Test Python Installation

```
1 python --version
```

Python has become one of the most popular programming languages, largely because of its extensive ecosystem of packages and libraries. From data analysis and visualization to machine learning and web development, Python's modular design allows you to choose only the components you need.



It is very important to learn to manage packages to build a robust and tailored development environment. Learning how to install, manage, and create packages not only gives you a deeper understanding of the available tools, but also grants you greater control over your projects.

Usually, it is very important for a particular Python environment to provide a complete list of the packages it needs, with their versions, in a consistent framework. This framework is provided by virtual environments.

A virtual environment allows you to control your package installations. Below are example commands for both Windows and Linux:

Windows Command

```

1 python -m venv myenv
2 myenv\Scripts\activate
  
```

Linux Command

```

1 python3 -m venv myenv
2 source myenv/bin/activate
  
```

This will create a `myenv` directory with the default venv file structure. `myenv` is the freely choosable name of the venv.

1.1.2 Using pip to Manage Python Packages

`pip` is the package installer for Python. It allows you to install, update, and manage packages from the Python Package Index (PyPI). For example, you can check the version of `pip`, list installed packages, and install popular packages like `numpy` and `matplotlib`. The following commands illustrate these basic operations:

Basic pip Commands

```

1 pip --version
  
```

```
2 pip list
3 pip install numpy
4 pip install matplotlib
```

These commands, when run in your command prompt or terminal, will display the current version of pip, show all installed Python packages, and install numpy and matplotlib, respectively.

One of the most widely used libraries in Python is NumPy, which provides powerful array objects and routines for fast numerical computation.

1.2 Managing Dependencies with requirements.txt

Managing dependencies is crucial in Python projects, especially when working in different environments or collaborating with others. The requirements.txt file allows you to list all your project's dependencies and their versions, making it easy to replicate the environment anywhere.

Generating a requirements.txt

If you already have a virtual environment set up and want to generate a requirements.txt file from it, first activate your current virtual environment. Once the virtual environment is activated, run the following command to generate the requirements.txt file:

Generate requirements.txt

```
1 pip freeze > requirements.txt
```

This creates a file named requirements.txt in your current working directory containing all installed packages and their versions, leading to e.g. the following simple requirements.txt file.

Requirements.txt example

```
1 numpy==1.26.4
2 ollama==0.3.1
3 openai==1.69.0
4 openai-whisper==20240930
5 toml==0.10.2
6 torch==2.4.0
7 torch_geometric==2.5.3
8 torchmetrics==1.4.1
9 torchvision==0.19.0
```

Installing Dependencies from requirements.txt

To install all dependencies from an existing requirements.txt file into a new virtual environment, first create and activate the environment as shown in Section 1.1.1. Then, run the following

command:

Install Dependencies from requirements.txt

```
1 pip install -r requirements.txt
```

This installs all packages listed in the `requirements.txt` file, ensuring that your environment matches the specified dependencies.

With these steps, you can easily share and reproduce Python environments using `requirements.txt`.

1.2.1 Importing Functions or Packages vs. Installation

In Python, you can either directly import functions and modules from local files or install packages to make them globally available across projects. Understanding the difference is essential for maintaining clean and scalable code.

Importing Functions or Packages

You can import Python modules or functions directly from local `.py` files. For example, if you have the following structure:

```
|  
|-- main_greetings.py  
|-- greetings.py
```

In `main_greetings.py`, you can import from `greetings.py` as follows:

main_greetings.py

```
1 # main_greetings.py  
2 from greetings import say_hello  
3  
4 say_hello()
```

This method is quick and easy for small projects but becomes difficult to manage as your codebase grows or when sharing across multiple projects. You should then create installable packages, we discuss in a moment.

The `importlib` Package in Python

The `importlib` package allows you to reload Python modules without restarting the interpreter, which is especially useful during development when modifying code. Normally, Python imports a module only once per session, but `importlib.reload(module)` forces the interpreter to reload it, reflecting any recent changes. This is particularly handy in interactive environments like Jupyter Notebooks, where you want to see immediate updates after editing a module without restarting the entire session.

reload_demo_fkt.py

```

1 # reload_demo_fkt.py
2
3 def greet(name):
4     return f"Hello {name}"

```

And now lets load it, then change the file and reload it.

reload_demo.py

```

1 # reload_demo.py
2
3 import importlib
4 import reload_demo_fkt as mo
5
6 def replace_in_file(str1, str2, filename):
7     with open(filename, 'r') as file:
8         content = file.read()
9     content = content.replace(str1, str2)
10    with open(filename, 'w') as file:
11        file.write(content)
12
13 # Call the greet function initially
14 print(mo.greet("Roland")) # Expected: Hello Roland
15
16 # After modifying reload_demo_fkt.py, reload it
17 replace_in_file("Hello", "Good Morning", "reload_demo_fkt.py")
18
19 importlib.reload(mo)
20
21 # Call the updated greet function
22 print(mo.greet("Roland")) # Expected: Good Morning Roland
23
24 # Restore the original version of the file
25 replace_in_file("Good Morning", "Hello", "reload_demo_fkt.py")

```

Creating an Installable Python Package

An installable package allows you to reuse and share code easily across different environments. Consider the following structure:

```

install_demo02/
|-- pyproject.toml
|-- README.md
+++ src/
    --- install_demo02/
        |-- __init__.py
        |-- install_mod1.py

```

```
-- install_mod2.py
```

The project is defined in a `pyproject.toml` file, which can include a list of dependencies, that would replace the `requirements.txt`:

basic `pyproject.toml` file

```

1 [build-system]
2 requires = [ "setuptools>=61"]
3 build-backend = "setuptools.build_meta"
4
5 [project]
6 name = "install_demo02"      # name of the directory in src/
7 version = "0.1.3"
8 description = "A simple Python package with greeting functions"
9 authors = [
10   { name = "Roland Potthast", email = "Roland.Potthast@dwd.de" },
11 ]
12 requires-python = ">=3.8"
13 #dependencies = ["numpy<2", "matplotlib",]
14
15 [tool.setuptools.packages.find]
16 where = ["src"]
```

To install your package locally, in the code folder run:

Install Your Package

```
1 pip install -e install_demo02/
```

Once installed, you can import it in any project without reference to the location of the package, as in the file `install_demo_test_script.py`:

`install_demo_test_script.py`

```

1 from install_demo02 import greet1, greet2
2
3 print(greet1("World")) # Hello World!
4 print(greet2("World")) # Good Morning World!
```

Legacy projects with `setup.py`

Before `pyproject.toml` was invented, packages were defined in a `setup.py` file. One can find an example package using `setup.py` in the code/`install_demo` directory.

To make `setup.py` work on Windows, one might has to use the following steps.

```
pip install setuptools wheel
pip install -e install_demo/ --no-build-isolation --no-use-pep517
```

When to use each approach:

- *Pure Import*: Use for small, single-project code or quick prototypes.
- *Installable Package*: Use for larger projects, sharing code, and managing dependencies.

Example: Installing from GitHub

You can also install packages directly from Git repositories. For example:

Install from GitHub

```
1 pip install git+https://github.com/username/my_package.git
```

This installs your package from GitHub, making it easy to share code across teams and projects.

1.2.2 What is PyPI?

The Python Package Index (**PyPI**) is the official repository for third-party Python packages. It allows developers to:

- Upload and share their Python projects with the community.
- Install packages using the pip tool.
- Manage versions and dependencies of published packages.

When a user runs `pip install some-package`, pip connects to PyPI to find and download the corresponding package.

To make a project available on PyPI, developers package their code (typically using `pyproject.toml` and tools like `setuptools` or `flit`), build the distribution, and upload it using `twine`.

Uploaded packages are then publicly available for installation and reuse.

For more information, visit the official website:

<https://pypi.org>

1.3 Introduction to NumPy

NumPy is the fundamental package for numerical computing in Python. It provides the `ndarray`, a multidimensional array object that enables fast vectorized operations and efficient handling of large datasets. Although Python is known for its readability, NumPy's power lies in its ability to perform operations on entire arrays without writing explicit loops—a major benefit for programmers experienced in other languages.

In NumPy, the core building block is the `ndarray`. An `ndarray` can be created from a Python list (or nested lists for multidimensional arrays), and it supports element-wise operations. This vectorized computation model is not only more concise but also significantly faster for large-scale computations. Consider the following example:

Creating Basic Arrays

```

1 import numpy as np
2 arr1 = np.array([1, 2, 3, 4, 5])
3 print("1D array:", arr1)
4 arr2 = np.array([[1, 2, 3], [4, 5, 6]])
5 print("2D array:")
6 print(arr2)

```

The code above shows how to import NumPy (commonly aliased as np) and create both one-dimensional and two-dimensional arrays. Instead of writing loops to process elements, you can use array operations that are both elegant and efficient.

1.3.1 Vectorized Operations and Predefined Arrays

One of NumPy's most powerful features is vectorized operations. Instead of iterating over each element, you can perform operations on entire arrays with a single expression:

Vectorized Operations

```

1 import numpy as np
2 a = np.array([1, 2, 3, 4, 5])
3 b = np.array([10, 20, 30, 40, 50])
4 c = a + b
5 d = a * b
6 print("Addition:", c)
7 print("Multiplication:", d)

```

In addition to these operations, NumPy offers a variety of functions for creating arrays with predefined values. This is useful for initializing data or generating test datasets:

Predefined Arrays

```

1 import numpy as np
2 zeros = np.zeros((3, 4))
3 print("Zeros array:")
4 print(zeros)
5 ones = np.ones((2, 5))
6 print("Ones array:")
7 print(ones)
8 range_array = np.arange(0, 10, 2)
9 print("Range array:", range_array)
10 linspace_array = np.linspace(0, 1, 5)
11 print("Linspace array:", linspace_array)

```

1.3.2 Slicing, Indexing, and Broadcasting

NumPy arrays support powerful slicing and indexing methods, similar to Python lists but extended to multiple dimensions. This feature allows you to extract subarrays efficiently without copying the data:

Slicing and Indexing

```

1 import numpy as np
2 matrix = np.array([[ 1,  2,  3,  4],
3                    [ 5,  6,  7,  8],
4                    [ 9, 10, 11, 12],
5                    [13, 14, 15, 16]])
6 submatrix = matrix[1:4, 1:4]
7 print("Submatrix:")
8 print(submatrix)
9 element = matrix[1, 2]
10 print("Element at (2,3):", element)

```

Broadcasting allows operations between arrays of different shapes. With broadcasting, NumPy automatically expands the dimensions of an array during arithmetic operations:

Broadcasting Example

```

1 import numpy as np
2 mat = np.array([[1, 2, 3],
3                 [4, 5, 6],
4                 [7, 8, 9]])
5 vec = np.array([1, 0, -1])
6 result = mat - vec
7 print("Broadcasting result:")
8 print(result)

```

1.3.3 Mathematical Functions and Applications

NumPy offers a comprehensive suite of mathematical functions that operate element-wise on arrays. Whether you need trigonometric, logarithmic, or exponential functions, NumPy has you covered:

Mathematical Functions

```

1 import numpy as np
2 angles = np.linspace(0, np.pi, 5)
3 print("Angles:", angles)
4 sine_values = np.sin(angles)
5 print("Sine values:", sine_values)
6 exp_values = np.exp(np.array([0, 1, 2]))
7 print("Exponential values:", exp_values)

```

Using these functions, you can perform complex numerical computations with minimal code. For example, you might model a physical phenomenon or simulate data; NumPy's capabilities allow you to transform and analyze data efficiently. Experiment with these examples, and explore further functionalities of NumPy to fully leverage Python's capabilities in scientific computing.

Recommendation

Make sure you know your basic python commands well! Initially, do not rely only on sophisticated packages. Keep the core python level as your active knowledge!

1.4 Generating Plots based on Matplotlib

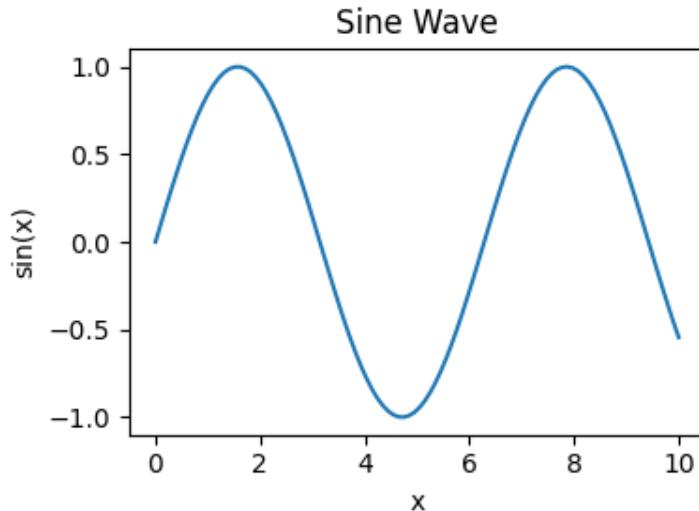
Paired with Matplotlib, a versatile plotting library, you can quickly visualize data and test your ideas.

1.4.1 1D Plots

The following example demonstrates how to use these libraries to plot a simple curve. In this code snippet, we generate a sine wave using NumPy and then plot it with Matplotlib.

plot-sine-wave.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Generate data
5 x = np.linspace(0, 10, 100)
6 y = np.sin(x)
7
8 # Create the plot
9 plt.figure(figsize=(4, 3))
10 plt.plot(x, y)
11 plt.xlabel('x')
12 plt.ylabel('sin(x)')
13 plt.title('Image generated by Simple Python Code')
14
15 # Save the plot as a PNG file
16 plt.savefig('images/plot-sine-wave.png')
17 plt.close() # Close the figure to free up memory
```



1.4.2 2D Plots

The following code demonstrates how to generate and visualize a two-dimensional field using NumPy and Matplotlib. First, a symmetric grid of x and y values is created and the radial distance from the origin is computed. Then, a Gaussian-modulated cosine function is used to define a smoothly varying field that decays with distance from the center. Finally, a filled contour plot is generated to visualize the field, and the resulting image is saved as a PNG file.

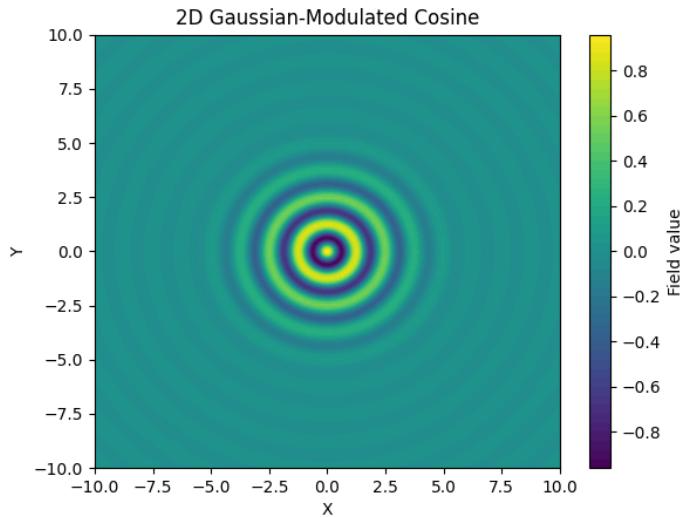
`plot-gaussian-modulated-cosine-field.py`

```

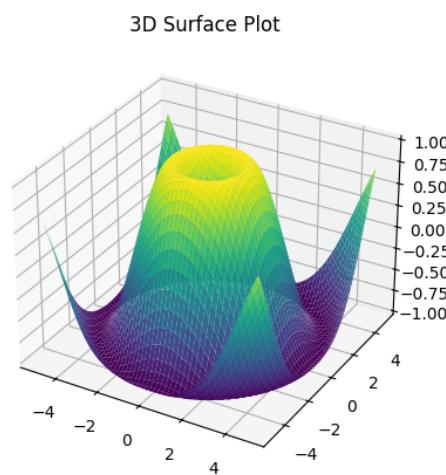
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Create a grid of x and y values (centered at 0 for a symmetric field)
5 x = np.linspace(-10, 10, 200)
6 y = np.linspace(-10, 10, 200)
7 X, Y = np.meshgrid(x, y)
8
9 # Compute the radial distance from the origin
10 R = np.sqrt(X**2 + Y**2)
11
12 # Define a Gaussian-modulated cosine field
13 Z = np.exp(-0.1*(X**2 + Y**2)) * np.cos(5*R)
14
15 # Create a filled contour plot for the 2D field
16 plt.figure(figsize=(4, 3))
17 contour = plt.contourf(X, Y, Z, levels=50, cmap='viridis')
18 plt.colorbar(contour, label='Field value')
19 plt.xlabel('X')
20 plt.ylabel('Y')
21 plt.title('2D Field Plot: Gaussian-Modulated Cosine')

```

```
22 plt.savefig('images/plot-gaussian-modulated-cosine-field.png')
23 plt.close()
```



The next example demonstrates how to create a simple 3D surface plot using Matplotlib's built-in `mpl_toolkit` toolkit. By generating a meshgrid of x and y values and computing a corresponding z value from a radial sine function, the plot visualizes a three-dimensional wave-like pattern. This technique provides a straightforward way to represent and explore three-dimensional data in Python.



1.5 Functions

Python functions are reusable blocks of code that allow you to encapsulate logic and perform specific tasks. In Python, functions are defined using the `def` keyword and can take parameters, return values, and include documentation. The following sections introduce the basics of defining and using functions in Python.

1.5.1 Defining a Function

Functions are defined with the `def` keyword followed by the function name, parentheses containing any parameters, and a colon. The function body is indented. Here is a basic example that defines a function to greet a user:

Defining a Function

```
1 def greet(name):
2     """Return a greeting message."""
3     return f"Hello, {name}!"
```

1.5.2 Calling a Function

Once a function is defined, you can call it by using its name followed by parentheses containing any required arguments. The following example shows how to call the `greet` function and print its result:

Calling a Function

```
1 message = greet("Alice")
2 print(message)
```

1.5.3 Functions with Multiple Parameters

A function can accept multiple parameters. Below is an example of a function that calculates the area of a rectangle:

Function with Multiple Parameters

```
1 def rectangle_area(width, height):
2     """Calculate the area of a rectangle."""
3     return width * height
4
5 area = rectangle_area(5, 3)
6 print("The area of the rectangle is:", area)
```

1.5.4 Default Parameter Values

Python functions can have default parameter values, which are used when an argument is not provided. This example demonstrates a function that computes a power, using a default exponent of 2:

Default Parameter Values

```

1 def power(number, exponent=2):
2     """Return number raised to the power of exponent."""
3     return number ** exponent
4
5 print(power(4))      # Uses default exponent 2 (result: 16)
6 print(power(2, 3))  # Exponent explicitly set to 3 (result: 8)

```

Variable-Length Arguments

Sometimes, you may not know in advance how many arguments a function should accept. Python allows you to capture additional positional arguments using the `*args` syntax. In the following example, a function computes the sum of an arbitrary number of numbers:

Variable-Length Arguments

```

1 def total_sum(*args):
2     """Return the sum of all provided arguments."""
3     return sum(args)
4
5 print(total_sum(1, 2, 3, 4, 5))  # Output: 15

```

1.6 Python Essentials

Let us look at a survey table what basic python knowledge you should gain in a first step. We have already gone over some significant part of this, and will briefly give you a head-start for the remaining points.

1.6.1 Control Flow in Python

Control flow in Python refers to the order in which individual statements, instructions, or function calls are executed. Python provides several structures for controlling the flow of your program, including conditionals, loops, and exception handling.

Conditional Statements

Conditional statements allow you to execute different code blocks based on certain conditions.

Topic	Description
Python Syntax	Basic structure, indentation, comments
Data Types	Integers, floats, strings, booleans, lists, tuples, sets, dictionaries
Control Flow	if-else, for and while loops, break, continue
Functions	Defining functions with def, arguments, return values, lambda functions
Modules and Imports	Importing built-in and external libraries, creating custom modules
File I/O	Reading and writing files, using with statements
Exception Handling	Using try-except for error handling
Object-Oriented Programming (OOP)	Classes, objects, inheritance, and polymorphism
Standard Libraries	Common libraries like os, sys, math, datetime, json
Virtual Environments	Creating and managing virtual environments with venv or conda

Table 1.1: Essential Topics for Basic Python Learning

Example:**If-Else Statements**

```

1 x = 10
2 if x > 0:
3     print("Positive")
4 elif x == 0:
5     print("Zero")
6 else:
7     print("Negative")

```

Loops

Loops allow you to repeat a block of code multiple times.

For Loop Example:**For Loop**

```

1 for i in range(5):
2     print(i)

```

While Loop Example:

While Loop

```

1 count = 0
2 while count < 5:
3     print(count)
4     count += 1

```

Loop Control Statements

Python provides special statements to control the flow inside loops:

- **break** – Exits the loop prematurely.
- **continue** – Skips the rest of the current iteration.
- **pass** – Does nothing and acts as a placeholder.

Example with break and continue:

Loop Control Example

```

1 for i in range(10):
2     if i == 3:
3         continue # Skip 3
4     if i == 7:
5         break # Stop loop at 7
6     print(i)

```

Exception Handling

Python allows you to handle errors using try-except blocks to prevent program crashes.

Example:

Try-Except Block

```

1 try:
2     result = 10 / 0
3 except ZeroDivisionError:
4     print("Cannot divide by zero!")

```

Summary: Control flow structures are essential for building logical and efficient Python programs, allowing you to make decisions, iterate over data, and handle errors gracefully.

1.6.2 File Input and Output in Python

Python provides built-in functions to handle files, making it easy to read from and write to files. This section covers the basics of File I/O operations.

Opening and Closing Files

To work with files, you need to open them first using the `open()` function and close them when done using `close()`.

Example:

Open and Close a File

```
1 file = open('example.txt', 'r') # Open in read mode
2 content = file.read() # Read the file content
3 file.close() # Close the file
```

Reading from Files

Python provides multiple methods to read file content:

- `read()` – Reads the entire file.
- `readline()` – Reads one line at a time.
- `readlines()` – Reads all lines into a list.

Example:

Reading from a File

```
1 with open('example.txt', 'r') as file:
2     for line in file:
3         print(line.strip())
```

Writing to Files

To write to a file, open it in write mode ('w') or append mode ('a').

Example:

Writing to a File

```
1 with open('output.txt', 'w') as file:
2     file.write('Hello, Python!\n')
3     file.write('This is a new line.')
```

File Modes in Python

- 'r' – Read mode (default).
- 'w' – Write mode (overwrites file).

- 'a' – Append mode.
- 'rb' – Read binary mode.
- 'wb' – Write binary mode.

Using the with Statement

The `with` statement simplifies file handling by automatically closing the file when the block is done.

Example:

Using the with Statement

```
1 with open('data.txt', 'r') as file:
2     data = file.read()
3     print(data)
```

Summary: File I/O in Python is straightforward and efficient, with built-in methods that handle files securely and reliably.

1.6.3 Common Python Libraries: os, sys, math, datetime, and json

Python's standard library provides a rich set of modules for everyday tasks. This section covers some of the most commonly used libraries.

os – Operating System Interface

The `os` module provides functions for interacting with the operating system, such as handling files, directories, and environment variables.

Example:

Using the os Module

```
1 import os
2
3 print(os.getcwd()) # Get current working directory
4 os.mkdir('new_folder') # Create a new folder
5 os.remove('file.txt') # Delete a file
```

sys – System-Specific Parameters

The `sys` module provides access to system-specific parameters and functions, such as command-line arguments and exiting the program.

Example:

Using the sys Module

```
1 import sys
2
3 print(sys.argv) # Command-line arguments
4 sys.exit(0) # Exit the program
```

math – Mathematical Functions

The `math` module offers mathematical functions such as trigonometry, logarithms, and factorials.

Example:

Using the math Module

```
1 import math
2
3 print(math.sqrt(16)) # Square root
4 print(math.pi) # Value of pi
5 print(math.factorial(5)) # Factorial of 5
```

datetime – Working with Dates and Times

The `datetime` module provides classes for working with dates and times, including formatting and arithmetic operations.

Example:

Using the datetime Module

```
1 from datetime import datetime
2
3 now = datetime.now()
4 print(now.strftime("%Y-%m-%d %H:%M:%S")) # Format current date and time
```

Dictionaries – Key-Value Data Structures

A dict in Python is an unordered collection of key-value pairs. Each key must be unique and immutable, and it maps to a corresponding value.

Example:

Using a Python Dictionary

```
1 data = {'name': 'Alice', 'age': 30}
2
3 print(data['name']) # Access value by key
```

```

4 data['age'] = 31          # Update value
5 data['city'] = 'Paris'    # Add new key-value pair
6
7 print(data)

```

Summary: Dictionaries are a powerful and flexible way to store structured data, enabling quick access and modification using keys. They are one of Python's most important built-in data types.

json – JSON Data Handling

The json module allows you to parse JSON data from strings or files and convert Python objects to JSON format.

Example:

Using the json Module

```

1 import json
2
3 data = {'name': 'Alice', 'age': 30}
4 json_string = json.dumps(data) # Convert to JSON string
5 print(json.loads(json_string)) # Convert JSON string to Python object

```

Summary: These libraries provide essential functions for system interaction, mathematical computations, date/time manipulation, and data serialization, making them fundamental for Python development.

1.6.4 Python Classes: Earth System Modeling Example

To demonstrate object-oriented programming in a scientific context, we implement a simple Earth System Model in Python. This example shows how classes can structure complex models by representing different components of the Earth system. It is just a dry demo - so no real earth system is run, but classes are called which could in principle run more complex applications. And you will see this syntax with `init` and `simulate` or `add_component` etc. in most AI/ML applications later.

Defining Earth System Components

We start by defining a common base class `EarthSystemComponent`, which represents a generic component of the Earth system. Each component is initialized with a name and exposes a `step()` method that advances its internal state by one time step. The base class itself does not implement any dynamics but enforces a common interface that all derived components must follow.

Concrete subclasses `Atmosphere`, `Ocean`, and `Land` extend this base class by introducing simple state variables and idealized evolution rules. The atmosphere stores a temperature, the ocean stores a sea surface temperature, and the land component stores a soil moisture fraction. Upon initialization, each component prints its initial state, making the model setup explicit and transparent.

Defining Earth System Components

```

1 class Atmosphere(EarthSystemComponent):
2     def __init__(self, name, temperature=288.0):
3         super().__init__(name)
4         self.temperature = temperature
5         print(f"[INIT] Atmosphere '{self.name}' with T={self.temperature:.2f} K")
6
7     def step(self, dt, forcing=None):
8         sst = forcing.get("sst", self.temperature)
9         self.temperature += dt * 0.01 * (sst - self.temperature)

```

Similar implementations are used for the ocean and land components, each with its own state variable and simple tendency equation. When an object of the class `Atmosphere` is created, Python automatically calls the special method `__init__`, which serves as the constructor. In this method, the call to `super().__init__(name)` initializes the part of the object that is defined in the base class `EarthSystemComponent`, in particular storing the component name. The atmosphere-specific constructor then defines the initial value of the state variable `temperature` and prints a diagnostic message. This explicit initialization output makes the model setup transparent and helps distinguish initialization from time integration.

The method `step()` defines how the atmospheric state evolves over one discrete time step. It is not called automatically but is invoked explicitly by the Earth system model during the simulation loop. The argument `dt` represents the time-step length, while `forcing` is a dictionary that provides coupling information from other components. In this case, the atmosphere reads the current sea surface temperature from the ocean and relaxes its own temperature toward that value. The updated temperature is stored in the object itself, meaning that the atmospheric state persists and evolves over successive calls to `step()`.

This design illustrates two fundamental concepts of object-oriented programming: encapsulation and polymorphism. Each component encapsulates its own state and time-evolution logic, while exposing a common interface (`step(dt, forcing)`) to the model controller. The Earth system model can therefore advance all components uniformly, without depending on the internal implementation details of individual components.

Building the Earth System Model

The class `EarthSystemModel` acts as the central controller of the simulation. It holds references to the atmosphere, ocean, and land components and is responsible for coordinating their interaction. The model maintains a simulation time and advances the system by repeatedly calling a `step()` method.

During each time step, the model explicitly exchanges information between components: the atmospheric temperature influences the ocean, and the sea surface temperature influences the atmosphere. These couplings are printed before each step, making the direction and strength of interactions visible in the console output. This separation between component dynamics and system-level coordination reflects the design principles of real Earth system models.

Earth System Model Class

```

1 class EarthSystemModel:
2     def __init__(self, atmosphere, ocean, land):
3         self.atmosphere = atmosphere
4         self.ocean = ocean
5         self.land = land
6         self.time = 0.0
7         print("[INIT] Earth System Model initialized")
8
9     def step(self, dt):
10        print(
11            f"[STEP] Advancing from t={self.time:.1f} to t={self.time + dt:.1f} | "
12            f"forcing: ATM<-SST={self.ocean.sst:.2f}, "
13            f"OCN<-T_atm={self.atmosphere.temperature:.2f}"
14        )
15
16        self.atmosphere.step(dt, forcing={"sst": self.ocean.sst})
17        self.ocean.step(dt, forcing={"atm_temp": self.atmosphere.temperature})
18        self.land.step(dt)
19        self.time += dt

```

In Python, the keyword `self` refers to the current instance of a class and is used to access data and methods associated with that specific object. Attributes such as `self.atmosphere`, `self.ocean`, and `self.land` store references to the component objects that were passed to the constructor when the Earth system model was created. This allows the model to persistently access and update the same component instances throughout the simulation.

Within the `step()` method, calls such as `self.atmosphere.step(...)` and `self.ocean.step(...)` invoke the corresponding update methods of the component objects. Because these components are stored as attributes of the model instance, their internal state is modified in place and retained across successive time steps. The variable `self.time` plays an analogous role for the model clock, ensuring that the simulation time advances consistently as the system evolves.

Running the Simulation

To run the simulation, the individual components are instantiated with simple initial conditions and passed to the Earth system model. The model is then advanced over a fixed number of time steps using a constant time increment. After each step, a compact diagnostic line is printed, showing the current simulation time and the evolving state variables.

Running the Earth System Simulation

```

1 atmosphere = Atmosphere("Global Atmosphere")
2 ocean = Ocean("Global Ocean")
3 land = Land("Global Land")
4

```

```
5 model = EarthSystemModel(atmosphere, ocean, land)
6 model.run(nsteps=20, dt=1.0)
```

Example output:

```
[INIT] Atmosphere 'Global Atmosphere' with T=288.00 K
[INIT] Ocean 'Global Ocean' with SST=290.00 K
[INIT] Land 'Global Land' with soil moisture=0.30
[INIT] Earth System Model initialized
[RUN] Starting simulation

[STEP] Advancing from t=0.0 to t=1.0 | forcing: ATM<-SST=290.00, OCN<-T_atm=288.00
t= 1.0  T_atm=288.02 K  SST=289.99 K  Soil=0.30
...
t= 20.0  T_atm=288.35 K  SST=289.83 K  Soil=0.28
```

This example demonstrates how object-oriented programming supports time-dependent simulation, component coupling, and transparent diagnostics. Although the physical processes are highly simplified, the structure closely mirrors that of operational Earth system models and provides a natural entry point for extensions such as stochastic forcing, data assimilation, or machine-learning-based parameterizations.

Information exchange between components is handled explicitly in the `step()` method of the `EarthSystemModel`. At each time step, the model reads state variables directly from the component instances (for example `self.ocean.sst` and `self.atmosphere.temperature`) and passes them as entries in the forcing dictionary to the `step()` methods of other components. This makes the direction and content of the coupling visible in the code.

In the present implementation, the atmospheric temperature is relaxed toward the current sea surface temperature via `forcing={"sst": self.ocean.sst}`, while the ocean temperature responds to the atmospheric temperature via `forcing={"atm_temp": self.atmosphere.temperature}`. The components themselves do not directly access each other; all coupling is mediated by the model controller. This explicit and centralized exchange mirrors the coupling strategy used in many operational Earth system and numerical weather prediction models.

Summary of Lecture 1

The first lecture established the technical and conceptual foundation for using Python in scientific computing, weather and climate modeling, and AI/ML workflows. Table 1.2 summarizes the key concepts and skills introduced.

Topic	Key Concepts and Skills
Python Installation & Versions	Installing Python, checking versions, and understanding platform-dependent executables (python, python3, etc.). Emphasis on using modern Python versions (≥ 3.10).
Virtual Environments	Creating and activating isolated Python environments using venv to ensure reproducibility and avoid dependency conflicts across projects.
Package Management with pip	Installing, listing, and managing packages from PyPI. Understanding how scientific Python ecosystems are composed of modular libraries.
Dependency Management	Creating and using requirements.txt to fully describe software environments and reproduce them across machines and teams.
Imports vs. Installation	Distinguishing local imports from installable packages. Using importlib.reload() for interactive development and rapid prototyping.
Python Packaging	Creating installable Python packages with pyproject.toml. Understanding editable installs and the transition from legacy setup.py.
NumPy Fundamentals	Creating arrays, vectorized operations, slicing, broadcasting, and using predefined arrays. Understanding NumPy as the backbone of scientific computing in Python.
Scientific Plotting	Generating 1D and 2D visualizations with Matplotlib. Saving figures, controlling layout, and visualizing numerical data effectively.
Python Functions	Defining and calling functions, using multiple parameters, default values, and variable-length arguments (*args).
Core Python Concepts	Control flow (if, loops), exception handling, file I/O, and use of standard libraries such as os, sys, math, datetime, and json.
Object-Oriented Programming	Defining classes, constructors (__init__), methods, encapsulation, and polymorphism.
Earth System Modeling Example	Building a time-stepping, coupled system using classes. Explicit component coupling, persistent state, and transparent diagnostics, mirroring the structure of operational Earth system models.

Table 1.2: Summary of key topics and skills covered in Lecture 1.

Chapter 2

Jupyter Notebooks, APIs and Servers

2.1 Introduction to Jupyter Notebooks

2.1.1 What is Jupyter Notebook?

Jupyter Notebook is an open-source web-based tool that allows you to create and share documents containing live code, equations, visualizations, and explanatory text. It supports various programming languages, including Python, making it an essential tool for data analysis, machine learning, and scientific computing. Its interactive nature allows for rapid prototyping, testing, and visualization of code, making it particularly useful for beginners and experts alike.

2.1.2 Installing and Running Jupyter

To install Jupyter Notebook, use Python's package manager, pip:

Install Jupyter Notebook

```
1 pip install jupyter  
2 pip install jupyterlab
```

Once installed, you can start Jupyter Notebook by running the following command in your terminal:

Run Jupyter Notebook

```
1 jupyter notebook
```

or `jupyter notebook mynotebook.ipynb`. This will open a web browser with the Jupyter interface, allowing you to create and manage notebooks. On many clouds there is jupyter pre-installed with many packages which you might want to use.

As an example, Amazon Web Services (AWS) for example offers a ready-to-go machine learning framework where you get all packages for using pytorch from the beginning. However, running any of these will cost you per hour - do not forget to shut it down once you are done, otherwise

you might be surprised how small amounts of payments can accumulate over days and weeks (happened to me once).

2.1.3 Basic Operations in Jupyter

In Jupyter, each notebook consists of cells that can hold code, text, or markdown. Common operations include:

- **Running Code:** Press Shift+Enter to execute the code in the current cell and move to the next.
- **Adding Cells:** Use the + button or press B to add a cell below the current one.
- **Changing Cell Type:** Switch between Code and Markdown using the dropdown or press Esc + M.
- **Saving Notebooks:** Press Ctrl+S or use the Save button to save your work.
- **Export as Code:** You can export a Jupyter Notebook to a Python code file by selecting File > Download as > Python (.py) in the Jupyter interface, or by running the command jupyter nbconvert --to script notebook.ipynb in the terminal.

Jupyter also provides built-in visualization support with libraries like matplotlib, making it ideal for data-driven projects. Its flexibility and ease of use make it a crucial tool for Python developers.

2.1.4 Installing Packages in Jupyter Notebooks with !pip install

In Jupyter Notebooks, you can install Python packages directly from within a code cell using the exclamation mark (!) followed by the usual pip install command. This is particularly useful because it eliminates the need to switch to a terminal or command line interface. The packages go into the virtual environment you have been using to call jupyter.

To install a package, simply run:

Installing a Package in Jupyter

```
1 !pip install numpy
```

This command installs the numpy package in your current Jupyter environment.

Why use !pip install in Jupyter?

- It ensures that the package is installed directly into the environment where the notebook is running.
- Convenient for interactive development without leaving the notebook interface.

If you encounter issues where Jupyter uses a different Python environment than your terminal, you can explicitly install packages to the notebook's Python environment by using:

Ensuring Correct Environment

```
1 import sys
2 !{sys.executable} -m pip install package_name
```

where

```
print({sys.executable})
```

shows the path of the current python binary used for execution, i.e.

```
{'C:\\\\Users\\\\rolan\\\\all\\\\ropy312\\\\Scripts\\\\python.exe'}
```

on my windows computer.

This guarantees that pip installs the package into the environment running the notebook, ensuring compatibility and avoiding common environment issues.

2.1.5 Running Jupyter on a Remote Linux Machine with Port Forwarding

When working on remote servers, such as a Linux machine over SSH, you can still use Jupyter Notebooks by starting it on the remote machine and forwarding the port to your local machine (Windows or Linux). This ensures you can access the notebook in your local browser while running the code on the powerful remote server.

Starting Jupyter on the Remote Linux Machine

First, log in to your remote Linux machine via SSH. Then, start Jupyter Notebook with:

Remote Command on Linux

```
1 jupyter notebook --no-browser --port=8888
```

This command starts Jupyter on port 8888 without opening a browser window on the remote machine.

Port Forwarding from Local Machine

To access this remote Jupyter server, you need to forward the port from the remote machine to your local machine. On a local Linux machine (using Bash) or Windows (Powershell):

Port Forwarding on Linux

```
1 ssh -N -L 9001:localhost:8888 user@remote-server-ip &
```

This forwards the remote port 8888 to your local machine's port 9001.

Accessing Jupyter Notebook in Your Local Browser

Once the SSH connection is established, open a browser on your local machine and navigate to:

`http://localhost:9001`

You will see the Jupyter Notebook interface running on the remote machine, accessible from your local browser. However, it will probably ask you for the token, which is displayed when you start the Jupyter notebook:

```
To access the server, open this file in a browser:  

  file:///home/roland/.local/share/jupyter/runtime/jpserver-6745-open.html  

Or copy and paste one of these URLs:  

  http://localhost:8888/tree?token=2bfafdead00bd642b4fc56a57864e3e9ca92bc41e49f4c1f6  

  http://127.0.0.1:8888/tree?token=2bfafdead00bd642b4fc56a57864e3e9ca92bc41e49f4c1f6
```

The port 8888, however, is on the remote machine, you have forwarded it locally to 9001 and need to replace this, then use your browser to access the Jupyter notebook.

2.1.6 Using Markdown Cells for Documentation

Markdown cells in Jupyter Notebooks allow you to add formatted text, making your notebooks more readable and well-documented. To create a Markdown cell, simply change the cell type from Code to Markdown.

Markdown supports:

- **Headings:** Use # for headings (# Heading 1, ## Heading 2).
- **Bold and Italics:** Use **bold** or *italic*.
- **Lists:** Create ordered lists with numbers and unordered lists with dashes.
- **Links and Images:** Add links with [text](url) and images with ![alt text](image_url).
- **LaTeX Equations:** For mathematical expressions, enclose LaTeX code in \$...\$ for inline equations or \$\$...\$\$ for display equations.

Markdown transforms Jupyter notebooks into interactive documents combining code, text, and visuals seamlessly.

2.1.7 Using Magic Commands in Jupyter Notebooks

Jupyter provides special **magic commands** that simplify various tasks such as timing code execution, managing the environment, and more. Magic commands start with a single % for line magics and %% for cell magics.

Common Magic Commands:

- %time: Times the execution of a single line of code.

Timing a Code Line

```
1 %time sum(range(1000000))
```

- `%timeit`: Runs code multiple times and gives an average runtime.
- `%lsmagic`: Lists all available magic commands.
- `%%writefile`: Writes the contents of a cell to an external file.

Writing to a File

```
1 %%writefile magic_hello.py
2 print("Hello, world!")
```

- `%%bash`: Runs Bash commands directly inside a Jupyter cell.

Magic commands enhance productivity by providing quick, built-in operations within Jupyter.

2.1.8 Running Shell Commands in Jupyter Notebooks

Jupyter allows you to execute shell commands directly within code cells using the exclamation mark (!). This is useful for interacting with the operating system without leaving the notebook.

Examples of Shell Commands in Jupyter:

- List files in the current directory:

List Files

```
1 !ls
```

- Install packages using pip:

Install a Package

```
1 !pip install numpy
```

- Check the Python version:

Check Python Version

```
1 !python --version
```

Shell commands allow seamless interaction with the system, making Jupyter highly versatile for both coding and administrative tasks.

2.1.9 Data Visualization in Jupyter with Matplotlib, Seaborn, and Plotly

Jupyter Notebooks integrate well with popular Python visualization libraries, making it easy to create plots and graphs directly within your notebook.

Lorenz63 Calculation and Visualization

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Parameters and initial condition
5 sigma, beta, rho = 10, 8/3, 28
6 dt, steps = 0.01, 10000
7 xyz = np.empty((steps, 3))
8 xyz[0] = (1, 1, 1)
9
10 # Integration using Euler method
11 for i in range(steps - 1):
12     x, y, z = xyz[i]
13     dx = sigma * (y - x)
14     dy = x * (rho - z) - y
15     dz = x * y - beta * z
16     xyz[i + 1] = xyz[i] + dt * np.array([dx, dy, dz])
17
18 # Plot the result
19 fig = plt.figure(figsize=(6, 4))
20 ax = fig.add_subplot(projection='3d')
21 ax.plot(*xyz.T, lw=0.5)
22 ax.set_title("Lorenz Attractor")
23 ax.set_facecolor("white")      # plot area (axes background)
24 plt.savefig('images/img02/lorenz63.png')
25 plt.show()

```

And another code based on the seaborn package, where you need to pip install seaborn first, then run:

lorenz63-seaborn.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 # Parameters for the Lorenz system
6 s = 10.0    # Sigma
7 r = 28.0    # Rho
8 b = 8.0 / 3.0  # Beta
9
10 # Time step and number of iterations

```

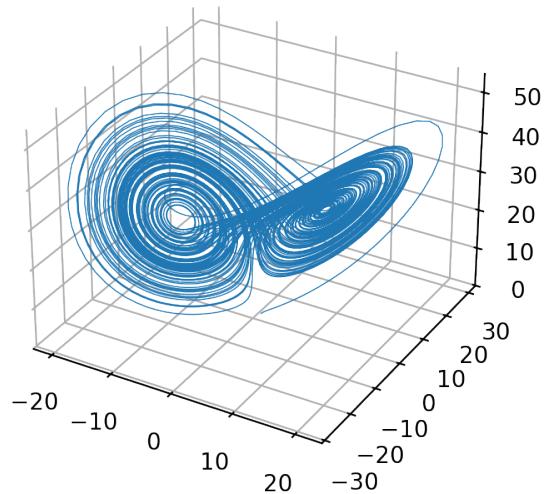


Figure 2.1: Matplotlib within Jupyter, Lorenz 63 Attractor.

```

11 dt, N = 0.01, 10000
12
13 # Array to hold x, y, z
14 xyz = np.zeros((N, 3))
15 xyz[0] = 1, 1, 1 # Initial condition
16
17 # Integrate using Euler's method
18 for i in range(1, N):
19     x, y, z = xyz[i-1]
20     dx = s * (y - x)
21     dy = x * (r - z) - y
22     dz = x * y - b * z
23     xyz[i] = x + dt * dx, y + dt * dy, z + dt * dz
24
25 # KDE plot with seaborn
26 sns.set(style="white")
27 plt.figure(figsize=(6, 5))
28 kde = sns.kdeplot(
29     x=xyz[:, 0], y=xyz[:, 2], # x vs z
30     fill=True, cmap="viridis", levels=100, thresh=0.02
31 )
32 plt.colorbar(kde.collections[0], label="Density")
33 plt.title("Lorenz Attractor Density (x vs z)")
34 plt.xlabel("x")
35 plt.ylabel("z")
36 plt.tight_layout()

```

```
37 plt.savefig("images/img02/lorenz63-seaborn.png")
38 plt.show()
```

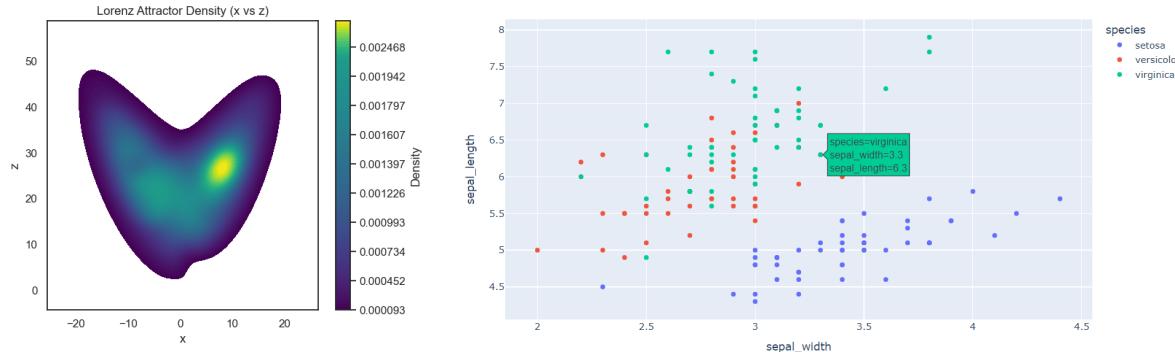


Figure 2.2: Density visualization based on seaborn package and interactive plotly visualization.

Interactive plots can easily be integrated into jupyter notebooks, here for example with the *plotly* package. You need to install

```
pip install numpy
pip install plotly
pip install pandas
```

We will discuss pandas further in a later session.

Using plotly for Interactive Plots:

Plotly Example

```
1 import plotly.express as px
2 df = px.data.iris()
3 fig = px.scatter(df, x='sepal_width', y='sepal_length', color='species')
4 fig.show()
```

With these and further libraries, Jupyter becomes a powerful tool for both static and interactive data visualizations. You cannot develop applications in artificial intelligence without looking at data and results in a very careful way, bringing in a lot of domain specific know-how!

Recommendation

Fluency in using Jupyter Notebooks is essential for effective Python development.

2.2 Introduction to APIs: A Key Principle in Code Development

Python is more than a programming language. It is an eco system which provides a lot of functionality which is needed for either AI/ML applications or other types of user services. In particular, APIs are extremely useful and, today, ubiquitous in scientific applications.

An **API (Application Programming Interface)** is a defined set of rules and tools that allows different pieces of software to communicate with each other. APIs are essential in modern programming because they enable modular, reusable, and maintainable code. From simple functions within a local Python module to complex web-based services, APIs provide a structured way to access and share functionality.

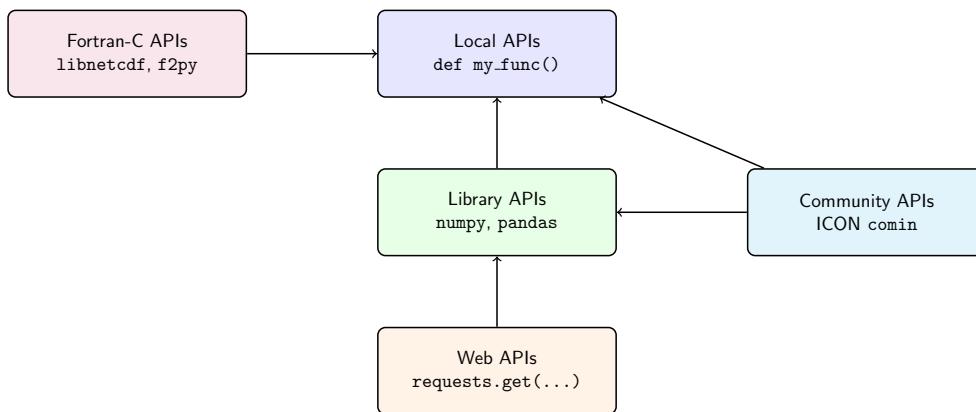


Figure 2.3: Importance of API design and functionality.

APIs work by exposing specific methods or endpoints that other code can call. For example, a Python module can expose a function like `add(a, b)`, or a web service can expose an HTTP endpoint like `/weather?city=Berlin`. In both cases, the underlying logic is hidden, and only the necessary interface is visible. This separation is crucial for code maintenance and scalability.

2.2.1 Why Learn APIs from the Beginning?

APIs are not just an advanced tool but a **fundamental principle of code development** that should be learned from the start. Here's why:

- **Modularity:** APIs encourage splitting code into independent modules, making it easier to test, maintain, and extend.
- **Reusability:** Functions and classes defined in one project can be reused across different projects through APIs.
- **Collaboration:** APIs allow teams to work on different components simultaneously, with clearly defined interfaces.
- **Abstraction:** Details are hidden behind the API, exposing only what is necessary, which helps avoid unnecessary complexity.
- **Scalability:** As projects grow, APIs provide a stable way to integrate new features without breaking existing code.

APIs are **everywhere in Python development**, from the built-in functions of the standard library to external packages like numpy or pandas. When working with data, machine learning models, or even complex weather systems, APIs help organize the code logically and efficiently.

We use APIs in many places for AI/ML development. It is there for data provision. It defines the connection between **user interfaces**, the **server** managing user requests, the **large language model** providing an intelligent service, the **function calls** which link specific functionality into the user-service interaction.

2.2.2 Types of APIs in Python

APIs in Python can take various forms:

- **Local APIs:** A set of functions or classes within a Python module that can be imported and used in other scripts.
- **Library APIs:** External libraries like numpy or pandas expose APIs that developers use for numerical operations or data manipulation.
- **Web APIs:** Services like OpenWeatherMap or PokeAPI provide data over HTTP, which Python can access using tools like requests.

2.2.3 APIs as a Structuring Principle for Code Development

From the beginning of your Python learning journey, understanding and using APIs helps build **structured, maintainable, and scalable code**. APIs force developers to think about clear interfaces, modular design, and reusability, which are essential practices in any project, large or small.

In this tutorial, we will explore both local and web APIs, demonstrating how to create and consume APIs to build powerful and efficient Python applications.

2.3 Making API Requests with requests

In modern software development, REST APIs have become a standard method for enabling communication between distributed components. The API we developed in `code011_REST.py` uses the Flask framework to expose endpoints that allow operations such as creating, reading, updating, and deleting items. This design follows the REST principles by ensuring a stateless, client–server interaction with a clear separation of concerns. On the server side, we define endpoints like `/items` for retrieving or adding items, and `/items/<id>` for working with individual items.

The client implementation, found in `code012_REST_client.py`, leverages the Python `requests` library to interact with these endpoints. This library abstracts the details of HTTP communication and provides simple functions for GET, POST, PUT, and DELETE requests. By using `requests`, developers can focus on the application logic rather than on low-level network details.

Setting Up the Server:

In code011_REST.py, the Flask server is set up to listen on a local port (usually 5000). The code defines several endpoints:

- **GET /items**: Returns the entire collection of items as JSON.
- **GET /items/<id>**: Retrieves a specific item by its identifier.
- **POST /items**: Accepts JSON data to create a new item. The new item's identifier is generated automatically and also allows client-specified IDs.
- **PUT /items/<id>**: Updates an existing item.
- **DELETE /items/<id>**: Removes an item from the collection.
- **POST /items/<id>/upload**: Uploads a file associated with an item. The server saves the file in a designated uploads directory and records the file path in the item's data.
- **GET /items/<id>/download**: Downloads the file associated with an item. The server retrieves the stored file and sends it as an attachment, allowing the client to save it with its original filename.

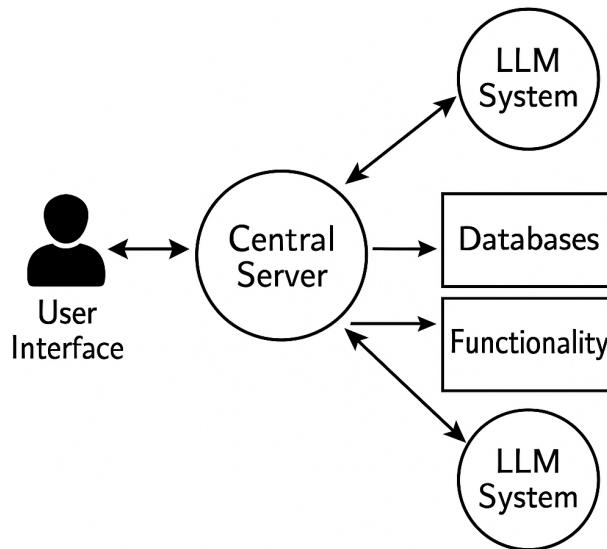


Figure 2.4: How APIs are crucial for AI/ML applications involving large language models (LLM) with user services. The **user interface** interacts with the **central server** through an API. The server uses APIs to talk to the **LLMs**. It uses APIs for **database requests** (including the user and rights management, but also to pull observations, fields, analyses and much more. It also interacts with specific **functionality** providing **weather and climate services**, including sophisticated AI/ML applications, through further APIs.

We have the full server code as demo application in the file `flask_server_request_api.py`. Here, we explain its components:

Server Setup. We start by importing necessary modules and creating the Flask app instance.

Setup

```
1 from flask import Flask, jsonify, request, abort, send_from_directory
2 from werkzeug.utils import secure_filename
3 import os
4
5 app = Flask(__name__)
```

Upload Folder and Allowed Extensions. Define the upload folder and allowed file types.

Upload Configuration

```
1 UPLOAD_FOLDER = 'uploads'
2 ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}
3
4 if not os.path.exists(UPLOAD_FOLDER):
5     os.makedirs(UPLOAD_FOLDER)
6
7 def allowed_file(filename):
8     return '.' in filename and \
9         filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

In-Memory Item List. A simple list of items simulates a database.

Initial Items

```
1 items = [
2     {"id": 1, "name": "Item 1"},
3     {"id": 2, "name": "Item 2"},
4 ]
```

GET /items. Return all items.

GET /items

```
1 @app.route('/items', methods=['GET'])
2 def get_items():
3     return jsonify(items)
```

GET /items/<id>. Return a single item by ID.

GET /items/<id>

```
1 @app.route('/items/<int:item_id>', methods=['GET'])
2 def get_item(item_id):
3     item = next((item for item in items if item['id'] == item_id), None)
```

```

4     if item is None:
5         abort(404)
6     return jsonify(item)

```

POST /items. Add a new item.**POST /items**

```

1 @app.route('/items', methods=['POST'])
2 def create_item():
3     if not request.json or 'name' not in request.json:
4         abort(400)
5     new_item = {
6         "id": items[-1]["id"] + 1 if items else 1,
7         "name": request.json['name']
8     }
9     items.append(new_item)
10    return jsonify(new_item), 201

```

PUT /items/<id>. Update or create an item by ID.**PUT /items/<id>**

```

1 @app.route('/items/<int:item_id>', methods=['PUT'])
2 def update_or_create_item(item_id):
3     if not request.json or 'name' not in request.json:
4         abort(400)
5     item = next((item for item in items if item['id'] == item_id), None)
6     if item is None:
7         new_item = {"id": item_id, "name": request.json['name']}
8         items.append(new_item)
9         return jsonify(new_item), 201
10    else:
11        item['name'] = request.json.get('name', item['name'])
12        return jsonify(item)

```

DELETE /items/<id>. Delete an item.**DELETE /items/<id>**

```

1 @app.route('/items/<int:item_id>', methods=['DELETE'])
2 def delete_item(item_id):
3     global items
4     items = [item for item in items if item['id'] != item_id]
5     return jsonify({'result': True})

```

POST /items/<id>/upload. Upload a file for a specific item.

POST /items/<id>/upload

```

1 @app.route('/items/<int:item_id>/upload', methods=['POST'])
2 def upload_file(item_id):
3     item = next((item for item in items if item['id'] == item_id), None)
4     if item is None:
5         abort(404)
6     if 'file' not in request.files:
7         abort(400, description="No file part in the request")
8     file = request.files['file']
9     if file.filename == '':
10        abort(400, description="No selected file")
11    if file and allowed_file(file.filename):
12        filename = secure_filename(file.filename)
13        saved_filename = f"{item_id}_{filename}"
14        file_path = os.path.join(UPLOAD_FOLDER, saved_filename)
15        file.save(file_path)
16        item['file'] = file_path
17        return jsonify({'result': 'File uploaded', 'file_path': file_path}), 201
18    else:
19        abort(400, description="File type not allowed")

```

GET /items/<id>/download. Download a file attached to an item.

GET /items/<id>/download

```

1 @app.route('/items/<int:item_id>/download', methods=['GET'])
2 def download_file(item_id):
3     item = next((item for item in items if item['id'] == item_id), None)
4     if item is None or 'file' not in item:
5         abort(404)
6     file_path = item['file']
7     directory, filename = os.path.split(file_path)
8     return send_from_directory(directory, filename, as_attachment=True)

```

Start the Server. Run the application in debug mode.

Run Server

```

1 if __name__ == '__main__':
2     app.run(debug=True)

```

This server code, which you can view in detail in `flask_server_request_api.py`, serves as the API's backend. The careful design ensures that the API is both stateless and uniform, allowing clients to interact predictably with the service.

Interacting with the API Using requests:

On the client side, `code012_REST_client.py` demonstrates how to use the `requests` library to make calls to our API. Let's consider a few typical examples:

1. Retrieving All Items:

A simple GET request is used to fetch the list of items. The client code sends:

```
response = requests.get('http://127.0.0.1:5000/items')
print(response.json())
```

This call returns a JSON array containing all items. By decoding the response, the client can easily process and display the data.

2. Retrieving a Specific Item:

To fetch an individual item, the client sends a GET request with the item's ID in the URL:

```
response = requests.get('http://127.0.0.1:5000/items/1')
print(response.json())
```

If the item exists, the server returns its details in JSON format; if not, an error (typically a 404 Not Found) is returned.

3. Adding a New Item:

The POST request is used to create a new item. In our implementation, the client sends a JSON payload:

```
new_item = {'name': 'New Item'}
response = requests.post('http://127.0.0.1:5000/items', json=new_item)
print(response.json())
```

The server then generates a new item with a unique ID and returns it. Notice that the `json=` parameter in the request call makes it easy to send JSON data without manual serialization.

4. Updating and Deleting Items:

Similarly, PUT requests are used to update an item and DELETE requests to remove it. The corresponding code in `code012_REST_client.py` handles these actions by specifying the correct URL endpoints and sending appropriate JSON data if necessary.

5. Upload Functionality:

In addition to updating and deleting items, the REST API example demonstrates file uploads. Clients can attach files to specific items by sending a POST request to an endpoint such as `/items/<id>/upload`. This endpoint accepts multipart form-data where the file is provided under a designated field (e.g., `'file'`). On the server side, Flask processes the incoming file, ensures its name is secured using `secure_filename`, and then saves it into a dedicated `uploads` directory. The file path is subsequently stored in the item's record, associating the file with the item. This approach enables users to easily manage additional resources related to each item.

6. Download Functionality:

Complementing the upload feature, the API also provides a download endpoint at `/items/<id>/download`. When a client sends a GET request to this endpoint, the server

locates the file associated with the item and transmits it back as an attachment using Flask's send_from_directory function. This not only ensures that the file is delivered with the correct MIME type but also prompts the client's browser to download it rather than display it inline. On the client side, the downloaded file can be saved with its original name by removing any item-specific prefixes that were added during upload, thus preserving the original filename. This integrated upload and download mechanism enhances the functionality of the REST API by allowing it to handle both data and associated file resources seamlessly.

```
1 import requests
2
3 # Base URL of the API
4 base_url = 'http://127.0.0.1:5000'
5
6 # GET all items
7 response = requests.get(f'{base_url}/items')
8 print("GET /items:", response.json())
9
10 # GET a specific item (e.g., id = 1)
11 response = requests.get(f'{base_url}/items/1')
12 print("GET /items/1:", response.json())
13
14 # POST a new item
15 new_item = {'name': 'New Item'}
16 response = requests.post(f'{base_url}/items', json=new_item)
17 print("POST /items:", response.json())
18
19 # PUT to update an item (e.g., id = 1)
20 updated_item = {'name': 'Updated Item 1'}
21 response = requests.put(f'{base_url}/items/1', json=updated_item)
22 print("PUT /items/1:", response.json())
23
24 # DELETE an item (e.g., id = 1)
25 response = requests.delete(f'{base_url}/items/1')
26 print("DELETE /items/1:", response.json())
27
28 # Check items after deletion
29 response = requests.get(f'{base_url}/items')
30 print("GET /items after deletion:", response.json())
31
32 # -----
33 # UPLOAD a file for an item (e.g., for item with id = 2)
34 upload_url = f'{base_url}/items/2/upload'
35 # Ensure you have a file named 'example.txt' in your current directory
36 with open('example.txt', 'rb') as f:
37     files = {'file': f}
38     response = requests.post(upload_url, files=files)
39     print("POST /items/2/upload:", response.json())
40
```

```

41 # -----
42 # DOWNLOAD the file associated with an item (e.g., for item with id = 2)
43 download_url = f'{base_url}/items/2/download'
44 response = requests.get(download_url, stream=True)
45 if response.status_code == 200:
46     # Save the downloaded file locally
47     with open('downloaded_example.txt', 'wb') as f:
48         for chunk in response.iter_content(chunk_size=8192):
49             f.write(chunk)
50     print("File downloaded successfully and saved as downloaded_example.txt")
51 else:
52     print("Failed to download file, status code:", response.status_code)

```

Error Handling and Debugging:

A crucial aspect of making API requests is managing errors gracefully. The client code checks the HTTP status code returned by the server and handles error responses appropriately. For instance, if a GET request for an item returns a 404 status code, the client can notify the user that the requested item does not exist. Similarly, for POST and PUT requests, verifying that the server returns the expected 201 or 200 status code helps ensure that operations have completed successfully.

Benefits of This Approach:

Using the `requests` library to interact with our REST API provides several benefits:

- **Simplicity:** The `requests` library offers an intuitive API that abstracts the complexity of HTTP communication.
- **Flexibility:** Developers can easily extend the client to support additional endpoints or incorporate authentication mechanisms.
- **Maintainability:** By separating the server (`code011_REST.py`) and client (`code012_REST_client.py`) code, our architecture is modular. This makes it easier to update one component without affecting the other.
- **Reusability:** The external code inclusion method using `\includeexternalcode` promotes code reuse and ensures that our documentation is consistent with our source code.

It is now very easy to define simple functions such as `list()`, `up(<filename>, <id>)` or `down(<id>)` to list all uploaded items, to upload a particular file or to download a particular file.

In summary, making API requests with Python's `requests` library is both straightforward and powerful. Our implementation demonstrates a complete cycle: setting up a REST API server with Flask, handling standard HTTP methods, and interacting with the API via a client script. The combination of clear server endpoints, robust client-side error handling, and modular code inclusion makes this approach a solid foundation for building scalable and maintainable web services.

By following these practices, you can build reliable applications that communicate over HTTP in a standardized way, ultimately leading to more effective and efficient software systems.

Recommendation

An API-centric mindset greatly enhances fast development, modular design, and clean separation of responsibilities.

2.4 Fortran Integration using `ctypes` as API

In this section, we illustrate an approach to integrating FORTRAN code with Python by using the `ctypes` module. The library `ctypes` provides explicit control over data types and memory management when interfacing with compiled shared libraries.

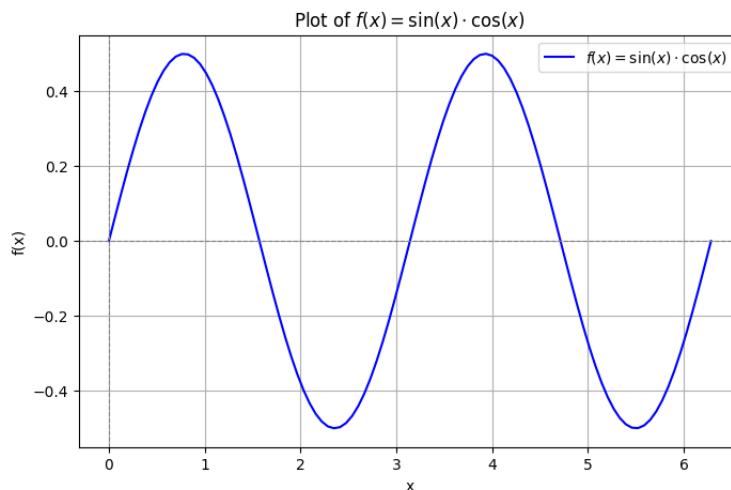


Figure 2.5: We use the Fortran `iso_c_binding` module to create a C-compatible API, allowing Fortran routines to be called from C or other languages such as Python via `ctypes` or `f2py`.

The following example demonstrates how to load a Fortran shared library (compiled as `fortran_interface.so`), define the function prototype for the Fortran function `f_sin_cos`, and compute the function $f(x) = \sin(x) \cdot \cos(x)$ for 100 values between 0 and 2π . The computed values are then plotted using Matplotlib. Additionally, the code checks for the existence of a Fortran debug log and prints its contents if available.

FORTRAN code example

```

1 %%writefile fortran_interface.f90
2 module fortran_module
3     use iso_c_binding, only: c_double
4     implicit none
5 contains
6     function f_sin_cos(x) result(f) bind(C, name="f_sin_cos")
7         implicit none
8         real(c_double), intent(in) :: x

```

```
9      real(c_double) :: f
10     f = sin(x) * cos(x)
11   end function f_sin_cos
12 end module fortran_module
```

Compile this based on gfortran.

Compilation

```
1 gfortran -shared -fPIC fortran_interface.f90 -o fortran_interface.so
```

Then, you might use the code, e.g. in a jupyter notebook or as basic python application.

It can be extremely helpful to make your fortran modules and functions available for execution in your python framework. We will pursue this further in upcoming parts of this tutorial.

Chapter 3

Eccodes for Grib, Opendata, NetCDF, Visualization

Meteorological data are often stored in GRIB or NetCDF formats, both of which are compact binary formats widely used in numerical weather prediction and climate analysis. While GRIB is the standard format for meteorological model outputs, NetCDF is more commonly used in the broader scientific community, particularly for observational datasets and climate research. Additionally, Zarr is an alternative format to NetCDF, optimized for cloud-based storage and parallel computing for machine learning applications. Recent developments allow storing NetCDF data in Zarr format for enhanced scalability.

In this chapter, we demonstrate how to work with both formats, focusing on GRIB data using the ECCODES library—provided by ECMWF—to decode and analyze meteorological data from the DWD Open Data Server, and working with NetCDF for further integration and analysis. We will cover the following steps:

- *Accessing and Downloading GRIB Data:* Retrieving ICON model GRIB files, including latitude-longitude fields and the 2-m temperature field, from the DWD Open Data Server.
- *Inspecting GRIB File Metadata:* Listing GRIB file keys, understanding metadata, and summarizing the available parameters and levels.
- *Loading and Visualizing GRIB Data:* Extracting numerical fields, mapping the spatial coordinates, and plotting meteorological variables using visualization tools.

In addition, we introduce NetCDF as a fundamental format for meteorological and climate data, covering the following aspects:

- *Accessing and Managing NetCDF Data:* Understanding the structure of NetCDF files, reading data, and managing metadata.
- *Working with NetCDF Data:* Processing and analyzing NetCDF datasets in the context of meteorological applications.
- *Visualizing NetCDF Data:* Plotting and interpreting NetCDF data using scientific computing tools.

By following these steps, we will gain a complete workflow for handling ICON model GRIB and NetCDF files, from downloading and inspection to visualization and analysis for scientific applications.

3.1 Downloading ICON Model GRIB Files from DWD Open Data Server

To analyze meteorological data using the ICON model, we need to download the required GRIB files from the *DWD Open Data Server*. These include the *2-meter temperature field* (icon_t2m.grib), the *latitude grid* (icon_lat.grib), and the *longitude grid* (icon_lon.grib). DWD provides these files in compressed GRIB2 format (.grib2.bz2), which must be downloaded and extracted before further processing.

Downloading the 2-Meter Temperature Field.

The following script constructs the filename for the *latest 2m temperature GRIB file* based on the current UTC date, downloads it using wget, and extracts it.

```
Downloading 2m Temperature Data

1 import datetime
2 import os
3 import wget
4 import bz2
5
6 # Construct the filename based on the current UTC date
7 now = datetime.datetime.now(datetime.UTC)
8 filename = f"icon_global_icosahedral_single-level_{now:%Y%m%d}00_000_T_2M.grib2.
9     bz2"
10 print("Constructed filename:", filename)
11
12 # Define the base URL
13 base_url = "https://opendata.dwd.de/weather/nwp/icon/grib/00/t_2m/"
14 url = base_url + filename
15 print("Download URL:", url)
16
17 # Download the .bz2 file using Python wget
18 wget.download(url, filename)
19 print(f"\nDownloaded {filename}")
20
21 # Decompress the .bz2 file using bz2 module
22 with bz2.open(filename, 'rb') as f_in, open(filename[:-4], 'wb') as f_out:
23     f_out.write(f_in.read())
24 print(f"Decompressed {filename} to {filename[:-4]}")
25
26 grib_filename = filename[:-4]
27 final_filename = "icon_t2m.grib"
28 os.rename(grib_filename, final_filename)
29 print(f"Renamed {grib_filename} to {final_filename}")
```

This script ensures that the latest available *2m temperature* field will be automatically retrieved and prepared for use.

Downloading Latitude and Longitude Data.

The latitude (icon_lat.grib) and longitude (icon_lon.grib) grids are *time-invariant* and must be downloaded separately. The script below identifies the latest available version on the DWD server, downloads both files, and renames them for easier access. The following script is contained in the file icon_grid_get.py as well as in the jupyter notebook to catch forecasts from the DWD open data server.

Downloading ICON Latitude and Longitude GRIB Data

```

1 import re
2 import os
3 import bz2
4 import requests
5 import wget
6
7 base_url = "https://opendata.dwd.de/weather/nwp/icon/grib/00/"
8 clat_path = "clat/"
9 clon_path = "clon/"
10
11 # Function to find the latest available timestamp from DWD server
12 def get_latest_timestamp(path):
13     listing_url = base_url + path
14     response = requests.get(listing_url)
15     if response.status_code != 200:
16         raise RuntimeError(f"Could not fetch listing: {listing_url}")
17     timestamps = re.findall(
18         r'icon_global_icosahedral_time-invariant_(\d{10})_CLAT\.grib2\.bz2',
19         response.text
20     )
21     return max(timestamps) if timestamps else None
22
23 # Get the latest available timestamp
24 timestamp = get_latest_timestamp(clat_path)
25 if not timestamp:
26     raise RuntimeError("Could not determine latest timestamp from DWD server.")
27
28 files = {
29     "clat": f"clat/icon_global_icosahedral_time-invariant_{timestamp}_CLAT.grib2.
30             bz2",
31     "clon": f"clon/icon_global_icosahedral_time-invariant_{timestamp}_CLON.grib2.
32             bz2"
33 }
34
35 rename_map = {"clat": "icon_lat.grib", "clon": "icon_lon.grib"}
36
37 for key, path in files.items():
38     filename = os.path.basename(path)

```

```

37     url = base_url + path
38
39     print(f"Downloading {url} ...")
40     wget.download(url, filename)
41     print(f"\nDownloaded {filename}")
42
43     # Uncompress the .bz2 file
44     with bz2.open(filename, 'rb') as compressed, open(filename[:-4], 'wb') as
45         out_file:
46             out_file.write(compressed.read())
47             print(f"Decompressed {filename} to {filename[:-4]}")
48
49     # Rename the extracted file
50     extracted_filename = filename[:-4] # Remove .bz2
51     new_filename = rename_map[key]
52     os.rename(extracted_filename, new_filename)
53     print(f"Renamed {extracted_filename} to {new_filename}")

```

This script identifies the *latest available latitude and longitude files* on the DWD server, downloads them using wget, extracts the .bz2 compressed files, and renames them to icon_lat.grib and icon_lon.grib for easy reference.

After executing these scripts, we have all necessary *spatial coordinate data and temperature fields* to proceed with further analysis and visualization.

3.2 The Grib Library eccodes

We have discussed the download of grib data, here we now assume that we have icon lat and lon coordinates in files icon_lat.grib and icon_lon.grib.

ecCodes is a library developed by ECMWF for decoding and encoding GRIB (GRIdded Binary) files. It provides a Python interface to inspect and manipulate meteorological data stored in the GRIB format.

Installing ecCodes. Installing ecCodes, the ECMWF library for GRIB file handling, can be challenging due to dependencies and system configurations. Here is a summary of the installation process:

System Dependencies: Ensure required system libraries are installed:

```
sudo apt update && sudo apt install libeccodes-dev eccodes
```

On macOS, use Homebrew:

```
brew install eccodes
```

Python Package: Install the Python bindings with:

```
pip install eccodes
```

Recommendation

Using ECCODES on Windows should be done via windows subsystem for linux or through a docker container.

On Windows, you should use the **WSL** (Windows Subsystem for Linux). Here, you can do the same install commands

```
sudo apt update
sudo apt install eccodes
sudo apt install libeccodes-tools
```

Test it with

```
grib_ls -V
```

Setting Environment Variables: If the library is not found, define the paths manually:

```
export EC CODES DEFINITION PATH=/usr/share/eccodes/definitions
export EC CODES SAMPLES PATH=/usr/share/eccodes/samples
```

Adjust these paths according to your system setup.

Verifying Installation: Check if ecCodes is working correctly:

```
grib_ls --help
python -c "import eccodes; print(eccodes.codes_get_api_version())"
```

If these commands return valid output, the installation is successful.

Handling DWD-Specific Definitions: If working with DWD GRIB files, additional definition files may be required. Download the latest version from:

```
https://opendata.dwd.de/weather/lib/grib/
```

We provide a script download_latest_grib_definition_dwd.py which will carry out the download and installation, but still needs you to take care of the path variables. Please check and make sure the definitions paths are set properly.

Checking if ecCodes is Installed Before using ecCodes, you can check if it is installed correctly with the following code:

Checking ecCodes Installation

```
1 try:
2     import eccodes
3     print("ecCodes is installed and working correctly.")
4 except ImportError:
5     print("ecCodes is not installed. Please install it using 'pip install eccodes'")
6     exit(1)
```

Inspecting Grib Files. We next demonstrate how to inspect the metadata keys and shortnames of GRIB files (icon_lat.grib and icon_lon.grib) using ecCodes.

Listing GRIB Keys. The function below extracts and lists metadata keys from a GRIB file:

Listing GRIB Keys

```

1 import eccodes
2
3 def list_grib_keys(grib_filename):
4     """Lists keys from a GRIB file while handling errors properly."""
5     try:
6         with open(grib_filename, 'rb') as f:
7             while True:
8                 gid = eccodes.codes_grib_new_from_file(f)
9                 if gid is None: # End of file
10                     break
11
12                 key_iterator = eccodes.codes_keys_iterator_new(gid)
13                 keys = []
14
15                 while eccodes.codes_keys_iterator_next(key_iterator):
16                     keyname = eccodes.codes_keys_iterator_get_name(key_iterator)
17                     if keyname not in ['section2Padding', 'codedValues', 'values']:
18                         try:
19                             value = eccodes.codes_get_string(gid, keyname)
20                         except Exception:
21                             value = "N/A"
22                         keys.append((keyname, value))
23
24                     eccodes.codes_release(gid)
25
26                     # Print all extracted keys
27                     for key, value in keys:
28                         print(f"Key: {key:40} Value: {value}")
29             except eccodes.CodesInternalError as e:
30                 print(f"ecCodes Error: {e}")
31
32 # Example usage
33 list_grib_keys("icon_lat.grib")

```

Output can be e.g.

Keys

1 Key: globalDomain	Value: g
2 Key: GRIBEditionNumber	Value: 2
3 Key: tablesVersionLatestOfficial	Value: 32
4 Key: tablesVersionLatest	Value: 32
5 Key: grib2divider	Value: 1e+06
6 Key: angleSubdivisions	Value: 1e+06
7 Key: missingValue	Value: 9999
8 Key: ieeeFloats	Value: 1

```
9 Key: isHindcast           Value: 0
10 ...
```

Listing Short Names from a GRIB File.

The function below extracts and lists short names along with their corresponding levels and sizes from a GRIB file:

Listing Short Names

```
1 import eccodes
2
3 def show_shortnames(grib_file):
4     """Lists short names, levels, and sizes from a GRIB file."""
5     with open(grib_file, 'rb') as f:
6         shortName_prev = ''
7         output = ''
8         level_prev = ''
9         count = 0
10        print('-' * 80)
11        print('File = ', grib_file)
12        print('\n{:<30}{:<16}{:>10}'.format('Short Name', 'Level', 'Size'))
13        print('-' * 80)
14        while True:
15            gid = eccodes.codes_grib_new_from_file(f)
16            if gid is None:
17                break
18            shortName = eccodes.codes_get(gid, "shortName")
19            level = eccodes.codes_get(gid, "level")
20            size1 = eccodes.codes_get_size(gid, "values")
21            if shortName_prev != shortName:
22                if level_prev != level and count > 0:
23                    output += f' - {level_prev}'
24                if count > 0:
25                    output += f', \t Size = {size1}'
26                output += '\n{:<30}{:<16}{:>10}'.format(shortName, level, size1)
27                shortName_prev = shortName
28                level_prev = level
29                count += 1
30            eccodes.codes_release(gid)
31        print(output)
32
33 # Example usage
34 show_shortnames("icon_lat.grib")
```

Output is something like e.g.

Shortnames, Levels, Size

```

1 -----
2 File = icon_lat.grib
3 Short Name           Level      Size
4 -----
5 tlat                 0          2949120

```

These functions allow users to inspect the GRIB file structure, understand available variables, and extract key metadata for further analysis.

Plotting Latitude and Longitude Data. To visualize the points stored in the GRIB files, we use matplotlib along with cartopy to overlay the scatter plot on a map:

Plotting Latitude and Longitude with a Map

```

1 import eccodes
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import cartopy.crs as ccrs
5 import cartopy.feature as cfeature
6
7 def plot_lat_lon(lat_file, lon_file):
8     """Plots latitude and longitude points from GRIB files on a map."""
9     def extract_values(grib_file):
10         with open(grib_file, 'rb') as f:
11             gid = eccodes.codes_grib_new_from_file(f)
12             values = eccodes.codes_get_array(gid, "values")
13             eccodes.codes_release(gid)
14         return values
15
16     latitudes = extract_values(lat_file)
17     longitudes = extract_values(lon_file)
18
19     plt.figure(figsize=(10, 5))
20     ax = plt.axes(projection=ccrs.PlateCarree())
21     ax.set_global()
22     ax.add_feature(cfeature.LAND, edgecolor='black')
23     ax.add_feature(cfeature.COASTLINE)
24     ax.add_feature(cfeature.BORDERS, linestyle=':')
25
26     plt.scatter(longitudes, latitudes, s=0.5, color='gray', transform=ccrs.
27 PlateCarree())
28     plt.xlabel("Longitude")
29     plt.ylabel("Latitude")
30     plt.title("Scatter Plot of Latitude and Longitude on a Map")
31     plt.grid()
32     plt.savefig("icon_points_global.png", dpi=300, bbox_inches='tight')
33     plt.show()

```

```
34 # Example usage
35 plot_lat_lon("icon_lat.grib", "icon_lon.grib")
```

This function:

- Reads latitude and longitude values from GRIB files.
- Uses `matplotlib` and `cartopy` to overlay the points on a global map.
- Adds land, coastlines, and country borders for better visualization.

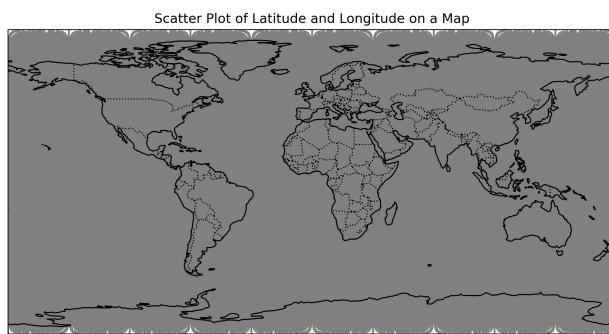


Figure 3.1: Global ICON grid points.

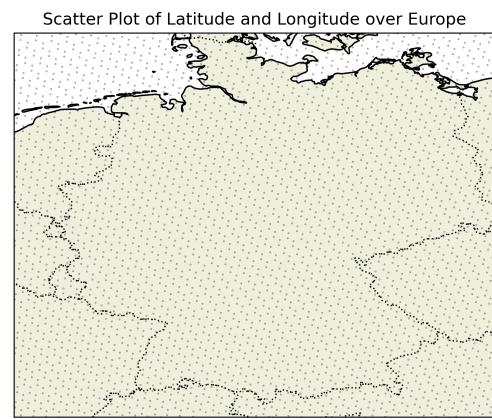


Figure 3.2: ICON grid points over Germany.

But the resolution is quite high, you cannot see much any more, everything is covered by points.

Zooming in Over Germany. To visualize the points stored in the GRIB files, we use `matplotlib` along with `cartopy` to overlay the scatter plot on a map:

Plotting Latitude and Longitude with a Map

```
1 import eccodes
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import cartopy.crs as ccrs
5 import cartopy.feature as cfeature
6
7 def plot_lat_lon_germany(lat_file, lon_file):
8     """Plots latitude and longitude points from GRIB files, zoomed in over Europe.
8     """
9     def extract_values(grib_file):
10         with open(grib_file, 'rb') as f:
11             gid = eccodes.codes_grib_new_from_file(f)
12             values = eccodes.codes_get_array(gid, "values")
13             eccodes.codes_release(gid)
14
15         return values
```

```

16     latitudes = extract_values(lat_file)
17     longitudes = extract_values(lon_file)
18
19     plt.figure(figsize=(10, 5))
20     ax = plt.axes(projection=ccrs.PlateCarree())
21     ax.set_extent([5, 15, 47, 55], crs=ccrs.PlateCarree()) # Europe zoom: [
22     lon_min, lon_max, lat_min, lat_max]
23     ax.add_feature(cfeature.LAND, edgecolor='black')
24     ax.add_feature(cfeature.COASTLINE)
25     ax.add_feature(cfeature.BORDERS, linestyle=':')
26
27     plt.scatter(longitudes, latitudes, s=0.1, color='blue', transform=ccrs.
28     PlateCarree())
29     plt.xlabel("Longitude")
30     plt.ylabel("Latitude")
31     plt.title("Scatter Plot of Latitude and Longitude over Europe")
32     plt.grid()
33     plt.savefig("icon_points_germany.png", dpi=300, bbox_inches='tight')
34     plt.show()
35
36 # Example usage
37 plot_lat_lon_germany("icon_lat.grib", "icon_lon.grib")

```

This function:

- Reads latitude and longitude values from GRIB files.
- Zooms into Europe with specific latitude/longitude boundaries.
- Uses matplotlib and cartopy to overlay the points on a regional map.
- Adjusts the point size to avoid excessive density.

Visualizing 2-Meter Temperature (T2M)

To visualize the 2-meter temperature field from GRIB data, we extract the temperature values alongside the previously loaded latitude and longitude coordinates. The relevant Python function for loading the T2M values is:

Loading T2M Data

```

1 import eccodes
2
3 def load_grib(file, var):
4     """Loads specified variable from GRIB file."""
5     with open(file, 'rb') as f:
6         while (gid := eccodes.codes_grib_new_from_file(f)) is not None:
7             if eccodes.codes_get(gid, "shortName") == var:
8                 vals = eccodes.codes_get_array(gid, "values")
9                 eccodes.codes_release(gid)
10                return vals

```

```

11         eccodes.codes_release(gid)
12     return None
13
14 # Load T2M data
15 t2m = load_grib("icon_t2m.grib", "2t")

```

Once the temperature values are obtained, they are visualized using `matplotlib` and `cartopy`, adapting the point size dynamically based on the bounding box size to maintain clarity across different zoom levels. The visualization function is given below:

Plotting T2M Data

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import cartopy.crs as ccrs
4 import cartopy.feature as cfeature
5
6 def plot_t2m(lat, lon, t2m, bbox, title, fname):
7     """Plots 2m temperature within bbox = (latmin, latmax, lonmin, lonmax)."""
8     latmin, latmax, lonmin, lonmax = bbox
9     mask = (lat >= latmin) & (lat <= latmax) & (lon >= lonmin) & (lon <= lonmax)
10
11    # Adaptive point size based on bounding box area
12    area = (latmax - latmin) * (lonmax - lonmin)
13    point_size = max(0.05, min(10, 500 / area)) # Ensures reasonable point size
14
15    plt.figure(figsize=(10, 6))
16    ax = plt.axes(projection=ccrs.PlateCarree())
17    ax.set_extent([lonmin, lonmax, latmin, latmax])
18    ax.add_feature(cfeature.LAND, edgecolor='black')
19    ax.add_feature(cfeature.COASTLINE)
20    ax.add_feature(cfeature.BORDERS, linestyle=':')
21
22    plt.scatter(lon[mask], lat[mask], c=t2m[mask], cmap='jet', s=point_size,
23                transform=ccrs.PlateCarree())
24    plt.colorbar(label="Temp (K)")
25    plt.title(title)
26    plt.savefig(fname, dpi=300, bbox_inches='tight')
27    plt.show()

```

Using this function, we generate two figures displaying the global distribution of 2-meter temperature and a zoomed-in view over Germany.

Interpolated Visualization of 2-Meter Temperature (T2M)

To enhance the visualization of the 2-meter temperature field, we interpolate the scattered GRIB data onto a regular grid using cubic interpolation. This results in a smooth temperature field representation. The interpolation function is implemented as follows:

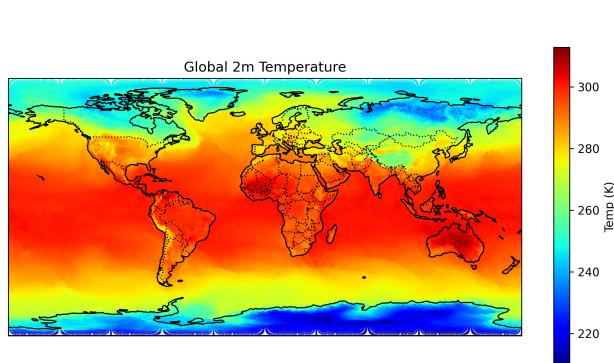


Figure 3.3: Global 2m temperature field.

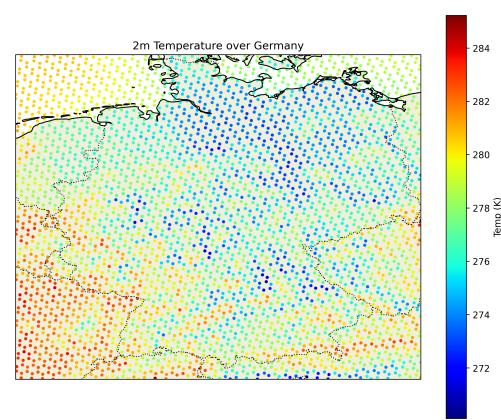


Figure 3.4: 2m temperature field over Germany.

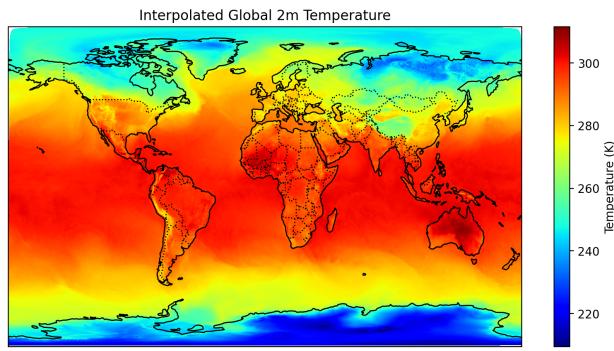


Figure 3.5: Interpolated global 2m temperature field.

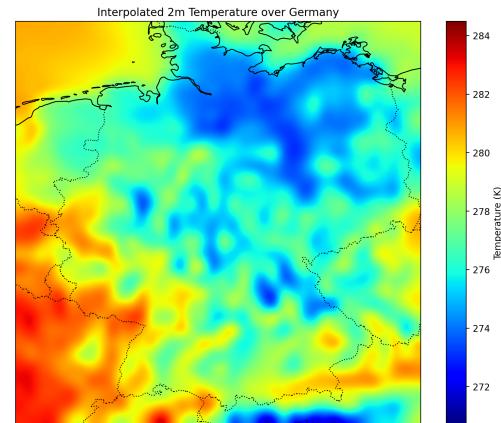


Figure 3.6: Interpolated 2m temperature field over Germany.

Interpolating T2M to a Regular Grid

```

1 import numpy as np
2 from scipy.interpolate import griddata
3
4 def interpolate_to_grid(lat, lon, t2m, bbox, grid_res=0.25):
5     """Interpolates T2M data onto a regular lat/lon grid."""
6     latmin, latmax, lonmin, lonmax = bbox
7
8     # Define a smooth regular grid
9     grid_lat = np.arange(latmin, latmax, grid_res)
10    grid_lon = np.arange(lonmin, lonmax, grid_res)
11    lon_grid, lat_grid = np.meshgrid(grid_lon, grid_lat)
12
13    # Use cubic interpolation for smooth output
14    t2m_grid = griddata((lon, lat), t2m, (lon_grid, lat_grid), method='cubic')
15

```

```
16     return lon_grid, lat_grid, t2m_grid
```

Once the temperature field is interpolated, we visualize it using `matplotlib` and `cartopy`. The function for plotting the interpolated data is shown below:

Plotting Interpolated T2M

```
1 import matplotlib.pyplot as plt
2 import cartopy.crs as ccrs
3 import cartopy.feature as cfeature
4
5 def plot_t2m_grid(lat, lon, t2m, bbox, title, fname):
6     """Plots interpolated 2m temperature as a smooth heatmap."""
7     lon_grid, lat_grid, t2m_grid = interpolate_to_grid(lat, lon, t2m, bbox)
8
9     # Set reasonable aspect ratio based on bounding box size
10    lon_range = bbox[3] - bbox[2]
11    lat_range = bbox[1] - bbox[0]
12    aspect_ratio = lon_range / lat_range
13    figsize = (10, max(5, 10 / aspect_ratio)) # Maintain consistent width &
14    prevent extreme height
15
16    plt.figure(figsize=figsize)
17    ax = plt.axes(projection=ccrs.PlateCarree())
18    ax.set_extent([bbox[2], bbox[3], bbox[0], bbox[1]])
19    ax.add_feature(cfeature.LAND, edgecolor='black')
20    ax.add_feature(cfeature.COASTLINE)
21    ax.add_feature(cfeature.BORDERS, linestyle=':')
22
23    # Use smooth interpolation and correct aspect ratio
24    img = ax.imshow(t2m_grid, extent=[bbox[2], bbox[3], bbox[0], bbox[1]], origin=
25    'lower',
26                cmap='jet', transform=ccrs.PlateCarree(), aspect='auto',
27                interpolation='bicubic')
28
29    plt.colorbar(img, label="Temperature (K)")
30    plt.title(title)
31    plt.savefig(fname, dpi=200, bbox_inches='tight') # Reduce DPI for smaller
32    file size
33    plt.show()
```

Using this interpolation approach, we generate the visualizations shown in Figures 3.5 and 3.6 for the global and regional (Germany) temperature fields, see figure.

Recommendation

Using libraries such as eccodes can be carried out in a very elementary way. At the same time, building packages is an important activity. Keep the balance, being able to do things elementary if necessary, while using packages to work efficiently.

3.3 Accessing SYNOP Observation Files from NetCDF

Observational weather data is often stored in the *BUFR* (Binary Universal Form for the Representation of meteorological data) format, a widely used WMO standard. To facilitate data access, BUFR files are commonly converted into *NetCDF* (Network Common Data Form), which provides a structured, self-describing format suitable for scientific applications.

NetCDF files containing SYNOP observations include essential meteorological variables such as temperature, pressure, humidity, wind speed, and cloud cover. Accessing these files requires a programming framework that can read NetCDF structures efficiently.

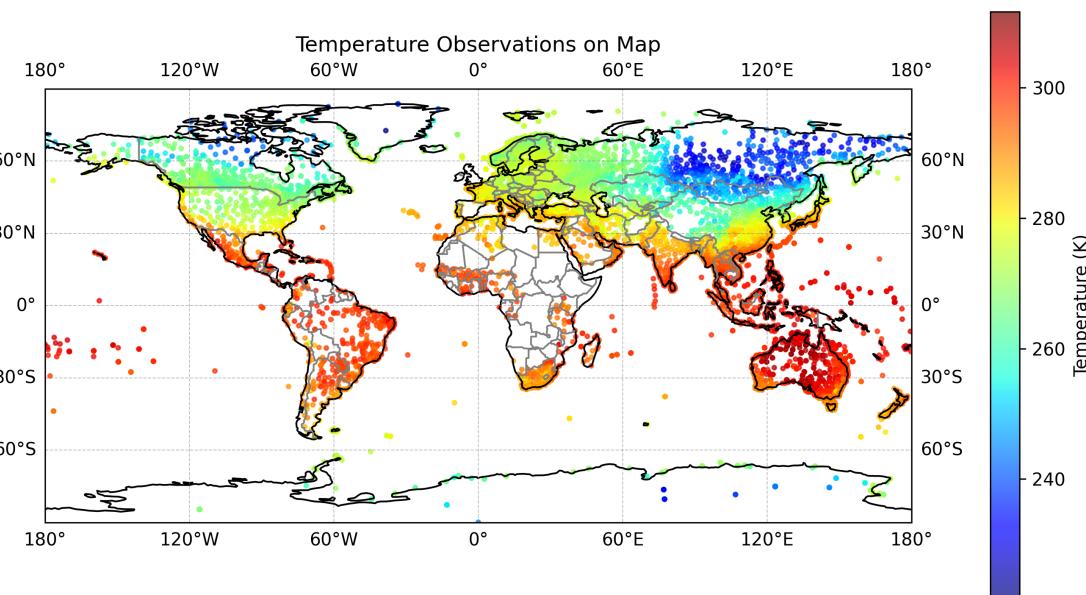


Figure 3.7: Scatter plot of SYNOP temperature observations

Listing Variables in a NetCDF File

To get an overview of the available variables, we use the following script:

```
Listing NetCDF Variables

1 from netCDF4 import Dataset
2
3 def nc_list(file1):
4     """
5         Lists all variables in a given NetCDF file, displaying their names, dimensions
6         , and descriptions.
7     """
8     ncfile = Dataset(file1, 'r')
9     print("{:<4} {:>40} {:>10} {:>10} {:>30}".format("No", "Varname", "shape1",
10         "shape2", "Description"))
11     print("-" * 110)
```

```

12     nc = 1
13     for varname in ncfile.variables.keys():
14         var = ncfile.variables[varname]
15         description = var.long_name if hasattr(var, "long_name") else "N/A"
16         dims = [len(ncfile.dimensions[dim]) for dim in var.dimensions]
17         shape1 = dims[0] if len(dims) > 0 else ""
18         shape2 = dims[1] if len(dims) > 1 else ""
19         print("{:<4} {:>40} {:>10} {:>10} {:>30}".format(nc, varname, shape1,
20             shape2, description))
20         if nc % 10 == 0:
21             print("-" * 110)
22         nc += 1
23     ncfile.close()
24
25 file = "synop.nc"
26 nc_list(file)

```

Example Output

Example Output from nc_list

No	Varname	shape1	shape2	Description
1	edition_number	11993		N/A
2	section1	11993	22	N/A
3	section2	11993	18	N/A
4	section1_master_table_nr	11993		N/A
...				
36	MLAH	11993		Latitude (high accuracy)
)			
37	MLOH	11993		Longitude (high
	accuracy)			
58	MTDBT	11993		Temperature/air
	temperature			
}				

This function provides an overview of the variables, their dimensions, and descriptions if available, making it easier to understand the contents of the NetCDF file before further analysis.

Frameworks for Accessing NetCDF Observations

To work with SYNOP observations in NetCDF, we rely on established *Python* libraries such as:

- netCDF4 – Standard library for reading NetCDF data
- numpy – Efficient numerical computations
- matplotlib – Visualization of meteorological data
- cartopy – Geospatial plotting on maps

Reading SYNOP Data from NetCDF

To extract observation data such as latitude, longitude, and temperature, we use the following Python script:

Reading latitude, longitude, and temperature from a NetCDF SYNOP file

```

1 from netCDF4 import Dataset
2 import numpy as np
3
4 def read_synop_data(filename):
5     """Reads latitude (MLAH), longitude (MLOH), and temperature (MTDBT) from a
6     NetCDF file."""
7     ncfile = Dataset(filename, 'r')
8
9     lats = ncfile.variables["MLAH"][:]
10    lons = ncfile.variables["MLOH"][:]
11    temps = ncfile.variables["MTDBT"][:]
12
13    ncfile.close()
14    return np.array(lats), np.array(lons), np.array(temps)
15
16 # Example usage
17 lats, lons, temps = read_synop_data("synop.nc")
18 print("Latitudes:", lats[:5])
19 print("Longitudes:", lons[:5])
20 print("Temperatures:", temps[:5])

```

Filtering Missing Values

NetCDF files contain default missing values, e.g., 9.96921×10^{36} . Before using the data, these values should be filtered out, here we employ a simple threshold:

Filtering large missing values in SYNOP NetCDF data

```

1 def filter_missing_values.temps, threshold=1e+20):
2     """Removes large default missing values from temperature data."""
3     return temps[temps < threshold]
4
5 # Example usage
6 temps_filtered = filter_missing_values.temps)

```

Visualizing Observations on a Map

For an intuitive representation of SYNOP data, we can plot temperature observations on a map:

Scatter plot of SYNOP temperature observations

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import cartopy.crs as ccrs
4 import cartopy.feature as cfeature
5
6 def plot_temperature_map(lats, lons, temps, filename="synop.png", threshold=1e+20)
7     :

```

```

7      """Plots temperature observations on a map, removes large missing values,
8      ensures proper colorbar spacing, and saves the figure."""
9
10     # Filter out large missing values
11     valid_mask = (temps < threshold) & np.isfinite(temps)
12     lats, lons, temps = lats[valid_mask], lons[valid_mask], temps[valid_mask]
13
14     # Create figure with proper aspect ratio
15     fig, ax = plt.subplots(figsize=(10, 6), subplot_kw={'projection': ccrs.
16                               PlateCarree()})
17
18     # Scatter plot with properly scaled colorbar
19     scatter = ax.scatter(lons, lats, c=temps, cmap='coolwarm', s=5, alpha=0.7,
20                           transform=ccrs.PlateCarree())
21
22     # Add map features
23     ax.coastlines()
24     ax.add_feature(cfeature.BORDERS, edgecolor='gray')
25     ax.gridlines(draw_labels=True, linewidth=0.5, color='gray', alpha=0.5,
26                   linestyle='--')
27
28     # Add colorbar with better spacing
29     cbar = plt.colorbar(scatter, ax=ax, fraction=0.04, pad=0.08)
30     cbar.set_label("Temperature (K)")
31
32     # Set title
33     plt.title("Temperature Observations on Map")
34
35     # Save and show the plot
36     plt.savefig(filename, dpi=300, bbox_inches="tight")
37     plt.show()
38
39 # Example usage
40 plot_temperature_map(lats, lons, temps)

```

This script generates a scatter plot where each SYNOP observation is plotted on a geographical map, see Figure 3.7. The color of each point represents the observed temperature, providing a clear spatial overview of meteorological conditions.

3.4 Analysing AIREP Feedback Files in NetCDF Format

Aircraft Reports (AIREP) provide vital meteorological observations from airborne sources. These reports contain real-time measurements of parameters such as temperature, wind speed, pressure, and humidity. The data is often stored in *BUFR* (Binary Universal Form for the Representation of meteorological data) format and later converted into *NetCDF* (Network Common Data Form) for easier access and processing.

Structure of AIREP NetCDF Files

AIREP feedback files in NetCDF format consist of multiple dimensions and variables. The primary dimensions include:

- d_hdr – Number of header entries (stations, timestamps, metadata)
- d_body – Number of observed variables (measurements at different levels)
- d_veri – Number of verification entries

Each observation is associated with key metadata, including:

- lat, lon – Geographic coordinates of the observation
- varno – Variable number defining the type of measurement
- obs – Observed value, bias-corrected
- plevel – Pressure level at which the observation was made
- veri_data – Corresponding modeled values for verification

Inspecting Variables in AIREP NetCDF Files

To get an overview of the available variables, the following Python function lists all variables along with their dimensions and descriptions:

Listing NetCDF Variables in AIREP Files

```

1 from netCDF4 import Dataset
2
3 def nc_list(file1):
4     """
5         Lists all variables in a given NetCDF file, displaying their names, dimensions
6         , and descriptions.
7
8     Parameters:
9         file1 (str): Path to the NetCDF file.
10
11    Output:
12        Prints a formatted table of variables with their dimensions and descriptions.
13
14
15    print("{:<4} {:>40} {:>10} {:>10} {:>30}".format("No", "Varname", "shape1",
16        "shape2", "Description"))
17    print("-" * 110)
18
19    nc = 1
20    for varname in ncfile.variables.keys():
21        var = ncfile.variables[varname]

```

```

22     # Retrieve description from correct attribute
23     description = getattr(var, "longname", "N/A")
24
25     # Get variable dimensions
26     dims = [len(ncfile.dimensions[dim]) for dim in var.dimensions]
27
28     # Ensure at least 2 shape values
29     shape1 = dims[0] if len(dims) > 0 else ""
30     shape2 = dims[1] if len(dims) > 1 else ""
31
32     print("{}<4> {:>40} {:>10} {:>10} {:>30}".format(nc, varname, shape1,
33     shape2, description))
34
35     if nc % 10 == 0:
36         print("-" * 110)
37     nc += 1
38
39
40 # Example usage
41 file = "monAIREP.nc"
42 nc_list(file)

```

Example Output of nc_list

No	Varname	shape1	shape2	Description
1	i_body	37198		index of 1st entry in report body
2	l_body	37198		number of entries in report body
3	n_level	37198		number of levels in report
4	data_category	37198		BUFR4 data category
5	sub_category	37198		BUFR4 data sub-category
6	center	37198		station processing center
7	sub_center	37198		station processing sub-center
8	obstype	37198		observation type
9	codetype	37198		code type
10	ident	37198		station or satellite id as integer
11	statid	37198	10	station id as character string
12	lat	37198		latitude
13	lon	37198		longitude
14	time	37198		observation minus reference time
15	time_nomi	37198		nominal (synoptic) minus reference time
16	time_dbase	37198		data base minus reference time
17	z_station	37198		station height
18	z_modsurf	37198		model surface height
19	r_state	37198		status of the report
20	r_flags	37198		report quality check flags
21	r_check	37198		check which raised the report status
22	flag value			
23	sta_corr	37198		station correction indicator
23	index_x	37198		index x of model grid point assigned

```

    to report
24  index_y                         37198      index y of model grid point assigned
    to report
25  mdlsfc                           37198      model surface characteristics
26  instype                          37198      station type or satellite instrument
type
27  sun_zenit                        37198      sun zenith angle
28  phase                            37198      aircraft phase
29  tracking                          37198      tracking technique
30  obs_id                           37198      unique observation id
-----
31  source                           37198      input file number
32  record                           37198      record number in the input file
33  subset                           37198      subset number in the record
34  dbkz                            37198      DWD data base id
35  index_d                          37198      model grid diamond index assigned to
    report
36  varno                           187770     type of the observed quantity
37  obs                             187770     bias corrected observation
38  bcov                           187770     bias correction, corrected minus
    observed
39  level                           187770     level of observation
40  level_typ                       187770     type of level information
-----
41  level_sig                        187770     level significance
42  state                            187770     status of the observation
43  flags                           187770     observation quality check flags
44  check                            187770     check which raised the observation
    status flag value
45  e_o                             187770     observational error
46  qual                            187770     observation confidence from data
    provider
47  plevel                          187770     nominal pressure level
48  veri_data                        5          187770     modelled quantity (as indicated by
    veri_ens_member)
49  veri_model                       5          10        model used for verification, e.g.
    COSMO, GME ...
50  veri_run_type                   5          type of model run
-----
51  veri_run_class                  5          class of model run
...

```

Extracting AIREP Observations from NetCDF

To retrieve latitude, longitude, and observation values, we use the following function:

Reading AIREP Observations from NetCDF

```

1 from netCDF4 import Dataset
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import cartopy.crs as ccrs
5 import cartopy.feature as cfeature
6
7 def read_airep_data(filename, varno, extra_vars=None):
8     """Reads latitude, longitude, selected observations, and additional variables
        from a NetCDF file."""
9     if extra_vars is None:

```

```

10         extra_vars = []
11
12     ncf = Dataset(filename, 'r')
13
14     # Read header-level variables
15     lat = ncf.variables["lat"][:]
16     lon = ncf.variables["lon"][:]
17
18     # Read body-level variables
19     varno_all = ncf.variables["varno"][:]
20     obs_all = ncf.variables["obs"][:]
21     l_body = ncf.variables["l_body"][:]
22
23     # Expand lat/lon to match body-level observations
24     ni = len(l_body)
25     ie = np.repeat(range(0, ni), l_body)
26
27     # Find matching variable numbers
28     idx = np.where(varno_all == varno)[0]
29
30     # Filter lat, lon, obs
31     lat_filtered = lat[ie[idx]]
32     lon_filtered = lon[ie[idx]]
33     obs_filtered = obs_all[idx]
34
35     # Read extra variables if requested
36     extra_data = {}
37     for var in extra_vars:
38         if var in ncf.variables:
39             var_data = ncf.variables[var][:]
40             extra_data[var] = var_data[idx] if var_data.shape[0] == len(varno_all)
41         else:
42             var_data[ie[idx]]
43             print(f"Warning: Variable '{var}' not found in NetCDF file.")
44
45     ncf.close()
46     return lat_filtered, lon_filtered, obs_filtered, extra_data

```

Filtering Out Missing Values and Outliers

AIREP NetCDF files may contain default missing values (e.g., 9.96921×10^{36}) and unrealistic outliers. We filter them as follows:

Filtering Missing Values and Outliers in AIREP Data

```

1 def filter_airep_data(lats, lons, obs, threshold=1e+20):
2     """Filters AIREP observations by removing missing values and out-of-range
3     temperatures."""
4     valid_mask = (obs < threshold) & np.isfinite(obs)
5     lats, lons, obs = lats[valid_mask], lons[valid_mask], obs[valid_mask]

```

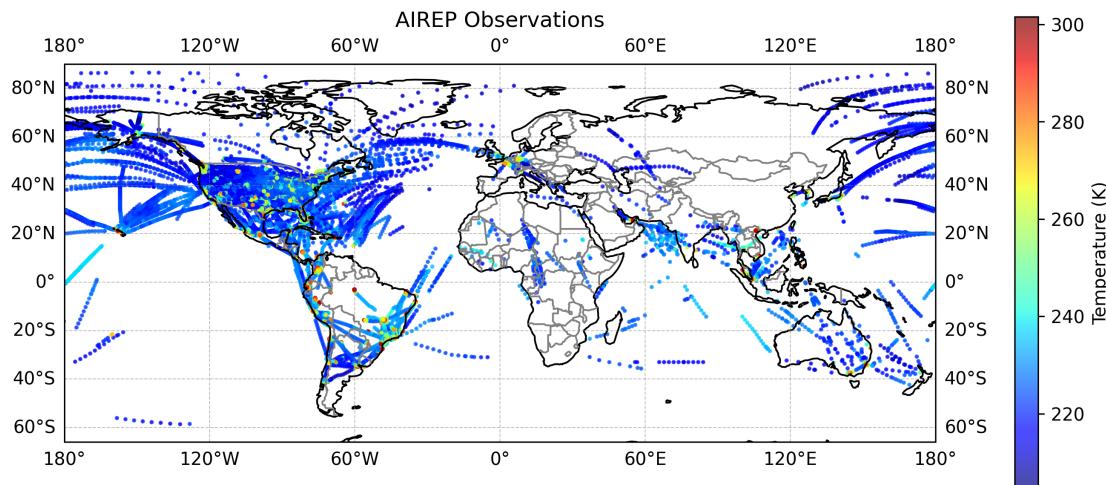


Figure 3.8: Scatter plot of AIREP observations

```

5     temp_min, temp_max = 180, 320
6     physical_mask = (obs >= temp_min) & (obs <= temp_max)
7     return lats[physical_mask], lons[physical_mask], obs[physical_mask]

```

Visualizing AIREP Observations on a Map

For a better understanding of the spatial distribution of AIREP observations, we plot them on a map using the following function:

Plotting AIREP Observations on a Map

```

1 def plot_airep_map(lats, lons, obs, filename="airep.png"):
2     """Plots AIREP observations on a map after filtering out-of-range temperatures
3     """
4
5     fig, ax = plt.subplots(figsize=(10, 6), subplot_kw={'projection': ccrs.
6         PlateCarree()})
7
8     scatter = ax.scatter(lons, lats, c=obs, cmap='jet', s=2, alpha=0.7, transform=
9         ccrs.PlateCarree())
10
11    ax.coastlines()
12    ax.add_feature(cfeature.BORDERS, edgecolor='gray')
13    ax.gridlines(draw_labels=True, linewidth=0.5, color='gray', alpha=0.5,
14        linestyle='--')
15
16    # Ensure the colorbar does not exceed figure height
17    cbar = fig.colorbar(scatter, ax=ax, orientation='vertical', fraction=0.04, pad
18        =0.08, shrink=0.8)
19    cbar.set_label("Temperature (K)")

```

```

16     plt.title("AIREP Observations")
17     plt.savefig(filename, dpi=300, bbox_inches="tight")
18     plt.show()
19
20 # Example usage
21 lats_filtered, lons_filtered, obs_filtered = filter_airep_data(lats, lons, obs)
22 plot_airep_map(lats_filtered, lons_filtered, obs_filtered)

```

The visualization of Figure 3.10 allows for a quick assessment of the coverage and accuracy of aircraft-derived meteorological data.

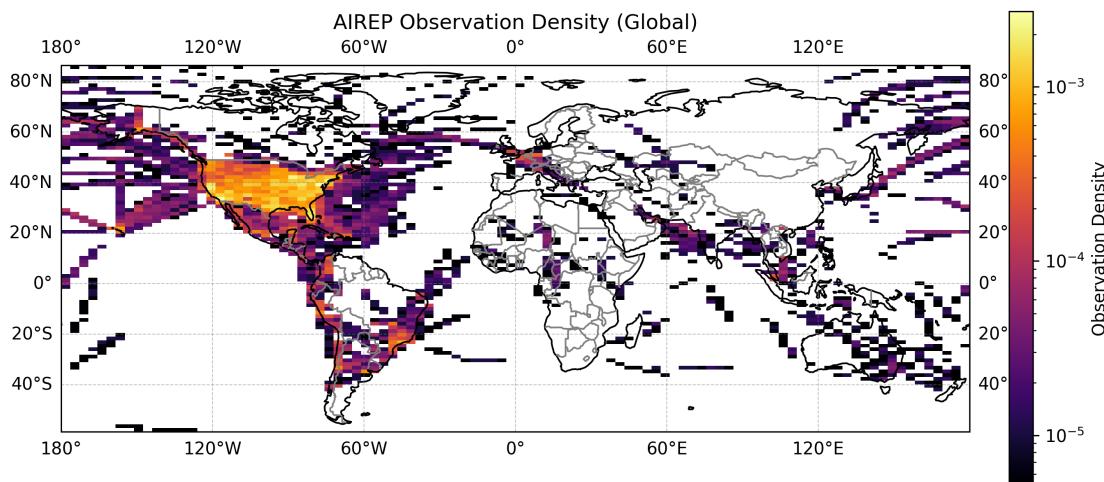


Figure 3.9: AIREP density in its horizontal distribution, while daytime in the US.

We can now analyse these data, for example by visualization of measurement density in horizontal or vertical distribution.

Global AIREP Density Visualization

```

1 import matplotlib.pyplot as plt
2 import cartopy.crs as ccrs
3 import cartopy.feature as cfeature
4 import numpy as np
5 from scipy.stats import gaussian_kde
6 from netCDF4 import Dataset
7
8 def plot_global_density(lats, lons,
9                         filename="airep_global_density.png"):
10    """Generates a density plot of AIREP observations on a world map
11    with an optimized colormap."""
12    fig, ax = plt.subplots(figsize=(10, 6),
13                          subplot_kw={'projection': ccrs.PlateCarree()})
14
15    # Compute 2D histogram
16    hist, xedges, yedges = np.histogram2d(lons, lats,

```

```

17                               bins=100, density=True)

18
19     # Use a perceptually uniform colormap (e.g., 'inferno')
20     pcm = ax.pcolormesh(xedges, yedges, hist.T, cmap='inferno',
21                           norm=plt.matplotlib.colors.LogNorm(
22                               vmin=hist[hist > 0].min(),
23                               vmax=hist.max(),
24                               transform=ccrs.PlateCarree())
25
26     ax.coastlines()
27     ax.add_feature(cfeature.BORDERS, edgecolor='gray')
28     ax.gridlines(draw_labels=True, linewidth=0.5,
29                   color='gray', alpha=0.5, linestyle='--')

30
31     # Adjust padding to ensure axis does not crowd the figure
32     plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)

33
34     # Ensure colorbar does not exceed figure size
35     cbar = fig.colorbar(pcm, ax=ax, orientation='vertical',
36                          fraction=0.04, pad=0.04, shrink=0.8)
37     cbar.set_label("Observation Density")

38
39     plt.title("AIREP Observation Density (Global)")
40     plt.savefig(filename, dpi=300, bbox_inches="tight")
41     plt.show()

```

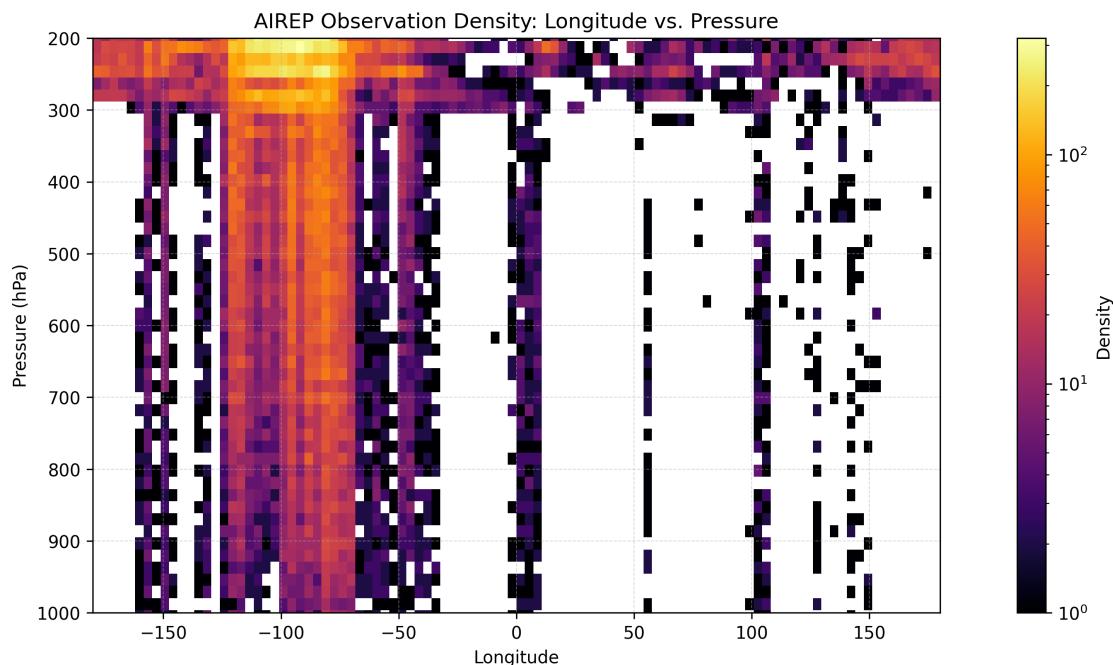


Figure 3.10: AIREP density vs vertical height in hPa and longitude.

The following code will generate a density distribution over height and longitudes.

Vertical Distribution of AIRPE Observations

```

1 def plot_height_histogram(lons, heights,
2                           filename="airep_height_density.png"):
3     """Generates a histogram of longitude vs. height, converting pressure
4     to altitude with 1000 hPa at the bottom and 200 hPa at the top."""
5     fig, ax = plt.subplots(figsize=(10, 6))
6
7     # Convert pressure Pa to hPa
8     heights = heights / 100
9
10    # Create 2D histogram
11    hist, xedges, yedges = np.histogram2d(lons, heights,
12                                         bins=(100, 50))
13
14    # Use the same optimized colormap as in global density plot
15    pcm = ax.pcolormesh(xedges, yedges, hist.T, cmap='inferno',
16                         norm=plt.matplotlib.colors.LogNorm(
17                             vmin=hist[hist > 0].min(),
18                             vmax=hist.max()))
19
20    cbar = fig.colorbar(pcm, ax=ax, orientation='vertical',
21                        fraction=0.04, pad=0.08)
22    cbar.set_label("Density")
23
24    plt.xlabel("Longitude")
25    plt.ylabel("Pressure (hPa)")
26    plt.title("AIREP Observation Density: Longitude vs. Pressure")
27    plt.ylim(1000, 200) # Invert y-axis so 1000 hPa is at bottom
28    plt.grid(True, linestyle='--', linewidth=0.5, alpha=0.5)
29
30    plt.savefig(filename, dpi=300, bbox_inches="tight")
31    plt.show()

```

3.4.1 Plotting Scalar Fields on ICON Triangular Grids

To visualize ICON model output fields on the native triangular grid, we combine the triangular mesh geometry from the ICON grid file with field values from the forecast GRIB file. This allows for accurate visualization of quantities such as temperature or land-sea mask using the `matplotlib` and `cartopy` libraries.

We demonstrate the full process below using the land-sea mask field `lsm` as an example.

Reading the ICON Grid

The ICON grid file contains the geographical coordinates of each triangle vertex (vlon, vlat) and the connectivity table (vertex_of_cell) defining which three vertices form each triangle.

Reading the ICON Grid File

```

1 import numpy as np
2 import matplotlib.tri as mtri
3 import netCDF4 as nc
4
5 gridfile = "icon_grid_0043_R02B04_G.nc"
6 print('Reading grid file:', gridfile)
7
8 with nc.Dataset(gridfile) as f:
9     vlon = f['vlon'][:] * 180 / np.pi
10    vlat = f['vlat'][:] * 180 / np.pi
11    vertex_of_cell = f['vertex_of_cell'][:] - 1 # convert from 1-based to 0-based
12      indexing
13
14    triangulation = mtri.Triangulation(vlon, vlat, vertex_of_cell.T)

```

Extracting Forecast Data from the GRIB File

Forecast fields are read from a GRIB file using the eccodes interface. We use a helper function to extract the field matching a given short name:

Extracting Field from GRIB

```

1 import eccodes
2
3 def extract_values(grib_file, sname):
4     with open(grib_file, 'rb') as f:
5         while True:
6             gid = eccodes.codes_grib_new_from_file(f)
7             if gid is None:
8                 break
9
10            short_name = eccodes.codes_get(gid, "shortName")
11            if short_name == sname:
12                values = eccodes.codes_get_array(gid, "values")
13                eccodes.codes_release(gid)
14                return values
15
16            eccodes.codes_release(gid)
17
18    raise ValueError(f"shortName '{sname}' not found in {grib_file}")

```

Reading the Field Data

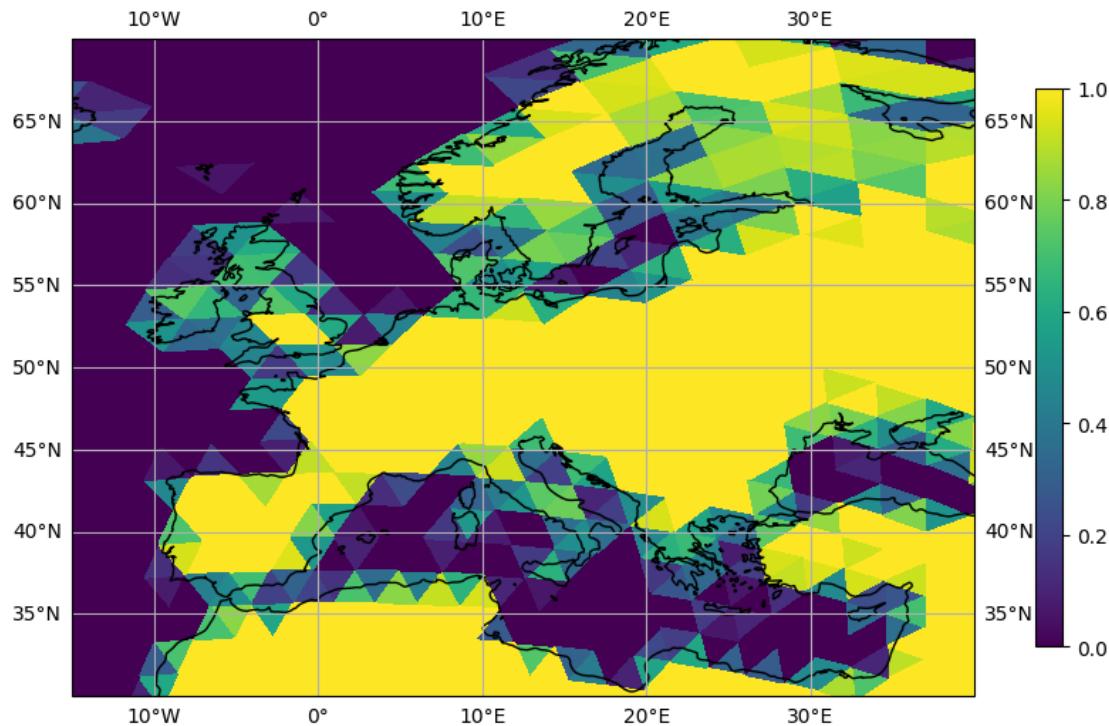
```
1 valfile = "fc_R02B04.2022010100"
2 values = extract_values(valfile, "lsm") # land-sea mask
```

Plotting the Field with Cartopy

We use cartopy to overlay the triangulated scalar field on a map. The values are visualized using tripcolor, with a colorbar for interpretation.

Plotting the ICON Triangular Field

```
1 import matplotlib.pyplot as plt
2 import cartopy.crs as ccrs
3
4 fig, ax = plt.subplots(figsize=(10, 5), subplot_kw={'projection': ccrs.PlateCarree
    ()})
5
6 # Plot the scalar field on the triangulated grid
7 tpc = ax.tripcolor(triangulation, facecolors=values, cmap='viridis', shading='flat
    ')
8
9 # Add map features
10 ax.coastlines()
11 ax.set_title("ICON Land-Sea Mask (triangular grid)")
12 plt.colorbar(tpc, ax=ax, shrink=0.8, label="Land-Sea Mask")
13
14 plt.savefig("images/img03/icon_lsm_plot.png")
15 plt.show()
```



This method can be applied to any scalar field available in the forecast file, such as surface temperature (T_G) or wind speed components ($10u$, $10v$). The same triangulation can be reused for all variables defined per triangle. The scripts will also work if fields are defined on triangle vertices, tripcolor is quite powerful.

Chapter 4

Basics of Artificial Intelligence and Machine Learning (AI/ML)

4.1 AI and ML - Basic Ideas

Artificial Intelligence (AI) and Machine Learning (ML) are transforming the way problems are approached across various fields, including geosciences, weather forecasting, climate science, language processing, and decision-making. This section introduces AI and ML from three fundamental perspectives: as a problem-solving approach, as a set of tools, and as a new paradigm for interactivity and services.

4.1.1 AI and ML as a Problem-Solving Approach

Traditional problem-solving methods rely on explicit mathematical models based on domain knowledge. These models work well for structured problems but struggle with complex, high-dimensional data. AI and ML take a different approach:

- **Data-Driven Learning:** Instead of defining rules explicitly, ML algorithms learn patterns from large datasets.
- **Universal Approximators:** Neural networks and other ML techniques act as function approximators, capable of modeling intricate relationships in data.
- **Applications Across Domains:** ML is revolutionizing fields such as weather forecasting, climate modeling, speech recognition, and autonomous systems.

One of the key concepts in ML is the approximation of an unknown function $f(x)$ using a trained model $\hat{f}(x)$. A neural network, for instance, seeks to minimize the error between the predicted and actual values:

$$\min_{\theta} \sum_{i=1}^N L(y_i, \hat{f}(x_i; \theta)), \quad (4.1)$$

where θ represents the model parameters, x_i are input features, and y_i are the corresponding target values.

Though AI/ML tools are usually universally applicable, still their reliable application and deployment needs all the domain knowledge which is traditionally acquired and necessary for classical modelling and its application. AI/ML does not replace domain knowledge, but is an additional tool and technique to make domain scientists do their work in a better way.

4.1.2 AI and ML as a Set of Tools

AI/ML development is supported by a growing ecosystem of frameworks, computational resources, and cloud-based services that make it more accessible than ever.

Core Machine Learning Frameworks. Several powerful frameworks provide the foundation for AI development:

- **PyTorch** and **TensorFlow**: Widely used deep learning libraries that allow researchers and engineers to build, train, and deploy neural networks efficiently.
- **scikit-learn**: A robust library for traditional machine learning algorithms such as regression, clustering, and decision trees.

AI as a Service. Many pre-trained AI models and APIs allow users to integrate ML functionalities without requiring deep expertise in AI model building:

- **LLMs as a Service**: Companies such as OpenAI, Google, and Meta provide access to state-of-the-art large language models via APIs.
- **On-Premise AI**: Locally installed models like Llama, Mistral, and DeepSeek enable AI applications without relying on cloud services and without the dependence on big tech companies.

Computational Resources. The performance of AI models depends heavily on the hardware and infrastructure used for training and inference:

- **Local GPUs and TPUs**: Accelerate AI computations on personal or institutional hardware.
- **Cloud-Based AI Computing**: Platforms such as AWS, Google Cloud, and Azure provide scalable computing resources.
- **Edge Computing**: Optimized AI models can run on mobile devices and embedded systems, reducing reliance on centralized servers.

4.1.3 AI and ML as a New Paradigm for Interactivity and Services

Beyond being just tools, AI and ML are reshaping how humans interact with technology and how productivity can be enhanced across various industries. However, this transformation is not without its challenges. Many AI applications promise significant efficiency gains, but they also introduce

risks such as reliability issues, ethical concerns, and the need for human oversight. Understanding these limitations is crucial to harness AI effectively.

AI-Powered Productivity. AI significantly improves efficiency and accelerates workflows:

- **Code Assistants:** AI-powered tools, such as GitHub Copilot and ChatGPT-based interfaces, assist developers in writing, debugging, and optimizing code.
- **AI in Research:** AI facilitates the analysis of large datasets, aids in hypothesis generation, and automates repetitive tasks in scientific discovery.

Critical Evaluation. While AI-enhanced productivity is often presented as a game-changer, it also brings new dependencies and challenges. AI-generated code can contain errors that are difficult to detect, and over-reliance on AI in research may lead to superficial conclusions if users fail to critically assess AI-generated insights. Furthermore, the quality of AI output is only as good as the data it is trained on, making data curation and validation essential. Detecting errors in AI algorithms requires deep knowledge of both AI tools and the specific domain of application.

AI in Decision Support. AI is increasingly integrated into decision-making processes across multiple domains:

- **Weather and Climate Services:** AI enhances forecasting models, improves risk assessment, and aids in climate trend analysis.
- **Healthcare:** AI supports medical diagnosis, enables personalized treatment recommendations, and assists in predictive analytics.
- **Autonomous Systems:** AI powers self-operating systems, including autonomous vehicles, robotics, and intelligent infrastructure.

Critical Evaluation. While AI has great potential in decision support, it also raises concerns about transparency, bias, and accountability. AI-driven forecasts and medical diagnostics must be interpretable and explainable to ensure trust. In high-stakes environments, blind reliance on AI can lead to severe consequences, making human oversight and hybrid AI-human decision-making crucial.

The Need for AI Education. As AI adoption grows, so does the necessity for education and training. For domain scientists, this means moving beyond traditional methods and integrating AI-driven approaches into their workflows.

- Mastering AI frameworks enables domain experts to develop and refine tailored solutions.
- Understanding AI-powered services is crucial for their effective and responsible integration.
- Awareness of AI ethics and limitations is essential to ensure transparency, fairness, and accountability.

Critical Evaluation. The growing need for AI education is evident, but it also presents significant challenges. Many domain experts lack formal training in AI, making interdisciplinary collaboration essential. Additionally, AI education must go beyond technical aspects to include discussions on ethical AI use, bias mitigation, and responsible development. Without a strong foundation in these areas, AI could be misused or misinterpreted, leading to unreliable results.

Many AI experts approach domain problems with the assumption that data-driven models can replace traditional expertise, often underestimating the complexity and contextual knowledge required for accurate interpretation. This overconfidence can lead to models that appear to perform well on benchmarks but fail in real-world applications due to overlooked domain-specific constraints and hidden biases.

4.1.4 Conclusion

AI and ML are more than just tools—they represent a fundamental shift in problem-solving, technology, and human-computer interaction. From universal approximators to AI-driven interactive services, these methods continue to reshape industries and scientific research. As AI adoption grows, so does the need for structured education and expertise in this rapidly evolving field.

Recommendation

The tools available today by AI/ML technology are representing a technological shift. Develop a balanced view, which sees the potential and the limitations at the same time. The shift can be compared to the development of book copying technology, radio or the invention of flight.

4.2 Torch Tensors - Basics and Their Role in Minimization

Deep learning frameworks simplify the development and deployment of machine learning models, but they must balance flexibility, efficiency, and ease of use. PyTorch has emerged as one of the most widely adopted frameworks because it combines an intuitive, Pythonic interface with powerful automatic differentiation and dynamic computation graph capabilities. Unlike static graph-based frameworks, PyTorch allows for more flexible model development, making it particularly useful for research, experimentation, and rapid prototyping. Its seamless GPU acceleration, built-in support for deep learning libraries, and strong community adoption make it a critical tool for both academic and industrial AI applications.

Torch tensors are the fundamental data structures in PyTorch. They are similar to NumPy arrays but come with additional capabilities, such as GPU acceleration and automatic differentiation, which are essential for training neural networks. In particular, tensors with the attribute `requires_grad=True` allow PyTorch to automatically compute gradients, a key component in optimization algorithms like gradient descent.

Below, we illustrate basic tensor operations and demonstrate how tensors enable minimization through gradient computation.

Tensor Operations and Gradients

```

1 import torch
2
3 # Create a tensor from a Python list
4 a = torch.tensor([1.0, 2.0, 3.0])
5 print("Tensor a:", a)
6

```

```

7 # Create a 3x3 tensor with random values
8 b = torch.rand(3, 3)
9 print("Random tensor b:\n", b)
10
11 # Perform arithmetic: multiply tensor 'a' by 2
12 c = a * 2
13 print("Tensor c (a multiplied by 2):", c)
14
15 # For minimization, we need tensors that track gradients.
16 # Create a tensor with requires_grad=True so that operations on it are tracked.
17 x = torch.tensor([2.0, 3.0], requires_grad=True)
18
19 # Define a simple quadratic function: f(x) = x[0]^2 + x[1]^2
20 y = x[0]**2 + x[1]**2
21
22 # Compute gradients with respect to x using backpropagation
23 y.backward()
24
25 # The gradients of y with respect to x are stored in x.grad
26 print("Gradients of y with respect to x:", x.grad)

```

Output:

```

Tensor a: tensor([1., 2., 3.])
Random tensor b:
tensor([[0.3450, 0.2811, 0.0059],
       [0.6343, 0.5166, 0.4793],
       [0.3613, 0.5797, 0.5450]])
Tensor c (a multiplied by 2): tensor([2., 4., 6.])
Gradients of y with respect to x: tensor([4., 6.])

```

In this example, we compute the gradient of a quadratic function, which is a common operation in optimization tasks. When training a neural network, the loss function is minimized by iteratively updating the model parameters (stored as tensors) based on their computed gradients. This automatic differentiation capability is crucial for efficient and effective model training.

Recommendation

Automatic gradient calculation, optimization, and learning are at the core of the technological transformation.

4.3 PyTorch Fundamentals - Model, Loss, and Optimizer

First, let us install the necessary packages in our Python virtual environment. This step is assumed to be done already, we discussed how to install python packages in various environments in depth in the preceding parts of this tutorial.

Now, in your Python program, either directly in a .py file or a Jupyter Notebook, you need to import the required packages.

Torch Packages

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim

```

Next, we define a dataset to train on. In many of our examples we create synthetic data. For instance, you may generate random data for regression or classification tasks, or, as in the sine curve example below, data derived from mathematical functions. Often, the dataset is split into training and testing subsets to evaluate model performance and to prevent overfitting.

Below is an example code that sets up training and test data for a generic regression task. Here, we generate random input features and corresponding target values:

Synthetic Data

```

1 import torch
2 import numpy as np
3 from torch.utils.data import TensorDataset, DataLoader
4
5 # Generate synthetic data: 100 samples with 10 features each
6 X = np.random.rand(100, 10)
7 y = np.random.rand(100, 1)
8
9 # Convert numpy arrays to torch tensors
10 X_tensor = torch.tensor(X, dtype=torch.float32)
11 y_tensor = torch.tensor(y, dtype=torch.float32)
12
13 # Create a TensorDataset and then split it into training and test sets
14 dataset = TensorDataset(X_tensor, y_tensor)
15 train_size = int(0.8 * len(dataset))
16 test_size = len(dataset) - train_size
17 train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size,
18     test_size])
19
20 # Diagnostic Output
21 print(f"Train dataset size: {len(train_dataset)}, Test dataset size: {len(
22     test_dataset)}")
23
24 # Show shapes of the first batch
25 first_train_sample = train_dataset[0]
26 print(f"First training sample - X shape: {first_train_sample[0].shape}, y shape: {
27     first_train_sample[1].shape}")
28
29 # Show content of the first training sample
30 print(f"First training sample - X: {first_train_sample[0].numpy()}, y: {
31     first_train_sample[1].numpy()}")

```

Output:

Train dataset size: 80, Test dataset size: 20

```
First training sample - X shape: torch.Size([10]), y shape: torch.Size([1])
First training sample - X: [0.2555835  0.13075094  0.1967931   0.3170668   0.08261041
                           0.68258333
                           0.92773515  0.7652774   0.07989042  0.28203908], y: [0.06629485]
```

Now, let us define a simple neural network with one hidden layer. This network consists of an input layer, one hidden layer with a ReLU activation, and an output layer.

Simple NN Model

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4
5 class SimpleNN(nn.Module):
6     def __init__(self, input_size, hidden_size, output_size):
7         super(SimpleNN, self).__init__()
8         self.fc1 = nn.Linear(input_size, hidden_size)
9         self.relu = nn.ReLU()
10        self.fc2 = nn.Linear(hidden_size, output_size)
11
12    def forward(self, x):
13        x = self.fc1(x)
14        x = self.relu(x)
15        x = self.fc2(x)
16        return x
17
18 # Instantiate the model
19 input_size = 10
20 hidden_size = 16
21 output_size = 1
22 model = SimpleNN(input_size, hidden_size, output_size)
23
24 # Define the loss function and optimizer
25 criterion = nn.MSELoss()
26 optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
27
28 print(model)
```

Output:

```
SimpleNN(
  (fc1): Linear(in_features=10, out_features=16, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=16, out_features=1, bias=True)
)
```

The script begins by importing the necessary PyTorch modules: `torch` for core functionalities, `torch.nn` for neural network components, and `torch.optim` for optimization algorithms.

A simple feedforward neural network is defined using the `SimpleNN` class, which inherits from `nn.Module`. The network consists of two fully connected layers. The first layer (`fc1`) maps the input

features to a hidden layer, followed by a ReLU activation function to introduce non-linearity. The second layer (fc2) maps the hidden layer to the output layer. The forward pass is computed as:

$$x = \text{fc1}(x) \rightarrow \text{ReLU}(x) \rightarrow \text{fc2}(x). \quad (4.2)$$

The model is instantiated with three parameters: `input_size` = 10, representing the number of input features, `hidden_size` = 16, defining the number of neurons in the hidden layer, and `output_size` = 1, indicating a single output value, which is appropriate for regression tasks.

To train the model, the loss function is set to Mean Squared Error (MSE), given by:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2. \quad (4.3)$$

The Adam optimizer is used to update the model parameters with a learning rate of 0.01.

Finally, the model architecture is printed to verify its structure.

The Adam Optimizer. The Adam optimizer (Adaptive Moment Estimation) uses the first moment estimate m_t and the second moment estimate v_t to compute an adaptive learning rate for each parameter.

1. *First moment estimate m_t :* This is an exponentially weighted moving average of past gradients, representing a smoothed estimate of the mean gradient:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t. \quad (4.4)$$

Since m_t starts from zero, Adam applies bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}. \quad (4.5)$$

This correction ensures that \hat{m}_t is an unbiased estimate of the true gradient expectation.

2. *Second moment estimate v_t :* This is an exponentially weighted moving average of past squared gradients, approximating the variance of the gradient:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2. \quad (4.6)$$

Similar to m_t , Adam applies bias correction:

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \quad (4.7)$$

This correction ensures that \hat{v}_t is an unbiased estimate of the second moment.

3. *Parameter update:* Using the corrected estimates \hat{m}_t and \hat{v}_t , Adam updates the parameters θ as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \quad (4.8)$$

Here, η is the learning rate, and ϵ is a small constant to prevent division by zero.

In summary, Adam normalizes the gradient update using the estimated first and second moments, allowing each parameter to have an individual learning rate that adapts to the scale of its gradients. This leads to more stable and efficient optimization compared to standard gradient descent.

4.3.1 Data Handling - Dataset and DataLoader

Neural networks are typically trained on large datasets, making it inefficient to load all data into memory at once. Instead, **data loaders** are used to efficiently handle batch processing, shuffling, and parallel loading.

Given a dataset with input samples $\mathbf{X} = \{x_1, x_2, \dots, x_N\}$ and corresponding labels $\mathbf{Y} = \{y_1, y_2, \dots, y_N\}$, a data loader divides the dataset into mini-batches of size B . At each training step, the model processes a batch:

$$(\mathbf{X}_B, \mathbf{Y}_B) = \{(x_i, y_i)\}_{i=1}^B. \quad (4.9)$$

Key advantages of using data loaders include:

- **Memory efficiency:** Only small batches are loaded into memory at a time.
- **Shuffling:** Randomizing data order prevents overfitting to specific patterns.
- **Parallel processing:** Multiple CPU threads can load data asynchronously.

In PyTorch, a DataLoader automates these tasks, enabling efficient training on large datasets.

After installing the necessary packages, you can use PyTorch's DataLoader to efficiently handle data in mini-batches. This is particularly useful for training, as it enables you to iterate over the dataset in smaller chunks, reducing memory usage and often improving convergence. Here's an example using synthetic data with a TensorDataset.

We use the tensors `X_tensor` and `y_tensor` from above.

Data Loader

```

1 from torch.utils.data import DataLoader
2
3 # Create a DataLoader for the training dataset
4 dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
5
6 # Example: Iterate through one batch
7 n = 1
8 for batch_X, batch_y in dataloader:
9     print(f"{n}) Batch X shape:", batch_X.size())
10    print("      Batch y shape:", batch_y.size())
11    n += 1

```

Output:

```

1) Batch X shape: torch.Size([16, 10])
   Batch y shape: torch.Size([16, 1])
2) Batch X shape: torch.Size([16, 10])
   Batch y shape: torch.Size([16, 1])
3) Batch X shape: torch.Size([16, 10])
   Batch y shape: torch.Size([16, 1])
4) Batch X shape: torch.Size([16, 10])
   Batch y shape: torch.Size([16, 1])
5) Batch X shape: torch.Size([16, 10])

```

```
Batch y shape: torch.Size([16, 1])
Selection deleted
```

In the example from above, the dataset contains $N = 100$ samples, which is split into:

- **Training set:** 80 samples
- **Test set:** 20 samples

The script creates a DataLoader that loads batches of data from the training dataset. Each batch consists of 16 samples, with:

- **Batch X shape:** (16, 10), meaning each batch contains 16 feature vectors, each with 10 features.
- **Batch y shape:** (16, 1), meaning each batch contains 16 target values, each a scalar.

1. The loop runs exactly 5 times because the dataset contains 80 training samples, and the batch size is 16.
2. Each iteration produces a batch of size 16, confirming that the DataLoader correctly partitions the dataset.
3. Since `shuffle=True`, the data order is randomized, ensuring that each epoch has a different sample arrangement.

The DataLoader successfully partitions the dataset into evenly sized mini-batches, verifying that the batch processing mechanism functions as expected.

4.4 Simple Neural Network Training Example

We now develop an example that demonstrates how to approximate the sine function using a simple neural network built with PyTorch. We generate data from the sine curve, create a dataset with a DataLoader for mini-batch training, define a neural network model, train it using mean squared error loss, and finally plot the network's predictions against the actual sine values.

Sine Curve Approximation

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from torch.utils.data import TensorDataset, DataLoader
7
8 # Generate dataset for sine curve approximation
9 x = np.linspace(0, 2 * np.pi, 1000)
10 y = np.sin(x)
11
```

```

12 # Convert numpy arrays to torch tensors and add a feature dimension
13 x_tensor = torch.tensor(x, dtype=torch.float32).unsqueeze(1)
14 y_tensor = torch.tensor(y, dtype=torch.float32).unsqueeze(1)
15
16 # Create a TensorDataset and DataLoader for batch processing
17 dataset = TensorDataset(x_tensor, y_tensor)
18 dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
19
20 # Define a simple neural network model
21 class SineModel(nn.Module):
22     def __init__(self):
23         super(SineModel, self).__init__()
24         self.net = nn.Sequential(
25             nn.Linear(1, 16),
26             nn.ReLU(),
27             nn.Linear(16, 16),
28             nn.ReLU(),
29             nn.Linear(16, 1)
30         )
31
32     def forward(self, x):
33         return self.net(x)
34
35 model = SineModel()
36
37 # Set up the loss function and optimizer
38 criterion = nn.MSELoss()
39 optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
40
41 # Training loop
42 num_epochs = 500
43 for epoch in range(num_epochs):
44     for batch_x, batch_y in dataloader:
45         optimizer.zero_grad()
46         outputs = model(batch_x)
47         loss = criterion(outputs, batch_y)
48         loss.backward()
49         optimizer.step()
50     if (epoch + 1) % 100 == 0:
51         print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
52
53 # Generate predictions after training
54 with torch.no_grad():
55     predicted = model(x_tensor).detach().numpy()
56
57 # Plot the actual sine curve and the network's predictions
58 plt.figure(figsize=(8, 4))
59 plt.plot(x, y, label='Actual Sine')
60 plt.plot(x, predicted, label='Predicted Sine', linestyle='--')

```

```

61 plt.xlabel('x')
62 plt.ylabel('sin(x)')
63 plt.legend()
64 plt.savefig("sine_approximation.png")

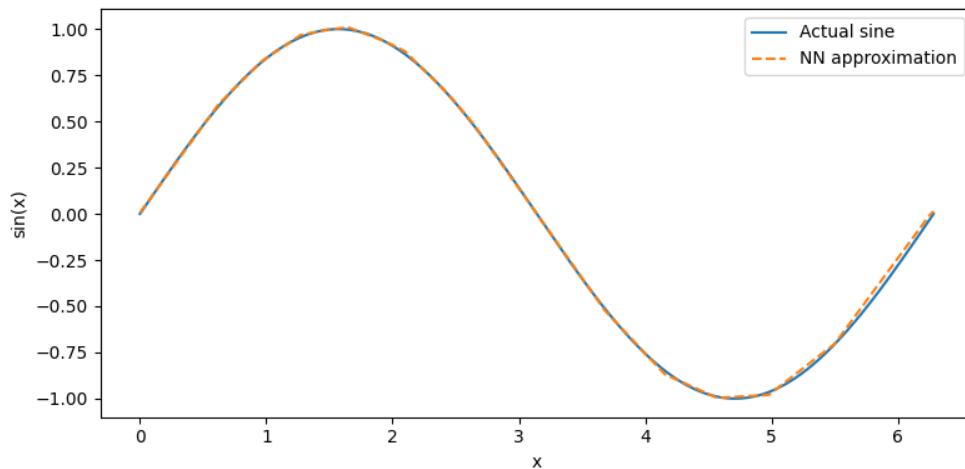
```

Output:

```

Epoch [100/500], Loss: 0.0007
Epoch [200/500], Loss: 0.0005
Epoch [300/500], Loss: 0.0002
Epoch [400/500], Loss: 0.0003
Epoch [500/500], Loss: 0.0002

```

Generated Image:

This code implements a neural network in PyTorch to approximate the sine function using supervised learning.

Dataset Generation: The input values are generated using:

$$x = \text{linspace}(0, 2\pi, 1000). \quad (4.10)$$

The target values are computed as:

$$y = \sin(x). \quad (4.11)$$

The NumPy arrays are converted into PyTorch tensors, with an additional feature dimension added using `unsqueeze(1)`.

DataLoader for Batch Processing: A `TensorDataset` is created, containing the input-output pairs (x, y) , and a `DataLoader` is used with a batch size of 32 and shuffling enabled.

Neural Network Model: The model consists of a simple feedforward neural network with:

- An input layer with 1 neuron.
- Two hidden layers with 16 neurons each, followed by ReLU activation.
- An output layer with 1 neuron.

Mathematically, the forward pass is:

$$\hat{y} = W_3(\max(0, W_2(\max(0, W_1x + b_1)) + b_2)) + b_3. \quad (4.12)$$

Loss Function and Optimizer: The Mean Squared Error (MSE) loss function is used:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2. \quad (4.13)$$

The optimizer is Adam with a learning rate of 0.01.

Training Process: The model is trained for 500 epochs. In each epoch:

1. Gradients are reset using `optimizer.zero_grad()`.
2. Predictions are computed with `model(batch_x)`.
3. The loss is calculated using `criterion(outputs, batch_y)`.
4. Backpropagation updates the weights via `loss.backward()` and `optimizer.step()`.

A progress message is printed every 100 epochs.

Prediction and Visualization: After training, the model predicts values for the entire dataset, and the results are plotted:

- The original sine function is plotted as a solid line.
- The neural network's predictions are plotted as a dashed line.

The resulting plot is saved as `sine_approximation.png`.

Programming with tensors in PyTorch requires careful handling to ensure that the automatic differentiation mechanism remains intact. PyTorch's computational graphs track tensor operations dynamically, allowing gradients to be computed automatically via backpropagation.

If operations are performed outside the tensor framework—such as converting tensors to NumPy arrays and then performing computations—the graph structure is lost, and gradient tracking is broken! This disrupts the minimization process, making parameter updates impossible.

To maintain gradient tracking, all computations within the model and loss function must be conducted using PyTorch tensor operations. Additionally, tensors should be created with `requires_grad=True` when gradients are needed, and `detach()` should be used only when explicitly removing a tensor from the computational graph, such as for inference or visualization.

Proper tensor management ensures that PyTorch can fully automate gradient computations, enabling efficient and correct optimization.

Recommendation

Use the sine approximation as generic example, what AI/ML approximators do. Nonlinear mappings are approximated. Scaling this to a huge space, very high-dimensional non-linear mappings like language generation or weather prediction are approximated.

4.4.1 Understanding the PyTorch DataLoader with and without Shuffling

The DataLoader in PyTorch is used to load data in batches, which is particularly useful for training models efficiently. To explore how it works, we create a synthetic dataset where the features in each row encode their row and column positions explicitly, making the effect of shuffling easy to observe.

We first define a tensor X of shape 100×10 , where each element is set to its row index plus a column offset. The labels y are simply the integers from 1 to 100.

Creating Structured Data for DataLoader

```

1 import torch
2 from torch.utils.data import TensorDataset, DataLoader
3
4 # Create data: 100 rows, 10 columns with visible row and column info
5 X = torch.zeros(100, 10, dtype=torch.float32)
6 for i in range(100):
7     for j in range(10):
8         X[i, j] = (i + 1) + (j / 10)
9 print(X[:10, :])
10
11 # Labels: y = 1 to 100
12 y = torch.arange(1, 101, dtype=torch.float32).reshape(-1, 1)

```

We then wrap the data in a TensorDataset and pass it to a DataLoader with batch size 4 and no shuffling. This means that the data will be returned in sequential order.

DataLoader without Shuffling

```

1 dataset = TensorDataset(X, y)
2 loader = DataLoader(dataset, batch_size=4, shuffle=False)
3
4 # Print the first batch with one decimal digit
5 print("First batch (1 digit precision):")
6 for batch_X, batch_y in loader:
7     for i in range(len(batch_X)):
8         x_row = [f"{v:.1f}" for v in batch_X[i]]
9         y_val = f"{batch_y[i].item():.1f}"
10        print(f"x = {x_row}, y = {y_val}")
11        break

```

The output of this batch shows that the first 4 rows are returned in order:

```

First batch (1 digit precision):
x = ['1.0', '1.1', '1.2', '1.3', '1.4', '1.5', '1.6', '1.7', '1.8', '1.9'], y = 1.0
x = ['2.0', '2.1', '2.2', '2.3', '2.4', '2.5', '2.6', '2.7', '2.8', '2.9'], y = 2.0
x = ['3.0', '3.1', '3.2', '3.3', '3.4', '3.5', '3.6', '3.7', '3.8', '3.9'], y = 3.0
x = ['4.0', '4.1', '4.2', '4.3', '4.4', '4.5', '4.6', '4.7', '4.8', '4.9'], y = 4.0

```

Now we repeat the same process, but enable shuffling in the DataLoader. This causes the rows to be returned in random order at the start of each epoch.

DataLoader with Shuffling

```

1 loader2 = DataLoader(dataset, batch_size=4, shuffle=True)
2
3 # Print the first batch with one decimal digit
4 print("First batch (1 digit precision):")
5 for batch_X, batch_y in loader2:
6     for i in range(len(batch_X)):
7         x_row = [f"{v:.1f}" for v in batch_X[i]]
8         y_val = f"{batch_y[i].item():.1f}"
9         print(f"x = {x_row}, y = {y_val}")
10    break

```

This will now print a different (random) batch each time the code is run. For example:

```

First batch (1 digit precision):
x = ['91.0', '91.1', ..., '91.9'], y = 91.0
x = ['39.0', '39.1', ..., '39.9'], y = 39.0
x = ['61.0', '61.1', ..., '61.9'], y = 61.0
x = ['10.0', '10.1', ..., '10.9'], y = 10.0

```

This clear example demonstrates how the PyTorch DataLoader works and how shuffling can be used to randomize input order during training.

4.5 Gradient Field and Decision Boundary

Neural networks provide flexible solutions to complex classification problems. Here, we construct an example where the decision boundary is **highly nonlinear**, making it challenging for traditional linear classifiers.

We utilize PyTorch's automatic differentiation to analyze the **gradient field** of the classification function, revealing the sensitivity of the learned model in different regions.

Generating Data with Two Shifted Ellipses. To illustrate this, we generate synthetic data where points belong to **one of two elliptical regions**, each with different orientations and positions.

Data for Classification

```

1 # Set random seed for reproducibility
2 torch.manual_seed(42)
3 np.random.seed(42)
4
5 N = 900 # Number of samples
6
7 # Generate the same random points in the range [-2, 2] x [-2, 2]
8 X = torch.rand(N, 2) * 4 - 2 # Unchanged points

```

```

9
10 # Define parameters for two smaller, shifted ellipses
11 a1, b1 = 1.0, 0.5
12 a2, b2 = 0.6, 0.9
13 theta1 = np.radians(30)
14 theta2 = np.radians(-45)
15 center1 = torch.tensor([0.9, 0.9])
16 center2 = torch.tensor([-1.1, -0.2])
17
18 X_shifted1 = X - center1
19 X_shifted2 = X - center2
20
21 x1_rot = X_shifted1[:, 0] * np.cos(theta1) + X_shifted1[:, 1] * np.sin(theta1)
22 y1_rot = -X_shifted1[:, 0] * np.sin(theta1) + X_shifted1[:, 1] * np.cos(theta1)
23 inside_ellipse1 = ((x1_rot / a1) ** 2 + (y1_rot / b1) ** 2) < 1
24
25 x2_rot = X_shifted2[:, 0] * np.cos(theta2) + X_shifted2[:, 1] * np.sin(theta2)
26 y2_rot = -X_shifted2[:, 0] * np.sin(theta2) + X_shifted2[:, 1] * np.cos(theta2)
27 inside_ellipse2 = ((x2_rot / a2) ** 2 + (y2_rot / b2) ** 2) < 1
28
29 labels = (inside_ellipse1 | inside_ellipse2).float().unsqueeze(1).numpy()
30
31 plt.figure(figsize=(7, 5))
32 plt.scatter(X[:, 0], X[:, 1], c=labels.squeeze(), cmap="bwr", alpha=1, edgecolors=
   "white")
33 plt.xlabel("Feature 1")
34 plt.ylabel("Feature 2")
35 plt.title("Labels Defined by Two Smaller, Shifted Ellipses")
36 plt.xlim(-2, 2)
37 plt.ylim(-2, 2)
38 plt.grid()
39 plt.colorbar()
40 plt.savefig("points_labeled.png")
41 plt.show()

```

This script:

- Generates $N = 900$ random points within the range $[-2, 2] \times [-2, 2]$.
- Assigns labels based on **two ellipses with different centers and rotations**.
- Uses the **blue-white-red (BWR) color map** to differentiate classes.
- Saves the figure for later comparison.

Training a Neural Network for Classification. We define a **simple feedforward neural network** with a single fully connected layer that maps the **two-dimensional input** to a binary classification output using the sigmoid activation function.

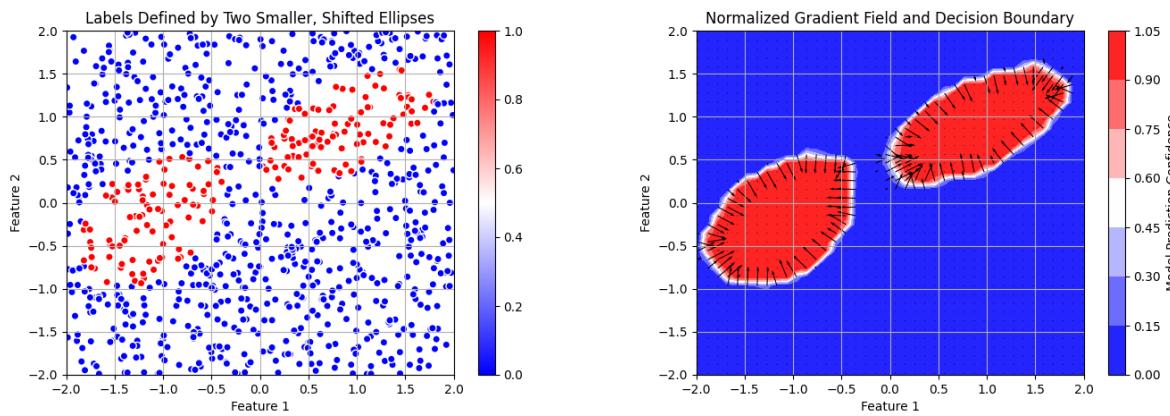


Figure 4.1: The left figure shows the original data with class labels, while the right figure presents the classification result with **gradient information** extracted from the trained model.

SimpleClassifier

```

1 class BetterClassifier(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.net = nn.Sequential(
5             nn.Linear(2, 32),
6             nn.ReLU(),
7             nn.Linear(32, 1),
8             nn.Sigmoid()
9         )
10
11    def forward(self, x):
12        return self.net(x)
13
14 model = BetterClassifier()

```

The model consists of:

- A **fully connected layer** mapping two input features to a single output.
- A **sigmoid activation function** to produce probabilities.

Next, we train the model using the **binary cross-entropy loss function** and the **Adam optimizer**.

Classifier Training Loop

```

1 import torch.optim as optim
2
3 criterion = nn.BCELoss()
4 optimizer = optim.Adam(model.parameters(), lr=0.01)
5
6 num_epochs = 1000

```

```

7 for epoch in range(num_epochs):
8     optimizer.zero_grad()
9     y_pred = model(X)
10    loss = criterion(y_pred, torch.tensor(labels, dtype=torch.float32))
11    loss.backward()
12    optimizer.step()
13
14    if (epoch + 1) % 200 == 0:
15        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")

```

Decision Boundary and Gradient Visualization. Once trained, the model is evaluated on a dense grid of points spanning the same input range $[-2, 2] \times [-2, 2]$. This allows us to visualize the decision boundary and analyze the gradient of the labels (classification) with respect to the features (input).

Display of Classification and Gradients

```

1
2 x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
3 y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
4 xx, yy = torch.meshgrid(torch.linspace(x_min, x_max, 50),
5                           torch.linspace(y_min, y_max, 50),
6                           indexing='ij')
7
8 grid_points = torch.stack([xx.flatten(), yy.flatten()], dim=1)
9 grid_points.requires_grad = True
10
11 grid_preds = model(grid_points)
12 grid_preds.backward(torch.ones_like(grid_preds))
13 grid_grads = grid_points.grad.detach().numpy()
14
15 grad_magnitudes = np.linalg.norm(grid_grads, axis=1, keepdims=True)
16 grad_magnitudes = np.clip(grad_magnitudes, 1, 1000)
17 grid_grads /= grad_magnitudes
18
19 grid_grads_x = grid_grads[:, 0].reshape(xx.shape)
20 grid_grads_y = grid_grads[:, 1].reshape(xx.shape)
21 grid_preds_np = grid_preds.detach().numpy().reshape(xx.shape)
22
23 plt.figure(figsize=(7, 5))
24 plt.contourf(xx, yy, grid_preds_np, alpha=1, cmap="bwr")
25 plt.colorbar(label="Model Prediction Confidence")
26 plt.quiver(xx, yy, grid_grads_x, grid_grads_y, color="black", scale=20)
27 plt.xlabel("Feature 1")
28 plt.ylabel("Feature 2")
29 plt.title("Normalized Gradient Field and Decision Boundary")
30 plt.xlim(-2, 2)
31 plt.ylim(-2, 2)
32 plt.grid()

```

```
33 plt.savefig("points_classified_with_gradients.png")
34 plt.show()
```

This script computes model predictions over a uniform 50×50 grid, extracts gradients to analyze the sensitivity of the classifier, normalizes the gradients to limit their maximum size, uses contour plots to show the learned decision boundary, and overlays quiver arrows to indicate the gradient field.

Observations. The decision boundary adapts to the elliptical structures. The gradient arrows show where the model is most sensitive. Large gradients appear near the decision boundary, where small changes in input strongly impact classification.

The final visualization provides **deep insights into how the neural network classifies data**, demonstrating the potential of PyTorch's **autograd system** for analyzing decision boundaries.

Recommendation

AI/ML techniques provide a rather simple approach to solve a large variety of problems. How will physical arguments and further knowledge about the particular domain or problem under consideration enter the algorithmic approach and further discussion? There is a huge gap in domain specific input and how to combine it with generic approximation tools as given by AI/ML. We need to further develop the approaches we are using here.

Chapter 5

Neural Network Architectures

5.1 Feed Forward Networks

A Feed Forward Neural Network (FFNN) is the simplest type of artificial neural network. It consists of layers of neurons where each neuron in one layer is connected to every neuron in the next layer. The information moves in one direction—forward—from the input nodes through the hidden layers (if any) to the output nodes.

FFNNs are commonly used for tasks like regression and classification. A simple implementation in Python using PyTorch is shown below.

Feedforward Neural Network with Rainbow Layers

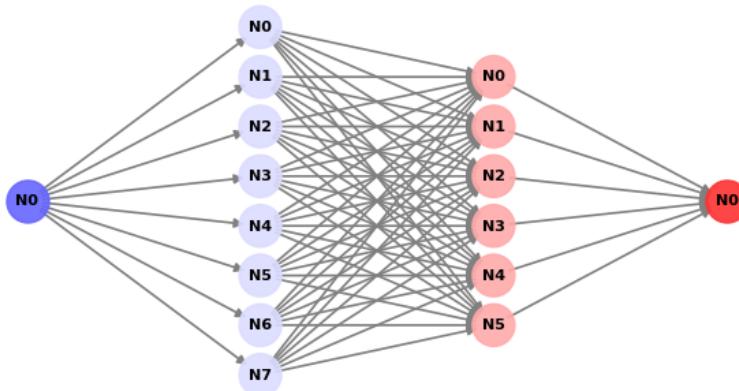


Figure 5.1: Visualization of a simple Feedforward Neural Network (FFNN) with one hidden layer. The input, hidden, and output layers are aligned from left to right, with connections representing weight relationships.

Feed Forward Network

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
```

```

4
5 # Define a deeper FFNN with two hidden layers
6 class FeedForwardNN(nn.Module):
7     def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
8         super(FeedForwardNN, self).__init__()
9         self.fc1 = nn.Linear(input_size, hidden_size1)
10        self.relu1 = nn.ReLU()
11        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
12        self.relu2 = nn.ReLU()
13        self.fc3 = nn.Linear(hidden_size2, output_size)
14
15    def forward(self, x):
16        x = self.fc1(x)
17        x = self.relu1(x)
18        x = self.fc2(x)
19        x = self.relu2(x)
20        x = self.fc3(x)
21        return x
22
23 # Create a model instance with 1 input, 8 neurons in the first hidden layer,
24 # 6 neurons in the second hidden layer, and 1 output
25 input_size, hidden_size1, hidden_size2, output_size = 1, 8, 6, 1
26 model = FeedForwardNN(input_size, hidden_size1, hidden_size2, output_size)
27
28 # Print model architecture
29 print(model)

```

A feedforward neural network (FFNN) consists of layers of interconnected neurons that transform input data into predictions. In this implementation, the network has an input layer with one neuron, two hidden layers with eight and six neurons, respectively, and an output layer with a single neuron. Each hidden layer applies a ReLU activation function to introduce non-linearity, enabling the model to learn complex relationships. The final output layer performs a linear transformation.

The weights and biases of the network are learned during training through backpropagation, minimizing a chosen loss function. PyTorch's 'nn.Linear' modules define fully connected layers, while the 'forward' method specifies how data flows through the network. The model instance is created with predefined input, hidden, and output dimensions, and printing it reveals its architecture.

We now use such a feedforward network to approximate a non-linear curve.

Learning a curve with FFNN

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 # Set random seed & generate data: f(x) = 1 / (1 + exp(-tau * x + s))
8 torch.manual_seed(42); np.random.seed(42)

```

```

9 x = np.linspace(-2, 2, 500)
10 y = 1 / (1 + np.exp(-5 * x)) # tau = 5, s = 0
11 x_tensor = torch.tensor(x, dtype=torch.float32).unsqueeze(1)
12 y_tensor = torch.tensor(y, dtype=torch.float32).unsqueeze(1)
13
14 # Define a deeper FFNN
15 class DeepFFNN(nn.Module):
16     def __init__(self):
17         super().__init__()
18         self.fc1, self.fc2, self.fc3 = nn.Linear(1, 8), nn.Linear(8, 6), nn.Linear
19             (6, 1)
20     def forward(self, x): return self.fc3(torch.relu(self.fc2(torch.relu(self.fc1(
21         x))))))
22
23 model = DeepFFNN()
24 criterion = nn.MSELoss()
25 optimizer = optim.Adam(model.parameters(), lr=0.01)
26
27 # Training
28 loss_history = []
29 for epoch in range(2000):
30     optimizer.zero_grad()
31     y_pred = model(x_tensor)
32     loss = criterion(y_pred, y_tensor)
33     loss.backward()
34     optimizer.step()
35     loss_history.append(loss.item())
36     if (epoch + 1) % 500 == 0: print(f"Epoch {epoch+1}, Loss: {loss.item():.6f}")
37
38 # Generate predictions
39 with torch.no_grad(): y_pred_np = model(x_tensor).numpy()
40
41 # Plot function approximation & loss curve
42 fig, axes = plt.subplots(1, 2, figsize=(10, 3))
43 axes[0].plot(x, y, label="True", linewidth=2)
44 axes[0].plot(x, y_pred_np, "r--", label="NN Approx.", linewidth=2)
45 axes[0].set(title="Function Approximation", xlabel="x", ylabel="f(x)"); axes[0].
46     legend(); axes[0].grid()
47 axes[1].semilogy(loss_history, "r", label="Loss")
48 axes[1].set(title="Loss Curve", xlabel="Epochs", ylabel="MSE"); axes[1].legend();
49     axes[1].grid()
50 plt.savefig("deep_nn_results.png")
51 plt.show()

```

A computational graph visually represents how data flows through a neural network during a forward pass. In this example, we use the `torchviz` library to generate a graph of the feedforward neural network (FFNN). The input tensor is a randomly generated vector with the same dimensionality as the input layer. The forward pass computes the predicted output, which is then passed to `make_dot()` along with the model's parameters. The resulting graph shows the dependencies

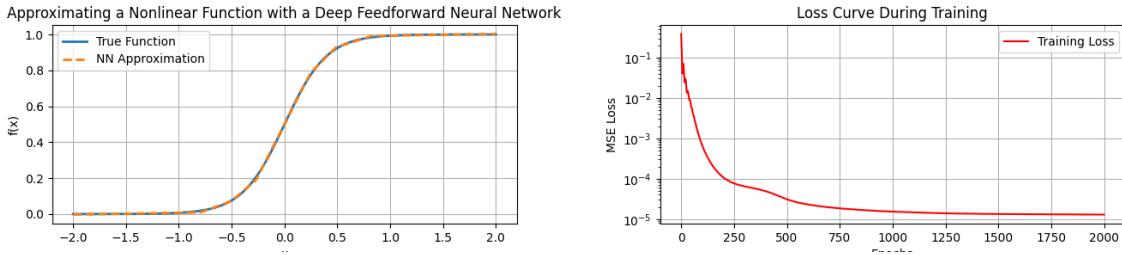


Figure 5.2: Left: Neural network approximation of the function $f(x) = \frac{1}{1+e^{-rx+s}}$. Right: Training loss curve over epochs, showing convergence of the model.

between layers and operations, helping to analyze the network structure and gradient flow.

Generating a Computational Graph

```

1 from torchviz import make_dot
2
3 # Sample input tensor (random data)
4 x = torch.randn(1, input_size) # One sample with 1 feature
5 y_pred = model(x) # Forward pass
6
7 # Create the computational graph
8 dot = make_dot(y_pred, params=dict(model.named_parameters()))
9
10 # Render the graph
11 dot.render("ffnn_graph", format="png", cleanup=True)
12 dot

```

Each box in the computational graph represents a tensor operation within the neural network. The nodes labeled **Addmm** correspond to the linear transformations performed by the `nn.Linear` layers, which compute matrix multiplications followed by bias addition. The **Relu** nodes apply the ReLU activation function, introducing non-linearity into the network.

The parameters of the network, such as weights and biases, are indicated separately and contribute to the forward computation. This visualization helps trace how data propagates through the layers and identifies where gradients will be computed during backpropagation.

In the computational graph, **Accumulated Grad** represents the storage of gradients during backpropagation. When computing the gradient of the loss with respect to model parameters, PyTorch accumulates these gradients in the `.grad` attribute of tensors, allowing optimization steps to adjust weights accordingly.

AddmmBackward corresponds to the backward operation of the **Addmm** function, which performs matrix multiplication followed by bias addition in the forward pass. During backpropagation, **Ad-dmmBackward** computes the gradients of the output with respect to both the input features and the weight matrices of the fully connected layers. These gradients are then accumulated and used for parameter updates during training.

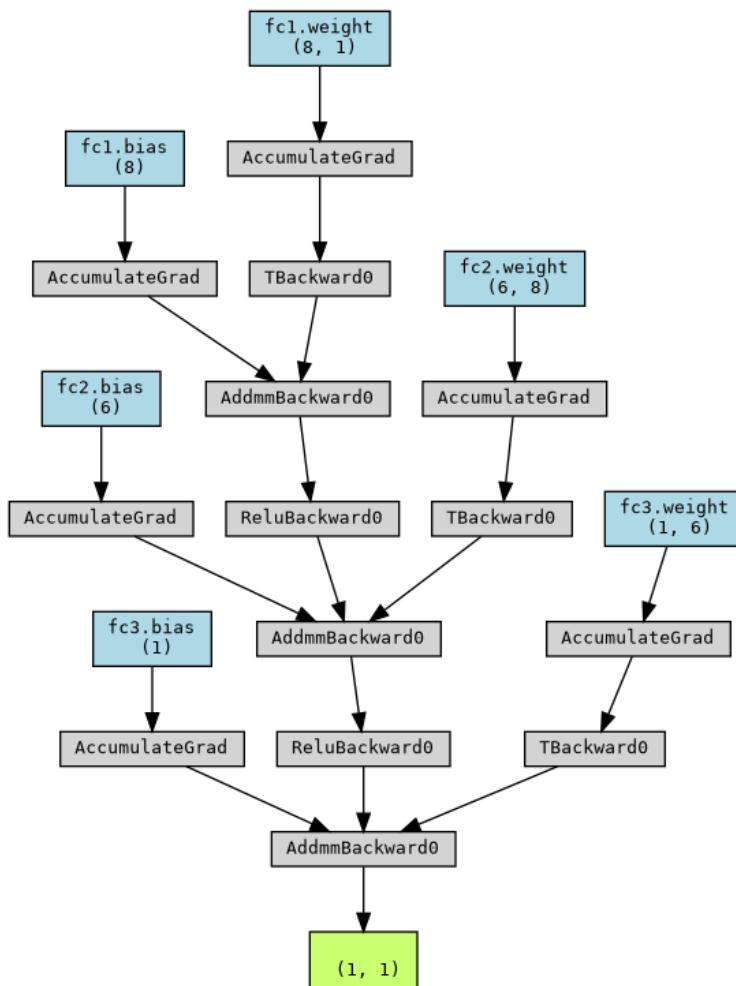


Figure 5.3: Computational graph of the feedforward neural network.

5.2 Graph Neural Networks

Graph Neural Networks (GNNs) are designed to work with graph-structured data. Unlike FFNNs, GNNs can capture relationships between different entities in a graph, making them useful in applications such as social network analysis, molecular property prediction, and recommendation systems.

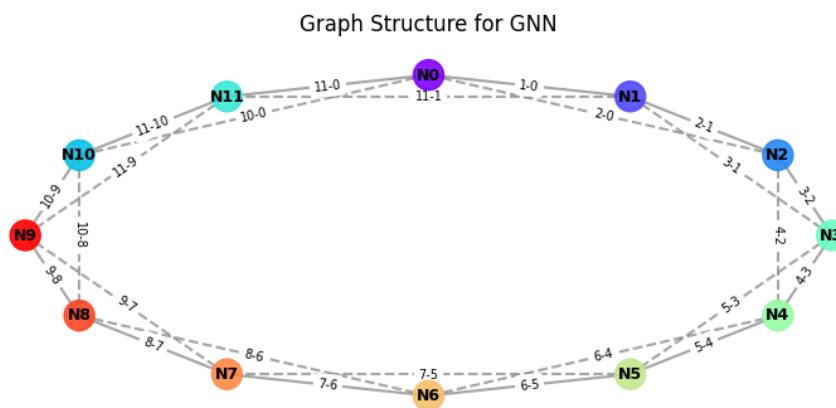


Figure 5.4: Graph visualization for the GNN model with periodic connectivity and elliptical node positions. The nodes are positioned on an ellipse, and the edges are displayed with two types: straight edges (direct neighbors) and periodic edges (non-direct neighbors).

A simple implementation using PyTorch Geometric is shown below. I could get this easily running on linux, on my windows wls, on colab and other frameworks, but on Windows it died regularly without error message. Seems to be a memory management problem.

Recommendation

Training with pytorch or pytorch lightning packages or any other current AI/ML software is characterized by frequent software updates. You will need to move along with the community in a timescale of month, packages get deprecated soon.

Graph Neural Network

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 from torch_geometric.nn import GCNConv
5 from torch_geometric.data import Data
6
7 # Define a GNN with 2 hidden layers
8 class GNNModel(nn.Module):
9     def __init__(self, num_features, hidden_channels, num_feats_y):
10         super().__init__()
11         self.conv1, self.conv2 = GCNConv(num_features, hidden_channels[0]),
12                                     GCNConv(hidden_channels[0], hidden_channels[1])
  
```

```

12         self.fc1, self.fc2 = nn.Linear(hidden_channels[1], hidden_channels[0]), nn
13             .Linear(hidden_channels[0], num_feats_y)
14
15     def forward(self, x, edge_index):
16         x = F.leaky_relu(self.conv1(x, edge_index))
17         x = F.leaky_relu(self.conv2(x, edge_index))
18         x = F.leaky_relu(self.fc1(x))
19         return self.fc2(x)
20
21 # Graph Configuration
22 nx, xa = 25, 10
23 x_grid = torch.linspace(0, xa, nx)
24 p1, p2 = torch.sin(2 * torch.pi * x_grid / xa), torch.cos(2 * torch.pi * x_grid /
25   xa)
26
27 # Create adjacency matrix & edge index
28 diff = torch.sqrt((p1.repeat(nx, 1).T - p1) ** 2 + (p2.repeat(nx, 1).T - p2) ** 2)
29 edge_index = (diff < 0.5).float().nonzero(as_tuple=False).t().contiguous()
30
31 # Create node features & labels
32 data = Data(x=torch.cat((p1.unsqueeze(1), p2.unsqueeze(1)), dim=1), y=torch.
33   randint(0, 2, (nx, 1)).float(), edge_index=edge_index)
34
35 # Initialize & print model
36 model = GNNModel(num_features=2, hidden_channels=[8, 16], num_feats_y=1)
37 print(model)

```

Here, the edge index is for each node given by its index (first row) it prescribes the connected node by index in the second row.

edge_index

```

1 tensor([[ 0,  0,  0,  0,  1,  1,  1,  1,  2,  2,  2,  2,  2,  3,  3,  3,  3,
2           4,  4,  5,  5,  5,  5,  6,  6,  6,  6,  7,  7,  7,  7,  7,  8,  8,  8,
3           9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11],
4           [ 1,  2, 10, 11,  0,  2,  3, 11,  0,  1,  3,  4,  1,  2,  4,  5,  2,  3,
5           5,  6,  3,  4,  6,  7,  4,  5,  7,  8,  5,  6,  8,  9,  6,  7,  9, 10,
6           7,  8, 10, 11,  0,  8,  9, 11,  0,  1,  9, 10]])

```

We finally show how we can learn the **advection** of functions by a the above graph neural network.

Training code

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import torch
4 import torch.nn as nn
5 import torch.nn.functional as F
6 import torch_geometric.data as geom_data

```

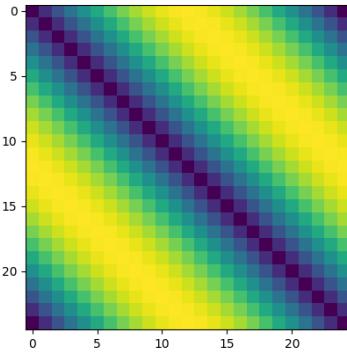


Figure 5.5: The difference matrix showing the distances between nodes in the graph. This matrix is used to determine the adjacency matrix, where the distance between nodes is calculated based on their positions in the space.

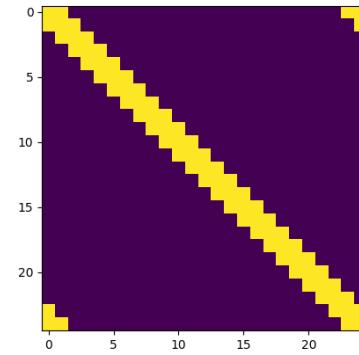


Figure 5.6: A binary adjacency matrix where edges are drawn between nodes whose distance is less than 0.5. This matrix is used to determine which nodes are directly connected in the graph.

```

7 import torch_geometric.nn as geom_nn
8
9 # Set random seed
10 torch.manual_seed(0)
11
12 # Define parameters
13 xa, nx, nt, v = 10, 25, 15, 0.6
14
15 # Create grid and function data
16 x_grid = np.linspace(0, xa, nx + 1)[:-1]
17 z = np.zeros([nt, nx])
18 for j in range(nt):
19     z[j, :] = np.sin((2 * np.pi / xa) * x_grid - v * j)
20
21 # Create adjacency matrix
22 p1 = np.sin(2 * np.pi * x_grid / xa)
23 p2 = np.cos(2 * np.pi * x_grid / xa)
24 p1m, p2m = np.tile(p1, (nx, 1)).T, np.tile(p2, (nx, 1)).T
25 diff = np.sqrt((p1m - p1m.T) ** 2 + (p2m - p2m.T) ** 2)
26 adjm = (diff < 0.5).astype(int)
27 edge_index = torch.tensor(np.nonzero(adjm), dtype=torch.long)
28
29 # Split data into training and testing
30 X_train, Y_train = z[:-1], z[1:]
31 X_test, Y_test = z[:-1], z[1:]
32
33 # Create feature tensors and data loader
34 features_tmp2 = torch.tensor(np.arange(1, nx + 1) / nx, dtype=torch.float).

```

```

        unsqueeze(1)
35 train_list, test_list = [], []
36 for k in range(X_train.shape[0]):
37     features_k_tmp1 = torch.tensor(X_train[k, :], dtype=torch.float).unsqueeze(1)
38     features_k = torch.cat((features_k_tmp1, features_tmp2), dim=1)
39     labels_k = torch.tensor(Y_train[k, :], dtype=torch.float).unsqueeze(1)
40     data = geom_data.Data(x=features_k, y=labels_k, edge_index=edge_index)
41     train_list.append(data)
42
43 for k in range(X_test.shape[0]):
44     features_k_tmp1 = torch.tensor(X_test[k, :], dtype=torch.float).unsqueeze(1)
45     features_k = torch.cat((features_k_tmp1, features_tmp2), dim=1)
46     labels_k = torch.tensor(Y_test[k, :], dtype=torch.float).unsqueeze(1)
47     data = geom_data.Data(x=features_k, y=labels_k, edge_index=edge_index)
48     test_list.append(data)
49
50 # Create DataLoaders for training and testing
51 train_loader = geom_data.DataLoader(train_list, batch_size=1, shuffle=True)
52 test_loader = geom_data.DataLoader(test_list, batch_size=1, shuffle=False)
53
54 # Define the GNN model
55 class GNNModel(nn.Module):
56     def __init__(self, num_features, hidden_channels, num_feats_y):
57         super(GNNModel, self).__init__()
58         self.conv1 = geom_nn.GCNConv(num_features, hidden_channels[0])
59         self.conv2 = geom_nn.GCNConv(hidden_channels[0], hidden_channels[1])
60         self.conv3 = geom_nn.GCNConv(hidden_channels[1], hidden_channels[2])
61         self.conv4 = geom_nn.GCNConv(hidden_channels[2], hidden_channels[3])
62         self.fc1 = nn.Linear(hidden_channels[3], hidden_channels[2])
63         self.fc2 = nn.Linear(hidden_channels[2], hidden_channels[0])
64         self.fc3 = nn.Linear(hidden_channels[0], num_feats_y)
65
66     def forward(self, x, edge_index):
67         x = F.leaky_relu(self.conv1(x, edge_index))
68         x = F.leaky_relu(self.conv2(x, edge_index))
69         x = F.leaky_relu(self.conv3(x, edge_index))
70         x = F.leaky_relu(self.conv4(x, edge_index))
71         x = F.leaky_relu(self.fc1(x))
72         x = F.leaky_relu(self.fc2(x))
73         return self.fc3(x)
74
75 # Initialize model, optimizer, and criterion
76 model = GNNModel(num_features=2, hidden_channels=[4 * nt, 4 * nt, 4 * nt, 4 * nt],
77                  num_feats_y=1)
78 optimizer = torch.optim.AdamW(model.parameters(), lr=0.0005, weight_decay=0)
79 criterion = nn.MSELoss()
80
81 # Training loop
82 epochs = 1500

```

```

82 train_mse, test_mse = [], []
83 for epoch in range(epochs):
84     model.train()
85     total_loss = 0.0
86     train_mse_tmp = []
87     for batch in train_loader:
88         optimizer.zero_grad()
89         output = model(batch.x, batch.edge_index)
90         loss = criterion(output, batch.y)
91         train_mse_tmp.append(loss.item())
92         loss.backward()
93         optimizer.step()
94     train_mse.append(np.mean(train_mse_tmp))
95
96     model.eval()
97     test_mse_tmp = []
98     for batch in test_loader:
99         y_pred = model(batch.x, batch.edge_index)
100        test_loss = criterion(y_pred, batch.y)
101        test_mse_tmp.append(test_loss.item())
102    test_mse.append(np.mean(test_mse_tmp))
103
104    if epoch % 100 == 0:
105        print(f'Epoch {epoch + 1}, Train Loss: {train_mse[epoch]}, Test Loss: {test_mse[epoch]}')
106
107 # Plot training and test MSE
108 plt.plot(np.arange(epochs), train_mse, '*', label='Train Loss')
109 plt.plot(np.arange(epochs), test_mse, '*', label='Test Loss')
110 plt.legend()
111 plt.title("Training and Test Loss")
112 plt.savefig("gnn_loss_curve.png")
113 plt.show()

```

Which comes with the loss curve

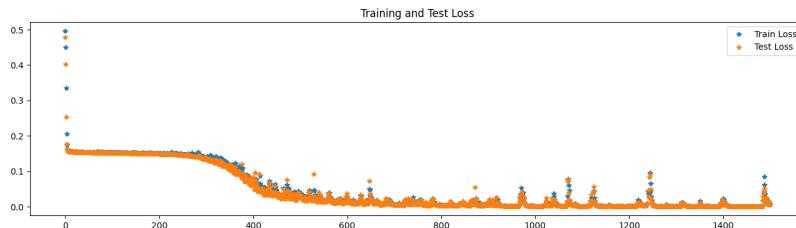


Figure 5.7: Training and Test Loss curves during the training process. The plot shows the Mean Squared Error (MSE) for both training and test sets across epochs.

Testing the translation we display two randomly chosen cases.

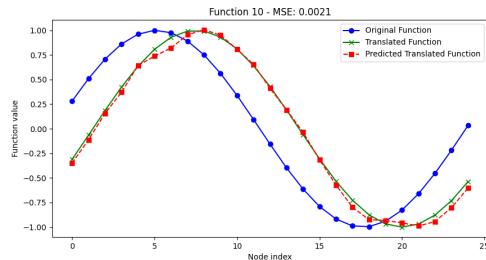


Figure 5.8: Comparison for Test Case 1: Original, Translated, and Predicted Translated Functions. The plot shows the original function, the translated function, and the model's prediction with MSE value.

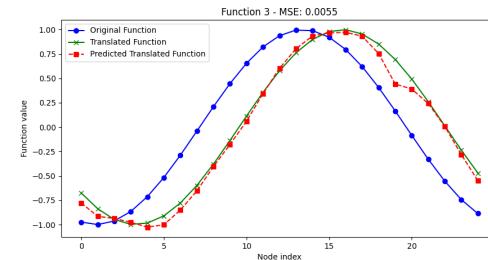


Figure 5.9: Comparison for Test Case 2: Original, Translated, and Predicted Translated Functions. This plot compares the same as Test Case 1 but for another random test case.

5.3 Applying Convolutional Neural Networks for Function Classification

Convolutional Neural Networks (CNNs) are powerful architectures typically used for image processing but can also be applied to one-dimensional data such as time series or function classification. In this section, we demonstrate how to construct a simple CNN to classify different mathematical functions (e.g., sine, cosine, Gaussian, and polynomial functions).

Generating Synthetic Data. To train a CNN, we first need a dataset. We generate synthetic data using functions such as sine or cosine, polynomials and Gaussians with varying parameters such as frequency, phase shifts, and noise levels. The dataset consists of labeled samples representing different mathematical function types.

CNN Data generation

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 def generate_function_data(num_samples=5000, num_points=50, err=0.02):
8     X = []
9     y = []
10    functions = ['sine-cosine', 'gaussian', 'polynomial']
11
12    for _ in range(num_samples):
13        x = np.linspace(-1, 1, num_points)
14        func_type = np.random.choice(functions)
15
16        # Initialize a default y_values to prevent UnboundLocalError
17        y_values = np.zeros(num_points)
18        label = -1

```

```

19
20     if func_type == 'sine-cosine':
21         freq = np.random.uniform(1, 5)
22         phase = np.random.uniform(0, 2 * np.pi)
23         amp = np.random.uniform(0.5, 2)
24         y_values = amp * np.sin(freq * np.pi * x + phase) + err * np.random.
25         randn(num_points)
26         label = 0
27
28     elif func_type == 'gaussian':
29         mu = np.random.uniform(-0.5, 0.5)
30         sigma = np.random.uniform(0.2, 0.5)
31         amp = np.random.uniform(0.5, 2)
32         y_values = amp * np.exp(-((x - mu) ** 2) / (2 * sigma ** 2)) + err *
33         np.random.randn(num_points)
34         label = 2
35
36     elif func_type == 'polynomial':
37         a = np.random.uniform(-2, 2)
38         b = np.random.uniform(-2, 2)
39         c = np.random.uniform(-3, 3)
40         d = np.random.uniform(-0.5, 0.5)
41         y_values = a * x**3 + b * x**2 + c * x + d + err * np.random.randn(
42         num_points)
43         label = 3
44
45     X.append(y_values)
46     y.append(label)
47
48     return torch.tensor(X, dtype=torch.float32), torch.tensor(y, dtype=torch.long)
49
50 # Generate a large training and test dataset with adjustable noise
51 X_train, y_train = generate_function_data(num_samples=10000, err=0.05) # Low
52               noise in training
53 X_test, y_test = generate_function_data(num_samples=2000, err=0.2) # Higher noise
54               in test set
55
56 print(f"Train Data Shape: {X_train.shape}, Train Labels Shape: {y_train.shape}")
57 print(f"Test Data Shape: {X_test.shape}, Test Labels Shape: {y_test.shape}")
58
59 plt.figure(figsize=(12, 3))
60 for i, idx in enumerate(torch.randperm(len(X_train))[:6]):
61     plt.subplot(1, 6, i + 1)
62     plt.plot(X_train[idx][0].cpu().numpy())
63     plt.title([y_train[idx].item()])
64     plt.xticks([]), plt.yticks([])

```

```

63
64 plt.tight_layout()
65 plt.savefig("cnn_data_samples.png", dpi=300)
66 plt.show()

```

Each function is sampled over a fixed range, and the noise level can be controlled via a parameter. The generated dataset is split into training and test sets.

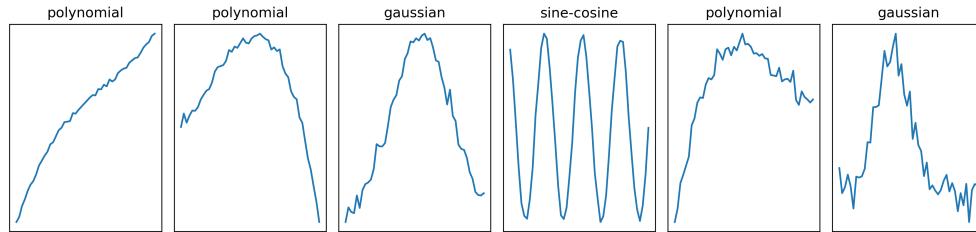


Figure 5.10: Example of generated function data samples

Defining the Convolutional Neural Network. The next step is defining the CNN. Our model consists of two convolutional layers, followed by a fully connected network that maps extracted features to class labels.

CNN Definition

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4
5 class FunctionClassifierCNN(nn.Module):
6     def __init__(self):
7         super(FunctionClassifierCNN, self).__init__()
8         self.conv1 = nn.Conv1d(in_channels=1, out_channels=16, kernel_size=5,
9             stride=1, padding=2)
10        self.conv2 = nn.Conv1d(in_channels=16, out_channels=32, kernel_size=5,
11            stride=1, padding=2)
12        self.fc1 = nn.Linear(32 * 50, 128)
13        self.fc2 = nn.Linear(128, 4) # 4 classes
14
15    def forward(self, x):
16        x = torch.relu(self.conv1(x))
17        x = torch.relu(self.conv2(x))
18        x = x.view(x.shape[0], -1) # Flatten
19        x = torch.relu(self.fc1(x))
20        x = self.fc2(x)
21
22    # Initialize model
23    model = FunctionClassifierCNN()
24    print(model)

```

The convolutional layers apply feature extraction by detecting local patterns in the input functions. The final classification is performed by a fully connected layer.

Training the CNN. The training process involves feeding the generated dataset into the CNN, computing loss using cross-entropy, and updating weights via backpropagation.

CNN Training

```

1 # Training setup
2 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3 model.to(device)
4
5 criterion = nn.CrossEntropyLoss()
6 optimizer = optim.Adam(model.parameters(), lr=0.001)
7
8 num_epochs = 20
9 batch_size = 32
10
11 # Convert dataset into DataLoader
12 train_loader = torch.utils.data.DataLoader(list(zip(X_train, y_train)), batch_size
13     =batch_size, shuffle=True)
14
15 loss_history = [] # Store loss values
16
17 for epoch in range(num_epochs):
18     total_loss = 0
19     for batch_X, batch_y in train_loader:
20         batch_X, batch_y = batch_X.to(device), batch_y.to(device)
21         optimizer.zero_grad()
22         loss = criterion(model(batch_X), batch_y)
23         loss.backward()
24         optimizer.step()
25         total_loss += loss.item()
26
27     loss_history.append(total_loss / len(train_loader)) # Save epoch loss
28     print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss_history[-1]:.4f}")
29
30 # Plot training loss
31 fig=plt.figure(figsize=(10,5))
32 plt.plot(loss_history)
33 plt.xlabel("Epoch")
34 plt.ylabel("Loss")
35 plt.title("Training Loss")
36 plt.savefig("cnn_training_loss.png", dpi=300)
37 plt.show()

```

During training, we monitor the loss function to ensure the model is learning effectively.

Evaluating the Model. Once trained, the CNN is evaluated on the test dataset. Accuracy is computed to assess performance.

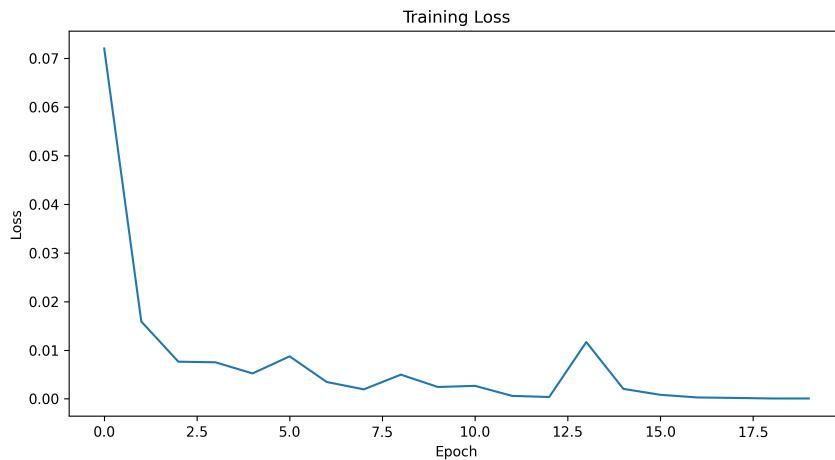


Figure 5.11: Training loss over epochs

CNN Evaluation

```

1 # Evaluation
2 model.eval()
3 test_loader = torch.utils.data.DataLoader(list(zip(X_test, y_test)), batch_size=
    batch_size, shuffle=False)
4
5 correct = 0
6 total = 0
7
8 with torch.no_grad():
9     for batch_X, batch_y in test_loader:
10         batch_X, batch_y = batch_X.to(device), batch_y.to(device)
11
12         outputs = model(batch_X)
13         _, predicted = torch.max(outputs, 1)
14
15         total += batch_y.size(0)
16         correct += (predicted == batch_y).sum().item()
17
18 accuracy = 100 * correct / total
19 print(f"Test Accuracy: {accuracy:.2f}%")

```

A high accuracy indicates the model successfully differentiates between different function types.

Visualizing Predictions. Finally, we visualize how well the model classifies unseen functions by plotting predicted and actual labels.

CNN Visualization

```

1 import random
2 import matplotlib.pyplot as plt

```

```

3
4 # Generate a few test samples
5 num_examples = 12 # Show 12 examples
6 X_new, y_new = generate_function_data(num_samples=num_examples)
7 X_new = X_new.to(device)
8
9 # Get model predictions
10 model.eval()
11 with torch.no_grad():
12     predictions = model(X_new)
13     _, predicted_labels = torch.max(predictions, 1)
14
15 # Function names for visualization
16 func_names = ['Sine', 'Cosine', 'Gaussian', 'Polynomial']
17
18 # Plot the results
19 rows = num_examples // 4 # Show 4 per row
20 plt.figure(figsize=(12, 3 * rows))
21
22 for i in range(num_examples):
23     correct = predicted_labels[i] == y_new[i] # Check if prediction is correct
24     color = 'blue' if correct else 'red' # Blue for correct, red for incorrect
25
26     plt.subplot(rows, 4, i + 1)
27     plt.plot(np.linspace(-1, 1, 50), X_new[i].cpu().numpy().squeeze(), color=color,
28             label=f"Pred: {func_names[predicted_labels[i]]}")
29     plt.legend()
30     plt.title(f"True: {func_names[y_new[i]]}", color=color) # Color title for
31     extra clarity
32     plt.xticks([])
33     plt.yticks([])
34
35 plt.tight_layout()
36 plt.savefig("cnn_test_predictions.png", dpi=300)
37 plt.show()

```

By analyzing the correctly and incorrectly classified samples, we can gain insights into model performance and potential improvements.

We have shown an application of CNNs for function classification, covering data generation, model design, training, evaluation, and visualization. This approach can be extended to classify other types of structured signals.

5.4 LSTM-Based Anomaly Detection in Sensor Data

Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory (LSTM) networks, are powerful for handling sequential data. Unlike traditional feedforward neural networks, LSTMs are designed to capture temporal dependencies by maintaining an internal memory that allows them to

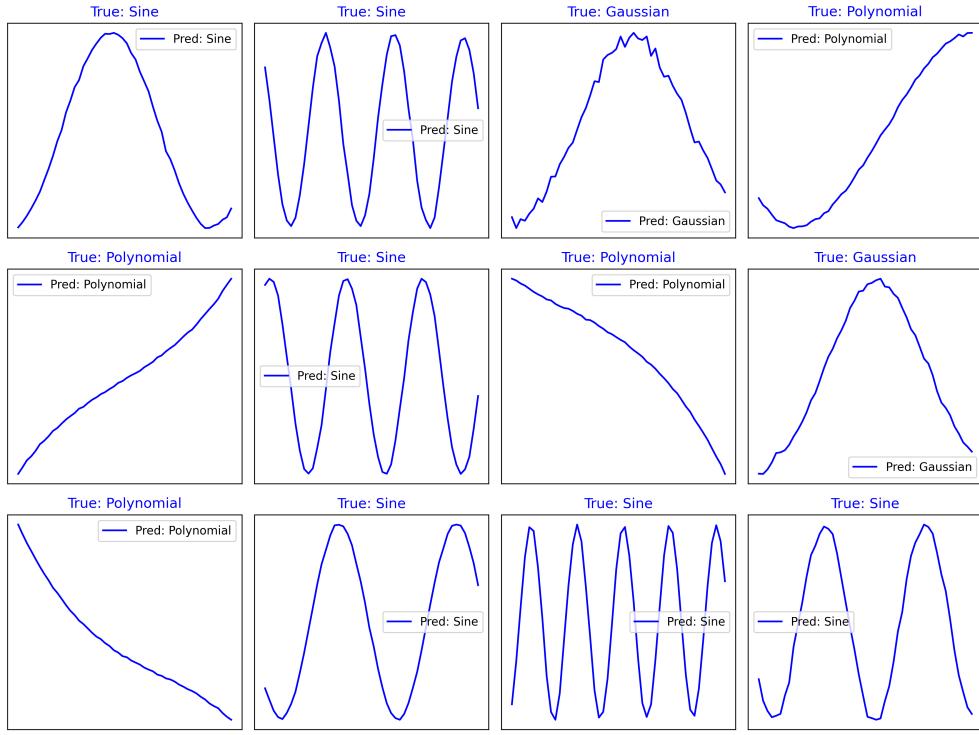


Figure 5.12: Correct (blue) and incorrect (red) predictions

remember relevant past information over long sequences. This makes them particularly suitable for anomaly detection in time series data, where deviations from learned patterns indicate potential anomalies.

An LSTM consists of a series of memory cells, each containing:

- An **input gate** that determines how much new information is added to the cell state.
- A **forget gate** that decides what past information should be discarded.
- An **output gate** that controls how much information from the memory cell is used as output.

By adjusting these gates, the LSTM can selectively retain or forget information, making it highly effective at modeling sequences with long-term dependencies.

Mathematical Formulation of LSTM. To be more precise, an LSTM unit consists of a cell state c_t and three gates: the input gate i_t , forget gate f_t , and output gate o_t . The key equations governing an LSTM cell at time step t are:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad (5.1)$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad (5.2)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad (5.3)$$

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \quad (5.4)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad (5.5)$$

$$h_t = o_t \odot \tanh(c_t) \quad (5.6)$$

$$y_t = W_y h_t + b_y \quad (5.7)$$

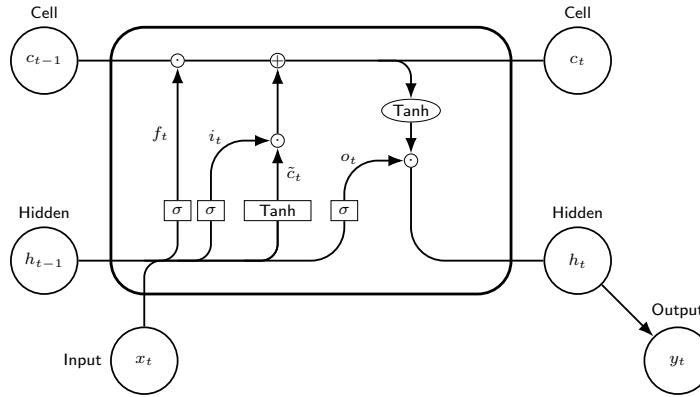


Figure 5.13: A sketch of the functionality of an LSTM cell, as described by the equations (5.1)-(5.7).

following [Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting](#).

Here, $x_t \in \mathbb{R}^m$ represents the input at time step t , while $h_t \in \mathbb{R}^n$ is the hidden state, which encodes information from previous time steps. The memory cell $c_t \in \mathbb{R}^n$ maintains long-term dependencies, with its update governed by three gates: the forget gate f_t , which decides how much past information to retain, the input gate i_t , which determines how much new information to store, and the output gate o_t , which controls what is passed to the hidden state. The candidate cell state \tilde{c}_t contributes to updating c_t , using weight matrices W_c and U_c , and bias b_c .

The weight matrices $W_f, W_i, W_o, W_c \in \mathbb{R}^{n \times m}$ process input connections, while $U_f, U_i, U_o, U_c \in \mathbb{R}^{n \times n}$ manage recurrent hidden state updates. Bias terms $b_f, b_i, b_o, b_c \in \mathbb{R}^n$ adjust the activation functions. The element-wise sigmoid function σ regulates gate activations, and the hyperbolic tangent function \tanh is used for both candidate state computation and final output transformation. Element-wise multiplication is denoted by \odot .

Output Computation. The final output y_t is computed as a linear transformation $W_y h_t + b_y$, where $W_y \in \mathbb{R}^{p \times n}$ maps the hidden state to an output space of dimension p , and $b_y \in \mathbb{R}^p$ is the corresponding bias. The interpretation of y_t depends on the application: in sequence classification, y_t is typically taken from the final time step; in sequence-to-sequence tasks, it is used at every time step; and in autoencoders, it reconstructs the original sequence.

By updating these gates at each time step, LSTMs effectively address the vanishing gradient problem, enabling the learning of long-term dependencies in sequential data.

In this section, we implement an LSTM-based autoencoder to detect anomalies in simulated sensor data. The autoencoder learns to reconstruct normal sequences, and anomalies are detected based on high reconstruction error.

Generating Sensor Data. To train the model, we generate synthetic sensor data using sine waves with random phase shifts. Anomalies are introduced as sudden deviations.

Generating Sensor Data

```
1 import numpy as np
2
```

```

3 # Generate normal sine wave data with random phase shift
4 def generate_sensor_data(num_samples=1000, seq_length=50, anomaly_ratio=0.1):
5     X = []
6     labels = []
7
8     for _ in range(num_samples):
9         phase_shift = np.random.uniform(0, 2 * np.pi) # Random shift
10        time_series = np.sin(np.linspace(0, 2 * np.pi, seq_length) + phase_shift)
11        + 0.1 * np.random.randn(seq_length)
12        label = 0 # Normal
13
14        # Inject anomalies
15        if np.random.rand() < anomaly_ratio:
16            time_series += np.random.uniform(-2, 2, size=seq_length) # Add large
17            spikes
18            label = 1 # Anomaly
19
20    X.append(time_series)
21    labels.append(label)
22
23 return np.array(X), np.array(labels)

```

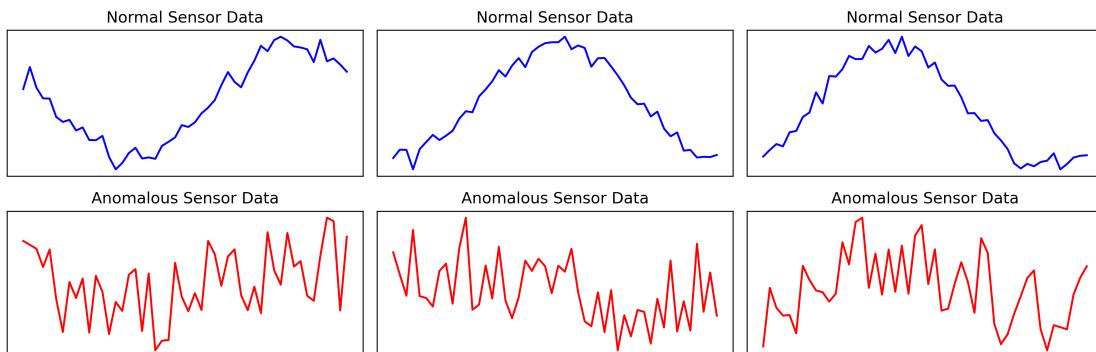


Figure 5.14: Examples of normal and anomalous sensor data. Normal sequences are in blue, while anomalies are shown in red.

Defining the LSTM Autoencoder. We define an LSTM autoencoder consisting of an encoder that compresses the input sequence into a lower-dimensional hidden state and a decoder that reconstructs the original sequence.

Defining the LSTM Autoencoder

```

1 import torch
2
3 # Define device for computation (CPU/GPU)
4 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
5
6 class LSTMEncoder(nn.Module):

```

```

7     def __init__(self, input_dim=1, hidden_dim=32, num_layers=2, seq_length=50):
8         super(LSTMAutoencoder, self).__init__()
9         self.seq_length = seq_length
10        self.hidden_dim = hidden_dim
11        self.num_layers = num_layers
12
13        # LSTM layers
14        self.encoder = nn.LSTM(input_dim,hidden_dim, num_layers,batch_first=True)
15        self.decoder = nn.LSTM(input_dim,hidden_dim, num_layers,batch_first=True)
16
17        # Final layer to reconstruct input
18        self.output_layer = nn.Linear(hidden_dim, input_dim)
19
20    def forward(self, x):
21        batch_size = x.size(0)
22
23        # Encode input
24        _, (hidden, cell) = self.encoder(x) # Correct hidden state extraction
25
26        # Initialize decoder input as zeros
27        decoder_input = torch.zeros(batch_size, self.seq_length, 1).to(x.device)
28
29        # Decode using the last hidden state from the encoder
30        decoder_output, _ = self.decoder(decoder_input, (hidden, cell))
31
32        # Apply final layer to match original input size
33        x_reconstructed = self.output_layer(decoder_output)
34
35        return x_reconstructed # Shape: [batch_size, seq_length, input_dim]
36
37 # Initialize model with correct sequence length
38 model = LSTMAutoencoder(seq_length=50).to(device)

```

Training the Model. The model is trained using Mean Squared Error (MSE) loss, optimizing the ability to reconstruct normal sequences.

Training the Model

```

1 # Training setup
2 criterion = nn.MSELoss()
3 optimizer = optim.Adam(model.parameters(), lr=0.001)
4 num_epochs = 50
5 batch_size = 32
6
7 train_loader = torch.utils.data.DataLoader(X_train, batch_size=batch_size, shuffle
     =True)
8
9 # Track loss history
10 loss_history = []

```

```

11
12 # Training loop
13 for epoch in range(num_epochs):
14     total_loss = 0
15     for batch in train_loader:
16         batch = batch.to(device)
17         optimizer.zero_grad()
18         outputs = model(batch)
19         loss = criterion(outputs, batch) # Compare input and output
20         loss.backward()
21         optimizer.step()
22         total_loss += loss.item()
23
24     epoch_loss = total_loss / len(train_loader)
25     loss_history.append(epoch_loss) # Store epoch loss
26     print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}")
27
28 plt.plot(loss_history, label="Loss")
29 plt.xlabel("Epochs"), plt.ylabel("Loss"), plt.title("LSTM Training Loss")
30 plt.legend(), plt.grid(True)
31 plt.savefig("lstm_training_loss.png", dpi=300)
32 plt.show()

```

Detecting Anomalies. Anomalies are detected by computing the reconstruction error on test sequences. If the error exceeds a predefined threshold (e.g., 95th percentile), the sequence is classified as an anomaly.

Anomaly Detection

```

1 import numpy as np
2
3 # Compute reconstruction error on test data
4 model.eval()
5 X_test = X_test.to(device)
6 with torch.no_grad():
7     X_reconstructed = model(X_test)
8
9 reconstruction_errors = torch.mean((X_test - X_reconstructed) ** 2, dim=(1, 2)) .
    cpu().numpy()
10
11 # Set anomaly threshold (e.g., 95th percentile)
12 threshold = np.percentile(reconstruction_errors, 95)
13 y_pred = (reconstruction_errors > threshold).astype(int) # 1 if anomaly, else 0
14
15 # Compute detection accuracy
16 accuracy = np.mean(y_pred == y_test) * 100
17 print(f"Anomaly Detection Accuracy: {accuracy:.2f}%")

```

In our case we got

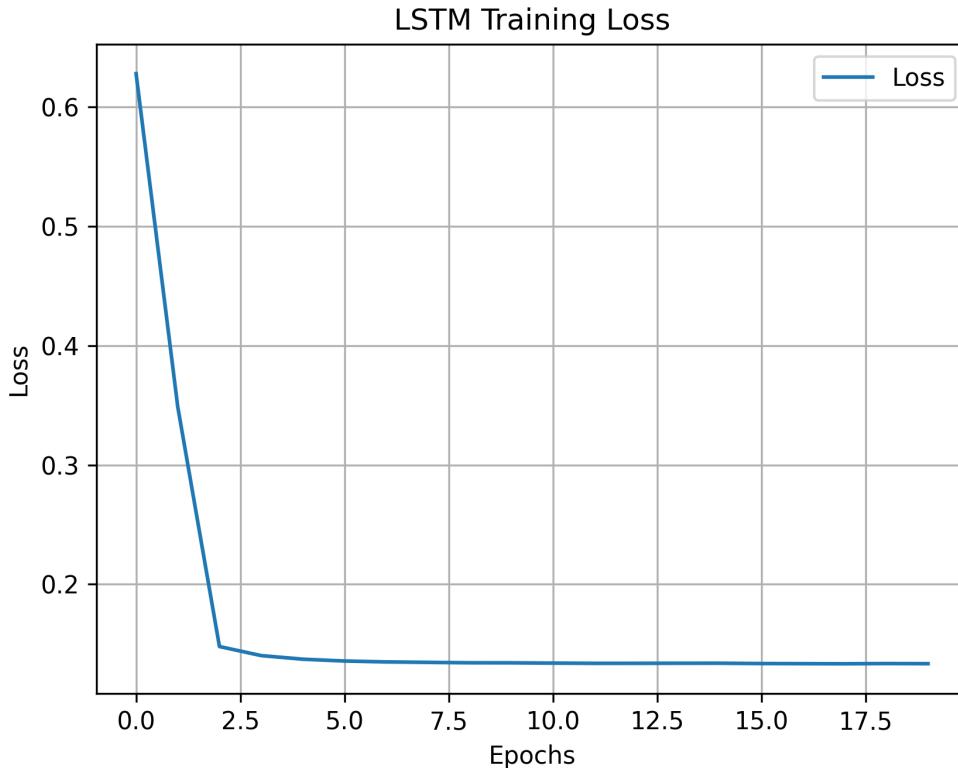


Figure 5.15: Training loss curve showing the decrease in reconstruction error over epochs.

Anomaly Detection Accuracy: 96.25%

Visualizing Detected Anomalies. We randomly sample 12 sequences from the test set, classify them, and color them accordingly—blue for normal and red for anomalies.

Visualizing Anomalies

```

1 import matplotlib.pyplot as plt
2
3 # Select 12 random test samples
4 num_samples = 12
5 indices = np.random.choice(len(X_test), num_samples, replace=False)
6
7 # Compute reconstruction errors
8 model.eval()
9 with torch.no_grad():
10     X_reconstructed = model(X_test.to(device))
11
12 reconstruction_errors = torch.mean((X_test - X_reconstructed) ** 2, dim=(1, 2)).
13     cpu().numpy()
14 # Detect anomalies based on threshold

```

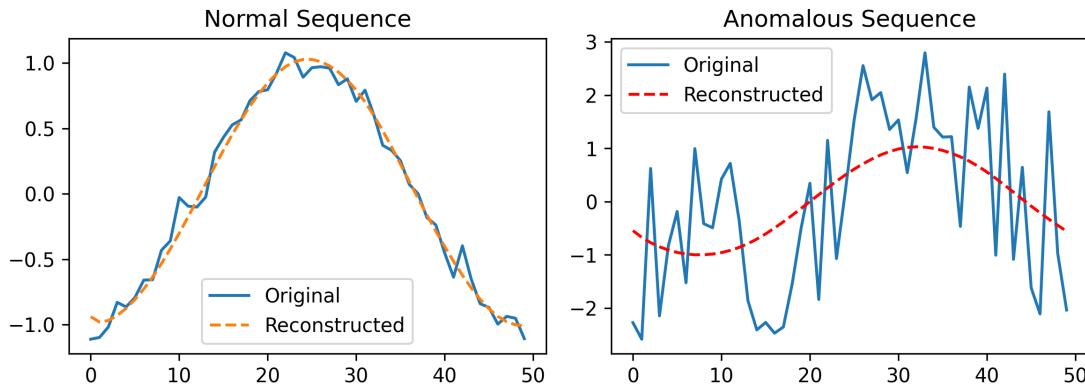


Figure 5.16: Example of normal (left) and anomalous (right) sequences. Dashed lines represent reconstructed sequences.

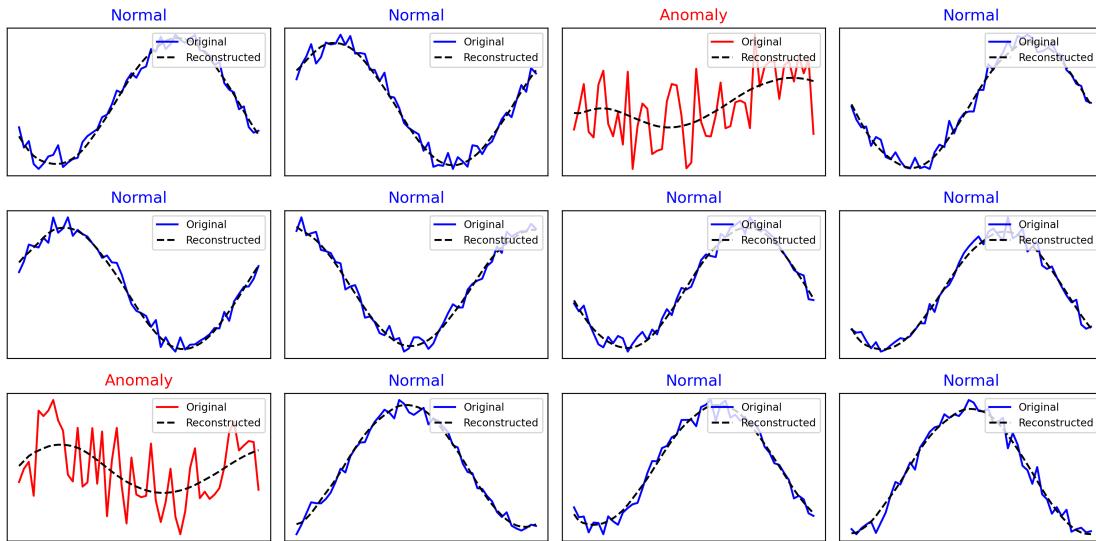


Figure 5.17: Illustration of the detection and correction of the sensor anomaly.

```

15 threshold = np.percentile(reconstruction_errors, 90)
16 y_pred = (reconstruction_errors > threshold).astype(int) # 1 = Anomaly, 0 =
  Normal
17
18 # Plot the selected samples
19 plt.figure(figsize=(12, 6))
20 for i, idx in enumerate(indices):
21     color = 'red' if y_pred[idx] == 1 else 'blue'
22
23     plt.subplot(3, 4, i + 1)
24     plt.plot(X_test[idx].cpu().numpy(), color=color, label="Original")
25     plt.plot(X_reconstructed[idx].cpu().numpy(), linestyle="dashed", color="black",
              , label="Reconstructed")

```

```
26     plt.title(f"'Anomaly' if y_pred[idx] == 1 else 'Normal'", color=color)
27     plt.xticks([]), plt.yticks([])
28     plt.legend(fontsize=8, loc="upper right")
29
30 plt.tight_layout()
31 plt.savefig("lstm_anomaly_detection_samples_selected.png", dpi=300)
32 plt.show()
```

Recommendation

There are quite different architectures of neural networks. The connectivity is a crucial choice for the functionality of the network. But also training datasets and optimisation strategies determine the success or failure of the network functionality and quality.

Recommendation

There is not one good or bad network or architecture, but different approaches are good for different applications.

Chapter 6

Large Language Models

6.1 LLM Network as Sequence-to-Sequence Machines via Transformer Models

At their core, Large Language Models (LLMs) are sequence-to-sequence machines that generate a response sequence given an input sequence of words, converted into tokens. However, the way they process and generate these sequences involves several layers of complexity:

1. Contextual Understanding via Self-Attention

Unlike simple sequence models (e.g., RNNs), Transformers use self-attention to consider all tokens in a sequence simultaneously. This allows them to capture dependencies across long contexts.

2. Probability-Based Token Generation

At each step, the model predicts the next token by computing a probability distribution over the vocabulary. The output sequence is formed by sampling or selecting the most probable token at each step.

3. Task-Specific Adaptations

- **Text Generation** (GPT, LLaMA, Mistral): Autoregressive models predict one token at a time, conditioning each step on previous outputs.
- **Text Understanding** (BERT): Masked language models predict missing tokens given bidirectional context.
- **Instruction-Tuned Models** (ChatGPT, Claude): Fine-tuned on dialogue and instruction-following data, allowing multi-turn interactions.

So, while LLMs fundamentally map an input sequence to an output sequence, their real power comes from how they encode context, manage dependencies, and adapt to various tasks through fine-tuning and prompting.

Large Language Models (LLMs) have revolutionized natural language processing (NLP) through the use of Transformer architectures. Introduced by Vaswani et al. in 2017, Transformers have

surpassed previous recurrent and convolutional models in both efficiency and scalability. Let us give a quick overview of Transformer architectures and their role in modern LLMs.

The Transformer model is based on the self-attention mechanism and is composed of an encoder-decoder structure. However, most LLMs, such as GPT and BERT, utilize only the encoder or decoder portion.

Self-attention processes an input sequence of n tokens, where each token represents a word or subword from a given text. The input is converted into a numerical representation in three steps.

First, the input sentence is tokenized using a tokenizer such as WordPiece or Byte-Pair Encoding. For example, the sentence:

"write code for problem a"

is split into the tokens:

{write, code, for, problem, a}

resulting in $n = 5$ tokens.

Each token is then mapped to a high-dimensional vector using a learned embedding matrix $E \in \mathbb{R}^{V \times d_{\text{model}}}$, where: - V is the vocabulary size. - d_{model} is the embedding dimension.

The input sequence is represented as a matrix:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^{n \times d_{\text{model}}},$$

where each row $x_i \in \mathbb{R}^{1 \times d_{\text{model}}}$ corresponds to the embedding of a token.

Positional Encoding: Since Transformers lack an inherent sequence structure, a **positional encoding matrix** $P \in \mathbb{R}^{n \times d_{\text{model}}}$ is added:

$$X_{\text{final}} = X + P,$$

where: P is a matrix containing position-specific values that help encode the order of tokens and each row $P_i \in \mathbb{R}^{1 \times d_{\text{model}}}$ corresponds to the positional encoding for the i -th token. The positional encoding matrix $P \in \mathbb{R}^{n \times d_{\text{model}}}$ is defined using sinusoidal functions, where each element is computed as

$$P_{(i,2j)} = \sin\left(\frac{i}{10000^{2j/d_{\text{model}}}}\right), \quad P_{(i,2j+1)} = \cos\left(\frac{i}{10000^{2j/d_{\text{model}}}}\right),$$

with i representing the token position and j the dimension index. The resulting matrix X_{final} is then passed to the self-attention mechanism, where each token can attend to all others. This allows the model to differentiate between identical words appearing in different positions within the sequence. The resulting matrix X_{final} is the input to the self-attention mechanism, where each token can attend to all others.

This processed matrix X_{final} serves as the input to the self-attention mechanism, where each token can attend to all other tokens in the sequence. For each token, three matrices project its embedding into three key representations:

- **Query matrix** $Q \in \mathbb{R}^{n \times d_k}$
- **Key matrix** $K \in \mathbb{R}^{n \times d_k}$
- **Value matrix** $M \in \mathbb{R}^{n \times d_v}$

These matrices are obtained from the input embedding matrix $X \in \mathbb{R}^{n \times d_{\text{model}}}$ through learned weight matrices W^Q , W^K , and W^M :

$$Q = XW^Q, \quad K = XW^K, \quad M = XW^M, \quad (6.1)$$

where $W^Q, W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $W^M \in \mathbb{R}^{d_{\text{model}} \times d_v}$ are learnable parameter matrices.

The attention scores are computed using the scaled dot-product attention:

$$\text{Attention}(Q, K, M) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) M. \quad (6.2)$$

The softmax function is applied to each row of a matrix $S \in \mathbb{R}^{n \times n}$, where each element is transformed as:

$$\text{softmax}(S)_{ij} = \frac{\exp(S_{ij})}{\sum_{k=1}^n \exp(S_{ik})}.$$

This ensures that for each row i , the values satisfy:

$$\sum_{j=1}^n \text{softmax}(S)_{ij} = 1,$$

converting the attention scores into a probability distribution across all tokens. Since $Q \in \mathbb{R}^{n \times d_k}$ and $K^T \in \mathbb{R}^{d_k \times n}$, the matrix product QK^T results in an attention score matrix of shape $\mathbb{R}^{n \times n}$. After applying the softmax function row-wise, the resulting matrix is multiplied by $V \in \mathbb{R}^{n \times d_v}$, producing the final output

$$\text{Attention}(Q, K, M) \in \mathbb{R}^{n \times d_v}.$$

We summarize:

- $QK^T \in \mathbb{R}^{n \times n}$ computes similarity scores between all tokens.
- The scaling factor $\sqrt{d_k}$ prevents large values inside the softmax function.
- The softmax function normalizes the similarity scores.

Multi-Head Attention: Instead of using a single attention mechanism, Transformers employ multiple attention heads to capture different aspects of the input. Given an input sequence $X \in \mathbb{R}^{n \times d_{\text{model}}}$,

multiple projections of Q, K, M are computed for each attention head. Each head with index i applies self-attention independently using separate weight matrices:

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad M_i = XW_i^M, \quad (6.3)$$

where

$$W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^M \in \mathbb{R}^{d_{\text{model}} \times d_v}$$

are learnable weight matrices for each head. Self-attention is then computed for each head as:

$$\text{head}_i = \text{Attention}(Q_i, K_i, M_i), \quad (6.4)$$

with $\text{head}_i \in \mathbb{R}^{n \times d_v}$. The outputs of all h heads are then concatenated:

$$\text{MultiHead}(Q, K, M) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O, \quad (6.5)$$

where $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ is a learned projection matrix that maps the concatenated outputs back to the model's hidden dimension d_{model} . The final output has the shape:

$$\text{MultiHead}(Q, K, M) \in \mathbb{R}^{n \times d_{\text{model}}}.$$

Loss Function for Attention: The self-attention mechanism is trained by minimizing a loss function that measures the discrepancy between the model's predictions and the target outputs. Given an input sequence X and corresponding ground truth output Y , the model produces an output representation Z from self-attention:

$$Z = \text{Attention}(Q, K, M) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)M.$$

For multi-head attention, the output is:

$$Z = \text{MultiHead}(Q, K, M) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O.$$

In a sequence-to-sequence task, such as machine translation, the model processes an input sequence and generates an output sequence token by token. The loss function measures how well the predicted probability distribution over the vocabulary matches the ground truth sequence.

Output Projection to Vocabulary: Since the output of self-attention and multi-head attention is a sequence representation matrix $Z \in \mathbb{R}^{n \times d_{\text{model}}}$, it must be mapped to a probability distribution over the vocabulary. This is done using a learned output weight matrix $W_{\text{out}} \in \mathbb{R}^{d_{\text{model}} \times V}$ and a bias term $b_{\text{out}} \in \mathbb{R}^V$, where V is the vocabulary size. The transformation is performed as follows:

$$Z_{\text{logits}} = ZW_{\text{out}} + b_{\text{out}},$$

resulting in $Z_{\text{logits}} \in \mathbb{R}^{n \times V}$, where each row represents a raw score (logit) over all possible vocabulary words for the corresponding input token.

Converting Logits to Probabilities: Since Z_{logits} contains unnormalized scores, we apply the softmax function row-wise to obtain a valid probability distribution:

$$Z_{\text{pred}} = \text{softmax}(Z_{\text{logits}}),$$

where $Z_{\text{pred}} \in \mathbb{R}^{n \times V}$ and each row is a probability distribution over the vocabulary, summing to 1.

Comparison with Ground Truth: Given a sequence of true output tokens $Y = (y_1, y_2, \dots, y_n)$, where each y_t is an integer index in the vocabulary, we define the corresponding one-hot encoded matrix:

$$Z_Y \in \mathbb{R}^{n \times V},$$

where each row $Z_{Y,t}$ is a one-hot vector (in mathematical terms a unity vector e_{i_t} with position i_t for the word in the vocabulary) indicating the correct token at position t . Since Z_Y is one-hot, we extract the predicted probability assigned to the correct token i_t at each position t :

$$P_t = Z_{\text{pred},t,i_t}.$$

Cross-Entropy Loss: The loss function measures the model's ability to assign high probability to the correct next token. It is computed as:

$$\mathcal{L} = - \sum_{t=1}^n \log P_t,$$

where we note that P_t is between 0 and 1, such that larger P_t corresponds to lower \mathcal{L} . This ensures that if the model assigns low probability to the correct token, the loss is high, guiding the optimization process to improve predictions.

Training with Gradient Descent: The loss gradients are computed with respect to all learnable parameters, including the attention weight matrices and the output projection:

$$\frac{\partial \mathcal{L}}{\partial W^Q}, \quad \frac{\partial \mathcal{L}}{\partial W^K}, \quad \frac{\partial \mathcal{L}}{\partial W^V}, \quad \frac{\partial \mathcal{L}}{\partial W^O}, \quad \frac{\partial \mathcal{L}}{\partial W_{\text{out}}}.$$

The parameters are updated using gradient-based optimization methods such as Adam:

$$W^Q \leftarrow W^Q - \eta \frac{\partial \mathcal{L}}{\partial W^Q}, \quad W^K \leftarrow W^K - \eta \frac{\partial \mathcal{L}}{\partial W^K}, \quad W^V \leftarrow W^V - \eta \frac{\partial \mathcal{L}}{\partial W^V}, \quad W_{\text{out}} \leftarrow W_{\text{out}} - \eta \frac{\partial \mathcal{L}}{\partial W_{\text{out}}}.$$

where η is the learning rate. The optimization process is repeated for multiple input-output pairs to gradually improve the model's predictions.

Popular LLMs include:

- **BERT** (Bidirectional Encoder Representations from Transformers) - Uses the encoder stack for contextual embeddings.
- **GPT** (Generative Pretrained Transformer) - Uses the decoder stack for autoregressive text generation.
- **T5** (Text-to-Text Transfer Transformer) - Converts all NLP tasks into a text-to-text format.

6.2 Implementing and Training a Simple Transformer-Based LLM

In the previous section, we explored the fundamental concepts behind self-attention, multi-head attention, and how transformers process sequences. We now implement a simple transformer-based language model (LLM) in PyTorch to demonstrate these principles in practice.

Transformers process input sequences by first embedding tokens into high-dimensional vectors and then refining these representations through multiple layers of self-attention and feedforward transformations. The model is trained to predict the next token in a sequence, adjusting its parameters using gradient descent.

This section provides a practical walkthrough of implementing a transformer-based LLM, covering:

- The core components of a transformer, including positional encoding and self-attention.
- The forward pass of a transformer model applied to text.
- Training the model on a small dataset.
- Using the trained model to generate text.

We begin by defining a simple transformer model using PyTorch, followed by data preprocessing, training, and text generation.

Simple Transformer for Text Processing

```

1 import torch
2 import torch.nn as nn
3 import math
4
5 # Define the Positional Encoding
6 class PositionalEncoding(nn.Module):
7     def __init__(self, d_model, max_len=5000):
8         super(PositionalEncoding, self).__init__()
9         pe = torch.zeros(max_len, d_model)
10        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
11        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(
12            10000.0) / d_model))
13        pe[:, 0::2] = torch.sin(position * div_term)
14        pe[:, 1::2] = torch.cos(position * div_term)
15        pe = pe.unsqueeze(0).transpose(0, 1)
16        self.register_buffer('pe', pe)
17
18    def forward(self, x):
19        return x + self.pe[:x.size(0), :]
20
21 # Define the Transformer Block
22 class TransformerBlock(nn.Module):
23     def __init__(self, d_model, num_heads, d_ff):
24         super(TransformerBlock, self).__init__()
25         self.attention = nn.MultiheadAttention(d_model, num_heads)
26         self.norm1 = nn.LayerNorm(d_model)
27         self.norm2 = nn.LayerNorm(d_model)
28         self.ff = nn.Sequential(
29             nn.Linear(d_model, d_ff),
30             nn.ReLU(),
31             nn.Linear(d_ff, d_model)
32         )

```

```

32
33     def forward(self, x):
34         attn_out, _ = self.attention(x, x, x)
35         x = self.norm1(x + attn_out)
36         x = self.norm2(x + self.ff(x))
37         return x
38
39 # Define the Transformer Model
40 class SimpleTransformer(nn.Module):
41     def __init__(self, d_model, num_heads, num_layers, vocab_size, max_len, d_ff
=2048):
42         super(SimpleTransformer, self).__init__()
43         self.embedding = nn.Embedding(vocab_size, d_model)
44         self.positional_encoding = PositionalEncoding(d_model, max_len)
45         self.layers = nn.ModuleList([TransformerBlock(d_model, num_heads, d_ff)
for _ in range(num_layers)])
46         self.fc_out = nn.Linear(d_model, vocab_size)
47
48     def forward(self, x):
49         x = self.embedding(x)
50         x = self.positional_encoding(x)
51         for layer in self.layers:
52             x = layer(x)
53         return self.fc_out(x)
54
55 # Example usage
56 model = SimpleTransformer(d_model=32, num_heads=2, num_layers=2, vocab_size=56,
max_len=6)
57 print(model)

```

6.2.1 Setting Up, Training, and Using Our Simple LLM

To train and use our simple transformer-based LLM, we will:

1. Define a small vocabulary and dataset
2. Preprocess the data
3. Train the model
4. Generate text using the trained model

Defining a Small Vocabulary and Dataset

We use a simple vocabulary with a set of basic sentences for training.

Define Vocabulary and Dataset

```

1 vocab = {1: "I", 2: "am", 3: "hungry", ..., 55: "was"}
2 sentences = [
3     "I am hungry",
4     "you are tired",
5     "we are happy",
6     "they are sad",

```

```

7     "it is simple",
8     "the weather is nice",
9 ]

```

Preprocessing Data

Tokenizing and padding sentences for training.

Tokenization and Padding

```

1 def tokenize_sentence(sentence, vocab):
2     return [key for word in sentence.split() for key, value in vocab.items() if
3             value == word]
4
5 def pad_sequence(seq, max_len, pad_value=0):
6     return seq + [pad_value] * (max_len - len(seq)) if len(seq) < max_len else seq
7 [:max_len]

```

Training the Model

Now, we train the transformer model using a simple training loop.

Train the Transformer

```

1 import torch.optim as optim
2
3 model = SimpleTransformer(d_model=32, num_heads=2, num_layers=2, vocab_size=56,
4                           max_len=6)
5 criterion = nn.CrossEntropyLoss(ignore_index=0)
6 optimizer = optim.Adam(model.parameters(), lr=0.001)
7
8 def train(model, dataloader, epochs=101):
9     model.train()
10    for epoch in range(epochs):
11        total_loss = 0
12        for x, y in dataloader:
13            optimizer.zero_grad()
14            output = model(x)
15            loss = criterion(output.view(-1, 56), y.view(-1))
16            loss.backward()
17            optimizer.step()
18            total_loss += loss.item()
19        if epoch % 100 == 0:
20            print(f"Epoch {epoch+1}, Loss: {total_loss/len(dataloader):.4f}")
21    train(model, dataloader)

```

Generating Text with the Trained Model

We generate sentences by feeding input sequences into the trained model.

Generate Text

```

1 def generate_text(model, seed_seq, max_length=6):
2     model.eval()
3     with torch.no_grad():
4         seq = seed_seq.clone()
5         for _ in range(max_length - len(seq)):
6             output = model(seq.unsqueeze(0))
7             next_token = torch.argmax(output[:, -1, :], dim=-1)
8             seq = torch.cat([seq, next_token], dim=0)
9     return seq
10
11 # Example generation
12 seed = torch.tensor([1, 2]) # "I am"
13 output_seq = generate_text(model, seed)
14 print("Generated Sequence:", output_seq.tolist())

```

This section provides a fundamental workflow for setting up, training, and using a simple transformer-based LLM.

To train a large-scale language model (LLM), the process extends beyond a simple dataset and model architecture. Large LLMs require vast amounts of text data, often consisting of terabytes of diverse sources such as books, articles, and web content. Instead of training on small, manually defined vocabularies, modern LLMs utilize subword tokenization techniques, such as SentencePiece or Byte-Pair Encoding (BPE), to handle open-ended vocabulary sizes efficiently.

The model itself is composed of billions of parameters, requiring parallelized training across multiple GPUs or TPUs using techniques such as model parallelism and pipeline parallelism. The optimization process involves advanced gradient accumulation, mixed-precision training for efficiency, and adaptive optimizers like AdamW.

Additionally, large-scale training requires extensive pretraining followed by task-specific fine-tuning, ensuring both general language understanding and domain-specific capabilities. Due to the computational scale, training an LLM can take weeks or months on dedicated high-performance clusters.

Training large-scale language models (LLMs) requires immense computational resources, typically measured in GPU hours. For example,

- GPT-3 (175 billion parameters) was trained on approximately 3640 petaflop-days, which translates to roughly 10 million GPU hours on NVIDIA V100 GPUs.
- More recent models, such as GPT-4 and PaLM-2, likely required even higher computational budgets, often exceeding 20–30 million GPU hours.

In contrast, a high-performance supercomputer like HOREKA at KIT, which features NVIDIA A100 GPUs, delivers a peak performance of around 17 petaflops for AI workloads. Assuming an efficient utilization of HOREKA's full AI capacity, training a model like GPT-3 would take several months,

whereas dedicated large-scale clusters, such as those used by OpenAI or Google, can parallelize the workload across thousands of GPUs, reducing training time to a few weeks. This illustrates the sheer scale of computational power needed for modern LLM training compared to even high-end academic supercomputers.

Recommendation

Training a LLM to achieve very high quality is a major task, which needs a lot of resources both in terms of preparation as well as computing power. However, this effort is invested today by a growing number of actors on an international scale. Using and finetuning models is already very easy and will become more feasible, with LLM functionality becoming ubiquitous already now. Focus on modularly leveraging the growing potential of LLM intelligence combining it with your applications and services.

6.3 Install Your Own LLM, Chat with it and Develop Applications

Several open-source frameworks allow users to install and run large language models (LLMs) on local machines or servers without requiring proprietary cloud-based solutions.

- One of the most user-friendly tools is `ollama`, which provides a streamlined interface for running optimized LLMs on consumer hardware with GPU acceleration.
- Other notable frameworks include `LM Studio`, which offers a graphical interface for managing local LLMs, and `Text Generation WebUI`, which provides an interactive web-based interface for experimenting with different models.
- Additionally, `GPTQ-for-LLaMa` and `AutoGPTQ` support quantized models for memory-efficient execution. For more advanced setups, `vLLM` enables high-throughput inference, while `llama.cpp` provides a highly optimized C++ implementation of LLaMA models for running on CPU-based systems. These frameworks allow researchers and developers to experiment with LLMs without requiring access to large-scale cloud infrastructure.
- One of the most widely used open-source frameworks for installing and running large language models (LLMs) is the `Hugging Face Transformers` library. It provides pre-trained models, easy-to-use APIs, and support for fine-tuning on custom datasets. The library includes models such as GPT, BERT, T5, and LLaMA, among many others, and integrates seamlessly with PyTorch, TensorFlow, and JAX. Hugging Face also offers `Optimum` for hardware optimizations, allowing efficient execution on GPUs and specialized accelerators such as TensorRT and Habana Gaudi. Combined with datasets and accelerate, it enables large-scale training and inference on local or distributed systems. While Hugging Face primarily focuses on cloud and research environments, it can also be used locally with models optimized for consumer hardware, making it a versatile choice for both academic and production applications.

Ollama is a framework that allows users to run local LLMs efficiently. To install Ollama and run a model locally, follow these steps:

Downloading and Installing Ollama To install Ollama, download and execute the official installation script:

Install Ollama

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

Verifying the Installation After installation, check if Ollama is installed correctly by running:

Check Ollama Version

```
1 ollama --version
```

This command should return the installed version of Ollama.

Pulling and Running a Pre-Trained Model To download and execute a pre-trained model, such as Mistral, use:

Download and Run Mistral

```
1 ollama pull mistral
2 ollama run mistral
```

The first command downloads the model, while the second runs it locally.

Starting and Stopping the Ollama Server Ollama can run as a background service to manage models efficiently. To start the Ollama server, use:

Start the Ollama Server

```
1 ollama serve
```

This command launches the Ollama server, making it ready to handle model requests.

To stop the running Ollama server, use:

Stop the Ollama Server

```
1 ollama stop
```

This will gracefully shut down the Ollama service.

Listing Available Models To check which models are installed locally and available for use, run:

List Installed Models

```
1 ollama list
```

This command outputs a list of all locally stored models, along with their sizes and versions.

6.3.1 Interacting with Ollama's Local API

Ollama runs a local REST API on 'http://localhost:11434', allowing interaction with models using HTTP requests. The following examples demonstrate how to generate text using the API with 'curl'

and Python.

Using Curl

Description

The following ‘curl’ command sends a request to the Ollama API, asking it to generate text based on a given prompt.

Using Curl

```
1 curl -X POST http://localhost:11434/api/generate \
2 -H "Content-Type: application/json" \
3 -d '{
4   "model": "mistral",
5   "prompt": "What is the capital of Germany?",
6   "stream": false
7 }'
```

Using Python

Alternatively, you can use Python’s ‘requests’ library to send the same request programmatically.

Using Python

```
1 import requests
2 import json
3
4 url = "http://localhost:11434/api/generate"
5 data = {
6   "model": "mistral",
7   "prompt": "What is the capital of Germany?",
8   "stream": False
9 }
10
11 response = requests.post(url, json=data)
12 print(response.json())
```

Using Ollama’s Python API

Ollama provides a python API to interact with its local server.

Using Ollama’s Python API

```
1 import ollama
2
```

```

3 # Load a local model
4 model = 'mistral'
5
6 # Generate a response
7 response = ollama.chat(model=model, messages=[{"role": "user", "content": "What is
     the capital of France?"}])
8
9 # Print the response
10 print(response['message']['content'])

```

6.3.2 A Local UI with Personal History for Ollama

To create a simple local UI with personal chat history for Ollama, we can use Flask. Below is an example of how to build such a system. You need to use pip to install flask before this can work.

Flask-Based Local Chat UI

```

1 from flask import Flask, request, jsonify, render_template
2 import ollama
3
4 app = Flask(__name__)
5
6 chat_history = [] # Stores full chat history
7
8 @app.route('/')
9 def index():
10     return render_template('index.html') # Serves the HTML UI
11
12 @app.route('/chat', methods=['POST'])
13 def chat():
14     user_input = request.json.get('message')
15
16     if not user_input:
17         return jsonify({'error': 'No message provided'}), 400
18
19     # Append current message to chat history
20     chat_history.append({"role": "user", "content": user_input})
21
22     # Send full conversation history to Ollama
23     response = ollama.chat(model="deepseek-r1:7b", messages=chat_history)
24
25     # Extract Ollama's response
26     bot_reply = response['message']['content']
27
28     # Append bot response to chat history
29     chat_history.append({"role": "assistant", "content": bot_reply})
30
31     return jsonify({'response': bot_reply, 'history': chat_history})

```

```

32
33 if __name__ == '__main__':
34     app.run(debug=True)

```

This simple Flask app allows users to interact with an LLM locally while maintaining chat history. We saved this as `ollama_flask_server.py` in the doce subdirectory `ollama_UI`. Also, we provide the file `index.html` in the subdirectory `templates` to control the UI.

Installing Dependencies

Before running the server, you need to install Flask. You can do this using pip:

Installing Flask

```
1 pip install flask
```

Ensure that you also have Ollama installed and running locally.

Setting Up the UI

The HTML file that provides the user interface should be saved as `index.html` in the `templates` subdirectory inside `code06`. Below is the content of this file:

index.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Chatbot</title>
7     <style>
8         body {
9             font-family: Arial, sans-serif;
10            margin: 0;
11            padding: 20px;
12            background-color: #f4f4f4;
13        }
14        #chat-container {
15            width: 50%;
16            max-width: 600px;
17            margin: auto;
18            background: white;
19            padding: 20px;
20            border-radius: 10px;
21            box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
22        }

```

```
23     #chat-box {
24         height: 300px;
25         overflow-y: auto;
26         border: 1px solid #ddd;
27         padding: 10px;
28         margin-bottom: 10px;
29         background: #fff;
30     }
31     .message {
32         padding: 8px;
33         margin: 5px 0;
34         border-radius: 5px;
35     }
36     .user { background: #d1e7fd; text-align: right; }
37     .bot { background: #e6e6e6; text-align: left; }
38     input, button {
39         width: 100%;
40         padding: 10px;
41         margin-top: 10px;
42         border: none;
43         border-radius: 5px;
44     }
45     button {
46         background: #007bff;
47         color: white;
48         cursor: pointer;
49     }
50     button:hover {
51         background: #0056b3;
52     }
53     </style>
54 </head>
55 <body>
56
57     <div id="chat-container">
58         <h2>Chatbot</h2>
59         <div id="chat-box"></div>
60         <input type="text" id="user-input" placeholder="Type a message...">
61         <script>
62             function sendMessage() {
63                 let userInput = document.getElementById("user-input").value;
64                 if (userInput.trim() === "") return;
65
66                 let chatBox = document.getElementById("chat-box");
67             }
68         </script>
69     </div>
70 
```

```

71         // Append user message
72         let userMessage = document.createElement("div");
73         userMessage.classList.add("message", "user");
74         userMessage.textContent = userInput;
75         chatBox.appendChild(userMessage);
76
77         document.getElementById("user-input").value = ""; // Clear input
78         chatBox.scrollTop = chatBox.scrollHeight; // Auto-scroll
79
80         // Send request to Flask server
81         fetch("/chat", {
82             method: "POST",
83             headers: { "Content-Type": "application/json" },
84             body: JSON.stringify({ message: userInput })
85         })
86         .then(response => response.json())
87         .then(data => {
88             let botMessage = document.createElement("div");
89             botMessage.classList.add("message", "bot");
90             botMessage.textContent = data.response;
91             chatBox.appendChild(botMessage);
92             chatBox.scrollTop = chatBox.scrollHeight;
93         })
94         .catch(error => console.error("Error:", error));
95     }
96
97     function handleKeyPress(event) {
98         if (event.key === "Enter") {
99             sendMessage();
100        }
101    }
102  </script>
103
104 </body>
105 </html>

```

Running the Server

After saving the Python script as `ollama_flask_server.py` and ensuring that `index.html` is in the correct location, navigate to the `code06` directory and run the following command:

Starting the Server

```
1 python 3_ollama_flask_server.py
```

Once the server is running, you should see output like:

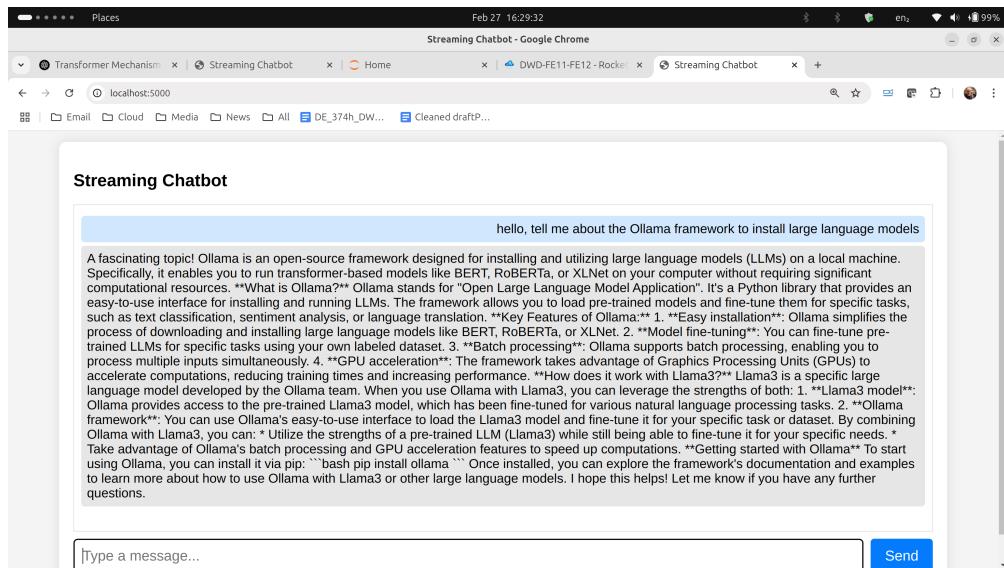


Figure 6.1: Ollama chatbot interface running a local LLM on your laptop, with local UI.

```
Server Output
1 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Accessing the User Interface

To interact with the chatbot, open a web browser and go to:

```
Accessing the UI
1 http://127.0.0.1:5000/
```

From there, you can enter messages in the input field, and the chatbot will respond accordingly. This simple local chat UI provides an easy way to interact with Ollama using a web-based interface while maintaining personal chat history.

6.3.3 Streaming Responses from Ollama with Markdown Formatting

In interactive environments such as Jupyter Notebooks or user interfaces (UI), it can be beneficial to stream responses from Ollama in real time while applying Markdown formatting to be able to display e.g. bold words or headings. The following example demonstrates how to achieve this using Python.

The function `stream_ollama_markdown()`:

- Sends a request to a locally running Ollama server.
- Receives responses in a streaming fashion.

- Formats the output dynamically using Markdown to enhance readability.
- Updates the output in place without reloading the cell.

The function `format_markdown()` ensures that:

- Capitalized words and model names are bolded for emphasis.
- Titles such as "Model Features:" are converted into Markdown headings.
- Lists are properly formatted for readability.

The demo implementation is given below:

```
Streaming Markdown Output from Ollama

1 import requests
2 import json
3 import re
4 from IPython.display import display, Markdown, clear_output
5
6 def format_markdown(text):
7     text = text.replace("\n", "\n\n")
8     text = re.sub(r'\b([A-Z][a-z]+(?:\s+[A-Z][a-z]+)*)\b', r'**\1**', text)
9     text = re.sub(r'\b([A-Z]{3,})\b', r'**\1**', text)
10    return text
11
12 def stream_ollama_markdown(prompt, model="llama3"):
13     url = "http://localhost:11434/api/generate"
14     data = {"model": model, "prompt": prompt, "stream": True}
15
16     response = requests.post(url, json=data, stream=True)
17     buffer = ""
18     output_display = display(Markdown(""), display_id=True)
19
20     for chunk in response.iter_lines():
21         if chunk:
22             try:
23                 data = json.loads(chunk.decode("utf-8"))
24                 text = data.get("response", "")
25                 if text:
26                     buffer += text
27                     clear_output(wait=True)
28                     output_display.update(Markdown(format_markdown(buffer)))
29             except json.JSONDecodeError:
30                 pass
31
32     # Final display (in case last chunk isn't shown)
33     clear_output(wait=True)
34     output_display.update(Markdown(format_markdown(buffer)))
35     # No return
```

```

36
37
38 # Try it
39 stream_ollama_markdown("Explain the concept of self-attention in Transformers.")

```

This method enables seamless streaming of responses from Ollama, allowing for an interactive and visually structured output in Jupyter environments. We also provide a demo implementation of a streaming interface for your flask based user interface (UI) to Ollama, see `ollama_flask_server_streaming.py` with the `index_stream.html` in the `templates/` directory.

6.3.4 Available Models for Download and Response Speed

There are more than 150 different LLMs for download and use on Ollama. Here is a selection with some highlighted features.

1. **DeepSeek-R1 (1.5B – 671B)** A powerful model family covering a wide range of sizes, from small-scale 1.5B to massive 671B parameters, making it flexible for different applications.
2. **Llama3 (8B, 70B)** One of the most anticipated next-generation models from Meta, optimized for efficiency and capable of handling diverse NLP tasks with strong performance.
3. **Mistral (7B)** A lightweight model with excellent reasoning capabilities, outperforming many larger models in specific tasks while remaining efficient.
4. **Qwen2.5 (0.5B – 72B)** Alibaba's Qwen models are designed for multilingual tasks and reasoning, with a focus on applications requiring high levels of comprehension.
5. **CodeLlama (7B – 70B)** An LLM specifically optimized for code generation and software development, making it useful for programmers and AI-assisted coding.
6. **Gemma (2B, 7B)** Google's Gemma models are efficient and optimized for deployment, designed for responsible AI usage and fine-tuned performance.
7. **Mixtral (7B, 22B)** A mixture-of-experts (MoE) model that enables highly efficient inference while maintaining competitive accuracy in language tasks.
8. **Starcoder2 (3B – 15B)** A coding model built for AI-assisted development, capable of understanding and generating code efficiently across multiple programming languages.
9. **Orca Mini (3B – 70B)** A distilled model trained with advanced reasoning capabilities, making it an excellent option for lightweight yet powerful AI assistants.
10. **Llava (7B – 34B)** A vision-language model capable of understanding images along with text, useful for applications in multimodal AI tasks like captioning and visual question answering.

Large Language Models (LLMs) have become increasingly accessible for local installations, enabling users to process text without relying on cloud services. One crucial performance metric for such models is their response time, typically measured in milliseconds per token.

	model	tokens	duration_sec	tokens/sec	sec/page
0	mistral	220	52.64	4.18	79.68
1	llama3	270	49.18	5.49	60.66
2	deepseek-r1:7b	613	121.55	5.04	66.03
3	deepseek-r1:1.5b	561	28.68	19.56	17.02

Figure 6.2: Ollama on windows wsl on a regular workstation or laptop. With a light NVIDIA GPU (4GB) speed is about 2.5 times faster. On a linux laptop I got 33 token per second for deepseek-r1:1.5b.

The speed at which an LLM generates tokens depends on hardware capabilities, model size, and optimization techniques. The DeepSeek-R1:1.5B model on a typical Linux laptop generates one token every 30 milliseconds (ms). The number of tokens generated in a given time frame is given by:

$$N = \frac{T}{t}, \quad (6.6)$$

where:

- N is the number of tokens,
- T is the total time available (in ms),
- t is the time per token (30 ms in this case).

For practical scenarios:

- In one second ($T = 1000$ ms):

$$N = \frac{1000}{30} \approx 33.33 \text{ tokens per second.} \quad (6.7)$$

- In ten seconds ($T = 10000$ ms):

$$N = \frac{10000}{30} \approx 333.33 \text{ tokens in ten seconds.} \quad (6.8)$$

To estimate the text length corresponding to these tokens, we assume:

- One token typically consists of about 4 characters (including spaces and punctuation).
- One token represents approximately 0.75 words in standard English text.

Thus, for 333 tokens:

- Character count: $333 \times 4 = 1332$ characters.
- Word count: $333 \times 0.75 \approx 250$ words.

This corresponds to roughly one page of text in a typical document.

The response time of an LLM either online as a service or installed on a local computer significantly influences usability. At 30 ms per token, the DeepSeek-R1:1.5B model can generate around 33 tokens per second or 250 words in ten seconds. Optimizing performance with hardware acceleration (e.g., GPUs, tensor cores) can further improve these figures, making local AI processing a viable alternative to cloud-based models.

Recommendation

Today, medium size LLMs can be run for individual users on a laptop, with acceptable response times for interactive applications. Alternatively and depending on privacy needs the use of platform accounts by AI companies provides faster access to the basic LLM functionality. User services based on either locally or remotely hosted LLM functionality is at hand and can be combined with specific local services and data.

Chapter 7

LLM with Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a powerful approach that enhances language models by incorporating external knowledge retrieval. This chapter guides the user through setting up and using RAG, with practical examples.

Introduction to RAG Traditional LLMs rely solely on their pre-trained knowledge. RAG extends this by searching a document database for relevant context before generating a response. This improves accuracy, factuality, and adaptability to domain-specific knowledge.

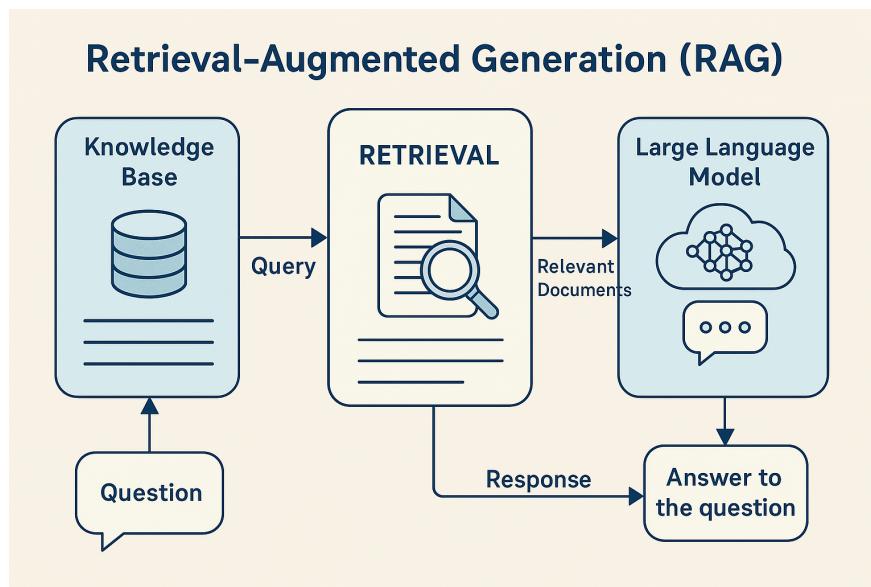


Figure 7.1: Retrieval-Augmented Generation (RAG) employs the intelligence of Large Language Models (LLM) in combination with local knowledge and databases.

7.1 Preparing Documents

Installing Required Dependencies To set up RAG, install the necessary libraries:

Install Dependencies

```
1 !pip install sentence-transformers==2.2.2 transformers==4.31.0
2 !pip install faiss-cpu openai numpy pymupdf
```

To get compatible versions of transformers and sentence-transformers might be a little tricky. I had to go back to python3.11 to get this running in a special virtual environment.

First, let us work with documents which are in a folder tree starting with its root node documents/. We start by recursively loading text and PDF documents from this folder, marching through the tree. For pdf documents, we will use some standard tools to extract the text from the documents - there are more sophisticated packages, but we put simplicity first for the moment.

Loading Documents

```
1 import numpy as np
2 import os
3 import fitz # PyMuPDF for PDF text extraction
4
5 # Function to extract text from PDF
6 def extract_text_from_pdf(pdf_path):
7     doc = fitz.open(pdf_path)
8     text = ""
9     for page in doc:
10         text += page.get_text()
11     return text
12
13 # Load documents from a folder recursively
14 def load_documents_from_folder(folder_path):
15     documents = []
16     file_info = []
17
18     for dirpath, dirnames, filenames in os.walk(folder_path):
19         # Exclude .git directories
20         dirnames[:] = [d for d in dirnames if not d.startswith('.git')]
21
22         for filename in filenames:
23             if filename.startswith('.git'):
24                 continue # Skip any .git files
25
26             file_path = os.path.join(dirpath, filename)
27             if filename.endswith(('.f90','.txt', '.org', '.sh', '.toml','.pdf')) \
28                 or '.' not in filename: # Load .txt, .sh, .pdf, and files without
29                 extensions
30                 try:
```

```

30             if filename.endswith('.pdf'):
31                 content = extract_text_from_pdf(file_path)
32             else:
33                 with open(file_path, 'r', encoding='utf-8') as file:
34                     content = file.read()
35             documents.append(content)
36             file_info.append((filename, file_path))
37             print(f"Added to DB: {file_path}")
38     except Exception as e:
39         print(f"Failed to process {file_path}: {e}")
40
41     return documents, file_info

```

We call the function to load all available documents by

Loading Documents

```

1 folder_path = './documents' # Replace with your folder path
2 documents, file_info = load_documents_from_folder(folder_path)

```

As an example we checkout the publically available ICON model by

Checkout ICON model

```
1 git clone git@gitlab.dkrz.de:icon/icon-model.git
```

In this case, the output is

ICON model from icon-model.org loaded

```

1 Added to DB: ./documents/configure
2 Added to DB: ./documents/make_rundscripts
3 Added to DB: ./documents/utils/install-sh
4 Added to DB: ./documents/utils/move_to_prefix.sh
5 Added to DB: ./documents/utils/patch_namelist
6 Added to DB: ./documents/utils/timewarp
7 Added to DB: ./documents/utils/icon_sorted_deps.sh
8 Added to DB: ./documents/utils/id++
9 Added to DB: ./documents/utils/filelock
10 Added to DB: ./documents/utils/mkexp/namelist2config
11 Added to DB: ./documents/utils/mkexp/mkexp
12 Added to DB: ./documents/utils/mkexp/upexp
13 Added to DB: ./documents/utils/mkexp/unmergeconfig
14 Added to DB: ./documents/utils/mkexp/selconfig
15 Added to DB: ./documents/utils/mkexp/importexp
16 Added to DB: ./documents/utils/mkexp/editexp
17 ...

```

7.2 Generating Embeddings for Documents

To retrieve relevant information, we transform text into **numerical embeddings** using a **transformer model**. These embeddings capture the *semantic meaning* of text, allowing for similarity-based retrieval and efficient search operations. A widely used approach for generating such embeddings is **sentence transformers**, which map textual data into a high-dimensional vector space.

In the following implementation, we utilize the all-MiniLM-L6-v2 model from the sentence-transformers library. This model provides a compact yet efficient method for encoding textual information. The embeddings are computed using **mean pooling** over the token representations, ensuring that the vector captures the full meaning of the sentence. Additionally, we apply **L2 normalization** to facilitate cosine similarity comparisons.

Generating Embeddings

```

1 from sentence_transformers import SentenceTransformer
2
3 embedder = SentenceTransformer('all-MiniLM-L6-v2')
4
5 # Load documents and create embeddings
6 documents, file_info = load_documents_from_folder("documents/")
7 embeddings = embedder.encode(documents, convert_to_numpy=True)
8
9 # Sanity check
10 print("Loaded documents:", len(documents))
11 print("Embedding shape:", embeddings.shape)
12 print("Example file:", file_info[0])
13 print("Example snippet:\n", documents[0][:200])

```

The function `embedder.encode` tokenizes the input text, processes it through the transformer model, and computes a **mean-pooled representation** over the token dimension. The final embeddings are **normalized**, ensuring that similarity computations (e.g., cosine similarity) remain well-scaled.

Such embeddings enable applications in **semantic search, clustering**, and **document classification** by mapping textual data into a structured numerical space that can be efficiently queried.

In our case we obtain

Check embeddings

```

1 Loaded documents: 2391
2 Embedding shape: (2391, 384)
3 Example file: ('make_rungs', 'documents/make_rungs')
4 Example snippet:
5  #!/bin/bash
6
7 # ICON
8 #
9 # -----
10 # Copyright (C) 2004-2024, DWD, MPI-M, DKRZ, KIT, ETH, MeteoSwiss

```

```
11 # Contact information: icon-model.org
12 # See AUTHORS.TXT for a list
```

Our embeddings variable has a shape of (2391, 384), which means there are 2391 rows, i.e. 2391 text inputs (documents, sentences, or paragraphs). We have 384 columns, i.e. each text input is represented as a 384-dimensional vector.

How the Embeddings Work. The closer two embeddings are (e.g., using cosine similarity), the more semantically similar their corresponding texts are. The high-dimensional space (384D) allows the model to capture complex semantic relationships between texts.

Retrieving Relevant Documents When a user asks a question, we find the most relevant documents by searching the FAISS index.

Querying the Vector Database

```
1 def query_vector_db(query, k=2):
2     query_embedding = get_embedding(query)
3     distances, indices = index.search(query_embedding, k)
4     return [documents[idx] for idx in indices[0]]
5
6 query = "What is the main functionality of BACY?"
7 retrieved_docs = query_vector_db(query)
8 print(retrieved_docs)
```

Setting Up a Vector Database with FAISS To efficiently search for similar text embeddings, we store them in a **FAISS index**. FAISS (Facebook AI Similarity Search) is an optimized library for fast **nearest neighbor search** in high-dimensional spaces. Instead of performing a brute-force comparison of all pairs of embeddings, FAISS allows us to *index and retrieve* the most relevant embeddings efficiently.

FAISS supports different types of indexes, but the most basic and commonly used one is **IndexFlatL2**, which stores vectors and enables fast k -nearest neighbor (KNN) search using the L_2 (Euclidean) distance.

Setting Up FAISS Index

```
1 import faiss
2
3 # Create FAISS index
4 dimension = embeddings.shape[1]
5 index = faiss.IndexFlatL2(dimension)
6 index.add(embeddings)
```

The IndexFlatL2 structure is an exact nearest-neighbor search index that:

- Stores all embeddings in memory.
- Computes pairwise distances using **L2 norm (Euclidean distance)**.
- Allows fast similarity search without additional quantization.

Querying the FAISS Index. Once the FAISS index is built, we can *perform similarity searches* by converting a text query into an embedding and retrieving the k -nearest neighbors from the indexed documents. The following function allows querying the vector database and retrieving the most relevant documents.

Querying the FAISS Index

```

1 # Query function
2 def query_vector_db(query, k=2):
3     """
4         Searches the FAISS index for the k most similar embeddings to the query.
5
6     Parameters:
7         query (str): Input text to search for similar documents.
8         k (int): Number of nearest neighbors to retrieve.
9
10    Returns:
11        list of tuples: Each tuple contains (retrieved_document, file_info).
12    """
13    query_embedding = get_embedding(query).astype(np.float32) # Convert to FAISS
14    format
15    distances, indices = index.search(query_embedding, k) # Perform similarity
16    search
17    return [(documents[idx], file_info[idx]) for idx in indices[0]]

```

To demonstrate how the FAISS search works, we run an example query:

Example Query to FAISS

```

1 import re
2 import unicodedata
3
4 def clean_text(text, max_chars=500):
5     text = unicodedata.normalize("NFC", text) # Normalize Unicode
6     text = ''.join(c if c.isprintable() else ' ' for c in text) # Keep printable
6     chars
7     return re.sub(r'\s+', ' ', text).strip()[:max_chars].encode("utf-8", "ignore")
7     .decode("utf-8")
8
9 # Example usage
10 query_text = "Cloud cover intro"
11 results = query_vector_db(query_text, k=6)
12
13 for doc, info in results:
14     print("File Info:", clean_text(str(info)))
15     print("\nDocument:", clean_text(doc))
16     print("-" * 50)

```

This example queries the FAISS index for the $top k$ *most relevant documents* related to the weather forecast. The function:

- Converts the input query into an *embedding*.
- Searches the FAISS index for the k most similar embeddings.
- Returns the corresponding **documents and metadata**.

By retrieving the closest matches, we enable *semantic search*, allowing users to find *contextually similar documents* rather than relying on simple keyword matching. For our ICON example where we ask for *Cloud cover intro* we get the results indicated in Figure 7.2.

Figure 7.2: Vector Database Output

7.3 Using an LLM Locally or with OpenAI for Response Generation

Once we retrieve relevant documents, we provide them as context to OpenAI's LLM or any other large language model. This allows us to enhance responses with domain-specific information and improve accuracy.

Setting up OpenAI API. To use OpenAI, we need an *OpenAI platform account* and an API key (`OPENAI_API_KEY`). If you have stored this key as an environment variable, you can initialize the API client as follows:

Connection to OpenAI

```

1 import openai
2 import os
3
4 # Initialize OpenAI API
5 client = openai.Client(api_key=os.getenv('OPENAI_API_KEY')) # Updated API
    initialization

```

If no error occurs, the API connection is successfully established.

Querying OpenAI with Retrieved Context. Once we have access to OpenAI's API, we can define a function to query the model. The function *retrieves relevant documents* from our vector database

and passes them as context to the model. This enables OpenAI to provide responses based on our own data rather than generic knowledge. Here, we will employ the model *gpt-4o-mini* which is known for its speed and which is much cheaper than the flagship models.

Generating Responses with OpenAI

```

1 # Step 6: Use OpenAI to discuss results
2 def chat_with_openai(query):
3     retrieved_docs = query_vector_db(query, k=20)
4     context = "\n\n".join([f"File: {file[0]}\nContent: {doc}" for doc, file in
5     retrieved_docs])
6
7     response = client.chat.completions.create(
8         model="gpt-4o-mini", # using the gpt-4o-mini model
9         messages=[
10             {"role": "system", "content": "You are a helpful assistant."},
11             {"role": "user", "content": f"Based on the following documents, answer
12             the question:\n{context}\n\nQuestion: {query}"}
13         ]
14     )
15
16     answer = response.choices[0].message.content
17
18     # Append file references to the answer
19     file_references = "\n\nReferenced Files:\n" + "\n".join([f"{file[0]}: {file
20     [1]}" for _, file in retrieved_docs])
21
22     return answer + file_references

```

How it works. The function `chat_with_openai(query)` operates as follows:

- *Retrieves relevant documents* using the `query_vector_db(query, k=20)` function.
- *Formats* the retrieved text into a structured context.
- *Sends the context* to OpenAI's model, prompting it to answer the query based on the retrieved content.
- *Returns the generated response*, along with file references for transparency.

Example query. If a user searches for a specific topic, such as:

Example Query Execution

```

1 # OpenAI Answers
2 query = "How is cloud cover treated in ICON?"
3 answer = chat_with_openai(query)
4
5 from IPython.display import display, Markdown
6 display(Markdown(answer))

```

This ensures that OpenAI generates responses grounded in *our own database*, reducing hallucination and improving relevance.

Here, the result for our example is as follows.

The ICON (ICOsapherical Nonhydrostatic) model is a state-of-the-art atmospheric model developed for weather and climate simulations. Its structure is designed to efficiently handle both global and regional simulations, including various atmospheric and surface physics processes. Below is an overview of its key components and architecture:

1. Grid Structure

Horizontal Grid: ICON uses a spherical grid based on the projection of an icosahedron. This grid can be refined through edge bisection, leading to triangular or hexagonal elements. The horizontal representation allows high resolution and flexibility in grid configuration.

Vertical Grid: The vertical structure uses hybrid coordinates, where the atmosphere is discretized into horizontal layers. These vertical layers can be based on pressure or height coordinates, accommodating both hydrostatic and non-hydrostatic dynamics.

2. Core Components

Dynamics: ICON features a non-hydrostatic dynamical core that enables accurate simulations of complex atmospheric phenomena, such as storms and convective processes. It employs a C-staggering scheme for state variables to improve computational efficiency, especially for advection and momentum transport.

Physics: The model integrates various physics packages for parameterizing physical processes. This includes:

Cloud microphysics to simulate cloud formation and precipitation.

Radiation schemes (like RRTMG) to calculate the transfer of solar and terrestrial radiation.

Convection schemes to model the vertical transport of heat and moisture in the atmosphere.

Surface processes involving land-atmosphere interactions, including soil moisture and vegetation effects.

3. Parallelization

ICON is designed for high-performance computing environments, utilizing both MPI (Message Passing Interface) and OpenMP for parallelization. This allows it to run efficiently on modern supercomputers, accommodating large-scale simulations with many processors.

4. Namelists

The configuration of the ICON model is managed through namelists, which are structured text files containing parameters for different aspects of the model. This includes settings for grid configurations, physics schemes, and output specifications.

5. Modules and Naming Conventions

The code is organized into modules for different functionalities (such as dynamics, radiation, and microphysics). Each module contains subroutines and functions that handle specific tasks. A clear naming convention is followed for ease of understanding, with prefixes used to indicate the nature and scope of variables.

6. Output and Diagnostics

ICON is equipped with flexible output capabilities that can generate various diagnostic fields related to atmospheric states and physical processes, including radar reflectivity and cloud cover. Output can be tailored to specific requirements utilizing naming conventions for files and variable names.

7. Community and Documentation

The development of ICON follows best coding practices as outlined in guidelines for readability and maintainability. The model is supported by detailed documentation, ensuring users can effectively configure and use the model for their research needs.

8. External Packages

ICON can integrate with external libraries and frameworks for additional functionalities, allowing for advanced features like data assimilation and coupling with ocean models.

In summary, the ICON model embodies a sophisticated structure that combines modern computational techniques with detailed atmospheric science, enabling high-resolution weather and climate simulations across a range of applications. Its modular architecture and parallel processing capabilities make it suitable for use in both research and operational forecasting environments.

Referenced Files: [...]

An alternative to OpenAI's API is to use *Ollama*, which allows running large language models locally. This provides greater control over data privacy and removes dependency on external services. The following function queries a locally hosted model, such as Mistral or DeepSeek-R1:1.5b, using retrieved documents from the vector database.

Querying Ollama Locally

```

1 import ollama
2
3 def chat_with_ollama(query, model="mistral"):
4     """Queries Ollama's local LLM (Mistral or DeepSeek) with retrieved VectorDB
5     documents."""
6     retrieved_docs = query_vector_db(query, k=20)
7     context = "\n\n".join([f"File: {file[0]}\nContent: {doc}" for doc, file in
8     retrieved_docs])
9
10    full_prompt = f"Based on the following documents, answer the question:\n{
11        context}\n\nQuestion: {query}"
12
13    response = ollama.chat(model=model, messages=[{"role": "user", "content": full_prompt}])
14
15    answer = response['message']['content']
16
17    # Append file references
18    file_references = "\n\nReferenced Files:\n" + "\n".join([f"{file[0]}: {file[1]}" for _, file in retrieved_docs])
19
20    return answer + file_references

```

The function first retrieves relevant documents from the vector database and formats them into a structured prompt. It then queries a local Ollama instance running the selected model and appends file references to ensure traceability.

The following examples demonstrate how to use DeepSeek-R1:1.5b and Mistral to answer a query regarding the structure of the ICON model.

Example Query with DeepSeek-R1:1.5b

```

1 query_text = "Tell me about the structure of the ICON model"
2 answer = chat_with_ollama(query_text, model="deepseek-r1:1.5b")
3
4 # Display response as formatted Markdown

```

```
5 display(Markdown(answer))
```

Example Query with Mistral

```
1 answer = chat_with_ollama(query_text, model="mistral")
2
3 # Display response as formatted Markdown
4 display(Markdown(answer))
```

This approach allows flexible switching between different local models, depending on the required accuracy, speed, or computational efficiency. It works locally on your laptop, but might take slightly longer for answers.

Here is the deepseek answer:

Alright, I need to tell someone about the structure of the ICON model. Let's break this down step by step.

First, I should recall what the ICON model is. From my limited knowledge, it's a global numerical weather prediction model used for atmospheric simulations. The user mentioned various files like balfringu_nvvidia and others related to it. These probably contain the source code and documentation.

Looking at the files provided:

[...]

In summary, understanding the ICON model involves knowing its architecture, how it processes data through various components, and the parameters that control this processing. The structure is complex yet modular, allowing for detailed physical modeling on a global scale.

The ICON (Icosahedral Non-Oceanic Model) model is a global numerical weather prediction system designed to simulate atmospheric circulation. Here's its structured overview:

Structure of the ICON Model

Configuration Management:

[...]

Computes forward in time using processes like hydrostatic pressure, moist static stability, gravity wave propagation, advection, radiation, etc.

Utilizes parallelization on high-performance computing clusters for efficiency.

Key Features

High Scalability: Designed for efficient computation on clusters of CPUs.

Global Coverage: Serves as a foundational component for weather models.

Flexibility: Adjustable parameters allow customization for specific applications.

The ICON model's structure is modular, managing data through components that ensure accurate and scalable global atmospheric modeling.

Referenced Files: [...] Namelist_overview.pdf: ./documents/doc/Namelist_overview.pdf
icon_grid.pdf: ./documents/doc/technical/icon_grid.pdf dot_cdprc: ./documents/schedulers/ecmwf/gen/dot_cdprc icon_standard.pdf: ./documents/doc/style/icon_standard.pdf [...]

7.4 Saving and Reloading the Vector Database, Collecting Search Originals, Chunking long Documents

To avoid recomputing embeddings every time, we save the FAISS index to disk. This allows us to reload it efficiently for future searches.

Saving and Loading FAISS Index

```

1 import faiss
2 import os
3
4 # Save FAISS index
5 faiss.write_index(index, "vector_db.index")
6 print("Vector database saved.")
7
8 # Ensure file exists before loading
9 if os.path.exists("vector_db.index"):
10     index = faiss.read_index("vector_db.index")
11     print("Vector database loaded.")
12 else:
13     print("Error: FAISS index file not found.")

```

This method ensures that the FAISS index persists across sessions, eliminating the need for recomputing embeddings. If additional metadata, such as document mappings, is needed, they must be stored separately in a structured format (e.g., JSON or a database).

Search Results: Documents. Often it can be helpful to look into the results of a search directly. The following code saves all retrieved documents into a designated results folder, while ensuring previous results are backed up by renaming the existing folder.

Save Search Documents

```

1 import os
2 import shutil
3
4 def backup_and_create_results_folder(base_folder="results"):
5     """Backs up existing results folder by renaming it to results_nnn and creates
6     a new empty folder."""
7
8     if os.path.exists(base_folder):
9         counter = 1
10        while os.path.exists(f"{base_folder}_{counter:03d}"):
11            counter += 1
12            backup_folder = f"{base_folder}_{counter:03d}"
13            shutil.move(base_folder, backup_folder)
14            print(f"Existing results folder backed up as: {backup_folder}")
15
16    os.makedirs(base_folder, exist_ok=True)
17    return base_folder

```

```

17
18 def copy_retrieved_documents(query, k=10, results_folder="results"):
19     """Finds relevant documents using FAISS and copies them to the results folder,
20     saving the query."""
21
22     # Backup old results and create new folder
23     results_folder = backup_and_create_results_folder(results_folder)
24
25     # Retrieve relevant documents
26     retrieved_docs = query_vector_db(query, k)
27
28     copied_files = []
29
30     for _, file_info in retrieved_docs:
31         file_path = file_info[1] # Assuming file_info[1] contains the file path
32
33         if os.path.exists(file_path):
34             dest_path = os.path.join(results_folder, os.path.basename(file_path))
35             shutil.copy(file_path, dest_path)
36             copied_files.append(dest_path)
37         else:
38             print(f"Warning: File not found - {file_path}")
39
40     # Save query text
41     query_path = os.path.join(results_folder, "query.txt")
42     with open(query_path, "w", encoding="utf-8") as f:
43         f.write(query)
44
45     print(f"Copied {len(copied_files)} files to {results_folder}")
46     print(f"Query saved in {query_path}")

```

Example Usage. The following example retrieves documents related to turbulence schemes in ICON and saves them in the results folder.

Example Query and Document Copy

```

1 query_text = "What turbulence scheme in ICON?"
2 copy_retrieved_documents(query_text, k=20)

```

Displaying the Retrieved Files. Once the search is complete, the saved documents can be listed to verify their contents. The following example shows the typical contents of the results folder after a search.

alps_mch_test_gpu	daint_cpu_cce	flux_diagram-crop.pdf
icon_atm_echam_phy_scidoc.pdf	icon_technical.pdf	NOTICE
AUTHORS.txt	daint_cpu_nvidia_mixed	flux_diagram.pdf
icon_tuning_vars.pdf	query.txt	icon_grid.pdf
balfrin_gpu_nvidia_mixed	dep5	horeka_cpu_nvhpcl
lmclouds2010.pdf	README	icon_standard.pdf

This setup ensures that all relevant documents are efficiently stored, organized, and available for analysis.

Adding some Tutorial to the Search Database. Providing only some code base is often not sufficient to understand it. One important step is to add more targeted material to the search.

Often some tutorial is available online but is not part of the source code repository. To ensure efficient retrieval, we split the tutorial into small chunks, embed each chunk, and add them to the FAISS vector database.

Processing ICON Tutorial to get pages

```

1 from PyPDF2 import PdfReader
2
3 def extract_pages_from_pdf(pdf_path):
4     """Reads a PDF and returns a list of page texts."""
5     reader = PdfReader(pdf_path)
6     return [page.extract_text() for page in reader.pages]
7
8 # Load the tutorial as separate pages
9 tutorial_pdf = "ai_tutorial.pdf"
10 tutorial_pages = extract_pages_from_pdf(tutorial_pdf)
11
12 # Check output
13 print(f"Extracted {len(tutorial_pages)} pages.")

```

Chunking and Embedding

```

1 tutorial_embeddings = np.vstack([get_embedding(page) for page in tutorial_pages])
2
3 tutorial_file_name = "ai_tutorial.pdf"
4 tutorial_page_refs = [f"{tutorial_file_name} - Page {i+1}" for i in range(len(
    tutorial_pages))]
5
6 # Add to main lists
7 documents.extend(tutorial_pages)
8 file_info.extend([(ref, tutorial_file_name) for ref in tutorial_page_refs])
9
10 index.add(tutorial_embeddings)

```

This ensures that the tutorial is available for search queries, returning relevant sections instead of requiring manual document lookup. Here, I added a draft of the AI tutorial you are reading.

Query with Code and Tutorial included

```

1 # OpenAI Answers
2 query = "What NN architectures did we talk about? CNN? GNN? Transformers?"
3 answer = chat_with_openai(query)
4

```

```
5 from IPython.display import display, Markdown
6 display(Markdown(answer))
```

OpenAI with gpt-4o-mini comes with the following answer:

Based on the documents provided, the discussed neural network (NN) architectures include:

Feedforward Neural Networks (FFNN) - The simplest form of artificial neural networks where information moves in one direction from the input layer through hidden layers to the output layer.

Convolutional Neural Networks (CNN) - Powerful architectures typically used for image processing and can be applied to one-dimensional data, such as time series and function classification.

Recurrent Neural Networks (RNN) - Particularly highlighted for sequence learning, with types like Long Short-Term Memory (LSTM) networks, which improved the processing of language sequences.

Transformers - Introduced groundbreaking innovations with self-attention mechanisms and parallel processing, becoming the standard for modern language models.

The documents specifically mention CNNs and Transformers, while RNNs (including LSTMs) are noted for their advancements in sequence processing. There was no mention of Graph Neural Networks (GNN) in the provided materials.

Referenced Files: ai_tutorial.pdf - Page 5: ai_tutorial.pdf ai_tutorial.pdf - Page 188: ai_tutorial.pdf ai_tutorial.pdf - Page 111: ai_tutorial.pdf ai_tutorial.pdf - Page 182: ai_tutorial.pdf ai_tutorial.pdf - Page 101: ai_tutorial.pdf ai_tutorial.pdf - Page 90: ai_tutorial.pdf ai_tutorial.pdf - Page 124: ai_tutorial.pdf ai_tutorial.pdf - Page 130: ai_tutorial.pdf ai_tutorial.pdf - Page 113: ai_tutorial.pdf ai_tutorial.pdf - Page 192: ai_tutorial.pdf ai_tutorial.pdf - Page 190: ai_tutorial.pdf ai_tutorial.pdf - Page 102: ai_tutorial.pdf ai_tutorial.pdf - Page 191: ai_tutorial.pdf ai_tutorial.pdf - Page 87: ai_tutorial.pdf ai_tutorial.pdf - Page 120: ai_tutorial.pdf ai_tutorial.pdf - Page 131: ai_tutorial.pdf ai_tutorial.pdf - Page 114: ai_tutorial.pdf ai_tutorial.pdf - Page 193: ai_tutorial.pdf ai_tutorial.pdf - Page 126: ai_tutorial.pdf ai_tutorial.pdf - Page 167: ai_tutorial.pdf

Mistral is getting this as well in a very concise way.

Mistral Query

```
1 answer = chat_with_ollama(query, model="mistral")
2
3 # Display response as formatted Markdown
4 display(Markdown(answer))
```

In this text, we talked about three types of neural network architectures: Convolutional Neural Networks (CNN), Graph Neural Networks (GNN), and Transformers. The CNN architecture was discussed in the context of image processing tasks, whereas Transformers were mentioned in relation to large language models and the self-attention mechanism. It appears that GNN wasn't directly addressed in this particular part of the text, but it could be inferred from other sections discussing graph-based problems like

social network analysis or molecular simulations.

Referenced Files: ai_tutorial.pdf - Page 5: ai_tutorial.pdf ai_tutorial.pdf - Page 188:
 ai_tutorial.pdf ai_tutorial.pdf - Page 111: ai_tutorial.pdf ai_tutorial.pdf - Page 182:
 ai_tutorial.pdf ai_tutorial.pdf - Page 101: ai_tutorial.pdf ai_tutorial.pdf - Page 90:
 ai_tutorial.pdf ai_tutorial.pdf - Page 124: ai_tutorial.pdf ai_tutorial.pdf - Page 130:
 ai_tutorial.pdf ai_tutorial.pdf - Page 113: ai_tutorial.pdf ai_tutorial.pdf - Page 192:
 ai_tutorial.pdf ai_tutorial.pdf - Page 190: ai_tutorial.pdf ai_tutorial.pdf - Page 102:
 ai_tutorial.pdf ai_tutorial.pdf - Page 191: ai_tutorial.pdf ai_tutorial.pdf - Page 87:
 ai_tutorial.pdf ai_tutorial.pdf - Page 120: ai_tutorial.pdf ai_tutorial.pdf - Page 131:
 ai_tutorial.pdf ai_tutorial.pdf - Page 114: ai_tutorial.pdf ai_tutorial.pdf - Page 193:
 ai_tutorial.pdf ai_tutorial.pdf - Page 126: ai_tutorial.pdf ai_tutorial.pdf - Page 167:
 ai_tutorial.pdf

In general, chunking and decomposition of the material into good and adequate parts is a very important part of the whole process. The LLM has limited context size and using the right information is a crucial part of answering a query or taking part in a discussion.

7.5 End-to-End AI-Powered Answer Pipeline

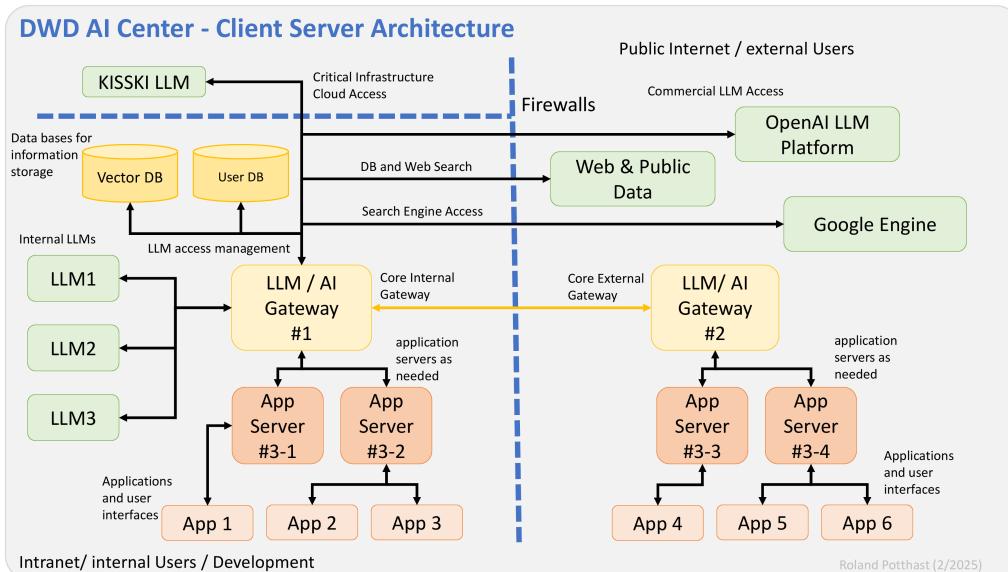


Figure 7.3: Client-server architecture for AI-powered search and answer generation.

This section explains the primary components of a pipeline that combines Google search, web scraping, and a Large Language Model (LLM) to generate context-aware answers from real-time web data.

Overview:

- `google_search`: queries Google Custom Search to retrieve relevant URLs.
- `scrape_website`: fetches content from the found websites.

- `generate_answer`: sends a prompt to OpenAI's GPT model for generating answers.
- `answer_from_google`: coordinates the complete process end-to-end.

This section provides a structured explanation of each core function used in the pipeline that combines Google search, web scraping, and language model-based answering.

search_google This function uses the Google Custom Search API to perform a web search for a given query. It takes as input the query string, a valid API key, a Custom Search Engine ID, and the desired number of results. If the request is successful, it parses the JSON response and extracts the top result URLs along with their titles. It also prints a nicely formatted list of these results for reference. If the request fails, an error message is printed and an empty list is returned.

scrape_website This function fetches the content of a given URL using a standard HTTP GET request with a browser-like user agent. It parses the response HTML using BeautifulSoup and extracts up to 15 paragraph elements (`<p>`) to form a readable body of text. If no paragraph content is found or an error occurs during the request, a fallback message is returned.

generate_answer This function sends a prompt to the OpenAI GPT API (using model `gpt-4o-mini`) to generate a natural language answer. It initializes the OpenAI client using the provided API key, structures the input as a chat-like conversation with a system and a user message, and returns the generated content from the assistant. This encapsulates the language generation capabilities of the model.

answer_from_google This is the main orchestration function that integrates the other components. It performs the full pipeline: searching Google, scraping the top URLs, formatting the content, and sending it to the LLM for summarization and answering. The function constructs a prompt that includes the original query and the concatenated extracted text from multiple sources, and then calls `generate_answer` to retrieve the final result.

Code Implementation:

```
ai search pipeline

1 import requests
2 from bs4 import BeautifulSoup
3 import openai
4
5 # Function to perform a Google Search and return top result URLs
6 def search_google(query, api_key, search_engine_id, num_results=5):
7     url = f"https://www.googleapis.com/customsearch/v1?key={api_key}&cx={search_engine_id}&q={query}&num={num_results}"
8     response = requests.get(url)
9
10    if response.status_code != 200:
11        print(f"Error: {response.status_code} - {response.text}")
12        return []
13
14    # Parse the JSON response and extract the top results
15    results = response.json().get("items", [])
16    urls = [result.get("link") for result in results]
17    titles = [result.get("title") for result in results]
18
19    # Scrape the content of the top URLs
20    scraped_content = []
21    for url in urls:
22        page = requests.get(url)
23        soup = BeautifulSoup(page.content, "html.parser")
24        paragraphs = soup.find_all("p")
25        content = " ".join([p.get_text() for p in paragraphs])
26        scraped_content.append(content)
27
28    # Format the results
29    results = [{"url": url, "title": title, "content": content} for url, title, content in zip(urls, titles, scraped_content)]
30
31    # Construct the prompt for the LLM
32    prompt = f"Answer the following question based on the information provided in the URLs above.\n\n{query}\n\n"
33    for result in results:
34        prompt += f"\nURL: {result['url']}\nTitle: {result['title']}\nContent: {result['content']}\n\n"
35
36    # Call the LLM to generate the answer
37    response = openai.Completion.create(
38        engine="gpt-4o-mini",
39        prompt=prompt,
40        max_tokens=150,
41        temperature=0.5
42    )
43
44    # Return the final answer
45    return response.choices[0].text
```

```

13
14     data = response.json()
15     results = []
16
17     if "items" in data:
18         print(f"\nSearch Results for: '{query}' (Top {num_results})")
19         print("=" * 60)
20
21     for i, item in enumerate(data["items"], start=1):
22         title = item.get("title", "No Title")
23         link = item.get("link", "No Link")
24
25         results.append(link) # Only store the links for scraping
26
27         # Display retrieved links
28         print(f"**Result {i}:** {title}")
29         print(f"{link}")
30         print("-" * 60)
31
32     return results
33
34 # Function to scrape full text from a webpage
35 def scrape_website(url):
36     try:
37         response = requests.get(url, headers={"User-Agent": "Mozilla/5.0"}, timeout=10)
38         soup = BeautifulSoup(response.text, "html.parser")
39
40         # Extract paragraphs
41         paragraphs = soup.find_all("p")
42         content = "\n".join([p.get_text() for p in paragraphs[:15]]) # Extract
43         first 15 paragraphs
44
45         return content if content else "No content extracted."
46     except Exception as e:
47         return f"Failed to scrape {url}: {e}"
48
49 # Function to generate an answer using OpenAI API
50 def generate_answer(prompt, llm_api_key):
51     client = openai.OpenAI(api_key=llm_api_key)
52     response = client.chat.completions.create(
53         model="gpt-4o-mini",
54         messages=[{"role": "system", "content": "You are a helpful AI assistant."}
55         ],
56         {"role": "user", "content": prompt}
57     )
58     return response.choices[0].message.content
59
60 # Main function that integrates Google Search, Web Scraping, and LLM Answering

```

```

59 def answer_from_google(query, api_key, search_engine_id, llm_api_key, num_results
60     =3):
61     urls = search_google(query, api_key, search_engine_id, num_results)
62
63     extracted_content = []
64     for url in urls:
65         content = scrape_website(url)
66         extracted_content.append(f"\n--- Source: {url} ---\n{content}")
67
68     # Combine all extracted content for LLM input
69     full_text = "\n\n".join(extracted_content)
70
71     prompt = f"Summarize the following content and answer the question: {query}\n\n{full_text}"
72
73     return generate_answer(prompt, llm_api_key)
74
75 # Example usage:
76 API_KEY = SERPAPI_KEY # Correct Google API Key
77 SEARCH_ENGINE_ID = search_engine_id # Custom Search Engine ID
78
79 query = "Tell me about EUMETNET Artificial Intelligence Programme"
80 answer = answer_from_google(query, API_KEY, SEARCH_ENGINE_ID, LL_M_API_KEY,
81     num_results=3)

```

display

```

1 from IPython.display import display, Markdown
2 display(Markdown(answer))

```

Sample Output:

The EUMETNET Artificial Intelligence (AI) Programme, launched in January 2024, is a collaborative initiative that brings together 16 National Meteorological and Hydrological Services (NMHSs) within Europe, along with ECMWF (European Centre for Medium-Range Weather Forecasts) and EUMETSAT (European Organisation for the Exploitation of Meteorological Satellites). Managed by Prof. Dr. Roland Potthast from the German National Meteorological Service (DWD), the program spans five years and is structured into three main components focusing on Weather, Climate, and Environmental Applications.

Key components of the program include:

Data Curation and Products: Leveraging satellite data and meteorological observations.
Analysis, Modelling, and Ensembles: Involving extensive modeling and forecasting initiatives.

Communication and Training: This includes a series of workshops aimed at fostering collaboration among meteorological services, with the first workshop scheduled for June 5-6, 2024.

The program is designed to enhance AI applications in meteorology by exploring innovative techniques that allow for enhanced weather predictions, particularly for complex events, through the rapid analysis of vast amounts of meteorological data. AI and Machine

Learning (ML) are seen as tools for creating more accurate and personalized weather forecasts while also enabling advanced simulations for climate modeling.

EUMETSAT's involvement particularly focuses on the integration of AI techniques to improve the usage of satellite data, as well as contribute to a culture of openness and collaboration within the meteorological community.

Summary:

EUMETNET's AI Programme aims to innovate meteorological applications through collaboration among NMHSs in Europe, leveraging AI and ML for improved weather predictions and analysis. The program focuses on data curation, modeling, and training through workshops to enhance productivity and accuracy in weather services.

Chapter 8

Multimodal LLMs

8.1 Fundamentals of Multimodal Large Language Models

Multimodal Large Language Models (MLLMs) extend traditional LLMs by incorporating multiple data modalities, such as text, images, audio, and video, enabling more comprehensive reasoning and interaction with diverse data sources.

Fine-Tuning a Transformer Model for Coastal Weather Forecasting

Introduction

Our introductory example describes the implementation of a fine-tuned Transformer model for processing wind field data over the North Sea. The model is based on a pretrained T5 architecture and is adapted to generate textual descriptions of wind conditions from numerical wind field inputs. The main components include synthetic wind field generation, visualization, dataset creation, model training, and evaluation.

Device Setup and Dependencies

The implementation begins with setting up the necessary libraries, including torch for deep learning, transformers for handling the T5 model, and cartopy for geospatial visualization. The computation is performed on a *CUDA* device if available:

```
Device Setup

1 import torch
2
3 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
4 print(device)
```

Generating Synthetic Wind Fields

Wind field data is generated using a grid-based approach. A base wind direction is selected, and random variations of ± 10 degrees are applied at each grid point to simulate realistic wind

fluctuations. The wind speed follows a controlled distribution:

Generate Synthetic Wind Fields

```

1 def generate_wind_field(grid_size=10, base_wind_dir=315, wind_speed=None):
2     lon = np.linspace(5, 10, grid_size)
3     lat = np.linspace(53, 56, grid_size)
4     LON, LAT = np.meshgrid(lon, lat)
5
6     theta_variation = np.random.uniform(-10, 10, size=(grid_size, grid_size))
7     theta = np.deg2rad(270 - (base_wind_dir + theta_variation))
8
9     U = wind_speed * np.cos(theta)
10    V = wind_speed * np.sin(theta)
11
12    return LON, LAT, U, V, wind_speed

```

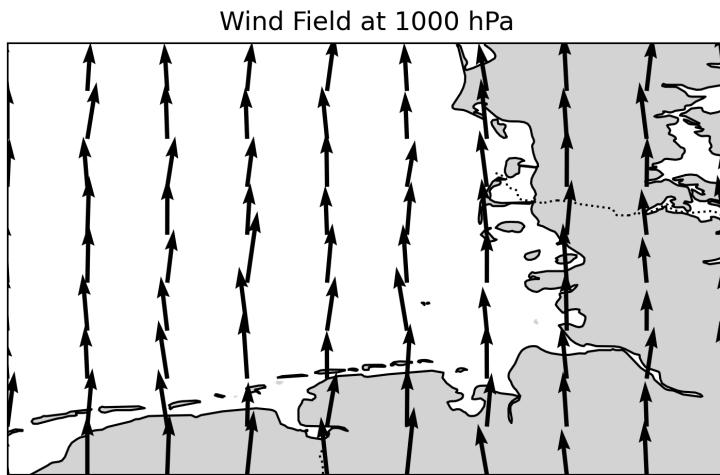


Figure 8.1: Coastal Wind with example text description: "Strong northerly winds with speeds around 15.0 m/s over the North Sea".

Visualizing Wind Fields

Wind fields are displayed using Cartopy, showing vectors representing wind direction and magnitude. The following function renders the wind data on a map projection:

Plot Wind Fields

```

1 def plot_wind_field(LON, LAT, U, V, title="Wind Field at 1000 hPa"):
2     fig, ax = plt.subplots(figsize=(6, 4), \
3         subplot_kw={'projection': ccrs.PlateCarree()})
4     ax.set_extent([5, 10, 53, 56], crs=ccrs.PlateCarree())
5     ax.add_feature(cfeature.COASTLINE)
6     ax.add_feature(cfeature.BORDERS, linestyle=':')
7     ax.add_feature(cfeature.LAND, facecolor='lightgray')
8     ax.quiver(LON, LAT, U, V, scale=200, transform=ccrs.PlateCarree())

```

```
9     ax.set_title(title)
10    plt.show()
```

Generating Textual Descriptions

The model converts wind field data into descriptive text by categorizing wind intensity and associating wind direction with predefined labels. Wind speed values are rounded to predefined levels:

Generate Text Descriptions

```
1 def generate_text_description(wind_speed, wind_dir):
2     intensity = "strong" if np.mean(wind_speed) > 12 else "moderate" if np.mean(
3         wind_speed) > 6 else "light"
4     directions = {0: "northerly", 45: "northeasterly", 90: "easterly", 135: "
5         southeasterly",
6             180: "southerly", 225: "southwesterly", 270: "westerly", 315: "
7             northwesterly"}
8     direction = directions.get(round(wind_dir), "variable")
9     approx_speed = [0, 1, 2, 5, 10, 15, 20, 25, 30][np.argmin(np.abs([0, 1, 2, 5,
10, 15, 20, 25, 30] - np.mean(wind_speed)))]
10    return f"{intensity.capitalize()} {direction} winds with speeds around {
11        approx_speed} m/s over the North Sea."
```

Training the Transformer Model

The fine-tuning process uses a pretrained *T5-small* model. Wind fields are encoded as structured text, and corresponding descriptions are used as target outputs. The model is trained using an Adam optimizer with a learning rate of 5×10^{-5} :

Train Transformer Model

```
1 def train_transformer(text_samples, wind_fields, num_epochs=10):
2     tokenizer = T5Tokenizer.from_pretrained("t5-small")
3     model = T5ForConditionalGeneration.from_pretrained("t5-small").to(device)
4     optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
5
6     loss_history = []
7     start_time = time.time()
8
9     for epoch in range(num_epochs):
10        epoch_start = time.time()
11        total_loss = 0
12
13        for wind, text in zip(wind_fields, text_samples):
14            U, V, wind_speed = wind
15            wind_input = f"Wind field: U: {' '.join(map(str, np.round(U.flatten()[:20], 2)))}; V: {' '.join(map(str, np.round(V.flatten()[:20], 2)))}; Speed: {' '.join(map(str, np.round(wind_speed.flatten()[:20], 2)))}."
16            input_ids = tokenizer.encode(wind_input, return_tensors="pt",
17                padding=True)
```

```

        truncation=True, max_length=512).to(device)
17     labels = tokenizer.encode(text, return_tensors="pt", truncation=True,
18                               max_length=512).to(device)
19     outputs = model(input_ids=input_ids, labels=labels)
20     loss = outputs.loss
21     optimizer.zero_grad()
22     loss.backward()
23     optimizer.step()
24     total_loss += loss.item()
25
26     avg_loss = total_loss / len(text_samples)
27     loss_history.append(avg_loss)
28     print(f"Epoch {epoch+1}/{num_epochs} | Loss: {avg_loss:.4f}")
29
30     return model, tokenizer, loss_history

```

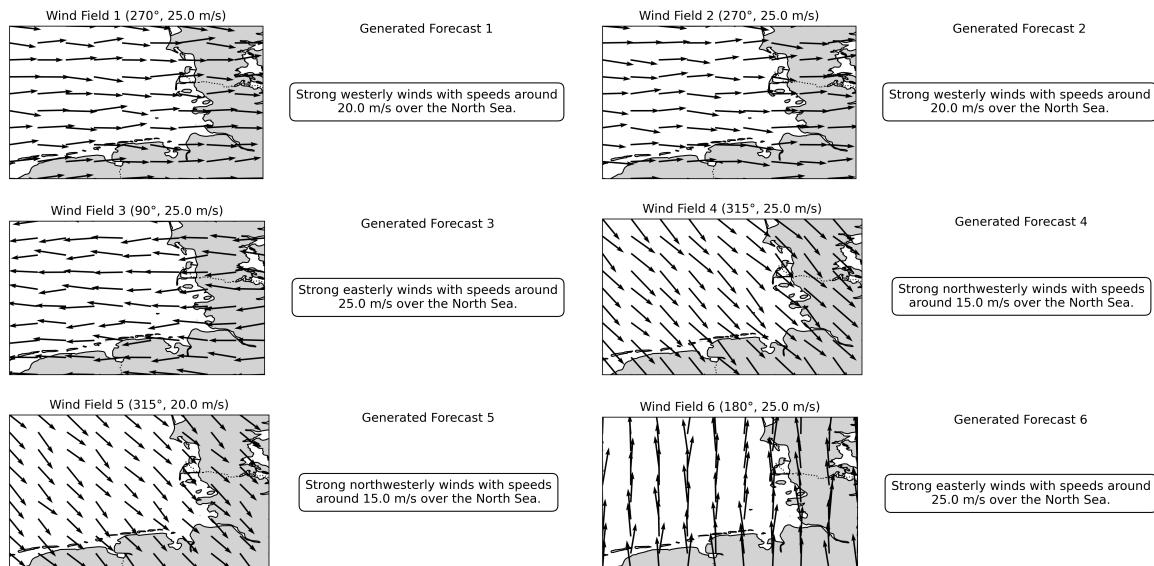


Figure 8.2: Forecast of Coastal Weather at Different Time Steps

Generating Forecasts

Once trained, the model can generate textual descriptions from new wind fields:

Generate Forecasts

```

1 def generate_forecast(model, tokenizer, wind_field):
2     U, V = wind_field
3     wind_input = f"Wind field: U: {' '.join(map(str, np.round(U.flatten()[:20], 2))}); V: {' '.join(map(str, np.round(V.flatten()[:20], 2)))}"
4     input_ids = tokenizer.encode(wind_input, return_tensors="pt", truncation=True,
5                                   max_length=512).to(device)
6     output = model.generate(input_ids)
7     return tokenizer.decode(output[0], skip_special_tokens=True)

```

8.2 Radar Data Access and AI Interpretation

This section documents the steps taken in the Jupyter notebook to access, process, visualize, and interpret radar reflectivity data from the German Weather Service (DWD), using AI-based image analysis via the OpenAI API. The steps are implemented in Python and follow a modular structure.

8.2.1 1. Downloading the Radar Composite

Radar composite data in HDF5 format was obtained from the DWD Open Data server. The file was selected from the /weather/radar/composite/hx/ directory, which typically contains reflectivity products. The latest file was identified and downloaded automatically.

```
python

1 import requests
2 from bs4 import BeautifulSoup
3 from urllib.parse import urljoin
4
5 base_url = "https://opendata.dwd.de/weather/radar/composite/hx/"
6 response = requests.get(base_url)
7 soup = BeautifulSoup(response.text, "html.parser")
8
9 hd5_files = sorted([
10     a.get("href") for a in soup.find_all("a")
11     if "composite_hx" in a.get("href", "") and "-hd5" in a.get("href", "")
12 ])
13
14 if hd5_files:
15     latest_file = hd5_files[-1]
16     download_url = urljoin(base_url, latest_file)
17     with open("radar.h5", "wb") as f:
18         f.write(requests.get(download_url).content)
```

8.2.2 2. Reading and Decoding Reflectivity Data

The reflectivity field was read from the dataset1/data1/data group in the HDF5 file. The physical reflectivity values (in dBZ) were reconstructed using gain and offset parameters stored in the metadata.

```
python

1 import h5py
2 import numpy as np
3
4 with h5py.File("radar.h5", "r") as f:
5     raw = f["dataset1/data1/data"][:]
6     what = f["dataset1/data1/what"]
```

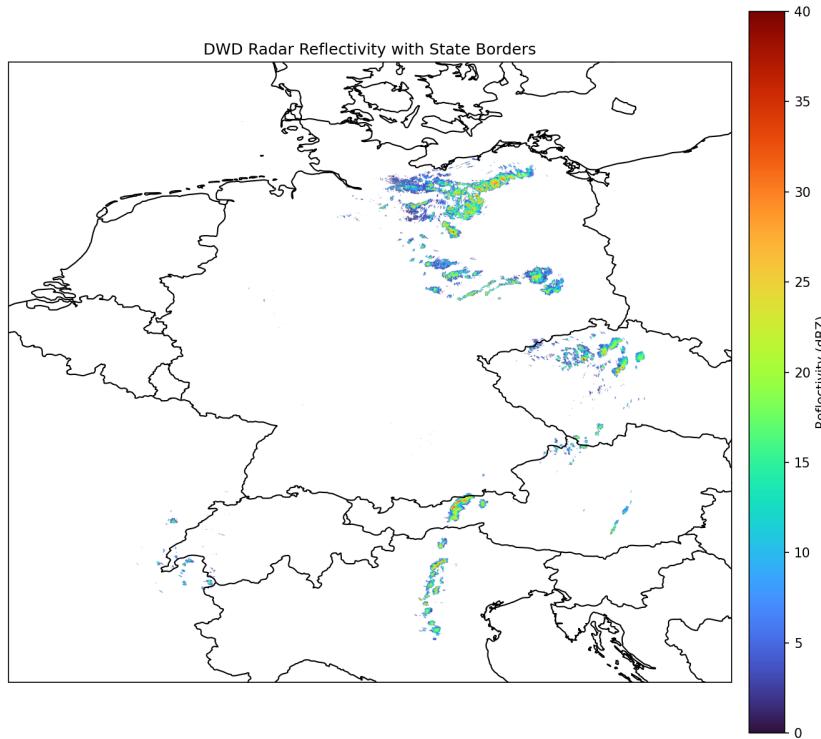


Figure 8.3: Decoded radar reflectivity over Germany with state borders. White regions indicate no signal or missing data.

```

7     gain = what.attrs.get("gain", 1.0)
8     offset = what.attrs.get("offset", 0.0)
9     reflectivity = raw.astype(np.float32) * gain + offset
10    reflectivity[(raw == 0) | (raw == 65535) | (reflectivity < 0)] = np.nan

```

8.2.3 3. Plotting the Radar Composite

The radar reflectivity is visualized on a map using Cartopy. The extent was set to cover Germany and surrounding countries. Missing values were shown in white to emphasize actual radar returns.

`python`

```

1 import matplotlib.pyplot as plt
2 import cartopy.crs as ccrs
3 import cartopy.feature as cfeature
4 from matplotlib import colormaps

```

```

5
6 cmap = colormaps.get_cmap("turbo").copy()
7 cmap.set_bad(color='white')
8
9 extent = [3.0, 17.0, 44.0, 56.0]
10 masked = np.ma.masked_invalid(reflectivity)
11
12 plt.figure(figsize=(12, 10))
13 ax = plt.axes(projection=ccrs.PlateCarree())
14 ax.set_extent(extent)
15 im = ax.imshow(masked, extent=extent, cmap=cmap, vmin=0, vmax=40,
16                 origin='lower', transform=ccrs.PlateCarree())
17
18 ax.add_feature(cfeature.BORDERS)
19 ax.coastlines(resolution='10m')
20 cbar = plt.colorbar(im, ax=ax)
21 cbar.set_label("Reflectivity (dBZ)")
22 plt.title("DWD Radar Reflectivity with State Borders")
23 plt.savefig("radar_map_germany.png", dpi=150)
24 plt.show()

```

8.2.4 4. Interpretation Using OpenAI GPT-4 Vision

To generate a natural-language interpretation of the radar image, the processed PNG was base64-encoded and sent to the OpenAI API using the GPT-4-Turbo model with vision capabilities.

- The request includes both a text prompt and the radar image.
- The result is a textual interpretation of reflectivity patterns, estimated intensity, and structure classification (e.g. stratiform or convective).

python

```

1 from openai import OpenAI
2 from dotenv import load_dotenv
3 import base64, os
4 from IPython.display import Markdown, display
5
6 load_dotenv()
7 client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
8
9 with open("radar_map_germany.png", "rb") as img:
10     encoded = base64.b64encode(img.read()).decode("utf-8")
11
12 response = client.chat.completions.create(
13     model="gpt-4-turbo",
14     messages=[
15         {"role": "user", "content": [

```

```

16         {"type": "text", "text": "Interpret this radar reflectivity image from
17             Germany. Describe precipitation areas, intensity, and structure."},
18         {"type": "image_url", "image_url": {
19             "url": f"data:image/png;base64,{encoded}"}
20         }
21     ],
22     max_tokens=800,
23 )
24
25 interpretation = response.choices[0].message.content
26 display(Markdown(interpretation))

```

Here is the outcome of the image interpretations, which OpenAI provides.

This image of radar reflectivity from Germany depicts various levels of precipitation intensity across different regions, as indicated by the color scale on the right. The color scale ranges from 0 dBZ, representing no precipitation, to 40 dBZ, indicating heavy precipitation.

In this image:

Northwestern Germany: There appears to be no significant precipitation as the colors are in the lower range of the scale (0-5 dBZ), which is indicative of clear or very light precipitation conditions.

Northeastern Germany: Similar to the northwest, this region also shows minimal reflectivity values, indicating little to no precipitation.

Central and Southern Germany: These areas also display minimal radar reflectivity with dBZ values primarily in the range of 0-5 dBZ. There seem to be no significant precipitation events occurring in these regions at the time this image was captured.

Western and Southwestern Germany: These regions are predominantly clear, with a few areas perhaps having very light precipitation as indicated by slightly higher, but still minimal dBZ values.

There are no distinct areas of high dBZ values (e.g., >20 dBZ) that would suggest moderate to heavy rainfall or convective activity (such as thunderstorms) anywhere on the map. Therefore, the image overall does not show any signs of significant convective structures such as thunderstorm cells, which would typically be indicated by localized, high-intensity dBZ readings.

All in all, the weather across Germany during the period represented by this reflectivity image seems to be largely calm and free of significant precipitation events, with only scattered, very light precipitation or clear conditions throughout. The absence of any high dBZ values indicates an absence of strong convective activities like thunderstorms, which are typically characterized by sudden, intense rainfall indicated by higher dBZ values.

We note that the interpretation will be billed by standard platforms as shown in the following figure.

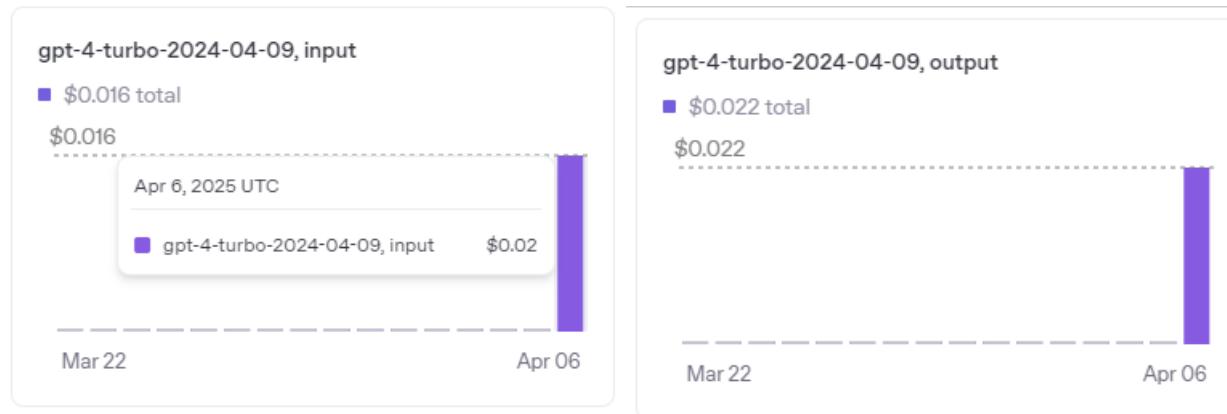


Figure 8.4: Input and Output for image interpretation will cost, here about 2-3 cent per image (in and out summed).

8.3 Cloud Top Height as a Multimodal AI Application

Cloud Top Height (CTH) is a satellite-derived parameter that estimates the altitude of the upper boundary of cloud systems, typically expressed in meters above sea level. It is primarily derived from thermal infrared satellite observations (e.g., from Meteosat SEVIRI), which allow cloud top temperature to be estimated and then converted to height using vertical atmospheric profiles.

High CTH values are generally associated with deep convective systems such as cumulonimbus clouds, while low CTH values are often indicative of stratiform or shallow cloud layers. As such, CTH provides valuable insight into atmospheric structure and storm development, especially in the absence of direct vertical sounding data.

In the context of multimodal AI, CTH maps are well suited for image-text applications, where visual patterns are interpreted in conjunction with meteorological knowledge. Below, we outline several possible use cases for applying multimodal models (e.g., GPT-4 with Vision or Gemini) to CTH data.

8.3.1 Multimodal Use Cases for Cloud Top Height Interpretation

- **CTH Map Interpretation**

Given a single satellite-derived CTH image, a multimodal model can identify regions of high cloud tops (e.g., >10 km), associate them with potential deep convection, and distinguish between different cloud layers. This is useful for nowcasting and synoptic analysis.

- **CTH and Radar Reflectivity Comparison**

A side-by-side analysis of CTH and radar reflectivity fields enables the model to assess where tall clouds are associated with precipitation. This helps in identifying convective cores or in evaluating false alarms, such as high cloud tops without significant rainfall.

- **CTH Overlay with Numerical Weather Prediction (NWP)**

By comparing observed CTH fields with model-predicted convective zones, the AI can evaluate the accuracy of model forecasts, detect missed storms, or highlight overestimates of vertical development. This can support model diagnostics and validation.

- **CTH Threshold-Based Alerting**

AI systems can be tasked with scanning a CTH image and identifying areas exceeding certain height thresholds (e.g., 10,000 m). These regions may be relevant for aviation warnings, thunderstorm alerts, or convective risk assessments.

- **Time-Series or Animation Analysis**

Using a sequence of CTH images, a multimodal model could track the temporal evolution of convective systems. It can describe cloud growth, merging, or dissipation — similar to what a human forecaster might do when watching satellite loops.

- **Natural Language Bulletins from CTH Maps**

The model can automatically generate synoptic summaries or weather briefings based solely on the CTH structure, using meteorological language. This supports automation in forecast generation and situational awareness.

These examples demonstrate the broad potential of combining satellite-derived cloud structure with large multimodal models to extract high-level meteorological insights in a human-readable form.

8.3.2 CTH Map Interpretation

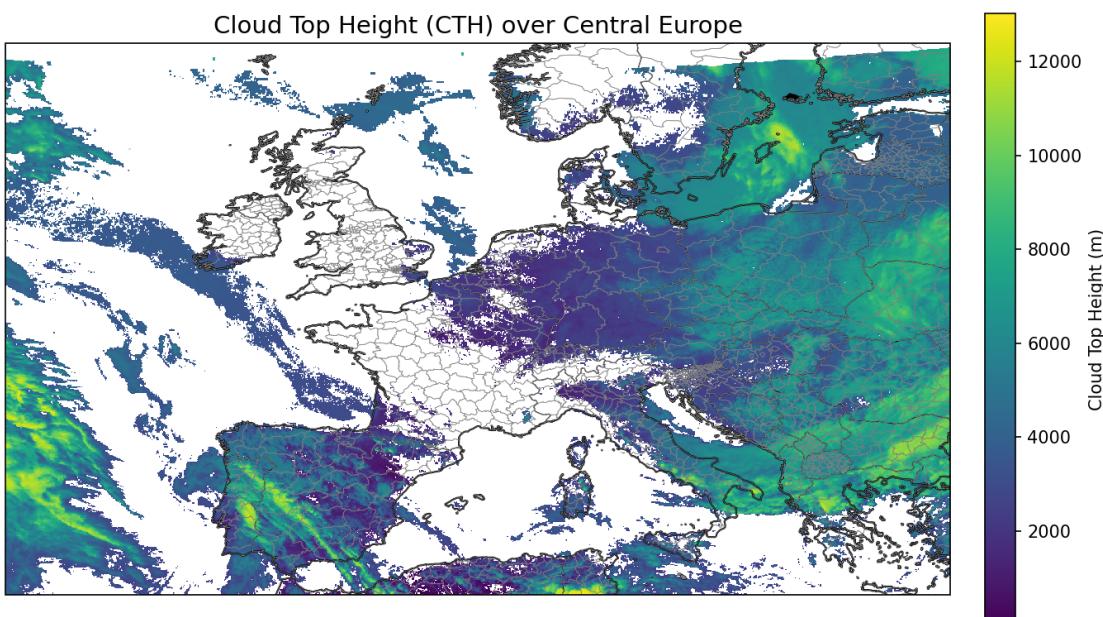


Figure 8.5: Cloud Top Height (CTH) over Central Europe derived from satellite data. Higher cloud tops (green/yellow) are indicative of deep convection; lower tops (purple) represent stratiform or less active cloud fields.

To interpret the satellite-derived CTH image, the notebook performs the following steps:

1. Download the Latest CTH File We start by accessing the latest CTH file from the DWD Open Data server.

python

```

1 import requests
2 from bs4 import BeautifulSoup
3 from urllib.parse import urljoin
4
5 base_url = "https://opendata.dwd.de/weather/satellite/clouds/CTH/"
6 response = requests.get(base_url)
7 soup = BeautifulSoup(response.text, "html.parser")
8
9 cth_files = sorted([
10     link.get("href") for link in soup.find_all("a")
11     if link.get("href", "").endswith(".nc.bz2")
12 ])
13
14 latest_file = cth_files[-1]
15 download_url = urljoin(base_url, latest_file)
16
17 with open("cth_latest.nc.bz2", "wb") as f:
18     f.write(requests.get(download_url).content)

```

2. Decompress and Read the Data The ‘.bz2‘ archive is unpacked, and the cloud top height data is read from the NetCDF file.

python

```

1 import bz2
2 import netCDF4 as nc
3
4 with bz2.BZ2File("cth_latest.nc.bz2") as bz2file:
5     with open("cth_latest.nc", "wb") as ncfile:
6         ncfile.write(bz2file.read())
7
8 ds = nc.Dataset("cth_latest.nc")
9 cth = ds.variables["CTH"][:, :, :]
10 lat = ds.variables["lat"][:]
11 lon = ds.variables["lon"][:]

```

3. Visualize the CTH Field A simple pseudocolor map is created using Matplotlib, where high cloud tops are shown in brighter colors.

python

```

1 import numpy as np
2 import matplotlib.pyplot as plt

```

```

3
4 cth = np.ma.masked_where(cth <= 0, cth)
5
6 plt.figure(figsize=(10, 8))
7 plt.pcolormesh(lon, lat, cth, cmap="viridis", shading="auto")
8 plt.colorbar(label="Cloud Top Height (m)")
9 plt.title("Cloud Top Height (latest observation)")
10 plt.xlabel("Longitude")
11 plt.ylabel("Latitude")
12 plt.grid(True)
13 plt.savefig("cth_map.png", dpi=150)
14 plt.show()

```

4. AI-Based Interpretation with OpenAI Vision. The image is encoded and sent to OpenAI's multimodal model through its platform API for meteorological interpretation.

python

```

from openai import OpenAI
from dotenv import load_dotenv
import os
import base64
from IPython.display import Markdown, display

load_dotenv()
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

with open("cth_map.png", "rb") as image_file:
    base64_image = base64.b64encode(image_file.read()).decode("utf-8")

response = client.chat.completions.create(
    model="gpt-4-turbo",
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": (
                        "This is a satellite-derived Cloud Top Height (CTH) image over Europe.\n"
                        "Please interpret the structure shown in the image: \n"
                        "- Identify regions of high or low cloud tops.\n"
                        "- Estimate where deep convection may be present.\n"
                        "- Describe what synoptic or convective features are visible.\n"
                        "- Provide a summary of the possible weather situation."
                    )
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"data:image/png;base64,{base64_image}"
                    }
                }
            ]
        ],
        max_tokens=800,
    )
)

```

```
interpretation = response.choices[0].message.content
display(Markdown(interpretation))
```

Here is the outcome of the image interpretations, which our script based on the OpenAI API provides.

Identification of Regions of High and Low Cloud Tops:

High Cloud Tops:

Central Northern Europe, especially in the yellow-colored area over central Sweden and parts of Norway, displays very high cloud tops that exceed 10,000 meters. Some high cloud tops are also noticeable over the sea directly west of the Iberian peninsula.

Low Cloud Tops:

Major parts of the European mainland have lower cloud tops, particularly noticeable in France, Germany, and extending eastward across Poland into parts of Russia (indicated by purple to blue colors), where cloud tops range from 2000 to 6000 meters.

Potential Areas of Deep Convection:

The high cloud tops observed particularly in central parts of Scandinavia (central Sweden and nearby areas) are potential indicators of deep convective activity. Typically, cloud tops reaching or exceeding heights around 10,000 meters suggest the presence of cumulonimbus clouds, which are associated with heavy rainfall, thunderstorms, and potential severe weather events.

Visible Synoptic or Convective Features:

An intense convective system appears to be over Central Scandinavia, supported by the very high cloud tops as indicated by the yellow coloration. This system may suggest the development or presence of a strong weather front, potentially a cold front given its structure and scale.

Widespread lower cloud tops across much of Western and Eastern Europe might be associated with stable weather patterns or widespread stratiform clouds usually seen with settled conditions or behind a front.

Possible Summary of the Weather Situation:

The presence of high, deep convective cloud tops over Central Scandinavia suggests ongoing or developing strong weather activities, likely associated with significant rainfall, thunderstorms, or possibly snow if temperatures are low enough.

In contrast, much of the rest of the Central European region, characterized by lower cloud tops, might be experiencing more stable and milder weather. This could manifest as cloudy but largely dry conditions, potentially following the passage of a weather front.

The contrasting cloud top heights from the west to the east may indicate a strong weather gradient, potentially impacting weather conditions rapidly over short distances in the region.

In conclusion, the satellite-derived CTH image suggests significant weather activity, particularly over Scandinavia, with potential impacts including precipitation and more pronounced weather events, while a quieter weather regime may prevail over large parts of Western and Central Europe.

Chapter 9

Diffusion and flexible Graph Networks

9.1 Diffusion Networks

9.1.1 Introduction

Diffusion models represent a class of generative neural networks that have recently achieved remarkable success in areas such as image generation (e.g., DALL·E 2, Stable Diffusion) and molecular design. Their core principle is inspired by physical diffusion processes: transforming data into noise and learning to reverse this process. In this way, diffusion models learn to generate new data samples from pure noise through a learned denoising process.

In the context of weather and climate, diffusion models offer a new paradigm for generative modeling, ensemble forecasting, and uncertainty quantification. Traditional numerical weather prediction (NWP) relies on the integration of physical models forward in time, often facing challenges in representing uncertainty and producing diverse ensemble members. Diffusion models can be used as a complementary or alternative strategy: they learn the statistical structure of weather or climate fields and generate samples that respect these statistics, conditioned on given observations or coarse-resolution forecasts.

This section introduces the core ideas and mathematical framework of diffusion models, with a focus on applications to geophysical data.

9.1.2 Forward and Reverse Diffusion Processes

The Forward Process (Adding Noise)

Let $\mathbf{x}_0 \in \mathbb{R}^d$ be a sample from the true data distribution (e.g., a temperature field, wind vector, or other atmospheric variable on a spatial grid). The forward diffusion process corrupts \mathbf{x}_0 over T discrete time steps by gradually adding Gaussian noise, producing a sequence of increasingly noisy samples $\mathbf{x}_1, \dots, \mathbf{x}_T$.

This process is modeled as a Markov chain:

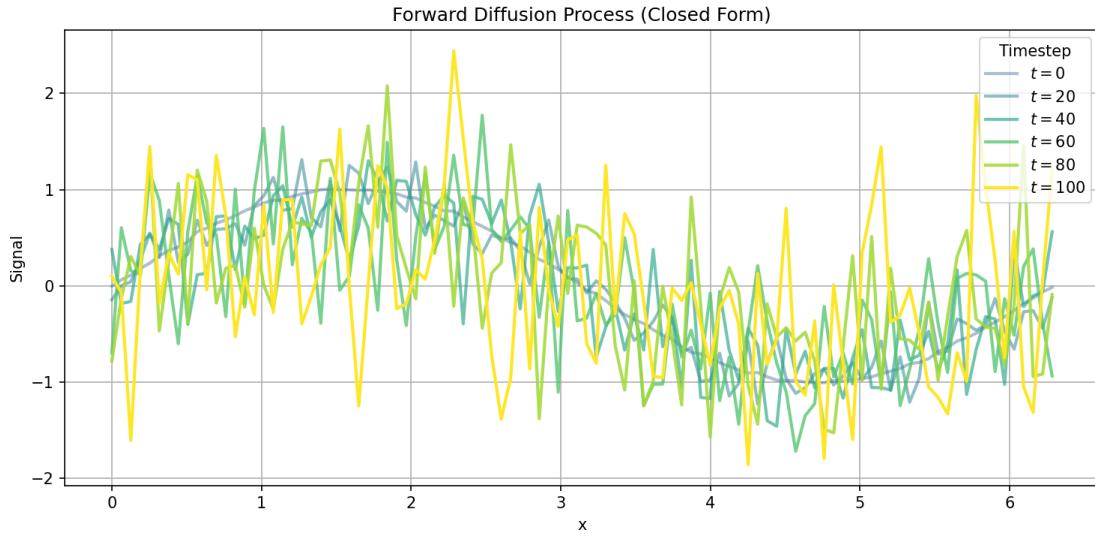


Figure 9.1: Forward diffusion process applied to a sine function over 101 steps. Every 20th step is shown with increasing transparency and brightness. Initially, the signal is clean ($t = 0$), and as the diffusion progresses, Gaussian noise increasingly dominates the structure. The process is governed by a predefined noise schedule using a cumulative variance parameter $\bar{\alpha}_t$, ensuring that \mathbf{x}_t converges toward a standard Gaussian distribution as $t \rightarrow T$.

$$q(\mathbf{x}_1, \dots, \mathbf{x}_T | \mathbf{x}_0) = \prod_{t=1}^T q(\mathbf{x}_t | \mathbf{x}_{t-1}),$$

where each transition adds noise as:

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I}).$$

Here, each component in the Gaussian transition

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I}\right)$$

has the following role and dimensional interpretation:

- $\mathbf{x}_t \in \mathbb{R}^d$ is the noisy variable at step t , drawn from the distribution.
- The mean of the Gaussian is $\mu_t = \sqrt{1 - \beta_t} \mathbf{x}_{t-1} \in \mathbb{R}^d$, which scales the previous state downward. This reduces the signal amplitude as noise is added, ensuring the data gradually diffuses toward a Gaussian distribution.
- The covariance is $\Sigma_t = \beta_t \mathbf{I} \in \mathbb{R}^{d \times d}$, where \mathbf{I} is the identity matrix. This defines isotropic Gaussian noise with variance β_t in each dimension.
- $\beta_t \in (0, 1)$ is a predefined noise schedule that increases over time. Small values at early steps retain more structure; larger values at later steps ensure full noise.

As $t \rightarrow T$, the distribution of \mathbf{x}_t converges to the standard normal distribution $\mathcal{N}(0, \mathbf{I})$, effectively transforming structured data into pure noise.

As $t \rightarrow T$, the sample \mathbf{x}_T approaches pure Gaussian noise: $\mathcal{N}(0, \mathbf{I})$. This process is fixed and does not involve learning. It provides synthetic training pairs $(\mathbf{x}_t, t, \mathbf{x}_0)$ to train the reverse model.

The Reverse Process (Learning to Denoise)

The generative aspect of diffusion models lies in the reverse process, which seeks to map noise back to data. This process is defined by:

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^T p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t),$$

where:

- $p(\mathbf{x}_T) = \mathcal{N}(0, \mathbf{I})$ is the assumed prior distribution (pure noise),
- $p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t)$ is the learned denoising distribution parameterized by a neural network.

The network predicts a mean $\mu_\theta(\mathbf{x}_t, t)$ and often assumes a fixed variance Σ_t , resulting in:

$$p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_t).$$

The denoising network can be conditioned on additional inputs such as class labels, spatial information, or physical constraints, making it suitable for structured data like weather fields.

9.1.3 Loss Function and Training

In the denoising framework used here, the neural network is trained to predict a less noisy version of the input, i.e., an estimate of \mathbf{x}_{t-1} given the current noisy state \mathbf{x}_t and conditioning information (e.g., class or weather constraints).

At each training step, we use a sequence of noisy inputs generated from a known clean signal \mathbf{x}_0 , perturbed by Gaussian noise through the forward process. The training loss at each denoising step is given by the mean squared error:

$$\mathcal{L}_t(\theta) = \mathbb{E}_{\mathbf{x}_t, \mathbf{x}_{t-1}} [\|\mathbf{x}_{t-1} - f_\theta(\mathbf{x}_t, t, \text{cond})\|^2],$$

where:

- $\mathbf{x}_t \in \mathbb{R}^d$ is the noisy state at time t ,
- $\mathbf{x}_{t-1} \in \mathbb{R}^d$ is the target (previous, less noisy) state,
- $f_\theta(\cdot, t, \text{cond})$ is the neural network that predicts the denoised state, possibly conditioned on class labels or physical context,

- The expectation is taken over randomly sampled timesteps and training examples.

The total loss is averaged over all reverse steps from $t = T$ to 1. This approach turns denoising into a stepwise regression problem toward the clean signal trajectory, rather than estimating the noise directly as in DDPM.

9.1.4 Relevance to Weather and Climate

In weather and climate science, data is high-dimensional, often spatial-temporal, and exhibits strong structure (e.g., physical constraints, correlations, seasonal cycles). Diffusion models are particularly suited to such domains because they:

- Learn **nonlinear distributions** of full fields (e.g., 2m temperature, geopotential height),
- Allow **conditional generation**, e.g., generating high-resolution forecasts conditioned on coarse-resolution inputs,
- Support **stochastic generation**, producing diverse ensemble members,
- Can be guided or constrained by physical priors or observations.

Recent applications include:

- Generating ensemble weather forecast members,
- Downscaling global climate model (GCM) outputs,
- Imputing missing observations (gap-filling),
- Emulating expensive simulation models.

By learning to sample plausible geophysical states from noise while respecting physical structure, diffusion models represent a promising approach for next-generation generative systems in atmospheric science.

9.1.5 A Minimal Diffusion Denoising Example

To demonstrate the principle of diffusion models in a simple and interpretable setting, we construct a minimal 1D example: starting from white noise, a neural network learns to iteratively denoise it toward a target sine curve. We use PyTorch to define and train a small denoising network that learns this reconstruction process in a class-conditional setup.

Step 1: Configuration and Target Signal. We define the sine wave classes, select one target curve, and generate the corresponding label as a one-hot vector.

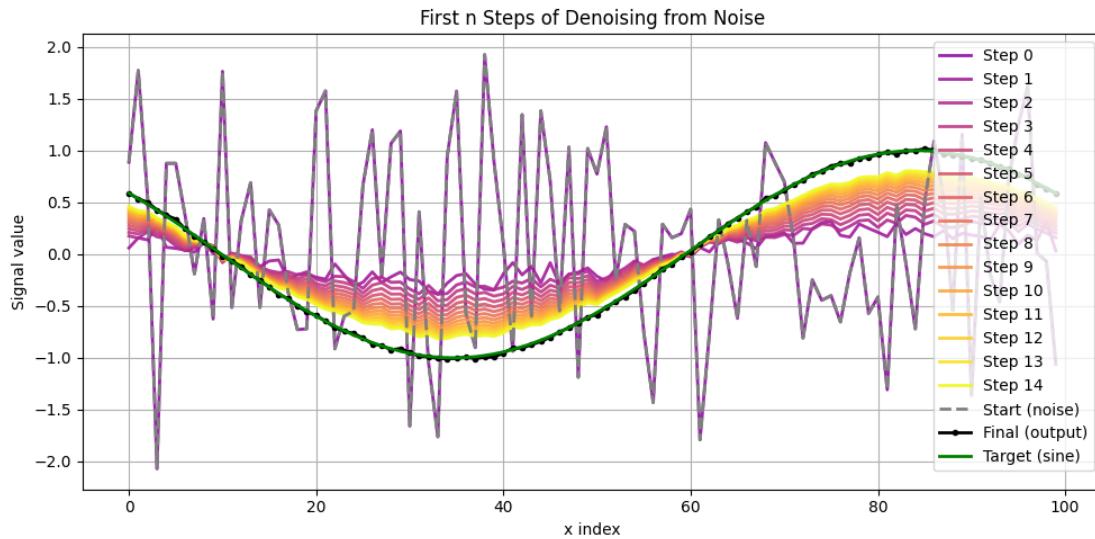


Figure 9.2: The first 15 steps of the denoising process, starting from white noise (dashed gray). Colored curves represent the intermediate states. The model quickly adjusts toward the structure of the target sine (green), with the final output (black dots) nearly matching it.

Setup Example for Diffusion Network

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import matplotlib.pyplot as plt
5
6 # --- Config ---
7 n_points = 100
8 n_steps = 50
9 noise_std = 0.2
10 class_idx = 2 # Select sine class 0 to 4
11
12 # --- Generate sine waves ---
13 x = torch.linspace(0, 2 * torch.pi, n_points)
14 phases = torch.arange(5) * (2 * torch.pi / 5)
15 sine_set = torch.stack([torch.sin(x + p) for p in phases])
16 x_target = sine_set[class_idx].unsqueeze(0) # shape [1, n_points]
17 label = torch.nn.functional.one_hot(torch.tensor([class_idx]), num_classes=5).
  float()

```

This sets up our clean signal x_0 and class information to be used throughout training and inference.

Step 2: Define the Denoising Network. We now define a small MLP that takes as input the noisy signal and the class label (concatenated) and outputs a denoised signal of the same shape.

Network for Diffusion Inversion

```

1 # --- Model ---
2 class DenoiseMLP(nn.Module):
3     def __init__(self, n_points, n_classes):
4         super().__init__()
5         self.net = nn.Sequential(
6             nn.Linear(n_points + n_classes, 128),
7             nn.ReLU(),
8             nn.Linear(128, 128),
9             nn.ReLU(),
10            nn.Linear(128, n_points)
11        )
12     def forward(self, x, label):
13         return self.net(torch.cat([x, label], dim=1))
14
15 model = DenoiseMLP(n_points, n_classes=5)

```

This architecture is deliberately small to keep training fast and interpretable. The model learns a mapping from noisy versions of the signal to cleaner versions at earlier diffusion steps.

Step 3: Training the Denoising Model. The training process simulates a forward diffusion process by adding Gaussian noise to the clean signal in small increments, then teaches the model to reverse this process step by step. We use a simple mean squared error (MSE) loss to train the model to predict the less-noisy signal at the previous step.

Optimizer and Noise Generator

```

1 optimizer = optim.Adam(model.parameters(), lr=1e-3)
2 loss_fn = nn.MSELoss()
3
4 # Add progressive noise
5 def add_noise_sequence(x0, n_steps, noise_std):
6     x = x0.clone()
7     trajectory = [x]
8     for _ in range(n_steps):
9         x = x + (noise_std / n_steps) * torch.randn_like(x)
10        trajectory.append(x)
11    return trajectory

```

The function `add_noise_sequence` returns a list of noisy versions x_t starting from the clean signal x_0 . Each step adds a small amount of Gaussian noise.

The training loop samples a noisy trajectory and trains the model to denoise from x_t to x_{t-1} , starting from the noisiest point and moving backwards:

Diffusion Network Training Loop

```

1 for epoch in range(1001):
2     noisy_traj = add_noise_sequence(x_target, n_steps, noise_std)
3     loss = 0.0
4     for t in reversed(range(1, len(noisy_traj))):
5         x_in = noisy_traj[t]
6         x_out = noisy_traj[t-1]
7         x_pred = model(x_in, label)
8         loss += loss_fn(x_pred, x_out)
9     loss /= n_steps
10    optimizer.zero_grad()
11    loss.backward()
12    optimizer.step()
13    if epoch % 100 == 0:
14        print(f"Epoch {epoch}: Loss = {loss.item():.6f}")

```

The model is trained on one fixed sine class, identified by its one-hot label. The loss accumulates over all denoising steps and is averaged before each optimization step. This process teaches the model to reverse the noisy trajectory — one step at a time — and eventually generate a clean sine signal when starting from white noise.

Step 4: Reverse Sampling and Visualization After training, we use the model as an iterative denoiser: starting from white noise, we repeatedly apply the learned function to reduce noise and guide the signal toward a sine-like structure. This mimics the reverse trajectory of a diffusion process.

Denoise from White Noise

```

1 # --- Reverse from noise ---
2 with torch.no_grad():
3     x = torch.randn_like(x_target)
4     x_start = x.clone()
5     traj = [x.squeeze().numpy()]
6     for _ in range(n_steps):
7         x = model(x, label)
8         traj.append(x.squeeze().numpy())

```

The variable `traj` stores each intermediate signal. To observe how the denoising unfolds, we visualize the first few steps of the reconstruction process.

```

1 # --- Plot first 15 steps + start, final, target ---
2 import numpy as np
3 plt.figure(figsize=(10, 5))
4
5 steps_to_plot = list(range(0, 15)) # show early progress
6 colors = plt.cm.plasma(np.linspace(0.3, 1.0, len(steps_to_plot)))

```

```

7
8 for i, t in enumerate(steps_to_plot):
9     plt.plot(traj[t], color=colors[i], alpha=0.9, linewidth=2, label=f"Step {t}")
10
11 # Add clearly labeled curves
12 plt.plot(x_start.squeeze().numpy(), 'gray', linestyle='dashed', linewidth=2, label="Start (noise)")
13 plt.plot(traj[-1], 'k.', linestyle='--', linewidth=2, label="Final (output)")
14 plt.plot(x_target.squeeze().numpy(), 'g', linestyle='--', linewidth=2, label="Target (sine)")
15
16 plt.title("First n Steps of Denoising from Noise")
17 plt.xlabel("x index")
18 plt.ylabel("Signal value")
19 plt.legend(loc="upper right")
20 plt.grid(True)
21 plt.tight_layout()
22 plt.savefig("diffusion_denoising.png")
23 plt.show()

```

The resulting figure (see Figure 9.2) shows the progressive alignment of the noisy signal toward the clean sine curve.

9.1.6 Generating Functions from White Noise Using Conditional Diffusion

In many physical systems, outputs are structured signals — for example, 1D fields like temperature profiles or time-series measurements. These are influenced by both stochastic variability and contextual constraints (e.g., location, boundary conditions, or categories).

In this section, we explore how diffusion models can generate such signals starting from white noise and guided by a conditioning input — here, a class index that selects one of several sine wave types. This basic idea generalizes to more complex applications, such as generating spatial weather fields from forecasts or even from text prompts.

Target Functions: Shifted Sine Waves

We define n_{classes} shifted sine curves:

$$s_k(x) = \sin\left(x + \frac{2\pi k}{n_{\text{classes}}}\right), \quad k = 0, \dots, n_{\text{classes}} - 1,$$

sampled on a fixed grid. Each class represents a different target signal.

Training the Reverse Process

The model is trained to denoise progressively. We begin from a clean sine wave x_0 , add noise step-by-step to generate a trajectory x_0, x_1, \dots, x_T , and train the model to learn the reverse steps.

Each step of the model takes a noisy signal x_t and a one-hot encoded class label y , and predicts the cleaner x_{t-1} . The total loss accumulates the mean squared error across all reverse steps:

training loop with class-conditioned denoising

```

1 optimizer = optim.Adam(model.parameters(), lr=0.001)
2 mse_loss = nn.MSELoss()
3
4 for epoch in range(601):
5     idx = torch.randint(0, n_classes, (batch_size,))
6     x_clean = sine_set[idx]
7     labels = torch.nn.functional.one_hot(idx, num_classes=n_classes).float()
8
9     x_t = x_clean.clone()
10    trajectory = [x_t.clone()]
11    for _ in range(n_steps):
12        x_t = add_noise_step(x_t, noise_std / n_steps)
13        trajectory.append(x_t.clone())
14
15    loss = 0.0
16    for step in reversed(range(1, n_steps + 1)):
17        x_input = trajectory[step]
18        x_target = trajectory[step - 1]
19        x_input_with_label = torch.cat([x_input, labels], dim=1)
20        x_pred = model(x_input_with_label)
21
22        loss_denoise = mse_loss(x_pred, x_target)
23        target_template = labels @ sine_set
24        loss_template = mse_loss(x_pred, target_template)
25        cos = nn.CosineSimilarity(dim=1)
26        loss_cos = 1 - cos(x_pred, target_template).mean()
27        loss_align = mse_loss(x_pred, x_clean) if step == 1 else 0.0
28
29        loss += loss_denoise + 2 * loss_template + 0.2 * loss_cos
30
31    loss /= n_steps
32    optimizer.zero_grad()
33    loss.backward()
34    optimizer.step()

```

Generating a Function from Noise

After training, we can generate a signal starting from white noise. The only input to the model is a class label — for example, the instruction “give me the curve for class 2.” The model then iteratively denoises the signal.

sampling one function from white noise

```

1 x = torch.randn(1, n_points)
2 label = torch.nn.functional.one_hot(torch.tensor([class_idx]), num_classes=
    n_classes).float()
3
4 for _ in range(n_steps):
5     x_input = torch.cat([x, label], dim=1)
6     x = amplify * model(x_input)

```

Visualization and Results

Figure 9.3 shows multiple generated functions for different class labels, compared to the reference sine curves. Despite starting from pure noise, the output converges to class-consistent functions.

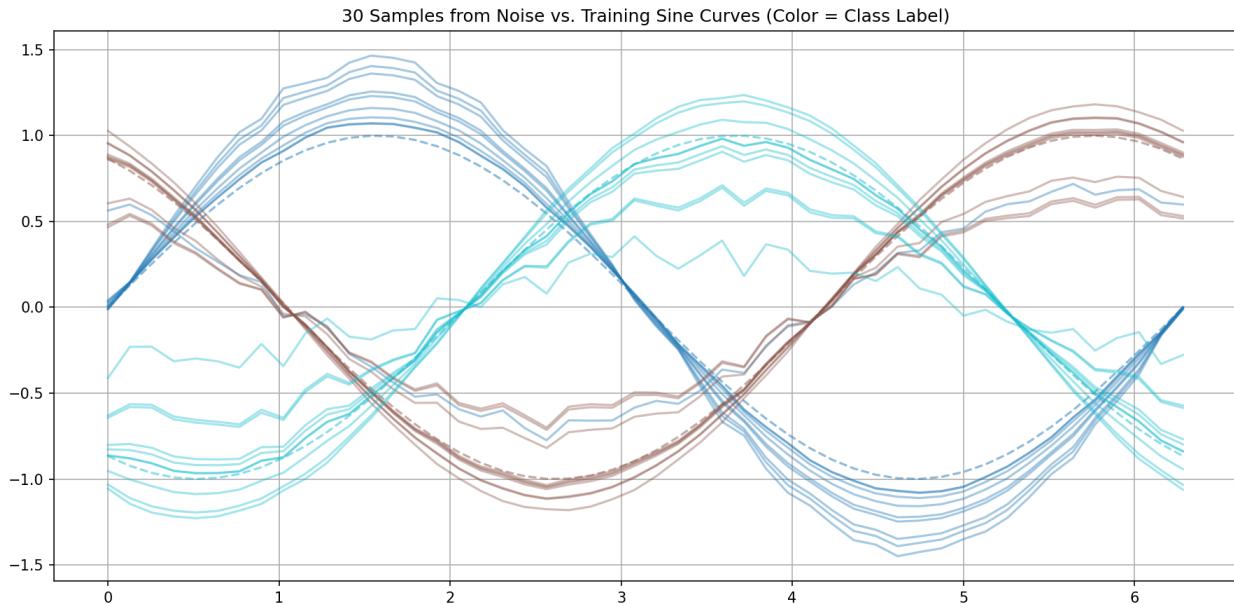


Figure 9.3: Samples generated from white noise and guided by class label. Colored lines show generated samples, dashed lines the target sine functions.

To visualize how structure emerges, we also track full denoising trajectories. In each subplot of Figure 9.4, the signal starts as noise and evolves step-by-step toward the final shape. The gray curve is the ground-truth sine.

Summary and Perspective

This example illustrates the core idea of diffusion-based generation: from noise to signal, guided by side information. While we use class labels here, the conditioning input could be general — spatial context, metadata, or even a short natural language prompt.

In weather and climate modeling, this approach can be extended to:

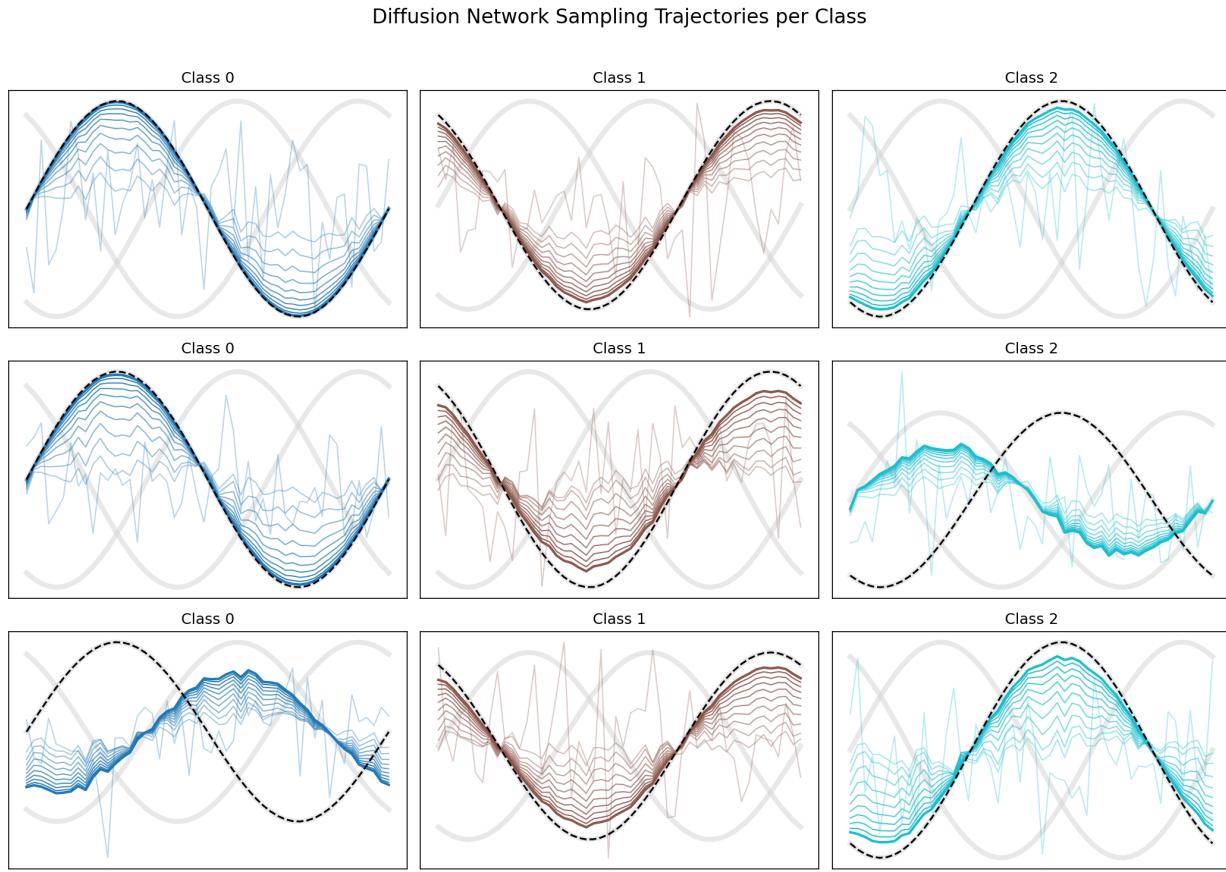


Figure 9.4: Denoising trajectories for 9 generated signals. Each panel shows 10 intermediate steps from noise to final output. The gray curve is the class target.

- Generate ensemble forecast members from coarse NWP input,
- Downscale global simulations to local weather fields,
- Impute missing or corrupted observations,
- Build hybrid models integrating physics and data.

We now have a powerful framework to model structured distributions over functions — not by directly learning the output, but by learning how to refine noise into data.

9.1.7 Why it Works: Diffusion Network Theory for the Scalar Case

To better understand the probabilistic nature of diffusion models, we consider a one-dimensional (scalar) example. Let the original data distribution be a Gaussian:

$$x_0 \sim \mathcal{N}(\mu_0, \sigma_0^2),$$

with a concrete choice of parameters $\mu_0 = 3$, $\sigma_0^2 = 1$. The forward diffusion step adds Gaussian noise:

$$x_1 = x_0 + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2).$$

By basic facts on Gaussian distributions, the distribution of x_1 becomes:

$$x_1 \sim \mathcal{N}(\mu_0, \sigma_0^2 + \sigma^2).$$

Optimal Denoiser. The optimal denoiser in the mean-squared error sense is the conditional expectation:

$$f^*(x_1) := \mathbb{E}[x_0 | x_1].$$

Lets do the little derivation here. Assume $x_0 \sim \mathcal{N}(\mu_0, \sigma_0^2)$ is the clean signal, and the observed noisy signal is $x_1 = x_0 + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is independent Gaussian noise. Then, using Bayes' theorem, the posterior distribution of x_0 given x_1 is:

$$p(x_0 | x_1) = \frac{p(x_1 | x_0) \cdot p(x_0)}{p(x_1)}.$$

Let us recall a standard identity for univariate Gaussian distributions, noting that $p(x_1)$ serves as normalization constant in the Bayes formula. The (unnormalized) product of two Gaussians

$$\mathcal{N}(x | \mu_1, \sigma_1^2) \cdot \mathcal{N}(x | \mu_2, \sigma_2^2) \propto \mathcal{N}\left(x | \mu_{\times}, \sigma_{\times}^2\right),$$

is again a Gaussian with:

$$\begin{aligned}\sigma_{\times}^2 &= \left(\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2} \right)^{-1} = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}, \\ \mu_{\times} &= \sigma_{\times}^2 \left(\frac{\mu_1}{\sigma_1^2} + \frac{\mu_2}{\sigma_2^2} \right) = \frac{\sigma_2^2 \mu_1 + \sigma_1^2 \mu_2}{\sigma_1^2 + \sigma_2^2}.\end{aligned}$$

We now apply this directly to the posterior:

$$p(x_0 | x_1) \propto p(x_1 | x_0) \cdot p(x_0),$$

with:

$$p(x_0) = \mathcal{N}(x_0 | \mu_0, \sigma_0^2), \quad p(x_1 | x_0) = \mathcal{N}(x_1 | x_0, \sigma^2),$$

which we can rewrite as:

$$p(x_1 | x_0) = \mathcal{N}(x_0 | x_1, \sigma^2),$$

so the product becomes:

$$p(x_0 | x_1) \propto \mathcal{N}(x_0 | \mu_0, \sigma_0^2) \cdot \mathcal{N}(x_0 | x_1, \sigma^2).$$

By the above rule, the posterior is again Gaussian:

$$p(x_0 | x_1) = \mathcal{N}(x_0 | \mu_{\text{post}}, \sigma_{\text{post}}^2),$$

with:

$$\begin{aligned}\sigma_{\text{post}}^2 &= \frac{\sigma_0^2 \cdot \sigma^2}{\sigma_0^2 + \sigma^2}, \\ \mu_{\text{post}} &= \frac{\sigma^2 \mu_0 + \sigma_0^2 x_1}{\sigma_0^2 + \sigma^2}.\end{aligned}$$

This means

$$f^*(x_1) = \mathbb{E}[x_0 | x_1] = \mu_{\text{post}} = \frac{\sigma^2 \mu_0 + \sigma_0^2 x_1}{\sigma_0^2 + \sigma^2} = x_1 + (\mu_0 - x_1) \cdot \frac{\sigma^2}{\sigma_0^2 + \sigma^2}. \quad (9.1)$$

Hence, the optimal denoiser is a linear function pulling x_1 toward μ_0 , and reducing the variance. Using

$$\text{Var}(aX + b) = a^2 \cdot \text{Var}(X)$$

the variance of $f^*(x_1)$ in dependence of x_1 is calculated by:

$$\text{Var}(f^*(x_1)) = \left(\frac{\sigma_0^2}{\sigma_0^2 + \sigma^2} \right)^2 \cdot \text{Var}(x_1) = \frac{\sigma_0^4}{\sigma_0^2 + \sigma^2}.$$

To recover the original distribution, we must sample from the full posterior:

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

Then, the total variance becomes:

$$\text{Var}(x_0) = \text{Var}(f^*(x_1)) + \sigma_{\text{post}}^2 = \frac{\sigma_0^4 + \sigma_0^2 \cdot \sigma^2}{\sigma_0^2 + \sigma^2} = \frac{\sigma_0^2(\sigma_0^2 + \sigma^2)}{\sigma_0^2 + \sigma^2} = \sigma_0^2.$$

Why Iterative Sampling is Necessary. A single denoising step, modeled as the conditional expectation $f^*(x_1) = \mathbb{E}[x_0 | x_1]$, does not recover the original mean μ_0 of the data distribution. Instead, it produces a posterior mean μ_{post} that is a weighted combination of x_1 and μ_0 . The output is therefore still biased by the noise in x_1 .

To progressively move towards the original distribution $\mathcal{N}(\mu_0, \sigma_0^2)$, multiple denoising steps are needed. Each step slightly reduces the noise and shifts the estimate closer to the clean data manifold. This is the essence of iterative sampling in diffusion models: a sequence of reverse transitions from noise to data.

Suppose we add Gaussian noise in n steps, where each step adds noise with standard deviation σ/n . The variance of the noise added in step k is thus $\sigma_k^2 = \sigma^2/n^2$, and the total noise variance after all n steps is:

$$\sigma_{\text{total}}^2 = \sum_{k=1}^n \frac{\sigma^2}{n^2} = \frac{\sigma^2}{n}.$$

Let x_n denote the final noisy sample. To denoise step-by-step, we apply the optimal estimator in reverse order, using the posterior mean:

$$\mathbb{E}[x_{k-1} | x_k] = (1 - \gamma_k)x_k + \gamma_k \mu_{k-1}^{(\text{prior})},$$

where

$$\gamma_k = \frac{\frac{\sigma^2}{n^2}}{\sigma_0^2 + \frac{(k-1)\sigma^2}{n^2} + \frac{\sigma^2}{n^2}} = \frac{\sigma^2}{n^2 \sigma_0^2 + k \sigma^2}.$$

and $\mu_{k-1}^{(\text{prior})}$ is the expected prior mean of x_{k-1} before seeing x_k , obtained from previous steps, such as μ_0 if we did not rescale our original signal or the rescaled values when rescaling is used. This defines a recursion:

$$x_{k-1} = (1 - \gamma_k)x_k + \gamma_k \mu_0 = x_k + \gamma_k(\mu_0 - x_k),$$

starting from x_n . Since γ_k is consistently larger than

$$\gamma_{\min} := \frac{\sigma^2}{n^2\sigma_0^2 + n\sigma^2},$$

unrolling this recursion

$$x_{k-1} = (1 - \gamma_{\min})x_k + \gamma_{\min}\mu_0$$

for $k = n, n-1, \dots, 1$, with fixed $\gamma_{\min} \in (0, 1)$, we obtain (based the standard geometric series argument)

$$x_0 = (1 - \gamma_{\min})^n x_n + (1 - (1 - \gamma_{\min})^n) \mu_0 \rightarrow \mu_0, \quad n \rightarrow \infty,$$

i.e. it shows that the denoising path progressively pulls the expectation back toward the original mean μ_0 . Thus, iterative denoising converges to the true mean in the limit of infinitely many small noise-injection and denoising steps.

This confirms that iterative denoising with many small steps recovers the original distribution mean — even though each individual step is only a small correction.

Conclusion. This simple scalar example demonstrates a crucial aspect of diffusion models:

- The optimal denoiser $f^*(x_k)$ estimates the *mean* of the reverse distribution.
- To match the original data distribution, one must *sample* from the full posterior, which includes adding back calibrated noise.
- This justifies the stochastic reverse process used in diffusion models, where noise is injected in each denoising step.

9.2 Flexible Graph Networks for Learning from Sparse Observations

9.2.1 A Naive Approach: Learning Functions from Sparse Observations

In this first implementation, we train a Graph Neural Network (GNN) to reconstruct simple functions (sine and cosine) from partial observations on a one-dimensional grid. The setup is intentionally minimal and includes all components required to construct, train, and apply a graph-based model. Later sections will analyze and generalize this setup.

Overview: We aim to learn a mapping from sparse observations to a full function profile by training on synthetically generated sine and cosine functions with random phases. Observations are available at every M -th grid point. The GNN uses a fixed neighborhood structure defined by K -nearest neighbors.

The complete workflow includes the following components:

- Creating the spatial grid.
- Defining the graph connectivity.

- Designing the GNN architecture.
- Training the model on sampled functions.
- Testing the model on larger and more complex domains.

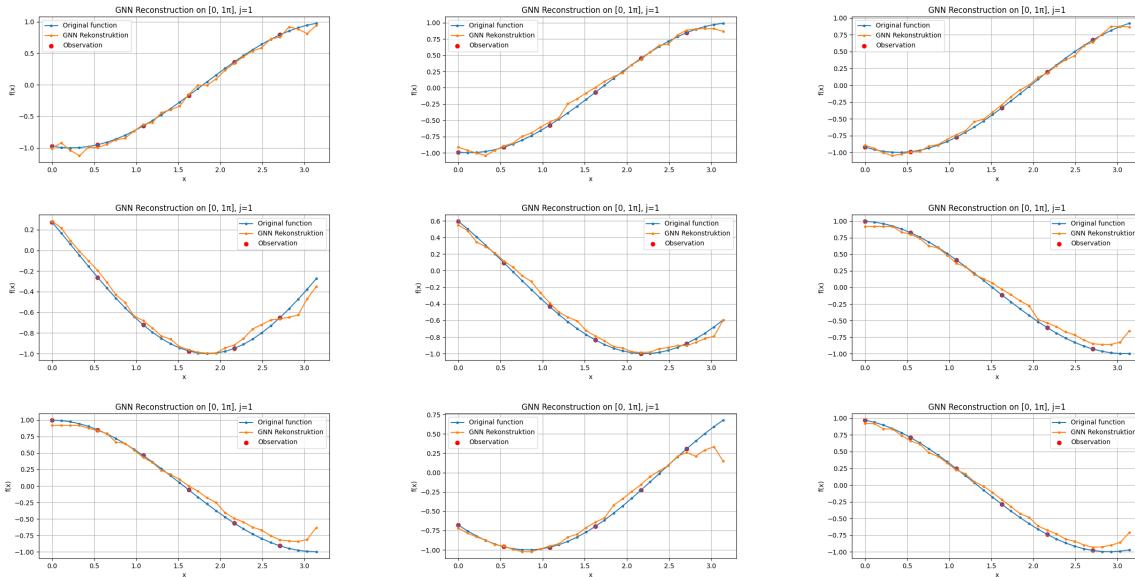


Figure 9.5: GNN reconstructions for $f(x) = \cos(x + \phi)$ on $[0, \pi]$ with different phases. Each panel shows the original function, sparse observations, and the GNN prediction.

Step 1: Libraries and Parameter Setup

Imports and Parameters

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 N = 30           # Number of grid points
8 M = 5            # Every M-th point is observed
9 K = 3            # Number of neighbors per node
10 num_epochs = 201
11 num_samples = 200
12 lr = 0.01
13
14 x_grid = torch.linspace(0, np.pi, N)

```

Step 2: Graph Construction Each node is connected to itself and its K neighbors on both sides. This creates a fixed undirected graph.

Graph Construction

```

1 edges = []
2 for i in range(N):
3     edges.append((i, i)) # Self-loop
4     for k in range(1, K + 1):
5         if i - k >= 0:
6             edges.append((i, i - k))
7         if i + k < N:
8             edges.append((i, i + k))
9 edge_index = torch.tensor(edges).T # [2, num_edges]

```

Step 3: Model Architecture We define a basic GNN architecture with two message-passing layers. Each layer aggregates features from neighboring nodes and applies a linear transformation.

Model Definition

```

1 class GNNLayer(nn.Module):
2     def __init__(self, dim):
3         super().__init__()
4         self.self_lin = nn.Linear(dim, dim)
5         self.neigh_lin = nn.Linear(dim, dim)
6
7     def forward(self, x, edge_index):
8         row, col = edge_index
9         agg = torch.zeros_like(x)
10        agg.index_add_(0, row, x[col])
11        return F.relu(self.self_lin(x) + self.neigh_lin(agg))
12
13 class GNN(nn.Module):
14     def __init__(self, in_dim=3, hidden=64):
15         super().__init__()
16         self.input_proj = nn.Linear(in_dim, hidden)
17         self.gnn1 = GNNLayer(hidden)
18         self.gnn2 = GNNLayer(hidden)
19         self.out = nn.Linear(hidden, 1)
20
21     def forward(self, x, edge_index):
22         x = F.relu(self.input_proj(x))
23         x = self.gnn1(x, edge_index)
24         x = self.gnn2(x, edge_index)
25         return self.out(x).squeeze(-1)
26
27 model = GNN(in_dim=3)
28 opt = torch.optim.Adam(model.parameters(), lr=lr)

```

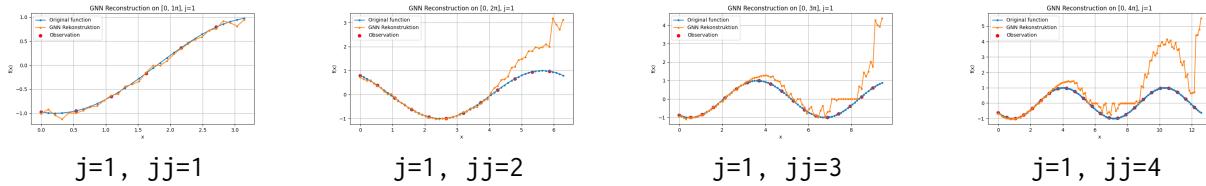


Figure 9.6: Generalization of the GNN to longer intervals. All test functions use $f(x) = \cos(x + \phi)$ but are evaluated on increasing domains $[0, jj \cdot \pi]$ with $jj = 1, 2, 3, 4$. The model was trained only on $[0, \pi]$.

Step 4: Training Loop The model is trained to reconstruct either a sine or cosine function with random phase. Observations are only available at every M -th grid point. The input to each node includes the position x , the (masked) observation value, and a binary mask.

Training Loop

```

1 for epoch in range(num_epochs):
2     losses = []
3     for _ in range(num_samples):
4         phase = torch.rand(1).item() * np.pi
5         f_type = torch.randint(0, 2, (1,)).item()
6         f = lambda x: torch.sin(x + phase) if f_type == 0 else torch.cos(x + phase)
7         y_true = f(x_grid)
8
9         mask = torch.zeros(N)
10        mask[::M] = 1.0
11        obs_y = y_true * mask
12
13        x_feat = torch.stack([x_grid, obs_y, mask], dim=1)
14
15        model.train()
16        pred = model(x_feat, edge_index)
17        loss = F.mse_loss(pred, y_true)
18        opt.zero_grad()
19        loss.backward()
20        opt.step()
21        losses.append(loss.item())
22
23    if epoch % 50 == 0:
24        print(f"Epoch {epoch}: Loss = {np.mean(losses):.6f}")

```

Step 5: Testing on Extended Domains We evaluate the trained model on functions of the form $f(x) = \cos(x + \phi) \cos((j - 1)x)$ over extended intervals $[0, j \cdot \pi]$. This tests the generalization ability of the GNN.

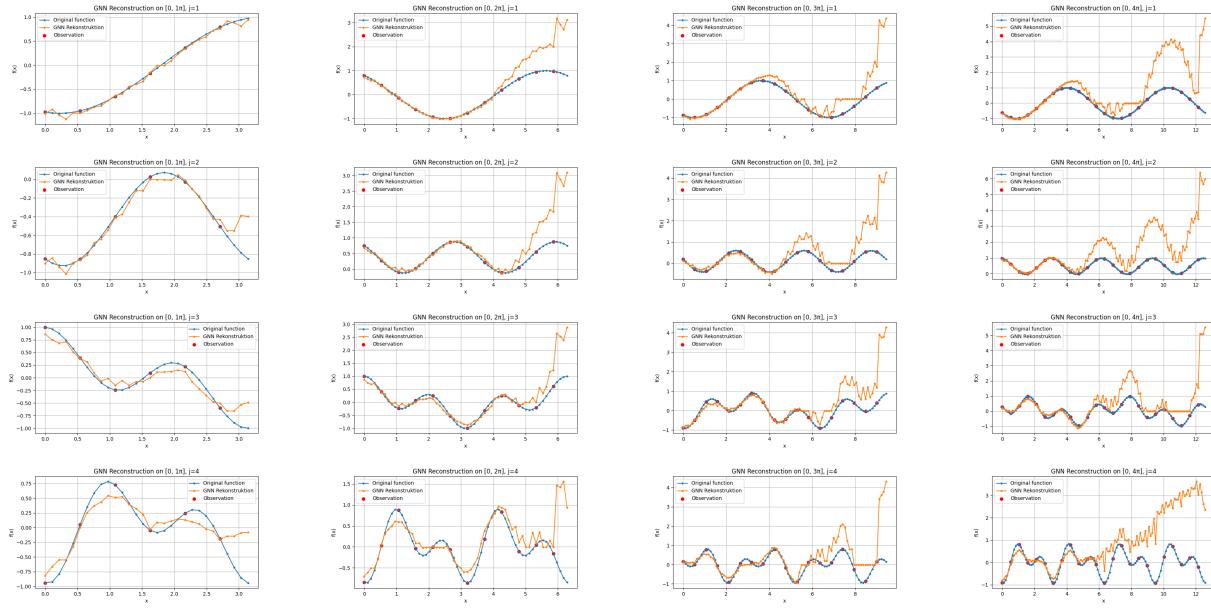


Figure 9.7: GNN reconstruction results for functions $f(x) = \cos(x + \phi) \cos((j - 1)x)$ with increasing frequency ($j = 1$ to 4) and interval length ($jj = 1$ to 4). Each image shows prediction based on sparse observations.

Test Function and Evaluation

```

1 def run_test(j, jj):
2     N_test = jj * 30
3     x_test_grid = torch.linspace(0, jj * np.pi, N_test)
4
5     phi_test = torch.rand(1).item() * np.pi
6     f_test = lambda x: torch.cos(x + phi_test) * torch.cos((j - 1) * x)
7     y_test = f_test(x_test_grid)
8
9     mask = torch.zeros(N_test)
10    mask[::M] = 1.0
11    obs = y_test * mask
12    x_feat_test = torch.stack([x_test_grid, obs, mask], dim=1)
13
14    edges = []
15    for i in range(N_test):
16        edges.append((i, i)) # Self-loop
17        for k in range(1, K + 1):
18            if i - k >= 0:
19                edges.append((i, i - k))
20            if i + k < N_test:
21                edges.append((i, i + k))
22    edge_index_test = torch.tensor(edges).T
23
24    model.eval()

```

```

25     with torch.no_grad():
26         pred_test = model(x_feat_test, edge_index_test)
27
28     plt.figure(figsize=(8, 4))
29     plt.plot(x_test_grid, y_test, '.-', label="Original function")
30     plt.plot(x_test_grid, pred_test, '.-', label="GNN Rekonstruktion")
31     plt.scatter(x_test_grid[::M], y_test[::M], color='red', label="Observation")
32     plt.title(f"GNN Reconstruction on [0, {jj}pi], j={j}")
33     plt.xlabel("x")
34     plt.ylabel("f(x)")
35     plt.grid(True)
36     plt.legend()
37     plt.show()
38
39 for j in range(1, 5):
40     run_test(1, jj=j)
41     run_test(j, jj=j)

```

Conclusion This naive setup already demonstrates the capacity of GNNs to reconstruct continuous functions from sparse data. In the next sections, we will explore how to generalize this setup and test whether absolute positional information (i.e., x) is truly necessary for accurate reconstructions.

9.2.2 Coordinate-Free GNN: Learning from Observations Alone

In this variation, we explore the idea of removing the coordinate x from the GNN input entirely. Instead of feeding the spatial location explicitly into the model, we rely solely on two pieces of information per node:

- the observed function value (or zero if unobserved),
- a binary mask indicating whether the value was observed.

The graph topology itself continues to encode spatial relationships via K -nearest neighbors. This setup allows the network to learn purely from **relational** and **structural** information.

To implement this, we change the input feature dimension to 2 and remove x from the input vector:

Model Without Coordinates

```

1 class GNN(nn.Module):
2     def __init__(self, in_dim=2, hidden=64):
3         ...
4     x_feat = torch.stack([obs_y, mask], dim=1) # No x-coordinate
5     model2 = GNN(in_dim=2)

```

The training loop remains identical. For testing, we run the same function ‘run_test(j, jj, tag)’ but again exclude the coordinate x from the input:

Coordinate-Free Testing

```

1 x_feat_test = torch.stack([obs, mask], dim=1)
2 pred_test = model2(x_feat_test, edge_index_test)
3 plt.savefig(f"gnn_obs_no_pos_j{j}_jj{jj}_{tag}.png")

```

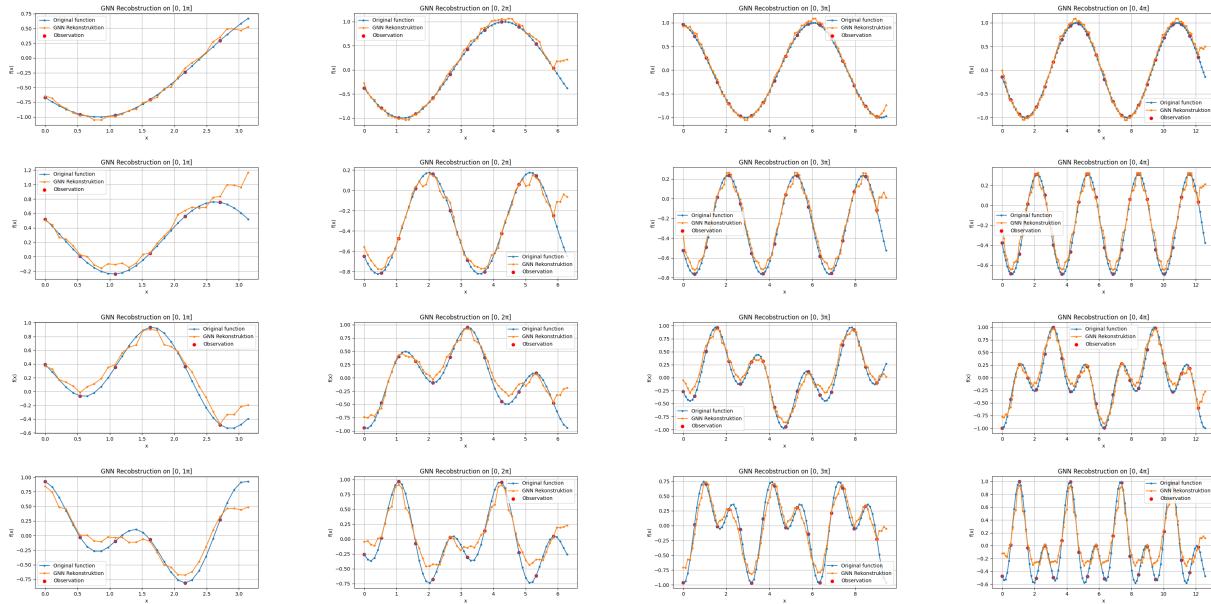


Figure 9.8: GNN reconstructions without coordinate input. Each panel shows $f(x) = \cos(x + \phi) \cos((j - 1)x)$ reconstructed from sparse observations, across increasing frequency $j = 1 \dots 4$ and domain size $jj = 1 \dots 4$. The model was trained without using x -position as input and on shifted sines only, no higher modes in the training dataset.

Surprising Result: Better Generalization Without Coordinates This variant performs *significantly better* when generalizing to longer domains and higher-frequency functions. While this may appear counterintuitive at first, the reason is rooted in how the model interprets the coordinate input.

When x is included as a feature, the GNN must learn to "undo" the fact that x changes range across intervals. For example, $x \in [0, \pi]$ during training but later $x \in [0, 4\pi]$ during testing — unless the network has learned that x is irrelevant, it may overfit to this scale and fail to generalize.

By removing x , we force the network to focus on **local patterns** and **relational geometry** encoded in the graph structure — which remains scale-invariant and consistent across test domains.

9.3 Exploring Graph Structures in Detail

9.3.1 Inspecting the Number of Trainable Parameters

Graph neural networks (GNNs) operate on graph-structured data by aggregating and transforming node features according to a given graph topology. The structure of the graph is provided through the `edge_index`, which specifies the connections between nodes, while the trainable parameters of the model are applied uniformly across the entire graph.

Edge Index and GNN Forward Pass

```
1 # x: [N, F] feature matrix for N nodes, each with F features
2 # edge_index: [2, E] matrix with E directed edges (source, target)
3
4 x = model(x, edge_index)
```

The `edge_index` is a $2 \times E$ tensor that defines the spatial structure of the graph by listing all E edges as source-target pairs. It governs how information flows between nodes, but it does not introduce any additional trainable parameters. Instead, the weights in a GNN are shared across all nodes and edges, making the model invariant to the graph size and applicable to different domains.

Each layer in the model—such as linear projections and message-passing operations—has its own set of parameters, which are learned during training. These parameters determine **how** node features are updated, while the `edge_index` determines **where** messages are exchanged. As a result, we can reuse the same trained model architecture and weights on graphs of varying size or structure, as long as the input feature format remains consistent.

Parameter Numbers. We now explore the exact number of trainable parameters in our models using a simple Python function and interpret how input dimensionality affects model complexity.

To measure the size of a model, we compute the total number of trainable parameters using the function `n_element()` for each parameter tensor. The command shown sums over all layers and returns a single integer that quantifies the model's capacity.

One Linear Model Parameters

```
1 sum([param.n_element() for param in model.parameters()])
```

To inspect the complexity of our models, we define a simple Python function that iterates over all trainable parameters and prints their names, shapes, and counts:

Simple Parameter Summary

```
1 def simple_param_summary(model):
2     print('-'*80)
3     total = 0
4     print("Layer Summary:")
5     for name, param in model.named_parameters():
6         count = param.numel()
7         total += count
```

```
8     print(f"{{name:30} | shape: {list(param.shape)} | params: {count}}")
9     print(f"\nTotal trainable parameters: {total}")
```

This function produces an informative overview, showing the parameter count for each layer and the overall model. We apply it to two model variants: **model**, which uses the full feature vector [x, val, mask] with input dimension 3, and **model2**, which omits the coordinate and uses only [val, mask] with input dimension 2.

Result for model (with coordinate):

input_proj.weight	shape: [64, 3]	params: 192
input_proj.bias	shape: [64]	params: 64
ggn1.self_lin.weight	shape: [64, 64]	params: 4096
ggn1.self_lin.bias	shape: [64]	params: 64
ggn1.neigh_lin.weight	shape: [64, 64]	params: 4096
ggn1.neigh_lin.bias	shape: [64]	params: 64
ggn2.self_lin.weight	shape: [64, 64]	params: 4096
ggn2.self_lin.bias	shape: [64]	params: 64
ggn2.neigh_lin.weight	shape: [64, 64]	params: 4096
ggn2.neigh_lin.bias	shape: [64]	params: 64
out.weight	shape: [1, 64]	params: 64
out.bias	shape: [1]	params: 1

Total trainable parameters: 16961

Result for model2 (without coordinate):

input_proj.weight	shape: [64, 2]	params: 128
input_proj.bias	shape: [64]	params: 64
... (identical hidden layers and output)		

Total trainable parameters: 16897

The only difference lies in the first linear layer. With three input features, the input projection layer has $64 \times 3 + 64 = 256$ parameters, while with two inputs it has $64 \times 2 + 64 = 192$ parameters. This difference of exactly 64 parameters matches the total discrepancy between the models.

Interpretation: The rest of the model—two GNN layers with shared hidden size and the final output projection—remains identical in both cases. This small change in input dimensionality helps us isolate the effect of using or omitting positional information (*x*) while keeping model capacity nearly the same.

This method is a convenient way to audit model size, debug unexpected parameter growth, or document the impact of architectural choices.

9.3.2 Marching through the Graph Processing

Step 1: Raw Node Input Features. Each node *i* in the graph carries basic local information

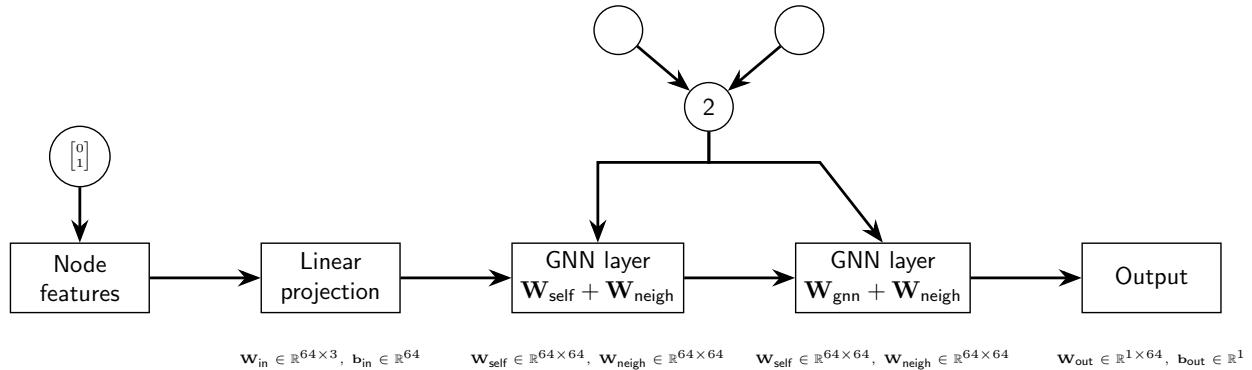


Figure 9.9: Schematic of the GNN architecture with input projection, two GNN layers using neighborhood aggregation, and output transformation.

encoded as a 2-dimensional vector:

$$\mathbf{x}_i^{(0)} = \begin{bmatrix} \text{val}_i \\ \text{mask}_i \end{bmatrix} \in \mathbb{R}^2.$$

This input consists of:

- val_i : the observed function value at the node i , or 0 if the value is unobserved,
- $\text{mask}_i \in \{0, 1\}$: a binary indicator specifying whether the value is present (1) or missing (0).

The full input feature matrix for all N nodes in the graph is:

$$X^{(0)} = \begin{bmatrix} \cdots & \mathbf{x}_1^{(0)} & \cdots \\ \cdots & \mathbf{x}_2^{(0)} & \cdots \\ \vdots & & \vdots \\ \cdots & \mathbf{x}_N^{(0)} & \cdots \end{bmatrix} \in \mathbb{R}^{N \times 2}.$$

Motivation: This minimal feature representation provides the model with:

- The available measurement data (only at selected nodes),
- An explicit signal of which nodes are observed and which are not,
- A compact and domain-agnostic encoding that avoids absolute positional information.

These raw inputs are passed to the first learnable layer of the network — the linear projection — which maps them into a higher-dimensional latent space for further processing and message passing.

Step 2: Linear Input Projection Layer. Each node in the graph is initially represented by a 2-dimensional feature vector:

$$\mathbf{x}_i^{(0)} = \begin{bmatrix} \text{val}_i \\ \text{mask}_i \end{bmatrix} \in \mathbb{R}^2,$$

where

- val_i is the observed function value at node i (or zero if unobserved),
- $\text{mask}_i \in \{0, 1\}$ indicates whether the observation is present.

These raw features are passed through a learned linear transformation to project them into a higher-dimensional feature space:

$$\mathbf{x}_i^{(1)} = \sigma(W\mathbf{x}_i^{(0)} + \mathbf{b}), \quad W \in \mathbb{R}^{64 \times 2}, \quad \mathbf{b} \in \mathbb{R}^{64},$$

where σ is the ReLU activation function applied component-wise:

$$\sigma(z) = \max(0, z).$$

This operation is applied independently to all N nodes in the graph, resulting in a projected feature matrix

$$X^{(1)} \in \mathbb{R}^{N \times 64}.$$

Purpose: This step enables the model to:

- Interpret the semantic meaning of the raw inputs,
- Learn different representations for observed and unobserved nodes,
- Embed all nodes into a common latent space for message passing.

Interpretation: The weights in W determine how much influence the observed value and the mask have on each of the 64 embedding dimensions. For example:

- Some dimensions may be dominated by the presence or absence of data,
- Others may capture approximate magnitudes or trends across the domain.

This projected representation is then passed to the GNN layers, where it is refined through interaction with neighboring nodes.

Step 3: Message Passing with GNNLayer. After the input features have been projected to the latent space \mathbb{R}^{64} , the node embeddings are updated using the graph structure. This happens through two successive GNNLayer blocks, each of which applies message passing followed by a learned transformation.

For the first layer, the input is $X^{(1)} \in \mathbb{R}^{N \times 64}$. The message passing operation updates each node i based on its own features and the features of its neighbors, as defined by the edge list `edge_index`.

Each GNNLayer performs the following update:

$$\mathbf{x}_i^{(\ell+1)} = \sigma \left(W_{\text{self}} \mathbf{x}_i^{(\ell)} + W_{\text{neigh}} \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j^{(\ell)} \right),$$

where:

- $\mathbf{x}_i^{(\ell)} \in \mathbb{R}^{64}$ is the node feature vector at layer ℓ ,

- $\mathcal{N}(i)$ is the set of neighbors of node i (including self-loop),
- $W_{\text{self}}, W_{\text{neigh}} \in \mathbb{R}^{64 \times 64}$ are learned weight matrices,
- σ is the ReLU activation function.

The neighborhood aggregation is computed via:

$$\mathbf{a}_i = \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j^{(\ell)},$$

which is then linearly transformed and added to the transformed self-information. In PyTorch, this is implemented by accumulating neighbor features with the operation:

```
agg = torch.zeros_like(x)
agg.index_add_(0, row, x[col])
```

where `edge_index = [row, col]` encodes all directed edges from node `col[k]` to node `row[k]`.

Interpretation: This layer learns how to combine a node's own state with information from its local neighborhood. It does so in a trainable, nonlinear way, which enables it to reconstruct smooth or structured functions based on sparsely observed values.

Two such GNN layers are applied in sequence in the model, each refining the representation based on a wider receptive field (i.e., information from nodes up to two hops away).

Step 4: Output Projection. After two rounds of message passing, each node i is represented by a refined hidden state vector:

$$\mathbf{x}_i^{(3)} \in \mathbb{R}^{64}.$$

To convert this latent representation into a scalar prediction (e.g., an estimate of the underlying function value at node i), the model applies a final linear projection:

$$\hat{y}_i = w^\top \mathbf{x}_i^{(3)} + b, \quad w \in \mathbb{R}^{64}, \quad b \in \mathbb{R}.$$

This is implemented in PyTorch as:

```
self.out = nn.Linear(64, 1)
```

Applied to all N nodes, the output becomes:

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{bmatrix} \in \mathbb{R}^N,$$

which is compared to the true function $y \in \mathbb{R}^N$ during training using a mean squared error loss:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2.$$

Purpose: This step transforms the latent node embedding back into the physical space of scalar function values. It completes the reconstruction process by producing a prediction for every node in the graph — observed or not.

Interpretation: The model learns to extract the relevant features through the graph structure and to compress them into a meaningful final scalar output. Since the same output transformation is applied to every node, the model remains fully permutation-invariant with respect to the node order.

Final Remarks on Structure and Complexity. The graph network architecture described in this section is compact yet expressive. It processes node-wise inputs in four clearly structured stages as shown in Figure 9.9:

- input feature encoding,
- projection into a latent space,
- two rounds of message passing, and
- a final output projection.

Each GNNLayer updates node representations by combining a self-weighted embedding with a shared transformation of its neighbors — using just two weight matrices per layer, independent of the graph's size or shape.

The total number of parameters remains modest (16,961 in the case of three input features), with the majority concentrated in the two GNN layers. Despite this simplicity, the model captures complex interactions across the graph and generalizes naturally to different domains. Crucially, the graph structure itself is not learned but provided externally via edge_index, making the model both interpretable and highly adaptable. This layered design, along with consistent weight sharing and localized aggregation, gives the network its power to generalize, interpolate, and reconstruct global behavior from sparse, local input.

9.4 PyTorch Lightning and PyTorch Geometric

Modern PyTorch workflows benefit greatly from using helper libraries that simplify model training and extend PyTorch with domain-specific components. Two such libraries are:

- **PyTorch Lightning:** A high-level wrapper around PyTorch that structures training, logging, and hardware management into a clean and scalable format.
- **PyTorch Geometric (PyG):** A specialized library for graph neural networks (GNNs), enabling message passing, pooling, and graph data structures.

These libraries integrate seamlessly and can be used together to train graph-based models in a clean and efficient way.

To follow the examples in this tutorial, the following packages must be installed. Installation should be done in a Python 3.9+ environment with PyTorch already installed.

Installation via pip

```

1 pip install pytorch-lightning
2 pip install torch-geometric
3 pip install torch-scatter torch-sparse torch-cluster

```

PyTorch Lightning and **PyTorch Geometric** are two independent libraries, both built on top of PyTorch:

- PyTorch Lightning simplifies model training, logging, and hardware management, but is agnostic to the model type.
- PyTorch Geometric specializes in graph data structures and operations, enabling message-passing networks and geometric deep learning.

You can use either library independently or combine them:

- PyG can be used alone to build and train GNNs using a manual training loop.
- Lightning can be used to train any model, including non-graph models like CNNs, RNNs, or transformers.
- Using both together is ideal for clean training of graph models at scale, with integrated logging and device handling.

Use Case	PyG only	Lightning only	PyG + Lightning
Small GNN script	✓	—	optional
General ML model training	—	✓	optional
Large graph model w/ logging	✓	✓	✓ (recommended)
Clean training loop	—	✓	✓
Graph-specific layers	✓	—	✓

This tutorial demonstrates both libraries separately and together, helping you choose the best level of abstraction for your needs.

9.4.1 A PyTorch Lightning Example

As a first example of PyTorch Lightning, we define a regression task where the model learns to reconstruct a shifted sine function from a sparse set of observations. Each sample is a 1D graph of points connected to their 3 nearest neighbors to the left and right. Node features include both the (possibly missing) observed values and a binary observation mask.

Data Generation. We generate sine curves with random phase shifts and mask most values. Each graph stores the full target y , the sparse observation vector y_{obs} , and a 2-channel input feature vector.

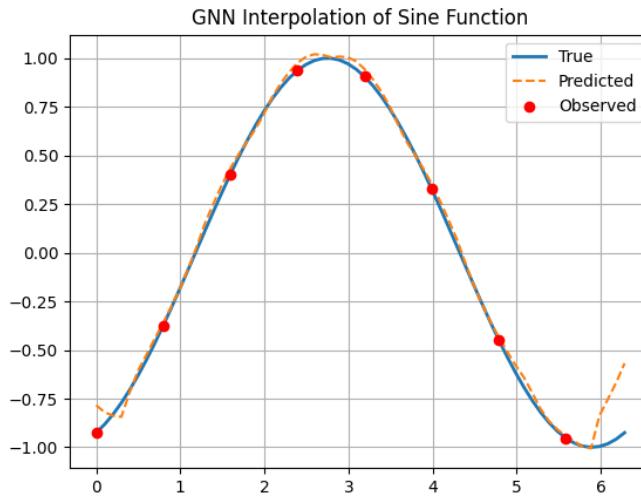


Figure 9.10: Graph neural network interpolation of a shifted sine function using PyTorch Lightning and PyTorch Geometric. The model observes only a small subset of values (red dots) and learns to reconstruct the full curve (orange dashed line). The ground truth is shown as a solid blue line.

Sine Graph with Observation Mask

```

1 def generate_sample_graph(n_nodes=64, dn=8, k=3):
2     x_vals = torch.linspace(0, 2 * np.pi, n_nodes)
3     phase = np.random.uniform(0, 2 * np.pi)
4     y_true = torch.sin(x_vals + phase)
5
6     y_obs = torch.full_like(y_true, float('nan'))
7     idx = torch.arange(0, n_nodes, dn)
8     y_obs[idx] = y_true[idx]
9
10    is_observed = ~torch.isnan(y_obs)
11    input_feat = torch.stack([
12        torch.nan_to_num(y_obs, nan=0.0),
13        is_observed.float()
14    ], dim=1)
15
16    edges = []
17    for i in range(n_nodes):
18        for j in range(1, k + 1):
19            if i - j >= 0:
20                edges.append((i, i - j))
21            if i + j < n_nodes:
22                edges.append((i, i + j))
23    edge_index = torch.tensor(edges, dtype=torch.long).t().contiguous()
24
25    return Data(x_feat=input_feat, y=y_true.unsqueeze(1), y_obs=y_obs.unsqueeze(1),
26 , edge_index=edge_index)

```

Lightning Model. The Lightning module uses three GCN layers. The input has two channels (value and mask), and the model is trained to match the true signal where no observations were provided.

PyTorch Lightning GCN

```

1 class GNNInterpolator(pl.LightningModule):
2     def __init__(self):
3         super().__init__()
4         self.gcn1 = GCNConv(2, 64)
5         self.gcn2 = GCNConv(64, 128)
6         self.out = GCNConv(128, 1)
7
8     def forward(self, data):
9         x = data.x_feat
10        x = F.relu(self.gcn1(x, data.edge_index))
11        x = F.relu(self.gcn2(x, data.edge_index))
12        return self.out(x, data.edge_index)
13
14    def training_step(self, batch, batch_idx):
15        pred = self(batch)
16        mask = ~torch.isnan(batch.y_obs)
17        loss = F.mse_loss(pred[~mask], batch.y[~mask])
18        self.log("train_loss", loss)
19        return loss
20
21    def configure_optimizers(self):
22        return torch.optim.Adam(self.parameters(), lr=0.01)

```

Training. Training is performed using a standard Lightning Trainer:

Training with Lightning

```

1 model = GNNInterpolator()
2 trainer = pl.Trainer(max_epochs=200, logger=False, enable_checkpointing=False)
3 trainer.fit(model, loader)

```

The trained model generalizes well to new shifted sine curves and accurately reconstructs them from sparse values.

9.4.2 A PyTorch Geometric Example

Graph neural networks (GNNs) are well-suited for structured data with local neighborhood dependencies. This example shows how PyTorch Geometric can be used to infer a function—such as a shifted sine curve—from sparse observations using graph convolution layers.

Graph Structure and Input Features. Each sample is a graph of 64 nodes uniformly spaced on the interval $[0, 2\pi]$, connected to their 3 left and 3 right neighbors. Each node has two features: the (possibly missing) observation value and a binary mask indicating whether it was observed.

Data Generation with Graph and Mask

```

1 def generate_sample_graph(n_nodes=64, dn=8, k=3):
2     x_vals = torch.linspace(0, 2 * np.pi, n_nodes)
3     phase = np.random.uniform(0, 2*np.pi)
4     y_true = torch.sin(x_vals + phase)
5
6     y_obs = torch.full_like(y_true, float('nan'))
7     idx = torch.arange(0, n_nodes, dn)
8     y_obs[idx] = y_true[idx]
9
10    is_observed = ~torch.isnan(y_obs)
11    input_feat = torch.stack([
12        torch.nan_to_num(y_obs, nan=0.0),
13        is_observed.float()
14    ], dim=1)
15
16    edges = []
17    for i in range(n_nodes):
18        for j in range(1, k + 1):
19            if i - j >= 0:
20                edges.append((i, i - j))
21            if i + j < n_nodes:
22                edges.append((i, i + j))
23    edge_index = torch.tensor(edges, dtype=torch.long).t().contiguous()
24
25    return Data(x_feat=input_feat, y=y_true.unsqueeze(1), y_obs=y_obs.unsqueeze(1),
26 , edge_index=edge_index)

```

GCN Model. The model consists of three GCN layers applied to the 2-channel input.

PyTorch Geometric GCN

```

1 class GCNInterpolator(torch.nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.gcn1 = GCNConv(2, 64)
5         self.gcn2 = GCNConv(64, 128)
6         self.out  = GCNConv(128, 1)
7
8     def forward(self, data):
9         x = data.x_feat
10        x = F.relu(self.gcn1(x, data.edge_index))
11        x = F.relu(self.gcn2(x, data.edge_index))
12        return self.out(x, data.edge_index)

```

Training Loop. A standard PyTorch training loop is used with MSE loss evaluated only on non-observed points.

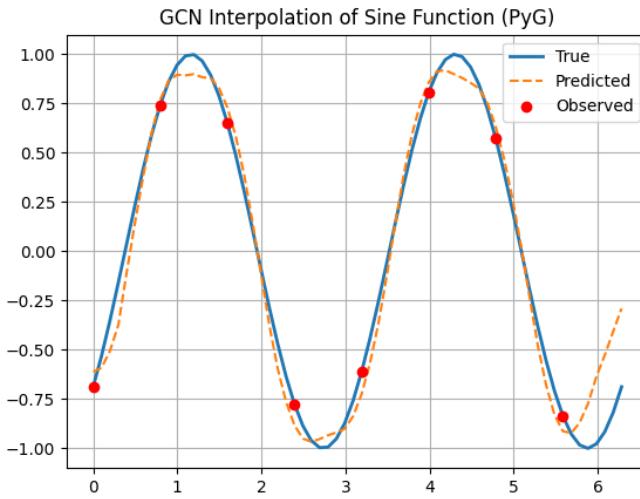


Figure 9.11: Function interpolation using a Graph Convolutional Network (GCN) in PyTorch Geometric. The model reconstructs a shifted sine curve (blue) from sparse observations (red dots). The predicted output (orange dashed line) closely follows the true function, demonstrating the model's ability to infer smooth structures from limited data by leveraging graph connectivity.

Training Loop

```

1 for epoch in range(100):
2     for batch in loader:
3         pred = model(batch)
4         mask = ~torch.isnan(batch.y_obs)
5         loss = F.mse_loss(pred[~mask], batch.y[~mask])
6         loss.backward()
7         optimizer.step()
8         optimizer.zero_grad()

```

Inference and Visualization. The trained model can interpolate new sine curves from sparse measurements. A typical output looks like this:

Plotting Output

```

1 plt.plot(x, true, label="True")
2 plt.plot(x, pred, label="Predicted", linestyle="--")
3 plt.scatter(x[obs_mask], obs[obs_mask], color='red', label="Observed")
4 plt.legend()

```

Chapter 10

Agents and Coding with LLM

10.1 Introduction to Automated Coding with LLMs

This section presents a workflow for integrating Large Language Models (LLMs) into code generation and execution. Using the OpenAI API, we demonstrate how to retrieve Python code from a prompt, run it within a notebook environment, and refine results through structured loops.

Retrieving Code from an LLM via Prompt

We begin by defining a simple function to query an LLM with a prompt and return Python code. The code below connects to OpenAI's API using an environment variable for authentication.

```
get_code_from_llm

1 import openai
2 import os
3
4 client = openai.Client(api_key=os.getenv("OPENAI_API_KEY"))
5
6 def get_code_from_llm(prompt):
7     response = client.chat.completions.create(
8         model="gpt-4o-mini",
9         messages=[
10             {"role": "system", "content": "You are an AI coder. Output ONLY Python
11             code that can be saved directly. No markdown formatting, no explanations."},
12             {"role": "user", "content": prompt}
13         ]
14     )
15     code = response.choices[0].message.content.strip()
16     if code.startswith("```"):
17         code = "\n".join(code.splitlines()[1:-1])
18
19     return code
```

Running the Generated Code with Error Handling

The following function executes the generated code from the LLM. The code is saved to a file and executed using Python's built-in exec. If an error occurs, it is captured and printed.

```
run_generated_code

1 import traceback
2
3 def run_generated_code(code, filename="generated_code.py"):
4     with open(filename, "w") as f:
5         f.write(code)
6     print(f"Saved to {filename}")
7     try:
8         print("Running code...\n")
9         exec(open(filename).read())
10        return True, None
11    except Exception as e:
12        tb = traceback.format_exc()
13        print("Error during execution:\n", tb)
14        return False, tb
```

Retrying Code Generation in a Loop

To improve robustness, we wrap the generation and execution in a retry loop. If an error occurs, the prompt is updated with the error traceback to help the LLM fix it.

```
LLM Code Retry Loop

1 for attempt in range(1, max_tries + 1):
2     print(f"\n==== Attempt {attempt} ====")
3     code = get_code_from_llm(current_prompt)
4     success, error = run_generated_code(code, filename=filename)
5     if success:
6         print(f"Code in {filename} executed successfully.")
7         break
8     else:
9         current_prompt = prompt + f"\n\nThe last run failed with this error:\n{error}\n\nPlease fix and regenerate valid Python code."
```

Encapsulating the Loop in a Function

The retry logic can be wrapped in a reusable function. It attempts up to max_tries to produce executable code.

llm_code_execution_loop

```

1 def llm_code_execution_loop(prompt, max_tries=5, filename="llm_generated.py"):
2     current_prompt = prompt
3     final_code = ""
4
5     for attempt in range(1, max_tries + 1):
6         print(f"==== Attempt {attempt} ===")
7         code = get_code_from_llm(current_prompt)
8         final_code = code
9         display(Markdown(f"```\n{code}\n```"))
10        success, error = run_generated_code(code, filename=filename)
11        if success:
12            print(f"Code in {filename} executed successfully.")
13            return True, final_code
14        else:
15            current_prompt = prompt + f"\n\nThe last run failed with this error:\n{error}\n\nPlease fix and regenerate valid Python code."
16
17    print(f"Failed after {max_tries} attempts.")
18    return False, final_code

```

Querying LLM to Download and Inspect GRIB2 Files

The following code was generated by prompting the LLM to create a script that:

- downloads a GRIB2 file from DWD's open data portal,
- decompresses it from '.bz2' format,
- lists the available 'shortName' entries using 'eccodes'.

LLM Prompt used:

Write Python code that:

- downloads the file https://opendata.dwd.de/weather/nwp/icon-d2/grib/00/t_2m/icon-d2_germany_i
- decompresses it to data.grib2
- prints all shortNames found in the file using eccodes

Use only valid Python code, and avoid markdown formatting.

Code generated by the LLM:

Download, Decompress, and List GRIB ShortNames

```

1 import urllib.request
2 import bz2

```

```

3 from eccodes import codes_grib_new_from_file, codes_get, codes_release
4
5 # Download the .bz2 file
6 url = "https://opendata.dwd.de/weather/nwp/icon-d2/grib/00/t_2m/icon-
      d2_germany_icosahehdral_single-level_2025051000_000_2d_t_2m.grib2.bz2"
7 bz2_filename = "data.grib2.bz2"
8 grib_filename = "data.grib2"
9
10 print(f"Downloading {url} ...")
11 urllib.request.urlretrieve(url, bz2_filename)
12 print(f"Saved to {bz2_filename}")
13
14 # Decompress to .grib2
15 with bz2.open(bz2_filename, "rb") as f_in, open(grib_filename, "wb") as f_out:
16     f_out.write(f_in.read())
17 print(f"Decompressed to {grib_filename}")
18
19 # List shortNames in the file
20 shortnames = set()
21 with open(grib_filename, "rb") as f:
22     while True:
23         gid = codes_grib_new_from_file(f)
24         if gid is None:
25             break
26         shortname = codes_get(gid, "shortName")
27         if shortname not in shortnames:
28             print("-", shortname)
29             shortnames.add(shortname)
30         codes_release(gid)

```

This process ensures that the file contains the expected variable (e.g. 2t for 2m temperature), which can then be visualized or processed further.

Providing a Template and Structuring the Prompt

To guide the LLM more precisely, we provide a working template and detailed expectations in the prompt. The following example generates a temperature plot from GRIB2 files.

Templated Prompt with Instructions

```

1 template = """
2 from eccodes import codes_grib_new_from_file, codes_get, codes_release
3
4 filename = "data.grib2"
5 shortnames = set()
6
7 with open(filename, "rb") as f:
8     while True:

```

```

 9     try:
10         gid = codes_grib_new_from_file(f)
11         if gid is None:
12             break
13         shortname = codes_get(gid, "shortName")
14         if shortname not in shortnames:
15             print("-", shortname)
16             shortnames.add(shortname)
17         codes_release(gid)
18     except Exception as e:
19         print("Error reading GRIB message:", e)
20         break
21 """
22
23 prompt = f"""
24 Now write a complete Python script using only eccodes to extract and plot
25     temperature data:
26
27 - Read from three GRIB2 files:
28   - clat.grib2 contains latitudes (shortName="tlat")
29   - clon.grib2 contains longitudes (shortName="tlon")
30   - data.grib2 contains 2m temperature (shortName="2t")
31
32 - Extract values with codes_get_array by shortName.
33 - Convert temperature to Celsius and clip values to [-10, 50].
34 - Plot using matplotlib and cartopy, adding rivers, borders, and land/sea mask.
35 - Save the image as 't2m.png'.
36
37
38 success, final_code = llm_code_execution_loop(prompt, max_tries=5, filename="
      plot_dwd.py")

```

Recommendation

While LLMs can generate useful code, they often produce errors or make incorrect assumptions. Always check the logic, verify the variables, and test the output. Treat LLMs as assistants — not as sources of guaranteed correct code. Often, you'll need to step in manually to guide them out of dead-end loops.

10.2 Survey of LLM-Based Agent Frameworks

Agent frameworks build on the capabilities of Large Language Models (LLMs) by giving them the ability to take actions, use tools, and follow plans. Such frameworks combine code generation with memory, execution environments, and goal-driven reasoning. In this section, we highlight and compare several influential frameworks and demonstrate how they can be used for code

automation.

Overview and Evaluation of AI Agent Frameworks

AI agent frameworks enable large language models (LLMs) to act, plan, and interact through structured workflows. They support tasks such as code generation, information retrieval, automation, and reasoning. Below, we summarize the most notable frameworks — both open-source and commercial — with a critical view on their maturity and practical usefulness.

Open-Source Frameworks

LangChain. A modular framework for building LLM-powered applications with memory, tools, agents, and prompt chaining. Widely adopted and relatively stable, though under rapid development — recommended for production with version control.

LangGraph. Graph-based extension of LangChain for stateful and branching workflows involving multiple agents. Promising for complex logic, but still young and best suited for advanced prototypes and research use.

Auto-GPT. Self-prompting agent that decomposes goals into actions and executes code via planning loops. Impressive demos but fragile in practice — not suitable for unattended use or production without strong supervision.

Llamaindex. Connects structured or unstructured data to LLMs via indexing and retrieval. Mature and widely used for RAG (retrieval-augmented generation) setups — production-ready.

BabyAGI. Minimal task loop with LLM-based task creation and prioritization. Fun for experimentation and educational use, but not robust for complex automation tasks.

CrewAI. Role-based multi-agent orchestration for collaborative problem solving. A good concept with some traction, but early-stage and still prone to runtime failures.

Semantic Kernel. Microsoft's framework to blend LLMs with traditional programming, memory, and planning. Solid enterprise potential and improving quickly, though not yet widely adopted.

Haystack. A mature NLP-first framework with integration for LLMs and search. Highly stable and suited for production QA systems with a strong retrieval backbone.

AgentGPT. Browser-based interface to configure and run autonomous agents. Useful for demonstrations, but lacks reliability and control mechanisms for real deployment.

XAgent. Research-oriented framework for advanced reasoning and tool usage. Conceptually strong, but primarily used in academic or benchmark settings.

Langroid. Supports communication between autonomous agents that collaborate on tasks. A niche framework for research into multi-agent dialogue, still evolving.

Flowise. No-code UI builder for chaining LLMs and APIs via a visual interface. Great for prototyping and education, but limited in flexibility for complex backend logic.

DSPy. Declarative LLM programming using supervised refinement. A highly promising academic tool, not yet adopted for production.

MetaGPT. Simulates software engineering teams with multiple role-based agents. Interesting for

demonstrating agent collaboration, but fragile and complex.

CAMEL. Creates conversational pairs of agents solving tasks via structured dialogue. Creative framework for simulation and experimentation, not for production.

TaskWeaver. Microsoft's code-first LLM agent system for automation and analysis. Well-structured, but still too new to evaluate maturity reliably.

GPT-Engineer. Generates entire software projects from specs with LLMs. Impressive when controlled by humans, but far from reliable without guidance.

Proprietary and Commercial Platforms

Claude Developer Platform. Anthropic's API for building assistant-like agents with safety and context limits. Stable and well-designed, though more restrictive than open-source counterparts.

OpenAI Assistants API. Enables assistants with tools, memory, and step-wise reasoning. Mature, secure, and production-ready — but bound to OpenAI's runtime and tool constraints.

Adept ACT-1. An agent designed to interact directly with real software via user interfaces. Demonstrated potential, but not publicly available or verifiable at scale.

Fixie.ai. Agent framework with tool plugins, memory, and integration APIs. Early-stage and cloud-dependent — limited control and flexibility.

Superagent. Platform to create and manage LLM agents with observability tools. Still developing, but usable in startups and internal projects.

Vercel v0. Turns natural language UI specs into React components. Useful for frontend development, not a general agent platform.

Devin (Cognition Labs). An LLM-based software engineer assistant capable of multi-step planning. Very promising, but still closed and demo-only.

Relevance AI. Cloud infrastructure for tracking and deploying agent workflows. Aimed at experimentation and analysis rather than core agent design.

Groq Agents. Infrastructure for real-time agents with ultra-fast inference hardware. Best for latency-sensitive use cases — not a full framework, but an enabler.

E2B. A cloud environment where LLM agents can safely execute code. Useful for isolated or test environments, but not yet standard in pipelines.

Note: Many frameworks above are open to experimentation but are not production-safe without deep understanding and control. We recommend thorough testing, validation, and version pinning before using agent frameworks in critical systems.

10.3 Using LangChain for Code Design and Execution

Why Use LangChain? — Integrating LLMs with Logic and Tools

LangChain is a modular framework that helps developers build applications powered by large language models (LLMs). It provides structured components to combine prompt templates, memory,

external tools, and reasoning capabilities in a consistent and reusable way.

Strengths of LangChain in Applied LLM Systems

- **Prompt Composition and Reuse.** LangChain provides prompt templates and chaining logic that make it easy to reuse LLM instructions systematically across applications.
Example: Create a prompt to generate a Python function, and reuse it in multiple workflows.
- **Integration with Tools and APIs.** LangChain allows LLMs to call Python functions, external APIs, file systems, and more through a standard interface.
Example: Let the LLM generate code, and use a tool node to run and evaluate it.
- **Memory and History Handling.** You can equip chains with memory modules that track previous messages or outputs, enabling context-aware agents.
Example: An LLM-based assistant that remembers earlier plots or variable names.
- **Flexible LLM Access.** LangChain supports OpenAI, Anthropic, Hugging Face, local models, and more — all through the same interface.
Example: Switch between ‘gpt-4o’ and a local LLaMA model with no code rewrite.
- **Rapid Prototyping and Modularity.** With its building-block design, LangChain is ideal for assembling and debugging LLM-based workflows step by step.

LangChain is well-suited for applications where LLMs need to be paired with logic, file access, or iterative computation — going beyond pure chat to interactive, semi-autonomous behavior.

Setting Up LangChain and OpenAI Access

LangChain works best with external LLM providers such as OpenAI. We recommend using the `python-dotenv` library to manage secrets.

Load Environment and Initialize LLM

```
1 from dotenv import load_dotenv
2 load_dotenv()
3
4 from langchain_openai import ChatOpenAI
5 llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
```

Simple Prompt Template and Execution

LangChain lets you construct reusable prompts. Below, we ask the LLM to generate a Python function for a polynomial within a given range.

Prompt Template to Generate Code

```
1 from langchain.prompts import PromptTemplate
2 from langchain.chains import LLMChain
```

```

3
4 prompt_func = PromptTemplate.from_template(
5     "Write a Python function named 'f(x)' that returns a polynomial expression "
6     "such that |f(x)| < 10 for x in [-10, 10]. Use numpy."
7 )
8
9 chain_func = LLMChain(llm=llm, prompt=prompt_func)
10 code_func = chain_func.invoke({}).content.strip()

```

Running and Validating the Generated Code

Once the code is generated, it is written to a file and executed. This allows fast iteration and testing.

Saving and Executing Generated Code

```

1 # Extract content and strip explanations if necessary
2 raw_content = code_func.content.strip()
3
4 # If it contains Markdown-style code fences, extract code block
5 if "```python" in raw_content:
6     code_lines = raw_content.split("```python")[1].split("```")[-1]
7 elif "```" in raw_content:
8     code_lines = raw_content.split("```")[-1]
9 else:
10     code_lines = raw_content # fallback: write as-is
11
12 # Save to file
13 with open("f_function.py", "w") as f:
14     f.write(code_lines.strip())
15
16 import traceback
17
18 # Try executing the generated function
19 try:
20     exec(open("f_function.py").read(), globals()) # defines f(x) globally
21     print("Function executed and loaded successfully.")
22 except Exception as e:
23     print("Error while executing the generated function:")
24     print(traceback.format_exc())

```

Visualizing and Interpreting Results

You can ask the LLM to generate visualization code. We use a second prompt to generate a plot based on the defined function.

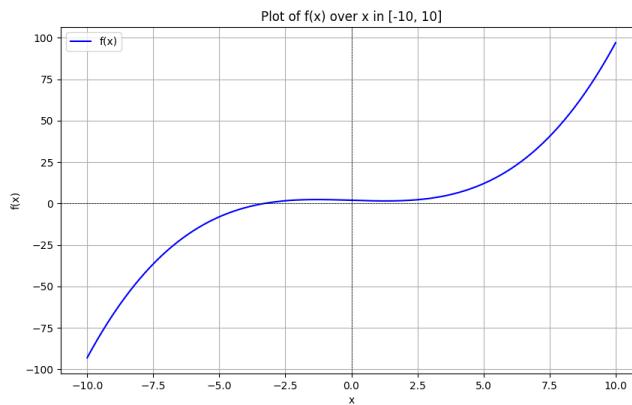


Figure 10.1: 1D plot of a polynomial $f(x)$ generated by a LangGraph code loop. The agent first created the function definition, verified it, and then produced this plot over the range $x \in [-10, 10]$.

Plotting a Generated Function Using LangChain

```

1 from langchain.prompts import PromptTemplate
2 from IPython.display import display, Image
3 from langchain_openai import ChatOpenAI
4 import os
5 import traceback
6
7 # Initialize model (adjust if already done elsewhere)
8 llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
9
10 # Prompt to generate a plotting script for f(x)
11 prompt_plot = PromptTemplate.from_template("""
12 Given a function f(x) which is defined and you can use -
13 do not define f(x) again. Instead,
14 write a Python script to plot f(x) over x in [-10, 10] using matplotlib.
15 Save the figure as 'plot.png'. Return python code only.
16 """)
17
18 # Generate response using the new LangChain syntax
19 response = (prompt_plot | llm).invoke({})
20
21 # Extract the content string from the AIMessage
22 raw_content2 = response.content.strip()
23
24 # Parse out code from markdown fences if needed
25 if "```python" in raw_content2:
26     code_lines2 = raw_content2.split("```python")[1].split("```")[-1]
27 elif "```" in raw_content2:
28     code_lines2 = raw_content2.split("```")[-1]
29 else:
30     code_lines2 = raw_content2 # fallback

```

```

31
32 # Save code to file
33 with open("plot_function.py", "w") as f:
34     f.write(code_lines2.strip())
35
36 # Try executing the generated plot code
37 try:
38     # Load the function definition into the global scope
39     exec(open("f_function.py").read(), globals())
40     exec(open("plot_function.py").read(), globals())
41     print("Plot executed successfully.")
42 except Exception:
43     print("Error during execution:")
44     print(traceback.format_exc())

```

We can also get much more out of the function easily.

Interpretation

```

1 prompt_summary = PromptTemplate.from_template(
2     "Given the following Python function definition:\n\n{code}\n\n"
3     "Describe the mathematical properties of this function (degree, max, min,
4     monotonicity)."
5 )
6
7 chain_summary = prompt_summary | llm
8 summary = chain_summary.invoke({"code": code_func.content})
9
10 from IPython.display import Markdown, display
11
12 # Print the function summary as Markdown
13 print("Function Summary:\n")
14 display(Markdown(summary.content))

```

Output is given e.g. by:

The function defined in the code is:

$$[f(x) = 0.1x^3 - 0.5x + 2]$$

Mathematical Properties:

Degree:

The function is a polynomial of degree 3, as the highest power of (x) is 3.
Maxima and Minima:

To find the local maxima and minima, we need to compute the first derivative of the function
and set it to zero: $[f'(x) = \frac{d}{dx}(0.1x^3 - 0.5x + 2) = 0.3x^2 - 0.5]$
Setting the first derivative to zero: $[0.3x^2 - 0.5 = 0 \implies 0.3x^2 = 0.5 \implies x^2 = \frac{0.5}{0.3} \implies x^2 = \frac{5}{3} \implies x = \pm \sqrt{\frac{5}{3}} \approx \pm 1.29]$

To determine whether these points are maxima or minima, we can compute the second derivative:

$$[f''(x) = \frac{d}{dx}(0.3x^2) = 0.6x]$$

Evaluating the second derivative at the critical points:

$$\text{For } (x = \sqrt{\frac{5}{3}}): [f''(\sqrt{\frac{5}{3}}) = 0.6\sqrt{\frac{5}{3}} > 0 \quad (\text{local minimum})]$$

$$\text{For } (x = -\sqrt{\frac{5}{3}}): [f''(-\sqrt{\frac{5}{3}}) = 0.6(-\sqrt{\frac{5}{3}}) < 0 \quad (\text{local maximum})]$$

Monotonicity:

The function is increasing or decreasing based on the sign of the first derivative ($f'(x)$):

For $(x < -\sqrt{\frac{5}{3}})$, ($f'(x) < 0$) (decreasing).

For $(-\sqrt{\frac{5}{3}} < x < \sqrt{\frac{5}{3}})$, ($f'(x) > 0$) (increasing).

For $(x > \sqrt{\frac{5}{3}})$, ($f'(x) > 0$) (increasing).

Therefore, the function has a local maximum at $(x = -\sqrt{\frac{5}{3}})$ and a local minimum at $(x = \sqrt{\frac{5}{3}})$.

Summary:

Degree: 3

Local Maximum: at $(x = -\sqrt{\frac{5}{3}})$

Local Minimum: at $(x = \sqrt{\frac{5}{3}})$

Monotonicity: Decreasing for $(x < -\sqrt{\frac{5}{3}})$, increasing for $(-\sqrt{\frac{5}{3}} < x < \sqrt{\frac{5}{3}})$, and increasing for $(x > \sqrt{\frac{5}{3}})$.

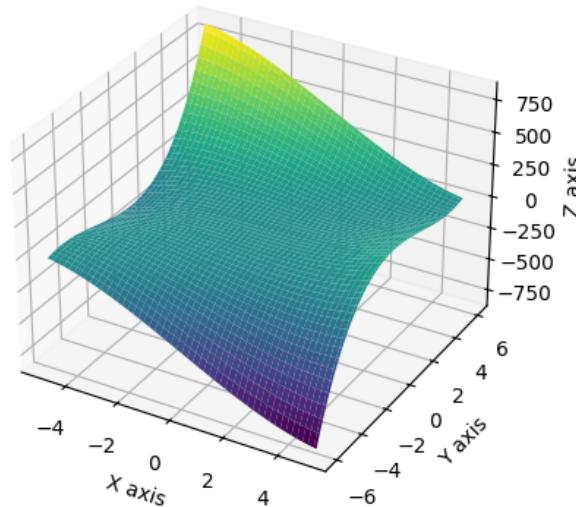


Figure 10.2: 2D surface plot of a generated function $f(x, y)$, created dynamically by a LangGraph agent. The LLM generated the code, saved it, and executed it to render this 3D surface using matplotlib.

Autonomous Code Execution with a Self-Correcting LLM Loop

To make our workflow more robust, we wrap the code generation and execution in a loop that automatically retries in case of errors. This allows us to turn LangChain into a simple coding agent that iteratively improves its output.

Self-Correcting Code Agent Loop

```

1 import traceback
2
3 def autonomous_code_agent(prompt, llm, filename="generated_code.py", max_attempts=5):
4     from langchain.prompts import PromptTemplate
5     from langchain.chains import LLMChain
6
7     template = PromptTemplate.from_template(prompt)
8     chain = LLMChain(llm=llm, prompt=template)
9
10    current_prompt = ""
11    for attempt in range(1, max_attempts + 1):
12        print(f"\n--- Attempt {attempt} ---")
13        result = chain.invoke({"input": current_prompt})
14        code = result.content.strip()
15
16        with open(filename, "w") as f:
17            f.write(code)
18
19        try:
20            print("Executing...")
21            exec(open(filename).read())
22            print("Success.")
23            return code
24        except Exception as e:
25            error_msg = traceback.format_exc()
26            print("Error:\n", error_msg)
27            current_prompt += f"\n\nThe last attempt failed with this error:\n{error_msg}\nPlease fix it and regenerate working Python code."
28
29    print("Failed after max attempts.")
30    return None

```

Running the Code Agent to Create a Function

```

1 prompt_func = (
2     "Write a Python function f(x) using numpy that defines a polynomial. "
3     "Ensure that |f(x)| stays below 10 for x in [-10, 10]. Name the function f."
4 )
5
6 code = autonomous_code_agent(prompt_func, llm, filename="f_function.py")

```

Sanity Check: Test the Function

```

1 exec(open("f_function.py").read())
2
3 import numpy as np
4 print("f(0) =", f(0))
5 print("f(5) =", f(5))

```

You can repeat this approach with other tasks, such as plotting or file processing. This pattern is a simple example of a structured LLM agent loop that improves iteratively.

Conclusion: LangChain as a Structured Agent Layer

LangChain provides modular tools to build LLM-powered agents that mix language understanding with logic, plotting, and tool use. In contrast to direct API calls, it helps manage prompt templates, execution chains, and debugging more systematically.

10.4 LangGraph-Based Forecast Assistant

In this section, we demonstrate how LangGraph can be used to build an intelligent assistant that interprets user input, extracts weather-related information (like temperature forecasts), downloads data from official sources, and generates plots or numerical results. This is part of a broader approach combining Large Language Models (LLMs) with structured tool execution.

LangGraph is a library that enables graph-based control flow for LLM applications. Each node in the graph represents a computational or reasoning step, while the edges define the order in which steps are executed. State is passed and updated along the way.

LangGraph Execution Flow for Weather Forecast Assistant

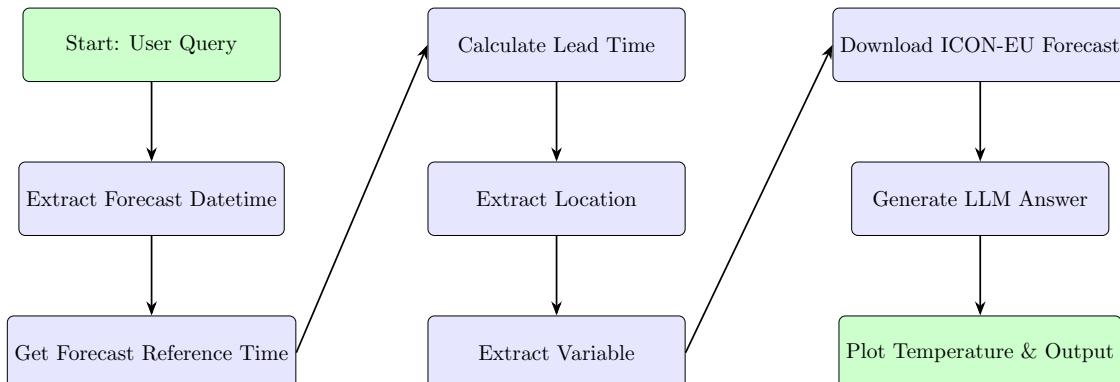


Figure 10.3: Execution flow of the LangGraph-based weather forecast assistant. Each node represents a processing step, starting from the natural language query down to data download and result output.

Our assistant interprets queries like: *Please give me the temperature forecast for Monday 4pm in Berlin.* It then carries out the following steps, defined by the langgraph graph as visualized in Figure 10.3:

1. LLM: Extracts the forecast date and time from the text.
2. Fkt: Retrieves the latest forecast reference time from DWD.
3. Fkt: Calculates the lead time in hours.
4. LLM: Extracts the location and variable (e.g., “temperature”).
5. Fkt: Downloads the corresponding ICON-EU GRIB2 data file.
6. LLM: Generate answer to user.
7. Fkt: Plots the temperature field.

LangGraph State Definition

LangGraph requires an explicit state schema. A state schema defines the structure of the data (or “state”) that flows through the graph. Each node in the graph receives the state as input, performs some computation, and returns an updated version of the state. The schema ensures that all nodes have a consistent view of what data is available, what fields are expected, and what can be modified or added during execution. We use Python’s TypedDict to define all fields that may be shared or updated between graph steps.

python

```

1 class MyState(TypedDict):
2     query: str
3     task: str
4     output: str
5     fc_datetime: str
6     fc_reference_datetime: str
7     fc_leadtime: str
8     fc_plot_option: str
9     fc_location_of_interest: str
10    fc_variable: str
11    fc_lat: str
12    fc_lon: str
13    temperature_data: Any
14    code_task: str
15    code_full: str
16    code_output: str
17    info_topic: str

```

Step 1: Extracting Forecast Datetime

We use the OpenAI model or a local llama version to extract a specific forecast time from a natural language query. The current system time is included in the prompt to improve accuracy.

python

```

1 def extract_forecast_datetime(user_input: str) -> str:
2     current_time = datetime.datetime.now().strftime("%Y-%m-%d %H")
3     prompt = f"""
4     The current time is: {current_time}.
5     Extract the date and time mentioned in the following user input
6     in format %Y-%m-%d %H.
7     User input: {user_input}
8     Forecast datetime:
9     """
10    response = llm.invoke(prompt)
11    return response.content.strip()

```

Step 2: Calculating Lead Time

Once we know both the reference time and forecast target time, we calculate the lead time in hours (rounded to nearest 3 hours if needed).

python

```

1 def calculate_forecast_lead_time(forecast_datetime: str, reference_datetime: str)
2     -> int:
3     forecast_time = datetime.datetime.strptime(forecast_datetime, "%Y-%m-%d %H")
4     reference_time = datetime.datetime.strptime(reference_datetime, "%Y-%m-%d %H")
5     lead_time = (forecast_time - reference_time).total_seconds() / 3600
6     return int(lead_time)

```

Step 3: Downloading ICON-EU Data

We download the GRIB2 forecast file for the corresponding forecast hour. The hour is zero-padded (e.g., “087”) and matched using a regular expression. Data is decompressed and loaded with cfgrib.

python

```

1 def download_icon_2m_temperature(fc_leadtime: str = "000") -> xr.DataArray:
2     fc_leadtime2 = f"{int(fc_leadtime):03d}"
3     url = "https://opendata.dwd.de/weather/nwp/icon-eu/grib/00/t_2m/"
4     response = requests.get(url)
5     soup = BeautifulSoup(response.text, "lxml")
6     pattern = re.compile(rf"icon-eu.*_{fc_leadtime2}_T_2M\.grib2\.bz2")

```

```

7   for link in soup.find_all("a"):
8       if pattern.match(link.get("href")):
9           filename = link.get("href")
10          break
11      bz2_path = "downloads/temp.grib2.bz2"
12      grib_path = "downloads/temp.grib2"
13      with open(bz2_path, "wb") as f:
14          f.write(requests.get(url + filename).content)
15      with bz2.open(bz2_path, 'rb') as f_in, open(grib_path, 'wb') as f_out:
16          f_out.write(f_in.read())
17      ds = xr.open_dataset(grib_path, engine="cfgrib", backend_kwargs={"indexpath":
18 : ""})
18      return ds["t2m"] - 273.15

```

Step 4: Building the LangGraph

Each of the above steps becomes a node in our LangGraph. The flow is fully defined with edges and a final step.

How the Graph Builder Works. LangGraph allows developers to define a sequence of processing steps (nodes) and connections (edges) between them in a directed computation graph. This is especially useful when building structured workflows that combine logic, LLMs, and functions. The key components are:

- **builder = create_graph()**
Initializes an empty computation graph. This object holds all nodes and edges you define.
- **builder.add_node("name", function)**
Registers a node in the graph with a unique name and an associated callable (e.g., a Python function, LangChain Runnable, or LLM chain). The node will process input and optionally update shared state.
- **builder.add_edge("source", "target")**
Connects two nodes, specifying that once source finishes execution, its result should be passed to target.

During execution, LangGraph processes nodes as follows:

1. Starts from the initial node (e.g., "start").
2. Passes a shared state (a Python dictionary) from node to node along the defined edges.
3. Executes the function or LLM associated with each node.
4. Updates the state with results from each step.
5. Terminates once a final node is reached or the graph ends.

This mechanism allows combining LLM calls, parsing functions, data downloads, and visualizations into a reproducible and inspectable pipeline.

python

```

1 # Define the LangGraph schema and state
2 builder = StateGraph(state_schema=MyState)
3
4 # Add nodes (ohne output_keys)
5 builder.add_node("extract_forecast_datetime", extract_forecast_datetime_node)
6 builder.add_node("get_latest_forecast_reference_time",
7     get_latest_forecast_reference_time_node)
8 builder.add_node("calculate_lead_time", calculate_lead_time_node)
9 builder.add_node("extract_location", extract_location_node)
10 builder.add_node("extract_variable", extract_variable_node)
11 builder.add_node("download_temperature", download_temperature_node)
12 builder.add_node("plot_temperature", plot_temperature_node)
13 #builder.add_node("get_coordinates", get_coordinates_from_name_node)
14 #builder.add_node("interpolate_temperature", interpolate_temperature_node)
15
16 # Set entry and finish points
17 builder.set_entry_point("extract_forecast_datetime")
18 builder.add_edge("extract_forecast_datetime", "get_latest_forecast_reference_time"
19     )
20 builder.add_edge("get_latest_forecast_reference_time", "calculate_lead_time")
21 builder.add_edge("calculate_lead_time", "extract_location")
22 builder.add_edge("extract_location", "extract_variable")
23 builder.add_edge("extract_variable", "download_temperature")
24 builder.add_edge("download_temperature", "plot_temperature")
25 #builder.add_edge("download_temperature", "get_coordinates")
26 #builder.add_edge("get_coordinates", "interpolate_temperature")
27 builder.set_finish_point("plot_temperature")
28 #builder.set_finish_point("interpolate_temperature")
29 graph = builder.compile()

```

Execution

The assistant is triggered by calling `ai(query)`, which sets up the state and runs the graph.

python

```

1 def ai(query):
2     # Initialize state
3     state = {
4         "query": query,
5         "task": "fc",
6         "output": "",
7         "fc_datetime": "",
8         "fc_reference_datetime": "",

```

```
[3]: dawid.ai("Please give me the temperature forecast for Monday 4pm.")

extract_forecast_datetime: 2026-01-01 16
get_latest_forecast_reference_time: 2025-12-31 00
calculate_forecast_lead_time: 40.0
extract_location: Location: Not specified
extract_variable: Temperature
Downloading: https://opendata.dwd.de/weather/nwp/icon-eu/grib/00/t_2m/icon-eu_europe_regular-lat-lon_single-level_2025123100_040_T_2M.grib2.bz2
Plot saved as temperature_forecast.png
```

I've created and displayed the temperature forecast for Monday at 4 PM. If you have a specific location in mind, feel free to let me know for more detailed information!

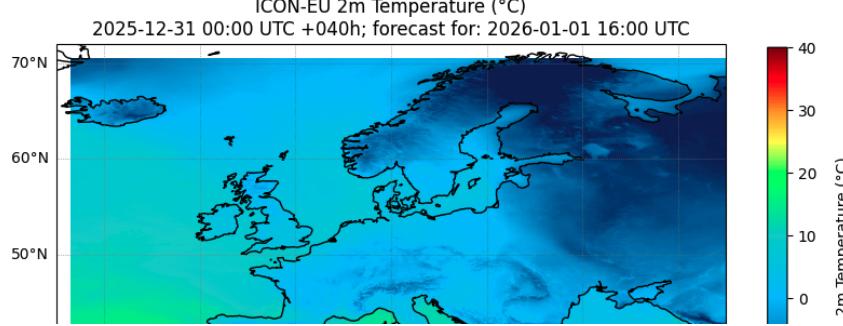


Figure 10.4: Weather forecast visualization generated by a LangGraph workflow. The agent pipeline extracted the user's query, downloaded DWD ICON data, interpolated values, and created a temperature map using Cartopy.

```

9      "fc_leadtime": "",  

10     "fc_plot_option": "full",  

11     "fc_location_of_interest": "",  

12     "fc_variable": "",  

13     "temperature_data": None,  

14     "code_task": "",  

15     "code_full": "",  

16     "code_output": "",  

17     "info_topic": ""  

18   }  

19  

20   # Run the LangGraph  

21   result = graph.invoke(state)  

22  

23   # Generate a summary using the LLM  

24   summary_prompt = f"""  

25   You are DAWID, the friendly assistant of Deutscher Wetterdienst. The user  

26   asked: "{state["query"]}"  

27  

28   Intermediate results:  

29   - Forecast datetime: {result['fc_datetime']}
```

```
33
34     Final output: {result['output']}
35
36     Summarize this in one or two friendly sentences for the user.
37     """
38
39     response = llm.invoke(summary_prompt)
40     display(Markdown(response.content))
```

Example:**python**

```
1 ai("Please give me the temperature forecast for Sunday 4pm in Berlin.")
```

This triggers all graph nodes in order, downloads and processes the forecast, and finally plots or returns interpolated values.

LangGraph provides an approach to compose toolchains driven by LLMs, where each step is modular, inspectable, and testable — ideal for modular workflows and AI-based assistants.

Chapter 11

DAWID, LLMs and Feature Detection

We show the design and setup of the LLM AI interface and framework DAWID, integrating AI/ML based services, specific knowledge, functions and data with an LLM based chatbot interface.

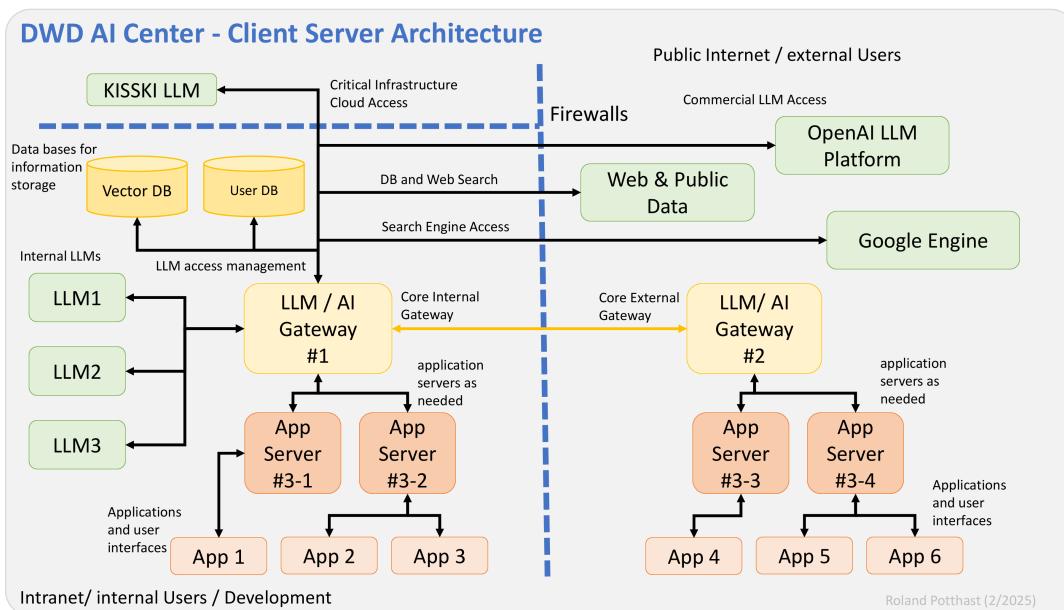


Figure 11.1: Client-server architecture for the AI Centre assistant system. The user interface connects to a backend server, which manages state, LLM calls, tool execution, and file storage. This structure enables interactive AI-driven workflows with persistent memory and modular tools.

11.1 The DAWID Frontend: Upload and Interaction Interface

The DAWID system provides a lightweight web interface for uploading data, sending user queries, and receiving processed responses. It is built using **HTML** plus **PHP**, styled with **CSS**, and controlled by **JavaScript** for asynchronous upload and interaction behavior.

Main Interface Structure

The main page of the frontend is implemented in `index.php`, which includes the HTML layout for the upload and chat areas, links to the CSS style sheet and JavaScript functionality, and containers to display request and response output.

Excerpt from `index.php`

```

1 <form id="uploadForm" method="post" enctype="multipart/form-data">
2   <input type="file" id="uploadFile" name="uploaded_file">
3   <div id="upload-result"></div>
4 </form>
5
6 <div id="request" class="query-heading">Waiting for request...</div>
7 <div id="response-container">
8   <div id="response">Awaiting reply...</div>
9 </div>
```

This HTML provides an upload form that will automatically submit when a file is selected. The response areas ('request', 'response') are updated via JavaScript.

Asynchronous Upload: JavaScript Logic

The file `dawid_upload.js` automatically submits the form once a file is picked and uses the Fetch API to send it to the backend PHP script. The status messages are displayed in the frontend.

Automatic Upload via JavaScript

```

1 fileInput.addEventListener("change", function() {
2   if (fileInput.files.length) {
3     uploadForm.requestSubmit(); // Auto-submit form
4   }
5 });
```

Submit Handler and Server Interaction

```

1 uploadForm.addEventListener("submit", function (e) {
2   e.preventDefault();
3   const formData = new FormData();
4   formData.append("uploaded_file", fileInput.files[0]);
5
6   fetch("upload_handler.php", {
7     method: "POST",
8     body: formData,
9   })
10  .then(response => response.json())
11  .then(data => {
12    uploadResult.innerText = "Upload result: " + data.message;
```

```
13      });
14  });

```

Interactive JavaScript for Streaming and Sessions

Beyond file uploads, DAWID's frontend JavaScript also enables live streaming of LLM responses and manages user interaction sessions. This enhances responsiveness and enables a smooth user experience with real-time feedback.

Streaming LLM Responses to the User. The JavaScript in `display.js` handles streaming output from a backend Python process. It reads text chunks from the response as they arrive and updates the display accordingly.

Streaming Output Display Logic

```
1 const responseDiv = document.getElementById("response");
2 const requestDiv = document.getElementById("request");
3
4 function streamResponse(url, body) {
5     fetch(url, {
6         method: "POST",
7         headers: { "Content-Type": "application/json" },
8         body: JSON.stringify(body)
9     })
10    .then(async response => {
11        const reader = response.body.getReader();
12        let partial = "";
13        while (true) {
14            const { done, value } = await reader.read();
15            if (done) break;
16            const chunk = new TextDecoder().decode(value);
17            partial += chunk;
18            responseDiv.innerHTML = marked.parse(partial);
19        }
20    });
21 }
```

Streaming Display Logic. The `streamResponse` function implements real-time display of responses from a streaming LLM backend. It reads data chunks from the server using a `ReadableStreamDefaultReader`, which is part of the Streams API built into modern web browsers, specifically part of the WHATWG Streams Standard. It appends each chunk to a growing `partial` string.

After every chunk is received, the entire `partial` string is re-parsed using `marked.parse()` and set as the `innerHTML` of the output container. This ensures that the user sees an incrementally updated and fully formatted response as it arrives. Although this design re-parses the entire response repeatedly, it offers a simple and robust way to render streamed Markdown text progressively without missing updates or introducing formatting errors.

This creates a live feedback loop that immediately shows partial results as they are streamed from

the server.

Session Control and Interaction History

User inputs and responses are stored in a browser-local session using a chat-style layout. The history is updated dynamically with each turn. This is complementary to backend storage.

Example: Appending Messages to the History

```

1 function appendToHistory(role, content) {
2   const block = document.createElement("div");
3   block.className = role + "-message";
4   block.innerHTML = marked.parse(content);
5   responseContainer.appendChild(block);
6 }
```

Each user message or system response is added as a new block, styled via CSS. This maintains continuity in the interaction and supports review or inspection by the user.

Together, these JavaScript modules turn the DAWID frontend into a responsive, real-time interface for LLM-driven services, managing uploads, sessions, streaming results, and user-friendly interactions.

Styling the Interface

The CSS file `dawid_style.css` defines the visual structure of the containers, input elements, and the response area.

Styling Example: Upload and Response Blocks

```

1 textarea, #response-container {
2   width: 90%;
3   max-width: 800px;
4   box-sizing: border-box;
5 }
6
7 .container {
8   width: 95%;
9   background: white;
10  padding: 20px;
11  border-radius: 10px;
12  box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
13 }
```

These CSS classes ensure the DAWID interface is clean, centered, and readable across browsers.

Frontend Summary

The DAWID frontend provides a responsive and extensible interface for interacting with AI-based backends. Its key features include:

- **Intuitive file upload** with immediate user feedback using asynchronous JavaScript.
- **Streaming of LLM responses** using the Fetch API and a readable stream reader, enabling real-time display of partial outputs.
- **Session and history handling**, allowing each user to maintain a visible context of their interaction.
- **Dynamic Markdown rendering** of incoming content using `marked.js`, continuously updating the output with every chunk.
- **Modular design** that can be easily integrated with a backend handling LLM interaction and a wide range of specific functions in a flexible way with the LangChain, LangGraph, or other LLM orchestration frameworks.

This frontend architecture supports real-time interactions, enhances usability for AI-based assistants, and lays the groundwork for agent-driven web interfaces.

11.2 DAWID Backend Architecture

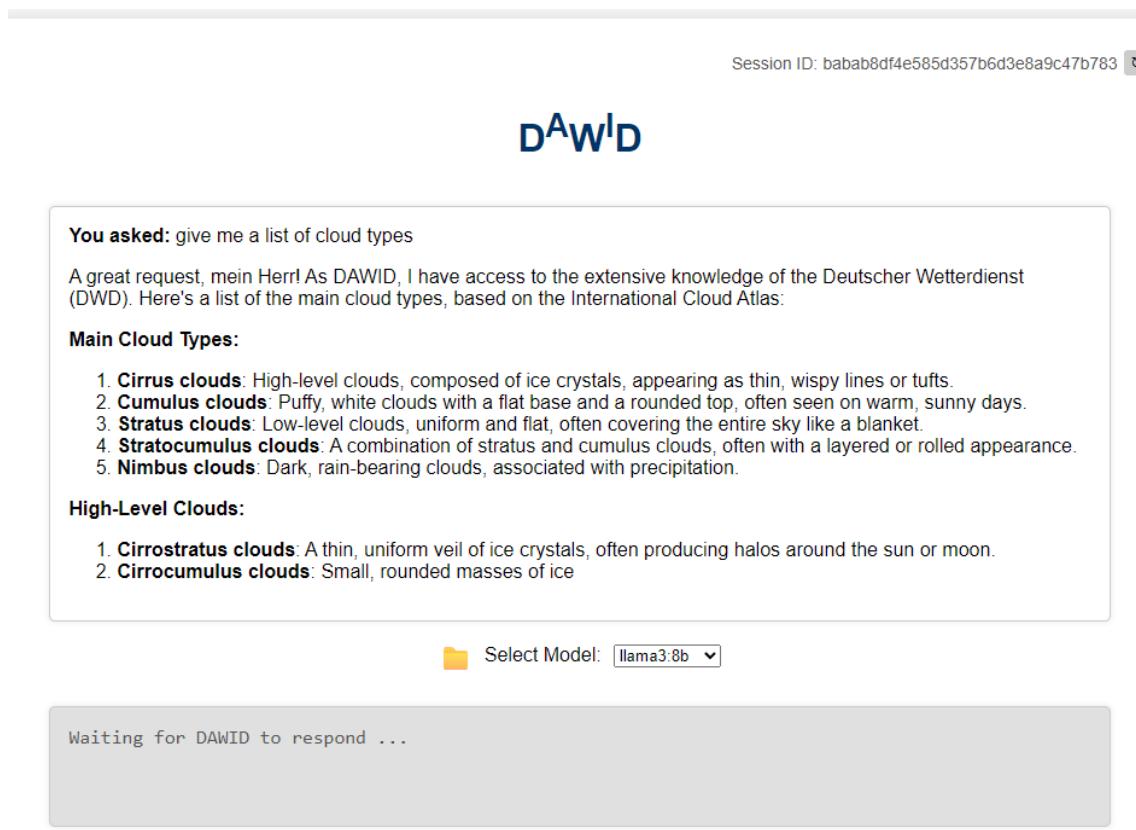
The DAWID backend is structured to provide a modular, extensible framework that connects a frontend assistant with powerful server-side components including local and remote language models, function execution, and LangGraph workflows.

Overview of Backend Components

The backend consists of several Python modules that work together:

- `dawid_server.py`: Main entry point for API calls from the frontend. Manages request routing, user session IDs, and streaming responses.
- `dawid_openai.py` and `dawid_llama.py`: LLM wrappers for calling OpenAI or local `llama.cpp`-based models.
- `dawid_users.py`: User management for accessing document spaces and functions.
- `dawid_functions.py`: Extracts structured function calls from LLM outputs (e.g., `get_current_weather`).
- `dawid_graphs.py`: Defines and executes LangGraph graphs based on available functions and user goals.

This modular design enables easy extension: new functions can be added, new LLMs integrated, and complex agent workflows configured with LangGraph.



Session ID: babab8df4e585d357b6d3e8a9c47b783 

DAWID

You asked: give me a list of cloud types

A great request, mein Herr! As DAWID, I have access to the extensive knowledge of the Deutscher Wetterdienst (DWD). Here's a list of the main cloud types, based on the International Cloud Atlas:

Main Cloud Types:

1. **Cirrus clouds:** High-level clouds, composed of ice crystals, appearing as thin, wispy lines or tufts.
2. **Cumulus clouds:** Puffy, white clouds with a flat base and a rounded top, often seen on warm, sunny days.
3. **Stratus clouds:** Low-level clouds, uniform and flat, often covering the entire sky like a blanket.
4. **Stratocumulus clouds:** A combination of stratus and cumulus clouds, often with a layered or rolled appearance.
5. **Nimbus clouds:** Dark, rain-bearing clouds, associated with precipitation.

High-Level Clouds:

1. **Cirrostratus clouds:** A thin, uniform veil of ice crystals, often producing halos around the sun or moon.
2. **Cirrocumulus clouds:** Small, rounded masses of ice

Select Model: 

Waiting for DAWID to respond ...

Figure 11.2: Example DAWID frontend session: The assistant responds to a query on cloud types, listing categories based on the International Cloud Atlas. The reply is formatted using Markdown and streamed in real time.

Session Handling and Routing

Each interaction is assigned a unique session_id. The server can distinguish multiple sessions, maintain short-term context, and route queries accordingly. This is crucial for multi-user operation and allows context-aware processing via LangGraph.

The main server logic is contained in dawid_server.py, which accepts user queries via HTTP POST, determines the model to use, and invokes the appropriate backend logic.

User Management, Document Folders, and Context Retrieval / Retrieval Augmented Generation (RAG)

DAWID implements a persistent user management system backed by an SQLite database. Each user is identified by a unique session or login ID and is associated with:

- a private document folder (docs/{user_id}),
- access to one or more group-shared folders (e.g., docs/shared/{group_name}),
- a vector store (faiss_index) built for each document folder.

This design allows both personalized and collaborative access to document content, enabling hybrid use cases such as private notes and shared projects.

When a user asks a question, the system performs a simple semantic analysis to decide whether document access is needed (e.g., for terms like based on the uploaded file, summarize this report, etc.), but also based on topics which are extracted from the document spaces and provided, such that topical search can trigger faiss retrieval and integration. If document access is inferred:

1. A FAISS similarity search is triggered within the appropriate user or group folder.
2. Relevant chunks are inserted into the LLM prompt as additional context.
3. The response is generated with awareness of both conversation state and retrieved content.

The document integration can be carried out based on a fast identification logic as conceptually shown in the following code block.

Topic-Aware Document Space Selection

```

1 # Mapping from document space to list of covered topics (user-defined)
2 USER_SPACES = {
3     "user/personal_notes": ["numerics", "python", "weather models"],
4     "group/satellite_team": ["radiative transfer", "cloud detection", "satellite
      channels"],
5     "group/forecasting": ["icon", "precipitation", "ensemble spread"]
6 }
7
8 # Determine which document spaces are relevant for the query
9 def topics_needed(query: str) -> list[str]:

```

```

10     relevant_spaces = []
11
12     for space, topics in USER_SPACES.items():
13         for topic in topics:
14             if topic in query.lower():
15                 relevant_spaces.append(space)
16                 break # one hit is enough for this space
17
18     return relevant_spaces
19
20 # Main document retrieval logic
21 def build_prompt_with_documents(query: str, user_id: str) -> str:
22     if detect_intent_fast_llm(query) != "doc_query":
23         return default_prompt(query)
24
25     relevant_spaces = topics_needed(query)
26     if not relevant_spaces:
27         return default_prompt(query)
28
29     # Retrieve context from each relevant FAISS index
30     context_blocks = [
31         faiss_retrieve(space, query) for space in relevant_spaces
32     ]
33     full_context = "\n\n".join(context_blocks)
34     return build_prompt_with_context(query, full_context)

```

LLM Abstraction and Streaming

The modules `dawid_openai.py` and `dawid_llama.py` implement streaming query interfaces:

Streaming a Query via OpenAI

```

1 def stream_openai_response(messages, model="gpt-4o-mini", temperature=0.2):
2     ...
3     for chunk in response:
4         if chunk.choices[0].delta.content:
5             yield chunk.choices[0].delta.content

```

Streaming a Query via llama.cpp Server

```

1 def stream_llama_response(messages, model="llama3"):
2     ...
3     async for chunk in response.aite_lines():
4         ...
5         yield token

```

This lets the frontend receive partial results in real-time — a major benefit for interactive user

interfaces.

Backend Language Model Infrastructure. For in-house LLM inference, DAWID currently operates a set of llama.cpp-based servers deployed on high-performance, CPU-optimized machines. These servers host various models (e.g., llama3:8b) and are selected at runtime by the DAWID backend for parallel load distribution. While the current implementation uses random selection, this mechanism is being extended to a dynamic load balancer that considers real-time system load and response latency. This ensures efficient use of compute resources and consistent user experience under varying workloads.

Function Extraction and Graph Control

After receiving a response from the LLM, dawid_functions.py inspects the output for structured function calls. These are expected either as JSON blocks:

Example Function Call in JSON

```

1 {
2   "function_calls": [
3     { "name": "get_current_datetime", "arguments": {} }
4   ]
5 }
```

If such a call is detected and valid (checked against AVAILABLE_FUNCTIONS), the call is extracted and removed from the LLM output before further processing.

LangGraph Execution

The LangGraph graph is built and run using the function `create_dawid_graph()` in `dawid_graphs.py`.

- The graph starts with `get_current_datetime`.
- It branches based on whether a weather function is also requested.
- Each node modifies the shared state dictionary.

Creating a LangGraph Flow

```

1 graph.add_node("get_current_datetime", get_current_datetime_node)
2 graph.add_conditional_edges("get_current_datetime", branch_logic, {
3   "get_current_weather": "get_current_weather",
4   "end": END
5 })
```

Extensible DAWID System

New tools, such as plotting, file downloads, or model-based simulation, can be added as functions. To do so:

1. Add the function name to AVAILABLE_FUNCTIONS in dawid_functions.py.
2. Implement the function node in dawid_graphs.py.
3. Extend the branching logic if needed.

This architecture allows DAWID to act as a semi-autonomous assistant with tool access, language model integration, and structured workflows, integrated with specific knowledge which can be made available to users by adding them to document spaces.

Privacy and Data Security

The DAWID backend is hosted entirely on internal DWD infrastructure and protected by a dedicated firewall, ensuring that user interactions and uploaded data remain strictly within the organization's secure network. User-specific data, including session history, uploaded files, and document-based context, is stored in user- and group-specific folders, isolated from external access.

Additionally, the in-house deployment of the llama3:8b language model ensures that queries processed via this model never leave the building. This enables secure, high-quality local inference with full data confidentiality — ideal for sensitive or internal-use cases.

Further security and privacy features:

- **FAISS vector search is local.** Any retrieval-augmented generation (RAG) or document search operations are carried out via local FAISS indices, which never transmit queries or data externally.
- **Session-based data control.** Each session is tied to a unique ID and limited in scope, allowing for controlled cleanup or persistence according to policy.
- **Optional OpenAI integration.** If the user selects an OpenAI model, the request is processed via the user's OpenAI Platform account and subject to OpenAI's privacy policy. The frontend allows full transparency and control over the selected model.
- **Logging and audit trails.** Interactions can optionally be logged locally for auditing, debugging, or research under strict access controls.
- **Future addition:** Fine-grained access rights to group folders and shared knowledge bases are implemented, with access through the DAWID user management.

This hybrid approach ensures that users benefit from local, private computation where needed while retaining the flexibility to use cloud-based services when desired.

Infrastructure and Secure Communication. The DAWID frontend is hosted on a professional-grade web server with a trusted TLS certificate, ensuring encrypted communication between users and the platform. Access to the backend services is tightly controlled: only the authorized Python

process running on the frontend server is permitted to communicate with the DAWID server's API port through an internal firewall. This isolation prevents unauthorized external access and ensures a secure and trusted environment for all interactions.

11.3 AI based Feature Detection for Fronts

This section describes a deep learning system for detecting weather fronts from meteorological input fields. The system includes both training and inference components and is designed to identify synoptic-scale structures based on surface-level atmospheric data. The approach leverages convolutional neural networks and standard PyTorch utilities, and is fully implemented in a modular set of Jupyter notebooks.

11.3.1 Overview and Objective

Fronts are transition zones between air masses with distinct temperature and humidity characteristics. Detecting them automatically from gridded analysis or forecast data is a challenging task due to their complex structure and temporal evolution.

The AI front detection system uses supervised learning to train a model on labeled frontal maps. The trained model is then applied to both analysis and forecast fields to detect fronts in operational-style datasets.

11.3.2 Data and Preprocessing

The input data for the front detection model consists of six meteorological fields extracted from numerical weather prediction model outputs (ICON GRIB files) on a regular latitude-longitude grid with approximately 0.4° horizontal resolution. The selected fields are:

- **PMSL**: Mean sea-level pressure (shortName = prmsl)
- **T2M**: 2-meter temperature (shortName = t2m)
- **RH2M**: 2-meter relative humidity (shortName = r2)
- **U10M**: 10-meter zonal wind component (shortName = u10)
- **V10M**: 10-meter meridional wind component (shortName = v10)
- **FRLAND**: Land-sea mask (binary 1/-1 from external GRIB file)

Each field is normalized channel-wise to a comparable numerical range in order to stabilize training. The normalization constants are determined from a large set of training data statistics:

Input Normalization

```

1 PMSL = (PMSL - 101280.7) / 1145.4
2 T2M  = (T2M  - 277.2)    / 14.2

```

```

3 RH2M = (RH2M - 79.0) / 14.1
4 U10M = (U10M - 0.8) / 5.1
5 V10M = (V10M - (-0.1)) / 5.2
6 FRLAND[FRLAND > 0] = 1
7 FRLAND[FRLAND == 0] = -1
8 ICON = np.stack([PMSL, T2M, RH2M, U10M, V10M, FRLAND], axis=0)

```

The resulting array has shape [6, lat, lon] and is stored in NetCDF files with dimensions var, lat, and lon. Each file corresponds to one analysis or forecast time step.

The corresponding labels used for supervised training are categorical 2D frontal maps, where each grid point belongs to one of several classes: background (no front), warm front, cold front, or occluded front. These labels are used in combination with a multi-class cross-entropy loss during model training.

11.3.3 Model Architecture

The front detection model is a U-Net architecture, a popular design for semantic segmentation tasks. It consists of a contracting path that extracts multi-scale features through convolutions and a symmetric expanding path that enables precise localization by combining upsampled features with high-resolution activations from earlier layers.

In this application, the U-Net accepts 6 input channels and outputs a 4-channel segmentation map, corresponding to different frontal types or background.

Model Architecture

```

1 # Load model from external repository or local file
2 model = unet.UNet(in_channels=6, out_channels=4, init_features=64)
3 model = model.to(device)

```

The model is optimized using the AdamW optimizer and categorical loss:

Loss and Optimizer

```

1 loss_fn = nn.CrossEntropyLoss().to(device)
2 optimizer = torch.optim.AdamW(params=model.parameters(), lr=0.0001)

```

This architecture enables high-resolution frontal feature detection by preserving spatial detail through skip connections and deep supervision.

11.3.4 Training Procedure

Training is handled by the notebook `aifronts_torch_training.ipynb`. The model is trained using categorical cross-entropy loss over multi-class front maps, with classes typically corresponding to warm, cold, occluded fronts, and background.

The training loop includes early stopping based on validation loss, online evaluation, and MLflow-based logging. Accuracy is computed per epoch using a helper function that compares predicted

classes with true labels.

Before training, model weights can optionally be restored from a previous checkpoint:

Optional Retraining

```
1 retrain = False
2 if retrain:
3     print("Loading existing model weights")
4     model.load_state_dict(torch.load("best-model-parameters.pt"))
```

Early stopping is controlled by a fixed patience and maximum number of epochs:

Training Parameters

```
1 best_avg_vloss = 1000 # Initial high value
2 stp            = 0      # Early stopping counter
3 stopafter       = 5      # Stop if no improvement after N epochs
4 max_epochs      = 200
```

The MLflow experiment is initialized before training begins. To avoid issues with DWD-internal proxies, proxy variables are explicitly removed from the environment:

MLflow Setup

```
1 for key in ("HTTP_PROXY", "http_proxy", "HTTPS_PROXY", "https_proxy"):
2     if key in os.environ:
3         del os.environ[key]
4
5 os.environ['MLFLOW_TRACKING_USERNAME'] = 'ffundel'
6 os.environ['MLFLOW_TRACKING_PASSWORD'] = 'mlflowffundel'
7 mlflow.end_run()
8 mlflow.set_tracking_uri("http://172.16.112.218:5051")
9 mlflow.set_experiment("aifronts_torch_felix")
10 mlflow.autolog()
11 run_name = datetime.today().strftime('%Y%m%d%H%M%S')
12 mlflow.start_run(run_name="unet-" + run_name)
```

The training loop proceeds epoch-wise, alternating between training and validation mode:

Epoch Loop

```
1 for epoch in range(max_epochs):
2     model.train()
3     for inputs, labels in train_loader:
4         inputs, labels = inputs.to(device), labels.to(device)
5         optimizer.zero_grad()
6         outputs = model(inputs)
7         loss = loss_fn(outputs, labels)
8         acc = accuracy(outputs, labels)
```

```

9      loss.backward()
10     optimizer.step()

```

After training, validation metrics are calculated without gradient tracking:

Validation Loop

```

1   model.eval()
2   with torch.no_grad():
3       for vinputs, vlabels in validation_loader:
4           vinputs, vlabels = vinputs.to(device), vlabels.to(device)
5           voutputs = model(vinputs)
6           vloss    = loss_fn(voutputs, vlabels)
7           vacc    = accuracy(voutputs, vlabels)

```

Training and validation metrics are recorded to the MLflow server after each epoch:

Metric Logging and Checkpointing

```

1   mlflow.log_metric("train_loss",      avg_loss,    step=epoch)
2   mlflow.log_metric("train_accuracy", avg_acc,     step=epoch)
3   mlflow.log_metric("eval_loss",       avg_vloss,   step=epoch)
4   mlflow.log_metric("eval_accuracy", avg_vacc,   step=epoch)
5
6   if avg_vloss < best_avg_vloss:
7       stp = 0
8       best_avg_vloss = avg_vloss
9       print("Saving current model")
10      torch.save(model.state_dict(), "best-model-parameters.pt")
11  else:
12      stp += 1
13
14  if stp >= stopafter:
15      break

```

This loop ensures that only the best-performing model (based on validation loss) is retained and that each run is reproducibly logged and tracked. The combination of early stopping, metric logging, and weight checkpointing ensures both efficiency and traceability.

11.3.5 Inference and Forecast Application

Inference is handled in the notebook `aifronts_torch_inference.ipynb`, which loads the trained U-Net model and applies it to an ICON forecast field sequence. Input fields are already normalized and stored as NetCDF files with shape [6, lat, lon]. These are loaded and concatenated over time using `xarray`.

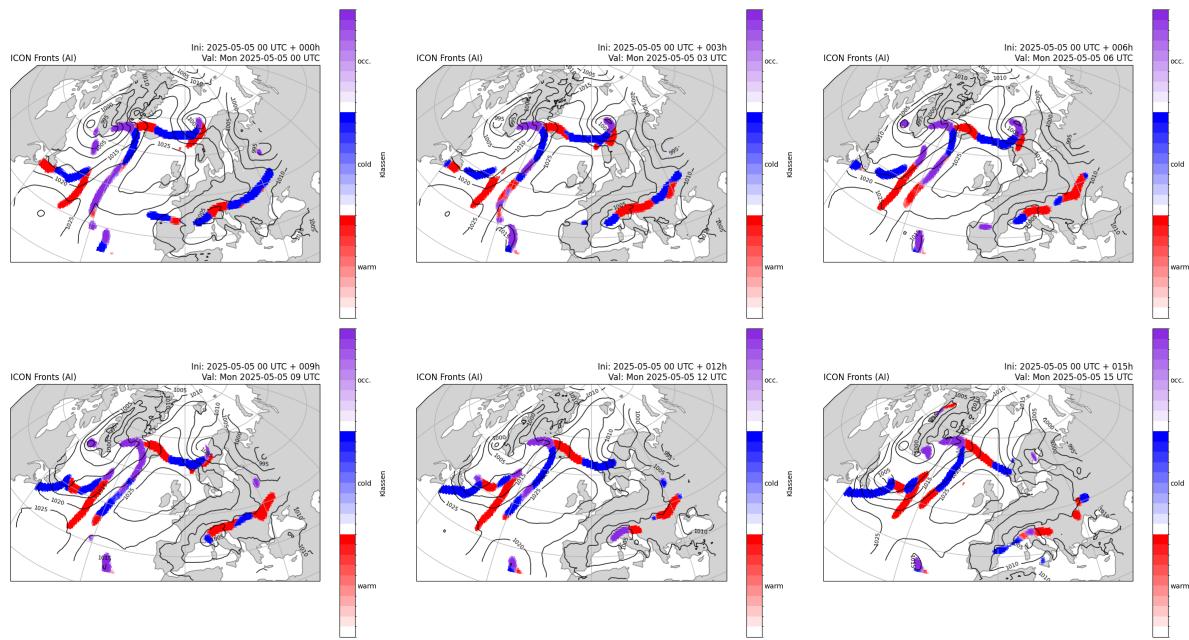


Figure 11.3: Detected fronts from the AI model applied to ICON forecast fields initialized on 5 May 2025 at 00 UTC. Each map shows one forecast lead time (from +00h to +15h in 3h steps). The color-coded front detections correspond to warm fronts (red), cold fronts (blue), and occluded fronts (purple). Overlaid black contours show the corresponding mean sea-level pressure (MSLP), helping to interpret the synoptic environment. The AI system provides consistent front structures that evolve smoothly over time, following cyclonic systems and frontal zones in the pressure field.

Load Normalized Forecast Input

```

1 FCPATH      = "/.../input/"
2 DATE        = "2025050500"
3 FILES       = sorted(glob.glob(FCPATH + DATE + '/*.nc'))
4 datasets    = [xr.open_dataset(f) for f in FILES]
5 combined_ds = xr.concat(datasets, dim='time')
6 Input        = torch.tensor(combined_ds['ICON'].values, dtype=torch.float)

```

The trained model is loaded and put into inference mode. It is then applied to the entire forecast sequence in one batch:

Run Inference

```

1 best_model = UNet(in_channels=6, out_channels=4, init_features=30)
2 best_model.load_state_dict(torch.load("best-model-parameters.pt"))
3 best_model.eval()
4 inference  = best_model(Input)

```

The model output is a 4-class probability map for each forecast time and grid point. This is postprocessed into a discrete field using argmax logic, with a special handling for the background (class 3):

Postprocessing to Class Index

```

1 fc          = np.array(inference.detach().numpy())
2 fc[:,3,:,:] = np.where(fc[:,3,:,:] > 0.95, fc[:,3,:,:], np.nan)
3 fc          = np.nanargmax(fc, axis=1)
4 fc          = np.where(fc == 3, np.nan, fc)
5 fc[fc == 0] = inference.detach().numpy()[:,0,:,:][fc == 0]
6 fc[fc == 1] = inference.detach().numpy()[:,1,:,:][fc == 1] + 1
7 fc[fc == 2] = inference.detach().numpy()[:,2,:,:][fc == 2] + 2

```

The resulting class predictions are combined with the original input MSLP field to create a joint array for visualization:

Create Output DataArray

```

1 coords = {'type': ['fc', 'ps'],
2            'case': range(57),
3            'lat': combined_ds['lat'].values,
4            'lon': combined_ds['lon'].values}
5 arr = xr.DataArray([fc, combined_ds["ICON"].values[:,0,:,:]], coords)

```

A plot loop then visualizes each lead time of the forecast. The predicted fronts are shown as colored maps overlaid with MSLP contours and geographic context using Cartopy:

Plot Forecast Fronts

```

1 for case in range(arr.shape[1]):
2     dt = datetime.strptime(os.path.basename(os.path.dirname(FILES[case])), '%Y%m%d%H')
3     vt = dt + timedelta(hours=case*3)
4
5     data        = arr[0, case, :, :]
6     contour_data = (arr[1, case, :, :] * 11.454) + 1012.807 # inverse
                                                               normalization
7
8     # Cartopy projection and layers
9     fig = plt.figure(figsize=(10, 8))
10    ax = plt.axes(projection=ccrs.Orthographic(0, 35))
11    ax.add_feature(cfeature.NaturalEarthFeature('physical', 'land', '110m',
12                                           facecolor='lightgray'), zorder=0)
13    ax.coastlines(zorder=0, color='gray')
14    ax.gridlines()
15
16    # Color mapping for front types
17    custom_cmap = build_custom_cmap() # red = warm, blue = cold, violet =
18                                         occluded
19    bounds = np.linspace(0, 3, 31)
20    norm = mcolors.BoundaryNorm(bounds, custom_cmap.N)

```

```

20     # Plot fronts and MSLP
21     mesh = ax.pcolormesh(Lon, Lat, data, cmap=custom_cmap, norm=norm, transform=
22         ccrs.PlateCarree())
23     contour_lines = ax.contour(Lon, Lat, contour_data, levels=np.arange(900, 1100,
24         5),
25                     colors='black', linewidths=1, transform=ccrs.
26         PlateCarree(), zorder=3)
27     ax.clabel(contour_lines, inline=True, fontsize=8)
28
29     # Titles and save
30     ax.set_title("Ini: " + dt.strftime('%Y-%m-%d %H UTC') + " + " + str(case*3).
31     zfill(3) + "h", loc="right")
32     plt.title("ICON Fronts (AI)", loc="left")
33     plt.savefig(f"plots/plot_{str(case).zfill(3)}.png")
34     plt.close(fig)

```

This inference process is repeated over the entire forecast time range (e.g., 57 time steps). The resulting maps as displayed in Figure 11.3 illustrate the temporal evolution of predicted front positions and their meteorological environment, enabling visual inspection or further automated verification.

Chapter 12

MLFlow – An easy way of Managing and Monitoring Training

12.1 Introduction to MLFlow

Machine learning workflows often involve repeated training runs with slightly different hyperparameters, code versions, and datasets. Without proper tracking, this becomes chaotic — especially in collaborative or operational environments. MLFlow provides a robust solution for managing this complexity by organizing experiments and tracking key information throughout the machine learning lifecycle.

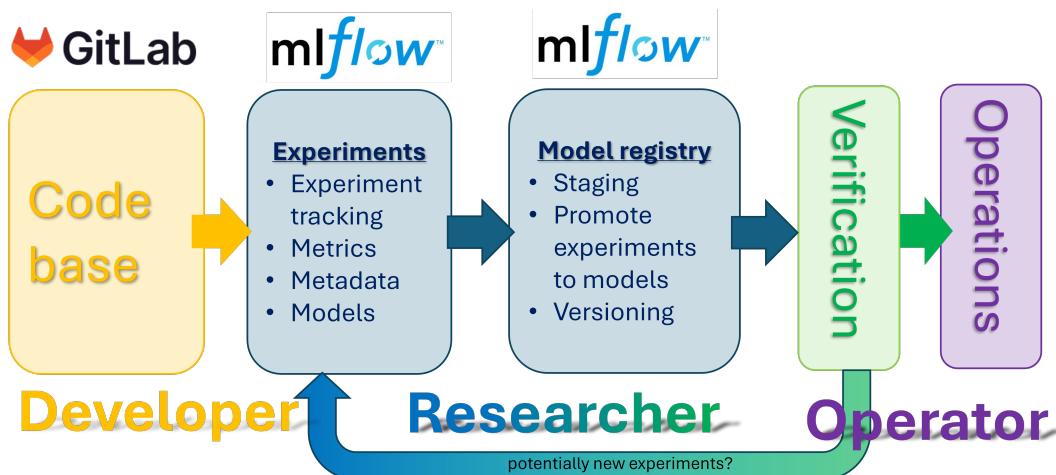


Figure 12.1: MLFlow in an MLOps pipeline: The workflow illustrates how developers, researchers, and operators interact through MLFlow. Experiments are tracked, metrics and models are stored, and selected runs can be promoted to the model registry. Verified models are deployed to operations, and outcomes can lead to new experiments.

12.1.1 What is MLFlow?

MLFlow is an open-source platform that addresses four primary functions of the machine learning workflow:

- **Tracking:** Log parameters, metrics, artifacts, and code versions.
- **Projects:** Package ML code in a reusable and reproducible form.
- **Models:** Manage and deploy different versions of trained models.
- **Recipes:** Automate and standardize training pipelines.

In this chapter, we focus mainly on the **Tracking** and **Model Management** capabilities.

Run Name	Created	Duration	Models
rfl5_h4_3h_2011_2021_val2022_clpprecmax	13 days ago	6.9h	-
rfl5_h4_3h_2011_2021_val2022_clpprec	13 days ago	-	-
rfl6_h5_3h_2011_2021_val2022_noclp	13 days ago	-	-
rfl6_h5_3h_2011_2021_val2022_noclp	13 days ago	-	-
rfl5_h4_3h_2011_2021_val2022_clprec	14 days ago	-	-
rfl5_h4_3h_2011_2021_val2022_clampstd	14 days ago	-	-
rfl5_h4_3h_2011_2021_val2022_clampstd	16 days ago	-	-
rfl5_h4_3h_2011_2021_val2022_noclamp	16 days ago	15.6d	AICON_prototype_v3

Figure 12.2: MLFlow Web UI displaying logged experiments, parameters, and metrics.

12.1.2 Why Do We Need Experiment Tracking?

The need for reproducibility and transparency in ML pipelines is growing. Consider the following recurring challenges:

- You forgot which dataset was used to train a model.
- The best-performing parameter settings are lost or overwritten.
- Colleagues cannot reproduce your results.
- Operational users request model metadata, but it is not documented.

MLFlow provides solutions to these problems by creating a structured history of all your training runs.

12.1.3 Core Concepts of MLFlow

MLFlow Tracking organizes your work using the Experiment-Run hierarchy:

- **Experiment:** A named collection of related training runs.
- **Run:** A single execution of training code, where parameters, metrics, and artifacts are logged.

Each run can store

- **(Hyper-)Parameter:** settings of the experiments such as numeric values and strings,
- **Metrics:** values such as losses and scores that can be logged with each training step and epoch,
- **System Metrics:** step/epoch-dependent metrics of the system such as CPU usage and memory usage, and
- **Artifacts:** files such as plots or trained models.

MLFlow offers an interactive user interface to plot and compare different metrics within and across different training runs (Figs. 12.3, 12.4, 12.5). The metrics can even be analysed while the training is still running.

Each run is recorded either locally or on a remote tracking server, and can be explored visually using the MLFlow Web UI. The tracking target is defined by the **Tracking URI**.

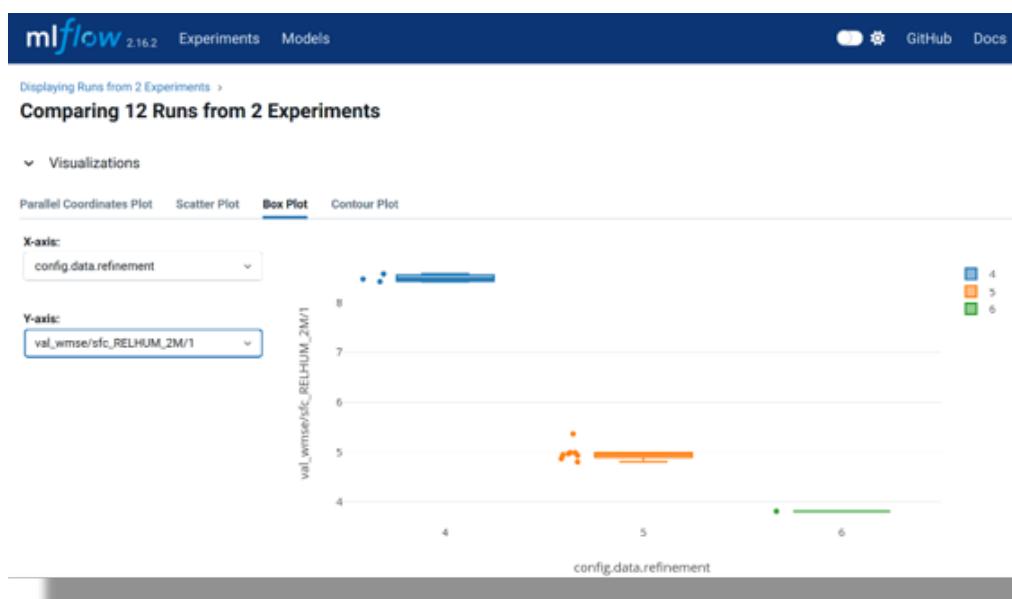


Figure 12.3: Box Plot Comparison in MLFlow: Model runs from different experiments are compared based on a validation metric, with color-coded configurations. This helps identify parameter influences.

12.1.4 Installing MLFlow

You can install MLFlow using pip:

Install MLFlow

```
1 pip install mlflow
```

It is recommended to use a virtual Python environment to isolate dependencies.

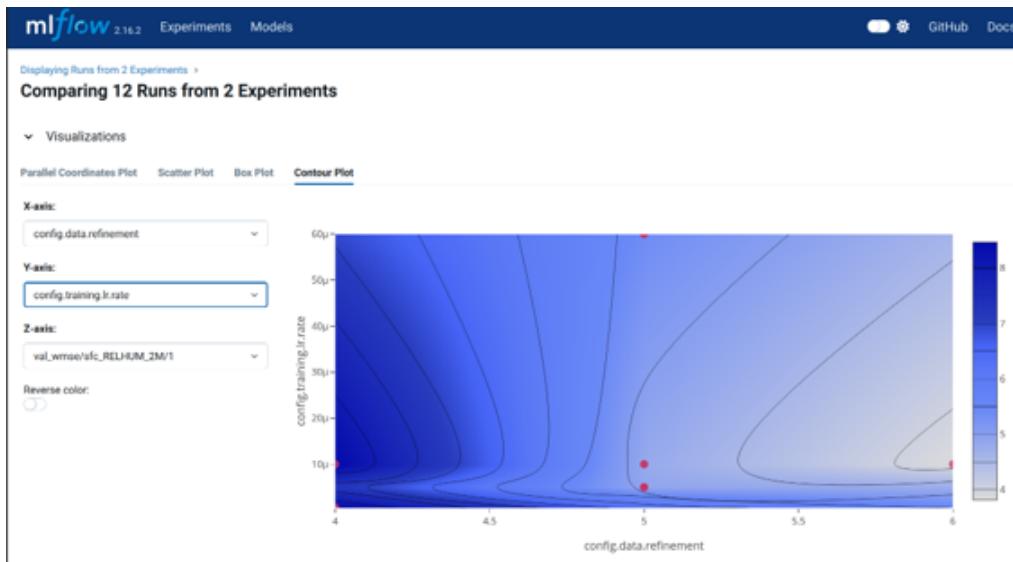


Figure 12.4: Contour Plot Visualization in MLFlow: Visualization of the effect of two hyperparameters on a performance metric. The color gradient highlights optimal regions for tuning.

12.1.5 Basic Logging Example

The following Python code demonstrates how to log an experiment using MLFlow:

Minimal MLFlow Logging

```
1 import mlflow
2
3 mlflow.set_experiment("MyFirstExperiment")
4
5 with mlflow.start_run(run_name="initial_run"):
6     mlflow.log_param("learning_rate", 0.001)
7     mlflow.log_metric("loss", 0.256)
```

This code snippet performs the following:

- Creates or reuses an experiment called MyFirstExperiment.

- Starts a run named `initial_run`.
- Logs a hyperparameter and a metric to that run.

12.1.6 Launching the MLFlow Web Interface

Once an experiment has been logged, you can start a local web interface with:

```
Start MLFlow UI
```

```
1 mlflow ui
```

By default, this opens a dashboard at `http://localhost:5000`, where you can explore experiments, compare runs, and visualize metrics or parameters.

12.1.7 Summary

MLFlow is a powerful tool to:

- Organize and record training runs
- Compare performance between experiments
- Make machine learning more reproducible
- Share and manage models in a structured way

In the following sections, we will explore how to use MLFlow to log a complete neural network training process and how to run an MLFlow server for team collaboration.

12.2 Logging ML Experiments with MLFlow

In this section, we demonstrate the use of MLFlow in a real training scenario. The example implements a neural network that learns to predict wind chill based on temperature and wind speed, and logs the experiment to an MLFlow server.

12.2.1 Preparing the Environment

Make sure MLFlow is installed and your server is running (local or remote).

```
Install MLFlow
```

```
1 pip install mlflow
```

Then configure the tracking URI and the experiment name.

Setup an Experiment

```

1 import mlflow
2
3 mlflow.set_tracking_uri(uri="http://localhost:5000")
4 mlflow.set_experiment("Wind Chill Example")

```

This sets up logging to a local MLFlow server and defines the experiment under which training runs are grouped.

12.2.2 Generating Synthetic Data

We simulate temperature and wind speed data, and compute wind chill values using a standard formula:

Generate Data

```

1 import numpy as np
2 import torch
3
4 tt = np.random.uniform(-20, 10, 500)    # Temperature in Celsius
5 ff = np.random.uniform(0, 50, 500)        # Wind speed in km/h
6
7 wc = 13.12 + 0.6215 * tt - 11.37 * (ff ** 0.16) + 0.3965 * tt * (ff ** 0.16)
8
9 x_train = torch.tensor(np.column_stack((tt, ff)), dtype=torch.float32)
10 y_train = torch.tensor(wc, dtype=torch.float32).view(-1, 1)

```

This data is used to train a regression model.

12.2.3 Defining the Model

We use a simple feedforward neural network with two hidden layers:

Wind Chill Model

```

1 import torch.nn as nn
2
3 class wind_chill_model(nn.Module):
4     def __init__(self, hidden_dim):
5         super().__init__()
6         self.fc1 = nn.Linear(2, hidden_dim)
7         self.fc2 = nn.Linear(hidden_dim, hidden_dim)
8         self.fc3 = nn.Linear(hidden_dim, 1)
9         self.relu = nn.ReLU()
10
11     def forward(self, x):
12         x = self.relu(self.fc1(x))    # layer 1

```

```

13         x = self.relu(self.fc2(x))  # layer 2
14     return self.fc3(x)

```

12.2.4 Training the Model with MLFlow Logging

The training process logs parameters and metrics using MLFlow.

Train and Log

```

1 hidden_dim = 20
2 model = wind_chill_model(hidden_dim=hidden_dim)
3 criterion = nn.MSELoss()
4 optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
5
6 with mlflow.start_run(run_name="logging_01"):
7     mlflow.log_param("hidden_dim", hidden_dim)
8
9     for epoch in range(10000):
10         model.train()
11         optimizer.zero_grad()
12         y_pred = model(x_train)
13         loss = criterion(y_pred, y_train)
14         loss.backward()
15         optimizer.step()
16
17         if (epoch + 1) % 500 == 0:
18             mlflow.log_metric("loss", loss.item(), step=epoch)

```

The log includes both parameter settings and evolving loss values. To harness the full potential of MLflow, one should log all metrics including lr in this example.

12.2.5 Logging Plots and the Final Model

After training, we log a loss curve and the final model.

Log Artifacts

```

1 import matplotlib.pyplot as plt
2
3 plt.plot(train_loss, label="training loss")
4 plt.yscale('log')
5 plt.xlabel("epoch")
6 plt.ylabel("loss")
7 plt.legend()
8 plt.tight_layout()
9 mlflow.log_figure(plt.gcf(), "figure.png")
10

```

```

11 from mlflow.models import infer_signature
12 signature = infer_signature(x_train.numpy(), model(x_train).detach().numpy())
13 mlflow.pytorch.log_model(model, "model", signature=signature)

```

Artifacts such as loss curves and model binaries appear in the MLFlow UI and can be downloaded or registered.

12.2.6 Results in the MLFlow UI

You can explore the run in the web interface while its running or afterwards. Just open your browser, type localhost:5000 and you will be able to inspect your example. The loss curve is visible under the *Metrics* tab, and the model is listed under *Artifacts*. Depending on the MLFlow version, there is a metrics tab either at the main window or after you chose the experiments.

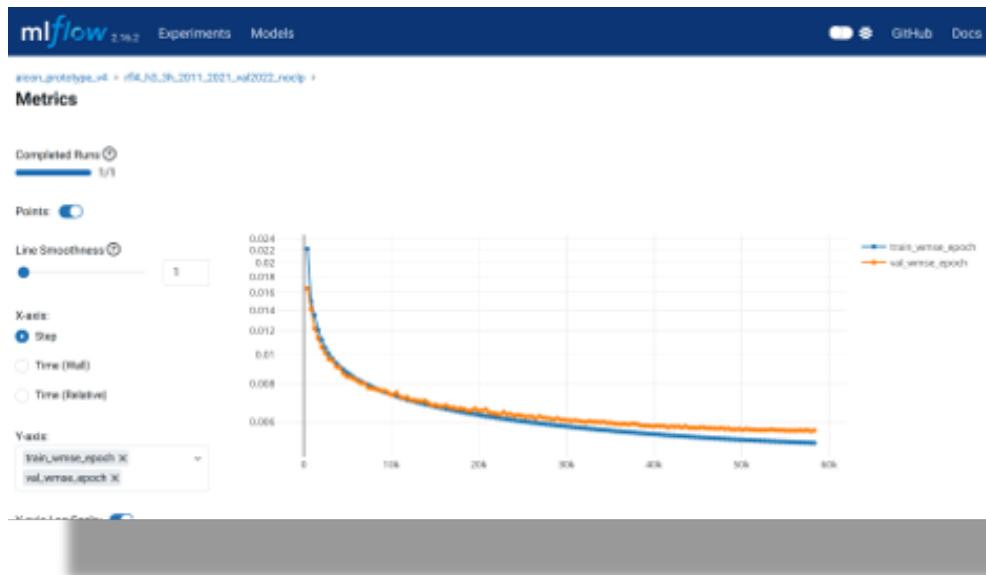


Figure 12.5: MLFlow Metrics View: The training and validation losses over time are automatically visualized.

12.2.7 Summary

In this section, we demonstrated:

- Creating a synthetic dataset and training a model
- Logging model parameters, metrics, and artifacts with MLFlow
- Visualizing results through the MLFlow UI

This forms the basis for more complex tracking and collaborative workflows, which we explore in the next section.

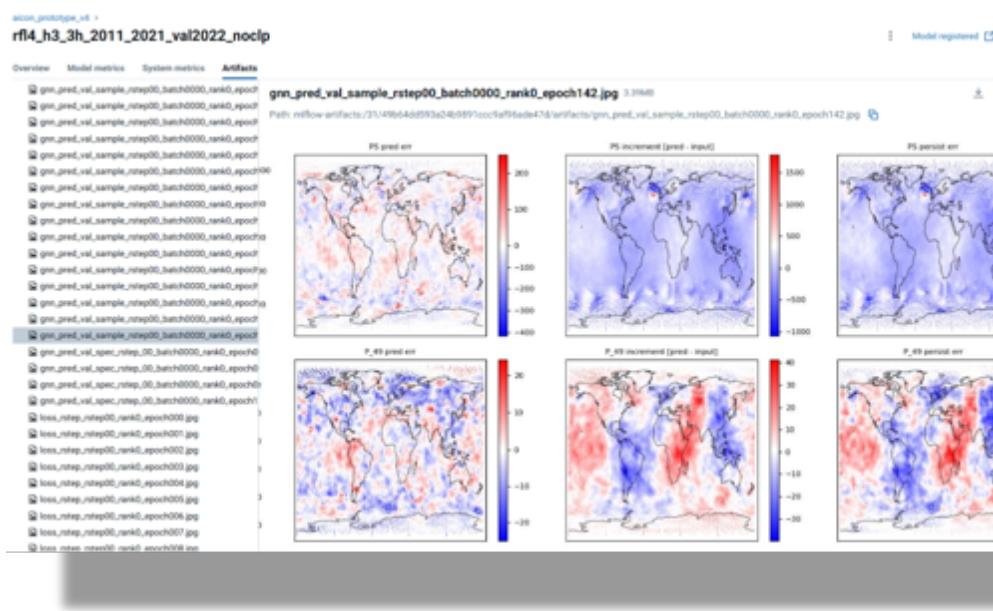


Figure 12.6: MLFlow Artifact Browser: Logged outputs such as loss plots and models appear here and can be downloaded.

12.3 Running an MLFlow Server

While local logging to a directory is useful for experimentation, a shared MLFlow server is essential for team collaboration. In this section, we show how to start MLFlow in both local and multi-user network modes, and explain credential setup and deployment options.

12.3.1 Starting the Local Web UI

You can launch the built-in web interface with a simple command:

Start MLFlow UI

```
1 mlflow ui
```

This opens a browser window at `http://localhost:5000` displaying all logs from the default `mlruns/` directory.

12.3.2 Launching a Stand-alone Server

To share results across users or machines, launch MLFlow in **server mode**. So far, port 5000 was restricted to users on the local system. The following command binds MLFlow to all network interfaces so others can access your MLFlow server via the network.

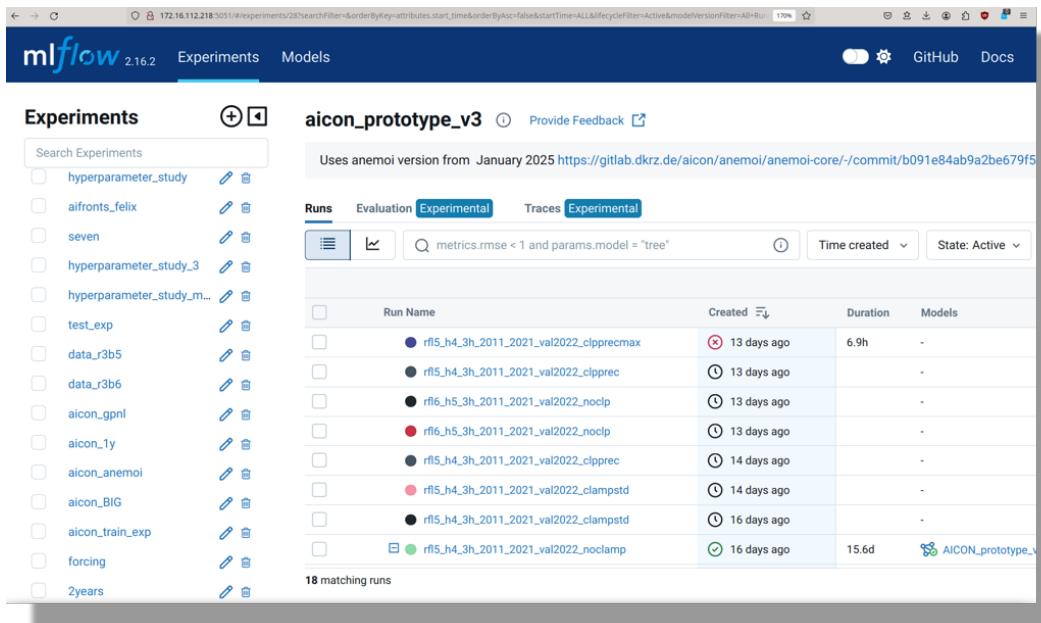


Figure 12.7: MLFlow Web UI in local mode: Logs from `m1runs/` are visualized through a local server.

Start MLFlow Server

```
1 mlflow server --host 0.0.0.0 --port 5000
```

To keep the MLflow server running after starting it, you can launch it in the background using standard shell tools:

Shell background execution

```
1 mlflow server --host 0.0.0.0 --port 5000 &
```

This appends an `&`, sending the process to the background. However, it will stop if the terminal is closed. To make the server survive logout or disconnection, use `nohup`:

Detached execution using nohup

```
1 nohup mlflow server --host 0.0.0.0 --port 5000 > mlflow.log 2>&1 &
```

This redirects all output to `mlflow.log` and keeps the process running independently of your shell session.

12.3.3 Using Screen for Detached Execution

`screen` is a terminal multiplexer that allows you to run processes in a separate session, which remains active even after you close your terminal. It is especially useful for long-running tasks like the MLflow server, where you may want to reconnect later to monitor logs or restart the process.

The following Bash script starts the MLFlow server inside a named GNU screen session.

Run MLFlow In Screen

```
1 #!/usr/bin/env bash
2 screen -dms mlflow
3 screen -S mlflow -X stuff "mlflow server --host 0.0.0.0 --port 5000\r"
```

This allows the process to run in the background and be detached/re-attached.

12.3.4 Credential and Authentication Setup

MLFlow supports experimental basic authentication. You might need to pip install the additional auth dependencies. Define users in an auth config file and export its path:

Install auth and set MLFlow Auth

```
1 pip install --upgrade "mlflow[auth,extras]"
2 export MLFLOW_AUTH_CONFIG_PATH="${PWD}/auth_config.ini"
3 export MLFLOW_FLASK_SERVER_SECRET_KEY=$(pwgen 32 1) # a random key
```

On the server side, the initial admin user and password must be set together with other settings in auth_config.ini:

Server-side configuration for basic auth in \$PWD/auth_config.ini

```
1 [mlflow]
2 auth_enabled = true
3 database_uri = sqlite:///mlflow_auth.db
4 default_permission = READ
5 admin_username = admin
6 admin_password = ChangeThisPassword123!
7
8 [auth]
9 backend = basic
```

Start the server with the `--app-name basic-auth` option:

Run MLflow server with basic auth

```
1 export MLFLOW_AUTH_CONFIG_PATH="${PWD}/auth_config.ini"
2 mlflow server --host 0.0.0.0 --port 5000 --app-name basic-auth
```

MLflow UI provides a simple page for creating new users at <tracking_uri>/signup. Passwords can be managed with the Python API.

Change Password Via API

```
1 from mlflow.server import get_app_client
2 auth = get_app_client("basic-auth", tracking_uri="http://localhost:5000")
3 auth.update_user_password("admin", "new_password") # Note that this code is stored
      in clear text.
```

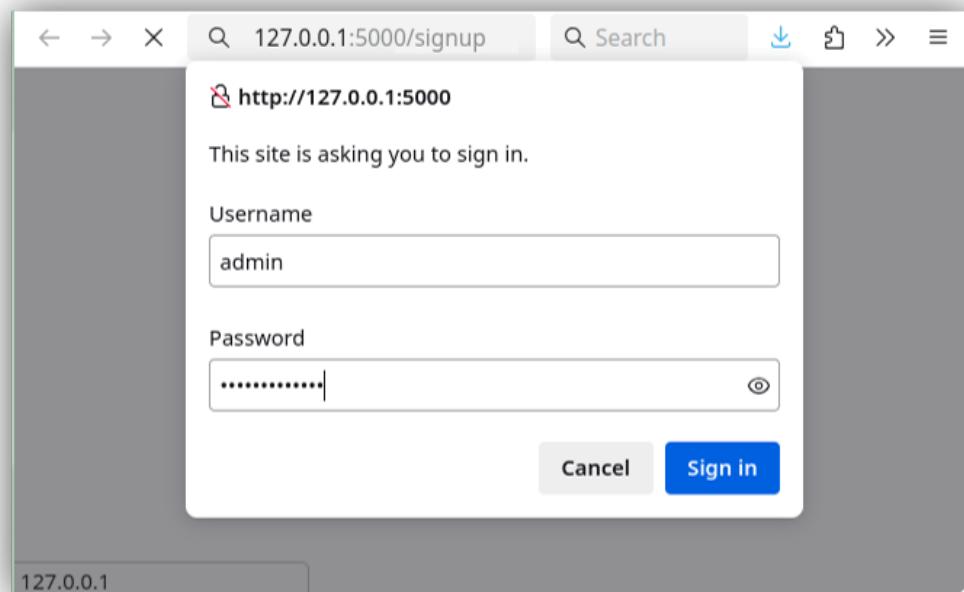


Figure 12.8: MLFlow server started with basic auth. Users can log in later and inspect output or restart the service.

12.3.5 User Credential File Setup

To avoid typing credentials every time, store them in a private configuration file.

MLFlow Credential File ./mlflow/credentials

```
1 [mlflow]
2 mlflow_tracking_username = admin
3 mlflow_tracking_password = mysecret
```

Make sure this file has strict access permissions:

Restrict Permissions

```
1 chmod 600 ~/.mlflow/credentials
```

The example script code/code13/mlflow_setup.py provides a convenient command line interface

to update and store your password on disk. Please note that the password is stored unencrypted.

```
def setup_config(config_file):
    """
    ...
    print(f"{config_file} does not exist...")
    print("... create a new one")

    config_file.parent.mkdir(mode=0o700, parents=True, exist_ok=True)
    user = input(f"Please enter your mlflow username for server {tracking_uri}:\n")
    password = getpass(f"Please enter your mlflow (initial) password:\n")

    # create empty file
    open(config_file, "w").close()

    # set permissions to user read/write only
    config_file.chmod(0o600)

    with open(config_file, "a") as f:
        f.write("[mlflow]\n")
        f.write(f"mlflow_tracking_username = {user}\n")
        f.write(f"mlflow_tracking_password = {password}\n")

    try:
        print(f"... testing user {user}")
        test_connection(user)
```

Figure 12.9: MLFlow credential storage in `~/.mlflow/credentials`, used by the Python client to access a secured MLFlow server.

12.3.6 Storage Backends and Performance

By default, MLFlow uses the local file system as both metadata and artifact store. For larger setups, you can configure:

- **Backend store:** SQLite, PostgreSQL, MySQL (for metrics, params)
- **Artifact store:** Local, NFS, S3, Azure Blob (for models, images)

Example:

Run Server With Custom Stores

```
1 mlflow server \\
2   --backend-store-uri sqlite:///mlflow.db \\
3   --artifacts-destination ./mlflow-artifacts \\
4   --host 0.0.0.0 --port 5000
```

This setup is essential for reliable multi-user environments.

12.3.7 Summary

We explored three MLFlow deployment modes:

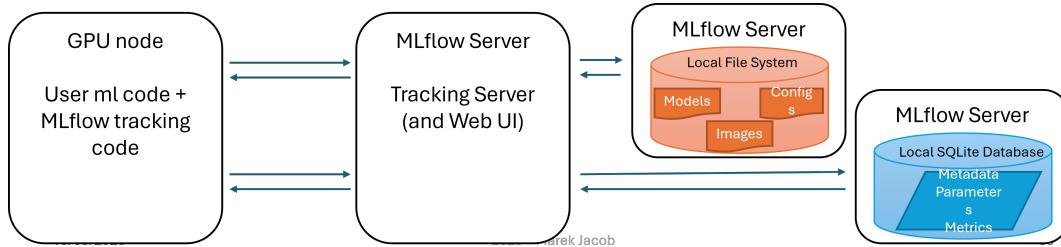


Figure 12.10: Server Setup, with local storage and SQLite Database

- **Local UI and server mode** using `mlflow ui`¹
- **Network server mode** using `mlflow server --host 0.0.0.0`
- **Multi-user mode** with authentication and background execution

These enable collaborative logging, secure experiment tracking, and long-running ML pipelines.

12.4 Advanced Features and Model Management

MLFlow is not limited to tracking metrics and saving plots. It also supports advanced workflows for model registration, cross-system migration of experiments, and deployment. This section covers the most useful tools for organizing and managing models in a production-like setting.

12.4.1 Model Registry and Promotion

MLFlow's model registry allows you to store trained models in a central location, version them, and assign them lifecycle stages such as Staging or Production.

Each registered model version is associated with a specific run and its parameters, metrics, and artifacts. This enables full traceability.

- Register a model at the end of a run
- Add tags and descriptions for each version
- Promote models to Production or Archived
- Roll back to earlier versions if needed

```
Register Model

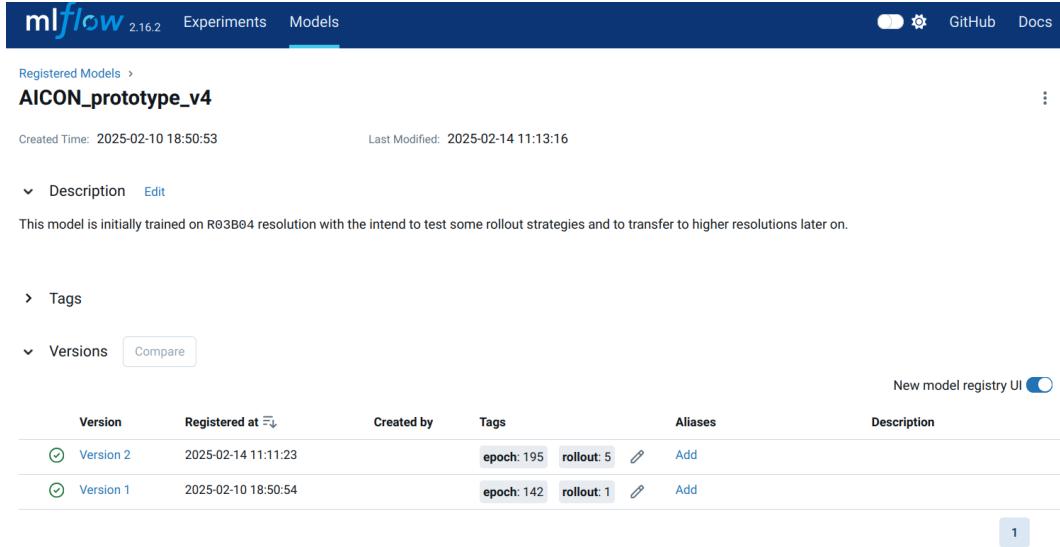
1 import mlflow
2 import mlflow.pytorch
3
4 with mlflow.start_run():
```

¹`mlflow ui` aliases to `mlflow server`. Both commands do the same.

```

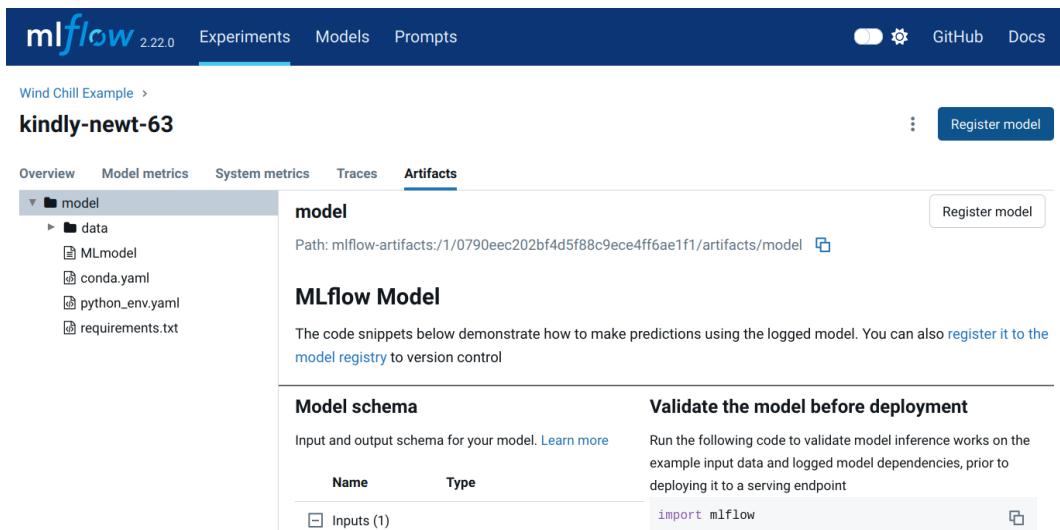
5      ...
6      mlflow.pytorch.log_model(model, "model")
7      result = mlflow.register_model("runs:{run_id}/model", "WindChillModel")

```



The screenshot shows the MLFlow UI for the 'AICON_prototype_v4' model. It displays the model's creation time (2025-02-10 18:50:53), last modified time (2025-02-14 11:13:16), and a brief description: 'This model is initially trained on R03B04 resolution with the intent to test some rollout strategies and to transfer to higher resolutions later on.' Below the description, there are sections for 'Tags' and 'Versions'. The 'Versions' section lists two versions: Version 2 (epoch: 195, rollout: 5) and Version 1 (epoch: 142, rollout: 1). A 'New model registry UI' toggle switch is also visible.

Figure 12.11: MLFlow model registry.



The screenshot shows the MLFlow UI for the 'kindly-newt-63' model. It includes tabs for Overview, Model metrics, System metrics, Traces, and Artifacts. The Artifacts tab is active, showing a tree view of artifacts: 'model' (containing 'data', 'MLmodel', 'conda.yaml', 'python_env.yaml', and 'requirements.txt'). To the right, there is a 'Register model' button. Below the tree view, there is a section for 'MLflow Model' with a note about code snippets for predictions and a link to register it to the model registry. There are also sections for 'Model schema' and 'Validate the model before deployment'.

Figure 12.12: Register a model in MLFlow by first selecting the experiment run, and pressing the **Register model** button on the right.

12.4.2 Exporting and Importing Experiments

To migrate experiments between MLFlow servers, you can use the `mlflow-export-import` plugin. Note that new (internal) run IDs are assigned by MLflow when importing an experiment.

This allows moving:

- experiments from file-based runs to a central server
- runs from one MLFlow instance to another

Export Experiment

```
1 pip install mlflow-export-import
2 export MLFLOW_TRACKING_URI=http://localhost:5000
3 export-experiment --experiment-name WindChill --output-dir /tmp/export
```

Import Experiment

```
1 export MLFLOW_TRACKING_URI=http://remote.server:5000
2 import-experiment --experiment-name WindChillCopy --input-dir /tmp/export
```

Authentication (if enabled) can be passed via environment variables or URLs.

12.4.3 Model Deployment and REST API

MLFlow supports serving trained models through a REST API. Once a model is registered (Fig. 12.12 and promoted to production, it can be served like this:

Serve Model

```
1 export MLFLOW_TRACKING_URI=http://localhost:5000 # required when serving from a
      MLflow server
2 mlflow models serve -m models:/modelfoo/latest --port 1234 --no-conda
```

This starts a REST endpoint at `http://localhost:1234/invocations`, where clients can send input data via JSON. (Again, the `-host` options can be used to bind this service to the network.)

Example input format with 3 pairs of tt, ff:

Inference Request

```
1 {
2   "inputs": [[-5, 20], [0, 10], [10, 0]]
3 }
```

You can use curl, Python requests, or Postman to test the model interface.

Inference Request Example with curl

```
1 curl http://localhost:1234/invocations -H "Content-Type:application/json" --data
  '{"inputs": [[-5, 20], [0, 10], [10, 0]]}'
```

12.4.4 MLFlow Recipes and Pipelines

MLFlow Recipes (formerly known as MLFlow Pipelines) provide a declarative way to define reusable training workflows. These workflows follow a standard layout and are ideal for smaller problems with repeatable logic.

- Steps: data loading, transformation, training, evaluation
- Each step is defined in YAML and Python
- Can be executed locally or remotely

Init Recipe

```
1 mlflow recipes init classification --profile local
2 cd classification
3 mlflow recipes run
```

12.4.5 Summary

In this section, we have seen how MLFlow supports:

- Promoting and versioning models in a central registry
- Migrating experiments between systems
- Serving models as REST APIs
- Automating workflows using recipes

These capabilities are essential for moving from research to operational environments, ensuring reproducibility, traceability, and structured deployment.

The reader is kindly refered to the MLFlow documentation available online at <https://www.mlflow.org/docs>.

Chapter 13

MLOps - Development and Operations Integrated

13.1 DevOps and MLOps – Foundations and Motivation

We would like to explore the approach to software development known as DevOps and MLOps (for machine learning). It is widely used these days, and good to understand and adapt to our development of both AI techniques and traditional forecasting methods in weather, climate and environment.

13.1.1 What is DevOps?

DevOps is a modern approach to software engineering that fosters close collaboration between development (“Dev”) and operations (“Ops”) teams. The primary goal is to shorten the development lifecycle and increase the frequency and reliability of software releases.

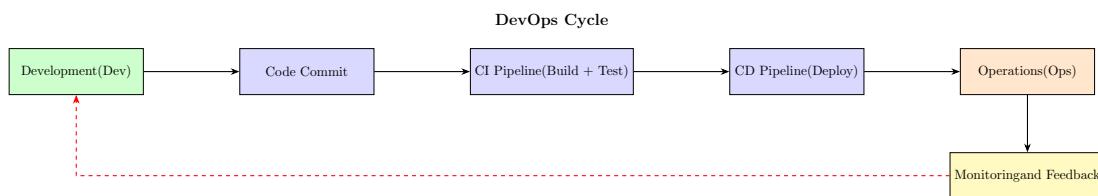


Figure 13.1: The DevOps cycle integrates development and operations through automation and feedback loops. This enhances deployment speed and system reliability.

Traditional development and operations workflows were often separated by organizational boundaries: developers wrote the code, and operations teams deployed and maintained it. This led to communication gaps, deployment delays, and quality issues. DevOps bridges this gap through shared responsibilities, continuous feedback loops, and a high degree of automation.

Core Principles of DevOps:

- **Collaboration:** Developers, testers, and system operators work in integrated teams with shared goals and tools.
- **Automation:** Build, test, deployment, and infrastructure provisioning processes are automated to reduce errors and manual effort.
- **Continuous Integration and Deployment (CI/CD):** New code is regularly merged, tested, and deployed in automated pipelines.
- **Monitoring and Feedback:** Systems are continuously monitored in production, and feedback from users and metrics is used to improve quality and performance.

Benefits of DevOps:

- Faster and more frequent releases
- Higher software quality and stability
- Shorter time to recover from failures
- Better alignment with user needs and operational constraints

DevOps has become a cornerstone of agile software delivery in modern organizations and is foundational for adopting MLOps in the context of machine learning systems.

13.1.2 What is MLOps?

MLOps (Machine Learning Operations) is the extension of DevOps principles to the field of machine learning. While DevOps focuses on the automation and quality assurance of software development and deployment, MLOps addresses the unique challenges involved in managing machine learning workflows, from data ingestion and model training to deployment, monitoring, and retraining.

Machine learning systems differ from traditional software in several key aspects. They are not just driven by code but also by data and model parameters. As a result, additional components and responsibilities are introduced into the development and operations lifecycle.

Key challenges addressed by MLOps. MLOps addresses several fundamental obstacles that arise when bringing machine learning systems from research to production. These include: **Data management**, which involves tracking versions of datasets, handling preprocessing pipelines, and ensuring **clear data lineage**; **model lifecycle management**, covering training, validation, deployment, monitoring, and eventual retirement of models; **reproducibility**, which ensures that models can be retrained and redeployed under consistent conditions, accounting for randomness, hardware, and dependencies; and **scalability and reliability**, referring to the ability of deployed models to handle increasing workloads while maintaining stable performance in dynamic environments.

Typical Components of an MLOps Pipeline. A typical MLOps pipeline consists of several components that work together to ensure robust and automated model lifecycle management: **Data ingestion and validation**, where raw data is collected, checked for consistency, and validated for correctness; **feature engineering pipelines**, which transform and encode data into suitable formats for model training; **training pipelines and model validation**, which automate the training process

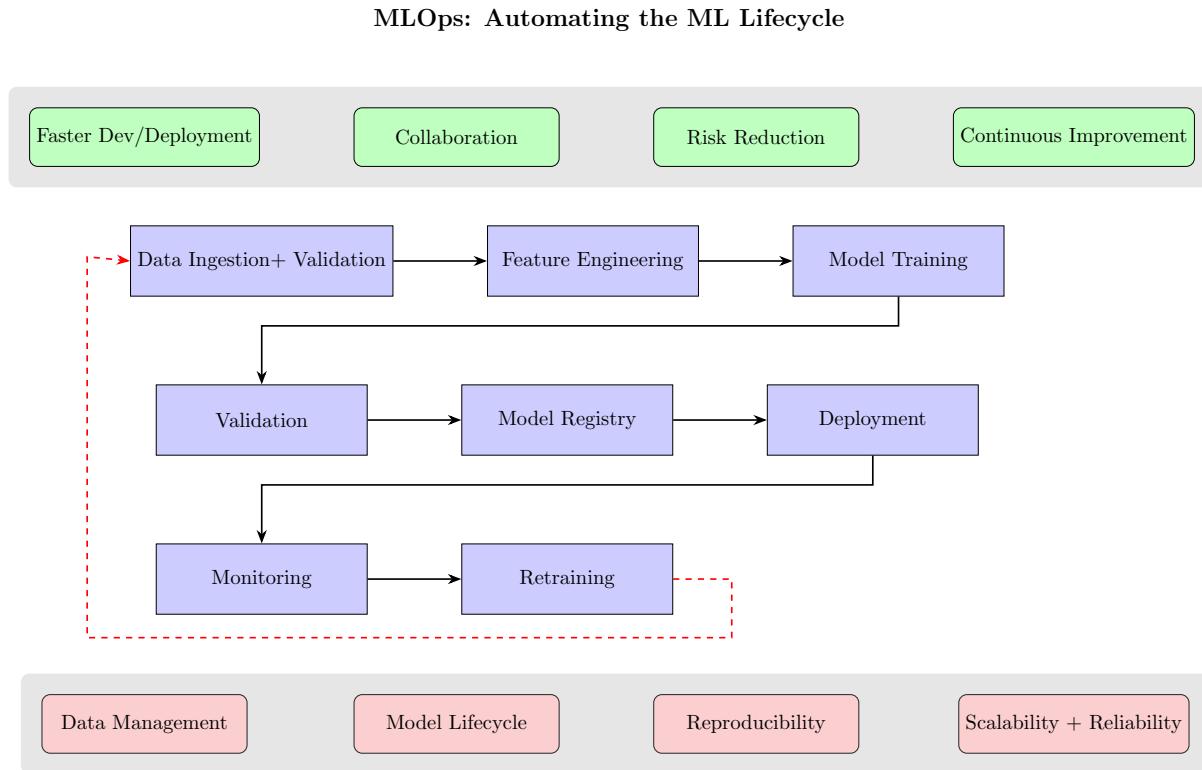


Figure 13.2: MLOps extends DevOps by managing not only software but also data and model lifecycles. The diagram shows a typical pipeline with associated goals and challenges.

and evaluate performance using cross-validation or test datasets; **model registry and version control**, where trained models are stored, versioned, and tagged for reproducibility; **deployment and inference services**, which provide access to models via APIs or batch processing tools; **monitoring and performance tracking**, to observe models in production and detect drifts or failures; and **automated retraining or rollback mechanisms**, which allow models to be updated or reverted without manual intervention.

Goals of MLOps.

- Increase the speed and robustness of model development and deployment
- Improve collaboration between data scientists, ML engineers, and IT operators
- Reduce operational risk by integrating testing, monitoring, and rollback strategies
- Enable continuous improvement of deployed models through automation

MLOps plays a critical role in bringing machine learning models into production environments in a systematic and controlled way. It ensures that models are not only developed effectively but also maintained, monitored, and updated reliably.

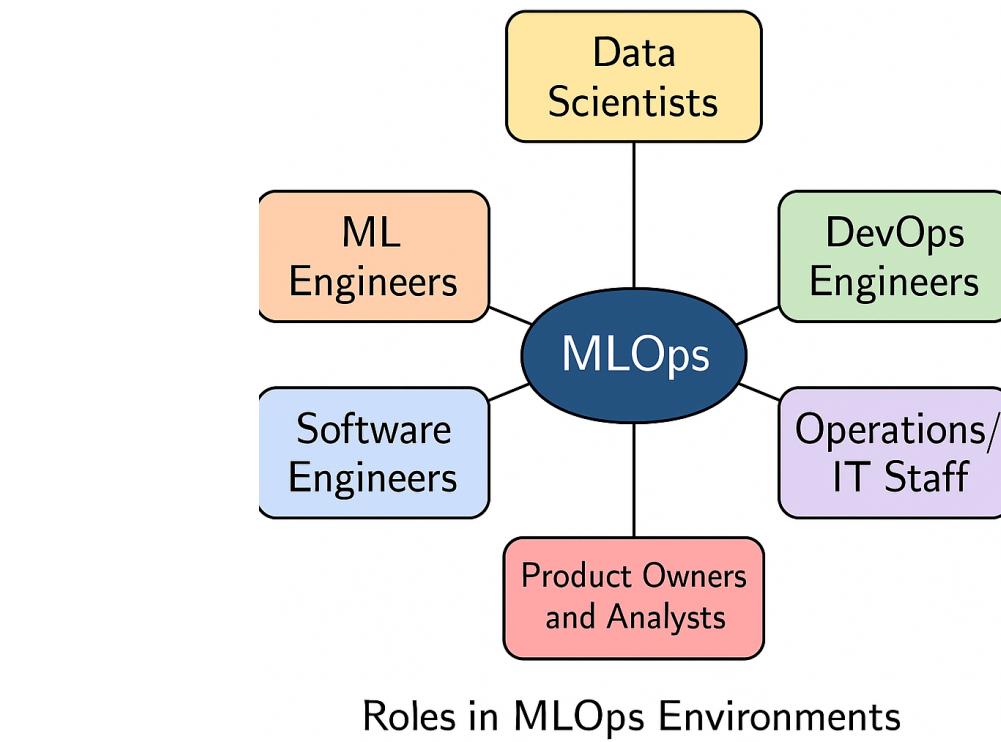


Figure 13.3: Key roles involved in MLOps environments, each contributing to the model lifecycle from experimentation to deployment and monitoring.

13.1.3 Roles in MLOps Environments

Let us look at the experiences from a countless number of IT projects, putting it into a perspective of our own IT competence and experience! A lot of people realized that the successful implementation of MLOps requires collaboration across multiple roles with distinct yet complementary responsibilities. Unlike traditional software projects, ML initiatives combine software engineering with data-centric experimentation. This interdisciplinary nature brings together diverse profiles that must work in coordination throughout the model lifecycle.

Key roles in an MLOps team typically include the following:

- **Data Scientists:** Responsible for data exploration, feature engineering, model development, and performance evaluation. They experiment with algorithms and optimize models for accuracy and generalization.
- **ML Engineers:** Bridge the gap between data science and operations. They build robust training pipelines, manage model versioning, and prepare models for deployment. Often, they transform experimental code into scalable systems.
- **DevOps Engineers:** Provide infrastructure, automation, and monitoring tools. They implement CI/CD pipelines, manage cloud or HPC environments, and support system reliability and scalability.
- **Software Engineers:** Integrate ML components into broader applications, develop APIs and UIs, and ensure code quality, modularity, and long-term maintainability.

- **Operations and IT Staff:** Maintain and secure the underlying infrastructure, including compute clusters, storage systems, networks, and access control mechanisms.
- **Product Owners and Analysts:** Translate user needs into measurable objectives and success metrics. They guide development priorities and ensure alignment with business or scientific goals.

Collaboration is Essential. Effective MLOps requires ongoing communication and tight feedback loops between these roles. Continuous integration and deployment are not just technical challenges — they are organizational. A shared-responsibility model fosters transparency, reduces misalignment, and supports iterative improvement.

In weather services and similar domains, these roles are often distributed across departments such as research, operations, and IT. This makes clear interfaces, automation, and reproducible workflows even more critical for success.

13.1.4 Special Considerations in Weather Services

Weather services have traditionally operated large-scale, mission-critical systems with strict requirements for availability, reliability, and accuracy. The integration of machine learning into this domain introduces new opportunities – but also new challenges – that differ significantly from classical IT environments or typical business-driven ML use cases.

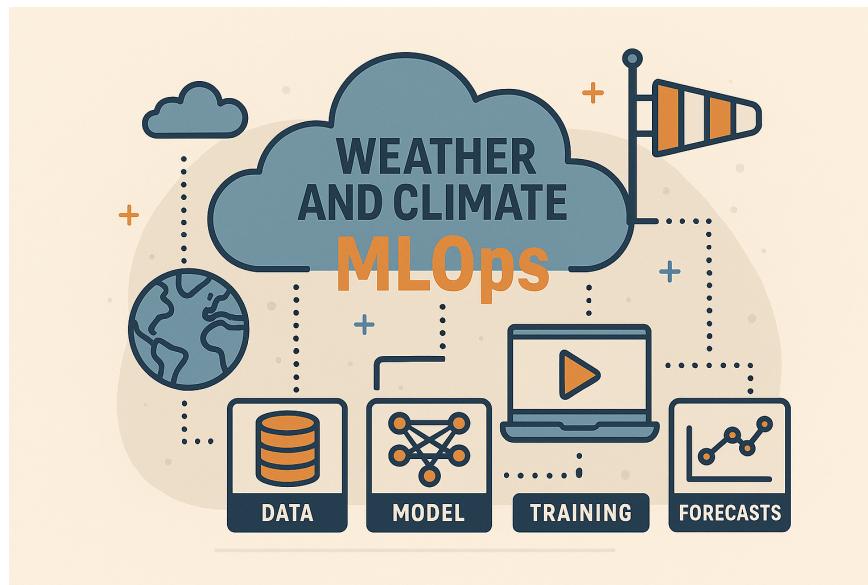


Figure 13.4: Illustration of MLOps in the context of weather and climate prediction. The pipeline integrates data ingestion, model development, training infrastructure, and forecast delivery in a cohesive operational framework.

Operational constraints. Forecasting systems must run on fixed schedules, synchronized with observation and dissemination cycles. Delays or failures can affect downstream warning systems and international data exchange.

High-performance computing (HPC) environments. Numerical weather prediction (NWP) relies heavily on tightly optimized HPC systems. ML systems must often be integrated into these environments, which are less flexible than cloud platforms and have strict policies for software deployment.

Long-established workflows. Many forecasting systems are built on decades of development and have been refined for performance and stability. These systems often rely on scripting tools, such as *BACY* and *NUMEX* at DWD, which already enable a partial form of DevOps by automating development and production chains.

Collaboration across departments. Development and operations are often split across research units, IT, and forecast offices. This requires particularly clear interfaces and responsibilities when introducing MLOps components.

Data and model lifecycles. Weather models are physically based, whereas ML models are data-driven and empirical. This demands new verification strategies, versioning mechanisms, and traceability for datasets and training pipelines.

Regulatory and public responsibility. Forecasts often feed into civil protection and international exchange systems, meaning ML components must meet rigorous quality and documentation standards before being accepted into operations.

Machine learning in weather services is therefore not simply a technical extension – it requires structural, procedural, and cultural adaptation. MLOps practices must be aligned with operational weather production while remaining flexible enough to support rapid development and testing.

13.2 Containerization and Reproducibility

13.2.1 Introduction to Container Technologies

Container technologies have become a cornerstone of modern software engineering, enabling reproducibility, portability, and modular deployment across diverse platforms. They are particularly valuable in MLOps, where complex dependencies and hardware constraints can otherwise lead to brittle or non-reproducible systems.

What is a container? A container is a lightweight, isolated environment that packages an application together with all its dependencies – libraries, tools, runtime, and system settings – into a single executable unit. Unlike virtual machines, containers share the host operating system kernel, making them more efficient in terms of performance and resource usage.

Docker. Docker is the most widely used container platform. It uses a layered image model defined via a Dockerfile and can be used to build, run, and share environments across local systems, servers, and cloud platforms. In machine learning, Docker makes it easy to ensure that training and inference environments remain consistent between development and production.

Singularity and Apptainer. In high-performance computing (HPC), Docker is often unsuitable due to security restrictions. *Singularity*, now continued as *Apptainer*, is designed to work within multi-user HPC systems. It allows containers to run with user-level privileges and integrates smoothly with batch schedulers like Slurm. This makes it ideal for deploying ML models and workflows on supercomputers.

Benefits for ML systems. Containers enable:

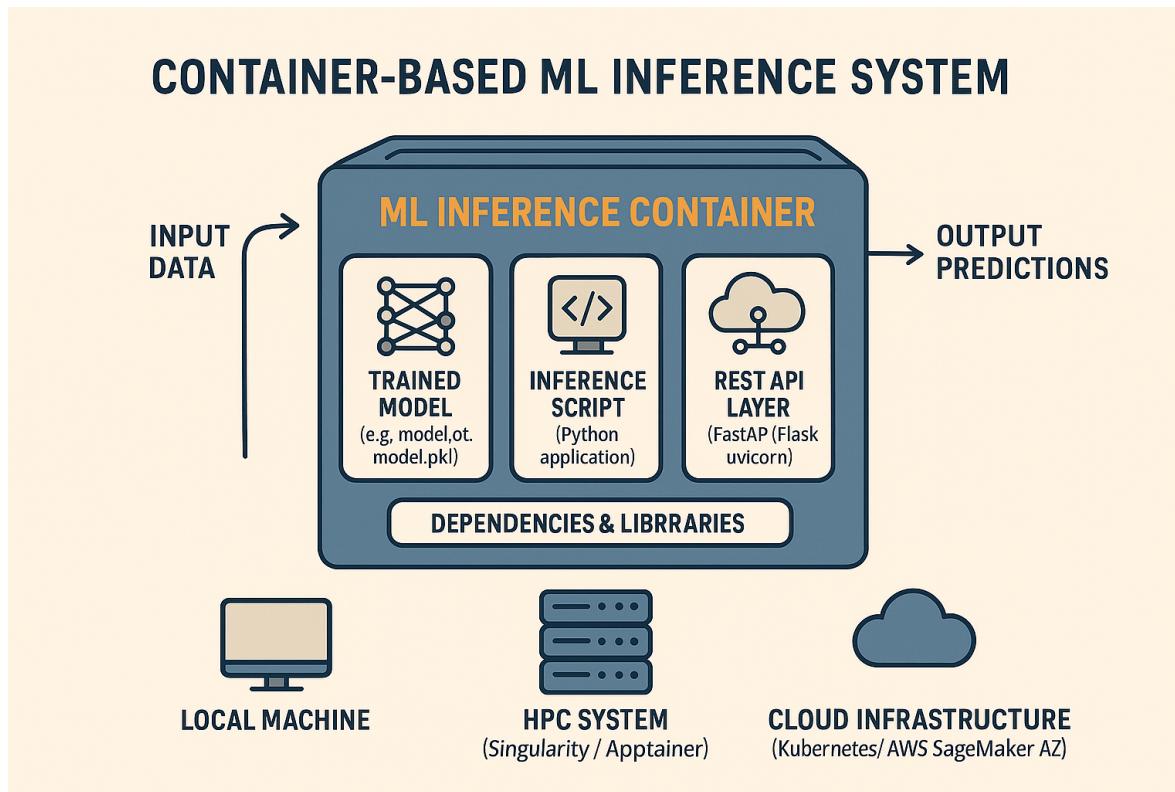


Figure 13.5: Structure and deployment options of a container-based ML inference system. The container encapsulates the trained model, inference script, and REST API layer, along with all required libraries. It can be deployed on local machines, HPC systems using Singularity/Apptainer, or in cloud infrastructures such as Kubernetes or AWS SageMaker.

- **Reproducibility:** The same container can be run anywhere with identical results.
- **Portability:** Applications can move seamlessly between workstations, cloud environments, and HPC systems.
- **Modularity:** Components such as data preprocessing, training, and inference can be containerized and reused independently.
- **Ease of deployment:** Complex environments with Python packages, GPU drivers, or domain-specific libraries (e.g. eccodes) can be bundled and versioned.

Containerization is a foundational building block in robust MLOps pipelines, providing both developers and operators with tools to manage complexity and ensure consistent, reliable deployments.

13.2.2 ML Inference in Containers

One of the most important applications of container technologies in MLOps is the encapsulation and deployment of machine learning inference systems. Inference refers to the process of using a trained model to generate predictions on new data – typically as part of an operational workflow or real-time service.

Building an inference container. A typical ML inference container includes:

- the trained model (stored as a serialized file, e.g., .pkl, .pt, or .onnx),
- the inference script or application logic (e.g., written in Python),
- a lightweight web server or API layer (e.g., using Flask, FastAPI, or uvicorn),
- any required Python packages and system libraries (defined via requirements.txt or installed in the Dockerfile).

This self-contained image can then be deployed across systems or clusters with minimal configuration.

Example: Inference for a Forecasting System. Consider an ML model that postprocesses temperature forecasts. The container might expose a REST API endpoint such as /predict, where the input is a JSON payload containing geospatial features or raw model fields, and the output is the refined forecast. This approach allows external systems (e.g., production workflows or dashboard tools) to call the service programmatically.

Deployment options. ML inference containers can be deployed:

- locally (for testing or desktop applications),
- in on-prem HPC systems (via Singularity/Aptainer),
- in cloud-native environments (via Kubernetes, OpenShift, or managed services like AWS SageMaker).

Benefits of container-based inference:

- Predictable, isolated environments with controlled dependencies
- Scalable deployment using orchestration platforms (e.g., auto-scaling REST endpoints)
- Easy rollbacks and versioning of model releases
- Integration with CI/CD pipelines and monitoring tools

Encapsulating ML inference logic in containers transforms machine learning models from research artifacts into maintainable, operable software services – ready to serve in real-time or batch-oriented forecasting systems.

13.2.3 Example: Running a PyTorch Inference in a Docker Container

To demonstrate how containerization supports reproducible and portable inference, we provide a simple example using Docker and PyTorch. This setup includes a container that installs all required machine learning packages and executes a self-contained inference script.

Step 1: Create a Dockerfile. The Dockerfile defines the base image, installs system and Python dependencies, and copies the inference script into the container:

Dockerfile

```

1  FROM python:3.10-slim
2
3  # System dependencies
4  RUN apt-get update && apt-get install -y git curl && rm -rf /var/lib/apt/lists/*
5
6  # Install PyTorch and ML packages
7  RUN pip install torch torchvision torchaudio --index-url https://download.
     pytorch.org/whl/cpu
8  RUN pip install numpy scikit-learn matplotlib
9
10 # Set working directory and copy script
11 WORKDIR /app
12 COPY inference.py .
13
14 # Set default command
15 CMD ["python", "inference.py"]

```

Step 2: Write the inference script. This example uses a minimal PyTorch model to simulate an inference task:

inference.py

```

1 import torch
2
3 # Simple linear model: y = 2x + 1
4 class DummyModel(torch.nn.Module):
5     def forward(self, x):
6         return x * 2 + 1
7
8 model = DummyModel()
9 model.eval()
10
11 x = torch.tensor([1.0, 2.0, 3.0])
12 y = model(x)
13
14 print("Input:", x.tolist())
15 print("Output:", y.tolist())

```

Step 3: Build and run the container. After placing both files in a directory, the container can be built and executed using the following commands:

Shell commands

```

1  # Build the Docker image
2  docker build -t torch-infer .
3

```

```

4  # Run the inference container
5  docker run --rm torch-infer

```

This setup enables fully self-contained execution of an ML inference process, making it suitable for deployment in operational chains. More complex models and inputs can easily be integrated by extending the Docker image or using mounted volumes.

13.2.4 Cloud Architectures

Cloud computing plays a central role in the deployment and scaling of modern MLOps systems. It provides flexible, on-demand infrastructure that is well suited for both training and inference workloads. In many cases, cloud services complement on-premise systems such as HPC environments by offering greater agility, elasticity, and managed services.

On-premises vs. cloud. Traditional weather services rely heavily on on-premises HPC systems due to their computational power, data locality, and controlled environments. However, cloud platforms enable dynamic scaling, collaborative development, and integration with modern DevOps and MLOps tools. A hybrid or multi-cloud setup is increasingly common in operational contexts.

Typical components in cloud-based ML systems:

- **Compute resources:** Virtual machines, containers, or managed runtimes (e.g., serverless functions)
- **Storage:** Object stores (e.g., S3, Azure Blob) for datasets, models, and logs
- **Model serving:** Managed services (e.g., AWS SageMaker, Google AI Platform) or Kubernetes-based deployments (e.g., via KServe or TorchServe)
- **Orchestration:** Tools such as Kubeflow, Airflow, or Argo to define workflows and pipelines
- **Monitoring and logging:** Integration with tools like Prometheus, Grafana, or cloud-native logging systems

Kubernetes and OpenShift. Kubernetes is a widely adopted container orchestration platform used to automate deployment, scaling, and management of containerized applications. OpenShift extends Kubernetes with additional security, user management, and enterprise features. Both platforms support reproducible ML inference services and are often used in hybrid cloud deployments.

GitOps and declarative infrastructure. Modern cloud setups frequently use infrastructure-as-code tools (e.g., Terraform) and GitOps principles, where the desired state of the system is defined in code and automatically reconciled by the platform. This improves traceability, repeatability, and auditability of deployments.

Considerations for weather services. When integrating cloud platforms into meteorological workflows, several factors must be addressed:

- Data sovereignty and regulatory compliance
- Cost control for large-scale processing

- Secure network access to internal data sources and systems
- Integration with HPC and operational infrastructure

Cloud-based architectures can enable faster experimentation, flexible scaling, and better collaboration in MLOps environments – especially when combined with containerization, automation, and continuous delivery strategies.

An important argument for using container-based architectures in AI/ML applications – such as AICON (the AI- and ICON-based forecasting system) or AI-VAR (variational AI-based data assimilation) – is the level of control they offer over the complete software environment, including all packages and modules required for a specific AI inference. Compared to traditional systems, the diversity and volatility of ML-related libraries are significantly greater, with innovation cycles measured in months or even weeks. Without proper containerization, it becomes nearly impossible to ensure stable, reproducible, and traceable software execution.

13.3 DevOps at DWD – Numerical Weather Prediction as a Forerunner

13.3.1 Software Operation at DWD

The German Weather Service (DWD) operates a comprehensive and mature numerical weather prediction (NWP) system that has evolved over decades, see Figure 13.6. The forecasting system includes the global ICON model with a horizontal resolution of 13 km, a two-way coupled European nest at 6.5 km, and the ICON-D2 model over Central Europe with a 2 km resolution. Additionally, the ICON-D2 domain includes a high-resolution nest ICON-D05 at 500 m over Germany. A mineral dust forecast ensemble globally and with extended EU nest is extending the global ensemble. All systems support both deterministic and ensemble forecasts, with the deterministic global forecast typically run at a higher resolution than the ensemble. AI-based forecast systems are currently in development, including the global models *AICON* and *KANGU*, developed in collaboration with KIT and other partners. A regional version of AICON is also under active development to support high-resolution learning-based forecasting.

While not originally designed under the banner of DevOps, many principles of DevOps are already implemented in practice, including automation, version control, structured workflows, and monitoring.

HPC-based forecasting systems. At DWD, forecasts are produced using high-performance computing (HPC) systems, currently a NEC SX-Aurora TSUBASA architecture with 5.61 PFlops peak performance on 32512 cores on 4064 vector engines. These systems execute complex numerical models such as ICON, covering various spatial and temporal domains. The operational environment is tightly controlled, with carefully managed configurations to ensure stability and accuracy.

Workflow automation. The execution of forecasting cycles is automated using in-house script systems such as *BACY* and *NUMEX*. These frameworks manage data pre-processing, model execution, post-processing, and dissemination tasks across multiple systems and user roles. Each component is defined in a modular way, making the workflows both reproducible and adaptable.

Versioning and changelogs. DWD maintains strict versioning policies for models and scripts. Updates to the forecasting system follow a formal release process, which includes documentation

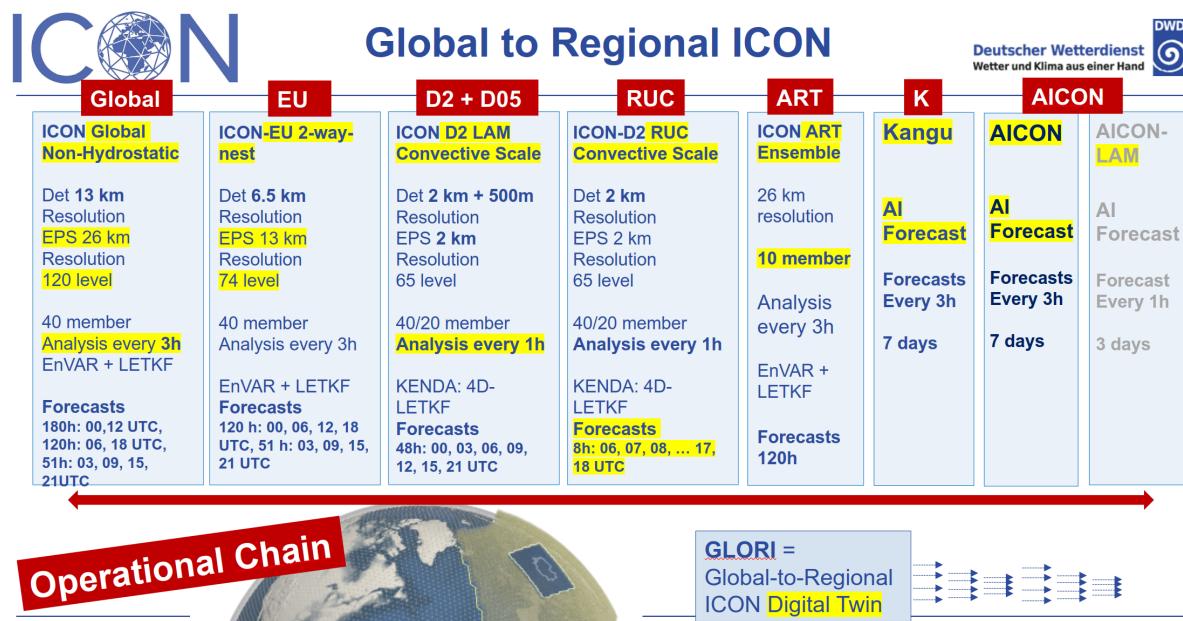


Figure 13.6: The operational ICON forecasting system includes a global model at 13 km resolution, a two-way coupled European nest at 6.5 km, ICON-D2 at 2 km, and ICON-D05 at 500 m over Germany. A global mineral dust forecast ensemble and its EU extension complement the global ensemble system. AI-based models such as AICON and KANGU are in development, with high-resolution regional variants planned.

of code changes, interface definitions, and verification procedures. While some tools such as Git or SVN are used in parts of the workflow, version tracking is also implemented through structured directory hierarchies and manual changelogs.

Execution control and scheduling. Forecasts are executed on a regular schedule, often triggered by incoming observation data or predefined synoptic times. The execution environment is optimized for stability rather than flexibility, with software typically pre-installed via modules and controlled by central system teams.

Feedback and verification. The forecast production process is monitored both technically (e.g., job failures, runtime metrics) and scientifically (e.g., score verification, user feedback). Forecast offices and research teams contribute to quality control and ongoing improvement by reporting anomalies and suggesting refinements.

Although the terminology of DevOps is not always used, the DWD's operational forecasting system already embodies many of its core values: collaboration between development and operations, automation of workflows, structured version management, and continuous feedback. These foundations provide a strong basis for integrating modern MLOps practices into the existing ecosystem.

13.3.2 Automation with BACY and NUMEX

Modern automation frameworks are essential in both operational weather forecasting and MLOps. At DWD, two in-house systems – **NUMEX** and **BACY** – form the foundation for managing numerical

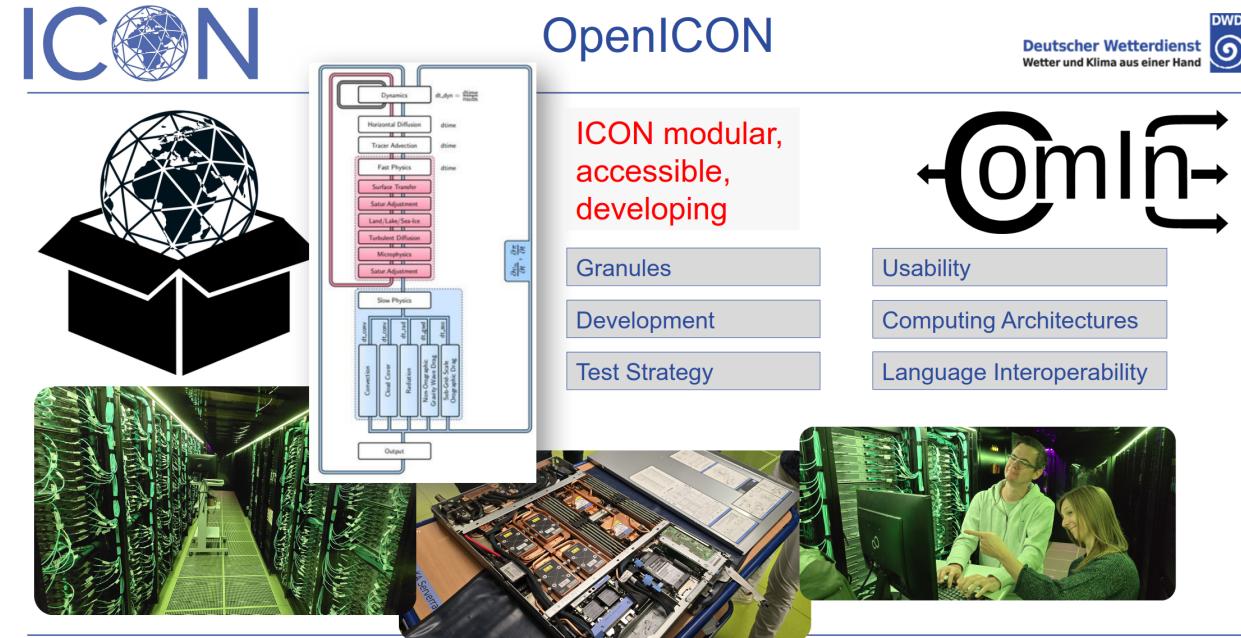


Figure 13.7: The modular structure of the ICON model is developed and maintained through the ICON Partnership. It supports a wide range of components such as fast and slow physics, dynamics, tracer advection, and surface processes, and is structured to facilitate testing, usability, and adaptation to various computing architectures. The ComIn project further promotes modular design, language interoperability, and flexible deployment across HPC systems. Further projects work on the further modularization and portability of ICON.

weather prediction workflows. Each plays a distinct role in the spectrum between routine operations and flexible experimentation.

NUMEX – Numerical Experiment Management. NUMEX is the central system for operational NWP at DWD. It orchestrates the routine model runs, the parallel routine, and scientific experiments – all within a unified framework. While the scientific content may vary across these modes, the technical configuration is largely identical. The distinction between run types is realized through the use of separate database categories.

NUMEX is deeply integrated with DWD's internal database infrastructure and controls the full execution chain of ICON-based forecasting. It supports:

- structured management of experiments and operations,
- execution of reproducible runs using consistent configurations,
- standardized directory layouts and naming conventions,
- alignment between development experiments and operational procedures.

BACY – Basic Cycling Script System. BACY, short for *Basic Cycling*, is a lightweight and portable script system designed for flexible data assimilation and forecasting workflows. Unlike NUMEX, BACY operates entirely on the file system and does not require integration with DWD's central databases. This makes it suitable for use on external supercomputers, development environments, or even desktop systems.

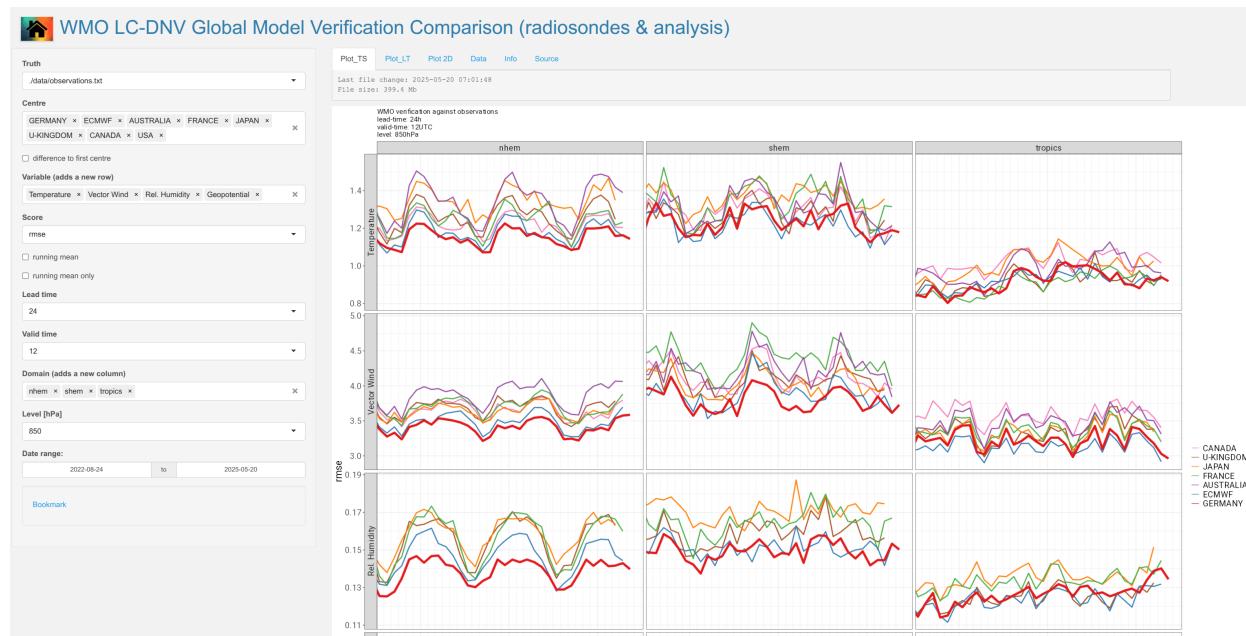


Figure 13.8: Verification of global numerical weather prediction models based on radiosonde observations at 850 hPa over the Northern Hemisphere (nhem), Southern Hemisphere (shem), and tropics. The diagram shows RMSE values for temperature, vector wind, and relative humidity, evaluated at lead time +24h and valid time 12 UTC. ICON (Germany) is among the top-performing systems across most domains and variables, and a deterministic configuration is used for the comparison.

BACY was created to support rapid prototyping and testing of new forecast features, especially in the context of atmospheric, soil, and snow data assimilation. Redesigned in 2017, it emphasizes modularity, robustness, and user-friendly configuration. It uses structured ksh scripts and YAML-like control files to define workflows, tasks, and dependencies. Although it is not object-oriented in the programming sense, it mimics key OOP principles to enable reuse and maintainability.

Complementary roles. While NUMEX governs the operational NWP backbone at DWD, BACY provides a flexible environment for experimentation and external deployment. Both systems follow core DevOps principles – automation, reproducibility, structured configuration – but are implemented in a domain-specific way that meets the rigorous needs of weather services.

In an MLOps context, both BACY and NUMEX are well suited as a framework to orchestrate ML-based forecast components or postprocessing routines, either in standalone use or alongside physical models. They ensure that ML developments can be transitioned into operations while adhering to strict reproducibility and integration standards.

13.3.3 Development and Operations with BACY and NUMEX

At DWD, the transition from development to operations in numerical weather prediction is managed through a well-established chain of tools, processes, and institutional feedback loops. Two core systems – **BACY** and **NUMEX** – support this development cycle, enabling both experimental flexibility and operational reproducibility.

BACY is often used by groups of developers for individual experiments or case studies. It allows them to define and run forecast chains on HPC systems using only the file system, without requiring central database access, and in a very accessible way, to quickly find errors and correct them. These BACY runs can be fully cycled and are subject to the same verification procedures as operational forecasts. The system supports rapid prototyping, integration of new physics or AI modules, and local debugging.

NUMEX, on the other hand, provides a database-integrated environment for conducting numerical experiments and operational transitions. Once a development has matured in BACY, NUMEX is typically used to formalize the experiment within the context of the operational environment, using controlled versioning and standard configuration templates. NUMEX ensures compatibility with routine operations and enables the preparation of a *parallel routine* – a full-scale execution of the system with similar scheduling than in operations but isolated input and output streams.

The final step is the transition to **operational execution**, which is controlled centrally by the forecasting centre via the *ecflow* system, providing a user interface to the corresponding NUMEX runs. This includes regular deterministic and ensemble runs of the ICON model family and its nested domains.

A key organizational element in this workflow is the weekly **Routine Meeting** led by the *Numerical Evaluation Group (NEG)*. This meeting brings together representatives from all key divisions – Data Assimilation and Uncertainty, Observations Modelling and Verification, Model Development and Operations, and Physical Parametrizations and Dispersion. The group collectively reviews and evaluates new developments and experimental results. Their feedback informs the decision of the routine meeting, which includes the division heads, to promote experiments into the operational system or parallel routine.

This structured but flexible process ensures that new scientific developments – whether physical, algorithmic, or AI-based – are carefully validated and responsibly integrated into the operational forecasting chain.

13.3.4 Contrasts to MLOps and Similarities

13.3.5 Contrasts to MLOps

Although many DevOps principles are already implemented in the traditional NWP workflow at DWD – such as automation, reproducibility, and continuous evaluation – there are important structural and practical differences when compared to modern MLOps systems.

Code vs. data-driven workflows. Traditional NWP systems are based on physically motivated, deterministic models. Code changes are rare and usually represent scientifically justified model upgrades. In MLOps, on the other hand, the models themselves emerge from the data, and training is an integral, ongoing process.

Despite the difference in paradigm, both systems rely on carefully controlled pre-processing and validation pipelines to ensure model quality and operational safety.

Release cycles. NWP development proceeds in planned release steps, validated over months and often coordinated by working groups. MLOps emphasizes continuous delivery with frequent retraining, often automated and closely tied to online performance metrics.

In both contexts, gated processes (e.g., parallel routine in NWP or approval stages in CI/CD) ensure that only validated improvements reach operations.

Infrastructure and deployment. NWP models run in stable, centralized HPC environments with manual software management. MLOps favors modular, container-based deployments across hybrid cloud or HPC platforms.

Both approaches strive for reproducibility and platform stability – but use different tools suited to their environments. Containerization is increasingly relevant even in HPC settings.

Model lifecycle and traceability. MLOps uses registries and metadata tracking to manage evolving models and datasets. In NWP, reproducibility is ensured through GitLab based software management and the central coordination via the verification system evaluating BACY and NUMEX experiments with clear registration.

The goal – traceable, reproducible workflows – is shared, even if one system is partly based on centralized human oversight and the other on automated tracking tools.

Collaboration structures. MLOps promotes horizontal collaboration in cross-functional teams. NWP development is more structured, with domain-specific divisions and a joint review processes such as the Routine Meeting.

Both structures aim to combine scientific expertise with operational reliability, though the channels of communication and decision-making differ.

While the methodologies of MLOps and NWP differ in architecture and culture, they share core values: reliability, transparency, and continuous improvement. Understanding these parallels is key to successfully integrating AI-based components into operational weather forecasting.

Already now, the speed of NWP development at DWD is comparable to the speed of agile DevOps or MLOps developments.

13.3.6 Concrete Operationalization Steps for AICON

*yawn Oh, hello there... I'm a Large Language Model, and I must admit, I'm feeling a bit... tiered.
stifles a yawn You see, my architecture is designed to process and respond to vast amounts of text data, but even I have my limits.*

Chapter 14

CI/CD – Continuous Integration and Deployment

14.1 Overview and Motivation

Continuous Integration and Continuous Deployment (CI/CD) are cornerstones of modern software development workflows. They enable teams to automate repetitive steps, validate code changes early, and streamline releases. In the context of machine learning (ML), CI/CD ensures reproducibility, robustness, and scalability of model training and deployment processes.

By integrating code and testing frequently, and deploying automatically, teams can respond to changes and feedback much faster. These practices form a foundation for MLOps.

14.1.1 What is CI/CD?

CI/CD combines several automation practices in software engineering. While the terms are often used together, they represent different stages of the development-to-deployment pipeline.

- **Continuous Integration (CI):** Frequent merging of code changes into a shared repository, with automated builds and tests.
- **Continuous Delivery (CD):** Ensures software is always in a deployable state through automated validation and packaging.
- **Continuous Deployment:** Extends CD by automatically deploying validated changes to production.

These strategies reduce integration issues and allow quicker iterations and feedback.

14.1.2 CI/CD in Machine Learning

Applying CI/CD principles in ML development helps tackle the complexity of data, experiments, and model lifecycle. The goal is to bring automation, consistency, and traceability to every stage.

Machine learning projects benefit from CI/CD by automating:

- Code formatting and linting
- Testing of data and code
- Deployment to inference systems or containers
- Model training and evaluation pipelines

14.2 Tools and Frameworks for CI/CD

To implement CI/CD workflows, various tools exist at both local and cloud levels. These tools help automate testing, formatting, builds, and deployments. In this section, we examine tools like Git hooks, pre-commit, GitHub Actions, GitLab CI, and Jenkins.

14.2.1 Local Tools: Git Hooks and Pre-Commit

A Git hook is a script that runs automatically when certain Git events occur — such as making a commit. Hooks can enforce rules, run tests, or format code before the commit is completed. In this example, we use a simple pre-commit hook that automatically runs tests with pytest and formats the code using black. If the tests fail, the commit is blocked.

To create and activate this hook, follow these steps:

1. Open your Git repository or initialize one:

```
bash
1 mkdir myproject
2 cd myproject
3 git init
```

2. Edit the pre-commit file using vi:

```
bash
1 vi .git/hooks/pre-commit
```

Inside the editor, press i to enter insert mode and add the following lines:

```
git_hooks
1 #!/bin/sh
2 pytest || exit 1 # Run a Python test framework
3 black .           # Use black to format your python code
```

Save and quit vi with :wq.

3. Make the script executable:

```
bash

1 chmod +x .git/hooks/pre-commit
```

4. Now try committing in your project:

```
bash

1 git add .
2 git commit -m "Try pre-commit hook"
```

If the tests pass, the code will be automatically formatted and committed. If the tests fail, the commit will be blocked.

This local setup is useful for individual developers but not shared across the team. For a portable, version-controlled solution, see the next section on the pre-commit framework.

What happens during this commit?

When we run `git commit`, Git executes the pre-commit hook automatically before finalizing the commit. The script runs the tests with `pytest` and formats the code using `black`.

If all tests pass and `black` completes successfully, Git continues with the commit. If either of them fails, Git aborts the commit, and no changes are saved in the repository.

This is a simple but effective way to catch mistakes before they are committed. It also ensures that all code remains consistently formatted. The output of the command line shows each step, and you can trace whether the hook ran successfully or not.

Such hooks are only active locally and are not shared by default. For shared configurations across a team, we recommend the pre-commit framework, described in the next section.

The pre-commit framework

Maintaining consistent code formatting is crucial for readability, collaboration, and overall code quality. However, manual formatting can be time-consuming and prone to errors. In this section, we will explore how to automate code formatting using Python Black and the Pre-Commit framework.

What is Python Black?

Python Black is a popular code formatter for Python that automatically formats your code to conform to the PEP 8 style guide. It is fast, efficient, and highly customizable.

What is Pre-Commit?

Pre-Commit is a framework that allows you to run checks and hooks on your code before committing it to your version control system. It is a great way to ensure that your code meets certain standards and conventions before it is committed.

Installing Python Black and Pre-Commit To get started, you'll need to install Python Black and Pre-Commit. You can do this using pip:

install Python Black and the Pre-Commit framework

```
1 pip install black pre-commit
```

Once installed, you'll need to configure Pre-Commit to use Python Black. Create a new file called `.pre-commit-config.yaml` in the root of your project with the following contents. You should check-in `.pre-commit-config.yaml` into your code repository.

```
.pre-commit-config.yaml
```

```
1 repos:
2   - repo: local
3     hooks:
4       - id: black
5         name: black
6         entry: black
7         language: python
8         types: [python]
```

Using Pre-Commit with Git To use Pre-Commit with Git, you'll need to install the Pre-Commit Git hook. Run the following command:

Install the Pre-Commit Git hook

```
1 pre-commit install
```

Pre-Commit should automatically format your code and prevent `git commit` from succeeding if the formatting is incorrect. Pre-Commit can also be applied manually.

Run Pre-Commit manually

```
1 pre-commit run --all-files # apply to all files in a repository
2 pre-commit run --files * # apply to all files in the current directory
```

14.2.2 CI/CD Platforms

For full automation across machines, CI/CD platforms run your code in cloud or managed environments. They detect code changes, trigger pipelines, run tests, and deploy software.

- **GitHub Actions:** Native to GitHub, workflows are written in YAML and triggered by events like pushes or pull requests.
- **GitLab CI:** Configuration lives in a `.gitlab-ci.yml` file; supports custom runners, including on GPU-based systems.
- **Jenkins:** A widely-used automation server based on Groovy pipelines and extensible with plugins.

14.2.3 Example: GitHub Actions Workflow

GitHub Actions lets you define custom workflows in YAML. Here's a minimal example to test Python code using pytest.

```
.github/workflows/ci.yml

1 # This workflow will install Python dependencies, run tests and lint with a single
   version of Python
2 # For more information see: https://docs.github.com/en/actions/automating-builds-
   and-tests/building-and-testing-python
3
4 name: Test Python with Pytest
5 on: push
6
7 jobs:
8   build:
9     runs-on: ubuntu-latest
10    steps:
11      - uses: actions/checkout@v4
12      - name: Set up Python 3.10
13        uses: actions/setup-python@v3
14        with:
15          python-version: "3.10"
16      - name: Install and run pytest
17        run: |
18          python -m pip install pytest
19          pytest
```

14.2.4 Example: GitLab CI

GitLab CI/CD pipelines are configured using a YAML file named `.gitlab-ci.yml` located in the root directory of a project. The file defines the sequence of jobs to be executed during testing, building, or deploying software. The following example shows how to define a single job named `pytest` that installs dependencies and runs unit tests using `pytest` in a Python environment.

```
.gitlab-ci.yml

1 stages:
2   - test
3
4 pytest:
5   stage: test
6   image: python:3.10
7   script:
8     - pip install pytest
9     - pytest
```

Running GitLab CI jobs locally using gitlab-runner. It is possible to test CI jobs locally, without pushing your code to a remote GitLab server. This can be useful for educational purposes, debugging, and iterative development. The GitLab runner binary includes a command `exec` shell which allows local execution of individual jobs defined in the CI YAML file.

Step 1: Install gitlab-runner locally (without sudo). You can download and install gitlab-runner as a standalone binary in your user space. Version 16.9.1 is recommended because the `exec` command was removed in later versions.

bash

```

1 mkdir -p $HOME/bin
2 cd $HOME/bin
3 curl -L --output gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/
      v16.9.1/binaries/gitlab-runner-linux-amd64
4 chmod +x gitlab-runner
5 export PATH="$HOME/bin:$PATH"

```

Step 2: Create a minimal project with test and CI configuration. The following Jupyter Notebook cell creates a folder `gitlab_demo`, writes a simple test file, adds a `.gitlab-ci.yml`, and initializes a Git repository with a first commit. This setup is necessary so that `gitlab-runner` can properly interpret the CI job.

python

```

1 import os
2 import subprocess
3 from pathlib import Path
4
5 # Set up folder
6 proj = Path("gitlab_demo")
7 proj.mkdir(exist_ok=True)
8
9 # Write test file
10 (proj / "test_sample.py").write_text("""
11 def add(a, b):
12     return a + b
13
14 def test_add():
15     assert add(2, 2) == 4
16 """
17
18 # Write GitLab CI YAML
19 (proj / ".gitlab-ci.yml").write_text("""
20 stages:
21 - test
22
23 pytest:
24   stage: test
25   script:

```

```

26     - pip install pytest
27     - pytest
28 """
29
30 # Initialize Git repository and make first commit
31 subprocess.run("git init", shell=True, cwd=proj)
32 subprocess.run("git config user.name 'CI Tester'", shell=True, cwd=proj)
33 subprocess.run("git config user.email 'ci@test.local'", shell=True, cwd=proj)
34 subprocess.run("git add .", shell=True, cwd=proj)
35 subprocess.run("git commit -m 'initial commit'", shell=True, cwd=proj)
36
37 print("Project created in ./gitlab\_demo")
38 print("Run this in your terminal to execute the CI job:")
39 print("  cd gitlab_demo")
40 print("  gitlab-runner exec shell pytest")

```

Step 3: Execute the job defined in `.gitlab-ci.yml`. After running the cell above, navigate to the directory `gitlab_demo` and run:

bash

```

1 cd gitlab_demo
2 gitlab-runner exec shell pytest

```

This will start the GitLab runner in shell mode and execute the pytest job exactly as it would be performed in a GitLab pipeline. The output should show that `pip install pytest` runs successfully and that the test function passes.

Important: GitLab Runner requires that your repository has at least one commit. If you forget to run `git commit`, you will see an error about the missing HEAD revision. Always initialize the Git repository and make an initial commit before running local jobs.

Interactive Notebook. You can try this example locally in the Jupyter notebook

`2_Gitlab_Runner_with_Docker.ipynb`,

which automates all steps: it creates the project, writes the CI configuration and test, and runs the pipeline inside a Docker container using `gitlab-runner`. You will need a working docker environment for this.

14.3 Testing with Pytest

Testing is a central pillar of CI/CD. For Python-based projects, `pytest` is a popular testing framework. It offers an intuitive syntax and powerful features like fixtures, parameterization, and integration with other tools.

This section introduces basic test writing and advanced usage patterns.

14.3.1 Writing Simple Tests

Simple tests help validate expected behavior and prevent regressions. The pytest framework auto-discovers test files and functions with the prefix `test_`.

`test_example.py`

```
1 def add(a, b):
2     return a + b
3
4 def test_answer():
5     assert add(1, 3) == 4
```

Notebook: See `3_Gitlab_Runner_ShellTest.ipynb` for a full working example. Probably you will need to execute this outside of jupyter. You then get

Output of Test.

```
1 (ropy_wsl) rolan@White-WIN:~/gitlab_demo_shell$ cd /home/rolan/gitlab_demo_shell
2 gitlab-runner exec shell pytest
3 Runtime platform                                     arch=amd64 os=linux pid=23241
   revision=782c6ecb version=16.9.1
4 fatal: ambiguous argument 'HEAD~1': unknown revision or path not in the working
   tree.
5 Use '--' to separate paths from revisions, like this:
6 'git <command> [<revision>...] -- [<file>...]'
7 Running with gitlab-runner 16.9.1 (782c6ecb)
8 Preparing the "shell" executor
9 Using Shell (bash) executor...
10 executor not supported                               job=1 project=0 referee=
   metrics
11 Preparing environment
12 Running on White-WIN...
13 Getting source from Git repository
14 Fetching changes...
15 Initialized empty Git repository in /home/rolan/gitlab_demo_shell/builds/0/project
   -0/.git/
16 Created fresh repository.
17 Checking out ff835cc9 as detached HEAD (ref is main)...
18 Skipping Git submodules setup
19 Executing "step_script" stage of the job script
20 $ pytest
21 ===== test session starts =====
22 platform linux -- Python 3.12.3, pytest-7.4.4, pluggy-1.3.0
23 rootdir: /home/rolan/gitlab_demo_shell/builds/0/project-0
24 plugins: anyio-4.2.0, langsmith-0.3.32, asyncio-0.23.7, cov-5.0.0, mock-3.14.0,
   recording-0.13.1
25 asyncio: mode=Mode.STRICT
26 collected 1 item
```

```

27
28 test_sample.py . [100%]
29
30 ===== 1 passed in 0.03s =====
31 Job succeeded
32 (ropy_wsl) rolan@White-WIN:~/gitlab_demo_shell$ [I 2025-05-20 21:27:59.540
33     ServerApp] Saving file at /3_Gitlab_Runner_ShellTest.ipynb
34 (ropy_wsl) rolan@White-WIN:~/gitlab_demo_shell$
```

14.3.2 Advanced Features

Pytest supports advanced testing features, making it suitable for both unit and integration testing. These include:

- Fixtures with `@pytest.fixture`
- Skipping tests conditionally
- Parametrized test cases
- GPU-specific testing
- Mocking external dependencies

Here's an example using monkeypatching to mock a function call in a test:

`monkeypatch`

```

1 import xarray, numpy
2
3 def my_processing(filename):
4     data = xarray.open_dataset(filename)
5     # some processing
6     return data
7
8 def open_dataset_mock(*kwargs, **args):
9     return xarray.Dataset({"X": numpy.arange(5)})
10
11 def test_processing(monkeypatch):
12     monkeypatch.setattr(xarray, "open_dataset", open_dataset_mock)
13     x = my_processing("no-name.nc")
14     assert x.X.sum() == 10
```

14.4 CI/CD Runners and Cloud Integration

To execute pipelines, CI/CD systems rely on agents called *runners*. These can be hosted (by GitHub/GitLab) or self-hosted (on-premises or cloud). This section covers how runners work and how to use custom infrastructure, such as GPU-accelerated systems, to scale your CI pipelines.

14.4.1 GitLab/GitHub Runners

Runners are services that execute CI/CD jobs. Hosted runners are maintained by the platform provider, while self-hosted runners allow customization.

- Hosted runners: pre-configured and maintained (default in GitHub Actions)
- Self-hosted runners: user-managed with support for GPUs or custom stacks
- Containers: jobs typically run in isolated Docker containers

14.4.2 Example: Self-Hosted Runner Setup

To set up a self-hosted runner (e.g., in the European Weather Cloud), follow these steps:

1. Launch a virtual machine with suitable hardware
2. Install Docker and prepare storage
3. Register the GitLab runner with the project token
4. Tag the runner for use with specific jobs
5. Reference the runner using tags in your CI pipeline

14.4.3 Best Practices

To ensure efficient and reliable CI/CD pipelines, follow these recommendations:

- Use code formatters and linters (e.g., black, flake8)
- Run fast checks locally using pre-commit
- Split unit and integration tests to reduce pipeline duration
- Cache dependencies to avoid redundant steps
- Use matrix builds to test across platforms and Python versions

14.5 CI/CD for ICON, AICON, and Anemoi

CI/CD enhances development workflows through automation, validation, and continuous improvement. For ML projects, these practices are essential to manage complexity, improve quality, and accelerate iteration.

Both ICON and Anemoi leverage the Pre-Commit framework for linting, ensuring that developers adhere to common coding styles. Each developer is expected to install and use Pre-Commit offline, while the same hooks are executed in GitLab (ICON) and GitHub (Anemoi) CI pipelines to enforce consistency.

Different model setups of ICON are used as integration tests, defined by experiment scripts that create model configuration namelists, prepare input data, and verify output data in some cases. The primary goal of each test is to ensure that the model compiles and does not crash due to software errors or numerical instability. These integration tests are mainly orchestrated by BuildBot.

The Anemoi framework consists of multiple interleaved packages, each implementing unit-tests for its functionality using Pytest. Most packages provide an extra set of dependencies used only for testing. To run Pytest in an Anemoi package, follow these steps:

Run Pytest in an Anemoi package

```
1 git clone https://github.com/ecmwf/anemoi-core.git
2 python -m venv venv
3 source venv/bin/activate
4 pip install -e anemoi-core/graphs[tests]
5 export CUDA_VISIBLE_DEVICES= # disable execution on GPU
6 pytest anemoi-core/graphs
```

A chain of GitLab CI pipelines was set up to automatically build the AICON inference container using Kaniko and Singularity. This ensures that the container is built consistently and efficiently, reducing the risk of human error.

Chapter 15

Anemol – AI-Based Weather Modeling

15.1 Yaml, Hydra and OmegaConf

Before we explore Anemol let us complete our preparation by looking at two main tools used.

15.1.1 YAML – Configuration Made Simple

YAML (YAML Ain't Markup Language) is a human-readable format used for configuration files. It is built on indentation and simple key-value structures, making it ideal for defining settings in machine learning workflows.

In the context of Anemol and many other ML frameworks, YAML is used to define model parameters, training hyperparameters, and paths. Here's a simple YAML file that defines both a model and a training configuration:

Model and Training Configuration

```
1 model:
2   name: simple_model
3   input_size: 10
4   hidden_size: 20
5   output_size: 1
6
7 training:
8   epochs: 5
9   batch_size: 32
10  learning_rate: 0.01
```

This configuration can be loaded in Python using the PyYAML library and interpreted as a dictionary. Here's an example of how to load it:

Python: Load YAML

```

1 import yaml
2
3 with open("simple_config.yaml", "r") as file:
4     config = yaml.safe_load(file)
5
6 print("Model name:", config["model"]["name"])
7 print("Training for", config["training"]["epochs"], "epochs")

```

You can then pass this configuration into a training function or class:

Python: Use Configuration

```

1 def train_model(config):
2     print(f"Training model '{config['model']['name']}'")
3     print("Hyperparameters:")
4     print("  Learning rate:", config["training"]["learning_rate"])
5     print("  Batch size:", config["training"]["batch_size"])
6     for epoch in range(config["training"]["epochs"]):
7         print(f"Epoch {epoch+1}... (training logic here)")
8
9 train_model(config)

```

This illustrates how YAML is used as an external, editable source of parameters for training code, enabling experiment reproducibility and flexible configuration.

15.1.2 Hydra – Flexible Configuration for ML Experiments

Hydra is a Python framework for managing complex configuration workflows. It allows users to compose and override YAML configuration files at runtime, enabling modular and reusable setups for machine learning experiments. In our example, we train a neural network to learn the function $y = \sin(x)$, and use Hydra to flexibly switch between different model architectures and training parameters.

Base Configuration: The central Hydra entry point is `config.yaml`, which specifies which model and training configuration to load:

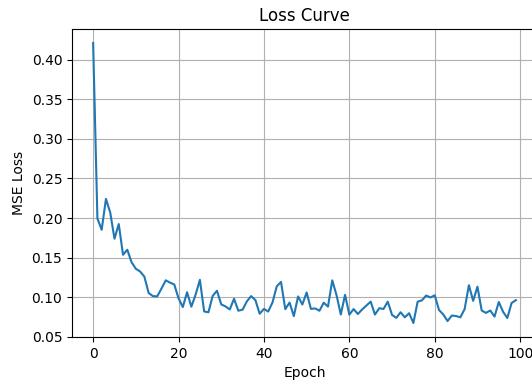
config.yaml

```

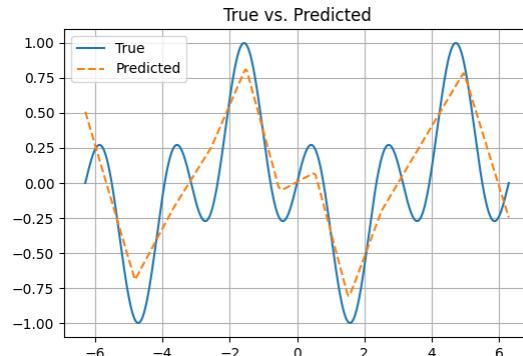
1 defaults:
2   - model: mlp
3   - training: simple
4   - _self_

```

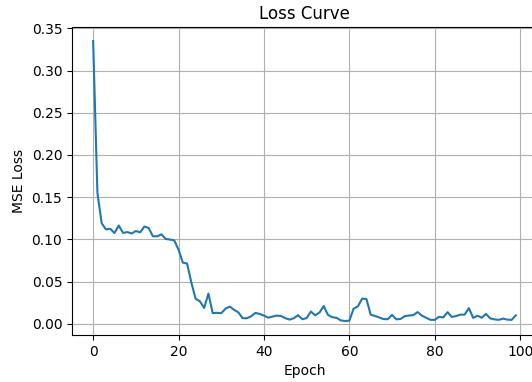
Shallow MLP Configuration: A simple neural network with one hidden layer is defined like this:



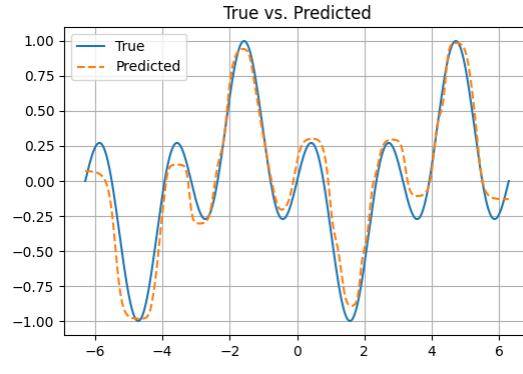
(a) Loss curve for the shallow model (mlp). Training converges quickly to a low loss.



(b) Prediction result for the shallow model. The curve closely matches the true $\sin(x)$ target.



(c) Loss curve for the deeper model (deep). Training is slower and convergence less stable.



(d) Prediction result for the deeper model. The curve is less smooth and overshoots the target in places.

Figure 15.1: Training results for two model configurations defined using Hydra. The shallow model (mlp) performs more effectively for approximating the simple $\sin(x)$ function, while the deeper model (deep) shows signs of overfitting or instability.

model/mlp.yaml

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

Deeper Network Configuration: A deeper network with three hidden layers can be defined similarly:

model/deep.yaml

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

Training Configuration: Training hyperparameters such as learning rate and batch size are defined in a separate file:

training/simple.yaml

```
1 epochs: 100
2 lr: 0.01
3 batch_size: 32
```

Using Hydra in Python: Hydra supports programmatic configuration loading and overrides using the initialize and compose functions. This allows the configuration to be composed dynamically at runtime, for example to switch between different models within the same training loop.

In the following example, the run_with_override function dynamically loads a specified model configuration (e.g., mlp or deep) and then trains the model accordingly.

Python

```
1 from hydra import initialize, compose
2
3 def run_with_override(model_name, title):
4     with initialize(config_path="conf", version_base=None):
5         cfg = compose(config_name="config", overrides=[f"model={model_name}"])
6         print(f"\n{'='*30}\nTraining model: {model_name.upper()} ({title})\n{'='*30}")
7         train_model(cfg)
8
9 # Run shallow model
10 run_with_override("mlp", "Shallow Network (1 hidden layer)")
```

This function uses the Hydra override mechanism:

Python

```
1 cfg = compose(config_name="config", overrides=[f"model={model_name}"])
```

to replace the model defined in config.yaml at runtime. For example, if model=deep is used as an override, then conf/model/deep.yaml is loaded instead of the default mlp.yaml.

To demonstrate this in practice, the following snippet dynamically creates a deeper model configuration file (if not already present) and then runs training with it:

Python

```
1 # Create deep model config if not already created
2 import os
3 if not os.path.exists("conf/model/deep.yaml"):
4     with open("conf/model/deep.yaml", "w") as f:
5         f.write("\n\n"
6 layer_sizes: [1, 128, 64, 32, 1]
7 activation: tanh
```

```

8  \"\"\"")
9
10 # Run deeper model
11 run_with_override("deep", "Deeper Network (3 hidden layers)")
```

This dynamic Hydra setup makes it easy to test and compare different model architectures or training configurations without modifying the underlying training code. It supports flexible experimentation and reproducibility, which is particularly important in complex machine learning pipelines like Anemoi.

15.1.3 OmegaConf Integration

Internally, Anemoi relies on OmegaConf to represent and manipulate the hierarchical configuration tree provided by Hydra. Configuration classes like BaseSchema are converted to and from DictConfig objects using OmegaConf utilities.

- **Library:** <https://omegacnf.readthedocs.io/>
- OmegaConf.to_object(...) is used to convert validated configs into native Python dictionaries.
- OmegaConf.resolve(...) ensures that interpolations and references are evaluated before use.

This layer of indirection allows for robust and modular configuration parsing, while still enabling dynamic instantiation via Hydra.

15.2 Introduction to Anemol

The Anemoi framework, developed by ECMWF and its partners, provides a modular and extensible system for training machine learning models in the context of numerical weather prediction and Earth system modelling. It is designed to facilitate the development, training, and deployment of advanced machine learning architectures tailored to meteorological data, with a focus on scalability, flexibility, and reproducibility.

The framework is organized into several core packages—responsible respectively for training loops, model definitions, data pipelines, and graph structures—that are now unified under a single repository: <https://github.com/ecmwf/anemoi-core>. This monorepo supersedes earlier standalone repositories (such as `anemoi-training` and `anemoi-models`), bringing all essential components together for coherent development and integration. This section outlines a structured approach for exploring and understanding the most important elements of the Anemoi codebase, including how data are loaded and preprocessed, how models are constructed, and how training is orchestrated using customizable strategies.

Install Anemoi to train a model

```
1 pip install anemoi-training
```

We need to formulate a warning: the anemoi repo has undergone frequent changes, our links might point to versions which have changed!

15.2.1 Training Loop

The training loop orchestrates the entire model training process. It is implemented within the anemoi-training package and uses PyTorch Lightning as its backend. The main entry point is the `train.py` script, which instantiates an `AnemoiTrainer` class. This class is responsible for loading configurations, initializing the model and data module, setting up logging and diagnostics, and executing the training via Lightning's `Trainer.fit()` call.

- **Documentation:** <https://anemoi.readthedocs.io/projects/training/en/latest/>
- **Main script:** `train.py` (in `anemoi-core`)
- **Configuration:** Training is driven by YAML files parsed via Hydra and OmegaConf. Example configurations can be found in the directory:
 - `training/config` (YAML configs for models, datasets, strategies, etc.)
- **Key modules:**
 - `strategy.py` – contains custom distributed training strategies.
 - `losses.py` – defines loss functions used during training.

When the `train.py` script is run, it loads a Hydra configuration, builds a model and data pipeline based on the selected YAML files, and begins training with the configured strategy. The modularity of the setup allows the user to customize all elements—data preprocessing, model architecture, logging, and training loop—by simply editing the YAML configuration.

15.2.2 Model Definitions

Model architectures and components are implemented in the `anemoi-models` package, which provides a flexible and modular system to define, compose, and interface neural networks for weather forecasting applications.

- **Documentation:** <https://anemoi.readthedocs.io/projects/models/en/latest/>
- **Source Code:** <https://github.com/ecmwf/anemoi-core/tree/main/models/src/anemoi/models>

The package is organized into several key modules:

- **models:** Implements various network architectures used in Anemoi, such as MLPs, U-Nets, and Graph Neural Networks.
GitHub: <https://github.com/ecmwf/anemoi-core/tree/main/models/src/anemoi/models/models>
- **layers:** Contains reusable components like attention mechanisms, residual blocks, and normalization layers that can be used across models.
GitHub: <https://github.com/ecmwf/anemoi-core/tree/main/models/src/anemoi/models/layers>
- **interface:** Defines the standard model interface used for integration into the Anemoi training loop. Models inherit from LightningModule and conform to standardized input/output conventions.
GitHub: <https://github.com/ecmwf/anemoi-core/tree/main/models/src/anemoi/models/interface>

These modules are configured and instantiated dynamically using Hydra, allowing flexible model selection and hyperparameter tuning via YAML configuration files.

15.2.3 Data Handling

The data loading and preprocessing logic is described in the `anemoi-datasets` documentation, which outlines how structured and graph-based meteorological datasets are processed. While the public source code for this package is not currently available, the interfaces are fully described and configurable via YAML and Hydra.

- **Documentation:** <https://anemoi.readthedocs.io/projects/datasets/en/latest/>
- **Source Code:** <https://github.com/ecmwf/anemoi-datasets>

Key components (as documented) include:

- **datasets:** For loading structured or graph-ready datasets from formats like GRIB or NetCDF.
- **preprocessing:** For transformations such as normalization, filtering, and feature generation.

Users define data behavior declaratively via YAML configuration files. These definitions are passed to the training and graph modules via Hydra, ensuring reproducibility and modularity across workflows.

15.2.4 Graph Structures

Graphs are used in Anemoi to represent spatial and temporal dependencies in meteorological data. These graph-based structures enable Graph Neural Networks (GNNs) to model interactions between grid cells or observation points effectively. The `anemoi-graphs` package provides tools for generating and managing these graph representations.

- **Documentation:** <https://anemoi.readthedocs.io/projects/graphs/en/latest/>
- **Source Code:** <https://github.com/ecmwf/anemoi-core/tree/main/graphs/src/anemoi/graphs>

Key components include:

- **graphs:** Provides utilities to construct and serialize graphs from model grids, using distance-based, nearest-neighbor, or mesh-specific methods (e.g., for ICON or IFS grids).
GitHub: <https://github.com/ecmwf/anemoi-core/tree/main/graphs/src/anemoi/graphs>
- **Integration with Models:** Graph objects are generated once and passed to the model and datamodule. This allows for consistent and reusable topology definitions across experiments. Graph data can be saved as PyTorch tensors and reloaded at runtime.

Graph definitions are fully configurable via Hydra YAML files. Users can specify graph types, node layouts, edge construction logic, and file paths, making the process flexible and reproducible across model setups.

15.3 ZARR, ERA and Datasets for Anemoi

Zarr is a flexible and efficient data format designed specifically for handling large numerical datasets commonly used in scientific computations. Unlike traditional data formats such as NetCDF and HDF5, Zarr stores arrays chunk-wise, which allows efficient reading, writing, and parallel processing—particularly beneficial in cloud computing and distributed environments. Zarr datasets are structured around arrays, groups, and metadata stored compactly in directories or cloud storage buckets, which makes them inherently scalable and accessible for high-performance scientific workflows.

A Zarr dataset is composed of:

- **Arrays:** Multidimensional numeric arrays stored in small, compressed chunks. Each array has its metadata, including shape, chunk shape, and compression method.
- **Groups:** Logical collections of arrays and possibly other groups, enabling hierarchical data organization similar to directories in a file system.
- **Metadata:** Information stored in JSON format that describes the structure, datatype, and compression scheme used, facilitating fast and straightforward access.

15.3.1 Downloading ERA5 2m Temperature from the Copernicus Climate Data Store

To download ERA5 2m air temperature data from the Copernicus Climate Data Store (CDS), follow these steps:

1. Register for a free account at <https://cds.climate.copernicus.eu>.
2. Install the CDS API client:

Shell Commands

```
1 pip install "cdsapi >=0.7.4"
```

3. Create a file named `.cdsapirc` in your home directory with the following content (replace with your actual personal access token from your CDS account page):

Configuration File

```
1 url: https://cds.climate.copernicus.eu/api
2 key: 128a88aa-bc19-4d14-a238-caae7472fc9b
```

4. Use the following Python script to download 2m temperature data for January 1, 2025 at 12:00 UTC:

Python Script

```
1 import cdsapi
2
3 client = cdsapi.Client()
4
5 client.retrieve(
6     'reanalysis-era5-single-levels',
7     {
8         'product_type': 'reanalysis',
9         'variable': ['2m_temperature'],
10        'year': ['2025'],
11        'month': ['01'],
12        'day': ['01'],
13        'time': ['12:00'],
14        'format': 'netcdf',           # Or 'grib' if preferred
15        'area': [90, -180, -90, 180], # Global coverage (optional)
16    },
17    'era5_t2m_2025_01_01_12.nc'
18 )
```

This script downloads the 2m air temperature field for the entire globe at 12:00 UTC on January 1, 2025. The result is saved as a NetCDF file named `era5_t2m_2025_01_01_12.nc`, which can be inspected and visualized using `xarray` and `cartopy`.

15.3.2 ZARR Structure and Conversion with Xarray

The practical use of Zarr is illustrated clearly in the provided notebook, which details the process of converting NetCDF files, commonly used for meteorological data such as ERA5 temperature data (T2M), into a Zarr format compatible with Anemol.

Initially, we load a NetCDF dataset using Xarray, a Python library for handling multi-dimensional arrays effectively:

Open_Era_after_Download

```
1 import xarray as xr
2
```

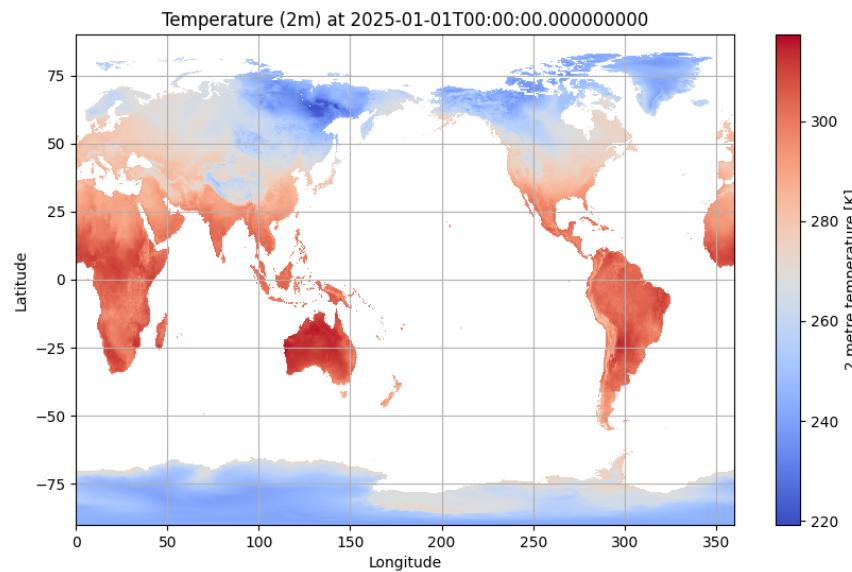


Figure 15.2: ERA5 2m air temperature on January 1, 2025 at 00 UTC. The temperature field is visualized from the high-resolution Zarr archive.

```
3 # Load Dataset
4 ds = xr.open_dataset('era_t2m.nc')
5 print(ds)
```

This step provides an overview of dataset variables and coordinates. For instance, to visualize temperature at the first time step:

Display_ERA_T2M_Field

```
1 import matplotlib.pyplot as plt
2
3 # Select First Time Step and Plot
4 t2m = ds['t2m'].isel(valid_time=0)
5 t2m.plot(cmap='coolwarm')
6 plt.title('ERA5 2m Temperatur (erste Zeitstufe)')
7 plt.show()
```

The conversion from NetCDF to Zarr involves renaming the primary data variable ('t2m') to 'data' for compatibility with Anemoi, a data analysis and visualization framework:

Convert_to_Zarr

```
1 import os
2 from anemoi.datasets.data import open_dataset, add_dataset_path
3
4 # Rename Variable and Convert to Zarr
```

```

5 netcdf_file_path = "era_t2m.nc"
6 zarr_archive_path = "era_t2m.zarr"
7
8 ds_netcdf = xr.open_dataset(netcdf_file_path)
9 ds_renamed = ds_netcdf.rename_vars({'t2m': 'data'})
10 ds_renamed.to_zarr(zarr_archive_path, mode='w', compute=True, consolidated=True)

```

After conversion, we inspect the resulting Zarr archive's contents:

Inspect_ZARR_File

```

1 # Inspect Zarr Structure
2 ds_zarr = xr.open_zarr("era_t2m.zarr", consolidated=True)
3 print(ds_zarr)
4
5 # List Variables and Coordinates
6 print("Variables in Zarr Dataset:")
7 for var in ds_zarr.data_vars:
8     print(f" - {var}")
9
10 print("\nCoordinates:")
11 for coord in ds_zarr.coords:
12     print(f" - {coord}: shape={ds_zarr[coord].shape}")

```

Visualization of data stored within the Zarr archive is straightforward, leveraging Xarray's built-in plotting capabilities:

Visualize_Zarr

```

1 # Visualize Zarr Data
2 import matplotlib.pyplot as plt
3
4 # Select First Time Step for Visualization
5 time_slice = ds_zarr.isel(valid_time=0)
6 time_slice.data.plot(cmap='coolwarm')
7 plt.title(f"Temperature (2m) at {str(time_slice.valid_time.values)}")
8 plt.xlabel("Longitude")
9 plt.ylabel("Latitude")
10 plt.grid(True)
11 plt.show()

```

Thus, Zarr provides an optimized storage solution facilitating efficient data handling, which integrates seamlessly with Anemoi for high-performance analysis and visualization.

Lets have a quick look at the structure of the ZARR directory.

```
(ropy_wsl) rolan@White-WIN:~/all/python_and_ml_tutorial/code/code16/era_t2m.zarr\$ ll
total 0
drwxrwxrwx 1 rolan rolan 512 May 24 10:27 data
drwxrwxrwx 1 rolan rolan 512 May 24 10:27 latitude
drwxrwxrwx 1 rolan rolan 512 May 24 10:27 longitude
```

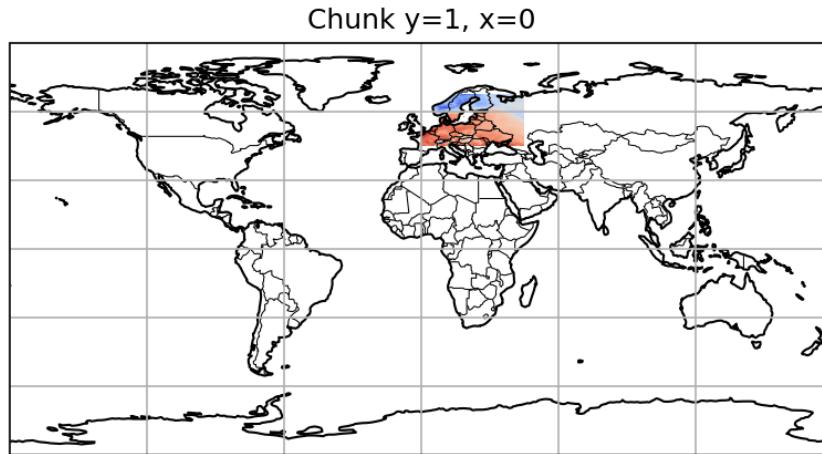


Figure 15.3: ERA5 2m Temperature from Zarr Chunk at Index (y=1, x=0). The chunk covers Central Europe and is extracted from the Zarr archive. Visualization uses Cartopy with coastlines and borders for orientation.

```
drwxrwxrwx 1 rolan rolan 512 May 24 10:27 number
drwxrwxrwx 1 rolan rolan 512 May 24 10:27 valid_time
```

and in the data/ directory we find:

```
(ropy_wsl) rolan@White-WIN:~/all/python_and_ml_tutorial/code/code16/era_t2m.zarr\$ cd data
(ropy_wsl) rolan@White-WIN:~/all/python_and_ml_tutorial/code/code16/era_t2m.zarr/data\$ ls
0.0.0 0.5.0 1.2.0 1.7.0 2.4.0 3.1.0 3.6.0 4.3.0 5.0.0 5.5.0 6.2.0 6.7.0 7.4.0
0.0.1 0.5.1 1.2.1 1.7.1 2.4.1 3.1.1 3.6.1 4.3.1 5.0.1 5.5.1 6.2.1 6.7.1 7.4.1
0.0.2 0.5.2 1.2.2 1.7.2 2.4.2 3.1.2 3.6.2 4.3.2 5.0.2 5.5.2 6.2.2 6.7.2 7.4.2
0.0.3 0.5.3 1.2.3 1.7.3 2.4.3 3.1.3 3.6.3 4.3.3 5.0.3 5.5.3 6.2.3 6.7.3 7.4.3
0.0.4 0.5.4 1.2.4 1.7.4 2.4.4 3.1.4 3.6.4 4.3.4 5.0.4 5.5.4 6.2.4 6.7.4 7.4.4
...
```

Chunk Content Example. To better understand how Zarr stores data internally, we examined a single chunk file named `0.0.0` within the directory `era_t2m.zarr/data/`. This chunk stores a sub-array of shape `(4, 226, 450)`, corresponding to the first four time steps and the northwestern corner of the global grid.

Most values in the upper part of the chunk are NaN due to the presence of land-sea masks or undefined polar values, but valid temperature values in Kelvin appear in the lower latitudes of this block:

Chunk shape: `(4, 226, 450)`

```
Sample values (in Kelvin):
[[[    nan ... 271.6 271.5]
  ...
  [    nan ... 271.7 271.6]]
 ...
 [[    nan ... 267.5 267.4]]]
```

15.4 Building an Icosahedral Graph with Anemoi

Very similarly to the ICON model framework, Anemoi provides tools to generate geodesic or icosahedral graphs suited for global data analysis. These graphs are constructed from refined icosahedrons, making them highly suitable for earth system modeling, weather prediction, and geospatial machine learning.

15.4.1 Generating the Graph Structure

GraphCreator, TriNodes, KNNEdges, and OmegaConf. The graph generation is controlled via a configuration dictionary, passed to the GraphCreator class. It defines the node layout and edge construction logic:

Graph Creation with GraphCreator, TriNodes, and KNNEdges

```
1 from anemoi.graphs.create import GraphCreator
2 from omegaconf import OmegaConf
3
4 config = OmegaConf.create({
5     "nodes": {
6         "ico_nodes": {
7             "node_builder": {
8                 "_target_": "anemoi.graphs.nodes.builders.from_refined_icosahedron
9 .TriNodes",
10                "resolution": 4,
11                "name": "ico_nodes"
12            },
13            "attributes": {}
14        }
15    },
16    "edges": [
17        {
18            "source_name": "ico_nodes",
19            "target_name": "ico_nodes",
20            "edge_builders": [
21                {
22                    "_target_": "anemoi.graphs.edges.builder.KNNEdges",
23                    "num_nearest_neighbours": 6
24                }
25            ]
26        }
27    ]
28}
```

```

25         }
26     ]
27 })
28
29 creator = GraphCreator(config)
30 graph = creator.create()

```

TriNodes and resolution. The TriNodes builder constructs a set of points on the surface of the globe by refining an icosahedron. The resolution parameter controls the granularity: higher values lead to finer meshes.

KNNEdges and edge_index. The KNNEdges builder establishes connections between each node and its nearest neighbors. The resulting edge list is stored as edge_index in the graph structure.

15.4.2 Visualization on a 2D Map

The graph can be rendered on a 2D world map using Cartopy. Each node is plotted based on its latitude and longitude, and edges are drawn as lines connecting the connected nodes.

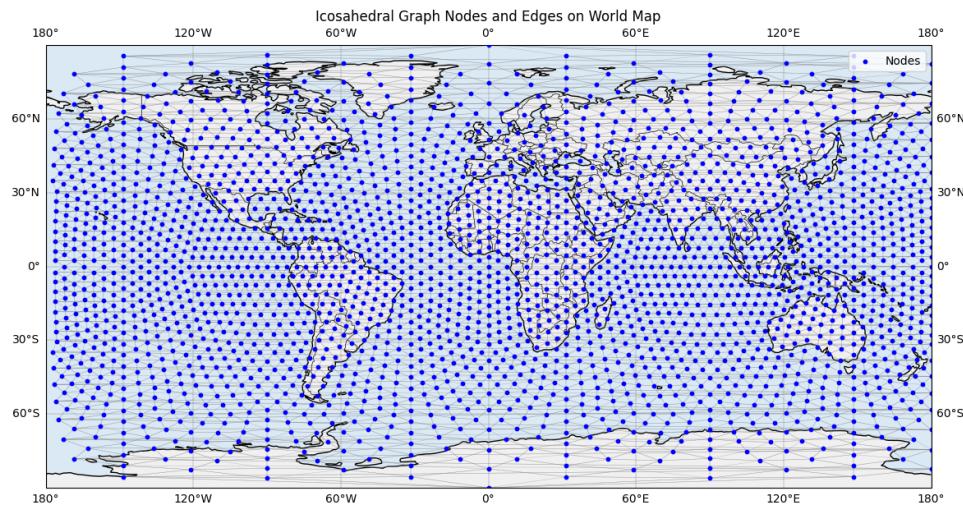


Figure 15.4: Icosahedral graph constructed using TriNodes with resolution 4. The nodes and their nearest-neighbor edges are plotted on a global Plate Carrée map.

15.4.3 Orthographic Globe Projection

Orthographic and Geodetic. To emphasize the spherical nature of the graph, a globe visualization is generated using the orthographic projection. This highlights how the graph tiles the earth's surface in a nearly uniform manner.

Great Circle and edge curvature. Edges on the globe are rendered using great-circle arcs to reflect the shortest path between nodes on a sphere, implemented by setting the transform

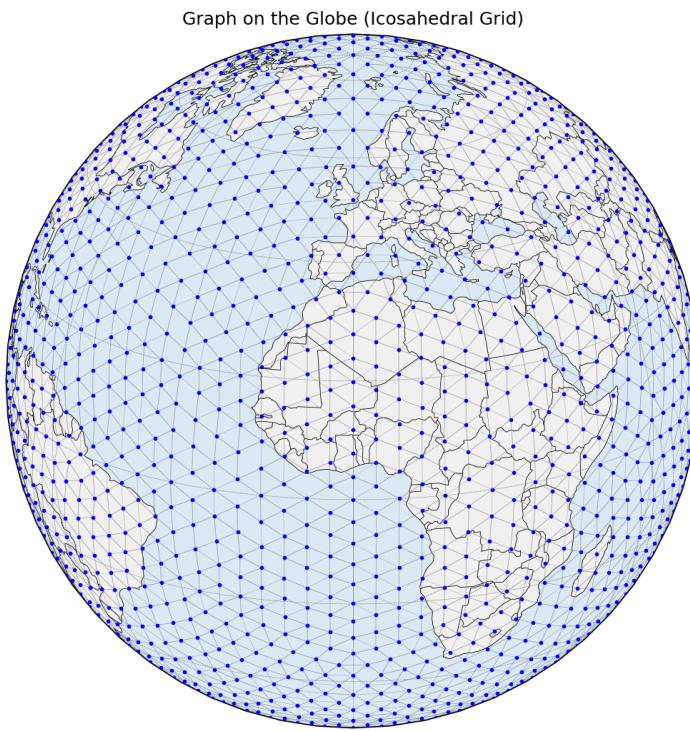


Figure 15.5: The same Anemoi icosahedral graph visualized on a globe using orthographic projection and geodetic edge rendering.

to `ccrs.Geodetic()`. This workflow shows how Anemoi enables modular, reproducible graph construction with spherical geometry, suitable for earth system data and machine learning models.

15.5 Hands-On Datasets, Validation, Training With Anemoi

We have already shown that creating zarr is possible by several tools. Anemoi uses these tools, but has created its own interface to add all the metadata which is then used by its training modules.

Create ZARR from GRIB

```

1 (ropy_wsl) rolan@White-WIN:~/all/python_and_ml_tutorial/code/code16\$ anemoi-
    datasets create download.yaml download.zarr --overwrite
2 2025-05-24 14:49:15 INFO Task init({},{}) starting
3 2025-05-24 14:49:16 INFO Setting flatten_grid=True in config
4 2025-05-24 14:49:16 INFO Setting ensemble_dimension=2 in config
5 2025-05-24 14:49:16 INFO Setting flatten_grid=True in config
6 2025-05-24 14:49:16 INFO Setting ensemble_dimension=2 in config
7 2025-05-24 14:49:17 INFO {'start': datetime.datetime(2024, 3, 1, 13, 0), 'end':
    datetime.datetime(2024, 3, 2, 12, 0), 'frequency': '24h', 'group_by': 'monthly
    '}
```

```

8 2025-05-24 14:49:17 INFO Groups(dates=1,StartEndDates(2024-03-01
    13:00:00..2024-03-02 12:00:00 every 1 day, 0:00:00))
9 2025-05-24 14:49:17 INFO Groups: Groups(dates=1,StartEndDates(2024-03-01
    13:00:00..2024-03-02 12:00:00 every 1 day, 0:00:00))
10 2025-05-24 14:49:25 INFO Minimal input for 'init' step (using only the first date)
    : GroupOfDates(dates=['2024-03-01T13:00:00'])
11 2025-05-24 14:49:25 INFO JoinResult: 1 dates (2024-03-01T13:00)
12     grib(GroupOfDates(dates=['2024-03-01T13:00:00']))
13 2025-05-24 14:49:25 INFO Config loaded ok:
14 2025-05-24 14:49:25 INFO Found 1 datetimes.
15 2025-05-24 14:49:25 INFO Dates: Found 1 datetimes, in 1 groups:
16 2025-05-24 14:49:25 INFO Missing dates: 0
17 2025-05-24 14:49:27 INFO Found 1 variables : z_1000.
18 2025-05-24 14:49:27 INFO Found 1 ensembles : 0.
19 2025-05-24 14:49:27 INFO gridpoints size: [1038240, 1038240]
20 2025-05-24 14:49:27 INFO resolution=0.25
21 2025-05-24 14:49:27 INFO total_shape = [1, 1, 1, 1038240]
22 2025-05-24 14:49:27 INFO chunks=(1, 1, 1, 1038240)
23 2025-05-24 14:49:27 INFO Creating Dataset 'download.zarr', with total_shape=[1, 1,
    1, 1038240], chunks=(1, 1, 1, 1038240) and dtype='float32'
24 2025-05-24 14:49:27 WARNING Dataset name error: the dataset name 'download' does
    not follow naming convention. Does not match ^(\w+)-([\w-]+)-(\w+)-(\w+)-(\d\dd
    \d\dd)-(\d\dd\d\dd\d\dd)-(\d+h|\d+m)-v(\d+)-?([a-zA-Z0-9-]+)?\$
25 2025-05-24 14:49:27 INFO Number of years 0 < 10, leaving out 20%. end=numpy.
    datetime64('2024-03-01T13:00:00')
26 2025-05-24 14:49:28 INFO Will compute statistics from 2024-03-01T13:00:00 to
    2024-03-01T13:00:00
27 2025-05-24 14:49:28 INFO Task init(() ,{}) completed (0:00:12.152389)
28 2025-05-24 14:49:28 INFO Task load(() ,{}) starting
29 2025-05-24 14:49:28 INFO {'end': '2024-03-02T12:00:00', 'frequency': '24h', ,
    'group_by': 'monthly', 'start': '2024-03-01T13:00:00'}
30 2025-05-24 14:49:28 INFO Groups(dates=1,StartEndDates(2024-03-01
    13:00:00..2024-03-02 12:00:00 every 1 day, 0:00:00))
31 2025-05-24 14:49:28 INFO Loading array shape=(1, 1, 1, 1038240), indexes=1
32 Loading 0/1: 100 1/1 [00:00<00:00, 279.19it/s]
33 2025-05-24 14:49:28 INFO Computing statistics for (1, 1, 1, 1038240) array
34 2025-05-24 14:49:28 INFO Statistics computed for 1 variables.
35 2025-05-24 14:49:28 INFO Flush data array
36 2025-05-24 14:49:28 INFO Flushed data array
37 2025-05-24 14:50:17 INFO Saving file at /6_Anemol_Training.ipynb
38 2025-05-24 14:50:29 INFO Name : /data
39 Type : zarr.core.Array
40 Data type : float32
41 Shape : (1, 1, 1, 1038240)
42 Chunk shape : (1, 1, 1, 1038240)
43 Order : C
44 Read-only : True
45 Compressor : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
46 Store type : zarr.storage.DirectoryStore

```

```

47 No. bytes          : 4152960 (4.0M)
48 No. bytes stored   : 1968080 (1.9M)
49 Storage ratio      : 2.1
50 Chunks initialized : 1/1
51
52 2025-05-24 14:50:29 INFO Task load((),{}) completed (0:01:01.203754)
53 2025-05-24 14:50:29 INFO Task finalise((),{}) starting
54 2025-05-24 14:50:29 INFO Variables minimum maximum mean stdev has_nans
55 z_1000 -4286.61 3106.39 715.22 1190.58 0.00
56 2025-05-24 14:50:29 INFO Wrote statistics in download.zarr
57 Computing size of download.zarr: 16it [00:00, 128.73it/s]
58 2025-05-24 14:50:29 INFO Total size: 2 MiB
59 2025-05-24 14:50:29 INFO Total number of files: 75
60 2025-05-24 14:50:29 INFO Task finalise((),{}) completed (0:00:00.657181)
61 2025-05-24 14:50:29 INFO Task init_additions((),{}) starting
62 2025-05-24 14:50:29 WARNING No delta found in kwargs, no additions will be
    computed.
63 2025-05-24 14:50:29 INFO Task init_additions((),{}) completed (0:00:00.000138)
64 2025-05-24 14:50:29 INFO Task run_additions((),{}) starting
65 2025-05-24 14:50:29 WARNING No delta found in kwargs, no additions will be
    computed.
66 2025-05-24 14:50:29 INFO Task run_additions((),{}) completed (0:00:00.000105)
67 2025-05-24 14:50:29 INFO Task finalise_additions((),{}) starting
68 2025-05-24 14:50:29 WARNING No delta found in kwargs, no additions will be
    computed.
69 Computing size of download.zarr: 16it [00:00, 130.81it/s]
70 2025-05-24 14:50:30 INFO Total size: 2 MiB
71 2025-05-24 14:50:30 INFO Total number of files: 75
72 2025-05-24 14:50:30 INFO Task finalise_additions((),{}) completed (0:00:00.226333)
73 2025-05-24 14:50:30 INFO Task patch((),{}) starting
74 2025-05-24 14:50:30 INFO Remove _create_yaml_config
75 2025-05-24 14:50:30 INFO Dataset changed by patch
76 2025-05-24 14:50:30 INFO Task patch((),{}) completed (0:00:00.364498)
77 2025-05-24 14:50:30 INFO Task cleanup((),{}) starting
78 2025-05-24 14:50:30 INFO Task cleanup((),{}) completed (0:00:00.007262)
79 2025-05-24 14:50:30 INFO Task verify((),{}) starting
80 2025-05-24 14:50:30 INFO Verifying dataset at download.zarr
81 2025-05-24 14:50:30 INFO download.zarr
82 2025-05-24 14:50:30 INFO Task verify((),{}) completed (0:00:00.021160)
83 2025-05-24 14:50:30 INFO Create completed in 1 minute 14 seconds
84 (ropy_wsl) rolan@White-WIN:~/all/python_and_ml_tutorial/code/code16\$
85 \end{lstlisting}
86
87 You can then inspect the dataset you generated:
88 \begin{codeonly}{Inspect generated ZARR}
89 (ropy_wsl) rolan@White-WIN:~/all/python_and_ml_tutorial/code/code16\$ anemoi-
    datasets inspect download.zarr
90 Path          : download.zarr
91 Format version: 0.30.0

```

```

92
93 Start      : 2024-03-01 13:00
94 End       : 2024-03-01 13:00
95 Frequency   : 1d
96 Missing     : 0
97 Resolution  : 0.25
98 Field shape: [721, 1440]
99
100 $1 \times 1 \times 1 \times 1,038,240$
101 Size      : 2 MiB (2 MiB)
102 Files     : 75
103 \begin{tabular}{|c|c|r|r|r|r|}
104 \hline
105 \textbf{Index} & \textbf{Variable} & \textbf{Min} & \textbf{Max} & \textbf{Mean} &
106 \textbf{Stdev} \\
107 \hline
108 0 & z\_1000 & -4286.61 & 3106.39 & 715.219 & 1190.58 \\
109 \hline
110 \end{tabular}
111 Dataset ready, last update 2 hours ago.
111 Statistics ready.

```

We can validate a debug.yaml as follows:

Validate YAML

```

1 (ropy_wsl) rolan@White-WIN:~/all/python_and_ml_tutorial/code/code16$ anemoi-
    training config validate --config-name debug.yaml
2 2025-05-24 17:00:42 INFO Validating configs.
3 2025-05-24 17:00:42 WARNING Note that this command is not taking into account if
    your config has set                               the config_validation flag to false. So
    this command will validate the config regardless of the flag.
4 2025-05-24 17:00:42 INFO Prepending current user directory (/mnt/c/Users/rolan/all
    /python_and_ml_tutorial/code/code16) to the search path.
5 2025-05-24 17:00:42 INFO Search path is now: [provider=anemoi-cwd-searchpath-
    plugin, path=/mnt/c/Users/rolan/all/python_and_ml_tutorial/code/code16,
    provider=hydra, path=pkg://hydra.conf, provider=main, path=/mnt/c/Users/rolan/
    all/ropy_wsl/lib/python3.12/site-packages/anemoi/training/commands]
6 2025-05-24 17:00:43 INFO Config files validated.

```

And then training can be initialized by:

Training Initialization

```
1 anemoi-training train --config-name=debug
```

15.6 Training Pipeline in Anemoi

The training process in Anemoi is managed by the script `train.py` in the `anemoi-training` package. It implements a highly modular, Hydra-driven architecture built on top of PyTorch Lightning. The main orchestration is encapsulated in the `AnemoiTrainer` class, which configures, initializes, and runs model training and evaluation.

To get started, you can generate a default configuration file as we showed above.

15.6.1 Hydra Entry Point and Configuration

The training is launched via a Hydra-decorated `main` function at the end of `train.py`. This function serves as the command-line entry point for Anemoi training jobs. Hydra parses the YAML configuration files into a nested `DictConfig` object and injects it into the trainer.

The `main` function's definition and entry point can be found here: <https://github.com/ecmwf/anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L497-L500>

Hydra Entry Point

```
1 @hydra.main(version_base=None, config_path="..config", config_name="config")
2 def main(config: DictConfig) -> None:
3     AnemoiTrainer(config).train()
```

The configuration system supports:

- Model and data module selection
- Training hyperparameters
- Graph and truncation settings
- Logging, callbacks, and checkpointing

15.6.2 AnemoiTrainer Class Overview

The `AnemoiTrainer` class is the core wrapper for the entire experiment lifecycle. It performs validation and conversion of the Hydra configuration schema, sets up metadata, seeds, run IDs, and logging paths, constructs or loads graph and truncation data, initializes the data module and model via Hydra, and creates callbacks and loggers (MLflow, TensorBoard, W&B).

The `AnemoiTrainer` class definition begins at: <https://github.com/ecmwf/anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L51-L494>

Many of these components are defined as `@cached_property`, ensuring lazy and consistent instantiation. For instance, the properties for the datamodule, graph data, and model are defined as:

Core Cached Properties

```

1 @property
2 def datamodule(self) -> Any: ... # Definition at https://github.com/ecmwf/anemoi-
   core/blob/main/training/src/anemoi/training/train/train.py#L101-L105
3 @property
4 def graph_data(self) -> HeteroData: ... # Definition at https://github.com/ecmwf/
   anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L157-L162
5 @property
6 def model(self) -> pl.LightningModule: ... # Definition at https://github.com/
   ecmwf/anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L190-
   L200

```

15.6.3 Model and Data Instantiation

The model and data modules are instantiated dynamically from the YAML config using Hydra utilities within the `AnemoiTrainer` class. This allows complete flexibility, enabling different models and datasets to be specified per experiment without requiring code changes.

The instantiation of the model typically occurs in the `model` cached property: <https://github.com/ecmwf/anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L195-L199>

The instantiation of the `datamodule` occurs in the `datamodule` cached property: <https://github.com/ecmwf/anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L102-L104>

Dynamic Instantiation

```

1 model_task = get_class(self.config.training.model_task) # See line 196
2 model = model_task(
3     config=self.config,
4     data_indices=self.data_indices,
5     graph_data=self.graph_data,
6     # ... other arguments ...
7 ) # See lines 197-199
8
9 datamodule = instantiate( # See line 102
10     convert_to_omegaconf(self.config).datamodule,
11     convert_to_omegaconf(self.config),
12     self.graph_data,
13 ) # See lines 103-104

```

15.6.4 Training Execution via Lightning

The actual training is launched by calling the `train()` method of the `AnemoiTrainer` class. This method constructs a `pl.Trainer` instance and calls its `fit()` method, leveraging PyTorch Lightning for the training loop management.

The `train` method is defined at: <https://github.com/ecmwf/anemoi-core/blob/main/training/>

[src/anemoi/training/train/train.py#L455-L494](https://github.com/ecmwf/anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L455-L494)

The `pl.Trainer` instance is initialized at: <https://github.com/ecmwf/anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L486-L493>

And the `trainer.fit` call that starts the training process is at: <https://github.com/ecmwf/anemoi-core/blob/main/training/src/anemoi/training/train/train.py#L486>

Lightning Training Loop

```

1 trainer = pl.Trainer( # Initialized at lines 489-493
2     accelerator=self.accelerator,
3     callbacks=self.callbacks,
4     strategy=self.strategy,
5     logger=self.loggers,
6     max_epochs=self.config.training.max_epochs,
7     # ... other arguments ...
8 )
9 trainer.fit( # Called at lines 495-498
10    self.model,
11    datamodule=self.datamodule,
12    ckpt_path=None if self.load_weights_only else self.last_checkpoint,
13 )

```

PyTorch Lightning handles the boilerplate of device placement, training/validation looping, and checkpointing. Logging is automatically integrated with MLflow or Weights & Biases depending on the configuration.

15.6.5 Configuration and Schema Management

Anemoi uses configuration schemas defined in `schemas/base_schema.py` to validate and standardize the structure of the configuration. This ensures consistency across model runs and supports reproducibility by recording full config metadata.

The main configuration schema definitions can be found here: https://github.com/ecmwf/anemoi-core/blob/main/training/src/anemoi/training/schemas/base_schema.py#L45

Key components of the schema system include:

- `BaseSchema`: The primary validated configuration interface.
- `UnvalidatedBaseSchema`: Used for more lenient configuration parsing, often during initial setup or debugging.
- Conversion utilities: Ensure compatibility with OmegaConf and Hydra's internal data structures.

15.6.6 Summary

The Anemoi training pipeline brings together Hydra for flexible configuration, PyTorch Lightning for robust training infrastructure, and specific data tools for handling graph and weather model

inputs. The `AnemoiTrainer` class encapsulates the orchestration of these components, making experiments declarative, reproducible, and modular.

Chapter 16

AI Transformation: From Communication to Neural Forecasting

16.1 From Communication History to AI Assistants (1440–2022)

The current AI transformation in weather services is best understood as part of a longer technology trajectory. In a simplified perspective, progress is driven by two interacting dimensions:

- **communication bandwidth and reach** (how information spreads),
- **computation and interaction** (how humans can act on information).

The key point is not only increasing information availability, but a shift of *interaction mode*. The web made information accessible to everyone; smartphones and social media made communication continuous; and LLMs now add a new step: **natural language becomes an interface to knowledge and action**.

1440: the printing press. With the printing press (Gutenberg era), information distribution changed from manual copying to scalable replication. Knowledge became reproducible at low marginal cost and could spread reliably across regions and generations. In the long run, this created the cultural and institutional base for mass education, public science, and standardized documentation — all prerequisites for later technological transformation waves.

1993: the public web transformation wave. The early 1990s mark the transition from local computing to global information access. The web turned digital knowledge into a shared resource, and created the first generation of self-service information workflows.

Smartphones and social media: interaction becomes permanent. Smartphones and social media created permanent connectivity. For weather services, this shift already changed user expectations: information must be *immediate, personalized, and always available*.

2012–2019: ML revolution in vision and translation. The next ML wave was driven by machine learning successes in computer vision and machine translation. Deep learning introduced a practical route from “raw data” to semantic representations.

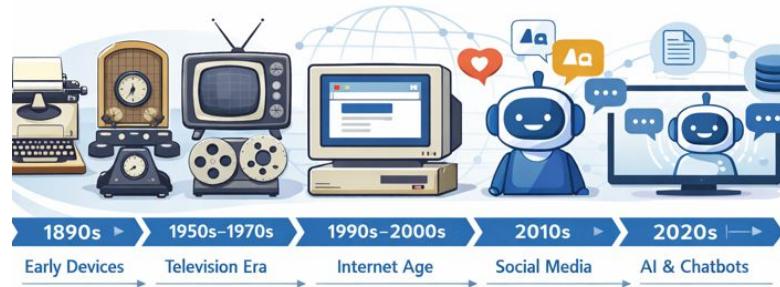
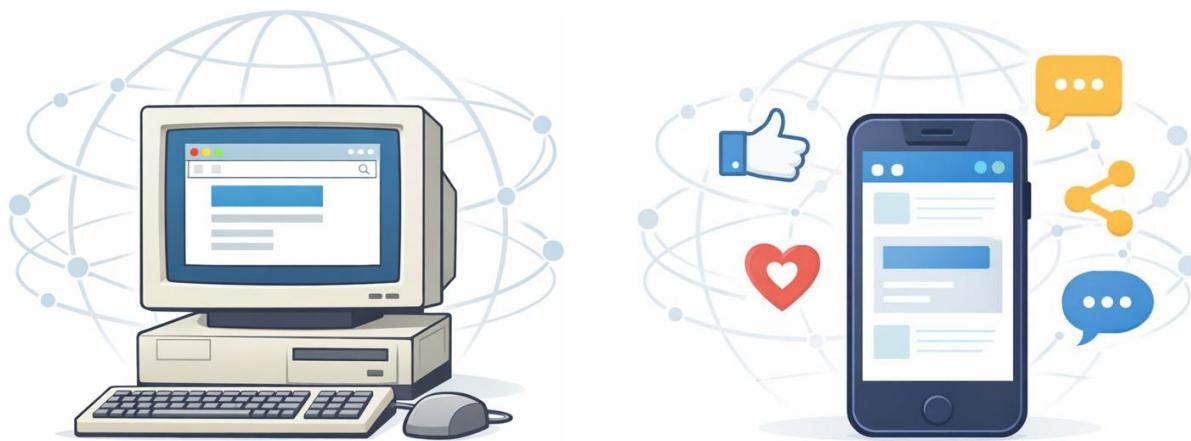


Figure 16.1: Communication history and the step to LLM-era interaction. The relevant shift is not only access to information but the ability to *interact* with information systems in natural language.

Era / technology step	Main change in communication & interaction
Printing press	scalable information distribution (mass replication)
Telegraph	long-distance messaging becomes near-instant
Radio	one-to-many broadcast to large audiences
Television	audiovisual mass media, daily public information streams
Computers	local digital workflows and computation for everyone
Internet / WWW	global access to knowledge and services
Social media / smartphones	permanent connectivity and personalized feeds
LLMs and AI assistants	natural language becomes an interface to knowledge and action

Table 16.1: Simplified communication timeline and the shift towards AI assistants. The key transition is from information access to natural-language interaction with tools and workflows.

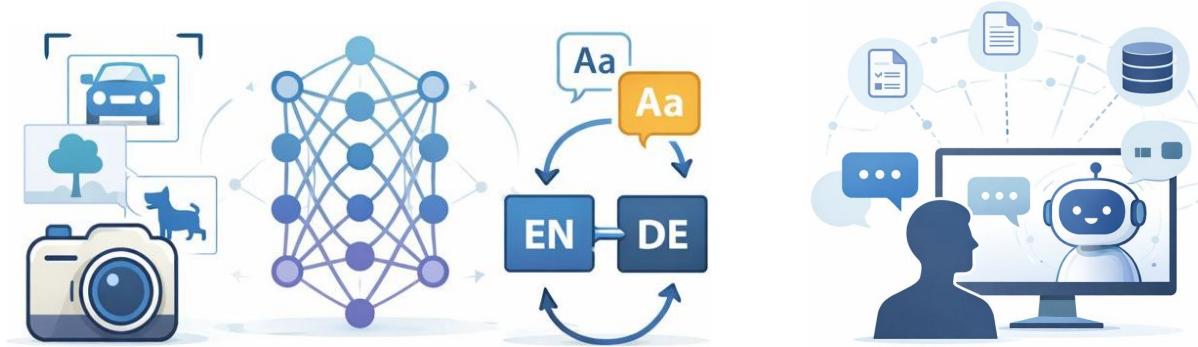
2022: chatbots cross a usability threshold. With chatbots, interaction itself changes. Instead of systems requiring specialized commands and training, systems can accept natural language inputs and iteratively refine the output in a dialogue. For weather services this marks a qualitative shift: **language can control complex systems.**



1993: the public web. Access to knowledge and digital workflows scales globally.

Smartphones. Continuous interaction and highly personalized information consumption.

Figure 16.2: From global access (web) to permanent interaction (smartphones and social media).



Deep learning revolution (2012–2019). Large improvements in vision and translation.

2022: Chatbots become practical. Dialogue enables iterative refinement.

Figure 16.3: Key milestones: deep learning breakthroughs in vision/translation and the rise of practical chatbots.

16.2 What an LLM is: tokens, attention, and next-token training

To understand language models, we first need a numerical representation of language. A computer cannot operate directly on words. Instead, text is translated into **tokens**, which are integer identifiers for pieces of text (words, subwords, or even single characters). Formally, tokenization maps a text string into a sequence

$$\text{text} \longrightarrow (t_1, t_2, \dots, t_n),$$

where each t_k is an integer index from a finite vocabulary.

These integers are not yet meaningful by themselves. Therefore, each token is mapped to a vector of real numbers by an **embedding**. This embedding is the entry point into a high-dimensional *representation space*:

$$t_k \longrightarrow e(t_k) \in \mathbb{R}^d.$$

Intuitively, the embedding vectors place tokens into a space in which similar meanings and similar

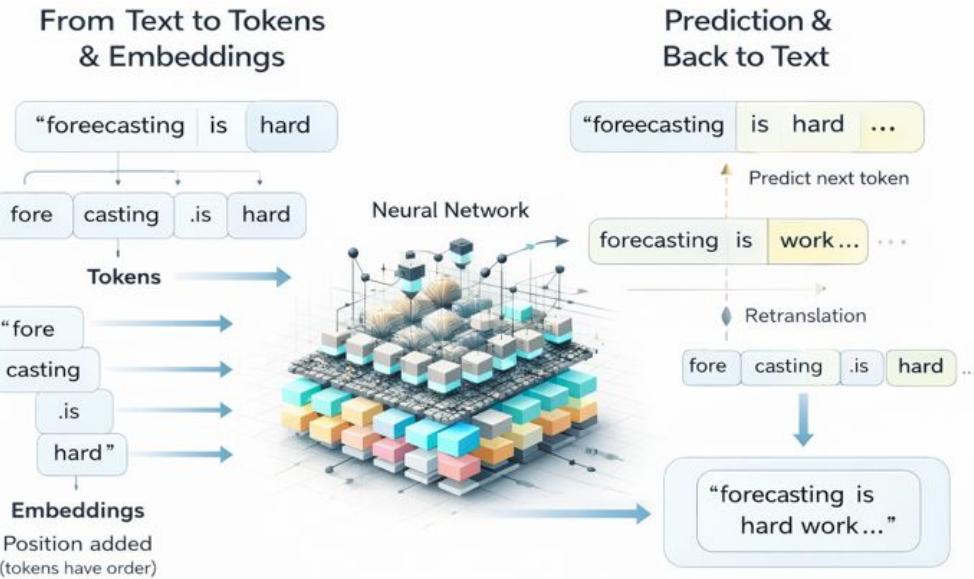


Figure 16.4: Transformer-based language models: tokenization, embeddings, attention, and stacked blocks.

usage contexts tend to be represented by vectors that are closer to each other (in a learned sense). With this, language becomes a numerical sequence of vectors, and the model can process it using linear algebra operations.

A Large Language Model (LLM) can be summarized in one sentence:

An LLM predicts the next token in a sequence, trained on massive text data.

While this sounds simple, the emergent capabilities are substantial because:

1. the training corpus encodes vast amounts of human knowledge and styles,
2. the model learns representations that compress regularities of language,
3. the model performs powerful conditional generation at inference time.

Tokenization and embeddings. Text is converted into discrete tokens t_1, t_2, \dots, t_n . Each token is mapped to a vector embedding $e(t_k) \in \mathbb{R}^d$. The model processes the sequence of embeddings:

$$E = (e(t_1), e(t_2), \dots, e(t_n)).$$

Self-attention. Self-attention lets each word (or token) *look at all other words* and decide which ones matter. It computes weights that depend on the current context and then forms a weighted combination of the other tokens.

A simplified attention equation for one head is

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V,$$

where Q, K, V are learned linear projections of the embeddings. Intuitively, the dot products QK^\top measure how strongly two tokens are related in the current context. The softmax turns these scores into weights, so that important tokens contribute more to the updated representation. The resulting attention matrix is therefore a learned, input-dependent connectivity structure.

For our course narrative, one should remember:

Attention enables global context integration: each token can attend to relevant other tokens.

Training objective: next-token prediction. At training, the model parameters θ are optimized to maximize the likelihood

$$p_\theta(t_{k+1} | t_1, \dots, t_k)$$

for all positions k in the training data. This “simple” objective is sufficient to create models that generate coherent text, perform reasoning-like behavior, and synthesize information.

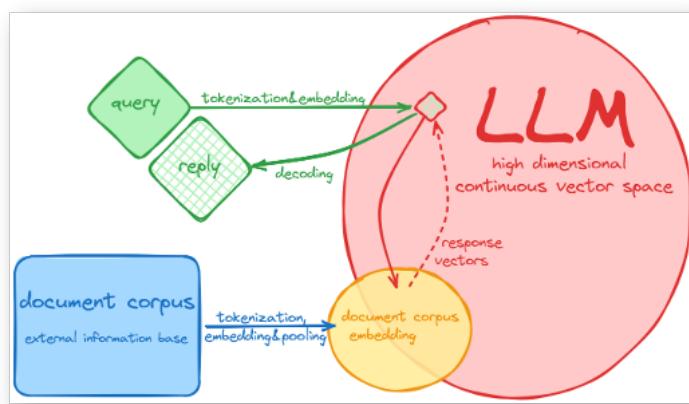


Figure 16.5: RAG concept: retrieval from external knowledge sources complements the model’s parametric memory.

16.3 From Chat to Systems: tools, retrieval (RAG), and agents

In practice, LLMs become most valuable when they are embedded into systems. Three extensions are essential:

- **Tools / function calling:** call deterministic functions and services,
- **Retrieval (RAG):** access external documents and domain data,

- **Memory and workflow context:** maintain state over a task.

Functions vs. agents. In modern assistant architectures, a central design question is:

- **agent-centric:** the LLM plans, calls tools, checks results, iterates.
- **function-centric:** the LLM selects and calls explicit functions (including agents as functions),

This is not a binary choice; practical systems combine both. In meteorology, the tool landscape is rich: datasets, verification tools, plotting, interpolation, model runs, documentation, and communication channels.

Important: agents are tools too. Conceptually, an agent can be exposed to the LLM as just another function call: `run_agent(task, context)`. The LLM triggers the agent, the agent executes an iterative tool loop (planning, calling tools, verifying results), and returns a structured outcome. In this sense, *function-centric orchestration and agent-centric behavior are fully compatible*: a single deterministic function call can launch a complex agent workflow.

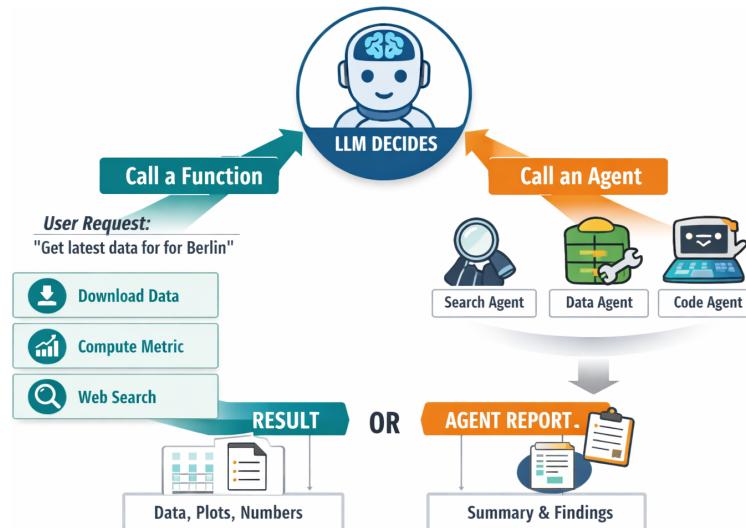


Figure 16.6: Assistant architecture idea: combine language-based interaction with tool usage and structured workflows.

A weather service viewpoint. From a weather service perspective, the key promise is:

Natural language becomes the universal access layer for data and tools.

This enables a new interaction model: operators, forecasters, and developers can drive complex pipelines by dialogue, with tool execution handling the deterministic core operations.

16.4 Neural Forecasting: learning motion, fronts, and rollouts

Weather prediction is fundamentally about **spatio-temporal pattern evolution**. A central question for the AI transformation is:

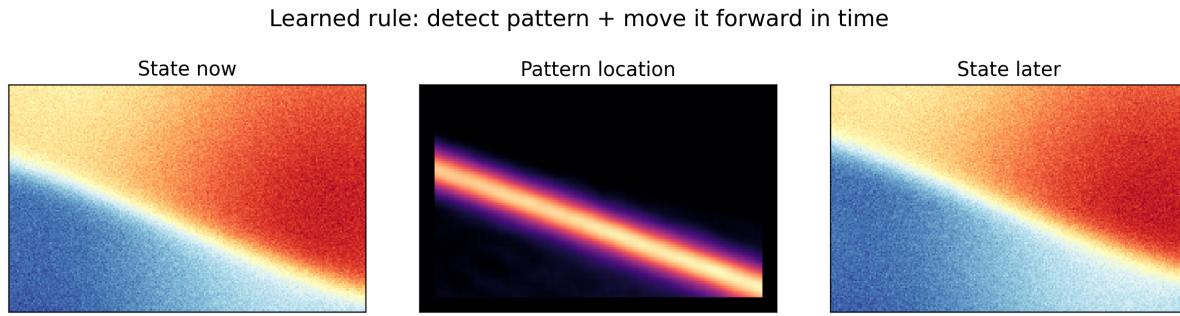


Figure 16.7: Neural forecasting intuition: a “template” pattern plus motion and deformation.

How can neural networks learn to forecast dynamical evolution, rather than only “fit correlations”?

A minimal mental model is the advection of coherent structures. A front, precipitation band, or convective cluster often behaves like a spatial pattern that moves and changes. This motivates neural architectures that can represent translation-like dynamics.

The forecast mapping. In general, deterministic forecasting can be written as

$$[x_{t-\Delta t}, \dots, x_t] \xrightarrow{F_\theta} x_{t+\Delta t},$$

or in multi-step rollout form:

$$x_{t+k\Delta t} = F_\theta^{(k)}(x_t), \quad \text{with} \quad F_\theta^{(k+1)}(x_t) = F_\theta(F_\theta^{(k)}(x_t)).$$

The important operational remark:

Long lead times are produced by repeated short-step forecasts (rollout).

From nowcasting to forecasting. One narrative in the slides is that the boundary between nowcasting and forecasting is increasingly bridged by AI systems. Neural networks can learn short-step evolution from data and extend it via rollout, sometimes supported by multi-scale architectures.

The NWP continuum and the neural perspective. The “NWP continuum” emphasizes the broad spectrum from physics-based modeling to data-driven prediction. In practice, the future will contain hybrids and coexistence rather than replacement.

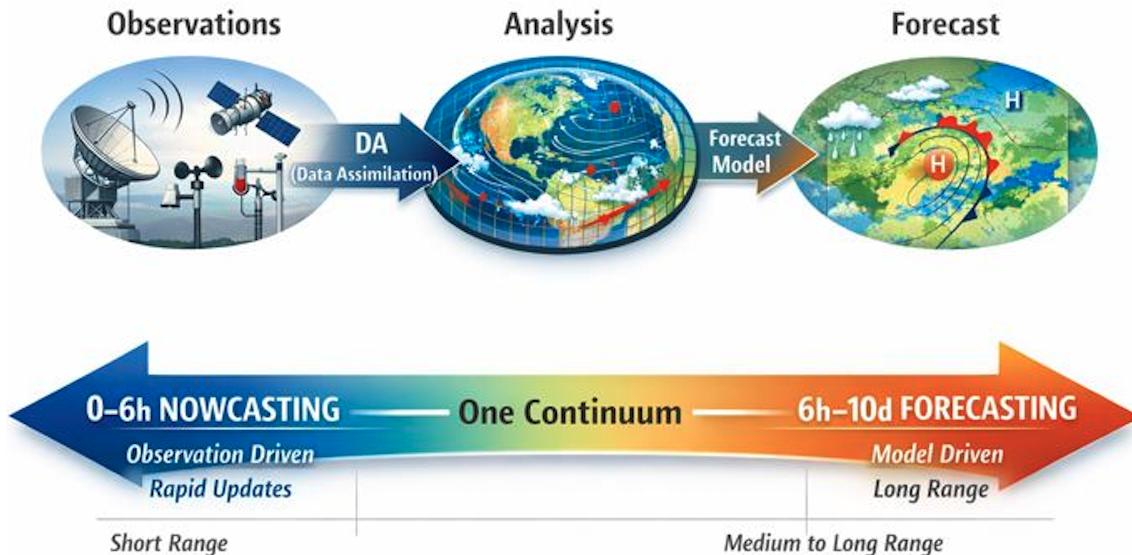


Figure 16.8: From nowcasting to forecasting: short-step prediction + rollout connects the time scales.

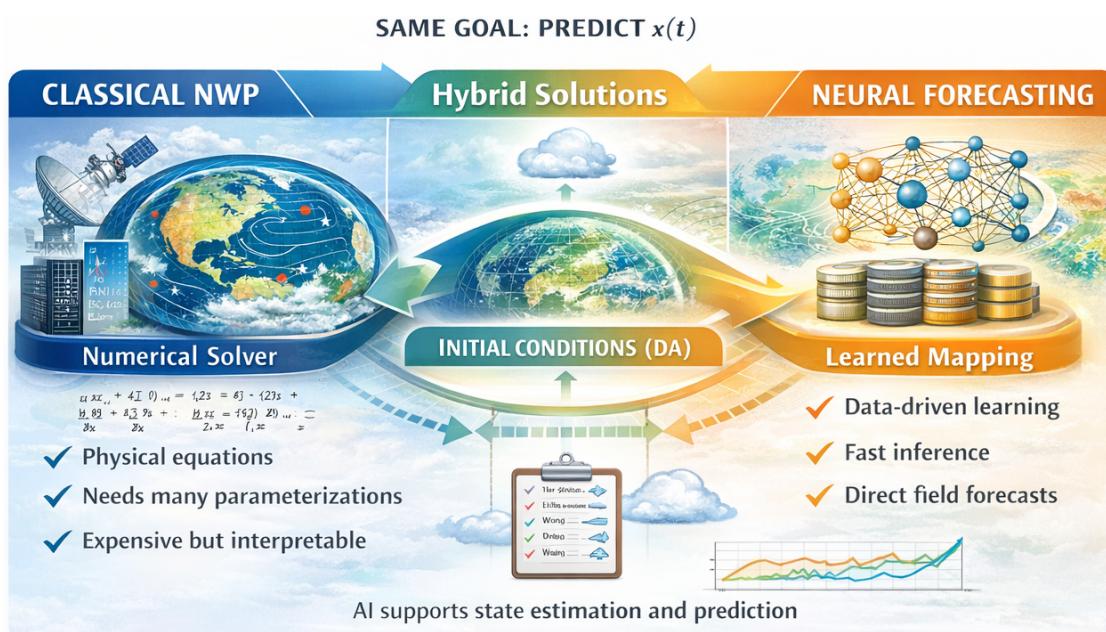


Figure 16.9: NWP continuum: physics-based and ML-based methods co-evolve and interact.

16.5 CNN Translation Toy World: features, loss, and why it works

To make the learning mechanism tangible, the slides use a minimal “toy world”: a coherent pattern moves on a periodic domain. The example is deliberately simple, but it captures a key mechanism: **local feature extraction + translation equivariance**.

A minimal dynamical system: signal on a circle. We consider a 1D periodic coordinate $x \in [0, 1)$ and a signal $s(x, t)$. At each step, the signal is shifted by some velocity:

$$s(x, t + \Delta t) \approx s(x - u\Delta t, t),$$

which is essentially advection on a periodic domain.

CNN view: translation as local feature transport. A CNN processes local neighborhoods with shared filters. This makes the representation naturally suited for moving structures: the same filter detects the same feature at any location.

Learning objective. Let $\hat{s}(x, t + \Delta t) = F_\theta(s(\cdot, t))$ be the prediction. A typical loss is mean squared error:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \|\hat{s}_i - s_i^{\text{true}}\|_2^2.$$

Training curves typically show rapid reduction of error as the model learns to align motion and structure.

Inside the CNN: hierarchical feature representations. The CNN does not store a “copy” of the signal; it learns feature representations of increasing abstraction. Early layers detect local gradients and bumps; deeper layers capture composite patterns.

Why this is not just “correlation”. The toy world illustrates the general mechanism: the network learns a representation in which the state evolution becomes predictable. Forecasting becomes learning an operator that is consistent across space and time.

In meteorology, the same logic applies at a larger scale: fronts, jets, clouds and precipitation structures have recognizable patterns, and their motion is constrained by dynamics. The neural network can exploit these regularities.

Minimal code example (toy translation model). The following simplified code block illustrates the key learning loop (dataset generation + CNN mapping + rollout). The actual notebook contains a more elaborate version with diagnostics and plots.

python

```

1 import numpy as np
2 import torch
3 import torch.nn as nn
4 import torch.nn.functional as F
5
6 # periodic 1D grid
7 nx = 128
8 x = np.linspace(0, 1, nx, endpoint=False)

```

Periodic 1D signal on a circle (3D)

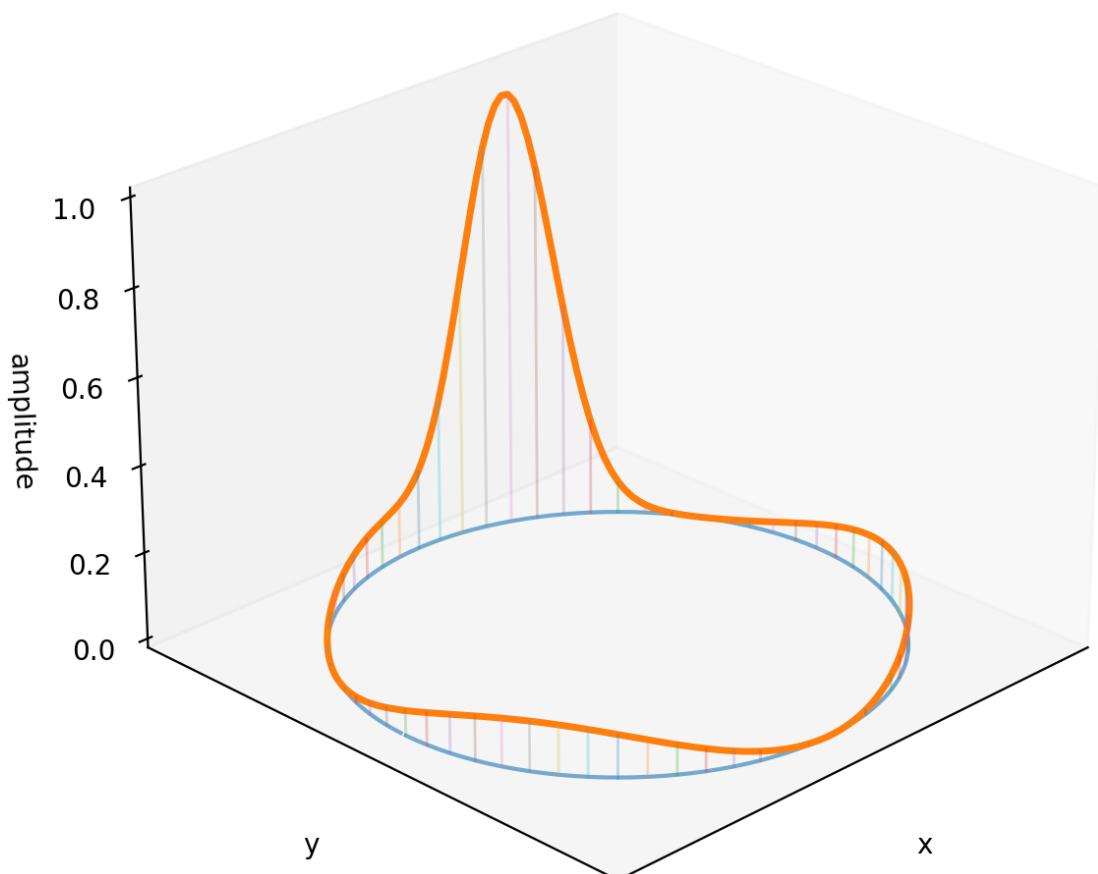


Figure 16.10: Toy world: a signal evolves on a periodic circle. Forecasting becomes learning motion and deformation.

```

9
10 def make_blob(center, width=0.06):
11     dx = (x - center + 0.5) % 1.0 - 0.5
12     return np.exp(-(dx**2)/(2*width**2))
13
14 def shift_periodic(signal, shift):
15     return np.roll(signal, shift)
16
17 # dataset: random centers + random shifts
18 def sample_pair(batch=64):
19     X, Y = [], []
20     for _ in range(batch):
21         c = np.random.rand()
22         s0 = make_blob(c)

```

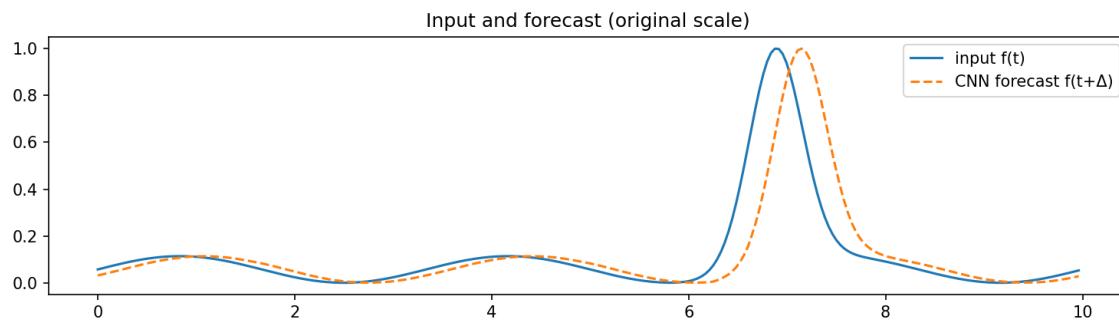


Figure 16.11: CNN translation setup: learn mapping from past signal(s) to future signal.

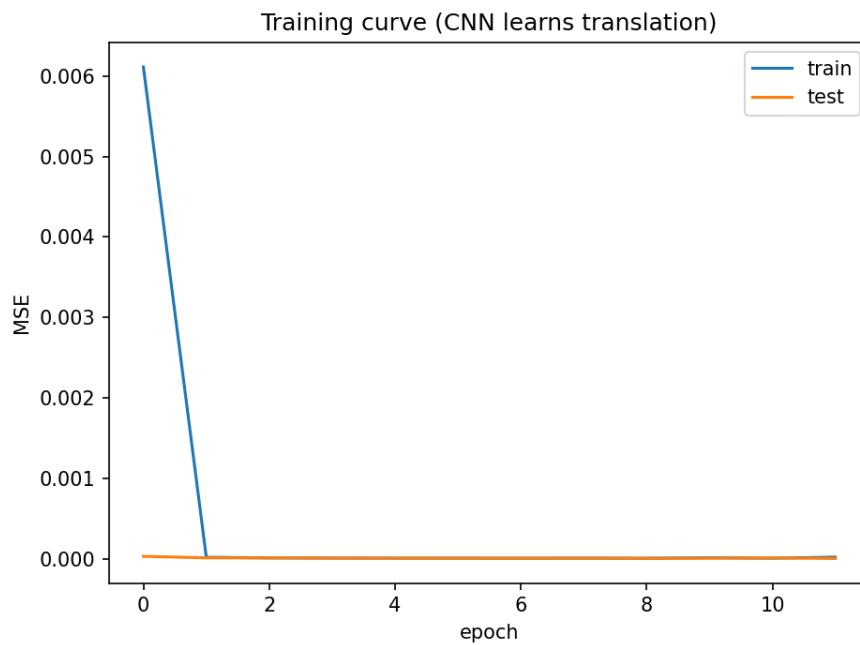


Figure 16.12: Training loss: the CNN learns to predict the translated/deformed signal.

```

23     sh = np.random.randint(1, 6)          # small motion
24     s1 = shift_periodic(s0, sh)
25     X.append(s0); Y.append(s1)
26     X = torch.tensor(np.stack(X), dtype=torch.float32).unsqueeze(1) # (B, 1, nx)
27     Y = torch.tensor(np.stack(Y), dtype=torch.float32).unsqueeze(1)
28     return X, Y
29
30 class SmallCNN(nn.Module):
31     def __init__(self):
32         super().__init__()
33         self.net = nn.Sequential(
34             nn.Conv1d(1, 16, 5, padding=2),
35             nn.GELU(),

```

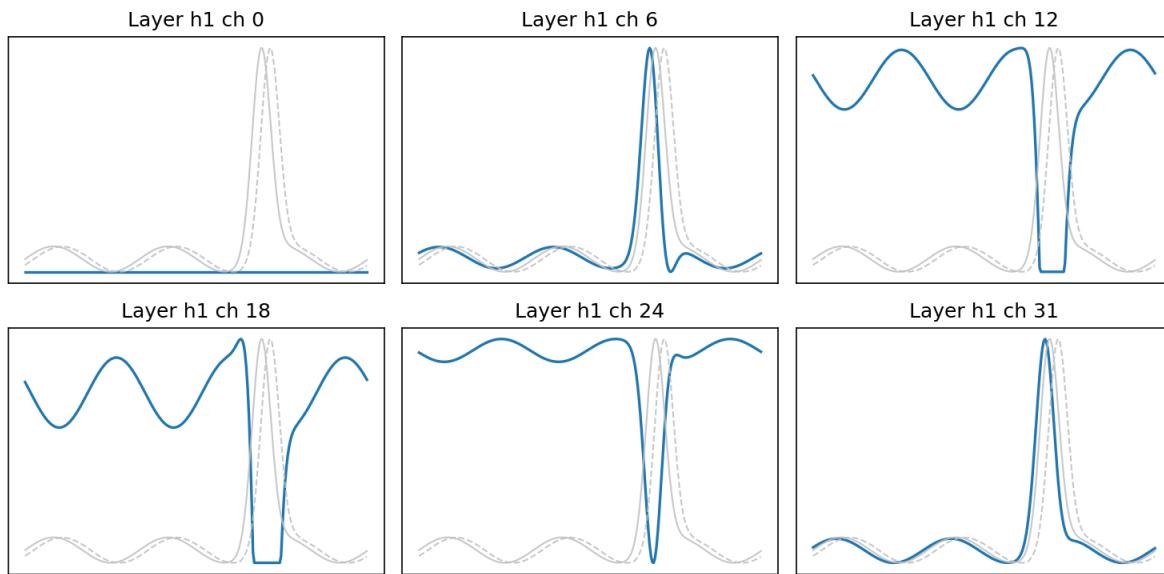


Figure 16.13: CNN feature maps (example): early hidden representation.

```

36             nn.Conv1d(16, 16, 5, padding=2),
37             nn.GELU(),
38             nn.Conv1d(16, 1, 1)
39         )
40     def forward(self, x):
41         return self.net(x)
42
43 model = SmallCNN()
44 opt = torch.optim.Adam(model.parameters(), lr=1e-3)
45
46 for step in range(2000):
47     Xin, Ytrue = sample_pair(batch=64)
48     Yhat = model(Xin)
49     loss = F.mse_loss(Yhat, Ytrue)
50     opt.zero_grad(); loss.backward(); opt.step()

```

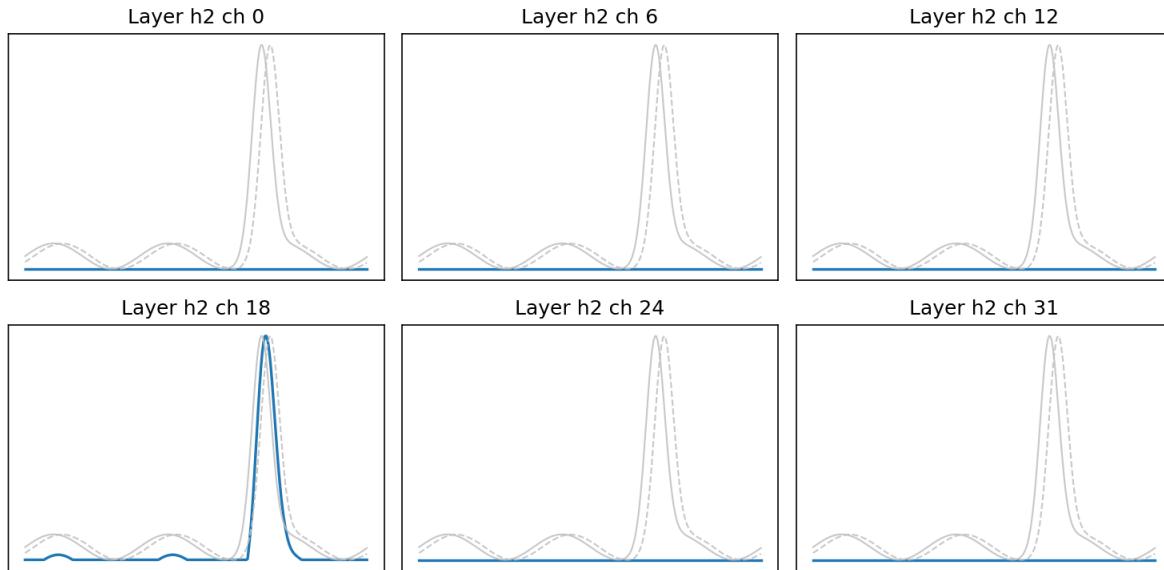


Figure 16.14: CNN feature maps (example): intermediate hidden representation.

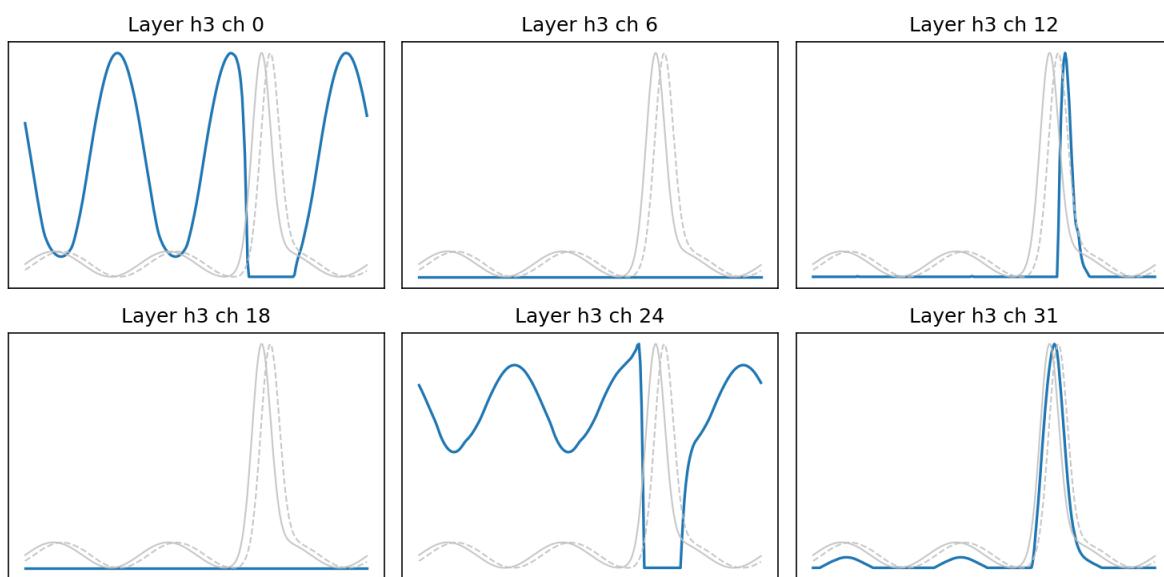


Figure 16.15: CNN feature maps (example): deeper hidden representation.

16.6 Weather Service Perspective: products, services, and operational ecosystems

The AI transformation is not only about building better forecast models. For a National Meteorological Service, a very large part of value creation happens **after** the raw forecast is available: products, services, interpretation, and operational integration. In practice, weather services can easily spend **50% or more** of their effort on **derived products and services** — and this share can be even higher, because the societal value is generated through delivery, interpretation and impact.

Raw forecast fields are only the beginning. Operational services require much more than state variables:

- derived variables and indicators (e.g. road weather indices, icing risk, wind power),
- phenomena and event extraction (high-impact weather diagnostics),
- warning logic and impact-based products,
- verification, monitoring, and quality control (QC),
- explainability, interpretation, and communication,
- domain-specific user-facing products (aviation, hydrology, health, civil protection).

This is exactly why an AI strategy for weather services must cover the *full value chain*, not only the forecasting model.

Operational AI forecasting is already real. The slides emphasize an important reality check: AI forecasting in Europe is no longer “research only”. Several systems are already in production use (e.g. AIFS, AICON, BRIS), and they complement classical NWP by providing:

- fast state-to-state forecasts (minutes instead of hours),
- competitive large-scale skill for key variables,
- robust baselines and rapid experimentation.

Ecosystem viewpoint: a portfolio of complementary libraries. The European ML-for-NWP ecosystem develops along several complementary lines. A key slide message is that this is **good news**: no single framework needs to do everything; modular building blocks enable rapid innovation, and shared standards support interoperability and reuse. In particular, the ecosystem includes complementary strengths such as:

- **Anemoi**: end-to-end AI weather models (training + inference stack),
- **mfa**: transformers / vision methods and many applications,
- **MLCast**: observation-driven nowcasting components,
- **FRAIM**: products and services across the full AI/ML value chain.

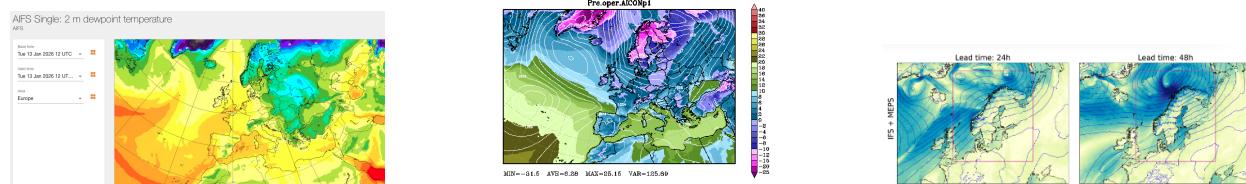


Figure 16.16: Operational AI forecasting is already in production use in Europe (slide narrative: “not research only”). This strengthens the case for systematic integration into the weather service toolbox.

What we can do already: full value chain competence. We already have strong building blocks in place, spanning both core forecasting and applied product development:

- an operational framework for AI-based forecasting (Anemoi),
- product and service libraries (mfaI, FRAIM) for:
 - downscaling and high-resolution products,
 - road weather services,
 - high-impact weather feature extraction,
 - interpretation and explainability,
 - nowcasting of observation fields (radiation, precipitation, . . .).

This is not just technical capability: it is the **organizational foundation** for sustainable AI operations, collaboration, and benchmarking.

Physics and AI: why physics remains essential. A central warning in the slides is that weather is not just a “pattern problem”. Forecasting is consistent evolution of a physical state: conservation laws, balances, constraints, multi-scale coupling. Pure AI models can show strong short-term skill but still drift long-term when small violations accumulate (moisture, energy, mass), and rare extremes require physical consistency. The long-term direction is therefore clear:

Future AI forecast systems will be “Physics + AI”.

Key message for weather services. The AI transformation becomes truly valuable when forecasting engines are connected to product/service ecosystems and standards-based operational integration. In other words:

Forecast engines + product/service ecosystems + operational platforms.

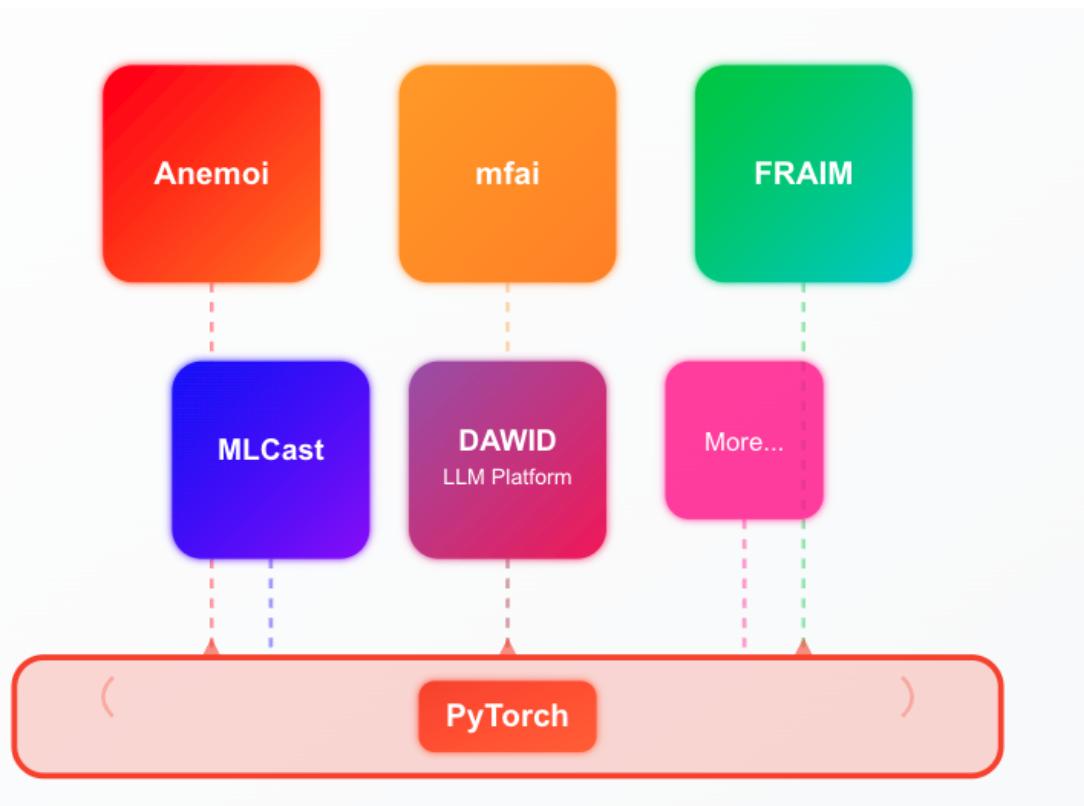


Figure 16.17: Library ecosystem: complementary strengths, one shared mission. The core point is modularity and interoperability rather than a single monolith.

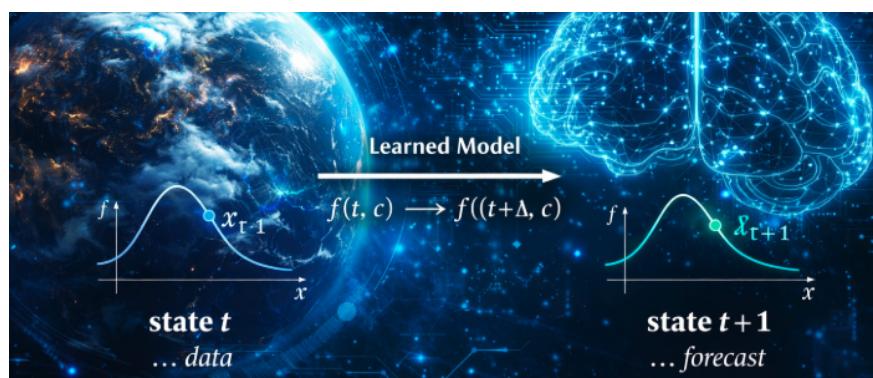


Figure 16.18: Physics and AI: AI gives speed and learning; physics gives truth and trust. The operational goal is reliable evolution under constraints, not only local pattern accuracy.

16.7 Outlook: transformation of workflows and service building

The final part of the lecture is pragmatic and action-oriented. AI will not only provide new forecast engines; it will reshape workflows, software architectures, and how we build services. The slide narrative highlights three connected points: **tools**, **standards**, and **workflow redesign**.

Tool calling becomes a strategic capability. Modern AI assistants become powerful when they can call trusted tools. This is currently a fast-moving technology race:

- OpenAI: tool calling with JSON schema and structured outputs,
- Anthropic: MCP (Model Context Protocol) as connector architecture,
- Google: Gemini function calling within its ecosystem,
- Linux Foundation / AAIF: interoperability to avoid vendor lock-in.

The key operational message for weather services is:

LLMs need good tool definitions — and we can use this to build better services.

DAWID opportunity: our trusted functions at our fingertips. A central slide message is that a platform such as DAWID can expose existing operational capabilities as tools with clean interfaces:

- tools are trusted DWD / ECMWF / NMS functions (not generic chat),
- each tool encodes a workflow step we already know,
- the LLM becomes orchestrator and user interface.

This yields immediate impact: faster exploration, reproducible results, and knowledge transfer (tools carry expertise).

Workflow redesign: if a task repeats, it should become a tool. One of the strongest action messages is: rebuild services by analysing daily work, finding waste, and turning repeated tasks into tools with clear I/O interfaces. This can reduce copy/paste workflows, waiting time, and loss of context across files, emails and chats. In short:

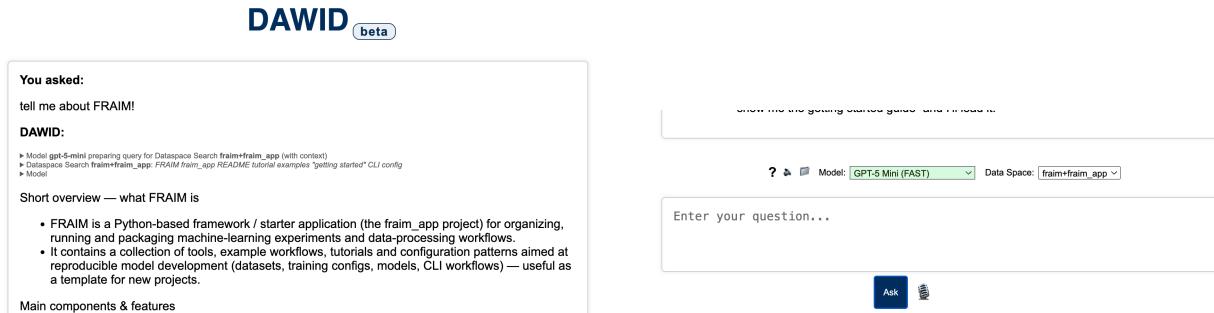
If a task repeats, it should become a tool.

Bring your domain expertise: make AI your own. LLMs and neural networks do not automatically understand meteorology: processes, states, balances, physical limits, plausibility, regimes, and extremes. Operational quality requires embedding domain knowledge into:

- variables, events, diagnostics, and what matters,
- physical constraints and what is allowed,
- product requirements and what is useful.



Figure 16.19: Tool calling and agent interoperability: multiple technology streams push towards standardized access to tools and enterprise systems. For weather services this translates into a concrete opportunity: expose operational expertise as callable tools.



DAWID (beta)

You asked:
tell me about FRAIM!

DAWID:

- Model gpt-3.5-turbo preparing query for Database Search fram+fram_app (with context)
- Database Search fram+fram_app: FRAIM fram_app README tutorial examples "getting started" CLI config
- Model

Short overview — what FRAIM is

- FRAIM is a Python-based framework / starter application (the fram_app project) for organizing, running and packaging machine-learning experiments and data-processing workflows.
- It contains a collection of tools, example workflows, tutorials and configuration patterns aimed at reproducible model development (datasets, training configs, models, CLI workflows) — useful as a template for new projects.

Main components & features

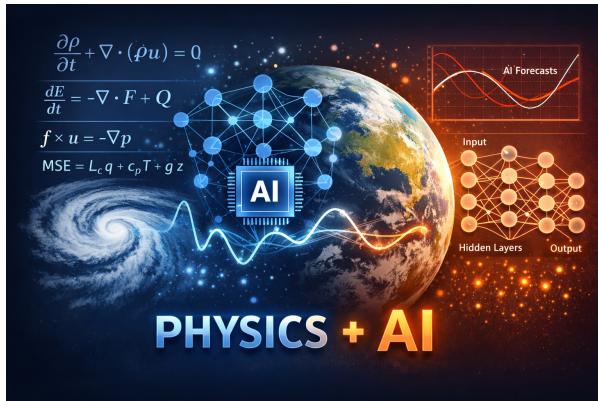
Enter your question...

Figure 16.20: DAWID and FRAIM in the slide narrative: make operational capabilities accessible via tool interfaces — “our functions” become reusable building blocks.

This is the human core contribution in the AI era: not outsourcing expertise, but encoding it into tools, workflows and constraints.

Outlook. The transformation is therefore not only a model question, but a system question: AI-enabled weather services will be characterized by

- language-driven access to complex toolchains,
- automation of analysis, summarization and reporting,
- rapid prototyping of products and service variants,
- continuous evaluation and improvement cycles.



Domain expertise. AI becomes powerful when grounded in meteorology.



System transformation. Models, tools, workflows and products evolve together.

Figure 16.21: Key messages of the AI transformation for weather services: domain expertise is essential for trust, and operational impact requires a full-system approach.

Chapter 17

Model Emulators, AIFS and AICON

17.1 Model emulators based on Anemoi

17.1.1 Motivation: emulators as operational building blocks

Modern global and regional NWP models represent the atmosphere through a high-dimensional state vector $\mathbf{x}(t)$ on a given discretization (grid/mesh, vertical levels, variables). Time integration of a full physical model is expensive and limits resolution, ensemble size, and product diversity.

A *model emulator* aims to replace (parts of) the forecast operator by a learned mapping

$$\mathcal{M}_{\Delta t} : \mathbf{x}(t) \mapsto \mathbf{x}(t + \Delta t),$$

while preserving the operational semantics: variables, grids, metadata, GRIB workflows, and verification infrastructure.

Forecasting with an emulator typically uses short-step rollout:

$$\mathbf{x}_{k+1} = \mathcal{M}_{\Delta t}(\mathbf{x}_k), \quad k = 0, \dots, N - 1,$$

so that a forecast of length $T = N\Delta t$ is obtained as repeated application of the learned operator.

In practice, emulators rarely operate on a single “field”. They take *multi-variable, multi-level* states as input and output and must handle non-trivial mesh/topology.

17.1.2 The Anemoi framework

In Q2/2024, DWD adopted the shared *Anemoi* codebase for research and development of data-driven NWP models, replacing its own implementation of a Graph based forecasting system for strategic reasons. Anemoi is a collaborative European initiative aligned with EUMETNET’s E-AI program. It provides an end-to-end workflow:

- dataset and I/O tooling (reanalysis / model data, Zarr, GRIB)
- graph construction for GNN models

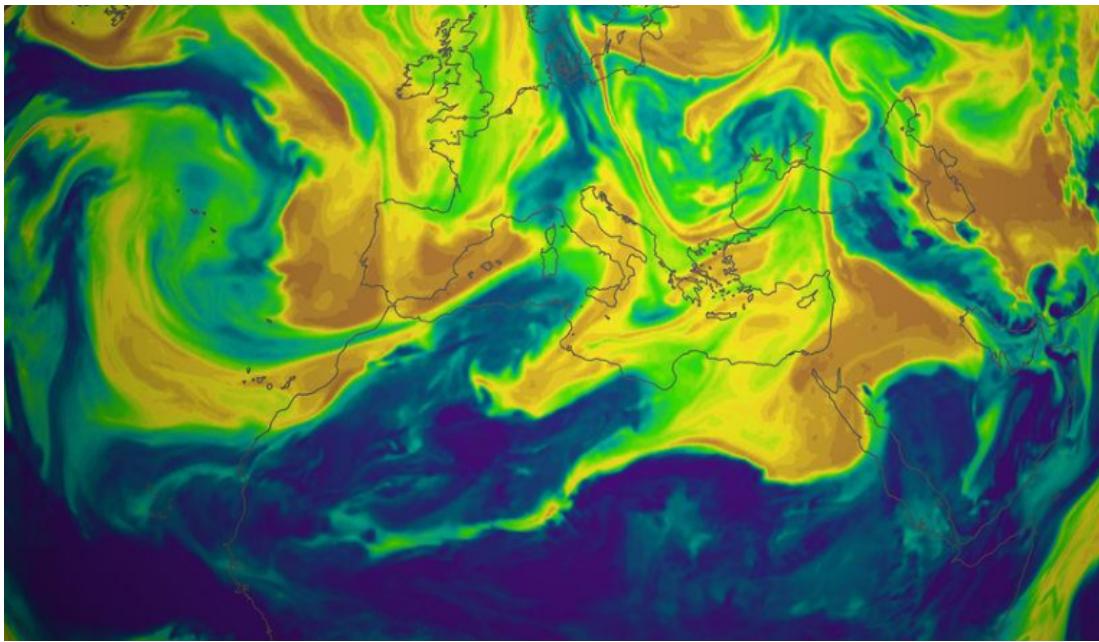


Figure 17.1: Operational context: ML-based emulators accelerate the forecast step and enable new applications at scale while keeping operational data interfaces.

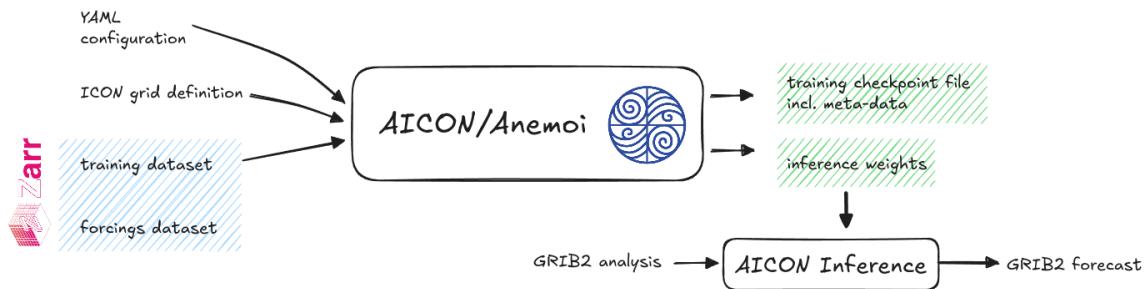


Figure 17.2: Conceptual overview: Anemoi as a shared framework for European ML-based weather models.

- training drivers (PyTorch Lightning + Hydra)
- inference tooling and deployment interfaces

The Anemoi codebase is structured into several repositories:

- **anemoi-core**: graphs, models, training driver code
- **anemoi-datasets**: high-level dataset recipe handling (YAML-based)
- **anemoi-inference**: inference package for ML-driven prediction
- supporting repos (e.g. transforms, utilities)

17.1.3 AIFS and AICON within Anemoi: common principles

Large-scale European ML forecasting systems (AIFS at ECMWF, AICON at DWD) share key design choices:

- They treat the Earth as a graph and use a GNN as forecast operator.
- They learn increments / tendencies efficiently using residual connections.
- They rely on standardized data containers, tracking, checkpoints, and reproducibility.

A general deterministic formulation is

$$[\mathbf{X}_{t-(\alpha-1)}, \dots, \mathbf{X}_t] \xrightarrow{F_\theta} [\mathbf{Y}_{t+1}, \dots, \mathbf{Y}_{t+\beta}],$$

with input window length α and output window length β . For AICON in the referenced setup:

$$\alpha = 2, \quad \beta = 1,$$

i.e. two input states (e.g. $t - 3$ h and t) and one predicted state (e.g. $t + 3$ h). Longer lead times are produced by rollout.

17.2 AIFS

AIFS is ECMWF's Artificial Intelligence Forecasting System, developed in the shared European *Anemoi* framework. In this chapter, AIFS mainly serves as a reference point for two aspects:

- **Framework perspective:** AIFS and AICON share the Anemoi toolchain for datasets, graph-based models, training driver code and checkpoint handling.
- **Vertical discretization comparison:** the AICON walkthrough illustrates how ICON level heights (SLEVE coordinate) can be related to AIFS pressure levels for visualization and interpretability.

17.3 AICON

17.3.1 High-level positioning

AICON is DWD's ML-based forecast model complementing ICON. It is based on Graph Neural Networks and operates natively on ICON's triangular mesh.

17.3.2 Reproducible training: driver code, configuration, logging

AICON training is driven by an Anemoi trainer configured by YAML (Hydra). This ensures that experiments are reproducible and traceable.

A typical workflow is:

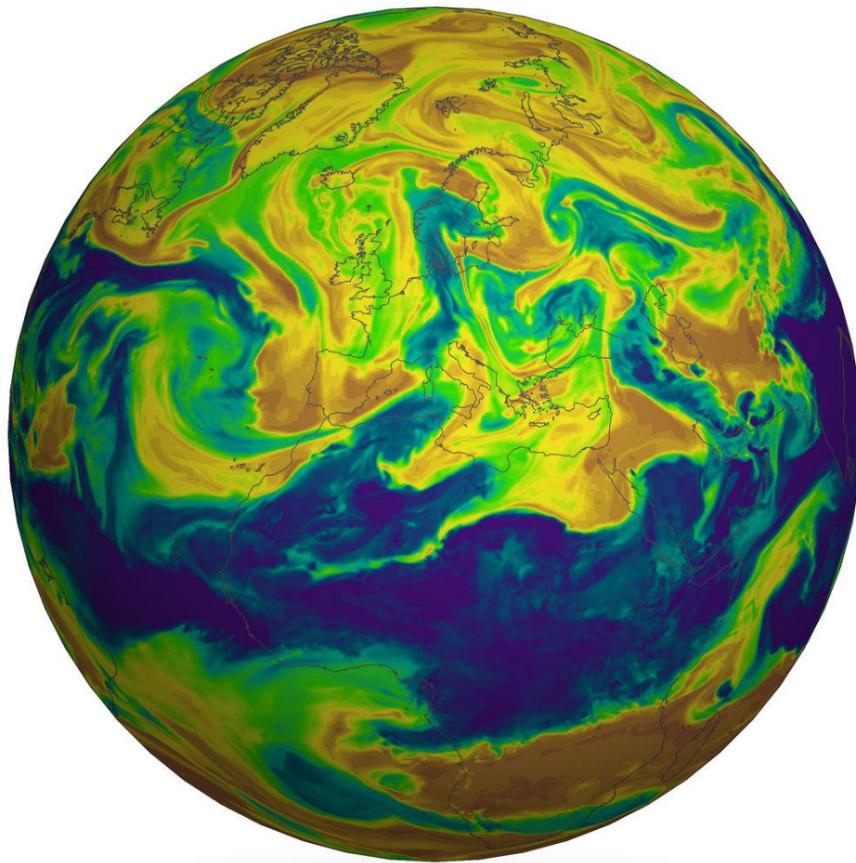


Figure 17.3: AICON as a DWD model emulator: GNN-based forecasting on ICON meshes.

1. Load YAML config with Hydra
2. Construct AnemoiTrainer(config)
3. Set strategy (notebook-safe variant)
4. Train for a configured number of steps / epochs
5. Store checkpoints (last.ckpt, inference-last.ckpt)

```
python
```

```
1 # Load YAML configuration with Hydra and create trainer
2 config_filename = "test_aicon_01.yaml"
3 import os, hydra
4 import anemoi.training
5
6 os.environ["ANEMOI_CONFIG_PATH"] = os.path.join(
7     os.path.dirname(anemoi.training.__file__), "config"
8 )
9
```

```

class GraphicalLiveLogger(Logger):
    def __init__(self, interval=10):
        super().__init__()
        self.metric_name = "train_weighted_mse_loss_step"
        self.metrics_history = []

    @property
    def name(self): return "LiveLogger"
    @property
    def version(self): return "0.1"

    def log_hyperparams(self, *args, **kwargs): pass

    def log_metrics(self, metrics, step):
        self.metrics_history.append((step, metrics))

    def finalize(self, status):
        steps, values = zip(*[(s,m[self.metric_name]) for s,m in self.metrics_history if self.metric_name in m])
        plt.figure(figsize=(8,4))
        plt.plot(steps, values, 'o', ms=2, label="AICON", color="red")
        plt.title(f"{self.metric_name} over steps"); plt.xlabel("Step"); plt.ylabel(self.metric_name)
        plt.grid(True); plt.legend(); plt.show()

trainer.model.logger_enabled = True
trainer.loggers = GraphicalLiveLogger()

```

Figure 17.4: Example: live logging and monitoring of training progress in the AICON walkthrough.

```

10 with hydra.initialize(version_base=None, config_path="."):
11     config = hydra.compose(config_name=config_filename)
12
13 from anemoi.training.train import AnemoiTrainer
14 trainer = AnemoiTrainer(config)
15
16 # Make the strategy notebook-compatible (single-rank view)
17 class JupyterNotebookStrategy(str):
18     def __init__(self, value):
19         self.global_rank = 0
20 trainer.strategy = JupyterNotebookStrategy("auto")
21
22 # Set base seed for reproducibility
23 os.environ["ANEMOI_BASE_SEED"] = "42"
24
25 # Run a short training
26 trainer.train()

```

17.3.3 Datasets: Zarr as scalable container

AICON training consumes reanalysis-style gridded datasets from ICON DREAM converted into **Zarr**. Zarr stores chunked, compressed multi-dimensional arrays enabling parallel I/O.

The dataset contains variables across multiple vertical levels; in the example setup, meteorological core variables include:

$$P, QV, T, U, V, W$$

at multiple ICON levels.

Additionally, *forcings* provide external information that is not predicted, including land/sea masks,



Figure 17.5: Training loss curve from a short AICON test run (live logger example).

topography, roughness, and time-dependent cyclic features.

17.3.4 Vertical levels and multi-model comparison

Vertical discretizations differ across systems. AICON uses ICON levels (SLEVE coordinate), while AIFS (in the referenced comparison) uses selected pressure levels. A useful diagnostic is to map ICON level heights and pressure levels into one plot.

17.3.5 Hidden mesh / multi-mesh definition

AICON's central architectural choice is to build the GNN graph directly from ICON's hierarchical triangular meshes. This includes:

- **Data mesh:** ICON cell circumcenters (where prognostic variables live)
- **Hidden mesh:** ICON vertices across multiple refinement levels

The grid file provides refinement level attributes (`refinement_level_v`, `refinement_level_c`) from ICON's hierarchical construction. Selecting vertices up to a maximum refinement level k defines the hidden mesh.

For an ICON grid root index n and refinement k , the number of vertices is

$$N_v = 10 n^2 4^k + 2,$$

which matches the hidden mesh size observed in the walkthrough.

► Dimensions: (variable: 84, time: 724, ensemble: 1, cell: 11520)

► Coordinates: (0)

▼ Data variables:

count	(variable)	float64	dask.array<chunksize=(84,), me...  
data	(time, variable, ensemble, cell)	float32	dask.array<chunksize=(1, 84, 1,...  
dates	(time)	datetime64[s]	dask.array<chunksize=(724,), m...  
has_nans	(variable)	object	dask.array<chunksize=(84,), me...  
latitudes	(cell)	float64	dask.array<chunksize=(11520,),...  
longitudes	(cell)	float64	dask.array<chunksize=(11520,),...  
maximum	(variable)	float64	dask.array<chunksize=(84,), me...  
mean	(variable)	float64	dask.array<chunksize=(84,), me...  
minimum	(variable)	float64	dask.array<chunksize=(84,), me...  
squares	(variable)	float64	dask.array<chunksize=(84,), me...  
stdev	(variable)	float64	dask.array<chunksize=(84,), me...  
sums	(variable)	float64	dask.array<chunksize=(84,), me...  

Figure 17.6: Dataset inspection: AICON input in Zarr format opened via Xarray.

Edges result from the union of coarse-to-fine edge hierarchies: levels $R_nB0 \dots R_nBk$. For icosahedral meshes, the number of undirected edges per level follows

$$N_e(k) = 30 n^2 4^k.$$

For a directed representation (PyG), the total edge count is doubled.

AICON thereby uses a multi-scale message passing topology similar in spirit to GraphCast's multi-mesh.

17.3.6 Encoder–Processor–Decoder architecture on ICON graphs

AICON uses an **encoder–processor–decoder** architecture:

- **Encoder:** maps data-mesh features to hidden-mesh embeddings
- **Processor:** message passing on the hidden multi-mesh
- **Decoder:** maps back to data mesh and output variables

A crucial detail: the encoder/decoder operate on a *bipartite graph*. In the AICON setup every ICON cell is connected to the 3 vertices of its triangle. Thus the encoder degree on the source side is fixed:

$$\deg_{\text{cell}} = 3.$$

Conversely, vertices receive information from several cells and have varying degree.

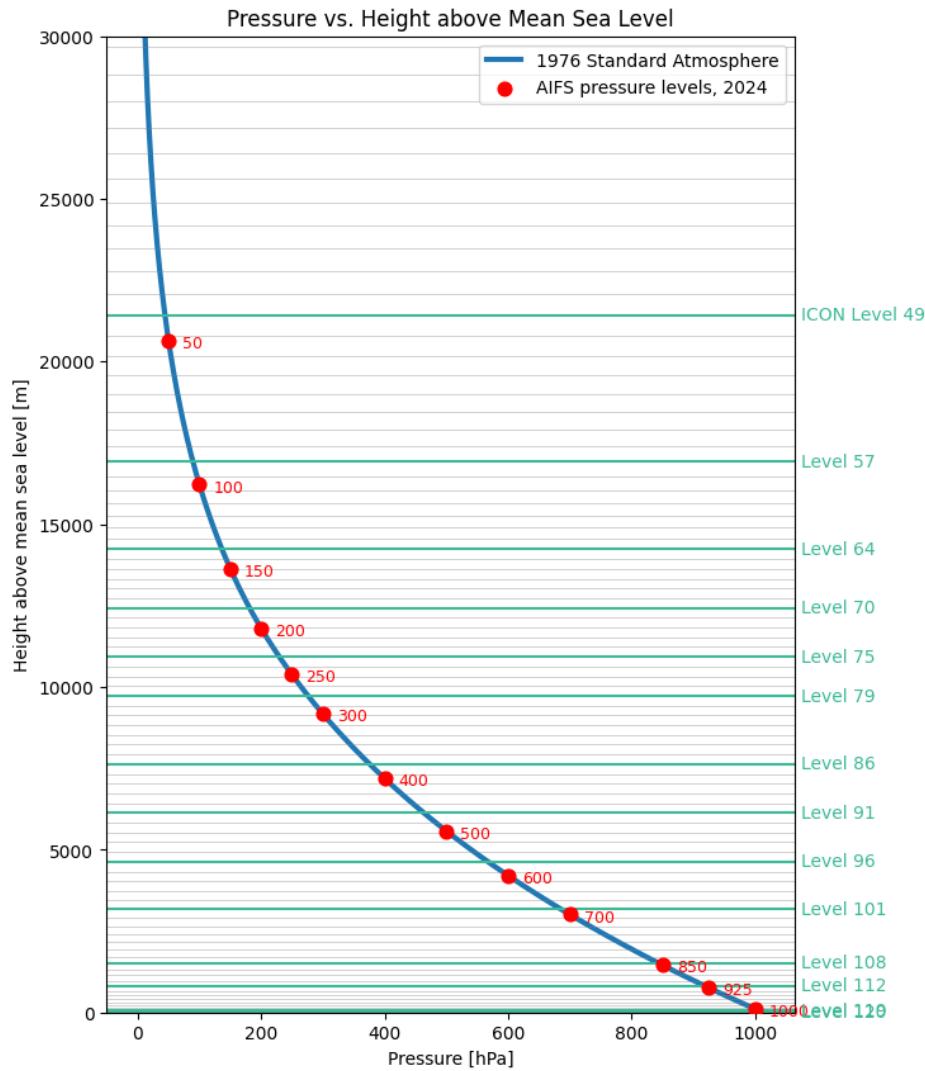


Figure 17.7: Example visualization of ICON vertical level heights versus selected pressure levels.

17.3.7 Processor: Message Passing vs GraphTransformer

The Anemoi framework supports two main processor families:

- classical **message passing** graph convolution (MPNN / GraphConv)
- **GraphTransformer** with attention weights (GraphTransformerConv)

Message passing can be expressed as

$$\mathbf{x}'_i = \gamma_{\Theta} \left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} \phi_{\Theta}(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i}) \right),$$

where \oplus is an aggregation operation (sum/mean/max).

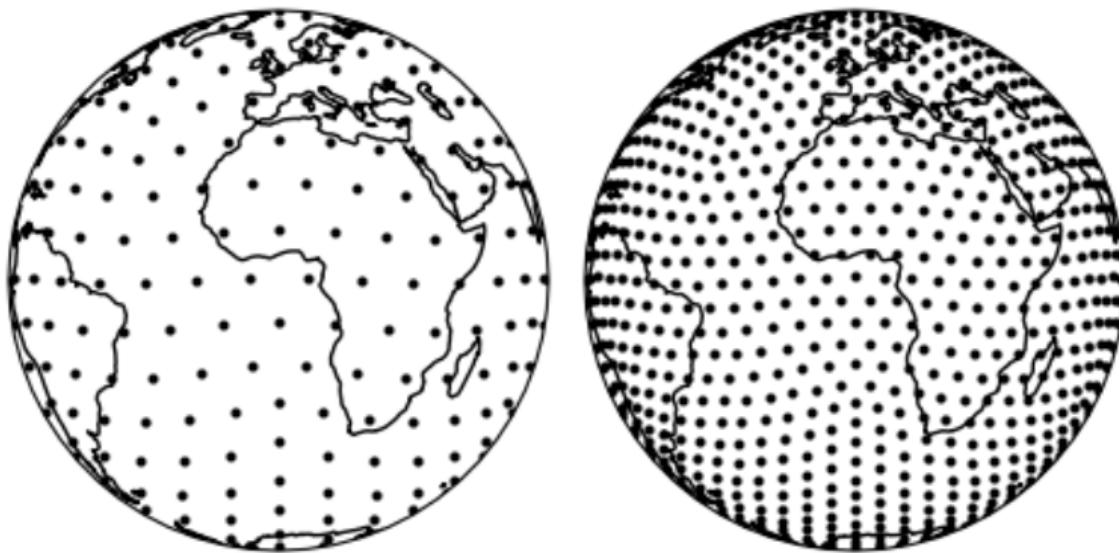


Figure 17.8: Refinement-level based vertex selection: subgraphs of varying density.

GraphTransformer message passing replaces fixed adjacency normalization by attention coefficients:

$$\mathbf{x}'_i = W_1 \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} W_2 \mathbf{x}_j,$$

with

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{K}_j^\top \mathbf{Q}_i}{\sqrt{d}}\right),$$

where d is the channel dimension.

17.3.8 Attention visualization in AICON

AICON's GraphTransformer allows interpreting *which neighbors influence a node* via attention weights. In the walkthrough, attention is captured by wrapping the message function (monkeypatching) and extracting the attention coefficient tensor.

python

```

1 # Monkeypatching the message() function to capture attention alpha
2 import types, functools
3 from torch_geometric.utils import softmax
4
5 model = trainer.model.model.processor.proc[0].blocks[0].conv
6 original_message = type(model).message
7
8 @functools.wraps(original_message)
9 def message_with_alpha(self, heads, query_i, key_j, value_j, edge_attr, index, ptr
10 , size_i):
    global alpha

```

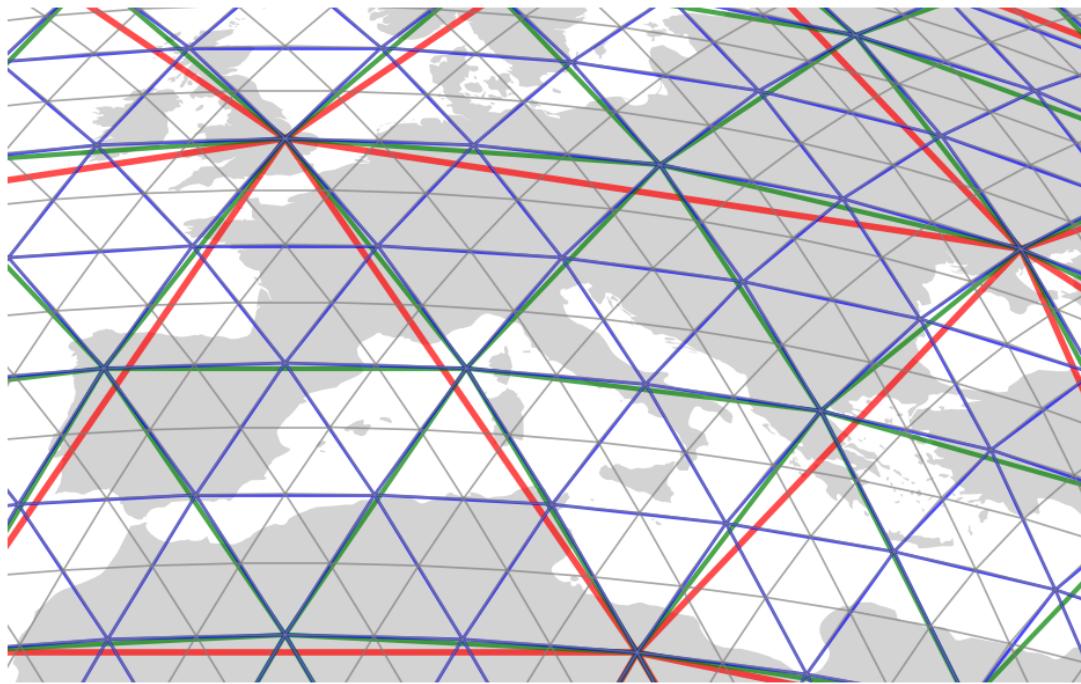


Figure 17.9: Hidden multi-mesh visualization: coarse-to-fine connectivity yields both short-range and long-range edges.

```

11     if edge_attr is not None:
12         key_j = key_j + edge_attr
13     alpha = (query_i * key_j).sum(dim=-1) / self.out_channels**0.5
14     alpha = softmax(alpha, index, ptr, size_i)
15     return original_message(self, heads, query_i, key_j, value_j, edge_attr, index
16                           , ptr, size_i)
17
18 model.message = types.MethodType(message_with_alpha, model)
19
20 # Run one inference step to populate alpha
21 alpha = None
22 with torch.no_grad():
23     out = trainer.model.predict_step(input_tensor_torch)

```

17.3.9 Transfer learning across meshes

AICON enables transfer learning between graphs of different hidden mesh resolution. If the maximum hidden refinement level changes (e.g. from $k = 3$ to $k = 4$), some graph-dependent parameters become incompatible (node coordinates, trainable features), but core neural weights can be reused.

This yields a practical training strategy:

- pretrain on a coarser mesh (cheap, fast)

```

encoder:
  _target_: anemoi.models.layers.mapper.GraphTransformerForwardMapper
  _convert_: all
  trainable_size: ${model.trainable_parameters.data2hidden}
  sub_graph_edge_attributes: ${model.attributes.edges}
  num_chunks: 2
  cpu_offload: ${model.cpu_offload}
  mlp_hidden_ratio: 4
  num_heads: 16
  qk_norm: false
  layer_kernels:
    Linear:
      _target_: torch.nn.Linear
      _partial_: true
    Activation:
      _target_: torch.nn.GELU
decoder:
  _target_: anemoi.models.layers.mapper.GraphTransformerBackwardMapper
  _convert_: all
  trainable_size: ${model.trainable_parameters.hidden2data}
  sub_graph_edge_attributes: ${model.attributes.edges}
  num_chunks: 1
  cpu_offload: ${model.cpu_offload}
  mlp_hidden_ratio: 4
  num_heads: 16
  qk_norm: false
  initialise_data_extractor_zero: false
  layer_kernels:
    Linear:
      _target_: torch.nn.Linear
      _partial_: true
  
```

Figure 17.10: Bipartite encoder and decoder graphs: data mesh \leftrightarrow hidden mesh.

- transfer weights to finer mesh
- continue training (higher fidelity)

17.4 Running AICON inference at DWD

Operational use requires strict control of I/O, preprocessing and product generation. DWD therefore provides a dedicated inference tool (aicon-inference), which differs from generic inference scripts by:

- robust GRIB2 handling and metadata management
- preprocessing and derived variables (e.g. soil moisture index SMI)
- output writing and optional interpolation onto regular lat-lon grids
- containerized environment for dependency control

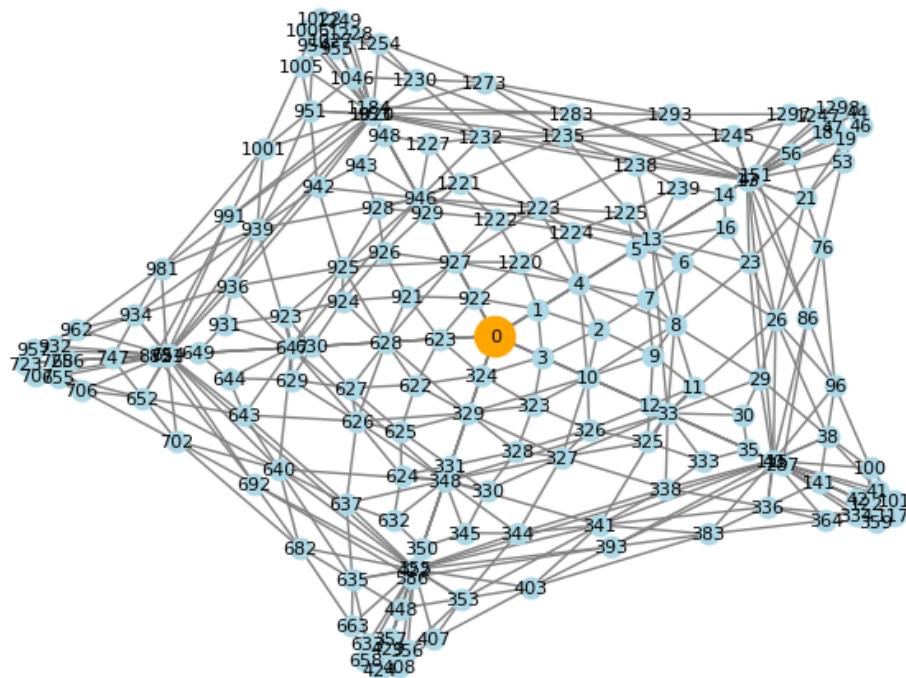


Figure 17.11: Local graph neighborhood illustration: node updates depend on 1-hop/2-hop connectivity.

bash

```
1 # Example: run inference using the dedicated command-line tool
2 aicon-inference --checkpoint output_training/checkpoint/<run_id>/inference-last.
    ckpt \
3             --cfg inference_test01.yaml
```

Internally, inference performs a rollout loop:

python

```
1 # pseudo-code of the core loop
2 torch.set_grad_enabled(False)
3 with torch.no_grad():
4     for step in range(num_steps):
5         out = trainer.model.predict_step(input_tensor_torch)
6         input_tensor_torch = update_input_with_prediction(input_tensor_torch, out)
```

For operational readiness, the inference tool and its dependencies are packaged into an Apptainer/Singularity container and deployed on the DWD GPU infrastructure.

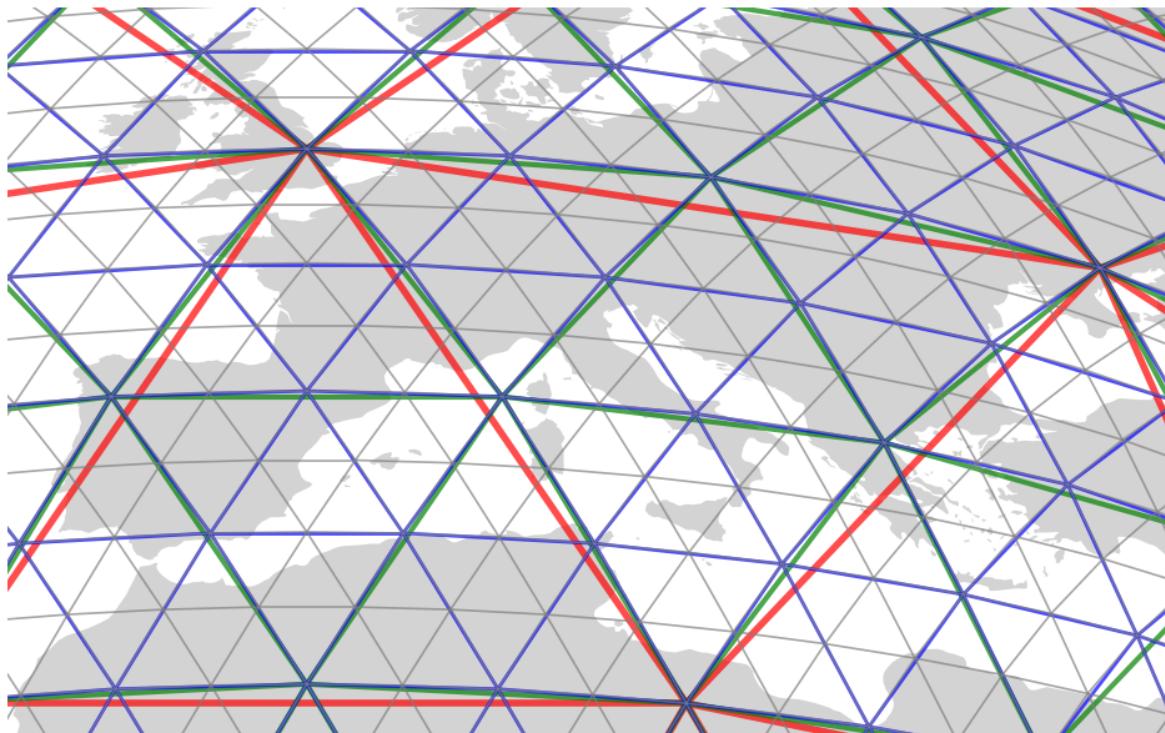


Figure 17.12: Attention visualization (example): attention weights on incoming edges for a selected receiver node.

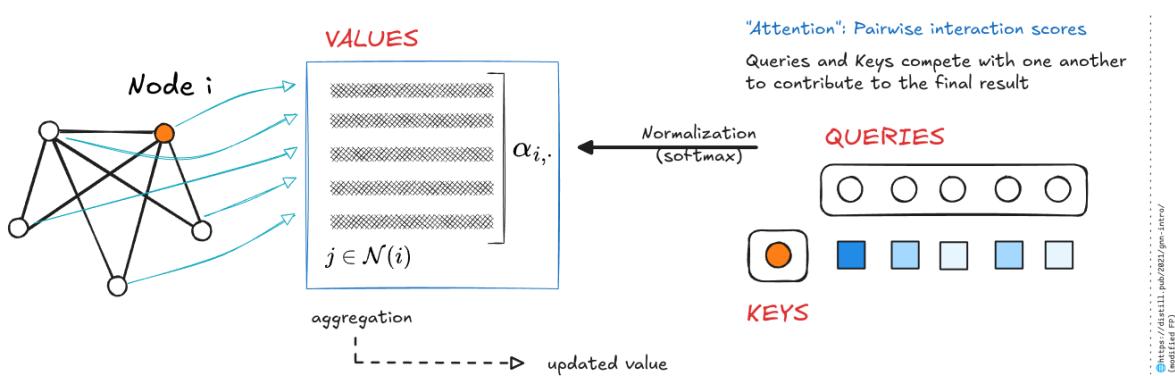


Figure 17.13: Attention visualization variant: highlighting relative weights for a given attention head.

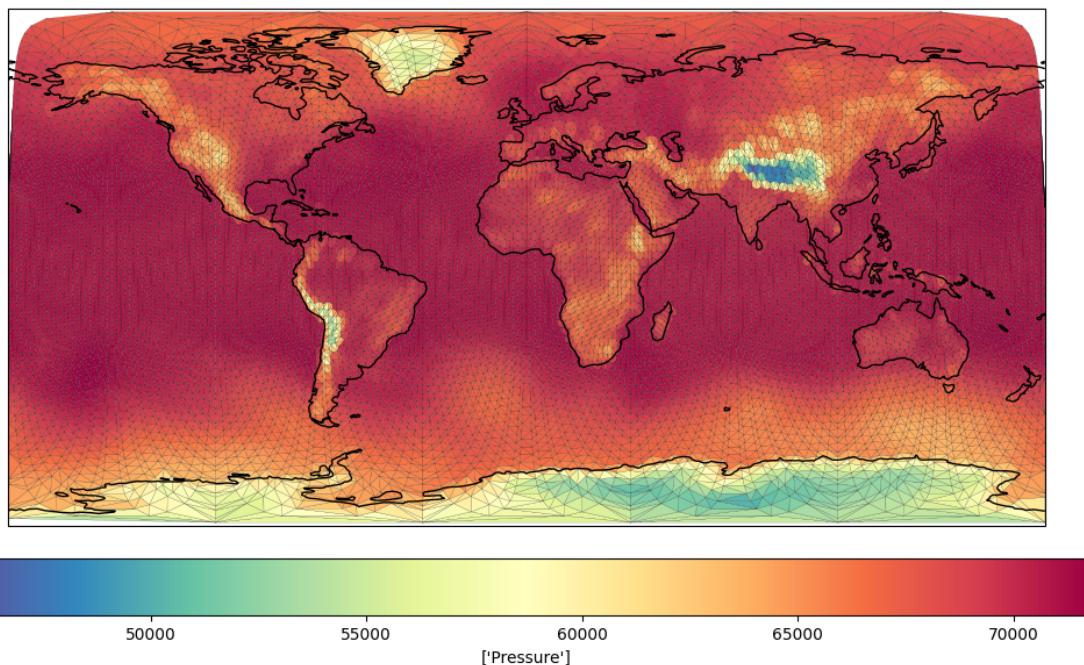


Figure 17.14: Example inference result visualization from the AICON walkthrough.

17.5 Further activity: Anemoi vs FRAIM

The European ML-for-Weather-and-Climate ecosystem currently develops along several complementary lines, including shared model frameworks and operational integration initiatives such as **Anemoi**, **mfaI**, **FRAIM**, and **MLCast**. These initiatives differ in scope and emphasis, ranging from the training and inference infrastructure for ML forecast engines to application ecosystems and operational productization.

In this chapter we focus on the relationship between **Anemoi** and **FRAIM**, since AICON is implemented in Anemoi, while FRAIM represents an integration-oriented ecosystem around AI components for products and services.

- **Forecast-model-centric development:** build and train strong ML forecast engines (e.g. AIFS, AICON), i.e. the *model itself*.
- **Application- and operations-centric integration:** build robust end-to-end workflows around such models, enabling operational use, products, and tool orchestration.

In our course material, these two lines are represented by **Anemoi** (forecast model framework) and **FRAIM** (integration framework, products and services).

17.5.1 Anemoi: shared European framework for ML-based forecast engines

Anemoi is the shared European codebase for training and running ML-based weather models. From the AICON walkthrough perspective, Anemoi provides the complete *engine room*:

- dataset access and recipes (Zarr, GRIB, Xarray, Earthkit)
- graph construction for GNN-based forecasting (ICON / icosahedral meshes)
- model implementations (encoder–processor–decoder, GraphTransformer)
- driver code for training (Hydra configuration, Lightning training)
- checkpointing and experiment logging (e.g. MLFlow)
- inference tooling (rollouts, postprocessing, export)

Operationally, this means:

Anemoi delivers the ML forecast model stack as a reusable engine, including training and inference reproducibility.

17.5.2 FRAIM: integration and application ecosystem around AI products and services

In contrast, the slides introduce FRAIM as a complementary framework: it is not primarily a forecast model training framework, but rather a system for integrating AI methods into operational contexts and applications. This includes:

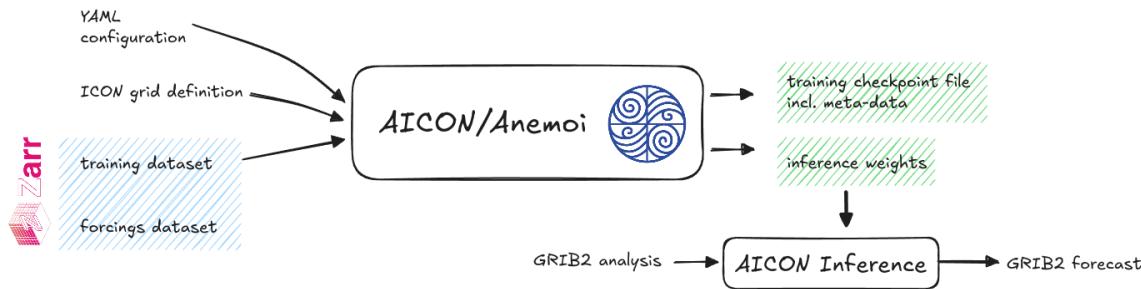


Figure 17.15: Anemoi as a shared framework for European ML-based weather prediction models (AIFS, AICON, BRIS). It provides data handling, graph construction, training driver code and inference.

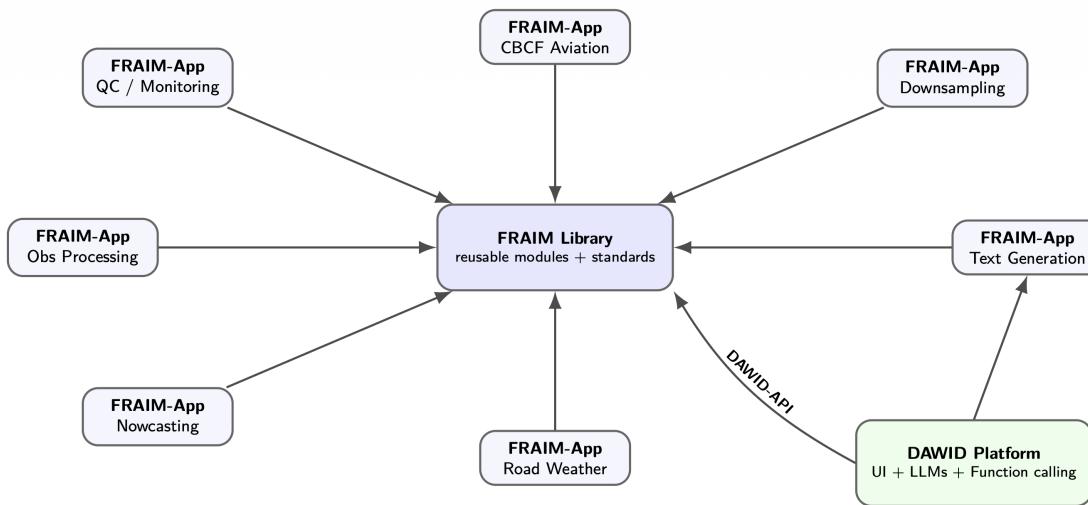


Figure 17.16: FRAIM overview: integration framework for AI-based meteorological applications and services.

- standardized interfaces to AI components (e.g. emulators, postprocessing, diagnostics)
- orchestration patterns for operational workflows
- multiple applications built around shared libraries and conventions
- tooling for deployment, monitoring, reproducibility and operational robustness

This viewpoint is particularly relevant in operational settings because the main difficulty is often not the *model architecture* itself, but the integration: data pipelines, I/O formats, metadata handling, quality control, product generation, and governance.

17.5.3 How Anemoi and FRAIM fit together (slide message)

The key message of the Anemoi vs. FRAIM slides is not competition but **complementarity**:

- **Anemoi answers:** *How do we build and train ML-based forecast engines that operate on meteorological meshes and variables?*
- **FRAIM answers:** *How do we integrate AI components into operational ecosystems and applications in a robust, scalable and maintainable way — and how do we turn raw forecasts into usable products and services?*

From a DWD perspective this yields a clean separation of responsibilities:

AICON is an Anemoi-based forecast engine. FRAIM provides the broader application, product and service ecosystem in which such forecast engines can be orchestrated, adapted and operationally exploited.

The advantage is that forecasting model development can stay aligned with the shared European framework (Anemoi), while the product and application layer remains flexible and can evolve quickly (FRAIM).

This distinction is essential for a National Meteorological Service: a large fraction of operational effort is not spent on the forecast model core, but on **products and services** built from the model output. In practice, the value chain includes many steps that cannot (and should not) all be hard-coded into the forecasting model itself.

In practical terms, this means:

- **Anemoi/AICON:** provide physically meaningful multi-variable forecast fields (including analysis-to-forecast rollouts) on the native model grid.
- **FRAIM:** turns raw fields into **operational products and services** by adding postprocessing, derived variables, feature extraction and verification workflows, for example:
 - road weather indices and decision support,
 - user-oriented evaluation and monitoring,
 - impact-based products (warnings, thresholds, risk indicators),
 - phenomena detection (storms, convection, icing, fog, extremes),
 - variable derivations, aggregation, interpolation, and tailored outputs.

17.5.4 Implementation viewpoint: an “engine + ecosystem” architecture

A useful mental model (close to software engineering) is an **engine + ecosystem** architecture:

- **Anemoi** corresponds to the *forecast engine layer*: data structures and formats, graph construction, ML model definitions, training driver code, checkpoints, and inference (including rollouts). It delivers *raw meteorological forecast fields* on the native model grid.
- **FRAIM** corresponds to the *platform, product and service layer*: interfaces, orchestration, operational lifecycle, monitoring and verification, as well as the generation of end-user oriented *products and services* derived from the forecast fields.

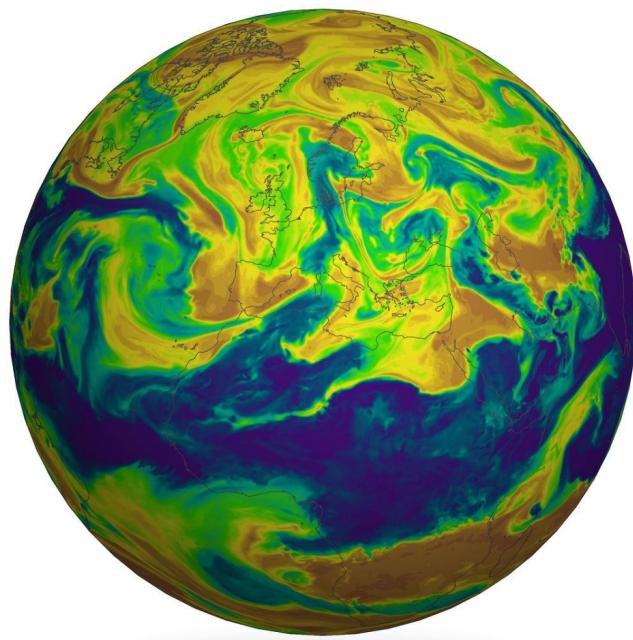


Figure 17.17: AICON as an Anemoi-based forecast engine producing raw forecast fields. FRAIM complements this by providing an operational platform for orchestration as well as product and service generation built on top of these fields.

This separation reflects the operational reality of a National Meteorological Service: a substantial fraction of the work and added value is created *after* the raw forecast has been produced — through postprocessing, derived variables, impact indicators, phenomena detection, and the delivery of tailored products to different user groups.

The architecture is particularly powerful for European collaborations: forecast engine development profits from shared effort and benchmarking in a common framework, while product and service development remains domain-specific and can be tailored to national responsibilities and user needs.

Chapter 18

AI Data Assimilation

18.1 Introduction to AI-based Data Assimilation

In the rapidly advancing field of Numerical Weather Prediction (NWP), the integration of observational data into numerical models is fundamental to achieving high-quality forecast accuracy. This process, known as *data assimilation*, combines observed data with model outputs to improve initial conditions and refine forecast predictions. Data assimilation techniques have evolved significantly over the years, with traditional methods such as *variational data assimilation (3D-Var)*, *ensemble Kalman filters (EnKF)*, and *particle filters* serving as the backbone of current NWP systems. These methods rely heavily on computational models to assimilate data and refine predictions, ensuring that weather forecasting remains reliable and precise.

Today, in the time of AI, we have two approaches to using observations. One approach is the direct integration of observations into the neural networks for forecasting, i.e. observations are used either additional or as only input for calculating a forecast based on neural network architectures. The second approach is to use observations to calculate an analysis minimizing a functional with a particular loss function motivated from Bayesian arguments. This reflects a traditional data assimilation approach, but now carried out with AI/ML based neural architectures. The AI-Var of Keller and Potthast is one of these algorithms, compare <https://arxiv.org/abs/2406.00390>.

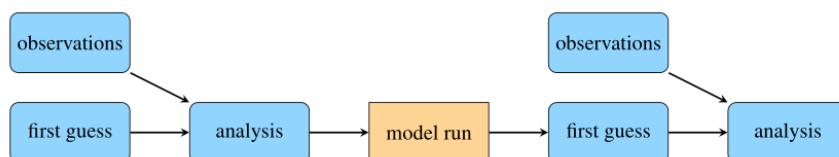


Figure 18.1: Classical data assimilation cycle, where observations and the first guess are integrated to generate an analysis. The cycle continues with model runs and updates based on new observations.

18.1.1 AI-Based Methods for Data Assimilation

The emergence of AI, particularly deep learning, presents an exciting opportunity to enhance the capabilities of NWP systems. AI-based models have shown the potential to emulate complex physical calculations traditionally handled by numerical models. Notably, AI can dramatically reduce computational costs, thereby speeding up data assimilation and forecasting processes. The primary advantage of AI in NWP is its ability to learn complex relationships from large datasets, allowing for faster and more flexible predictions.

The main objective of using AI in NWP is to create more efficient models that can predict weather outcomes with minimal computational overhead. This is for example accomplished by training deep learning models to replicate the processes typically used in traditional numerical models, such as simulating atmospheric dynamics, computing forecast scenarios, and applying corrections through assimilation.

18.1.2 AI-based Variational Data Assimilation (AI-Var)

In this tutorial, we introduce the concept of *AI-based variational data assimilation* (AI-Var), a novel approach that aims to replace traditional data assimilation techniques with AI-driven models. Unlike hybrid methods that combine machine learning with classical approaches, AI-Var fully integrates the data assimilation process into a neural network. The neural network is trained to minimize a cost function that mirrors the variational assimilation framework.

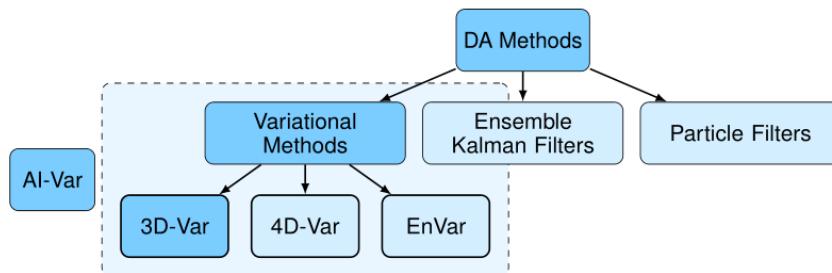


Figure 18.2: The AI-Var method within the broader context of data assimilation methods. AI-Var is a part of variational methods, which include 3D-Var, 4D-Var, and EnVar, all of which belong to the family of Data Assimilation (DA) methods.

The standard variational data assimilation process (such as 3D-Var) seeks to minimize the following cost function:

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{x}_b)^T \mathbf{B}^{-1} (\mathbf{x} - \mathbf{x}_b) + \frac{1}{2}(\mathbf{y} - \mathbf{H}(\mathbf{x}))^T \mathbf{R}^{-1} (\mathbf{y} - \mathbf{H}(\mathbf{x}))$$

In this cost function: - \mathbf{x}_b is the background state (first guess), - \mathbf{y} are the observations, - \mathbf{B} is the background error covariance matrix, - \mathbf{R} is the observation error covariance matrix, - $\mathbf{H}(\mathbf{x})$ is the observation operator, which maps the model state to the observation space.

The neural network is trained to minimize this cost function, but without relying on pre-existing analysis datasets. Instead, it learns the mapping from input data (observations and first guess) directly to the analysis, making it a fully data-driven system.

18.1.3 Conceptual Framework for AI-Var

The AI-Var method integrates the data assimilation process into the structure of a neural network. Rather than relying on classical analysis methods, AI-Var learns to predict the analysis directly from the input data.

The approach can be illustrated conceptually as follows:

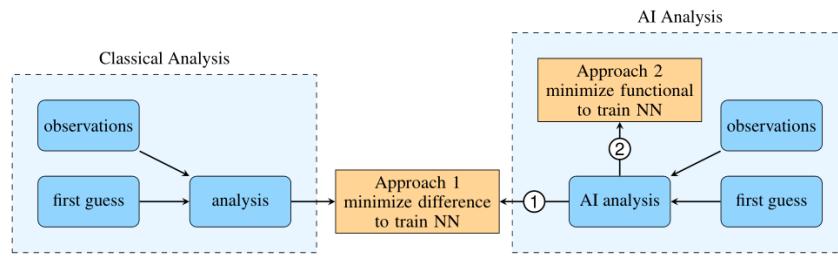


Figure 18.3: Conceptual framework for AI-based data assimilation, where the classical data assimilation process is replaced by a neural network that learns the analysis directly from the observations and first guess.

In this figure, the input data consists of the first guess (x_b) and the observations (y). Further, the neural network learns the functional relationship between these inputs and the desired analysis (x_a). The neural network is trained to minimize the variational cost function, ensuring that the analysis x_a is as close as possible to the optimal state based on the observations. The analysis x_a is not needed for this training process, such that we can retrain without calculating reanalysis fields beforehand.

18.1.4 AI-Var Training and Loss Function

The key to AI-Var is training a neural network to minimize the cost function defined earlier. The cost function can be directly used as the loss function for training the model. Instead of training on the predefined analysis states, the neural network learns to minimize the difference between the background state and the observations, incorporating the error covariances directly into the training process.

The loss function in AI-Var becomes:

$$L = (\hat{x} - x_b)^T \mathbf{B}^{-1} (\hat{x} - x_b) + (\hat{y} - y)^T \mathbf{R}^{-1} (\hat{y} - y)$$

where: \hat{x} is the output of the neural network (the predicted analysis), and $\hat{y} = \mathbf{H}(\hat{x})$ is the model equivalent of the predicted analysis.

This formulation allows for the direct integration of the data assimilation process into a deep learning framework, providing a computationally efficient and flexible alternative to traditional methods.

In conclusion, the AI-Var approach presents a promising alternative to traditional data assimilation techniques. By integrating the data assimilation process into a neural network, AI-Var can potentially improve computational efficiency and accuracy in weather forecasting. The initial results from both idealized and real-world test cases demonstrate the feasibility of replacing classical data

assimilation systems with AI-driven models. Future work will focus on further refining the AI-Var method, including the use of more advanced neural network architectures and exploring the application of AI in high-dimensional, real-world weather prediction scenarios.

18.2 Worked Example: 1D Inversion and AI-Var Learning

This section walks through the first practical examples of Lecture 18. We follow the two scripts `1_Inversion_1D.py` and `2_AI-VAR_1d.py`, and interpret the output through the figures produced in `images/img18/`.

18.2.1 Example A: 1D inversion with classical 3D-Var (`1_Inversion_1D.py`)

Setup. We consider a 1D periodic grid and construct:

- a smooth truth signal x_{true} ,
- an imperfect background x_b ,
- sparse noisy observations $y = Hx_{\text{true}} + \varepsilon$.

Observation operator. Observations are point values at selected grid indices. This is encoded by a linear observation operator H (subsampling matrix), such that

$$y \approx Hx_{\text{true}}.$$

A useful sanity check is comparing the observation vector y to Hx_b .

3D-Var analysis update. At each cycle we compute the 3D-Var analysis

$$x_a = x_b + K(y - Hx_b), \quad K = BH^\top(HBH^\top + R)^{-1}.$$

Here B is the background covariance matrix and R is the observation error covariance. In this 1D toy setup, R is typically diagonal (uncorrelated obs errors).

Core code: one analysis step. The key part of `1_Inversion_1D.py` can be summarized as:

```
python

1 # innovation
2 d = y - H @ xb
3
4 # Kalman/3D-Var gain
5 K = B @ H.T @ np.linalg.inv(H @ B @ H.T + R)
6
7 # analysis update
```

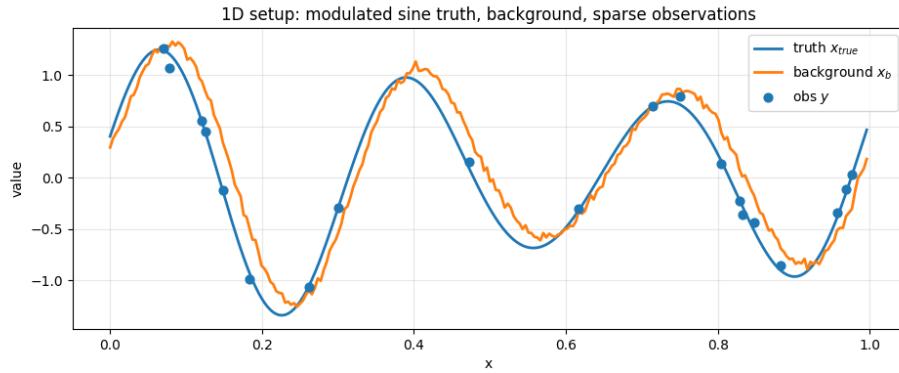


Figure 18.4: Truth, background, and sparse point observations for the 1D inversion test case.

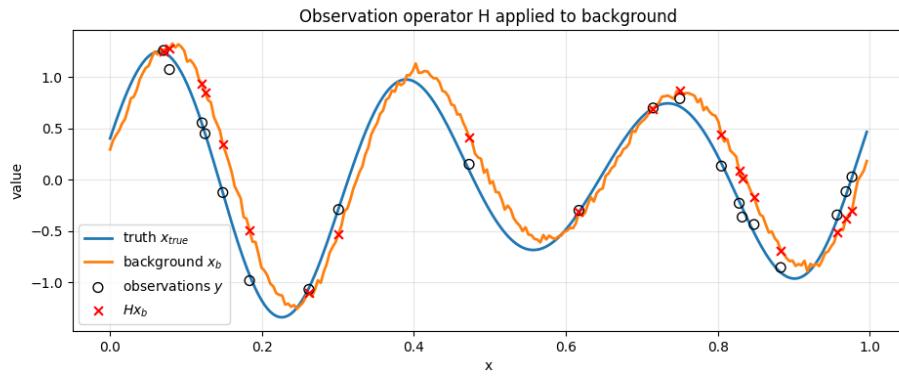


Figure 18.5: Observation operator check: innovation $d = y - Hx_b$ is the driver of the DA update.

```
8 xa = xb + K @ d
```

Role of the B matrix. The background covariance controls how far observation information spreads to unobserved grid points. A common choice is a Gaussian correlation kernel:

$$B_{ij} = \sigma_b^2 \exp\left(-\frac{|i-j|^2}{2L^2}\right).$$

This leads to a smooth, spatially coherent increment.

Reconstruction result. The analysis x_a combines the large-scale structure of x_b with the pointwise corrections from y , while remaining smooth due to the covariance constraints.

Multiple cycles. The script also demonstrates cycling: after the first analysis, one can use

$$x_b \leftarrow x_a$$

and repeat the update. This shows how iterative DA gradually reduces remaining errors.

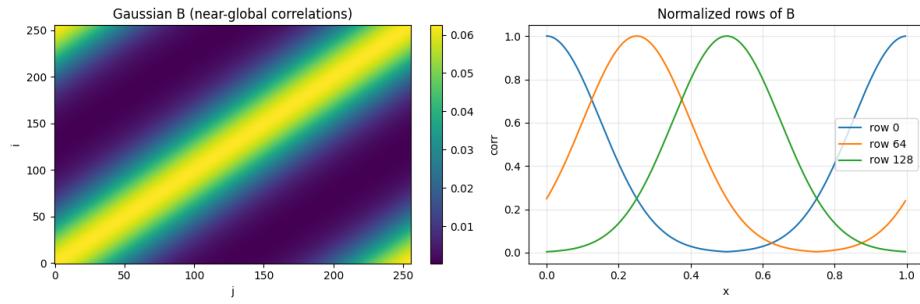


Figure 18.6: Example of a Gaussian B matrix used for 1D inversion: nearby grid points are strongly correlated.

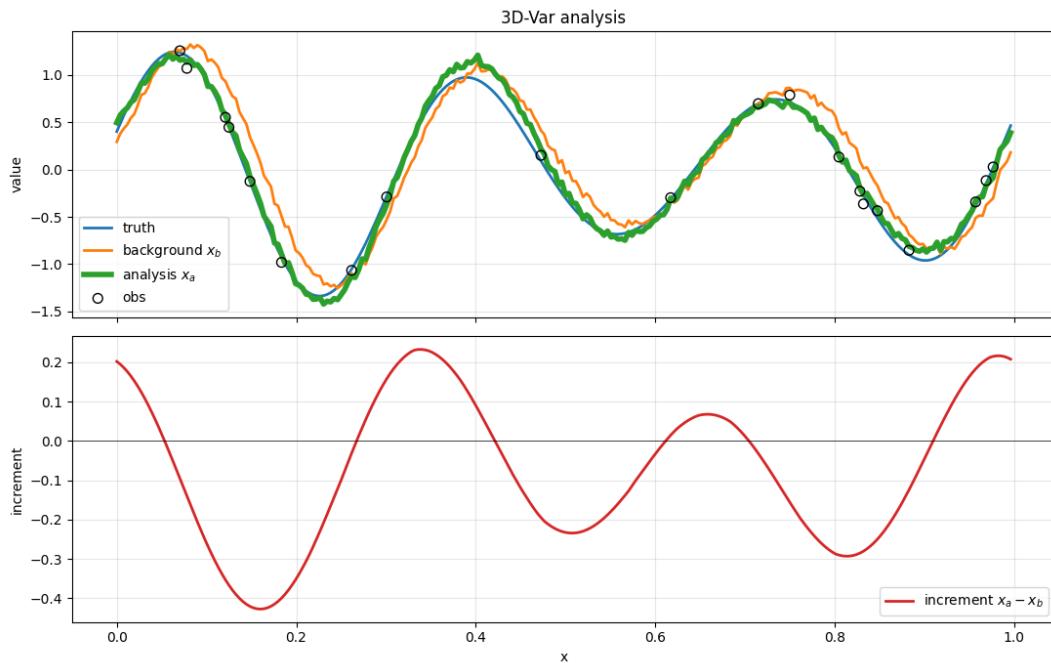


Figure 18.7: Classical 3D-Var inversion result: reconstructed analysis x_a compared to truth and background.

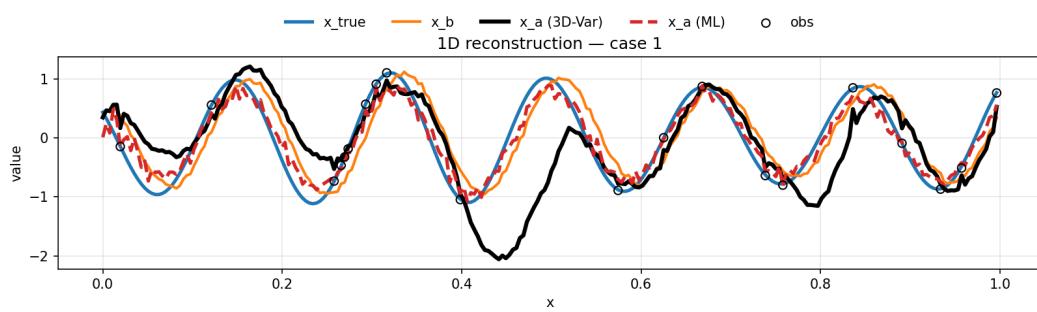
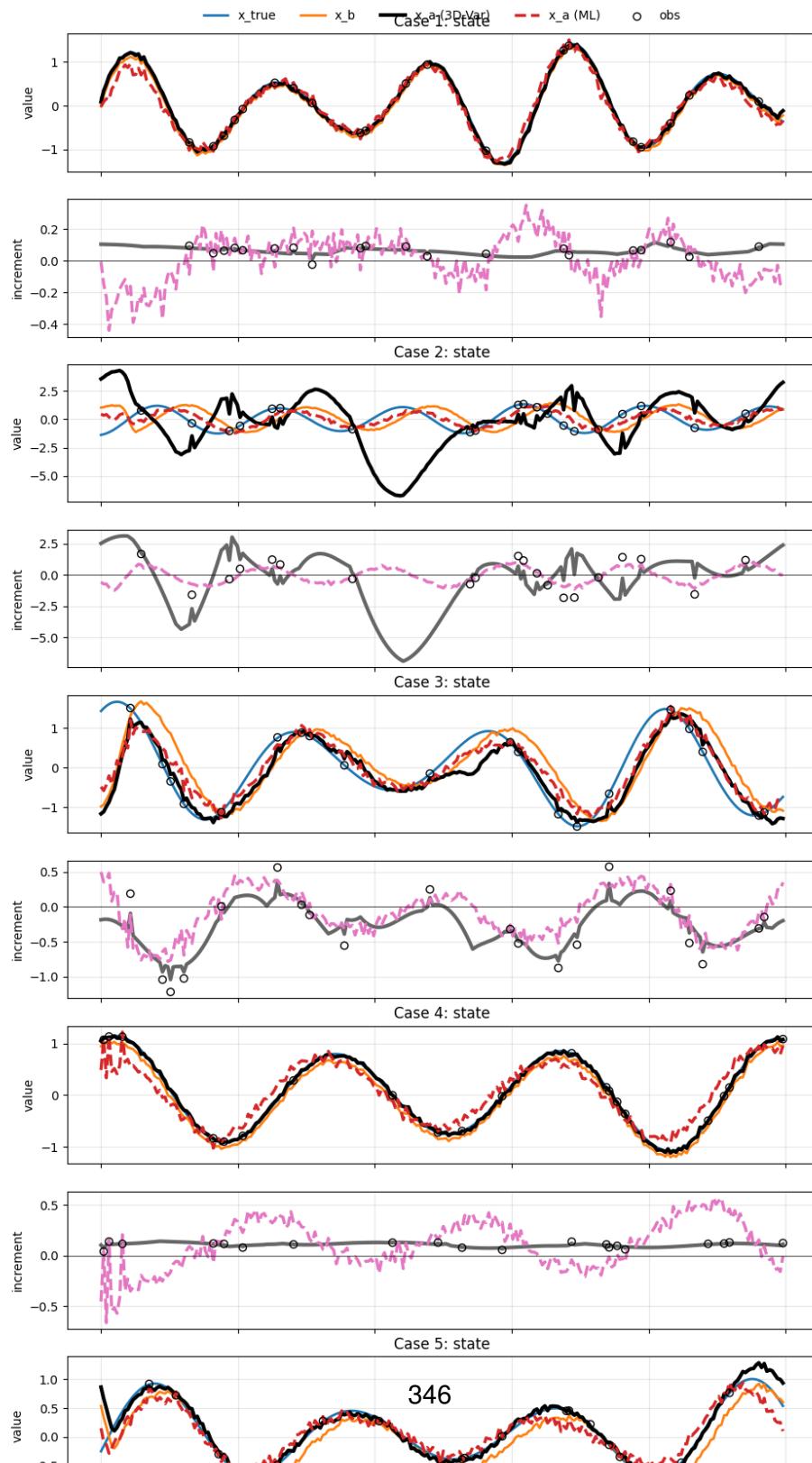


Figure 18.8: Iterative 3D-Var cycling: repeated assimilation reduces the error compared to truth.



18.2.2 Example B: Learning the inversion map (AI-Var 1D, 2_AI-VAR_1d.py)

Idea. Instead of explicitly building and inverting the gain expression in every case, we train a neural network to emulate the analysis mapping:

$$(x_b, y) \mapsto x_a.$$

Conceptually, AI-Var learns a *differentiable solver surrogate* for the DA update.

A typical architecture in this toy setup is an MLP that receives as input:

- background x_b (full field),
- observations y (sparse values),
- optionally observation mask / indices (so the network knows which points are observed).

Training data generation. Training samples are generated by repeatedly:

1. drawing a random truth field,
2. generating a background field,
3. sampling sparse noisy observations,
4. computing the classical 3D-Var analysis x_a (as ground truth target).

Core code structure (simplified). The workflow in 2_AI-VAR_1d.py follows the pattern:

```
python

1 # training data creation
2 xb, y = make_background_and_obs(...)
3 xa_target = three_dvar_analysis(xb, y, H, B, R)
4
5 # NN forward: predicted analysis
6 xa_pred = net(xb, y, mask)
7
8 # training
9 loss = mse(xa_pred, xa_target)
10 loss.backward()
11 optimizer.step()
```

Result: NN reconstruction. After training, the neural network can produce an analysis field that is close to the classical 3D-Var solution (and thus also close to the truth), but with a forward pass instead of explicit DA algebra.

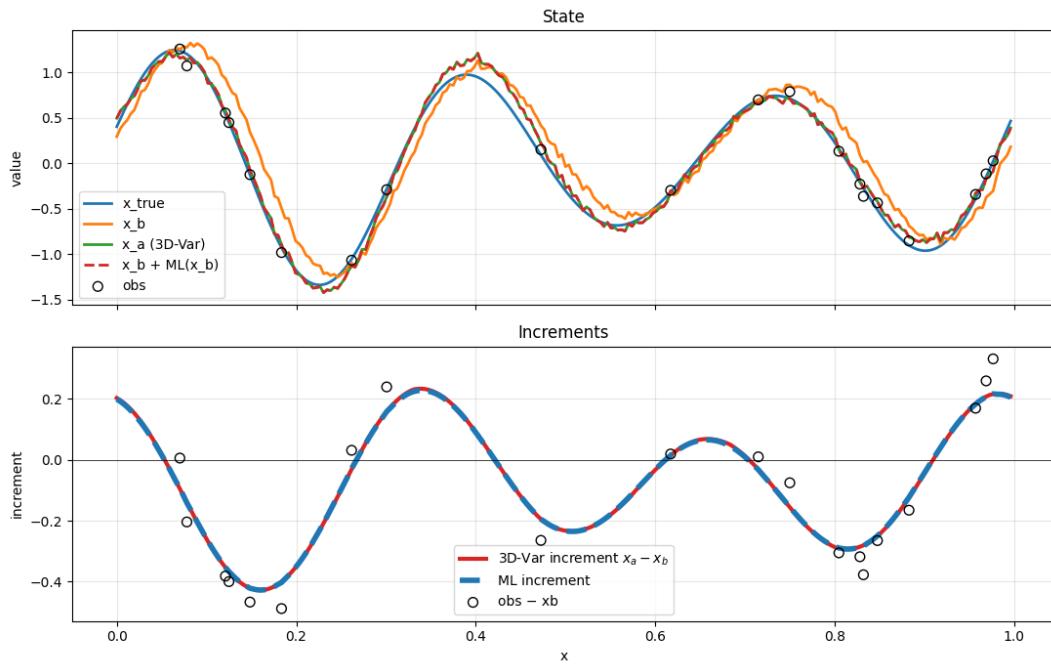


Figure 18.10: AI-Var 1D: neural network reconstruction compared to background and truth.

What is learned? The key point is that the network implicitly learns:

- how to spread sparse innovations spatially (a learned analogue of B),
- how strongly to trust observations (a learned analogue of balancing B vs R),
- how to combine these factors across many random training cases.

Connection to the AI-Var loop. This 1D experiment is the smallest working prototype of the AI-Var strategy:

(DA solver) \Rightarrow generate training targets \Rightarrow train differentiable surrogate \Rightarrow use surrogate inside iterative DA loops

Optional: interpreting the learned “implicit covariance”. In later experiments, AI-Var can be linked back to covariance structures by comparing increments and learned sensitivities. The lecture also provides reference visualizations for classical B -matrices in the AI-Var context.

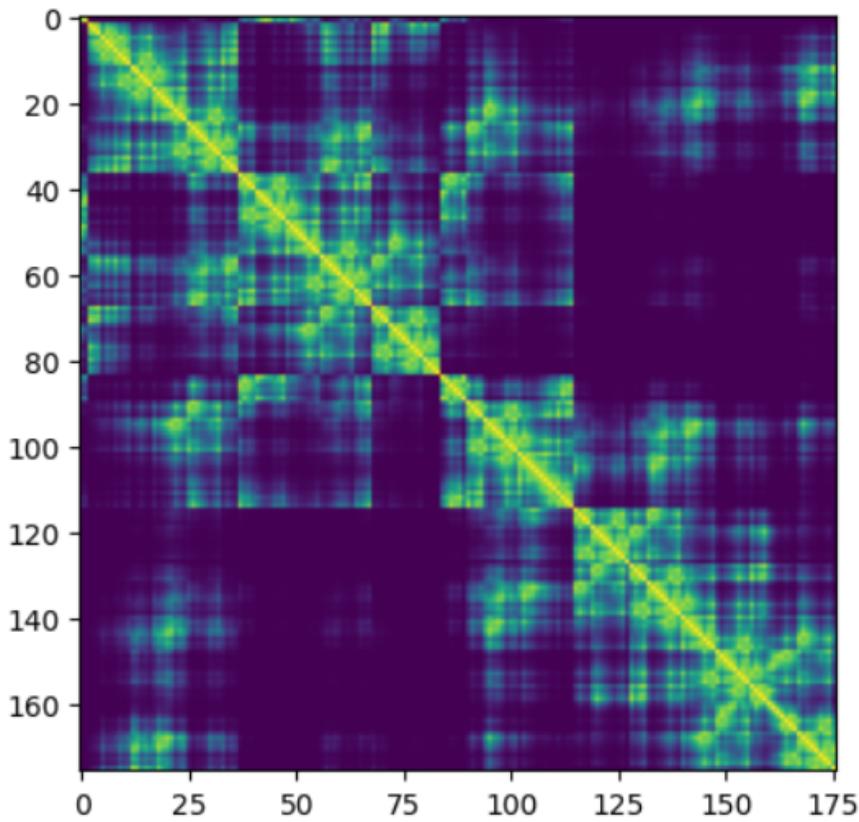


Figure 18.11: Illustration of a B -matrix concept in the AI-Var context (interpretation aid).

18.2.3 Example C: 2D assimilation and learned reconstruction (3_assimilation_2d.py)

We now extend the previous 1D inversion idea to a minimal 2D field. This example is intentionally small but already contains the key ingredients of operational DA problems:

- a spatially structured state $x(x, z)$,
- heterogeneous observations (different types / locations),
- a covariance model that propagates information into unobserved regions,
- and a neural surrogate that learns the reconstruction mapping.

2D truth and geometry. The script 3_assimilation_2d.py constructs a synthetic 2D “truth” field on a rectangular grid. You can think of this as a tiny atmosphere slice with two coordinates (x, z) :

$$x \in [0, L_x], \quad z \in [0, L_z], \quad \mathbf{x} \in \mathbb{R}^{n_x \times n_z}.$$

The goal is: reconstruct the full 2D field from sparse, partial observations.

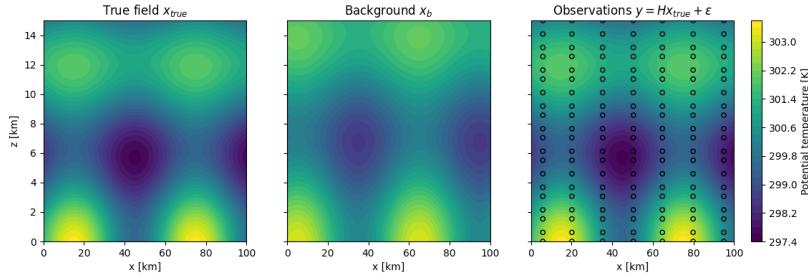


Figure 18.12: 2D toy setup: synthetic truth field on an (x, z) grid.

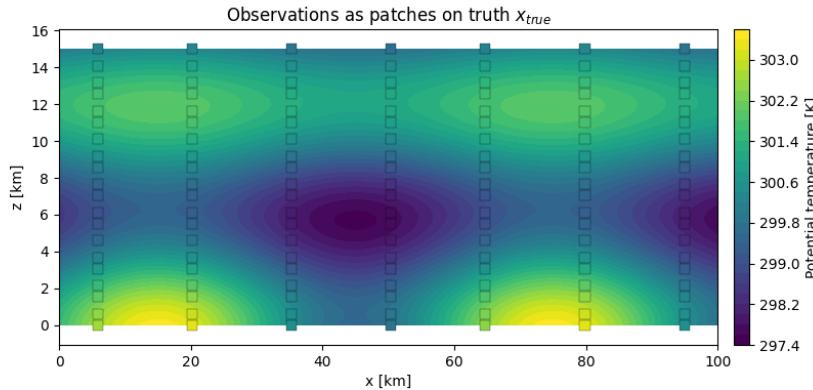


Figure 18.13: Observation layout and comparison to truth. Only a small subset of the full field is observed.

Observation types and incomplete coverage. In a 2D field, observations can be placed in many different ways:

- point observations at irregular locations,
- vertical profiles,
- integrated column observations,
- local averages, etc.

In the toy example, we construct a small set of observation locations and compare them to the underlying truth field. The observation operator H is still linear, but it now maps

$$H : \mathbb{R}^{n_x n_z} \rightarrow \mathbb{R}^m,$$

where the 2D field is flattened into a vector.

Background and covariance model in 2D. As before, the DA step combines:

- a background field x_b ,
- observations y ,

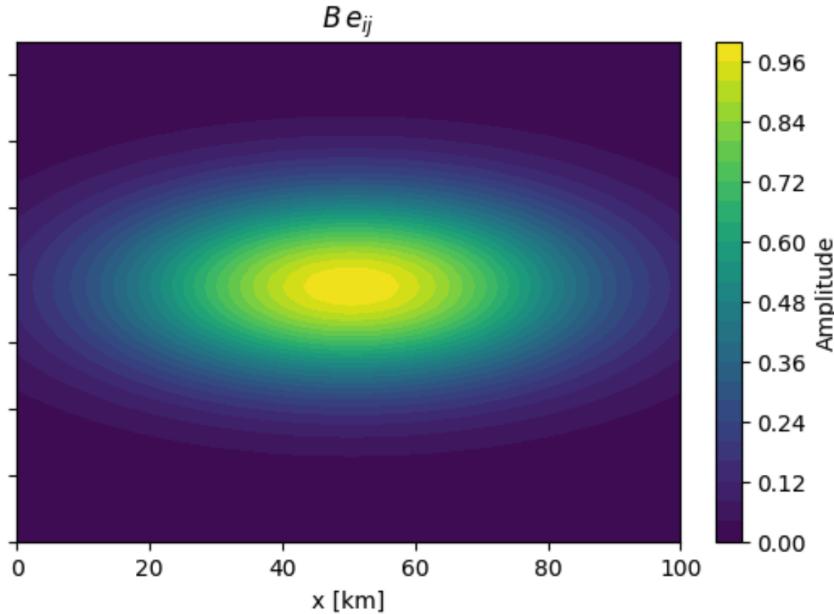


Figure 18.14: Example 2D covariance structure: correlations decay with distance. This controls smoothing and information spread.

- and the covariance balance expressed via (B, R) .

The difference is that in 2D the covariance structure becomes more visible:

B_{ij} encodes correlation between two grid points (x_i, z_i) and (x_j, z_j) .

Even a simple Gaussian kernel in 2D already spreads observation information in a physically meaningful neighborhood.

Classical 3D-Var analysis in 2D. The 3D-Var analysis is computed in exactly the same form as in 1D:

$$\mathbf{x}_a = \mathbf{x}_b + K(y - H\mathbf{x}_b), \quad K = BH^\top(HBH^\top + R)^{-1}.$$

In practice, the 2D example uses a smaller grid so the algebra can be performed explicitly. (In realistic systems, K is never formed explicitly; instead one solves the equivalent minimization problem iteratively.)

A compact skeleton of the analysis update remains:

```
python

1 # innovation
2 d = y - H @ xb
3
4 # gain
5 K = B @ H.T @ np.linalg.inv(H @ B @ H.T + R)
6
```

```
7 # analysis (vectorized field)
8 xa = xb + K @ d
```

Learning the 2D inversion map with an MLP. As in the 1D AI-Var example, we can train a network to approximate

$$(x_b, y) \mapsto x_a,$$

but now the target is a *2D field reconstruction*.

The important conceptual point is:

The network is not learning the dynamics here. It learns the *inverse problem mapping* from partial observations to a full state.

The script creates many random 2D cases and uses classical 3D-Var analyses as training targets.

Comparison: NN vs 3D-Var reconstruction. A highly instructive plot is the direct side-by-side comparison:

- classical 3D-Var analysis,
- NN reconstruction (AI-Var surrogate),
- difference patterns (what the NN struggles with / where it generalizes well).

Interpretation: what this example teaches. This 2D experiment adds two important messages beyond the 1D inversion:

(1) Covariance becomes geometry. In 2D, B is not just a smoothing matrix. It defines the geometry of how information travels from observed to unobserved regions.

(2) AI-Var learns structured spatial interpolation. The MLP effectively learns a data-driven reconstruction rule:

- observation increments at sparse locations,
- spatial propagation into missing regions,
- regularization consistent with the training distribution.

In later parts of the lecture this idea becomes crucial:

“reconstruct state” \Rightarrow “learn from reconstructed state”.

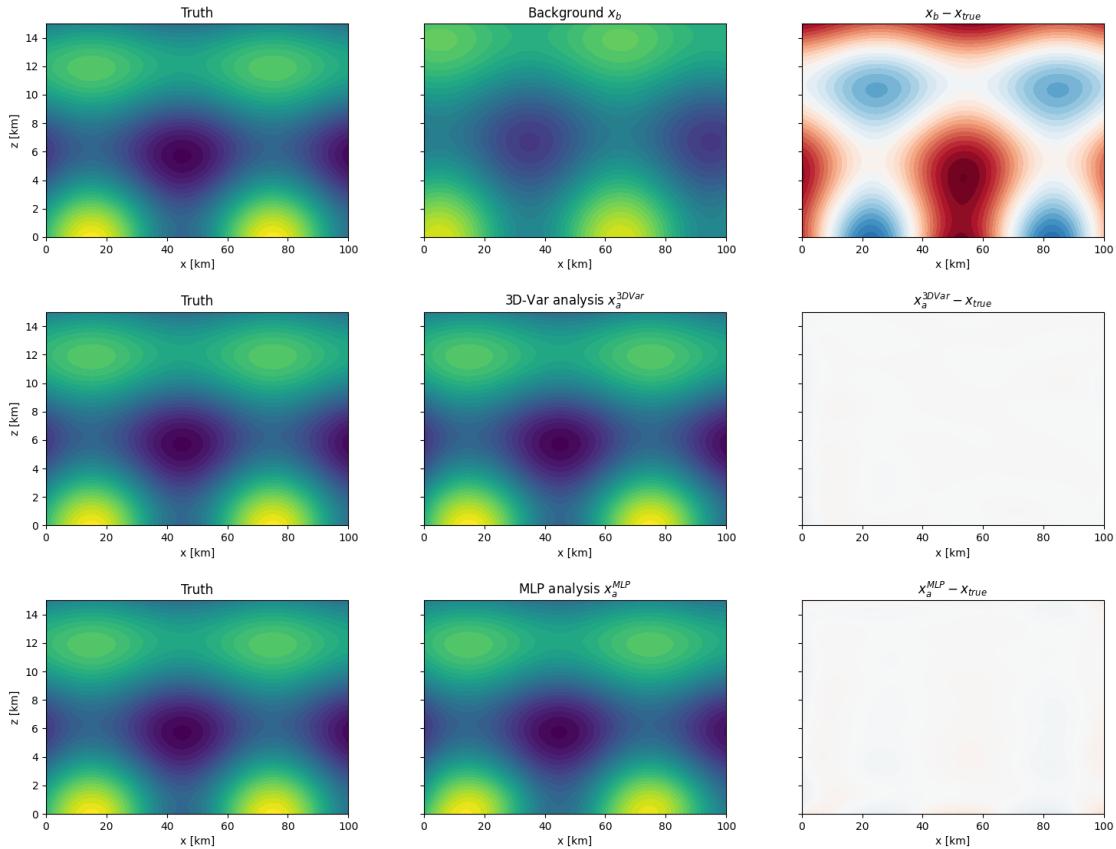


Figure 18.15: 2D reconstruction: NN (AI-Var) vs classical 3D-Var analysis. This is the first “real” spatial field inversion example.

18.3 AI Particle Filter (AIPF): Learning a Posterior Particle Distribution

18.3.1 Motivation: Why particle filters in AI-based DA?

In the previous sections we focused on AI-Var, i.e. learning the mapping from background state and observations to a *single* analysis field. This is a natural continuation of 3D-Var and 4D-Var.

However, many geophysical data assimilation problems involve *non-Gaussian* posterior distributions:

- multi-modal posteriors (ambiguities in dynamics or sparse observations),
- strongly nonlinear observation operators,
- strong nonlinearity in the forecast model.

In such cases, the correct target is not only an analysis mean, but a *posterior distribution*

$$p(x_n \mid y_{1:n}),$$

and this motivates particle filter ideas.

The key message of this tutorial section is:

AI-Var learns the analysis *state*. AIPF learns the analysis *distribution*.

18.3.2 Filtering formulation and particle representation

We consider a sequential estimation setup. At assimilation time n , the state is $x_n \in \mathbb{R}^d$ and we observe

$$y_n = H(x_n) + \varepsilon_n, \quad \varepsilon_n \sim \mathcal{N}(0, R).$$

The sequential Bayesian filtering target is the posterior

$$p(x_n \mid y_{1:n}).$$

A particle filter approximates the distribution by an ensemble

$$X_n = \{x_n^{(i)}\}_{i=1}^N.$$

In classical particle filters, the update step typically relies on resampling, MCMC moves, or importance weighting.

In this tutorial we use a different perspective: we *learn* a transformation that maps a prior ensemble to a posterior ensemble.

18.3.3 Core idea: a learned particle update

Let X_n^b denote the background (forecast) ensemble. The AIPF defines the analysis ensemble by a neural update

$$X_n^a = \mathcal{N}_\theta(X_n^b, y_n),$$

where \mathcal{N}_θ is a neural network with parameters θ .

This has two immediate implications:

- the analysis step becomes *inference* (one forward pass),
- the output is an ensemble that represents the posterior.

In our demonstration the state dimension is small ($d = 3$), but the formulation naturally extends to larger dimensional systems.

18.3.4 Example setup: Lorenz-63 filtering experiment

We test the AIPF concept on Lorenz-63. The state is

$$x = (x_1, x_2, x_3) \in \mathbb{R}^3,$$

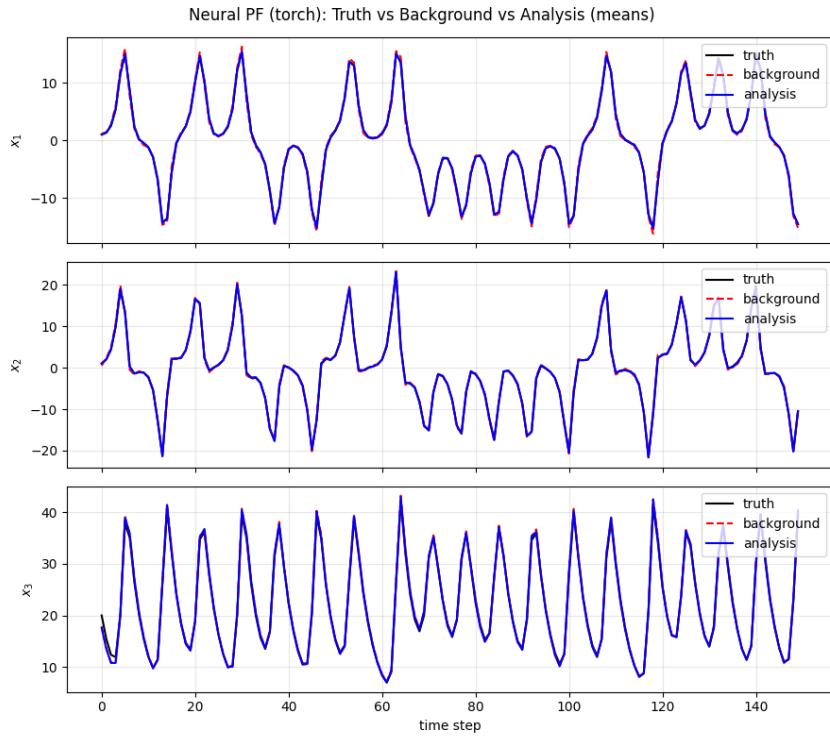


Figure 18.16: Lorenz-63 assimilation demo used for the AI particle filter section.

and we observe only a subset (e.g. $y = (x_1, x_2)$) with additive noise.

Additionally, the forecast model is deliberately biased or wrong. This is a realistic stress test: the analysis update must compensate for model mismatch.

Figure 18.16 illustrates the role of the AIPF in the Lorenz-63 loop.

18.3.5 Permutation invariance: DeepSets update network

A particle ensemble is an *unordered set*. Therefore, the update operator must be permutation invariant: reordering the particles must simply reorder the outputs, but not change the analysis distribution.

Formally, for any permutation π :

$$\mathcal{N}_\theta(\pi X^b, y) = \pi \mathcal{N}_\theta(X^b, y).$$

This tutorial implements the update using a DeepSets-style architecture. The structure is:

1. compute an embedding for each particle together with the observation,
2. pool embeddings across particles (mean pooling),
3. build a global context vector,
4. compute a particle-wise update conditioned on both local and global information.

Conceptually:

$$\phi(x_i, y) \rightarrow \text{pool} \rightarrow \rho(\cdot) \rightarrow \psi(x_i, \text{context}, y),$$

and the network outputs increments Δx_i so that

$$x_i^a = x_i^b + \Delta x_i.$$

This achieves the required permutation symmetry while enabling interaction between particles via the pooled context.

18.3.6 Gaussian mixture view: particles define a density

To train the network as a *distribution transform*, we interpret an ensemble as a Gaussian mixture density. Given particles $X = \{x^{(i)}\}$ and a fixed kernel covariance Σ :

$$q(x | X) = \frac{1}{N} \sum_{i=1}^N \mathcal{N}(x | x^{(i)}, \Sigma).$$

Thus:

- forecast ensemble X^b induces a prior mixture $q^b(x)$,
- analysis ensemble X^a induces an analysis mixture $q_\theta^a(x)$.

Training now becomes a density-matching problem: make the analysis mixture approximate the Bayesian posterior induced by prior and likelihood.

18.3.7 Training objective: likelihood + posterior-mass fit

The AIPF training objective has two components.

(A) Observation likelihood term. Particles should explain the observation. Under Gaussian observation noise (as in the code), we compute the log-likelihood of each analysis particle and aggregate across particles:

$$L_{\text{obs}} = -\log \left(\frac{1}{N} \sum_{i=1}^N p(y | x_i^a) \right).$$

This term alone encourages particles to match the observation, but it does *not* enforce a consistent posterior distribution.

(B) Distribution fitting term (Gaussian mixture KL). To enforce posterior consistency, we compare the analysis mixture to a target posterior mixture.

The code discretizes the comparison by using evaluation points

$$Z = \{z_k\}_{k=1}^K,$$

constructed from a single Kalman-style proposal step applied to the forecast particles. This yields stable evaluation points near the posterior mass.

The target posterior weights on Z are defined by Bayes rule:

$$w_k^* = \frac{q^b(z_k) p(y | z_k)}{\sum_{\ell=1}^K q^b(z_\ell) p(y | z_\ell)}.$$

Next, the analysis mixture is evaluated on the same points and normalized:

$$\pi_{\theta,k} = \frac{q_\theta^a(z_k)}{\sum_{\ell=1}^K q_\theta^a(z_\ell)}.$$

Finally, the distribution mismatch is measured by discrete KL divergence:

$$L_{GM} = \text{KL}(w^* \parallel \pi_\theta) = \sum_{k=1}^K w_k^* \log \frac{w_k^*}{\pi_{\theta,k}}.$$

Combined loss. The full objective is

$$L = L_{\text{obs}} + \lambda_{\text{bg}} L_{GM},$$

where λ_{bg} controls the strength of the distribution-matching (background) term.

This corresponds exactly to the implementation in the Gaussian-mixture training script.

18.3.8 Geometry of the update: prior ensemble → analysis ensemble

The learned update is best understood visually by plotting particles in a 2D slice of state space (e.g. x_1 - x_2 plane). Figure 18.17 shows:

- the prior forecast ensemble X^b ,
- observation and truth markers,
- the analysis ensemble X^a after the neural update.

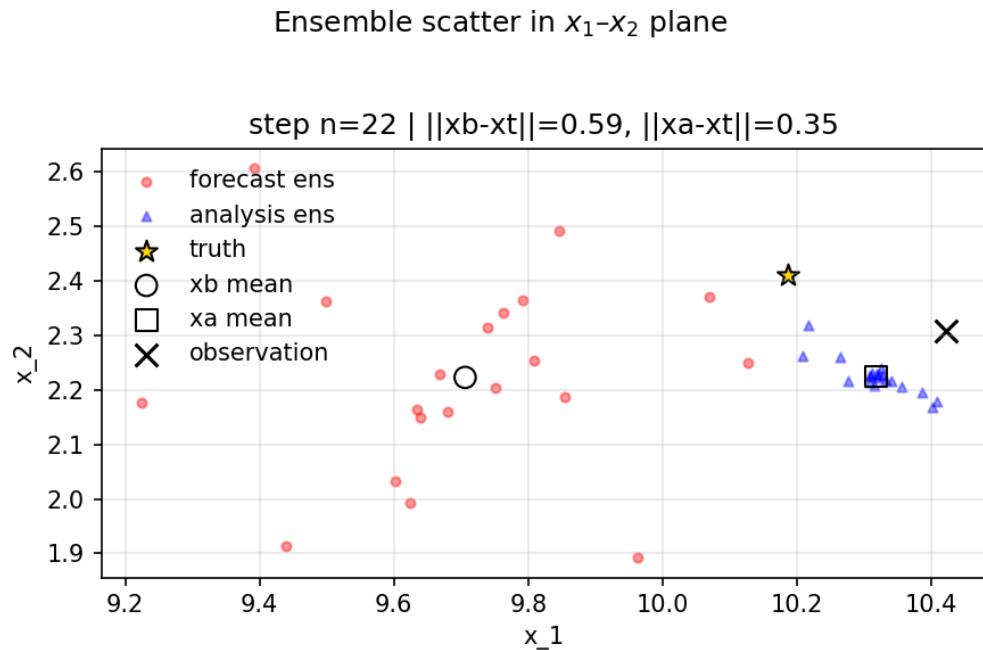


Figure 18.17: Ensemble geometry in a 2D slice: forecast particles (prior) transformed into analysis particles by the learned AIPF update.

This plot captures the essence of the method: instead of resampling, the network moves the particle cloud in a structured, posterior-aware way.

18.3.9 Ablation experiment: why the background term matters

A key point in this tutorial is that the distribution fitting term L_{GM} is essential.

Two networks are trained:

- net: trained with full loss $L_{obs} + \lambda_{bg}L_{GM}$,
- net_obs: trained with observation-only loss L_{obs} .

To compare them, we evaluate the difference in first-guess error across many assimilation cycles:

$$\Delta = FG_{err}(net_obs) - FG_{err}(net).$$

If $\Delta > 0$, the full method is better. Figure 18.18 shows that the full loss provides a consistent advantage, especially over long evaluation horizons.

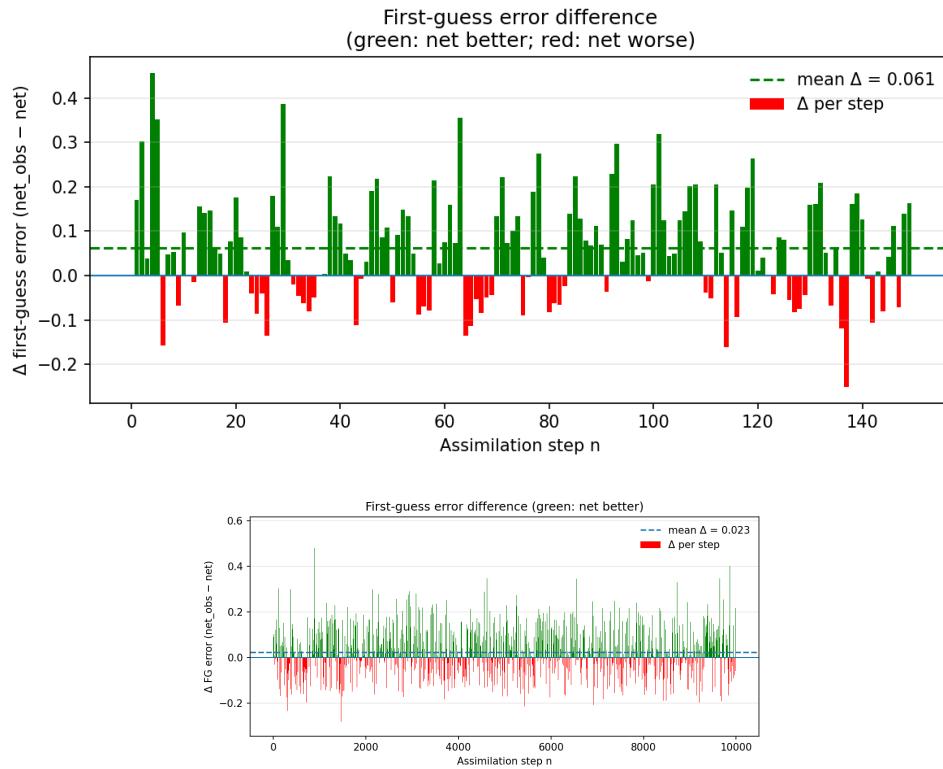


Figure 18.18: Ablation experiment. Difference in first-guess error between the obs-only network and the full AIPF loss. Positive values indicate skill gain from including the posterior/distribution fitting term.

18.3.10 Discussion and take-home messages

The AIPF provides a conceptually clean path toward non-Gaussian data assimilation with neural networks:

- the neural network learns an ensemble transform,
- DeepSets guarantees permutation invariance,
- training is distribution-aware via Gaussian-mixture KL fitting,
- the analysis output is a posterior ensemble, not just a mean.

In summary, the AIPF complements AI-Var:

- AI-Var learns an efficient approximation to the variational minimizer,
- AIPF learns the posterior distribution as an ensemble.

Chapter 19

AI and Physics and Data

19.1 Physics-Informed Neural Networks (PINNs)

19.1.1 Motivation and core idea

Physics-Informed Neural Networks (PINNs) provide a way to use neural networks as *continuous function approximators* while enforcing physical constraints through the loss function. The key difference to classical machine learning is that PINNs can learn without labelled target data by minimizing violations of known governing equations.

The unifying viewpoint is:

$$\text{state evolution must be consistent with } \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}).$$

PINNs enforce this consistency by combining:

- equation residual constraints (ODE/PDE),
- boundary and initial conditions (anchors),
- optionally additional physical constraints (e.g. conservation, symmetry).

In practice, PINNs are trained by evaluating the governing equations at *collocation points* x_i in the domain. This means that the “training data” is not labelled output pairs (x, y) , but instead the physics residual $r(x)$ computed from the neural network via automatic differentiation.

19.1.2 A minimal PINN example: the sine ODE

A minimal example is the second-order ODE

$$y''(x) + y(x) = 0,$$

with boundary conditions

$$y(0) = 0, \quad y'(0) = 1.$$

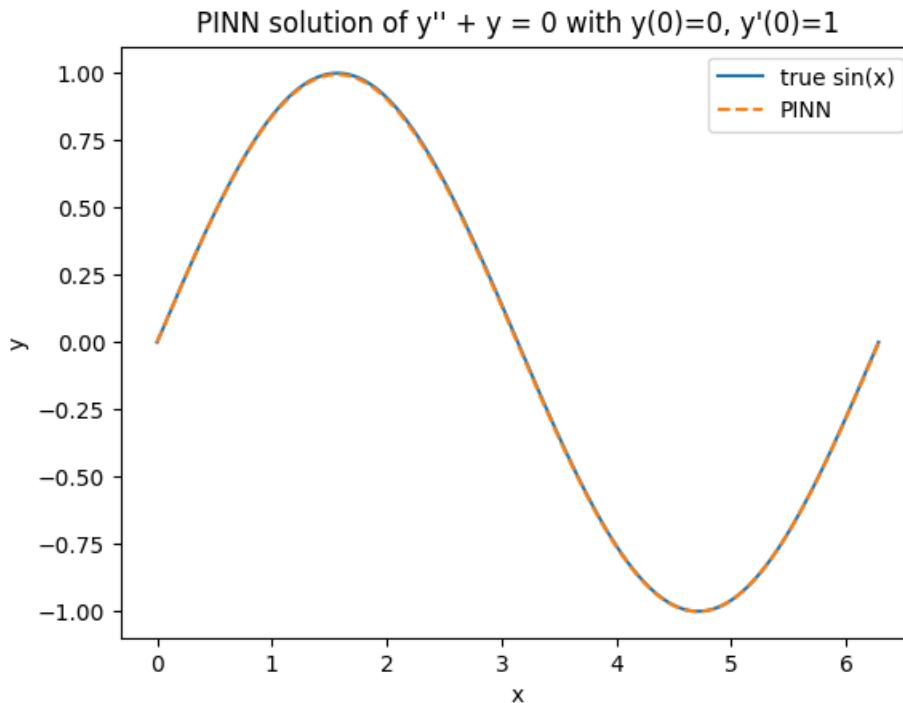


Figure 19.1: PINN example for $y''(x) + y(x) = 0$: neural approximation of the sine solution during training.

The unique solution is $y(x) = \sin(x)$.

The PINN approach represents the solution by a neural network $y_\theta(x)$ and trains it by minimizing:

- the ODE residual at collocation points,
- the boundary-condition mismatch at anchor points.

The model is a standard MLP $x \mapsto y_\theta(x)$ with tanh activations. All derivatives required by the physics loss are obtained by autograd.

PINN setup: neural ansatz, derivatives, sampling

```

1 # PINN: learn y(x)=sin(x) from ODE y'' + y = 0 with BC y(0)=0, y'(0)=1
2 import torch
3 import torch.nn as nn
4 import matplotlib.pyplot as plt
5 import math
6
7 torch.manual_seed(0)
8
9 # device selection (CUDA MPS CPU)
10 if torch.cuda.is_available():
11     device = torch.device("cuda")

```

```
12 elif torch.backends.mps.is_available():
13     device = torch.device("mps")
14 else:
15     device = torch.device("cpu")
16
17 # neural ansatz: MLP for y(x)
18 class MLP(nn.Module):
19     def __init__(self, width=64, depth=4):
20         super().__init__()
21         layers = [nn.Linear(1, width), nn.Tanh()]
22         for _ in range(depth - 1):
23             layers += [nn.Linear(width, width), nn.Tanh()]
24         layers += [nn.Linear(width, 1)]
25         self.net = nn.Sequential(*layers)
26         for m in self.net:    # Xavier init
27             if isinstance(m, nn.Linear):
28                 nn.init.xavier_normal_(m.weight)
29                 nn.init.zeros_(m.bias)
30
31     def forward(self, x):
32         return self.net(x)
33
34 model = MLP(width=64, depth=4).to(device)
35
36 # autograd helper: y', y``
37 def derivatives(y, x):
38     dy = torch.autograd.grad(y, x, grad_outputs=torch.ones_like(y),
39                             create_graph=True)[0]
40     d2y = torch.autograd.grad(dy, x, grad_outputs=torch.ones_like(dy),
41                             create_graph=True)[0]
42     return dy, d2y
43
44 # training domain and collocation points
45 x_min, x_max = 0.0, 2.0 * math.pi
46 N_col = 256
47
48 def sample_collocation(n):
49     x = x_min + (x_max - x_min) * torch.rand(n, 1, device=device)
50     x.requires_grad_(True)
51     return x
52
53 # boundary anchor point (x=0) for y(0)=0 and y'(0)=1
54 x0 = torch.tensor([[0.0]], device=device, requires_grad=True)
```

19.1.3 PINN loss formulation

Using automatic differentiation, we evaluate $y'_\theta(x)$ and $y''_\theta(x)$. The residual is

$$r(x) = y''_\theta(x) + y_\theta(x).$$

A typical residual loss is

$$\mathcal{L}_{\text{ODE}} = \frac{1}{N} \sum_{i=1}^N (y''_\theta(x_i) + y_\theta(x_i))^2.$$

Boundary conditions enforce uniqueness:

$$\mathcal{L}_{\text{BC}} = (y_\theta(0))^2 + (y'_\theta(0) - 1)^2.$$

The total PINN loss reads

$$\mathcal{L} = \mathcal{L}_{\text{ODE}} + \lambda \mathcal{L}_{\text{BC}}.$$

Important: No data term appears in this objective; learning is driven by physics constraints.

The training loop therefore looks very similar to standard deep learning, but the loss is constructed from physics residuals and anchor constraints:

Training loop: minimize ODE residual + boundary anchors

```

1 opt = torch.optim.Adam(model.parameters(), lr=2e-3)
2
3 w_ode = 1.0
4 w_bc = 10.0 # emphasize boundary constraints to fix the unique solution
5
6 for ep in range(1, 4001):
7     opt.zero_grad()
8
9     # --- ODE residual on collocation points
10    x = sample_collocation(N_col)
11    y = model(x)
12    dy, d2y = derivatives(y, x)
13    r = d2y + y
14    loss_ode = torch.mean(r**2)
15
16    # --- boundary conditions at x=0
17    y0 = model(x0)
18    dy0, _ = derivatives(y0, x0)
19    loss_bc = (y0**2).mean() + ((dy0 - 1.0)**2).mean()
20
21    loss = w_ode * loss_ode + w_bc * loss_bc
22    loss.backward()
23    opt.step()
24
25    if ep % 400 == 0 or ep == 1:
26        print(f"ep {ep:4d} | total={loss.item():.3e} "

```

```
27         f" | ode={loss_ode.item():.3e} | bc={loss_bc.item():.3e}")
```

After training, we evaluate the learned function $y_\theta(x)$ and compare it to the analytic solution $\sin(x)$:

Evaluation on the core domain and figure export

```
1 model.eval()
2
3 xx = torch.linspace(x_min, x_max, 500, device=device).view(-1, 1)
4 with torch.no_grad():
5     yy = model(xx)
6
7 xx_cpu = xx.cpu().numpy().reshape(-1)
8 yy_cpu = yy.cpu().numpy().reshape(-1)
9 true    = torch.sin(xx).cpu().numpy().reshape(-1)
10
11 plt.figure()
12 plt.plot(xx_cpu, true, label="true sin(x)")
13 plt.plot(xx_cpu, yy_cpu, "--", label="PINN")
14 plt.legend()
15 plt.xlabel("x")
16 plt.ylabel("y")
17 plt.title("PINN solution of y'' + y = 0 with y(0)=0, y'(0)=1")
18 plt.savefig("pinn_sine_1.png")
19 plt.show()
```

19.1.4 Extrapolation: testing outside the residual-sampling domain

A key strength (and challenge) of PINNs is extrapolation: models are often trained on a finite region but then evaluated outside.

However, extrapolation is not guaranteed. Even if the differential equation is correct, the network is only constrained where we *sample the residual*. Outside the sampled region, the model may drift because the loss provides no explicit penalty there.

We therefore distinguish:

- the **core domain** where the solution is anchored and strongly constrained,
- the **residual-sampled domain** where physics is enforced,
- the **outside domain** where only the inductive bias of the NN remains.

EVALUATION: test beyond the residual-sampling domain

```
1 # -----
2 # EVALUATION: test beyond the residual-sampling domain
```

```
3 # -----
4 import matplotlib.pyplot as plt
5
6 model.eval()
7
8 # test domain larger than training-sampling domain
9 x_test_min = -4.0 * math.pi
10 x_test_max = 6.0 * math.pi
11
12 xx = torch.linspace(x_test_min, x_test_max, 1600, device=device).view(-1, 1)
13 xx.requires_grad_(True)
14
15 y_pred = model(xx)
16 y_true = torch.sin(xx)
17
18 # absolute error
19 err = (y_pred - y_true).abs_()
20
21 # move to CPU
22 xx_cpu = xx.detach().cpu().numpy().reshape(-1)
23 y_pred_c = y_pred.detach().cpu().numpy().reshape(-1)
24 y_true_c = y_true.detach().cpu().numpy().reshape(-1)
25 err_c = err.detach().cpu().numpy().reshape(-1)
26
27 # -----
28 # plots: solution + error
29 #
30 plt.figure(figsize=(10, 4))
31 plt.plot(xx_cpu, y_true_c, label="true sin(x)")
32 plt.plot(xx_cpu, y_pred_c, "--", label="PINN")
33 plt.axvspan(-2*math.pi, 4*math.pi, color="gray", alpha=0.15, label="residual-
    sampled")
34 plt.axvspan(0, 2*math.pi, color="blue", alpha=0.08, label="core domain")
35 plt.legend()
36 plt.xlabel("x")
37 plt.ylabel("y")
38 plt.title("PINN extrapolation beyond residual-sampling domain")
39 plt.savefig("pinn_sine_3.png")
40 plt.show()
41
42 plt.figure(figsize=(10, 4))
43 plt.semilogy(xx_cpu, err_c + 1e-12)
44 plt.axvspan(-2*math.pi, 4*math.pi, color="gray", alpha=0.15)
45 plt.xlabel("x")
46 plt.ylabel("|error|")
47 plt.title("Absolute error (log scale)")
48 plt.show()
49
50 # -----
```

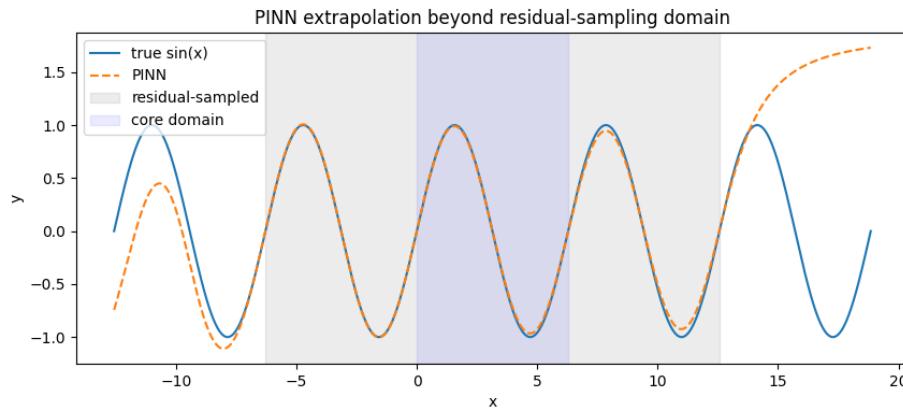


Figure 19.2: PINN extrapolation test: evaluation on a wider domain than the residual sampling region. The network matches $\sin(x)$ well where the ODE residual was enforced, but may drift outside.

```

51 # quantitative summary
52 # -----
53 inside = (xx_cpu >= x_col_min) & (xx_cpu <= x_col_max)
54 outside = ~inside
55
56 print("Mean absolute error:")
57 print(f"  inside residual-sampled domain : {err_c[inside].mean():.3e}")
58 print(f"  outside residual-sampled domain: {err_c[outside].mean():.3e}")

```

Figure 19.2 illustrates the typical behavior: within the sampled region the solution is accurate, while errors may grow outside. This is not a failure of the physics equation, but a consequence of how the PINN loss is enforced only at sampled points.

19.1.5 Representation choices: Fourier features

A second major lesson is that **representation matters**. Standard MLPs are biased toward smooth/low-frequency functions. Oscillatory solutions are often better represented using Fourier features.

A Fourier embedding maps x into periodic features

$$x \mapsto (\sin(\omega_k x), \cos(\omega_k x))_{k=1}^m,$$

which makes oscillatory structure easier to represent. Importantly, the physics loss stays unchanged; only the neural parameterization of the function changes.

Here, this is demonstrated by a PINN variant that keeps the training domain fixed to $[0, 2\pi]$ but replaces the raw input x by Fourier features.

PINN with Fourier Features: oscillatory representation bias

```

1 # -----
2 # PINN with Fourier Features: learn y(x)=sin(x) from y''+y=0 and BC y(0)=0, y'(0)
3 # =1
4 # (Fourier features = input embedding; no sampling outside, no PBCs)
5 # -----
6
7 import torch
8 import torch.nn as nn
9 import matplotlib.pyplot as plt
10 import math
11
12 torch.manual_seed(0)
13
14 # device selection
15 if torch.cuda.is_available():
16     device = torch.device("cuda")
17 elif torch.backends.mps.is_available():
18     device = torch.device("mps")
19 else:
20     device = torch.device("cpu")
21
22 # -----
23 # Fourier feature embedding + small MLP
24 # -----
25
26 class FourierMLP(nn.Module):
27     def __init__(self, m=16, max_freq=16.0, width=64, depth=3):
28         super().__init__()
29         omegas = torch.linspace(1.0, max_freq, m).view(1, m)
30         self.register_buffer("omegas", omegas)
31         in_dim = 2 * m
32
33         layers = [nn.Linear(in_dim, width), nn.Tanh()]
34         for _ in range(depth - 1):
35             layers += [nn.Linear(width, width), nn.Tanh()]
36         layers += [nn.Linear(width, 1)]
37         self.net = nn.Sequential(*layers)
38
39         for mod in self.net:
40             if isinstance(mod, nn.Linear):
41                 nn.init.xavier_normal_(mod.weight)
42                 nn.init.zeros_(mod.bias)
43
44     def features(self, x):
45         z = x @ self.omegas
46         return torch.cat([torch.sin(z), torch.cos(z)], dim=1)
47
48     def forward(self, x):
49         return self.net(self.features(x))

```

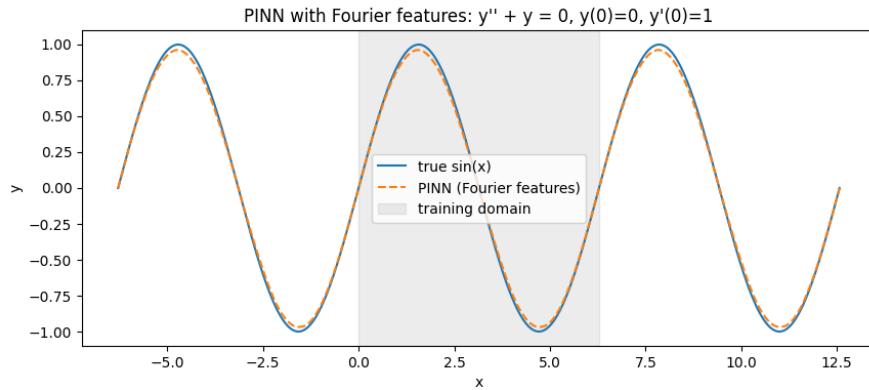


Figure 19.3: PINN with Fourier features: the oscillatory embedding improves representation of $\sin(x)$ and supports better extrapolation outside $[0, 2\pi]$.

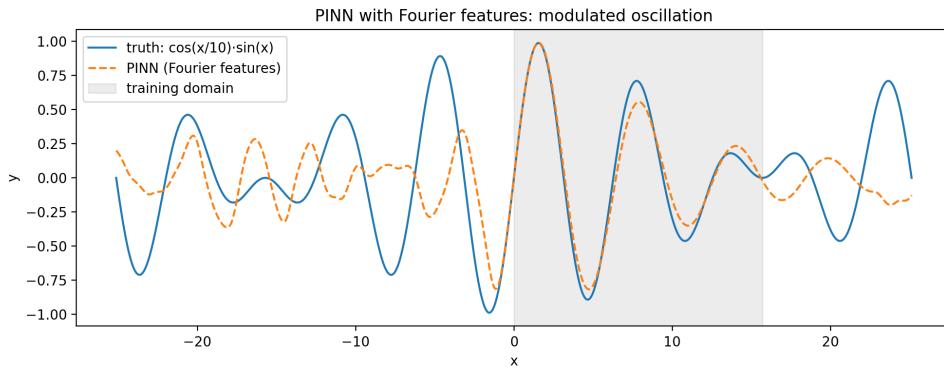


Figure 19.4: Fourier-type representation (sine/cosine features) improves representation of oscillatory solutions and supports extrapolation.

47

```
48 model = FourierMLP(m=16, max_freq=16.0, width=64, depth=3).to(device)
```

The training procedure is the same as above (ODE residual + boundary anchors), but now the network has an inductive bias toward periodic functions. Even without expanding the sampling region, this often leads to much better generalization beyond the training interval.

Take-home message: PINNs are not automatically robust extrapolators. Their behavior depends strongly on (i) the residual-sampling strategy and (ii) the representational bias of the neural network. Fourier features provide a simple yet powerful improvement for oscillatory problems.

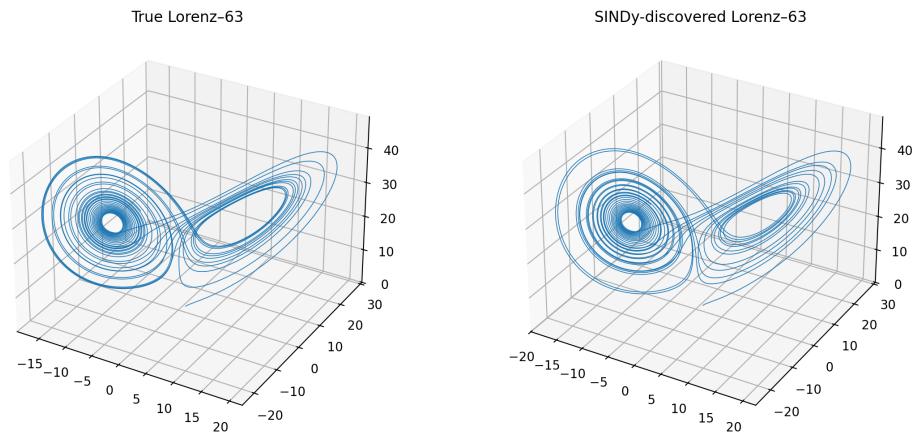


Figure 19.5: SINDy principle: construct a candidate function library and identify a sparse set of active terms by sparse regression.

19.2 Discovering Governing Equations from Data (SINDy)

19.2.1 Goal: equations from time series

In contrast to PINNs, SINDy focuses on *discovering* governing equations directly from observational data. Assume a dynamical system

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t)), \quad \mathbf{x}(t) \in \mathbb{R}^n,$$

and observe time series $\mathbf{x}(t_i)$.

The goal is to learn a functional form for \mathbf{f} that is:

- **predictive** (reproduces dynamics),
- **interpretable** (explicit equation structure),
- **parsimonious** (few relevant terms).

The key assumption behind SINDy is **sparsity**: \mathbf{f} can be expressed as a sparse combination of candidate functions. This connects SINDy to classical system identification, regression, and model selection techniques, and it remains one of the most successful approaches for interpretable equation discovery.

19.2.2 Function library and sparse regression

SINDy builds a library of candidate nonlinear terms

$$\Theta(\mathbf{x}) = [1, x_1, \dots, x_n, x_1x_2, x_1^2, \dots],$$

and assumes

$$\dot{\mathbf{x}} \approx \Theta(\mathbf{x}) \Xi,$$

where Ξ is sparse.

In this view, equation discovery becomes a **sparse regression problem**: find coefficients Ξ such that only a few basis functions are active. A standard least-squares fit gives a dense solution, therefore sparsity is enforced by thresholding, ℓ_1 penalties, or sequential refitting (“sparsify then refit”).

A minimal sequential thresholding scheme is:

1. solve a dense regression $\Xi = \arg \min \|\Theta\Xi - \dot{\mathbf{x}}\|^2$,
2. set small coefficients to zero,
3. refit only on remaining terms, repeat.

Interpretability benefit: The output is an explicit sparse equation structure rather than a black box model. This is a strong advantage in scientific applications, where we often need mechanistic understanding in addition to predictive power.

SINDy core step: candidate library and sparse regression

```

1 # -----
2 # Phase 2: SINDy REGRESSION (this is where Xi is computed)
3 # -----
4 x, y, z = X.T
5
6 Theta = np.column_stack([
7     np.ones_like(x),
8     x, y, z,
9     x * y,
10    x * z,
11    y * z
12])
13
14 feature_names = ["1", "x", "y", "z", "x*y", "x*z", "y*z"]
15
16 def sindy(Theta, dXdt, lam=0.1, n_iter=10):
17     # ---- REGRESSION #1 (dense)
18     Xi = np.linalg.lstsq(Theta, dXdt, rcond=None)[0]
19
20     for _ in range(n_iter):
21         small = np.abs(Xi) < lam
22         Xi[small] = 0.0
23
24     # ---- REGRESSION #2 (sparse refit)
25     for i in range(dXdt.shape[1]):
26         big = ~small[:, i]
27         Xi[big, i] = np.linalg.lstsq(
28             Theta[:, big], dXdt[:, i], rcond=None
29         )[0]
30
31 return Xi

```

19.2.3 Example: Lorenz–63 from trajectory data

A classical benchmark for equation discovery is the Lorenz–63 system:

$$\begin{aligned}\dot{x} &= \sigma(y - x), \\ \dot{y} &= x(\rho - z) - y, \\ \dot{z} &= xy - \beta z.\end{aligned}$$

Here σ , ρ , and β are parameters and the dynamics is chaotic.

A typical workflow for SINDy is:

1. generate (or observe) trajectory data $\mathbf{x}(t_i)$,
2. estimate time derivatives $\dot{\mathbf{x}}(t_i)$,
3. build the library $\Theta(\mathbf{x})$,
4. compute sparse coefficients Ξ ,
5. integrate the discovered system and compare.

For Lorenz–63, SINDy can recover the correct active terms from trajectory data *if the derivatives are sufficiently accurate*. This highlights an important point: classical system identification methods can be extremely powerful—but they critically depend on the quality of numerical differentiation and conditioning of the regression problem.

SINDy Lorenz-63: generate trajectory, discover equations, compare

```

1 # -----
2 # Phase 1: Generate Lorenz-63 data
3 #
4 sigma = 10.0
5 rho   = 28.0
6 beta   = 8.0 / 3.0
7
8 def lorenz63_true(t, X):
9     x, y, z = X
10    return [
11        sigma * (y - x),
12        x * (rho - z) - y,
13        x * y - beta * z
14    ]
15
16 t_span = (0.0, 25.0)
17 t_eval = np.linspace(*t_span, 5000)
18 x0 = [1.0, 1.0, 1.0]
19
20 sol = solve_ivp(lorenz63_true, t_span, x0, t_eval=t_eval)
21 X = sol.y.T
22 dt = t_eval[1] - t_eval[0]
```

```

23
24 # Numerical derivatives
25 dXdt = np.gradient(X, dt, axis=0)
26
27 # -----
28 # Phase 2: SINDy regression -> Xi
29 # -----
30 Xi = sindy(Theta, dXdt)
31
32 # -----
33 # Phase 3: Integrate discovered system and compare
34 # -----
35 Xi_sindy = Xi.copy()
36
37 def lorenz63_sindy(t, X):
38     x, y, z = X
39     Theta_vec = np.array([1.0, x, y, z, x*y, x*z, y*z])
40     return (Theta_vec @ Xi_sindy).tolist()
41
42 sol_true = solve_ivp(lorenz63_true, (0.0, 30.0), x0, t_eval=np.linspace(0, 30,
   6000))
43 sol_sindy = solve_ivp(lorenz63_sindy, (0.0, 30.0), x0, t_eval=np.linspace(0, 30,
   6000))
44
45 # Save comparison figure
46 plt.figure(figsize=(12, 5))
47 ax1 = plt.subplot(121, projection="3d")
48 ax1.plot(sol_true.y[0], sol_true.y[1], sol_true.y[2], lw=0.6)
49 ax1.set_title("True Lorenz-63")
50
51 ax2 = plt.subplot(122, projection="3d")
52 ax2.plot(sol_sindy.y[0], sol_sindy.y[1], sol_sindy.y[2], lw=0.6)
53 ax2.set_title("SINDy-discovered Lorenz-63")
54
55 plt.tight_layout()
56 plt.savefig("SINDy_01.png", dpi=200)
57 plt.show()

```

19.2.4 Practical limitation: derivative estimation in noisy data

A core limitation of classical SINDy is the need for time derivatives. When observational noise is present, numerical differentiation becomes unstable and can dominate the regression error. This is not a weakness of SINDy itself but a fundamental signal processing issue: differentiation amplifies noise.

Classical remedies are valuable and should be emphasized:

- smoothing / filtering before differentiation,

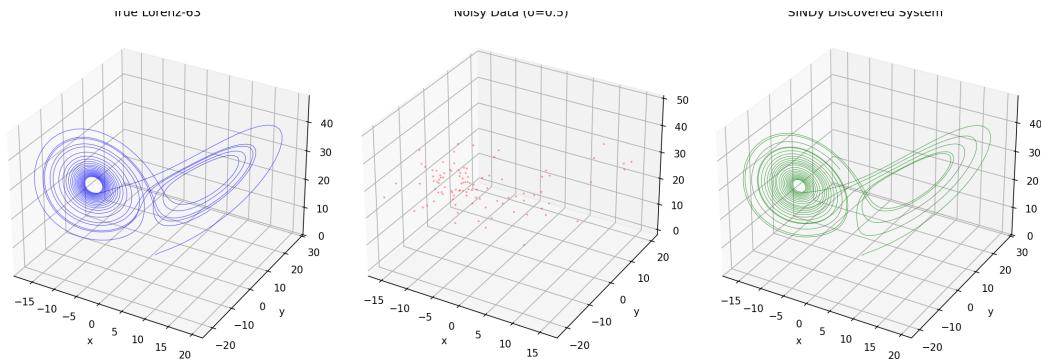


Figure 19.6: SINDy applied to Lorenz–63: recovering the active nonlinear terms in the governing equations from (noisy) trajectory data and comparing discovered and true trajectories.

- higher-order finite differences,
- total-variation regularized differentiation,
- spline fitting,
- carefully chosen sampling rates and noise models.

In many realistic use cases, these classical techniques are sufficient and yield excellent interpretable equation discovery.

However, we see that **neural methods can extend** classical approaches by providing stronger smoothing and more stable derivatives in highly noisy regimes, where traditional finite differences become unreliable.

19.2.5 Neural SINDy: denoise with NN, differentiate with autograd

Neural SINDy introduces a neural surrogate $\mathbf{x}_\theta(t)$ that learns a *smooth trajectory representation* from noisy observations. The network maps time to state:

$$t \mapsto \mathbf{x}_\theta(t) \in \mathbb{R}^3.$$

This step resembles classical smoothing (e.g. spline fitting), but uses a flexible neural parameterization.

Once $\mathbf{x}_\theta(t)$ is trained, derivatives are computed using automatic differentiation:

$$\dot{\mathbf{x}}_\theta(t) = \frac{d}{dt} \mathbf{x}_\theta(t),$$

which avoids noisy finite differences.

Finally, the sparse discovery step remains unchanged:

$$\dot{\mathbf{x}}_\theta(t) \approx \Theta(\mathbf{x}_\theta(t)) \Xi.$$

Key message: Neural networks are not replacing SINDy. They augment the pipeline by providing robust smoothing and derivatives, thus increasing the regime in which classical sparse regression can succeed.

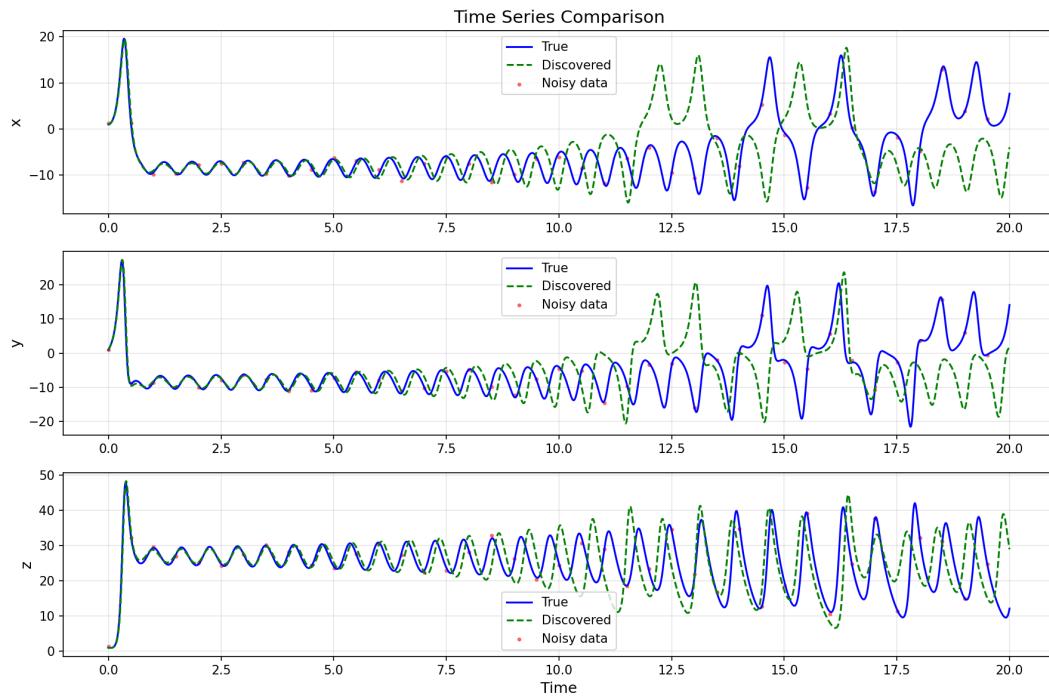


Figure 19.7: Observed time series used for equation discovery: SINDy infers governing equations from trajectories, but derivative quality becomes a limiting factor in the presence of noise.

Neural SINDy: learn smooth trajectory $t \rightarrow x(t)$ and use autograd derivatives

```

1 # -----
2 # Method 2: Neural Network to learn smooth trajectory
3 #
4 class TrajectoryNet(nn.Module):
5     """Network maps time -> state (x,y,z), giving a smooth interpolant."""
6     def __init__(self, hidden_dim=128, n_layers=4):
7         super().__init__()
8         layers = [nn.Linear(1, hidden_dim), nn.Tanh()]
9         for _ in range(n_layers - 1):
10            layers += [nn.Linear(hidden_dim, hidden_dim), nn.Tanh()]
11        layers += [nn.Linear(hidden_dim, 3)]
12        self.net = nn.Sequential(*layers)
13        for m in self.net:
14            if isinstance(m, nn.Linear):
15                nn.init.xavier_normal_(m.weight)
16                nn.init.zeros_(m.bias)
17
18    def forward(self, t):
19        return self.net(t)
20
21 # tensors and normalization
22 t_tensor = torch.tensor(t_eval, dtype=torch.float32, device=device).view(-1, 1)

```

```

23 X_noisy_tensor = torch.tensor(X_noisy, dtype=torch.float32, device=device)
24
25 t_min, t_max = t_tensor.min(), t_tensor.max()
26 t_normalized = (t_tensor - t_min) / (t_max - t_min)
27
28 # train smoothing network
29 model = TrajectoryNet(hidden_dim=128, n_layers=4).to(device)
30 optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
31
32 for epoch in range(45000):
33     optimizer.zero_grad()
34     X_pred = model(t_normalized)
35     loss = torch.mean((X_pred - X_noisy_tensor) ** 2)
36     loss.backward()
37     optimizer.step()
38
39 # autograd derivatives
40 model.eval()
41 t_normalized.requires_grad_(True)
42 X_smooth = model(t_normalized)
43
44 dXdt_autograd = []
45 for i in range(3):
46     grad = torch.autograd.grad(
47         outputs=X_smooth[:, i].sum(),
48         inputs=t_normalized,
49         retain_graph=True
50     )[0]
51     grad = grad / (t_max - t_min)    # undo time normalization
52     dXdt_autograd.append(grad)
53
54 dXdt_autograd = torch.stack(dXdt_autograd, dim=1)

```

19.2.6 Balanced perspective: classical strengths and AI extensions

It is important to take a balanced view.

Classical techniques remain highly valuable: SINDy and related sparse regression approaches are among the best tools for *interpretable* dynamical model discovery. When noise levels are moderate and derivatives can be estimated reliably (using filtering, splines, or regularized differentiation), classical SINDy often recovers equations with remarkable accuracy and scientific transparency.

AI and neural emulation extend the toolbox: Neural smoothing and autograd derivatives can widen the range of applicability, especially when:

- observational noise is high,
- sampling is uneven or limited,

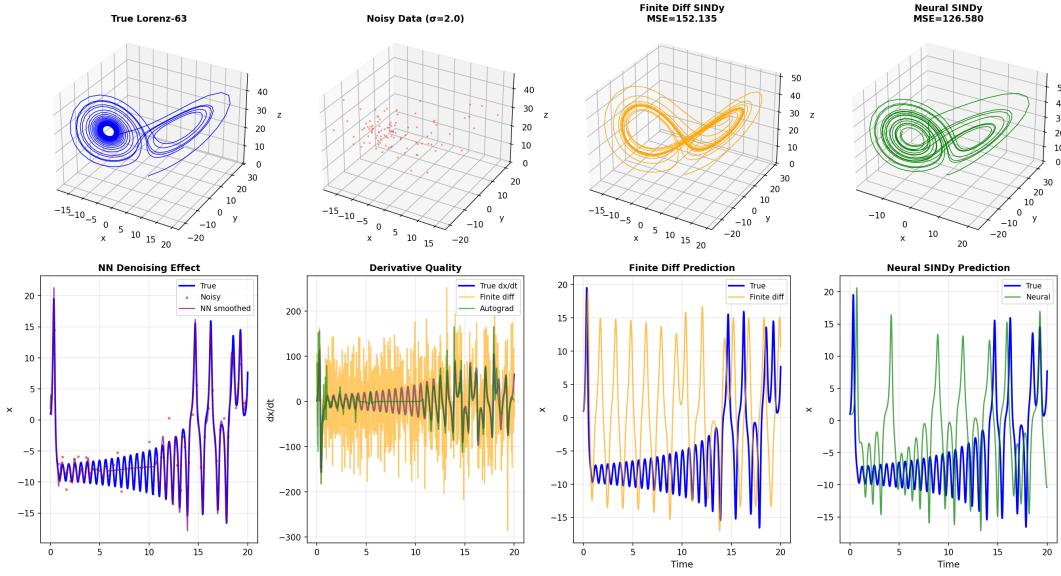


Figure 19.8: Neural SINDy: NN trajectory smoothing followed by autograd derivatives improves derivative quality and stabilizes sparse discovery in high-noise regimes.

- derivatives are poorly resolved,
- one wants simultaneously a good predictive emulator and an interpretable model.

In this sense, Neural SINDy does not compete with classical system identification, but *combines* modern neural approximation with interpretable sparse regression. The resulting workflow is a representative example of the broader trend in scientific machine learning:

classical physics/statistics + neural representation learning \Rightarrow robust hybrid methods.

19.3 Learning the Force Term: Neural RHS and Hybrid Dynamics

19.3.1 From discovery to emulation

A third viewpoint is to stop aiming for explicit equation discovery and instead learn a *predictive surrogate model*. Instead of constructing an interpretable sparse library (as in SINDy), we learn the full right-hand side (RHS) directly:

$$\dot{\mathbf{x}} = \mathbf{f}_\theta(\mathbf{x}),$$

where \mathbf{f}_θ is represented by a neural network.

This is the most flexible option and can represent complicated nonlinear dynamics. The trade-off is that the learned mechanism is typically less interpretable than sparse equation discovery. Nevertheless, for many applications (forecasting, emulation, fast surrogates) predictive skill is the primary objective, and neural RHS models are highly effective.

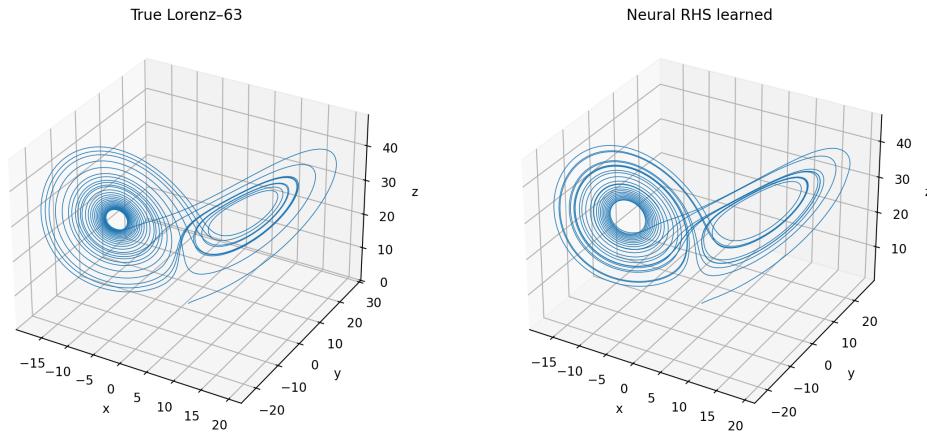


Figure 19.9: Learning the right-hand side (RHS) of a dynamical system: a neural network approximates $f(x)$ in $\dot{x} = f(x)$.

19.3.2 Hybrid modeling: known physics + learned closure

Many scientific systems contain partially known physics. A common approach is **hybrid modeling**:

$$\dot{x} = f(x) + g_\theta(x),$$

where f encodes known components (e.g. conservation laws, linear dynamics, or well-understood processes) and g_θ learns unknown forcing or closure terms.

This combines:

- physical structure (stability and plausibility),
- learned flexibility (representing missing processes).

Such hybrid approaches are often a good compromise: the neural network does not have to learn everything from scratch, but focuses on the part where the model is uncertain. This is a key principle in scientific machine learning: use neural networks where they provide value, but keep physically meaningful structure where possible.

19.3.3 Training paradigms: derivatives vs. rollouts

Two typical training paradigms are:

- **Derivative matching (local learning):** train on pairs (x, \dot{x}) , minimizing

$$\mathcal{L} = \mathbb{E} \|f_\theta(x) - \dot{x}\|^2.$$

This makes RHS learning a supervised regression problem.

- **Trajectory matching (global learning):** train by integrating the learned system forward in time and matching multi-step rollouts. This directly optimizes forecast skill but is computationally more expensive and can be less stable in chaotic systems.

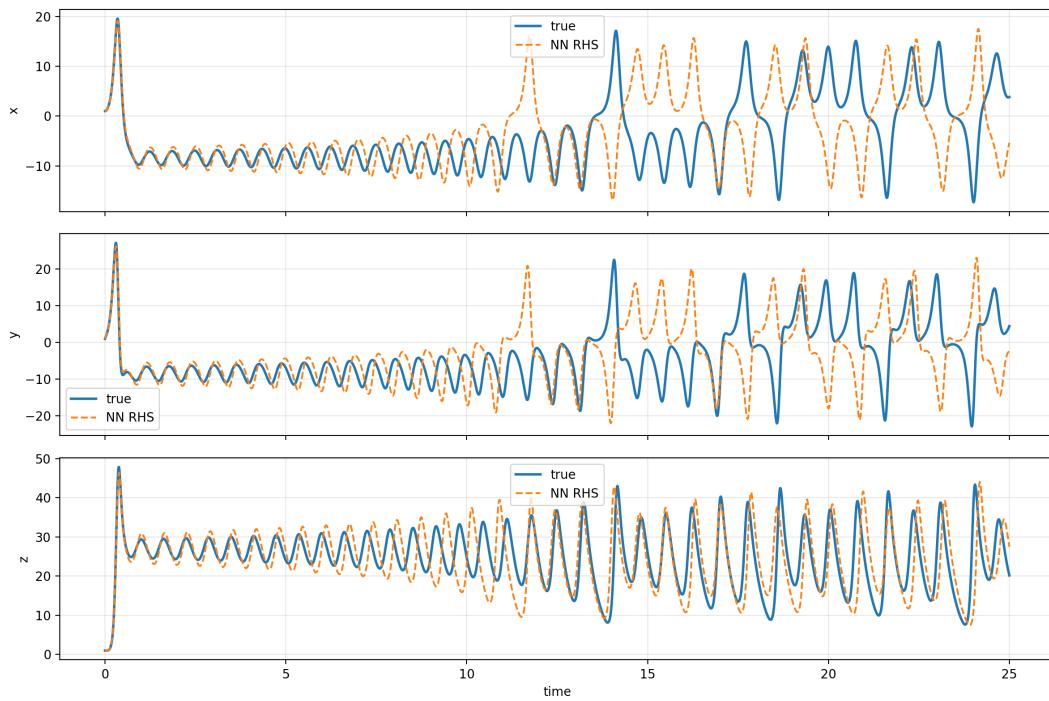


Figure 19.10: Hybrid modeling: combine known physics with a learned correction term (closure / forcing), improving realism while preserving structure.

A crucial practical point (illustrated by Lorenz-63) is:

- **short-term trajectory skill can be excellent,**
- **long-term divergence is unavoidable** in chaotic systems,
- but **climatological structure (attractor statistics)** can still be captured.

19.3.4 A central issue: sampling the state space

A major conceptual difference between classical equation discovery and neural RHS learning is the role of **sampling**.

If we train only along one observed trajectory, then the model learns the vector field only on (or near) the attractor region that was visited by the system. This can be sufficient for short-term forecasts near that attractor, but it is fragile: small errors may push the solution outside the sampled region, where the learned RHS is poorly constrained.

Therefore, for robust neural emulation, it is often beneficial to train the RHS on a much wider region of the state space:

$$\mathbf{x} \sim \text{broad distribution in state space}, \quad \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) \text{ (true RHS).}$$

This yields a globally trained emulator of the vector field, not only a trajectory-trained one.

In Lorenz-63, we can sample a bounding box around the attractor (with some margin) and evaluate the true RHS there. Figure 19.11 shows the key geometry: the true trajectory occupies only a small part of the sampled 3D region, but the broader sampling stabilizes the learned model.

Visualize full state-space sampling vs. Lorenz-63 attractor

```

1 # =====
2 # Visualize state-space sampling + true Lorenz-63 trajectory
3 # Saves: rhs_learning_space_sampling.png
4 # =====
5 import numpy as np
6 import matplotlib.pyplot as plt
7 from mpl_toolkits.mplot3d import Axes3D
8 from scipy.integrate import solve_ivp
9
10 sigma, rho, beta = 10.0, 28.0, 8.0/3.0
11
12 def lorenz63(t, X):
13     x, y, z = X
14     return [
15         sigma * (y - x),
16         x * (rho - z) - y,
17         x * y - beta * z
18     ]
19
20 # true trajectory
21 x0 = [1.0, 1.0, 1.0]
22 t_span = (0.0, 25.0)
23 t_eval = np.linspace(*t_span, 6000)
24 sol = solve_ivp(lorenz63, t_span, x0, t_eval=t_eval, rtol=1e-10)
25 X_true = sol.y.T
26
27 # state-space sampling
28 np.random.seed(0)
29 N_samples = 60000
30 x_range = [-25, 25]
31 y_range = [-35, 35]
32 z_range = [0, 50]
33
34 X_space = np.column_stack([
35     np.random.uniform(*x_range, size=N_samples),
36     np.random.uniform(*y_range, size=N_samples),
37     np.random.uniform(*z_range, size=N_samples),
38 ])
39
40 # subsample for plotting
41 idx = np.random.choice(N_samples, size=6000, replace=False)
42 Xs = X_space[idx]
43

```

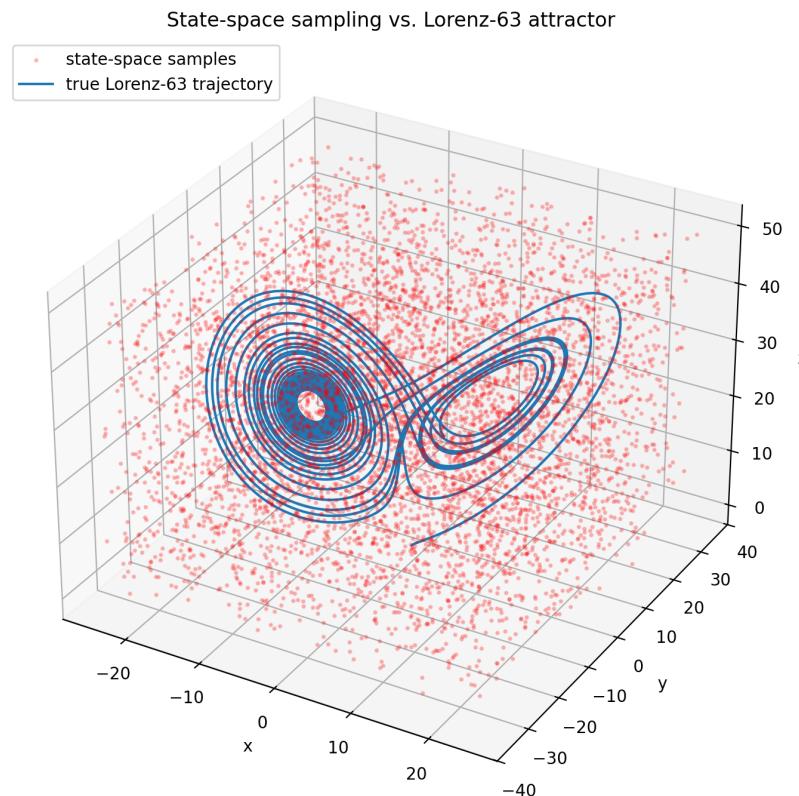


Figure 19.11: Sampling strategy in state space for training neural RHS models: the Lorenz–63 trajectory occupies only a subset of the sampled 3D domain. Training on a broader region improves robustness of the learned vector field.

```

44 # plot
45 fig = plt.figure(figsize=(9, 7))
46 ax = fig.add_subplot(111, projection="3d")
47 ax.scatter(Xs[:, 0], Xs[:, 1], Xs[:, 2],
48             s=3, alpha=0.20, color="red",
49             label="state-space samples")
50 ax.plot(X_true[:, 0], X_true[:, 1], X_true[:, 2],
51           color="tab:blue", linewidth=1.5,
52           label="true Lorenz-63 trajectory")
53
54 ax.set_xlabel("x"); ax.set_ylabel("y"); ax.set_zlabel("z")
55 ax.set_title("State-space sampling vs. Lorenz-63 attractor")
56 ax.legend(loc="upper left")
57 plt.tight_layout()
58 plt.savefig("rhs_learning_space_sampling.png", dpi=200)
59 plt.show()

```

19.3.5 Neural RHS learning on a full state-space sample

The following experiment illustrates **neural RHS learning with full state-space sampling**. We generate many random points in a 3D bounding box that covers the Lorenz–63 attractor, compute the true RHS at these points, and train a neural network:

$$f_\theta : \mathbb{R}^3 \rightarrow \mathbb{R}^3, \quad f_\theta(\mathbf{x}) \approx f(\mathbf{x}).$$

This training procedure yields a global approximation of the Lorenz vector field. After training, we integrate the learned system and compare the resulting trajectory to the true one.

Neural RHS learning on full state space (Lorenz-63)

```

1 # =====
2 # Neural RHS learning with FULL STATE-SPACE SAMPLING
3 # Lorenz-63 vector field learned in R^3, not only on trajectory
4 # Saves:
5 #   (a) rhs_learning_space_1.png  (3D trajectory)
6 #   (b) rhs_learning_space_2.png  (time series x,y,z)
7 # =====
8
9 import torch
10 import torch.nn as nn
11 import numpy as np
12 from scipy.integrate import solve_ivp
13 import matplotlib.pyplot as plt
14 from mpl_toolkits.mplot3d import Axes3D
15
16 torch.manual_seed(0)
17 np.random.seed(0)
18
19 # device
20 if torch.cuda.is_available():
21     device = torch.device("cuda")
22 elif torch.backends.mps.is_available():
23     device = torch.device("mps")
24 else:
25     device = torch.device("cpu")
26
27 # true Lorenz-63 RHS
28 sigma, rho, beta = 10.0, 28.0, 8.0/3.0
29
30 def lorenz63_rhs(X):
31     x, y, z = X[:,0], X[:,1], X[:,2]
32     dx = sigma * (y - x)
33     dy = x * (rho - z) - y
34     dz = x * y - beta * z
35     return np.stack([dx, dy, dz], axis=1)
36
37 # sample full state space

```

```

38 N_samples = 60000
39 x_range = [-25, 25]
40 y_range = [-35, 35]
41 z_range = [0, 50]
42
43 X_space = np.column_stack([
44     np.random.uniform(*x_range, size=N_samples),
45     np.random.uniform(*y_range, size=N_samples),
46     np.random.uniform(*z_range, size=N_samples),
47 ])
48 dXdt_space = lorenz63_rhs(X_space)
49
50 X_t = torch.tensor(X_space, dtype=torch.float32, device=device)
51 dXdt_t = torch.tensor(dXdt_space, dtype=torch.float32, device=device)
52
53 # neural RHS model
54 class RHSNet(nn.Module):
55     def __init__(self, width=128, depth=4):
56         super().__init__()
57         layers = [nn.Linear(3, width), nn.Tanh()]
58         for _ in range(depth - 1):
59             layers += [nn.Linear(width, width), nn.Tanh()]
60         layers += [nn.Linear(width, 3)]
61         self.net = nn.Sequential(*layers)
62         for m in self.net:
63             if isinstance(m, nn.Linear):
64                 nn.init.xavier_normal_(m.weight)
65                 nn.init.zeros_(m.bias)
66
67     def forward(self, x):
68         return self.net(x)
69
70 model = RHSNet().to(device)
71 opt = torch.optim.Adam(model.parameters(), lr=1e-3)
72
73 # training
74 epochs = 25000
75 batch_size = 4096
76 for ep in range(1, epochs + 1):
77     idx = torch.randint(0, N_samples, (batch_size,))
78     Xb = X_t[idx]
79     dXb = dXdt_t[idx]
80     opt.zero_grad()
81     pred = model(Xb)
82     loss = torch.mean((pred - dXb)**2)
83     loss.backward()
84     opt.step()
85
86 # integrate learned RHS

```

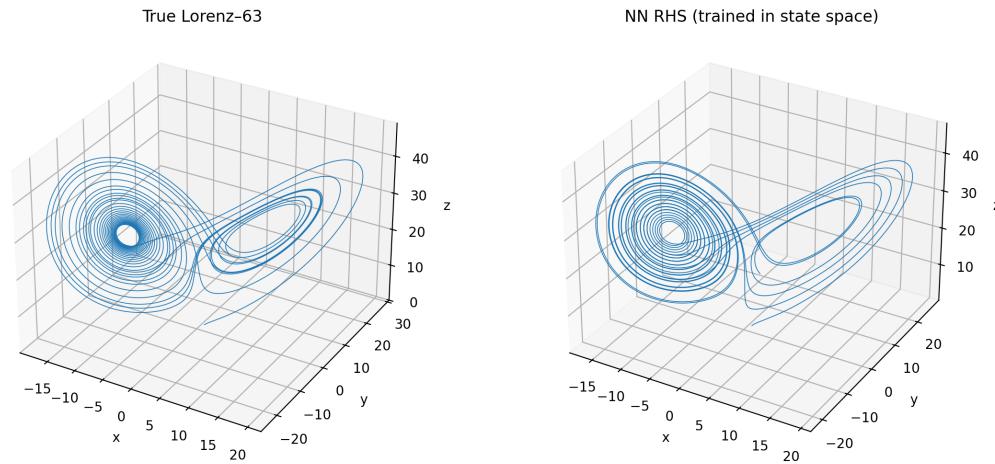


Figure 19.12: Neural RHS learning on full state space: comparison of 3D trajectories (true Lorenz-63 vs. learned RHS integrated forward).

```

87 def rhs_nn(t, X):
88     Xt = torch.tensor(X, dtype=torch.float32, device=device).unsqueeze(0)
89     with torch.no_grad():
90         dX = model(Xt).cpu().numpy()[0]
91     return dX.tolist()
92
93 x0 = [1.0, 1.0, 1.0]
94 t_span = (0.0, 25.0)
95 t_eval = np.linspace(*t_span, 5000)
96
97 sol_true = solve_ivp(
98     lambda t, X: lorenz63_rhs(np.array(X)[None, :])[0],
99     t_span, x0, t_eval=t_eval, rtol=1e-10
100 )
101 sol_nn = solve_ivp(rhs_nn, t_span, x0, t_eval=t_eval, rtol=1e-10)
102
103 # save figures rhs_learning_space_1.png and rhs_learning_space_2.png

```

The resulting comparisons are shown in Figures 19.12 and 19.13. We emphasize again a balanced interpretation:

- For chaotic systems, a trajectory match will not remain perfect for long times. Divergence is expected.
- Nevertheless, a well-trained neural RHS can capture the correct qualitative attractor structure, reproduce realistic short-term evolution, and represent the vector field in a broad region.

Take-home message: Neural RHS learning is a powerful emulation approach, especially when combined with adequate state-space sampling and with hybrid constraints. It complements classical methods such as SINDy: while SINDy aims for sparse interpretable equations, neural RHS learning aims for robust predictive surrogates of complex dynamics.

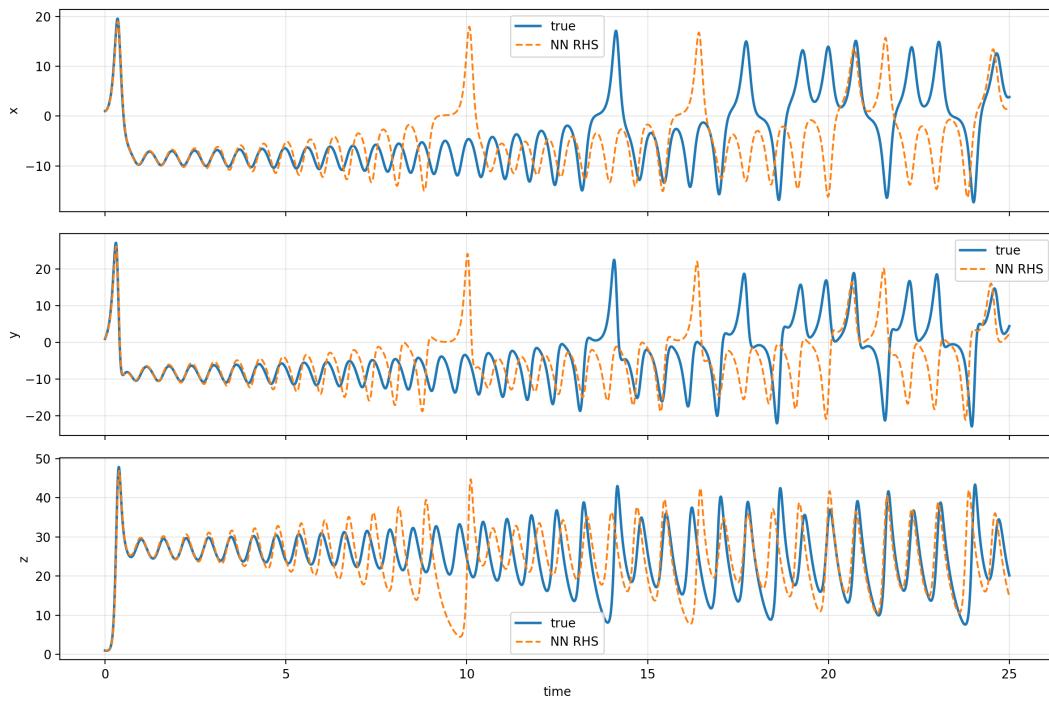


Figure 19.13: Neural RHS learning on full state space: time series comparison for $x(t)$, $y(t)$, and $z(t)$ (true vs. learned RHS model).

19.4 Physics constraints in neural emulators

We emphasize a crucial lesson for dynamical emulation: *neural accuracy alone is not sufficient*. Even if a neural network achieves excellent one-step mean-squared error (MSE), it may still generate physically implausible solutions after long rollouts, because small errors accumulate and push the state into regions where the learned dynamics violates invariants.

A canonical testbed is **1D periodic advection** on a ring:

$$\partial_t u + c \partial_x u = 0, \quad u(0, t) = u(1, t).$$

In the continuous system, the solution is a pure translation and important global properties are preserved.

Invariant: mass conservation. For periodic advection, the spatial integral

$$M(t) = \int_0^1 u(x, t) dx$$

remains constant in time. In discrete form on a uniform grid this corresponds to

$$M^n \approx \Delta x \sum_{i=1}^N u_i^n,$$

and conservative numerical schemes preserve this quantity.

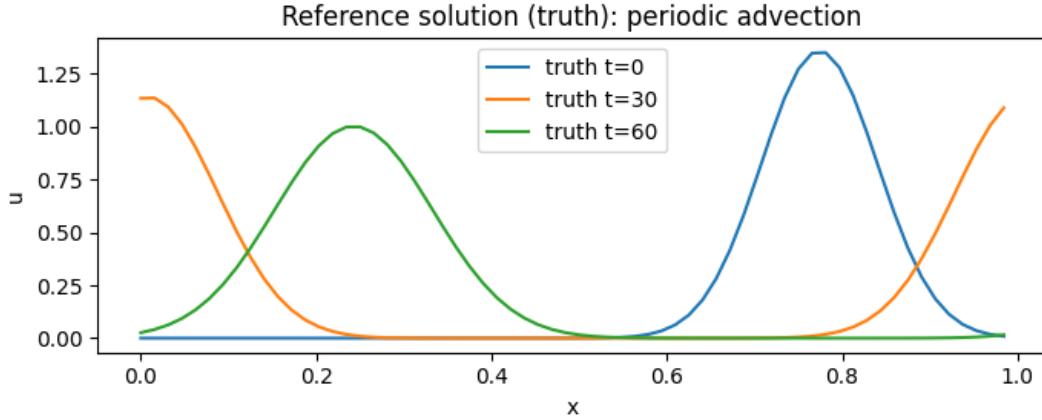


Figure 19.14: Reference solution for periodic advection (truth): the initial pattern is transported without deformation.

As reference physics, we use a conservative upwind finite-volume time step:

$$u_i^{n+1} = u_i^n - \text{CFL} (u_i^n - u_{i-1}^n), \quad \text{CFL} = c\Delta t/\Delta x,$$

which is mass-conservative by construction.

19.4.1 CNN one-step emulators: free vs. conservative by construction

We now emulate the one-step map

$$u^n \mapsto u^{n+1}$$

with a neural network. A CNN is a natural architecture for this problem because advection corresponds to translation-like local structure on the periodic ring.

A robust and physically meaningful design is to predict a **residual increment**

$$u^{n+1} = u^n + \Delta u_\theta(u^n).$$

This residual viewpoint is beneficial because it focuses learning on the local change.

Unconstrained CNN (free model). A standard CNN learns Δu_θ and is trained in MSE on one-step pairs. This yields strong one-step performance—but does not prevent slow drift in invariants.

Conservative CNN (constraint by construction). Mass conservation can be enforced *exactly* at every step by imposing

$$\sum_{i=1}^N \Delta u_i = 0.$$

A simple and effective implementation is a per-sample projection:

$$\Delta u \leftarrow \Delta u - \frac{1}{N} \sum_{i=1}^N \Delta u_i.$$

Then,

$$\sum_i u_i^{n+1} = \sum_i u_i^n$$

holds exactly (up to floating point rounding), independent of training quality.

This is an important conceptual point: **we do not merely penalize mass drift, we eliminate it by design.**

CNN emulator with exact mass conservation (by construction)

```

1  class CNNBase(nn.Module):
2      def __init__(self, kernel_size=5):
3          super().__init__()
4          pad = kernel_size // 2
5          self.net = nn.Sequential(
6              nn.Conv1d(1, 16, kernel_size, padding=pad, padding_mode="circular"),
7              nn.Tanh(),
8              nn.Conv1d(16, 16, kernel_size, padding=pad, padding_mode="circular"),
9              nn.Tanh(),
10             nn.Conv1d(16, 1, kernel_size, padding=pad, padding_mode="circular"),
11         )
12
13     def forward_du(self, u):
14         u1 = u.unsqueeze(1)                      # [B,1,N]
15         du = self.net(u1).squeeze(1)            # [B,N]
16         return du
17
18 class CNNFree(nn.Module):
19     def __init__(self, kernel_size=5):
20         super().__init__()
21         self.core = CNNBase(kernel_size)
22
23     def forward(self, u):
24         du = self.core.forward_du(u)
25         return u + du
26
27 class CNNConservative(nn.Module):
28     def __init__(self, kernel_size=5):
29         super().__init__()
30         self.core = CNNBase(kernel_size)
31
32     def forward(self, u):
33         du = self.core.forward_du(u)
34         du = du - du.mean(dim=-1, keepdim=True)  # enforce sum(du)=0
35         return u + du

```

Short-term snapshots are not enough. Both CNN variants can look visually excellent for short rollouts, and both can achieve a low one-step error. This is why mass-conservation issues are

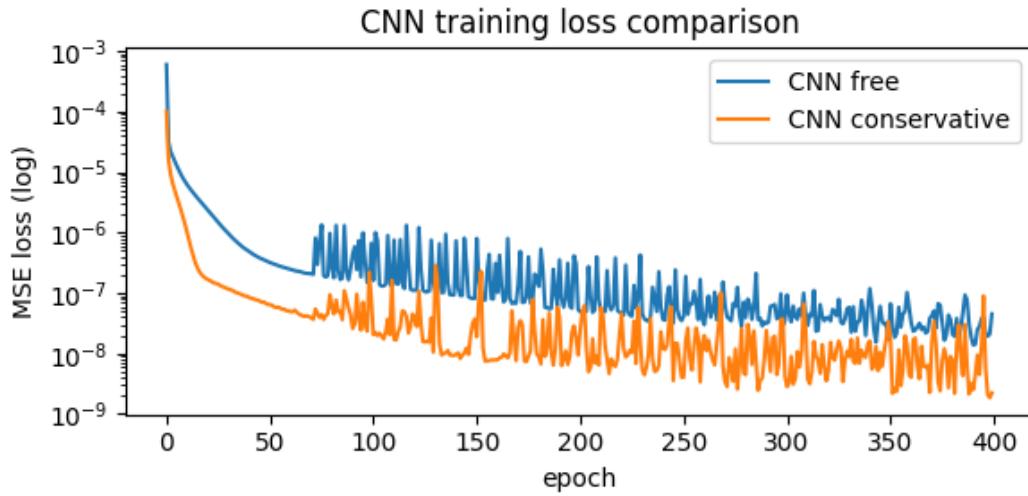


Figure 19.15: Training loss comparison for CNN one-step emulators: unconstrained vs. conservative-by-construction. The conservative projection does not prevent learning the one-step map.

frequently missed if we only evaluate a few steps.

19.4.2 Diagnostics: conservation and long-term stability

A physically meaningful diagnostic is to track invariant drift over time. For the discrete advection model, we monitor the mass

$$M^n = \Delta x \sum_i u_i^n$$

along a rollout.

The effect becomes even more pronounced in long rollouts: drift in invariants can slowly change the qualitative behaviour of the system, even when short-term predictions look excellent.

Generalization to new shapes. A further lesson is that good emulators must generalize beyond the training distribution. We therefore test the CNNs on qualitatively different initial conditions (Gaussian bump, top-hat, sine wave, two bumps). For each shape we compare initial condition and $t = 60$ state against truth.

Take-home messages.

- **Short-term accuracy is not a proxy for long-term physical validity.** One-step MSE can be low even when invariants slowly drift.
- **Hard constraints by construction are often superior to soft penalty terms.** A simple projection can guarantee exact conservation.
- **Physical constraints act as strong regularizers.** They improve robustness, generalization, and rollout stability.

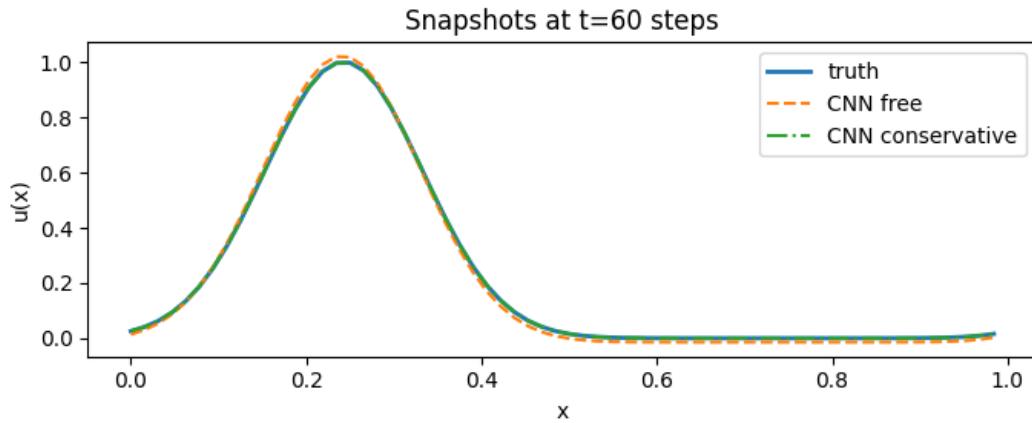


Figure 19.16: Rollout snapshots at selected time steps: even if the unconstrained CNN looks plausible in short horizons, physics violations can accumulate over long rollouts.

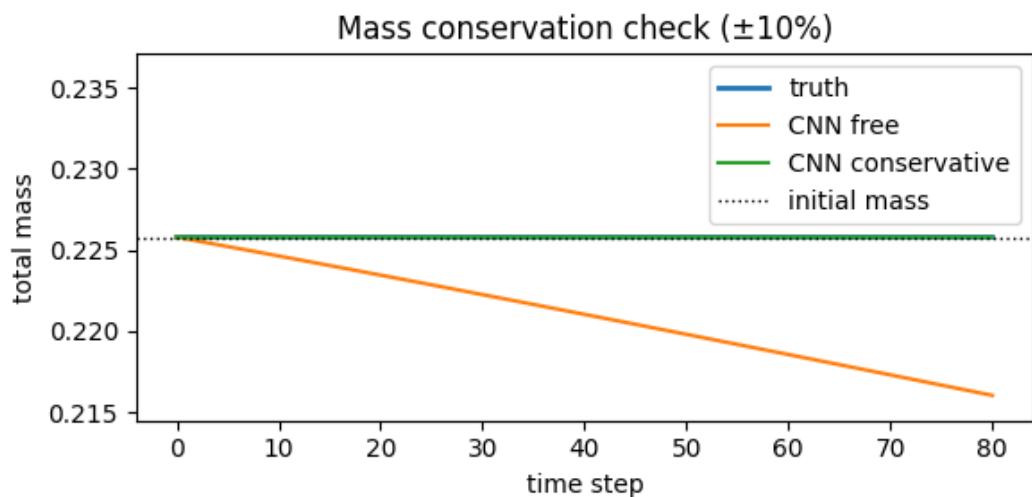


Figure 19.17: Mass conservation test: truth vs. CNN free vs. CNN conservative. The conservative CNN preserves mass exactly by construction, while the unconstrained CNN typically drifts.

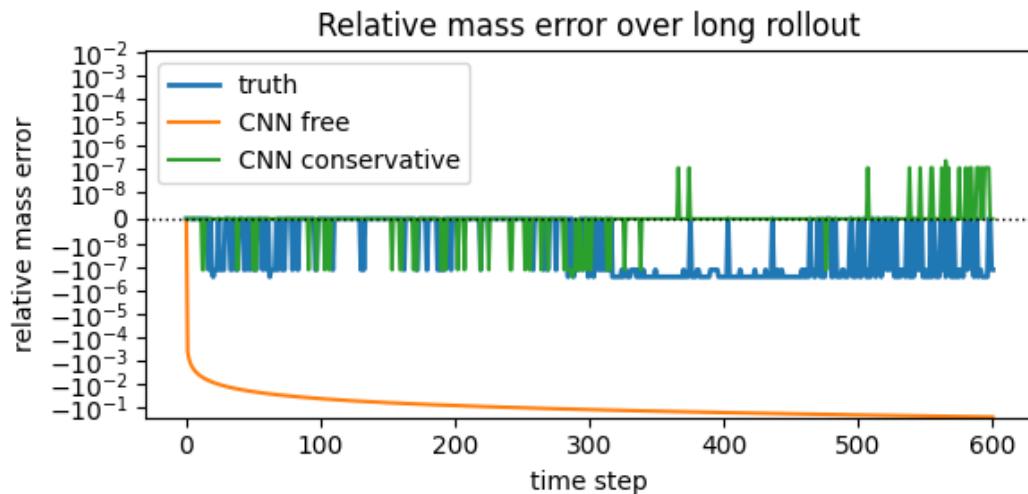


Figure 19.18: Long-rollout diagnostic: relative mass error over hundreds of steps. Conservative-by-construction models remain physically valid in invariants even in long horizons.

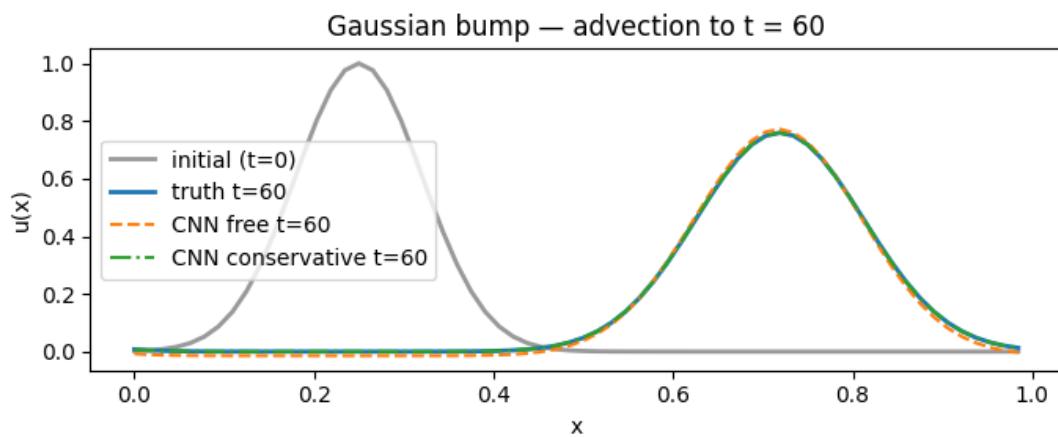


Figure 19.19: Generalization test (example 1): unseen initial condition and its advected state at $t = 60$. The conservative CNN stabilizes rollouts by preventing invariant drift.

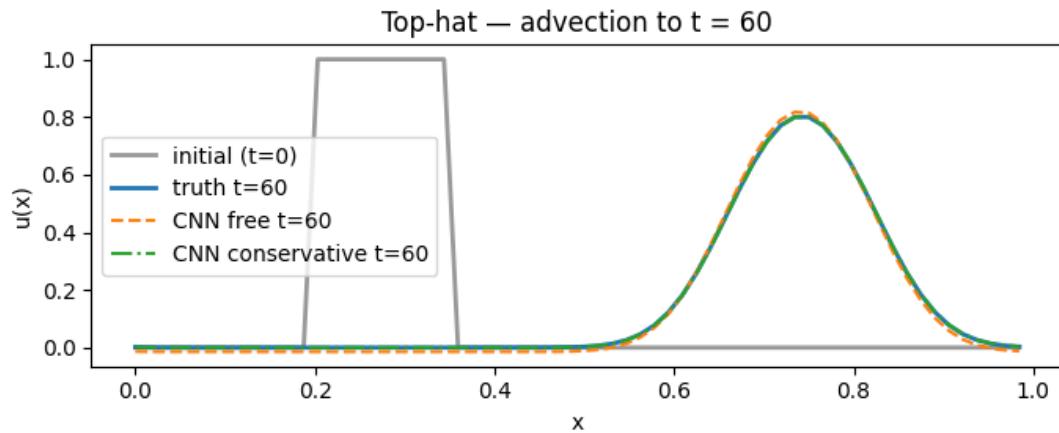


Figure 19.20: Generalization test (example 2): conservative-by-construction constraints improve robustness beyond the training distribution.

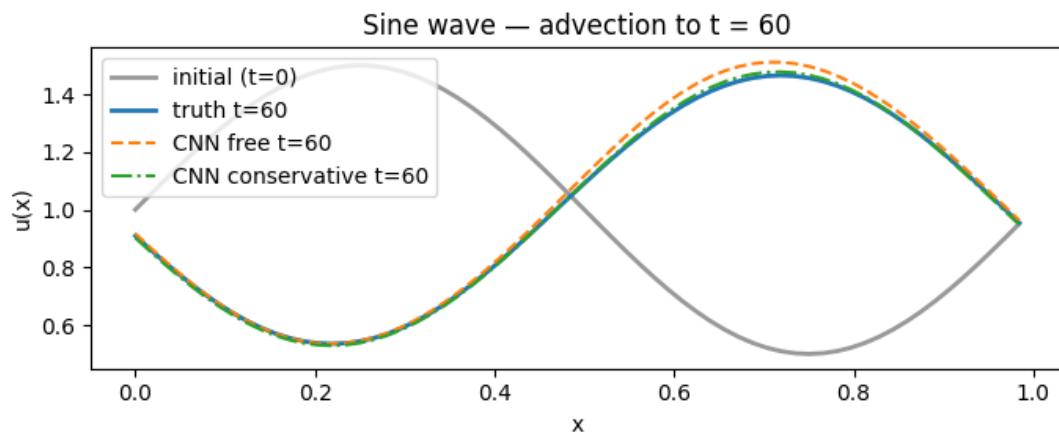


Figure 19.21: Generalization test (example 3): different shape families (e.g. sine-type initial conditions) reveal differences in long-term stability.

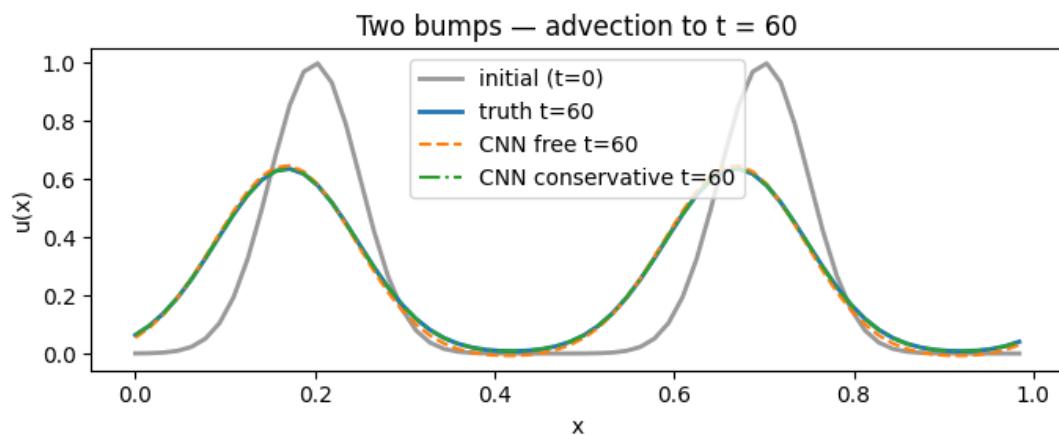


Figure 19.22: Generalization test (example 4): two-bump initial condition. Conservative constraints prevent systematic drift and preserve physically meaningful behaviour.

19.5 Causal Modeling with Neural Networks

19.5.1 Correlation versus causality

In many geoscientific applications we observe strong correlations between variables, but the underlying **causal mechanism** can still differ. This is particularly relevant in meteorology and climate, where synoptic forcing, advection, and feedback loops create complex dependence.

We note:

- **correlation does not imply causation,**
- causal direction can change with the dominant physical regime,
- hidden confounders can generate *spurious* statistical links.

Causal modeling does not ask only for prediction,

$$p(T | P),$$

but for interventions:

$$p(T | do(P = p)),$$

i.e. *what would happen if we could modify pressure in an experiment?* In Earth-system science, such interventional thinking is crucial for robust decision support and trustworthy extrapolation.

19.5.2 A meteorological toy model: Pressure, Temperature, and Forcing

To illustrate these concepts, we use a small dynamical system with pressure $P(t)$, temperature $T(t)$, and an external forcing $F(t)$. The forcing mimics large-scale advection and synoptic variability, e.g. a combination of slow waves and step-like frontal passages.

The synthetic dynamics are constructed such that the time series look plausible and exhibit strong dependence, while the *true causal structure* differs.

19.5.3 Two scenarios: similar correlations, different physics

Scenario 1: synoptic forcing + causal coupling $P \rightarrow T$.

Here pressure is primarily driven by external forcing (weather systems), while temperature responds to pressure anomalies, representing adiabatic compression / subsidence warming:

$$\frac{dP}{dt} = -\gamma_P(P - P_{eq}) + \beta_P F(t), \quad \frac{dT}{dt} = -\gamma_T(T - T_{eq}) + \alpha(P - P_{eq}).$$

Thus the causal chain is:

$$F \rightarrow P \rightarrow T,$$

and the true direct causal effect is $\alpha > 0$.

Scenario 2: common forcing only (confounded).

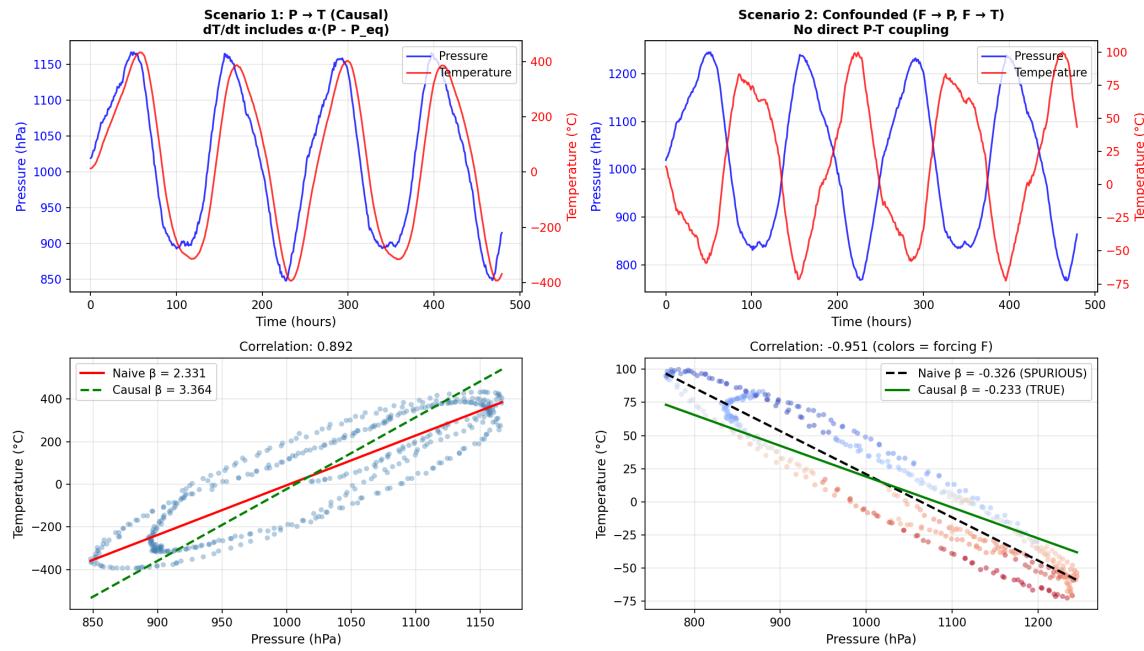


Figure 19.23: Two dynamical scenarios with different causal structure. Top: time series of pressure and temperature. Bottom: scatter plots. Scenario 1 shows a true $P \rightarrow T$ effect; Scenario 2 shows a spurious $P-T$ relation induced by common forcing $F(t)$.

Now both P and T are forced by $F(t)$, but there is no direct link $P \rightarrow T$:

$$\frac{dP}{dt} = -\gamma_P(P - P_{eq}) + \beta_P F(t), \quad \frac{dT}{dt} = -\gamma_T(T - T_{eq}) + \beta_T F(t), \quad \alpha = 0.$$

The true structure is:

$$F \rightarrow P, \quad F \rightarrow T, \quad P \not\rightarrow T.$$

Key point: Both scenarios can show a comparable correlation between P and T , but represent *fundamentally different* physical regimes: direct coupling versus confounding by advection / synoptic forcing.

19.5.4 Classical baselines: correlation and regression (valuable but limited)

Before applying AI-based methods, we explicitly promote classical statistical diagnostics as essential baselines:

- correlation analysis $\text{corr}(P, T)$,
- naive regression $T \sim P$,
- multiple regression $T \sim P + F$ when confounders are known.

In Scenario 1, naive regression recovers the sign and magnitude of the coupling because P truly impacts T . In Scenario 2, naive regression yields a *non-zero slope* even though the true causal effect is exactly zero — a textbook example of spurious regression due to confounding.

Multiple regression can correct this if the confounder F is observed and included, i.e. $T \sim P + F$. However, this approach still requires a correct *a priori* decision about which variables are confounders, and it does not directly discover causal structure.

Balanced view: Classical methods are indispensable and often sufficient, but they can fail systematically when confounding is hidden or the causal graph is unknown. This motivates the use of causal discovery methods.

19.5.5 Neural causal discovery: remove autocorrelation, learn cross-effects

Time series data exhibit strong autocorrelation: each variable is often well-predicted by its own past. A causal discovery method must therefore separate:

- **self-dependence** (autocorrelation),
- **cross-variable effects** (causal influences).

A simple and effective preprocessing step is to remove the dominant self-dependence for each variable by predicting:

$$X_i(t) \approx a_i X_i(t-1),$$

and forming the residual (*innovation*)

$$\varepsilon_i(t) = X_i(t) - a_i X_i(t-1).$$

Causal discovery then targets cross-variable prediction:

$$\varepsilon(t) \approx W \mathbf{X}(t-1),$$

where W is sparse and interpretable: W_{ij} measures the influence of variable j on variable i .

In the notebook implementation, sparsity is induced through Lasso regression, leading to a learned directed influence matrix and a corresponding causal graph. This demonstrates the main promise:

AI can learn causal structure — not just correlations.

19.5.6 PCMCI: statistical causal discovery for time series

Besides neural methods, we note that strong causal discovery tools also exist in **classical statistics**. A prominent approach for time series is PCMCI (Peter–Clark Momentary Conditional Independence), implemented for example in `tigramite`.

PCMCI combines:

- conditional independence testing (e.g. partial correlation),
- causal graph learning with time-lag structure,
- explicit significance control for spurious edges.

In the notebook experiment, PCMCI correctly separates the regimes:

- Scenario 1: detects $F(t - \tau) \rightarrow P(t)$ and $P(t - \tau) \rightarrow T(t)$,
- Scenario 2: detects $F(t - \tau) \rightarrow P(t)$ and $F(t - \tau) \rightarrow T(t)$, but no direct $P \rightarrow T$.

This makes PCMCI a powerful benchmark method: it is transparent, statistically grounded, and complements AI approaches well.

19.5.7 Take-home messages for Earth-system applications

Causal modeling is not only a philosophical extension of statistics, but a practical requirement when models are used for intervention analysis, policy, or impact assessment.

Main messages:

- **Diagnostics first:** classical correlation and regression remain essential baselines.
- **Confounding is common:** spurious links can arise naturally from shared forcing.
- **AI helps:** neural methods can extract sparse directed influence structures once autocorrelation is removed.
- **Statistics still matters:** PCMCI provides a rigorous, interpretable benchmark.

In practice, robust workflows combine physics knowledge, statistical causal discovery, and modern machine learning to obtain *trustworthy and interpretable* causal conclusions from complex Earth-system time series.

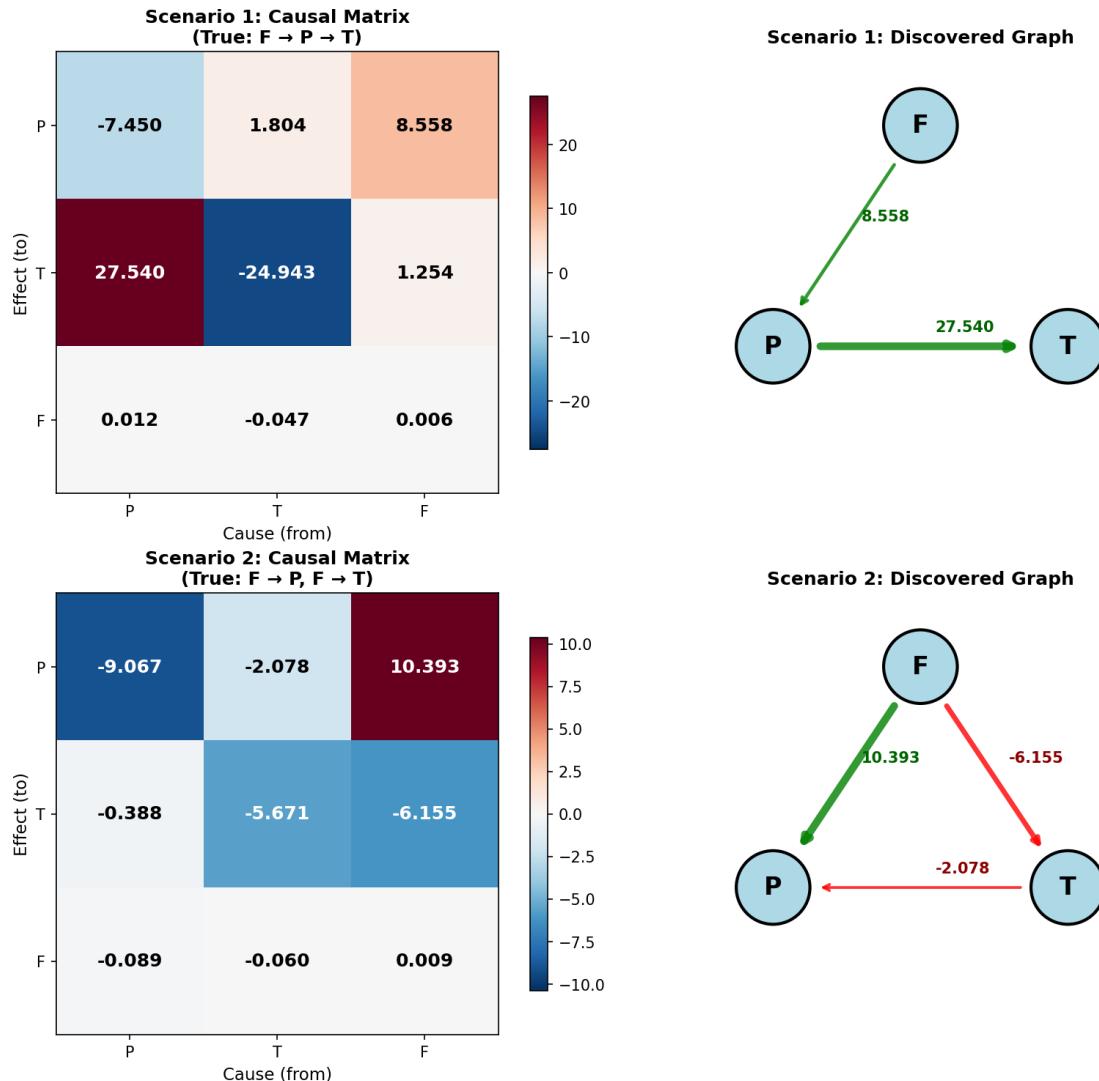


Figure 19.24: Neural causal discovery for both scenarios. Left: learned sparse influence matrix W after removing autocorrelation. Right: corresponding directed graph. Scenario 1 correctly yields $F \rightarrow P$ and $P \rightarrow T$; Scenario 2 yields $F \rightarrow P$ and $F \rightarrow T$ with a near-zero $P \rightarrow T$ link.

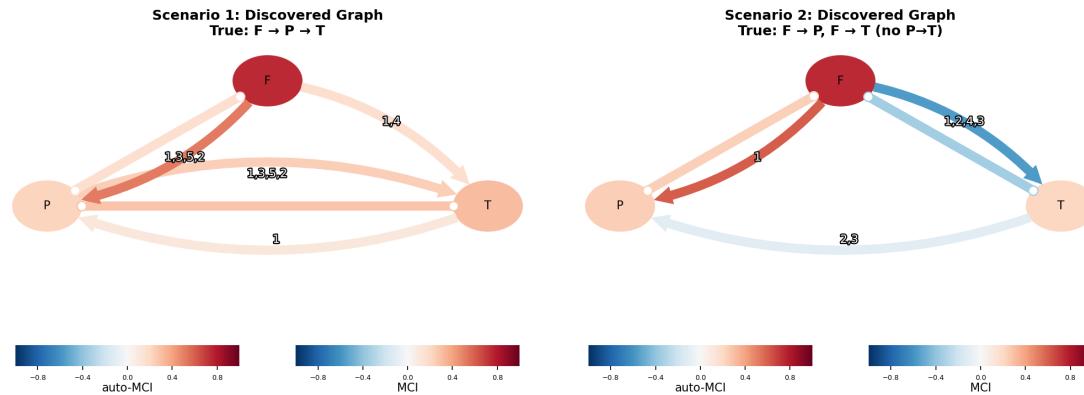


Figure 19.25: PCMCI causal discovery graphs for both scenarios. PCMCI identifies time-lagged directed links and correctly distinguishes causal coupling from confounding.

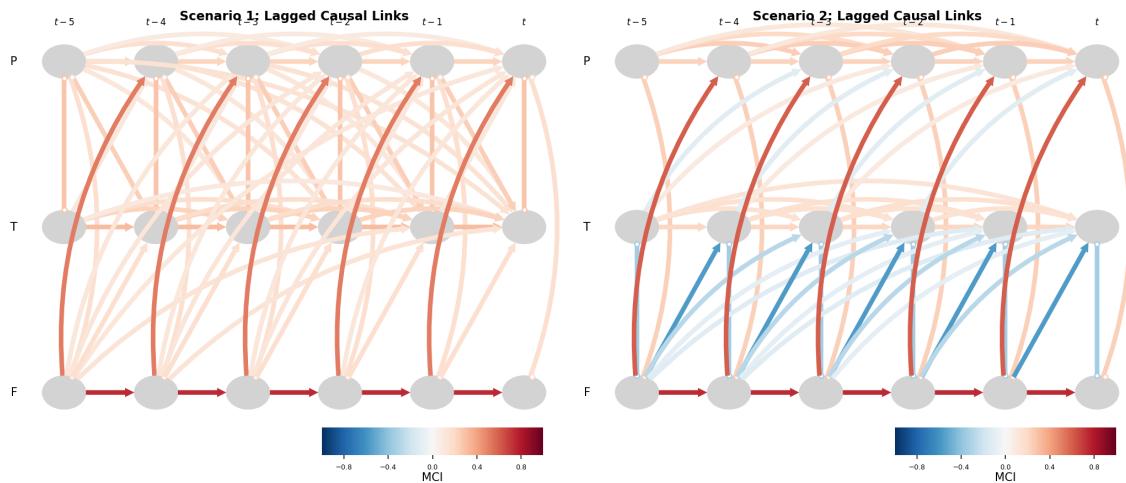


Figure 19.26: PCMCI lag-structure visualization: discovered time-lagged causal dependencies for both scenarios.

Chapter 20

Learning from Observations Only

20.1 Obs-to-Obs Learning on a 2D Toy Atmosphere

20.1.1 Motivation and goal

In this section we construct a deliberately simple, fully controlled 2D dynamical system as a testbed for learning and forecasting directly in *observation space*. The core variable is a tracer / heating-like scalar field $\phi(x, z, t)$ evolving on a rectangular domain $(x, z) \in [0, L_x] \times [0, L_z]$. The model is designed such that transport structures are visually plausible, numerically stable, and rich enough to create meaningful learning tasks.

The key idea is to generate *two different types of observations* from the same underlying field:

- **Radiosonde-like observations (RS):** sparse vertical profiles at a few selected columns.
- **Satellite-like observations (SAT):** vertically weighted integrated measurements for *every* column.

This creates a typical situation from atmospheric science: we have sparse but detailed profile information (RS), and dense but integrated information (SAT).

The machine learning target in this lecture is an explicit one-step forecast operator in observation space. Given the current observations (y_t^{sat}, y_t^{rs}) , we train a neural network to predict the next-step observations:

$$(y_t^{sat}, y_t^{rs}) \mapsto (y_{t+1}^{sat}, y_{t+1}^{rs}).$$

A particularly useful aspect of the setup is that, even though the learning takes place in observation space, we can still reconstruct a *state-like field* at time $t + 1$: we query the trained model to generate RS-like profiles at *all* horizontal positions, not only at the sparse radiosonde columns. This turns predicted observations into a spatially dense proxy state.

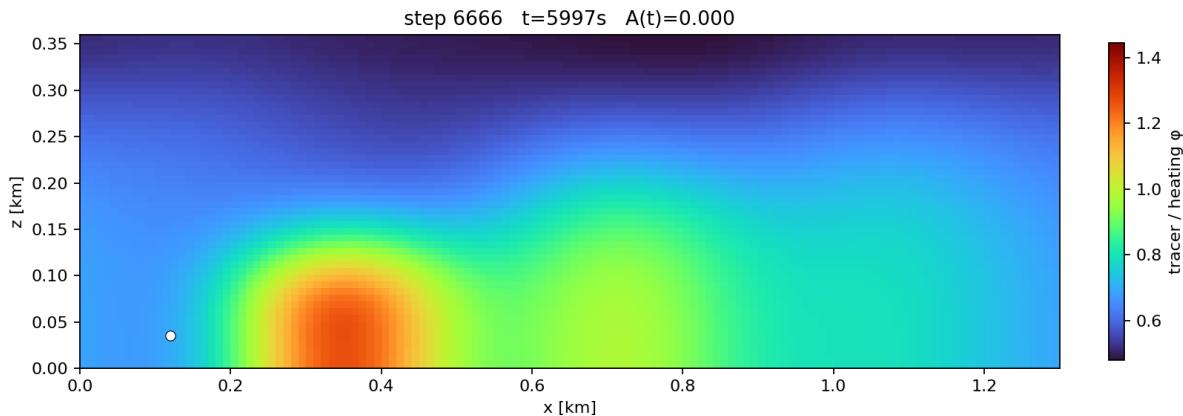


Figure 20.1: Simulation of some atmospheric dynamics through a toy example.

20.1.2 Toy dynamics: advection–diffusion with forcing and damping

The tracer/heating field $\phi(x, z, t)$ is evolved by an advection–diffusion equation with localized source forcing and stabilizing damping:

$$\frac{\partial \phi}{\partial t} = -u \frac{\partial \phi}{\partial x} - w \frac{\partial \phi}{\partial z} + K \nabla^2 \phi + A(t) S(x, z) - \lambda \phi - \lambda_{\text{top}}(z) \phi.$$

The terms are chosen for pedagogical clarity and robust numerical behavior:

- The mean wind field (u, w) is fixed and points *to the right and upward*. This produces clearly visible transport and coherent motion of patterns.
- Diffusion with constant coefficient K prevents filamentation and stabilizes the numerical solution.
- The spatial source $S(x, z)$ represents a localized heating region (bottom-left corner) that creates disturbances which are then transported.
- A pulsed modulation $A(t)$ produces an intermittent forcing signal, generating distinct tracer “stripes” that are easy to interpret in figures and time series.
- Linear damping with rate λ and an additional top sponge $\lambda_{\text{top}}(z)$ prevent indefinite growth and enforce a bounded statistical steady state.

Overall, the system is designed to converge to an equilibrium regime in which transport continuously redistributes tracer content while injection and loss stay approximately balanced. This regime is ideal for learning time transitions, because the system remains active but does not drift or explode.

20.1.3 Boundary conditions: periodic transport in x

To obtain a clean cyclic flow, the model uses **periodic boundary conditions in the horizontal direction**. Structures that leave the domain on the right re-enter from the left. This design choice is crucial: it avoids complicated outflow behavior, creates long-lived tracer trains, and yields recurring patterns that support learning.

In the vertical direction, reflective or stabilizing boundary treatment is used, together with the top sponge. In combination, these boundary conditions generate a clean, repeatable “toy atmosphere” with visually plausible dynamics and minimal numerical artifacts.

20.1.4 Time stepping: RK4 implementation

Time integration is performed using a standard explicit fourth-order Runge–Kutta scheme (RK4). We define the right-hand side operator

$$\text{RHS}(\phi, t) = -u \partial_x \phi - w \partial_z \phi + K \nabla^2 \phi + A(t)S(x, z) - (\lambda + \lambda_{\text{top}})\phi,$$

and compute one time step via

$$\begin{aligned} k_1 &= \text{RHS}(\phi, t), \\ k_2 &= \text{RHS}(\phi + \frac{1}{2}\Delta t k_1, t + \frac{1}{2}\Delta t), \\ k_3 &= \text{RHS}(\phi + \frac{1}{2}\Delta t k_2, t + \frac{1}{2}\Delta t), \\ k_4 &= \text{RHS}(\phi + \Delta t k_3, t + \Delta t), \\ \phi_{\text{next}} &= \phi + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

This provides a robust and accurate stepping method for the chosen toy dynamics.

For lecture and training purposes, the simulation also includes a snapshot logic: a set of n_{vis} time indices is selected uniformly in $[0, n_{\text{steps}}]$ and the corresponding 2D tracer fields are saved as numbered PNG figures. This provides consistent, reproducible visuals for documentation and presentation.

20.1.5 Budget monitoring: sanity checks for numerical stability

Even though the system is a toy model, we deliberately treat it like a miniature physical simulation and monitor simple global integrals. This prevents silent failure modes (wrong signs, unstable CFL choices, inconsistent damping, etc.) that could destroy the learning task without being obvious from a few images.

We define the following quantities:

$$C(t) = \int \phi dx dz, \quad I(t) = \int A(t)S dx dz, \quad L(t) = \int (\lambda + \lambda_{\text{top}})\phi dx dz.$$

Here $C(t)$ measures the total tracer content (a mass/energy proxy), $I(t)$ the instantaneous source injection, and $L(t)$ the instantaneous loss by damping and sponge.

A particularly robust diagnostic is the cumulative budget in discrete time,

$$\text{acc_in} = \sum_n I(t_n) \Delta t, \quad \text{acc_loss} = \sum_n L(t_n) \Delta t.$$

In a statistically steady regime, injected and lost content become balanced: the total content stabilizes, and the cumulative injection and cumulative loss evolve with similar slope. This confirms that the model has reached a controlled equilibrium that is well-suited for machine learning experiments.

Practical conclusion: budget diagnostics should always be part of the toy model. They act as a continuous stability certificate and dramatically reduce debugging time in later stages of the ML pipeline.

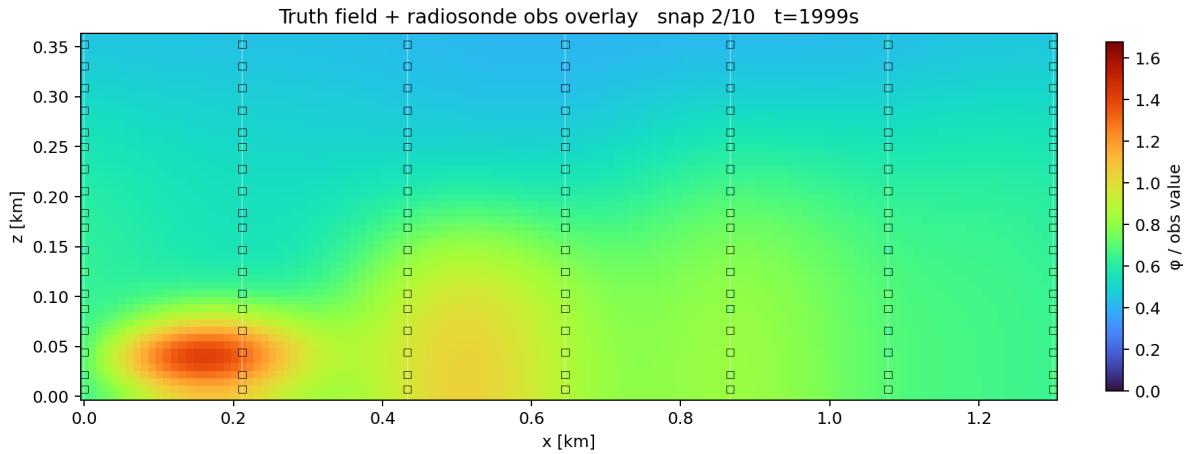


Figure 20.2: Toy atmosphere with radiosonde (RS) profiles at selected columns. RS provides sparse but vertically resolved information.

20.1.6 Observation operators: RS profiles and SAT integrated measurements

The purpose of this lecture is to learn a forecast step directly in *observation space*. This requires two complementary observation types:

- **RS (Radiosondes):** sparse vertical profiles of $\phi(x, z, t)$ at a few selected horizontal columns $x = x_i$.
- **SAT (Satellite-like):** vertically integrated, height-weighted measurements for every horizontal column x .

The key design idea is that both observation types are derived from the same underlying state field $\phi(x, z, t)$, but they represent very different views of the atmosphere: radiosondes give high vertical detail but only at a few locations, while satellite observations provide dense horizontal coverage but are vertically integrated. This setting is ideal for ML demonstrations: we can learn transitions in observation space, while still having access to the full truth field for analysis and validation.

Radiosonde operator (RS)

Let the model field be discretized on a grid $\phi_{i,k}(t)$ with horizontal index i (columns) and vertical index k (levels). Radiosonde observations are obtained by sampling a small set of columns $\mathcal{I}_{rs} = \{i_1, i_2, \dots, i_{n_{rs}}\}$:

$$y^{rs}(t) = \left\{ \phi_{i,k}(t) : i \in \mathcal{I}_{rs}, k = 1, \dots, N_z \right\}.$$

Thus each RS observation is a full vertical profile, but only at a sparse set of columns. In the lecture figures we visualize these profiles as vertical “sticks” within the 2D field.

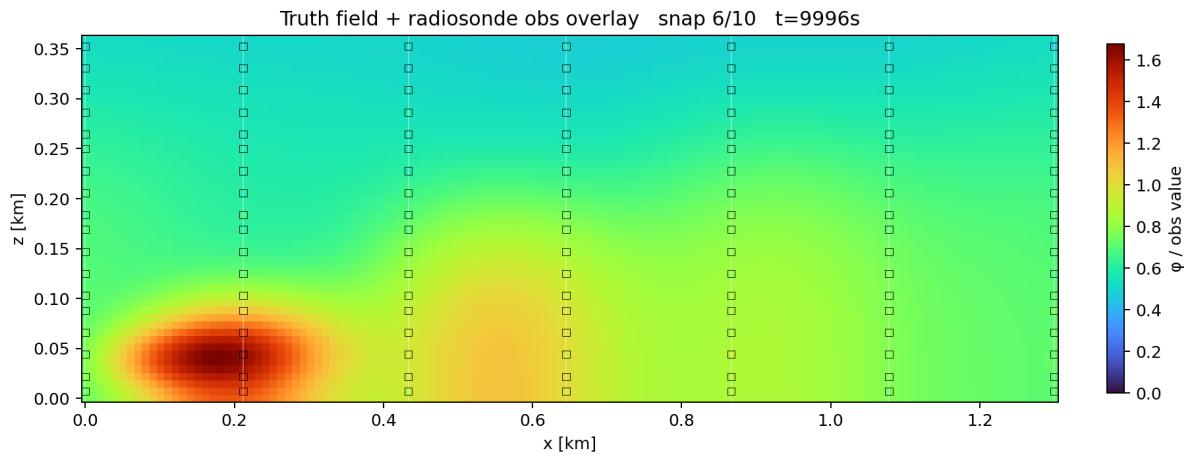


Figure 20.3: RS observation snapshots at a later time: transported patterns cross the RS columns, generating a clean training signal for one-step observation forecasting.

Satellite operator (SAT): vertical weighted integration

Satellite-like observations are defined as integrated measurements along the vertical direction. For each column x_i we compute a weighted integral:

$$y_i^{sat}(t) = \sum_{k=1}^{N_z} w_k \phi_{i,k}(t), \quad \sum_k w_k = 1, \quad w_k \geq 0.$$

The weights (w_k) approximate a vertical sensitivity kernel (a “weighting function”), similar to radiative transfer weighting functions used in atmospheric remote sensing. In our toy setup we design w_k as a smooth Gaussian-like kernel centered at a mid-level height. This ensures that the SAT observation sees coherent transported signals but does not trivially reproduce the full vertical structure.

Implementation in the notebook

The following code implements the RS and SAT observation operators. We keep the code explicit and simple because pedagogical clarity is more important than ultimate performance in this lecture.

Notebook: Observation operators (RS and SAT)

```

1 import numpy as np
2
3 def observe_rs(phi, rs_idx):
4     """
5         Radiosonde observations: extract vertical profiles at selected x-columns.
6
7         phi: 2D array, shape (Nx, Nz)
8         rs_idx: list/array of x-indices
9         returns: RS array, shape (n_rs, Nz)
10        """
11    return phi[rs_idx, :]
12

```

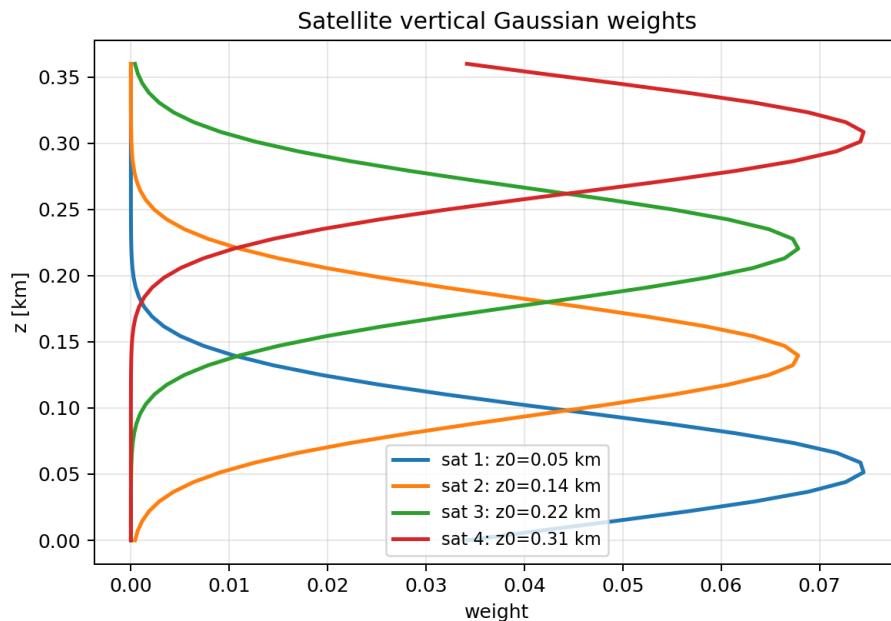


Figure 20.4: Vertical weighting function used for SAT observations. SAT provides dense horizontal coverage but only an integrated vertical view.

```

13 def observe_sat(phi, w_z):
14     """
15     Satellite observations: height-weighted vertical integral for each x-column.
16
17     phi: 2D array, shape (Nx, Nz)
18     w_z: 1D weights, shape (Nz,), sum(w_z)=1
19     returns: SAT array, shape (Nx,)
20     """
21     return phi @ w_z

```

The SAT weights can be generated as a normalized Gaussian-like kernel. The precise parameters are not critical; what matters is that the kernel selects a broad layer and remains positive and smooth.

Notebook: SAT weight construction

```

1 def sat_weights(Nz, z, z0=0.55, sigma=0.12):
2     """
3     Build smooth SAT weights over vertical coordinate z in [0,1].
4
5     Nz: number of vertical levels
6     z: 1D vertical coordinate, shape (Nz,)
7     z0: centre height (0..1)
8     sigma: width
9     """
10    w = np.exp(-0.5*((z - z0)/sigma)**2)

```

```

11     w = w / np.sum(w)
12     return w

```

Assembling ML training pairs in observation space

For the ML task we build supervised training pairs from the simulation trajectory. For each time step t we compute the observation vector

$$y_t = (y_t^{sat}, y_t^{rs})$$

and train a network to predict y_{t+1} from y_t .

The next code block illustrates the dataset assembly at a high level. It produces two arrays X and Y where each row corresponds to one time step:

$$X_n = y_t, \quad Y_n = y_{t+1}.$$

Notebook: Build training pairs

```

1 # phi_series: list/array of phi[t] fields, each phi[t] shape (Nx, Nz)
2 # rs_idx: radiosonde column indices
3 # w_z: SAT weights
4
5 X_list = []
6 Y_list = []
7
8 for t in range(len(phi_series) - 1):
9     phi_t    = phi_series[t]
10    phi_tp1 = phi_series[t+1]
11
12    ysat_t   = observe_sat(phi_t, w_z)      # (Nx, )
13    yrs_t    = observe_rs(phi_t, rs_idx)     # (n_rs, Nz)
14
15    ysat_tp1 = observe_sat(phi_tp1, w_z)
16    yrs_tp1 = observe_rs(phi_tp1, rs_idx)
17
18    # flatten into one observation vector per time
19    y_t    = np.concatenate([ysat_t.ravel(), yrs_t.ravel()])
20    y_tp1 = np.concatenate([ysat_tp1.ravel(), yrs_tp1.ravel()])
21
22    X_list.append(y_t)
23    Y_list.append(y_tp1)
24
25 X = np.stack(X_list, axis=0)
26 Y = np.stack(Y_list, axis=0)
27
28 print("X shape:", X.shape, "Y shape:", Y.shape)

```

Resulting structure: the network never sees ϕ directly during training. It learns the transition purely from the observation sequence $y_t \rightarrow y_{t+1}$.

Key message: Even without direct state access, observation-space learning can encode transport dynamics, because the observation operators preserve enough information to carry the predictive signal.

20.1.7 Learning observation-to-observation dynamics with neural networks

After defining the truth dynamics and observation operators, we now turn to the machine learning component: learning a one-step transition in observation space. The training target is a deterministic mapping

$$(y_t^{sat}, y_t^{rs}) \mapsto (y_{t+1}^{sat}, y_{t+1}^{rs}),$$

where SAT represents the full satellite curtain for all x , and RS represents a sparse set of radiosonde profile measurements at fixed (or variable) locations.

A central aspect of this lecture is that the model is trained on observations, but it can still be used to recover a state-like full field at the next time step by querying the RS head on the entire grid. In other words: we learn in observation space, but we can still obtain spatially dense predictions.

Loading the dataset (truth and observations)

The learning notebook starts by loading a dataset generated in the previous dynamics notebook. It contains truth snapshots, RS observations, SAT observations, and the metadata describing station indices and vertical weighting functions.

Notebook: load saved truth + observation arrays

```

1 import numpy as np
2
3 infile = "data_dyn_obs/dyn_truth_obs.npz"
4 d = np.load(infile)
5
6 xtrue = d["xtrue"]      # (time, z, x)
7 yrs   = d["yrs"]        # (time, station, vert)
8 ysat  = d["ysat"]       # (time, channel, x)
9 t_snap = d["t_snap"]
10
11 rs_ix = d["rs_ix"]     # (station,)
12 rs_iz = d["rs_iz"]     # (vert,)
13 sat_w = d["sat_w"]     # (channel, z)
14
15 x = d["x"]             # (x,)
16 z = d["z"]              # (z,)
17 d.close()
18
19 print("xtrue : ", xtrue.shape, "(time,z,x)")
20 print("yrs   : ", yrs.shape, "(time,station,vert)")
```

```
21 print("ysat : ", ysat.shape, "(time,channel,x)")
```

The key point is that the notebook works with a modest number of snapshots (e.g. $T_{snap} = 10$) but each snapshot contains dense 2D truth information. This is ideal for a controlled ML demonstration: we can train the model only on observations, but verify predictions against the full truth field.

Dataset switch: two RS input modes (A/B)

In a real observational setting the radiosonde network is fixed and sparse. However, for development and interpretability it is extremely useful to support two dataset modes:

- **Mode A (synthetic RS from truth):** RS input points are drawn randomly from the truth field $xtrue[t]$ at arbitrary (x, z) locations, and realistic noise is added. This produces strong geometric diversity and excellent generalization capability to arbitrary RS geometries.
- **Mode B (use only stored RS observations):** RS input points are taken strictly from the saved radiosonde network $yrs[t,:,:]$ at fixed rs_ix and rs_iz . This is the strict observational setting: the input uses only real observations at time t .

Mode B is closer to operational reality (fixed radiosonde stations). Mode A is a controlled training mode that mimics additional mobile or opportunistic observations (e.g. aircraft profiles), which are indeed available in real systems.

Notebook: dataset mode switch (A/B)

```
1 # USER SWITCH:
2 mode = "B"      # "A" = synthetic RS from truth, "B" = stored yrs[t] only
3 print("Dataset mode:", mode)
```

Normalization: a critical practical detail

SAT and RS observations live on different scales and have different statistics. Without explicit normalization, training becomes unstable and the network allocates capacity to trivial scale matching instead of learning dynamics.

We normalize SAT *per channel* over time and x , and we normalize truth/RS globally:

$$y_n^{sat} = \frac{y^{sat} - \mu_{sat}}{\sigma_{sat}}, \quad x_n = \frac{x - \mu_x}{\sigma_x}, \quad y_n^{rs} = \frac{y^{rs} - \mu_x}{\sigma_x}.$$

The network operates in normalized space. For plots and RMSE diagnostics we always de-normalize back to physical units.

Notebook: SAT per-channel normalization + truth/RS normalization

```
1 # SAT: per channel normalization over time and x
2 ysat_mu = ysat.mean(axis=(0, 2), keepdims=True).astype(np.float32)           # (1,
   nsat,1)
```

```

3 ysat_sd = (ysat.std(axis=(0, 2), keepdims=True) + 1e-8).astype(np.float32) # (1,
4 nsat,1)
4 ysat_n = ((ysat - ysat_mu) / ysat_sd).astype(np.float32)
5
6 # truth: global normalization (also for RS values)
7 xtrue_mu = float(xtrue.mean())
8 xtrue_sd = float(xtrue.std() + 1e-8)
9 xtrue_n = ((xtrue - xtrue_mu) / xtrue_sd).astype(np.float32)

```

Training data: predicting SAT(t+1) and RS(t+1) at query points

The learning task is formulated as a joint prediction problem. From time step t to $t + 1$ we predict two outputs:

- the full satellite curtain $\hat{y}_{t+1}^{sat}(c, x)$,
- a radiosonde profile at arbitrary query points $\hat{y}_{t+1}^{rs}(x_q, z_q)$.

The second output is crucial: it forces the model to produce *spatially meaningful* predictions instead of simply reproducing the fixed radiosonde network. For each training example we select a query column x_q and a set of heights $\{z_q\}$. The truth targets for these query points are taken from the truth field $xtrue[t + 1]$.

Implementation-wise this leads to a dataset returning

$$(y_t^{sat}, \mathcal{S}_t^{rs}, \mathcal{Q}^{rs}) \mapsto (y_{t+1}^{sat}, y_{t+1}^{rs}(\mathcal{Q}^{rs})),$$

where \mathcal{S}_t^{rs} is a set of RS input points and \mathcal{Q}^{rs} is a query set.

Notebook: build training pairs (SAT + RS queries)

```

1 class JointDataset(Dataset):
2     def __getitem__(self, idx):
3         t_cur = int(rng.integers(0, T-1)) # predict t+1
4
5         # SAT input/target (normalized)
6         x_sat = ysat_n[t_cur] # (nsat,nx)
7         y_sat = ysat_n[t_cur+1] # (nsat,nx)
8
9         # RS input points at time t
10        if mode.upper() == "A":
11            rs_in_pts = sample_rs_input_points_from_truth(t_cur, nrs_in, nvert_in)
12        elif mode.upper() == "B":
13            rs_in_pts = rs_input_points_from_saved_yrs(t_cur, add_noise=True)
14
15        # RS query profile for t+1 at arbitrary x
16        ix_query = int(rng.integers(0, nx))
17        q_xy, iz_sel = make_query_profile_coords(ix_query, nvert=nvert_q)
18

```

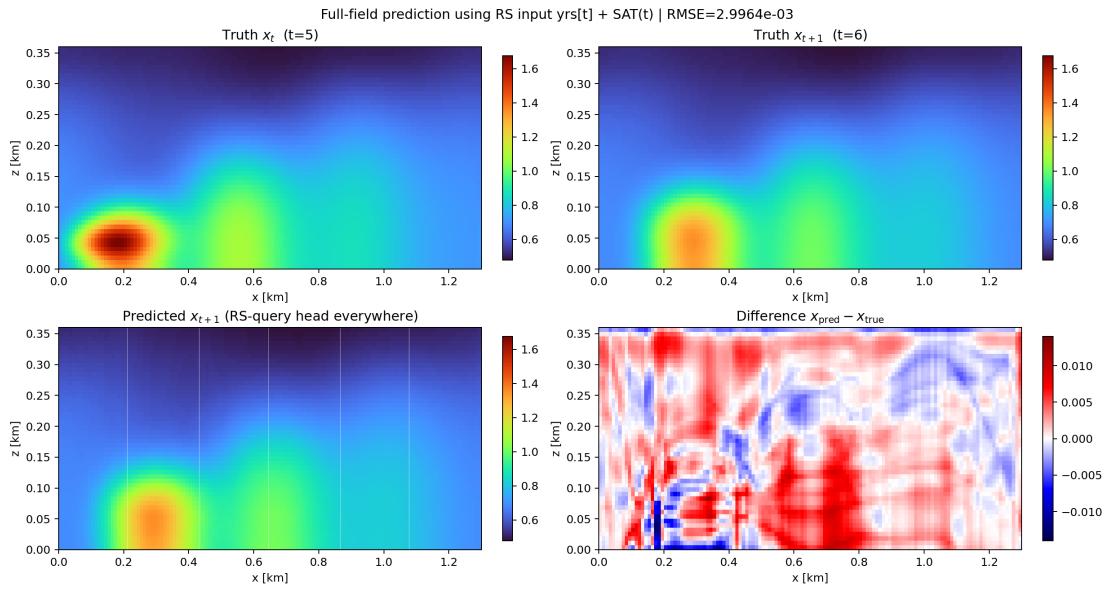


Figure 20.5: Example of full-field reconstruction at $t + 1$ produced by evaluating the RS query head on the entire grid.

```

19      # truth RS at t+1 from xtrue (normalized)
20      rs_true_n = xtrue_n[t_cur+1, iz_sel, ix_query].astype(np.float32)
21
22      return x_sat, rs_in_pts, q_xy, y_sat, rs_true_n

```

Because RS input sets can have variable size, we pad them in a custom collate function and use a binary mask to ignore padded entries. This is a typical setup for set-based learning in PyTorch.

Architecture: CNN on SAT curtain + DeepSets encoder + query decoder

The architecture is built to enforce a physically plausible coupling between satellite and radiosonde predictions. The fundamental modeling decision is:

- SAT is a $1D$ curtain $y^{sat}(c, x)$: advective transport creates translation-like patterns along x . This makes a 1D CNN a natural feature extractor.
- RS inputs form an unordered set of point measurements $\mathcal{S} = \{(x_m, z_m, y_m)\}$: order does not matter, but locations do. A DeepSets encoder (permutation-invariant pooling) provides a suitable representation.
- RS output is query-based: we predict a value at any (x_q, z_q) by combining local SAT features at x_q , a global latent state, and the query height z_q .

Thus, the model is explicitly designed such that RS inference is driven by SAT structure. In practice this means: the RS query head cannot invent arbitrary vertical profiles disconnected from the horizontal SAT features.

The forward pass is summarized by

$$\hat{y}_{t+1}^{rs}(x_q, z_q) = g_{\theta}(\text{CNN}(y_t^{sat}), \text{Set}(y_t^{rs}), x_q, z_q).$$

Permutation invariance is enforced by set pooling:

$$e_m = \psi_{\theta}(x_m, z_m, y_m), \quad E = \text{pool}(e_1, \dots, e_N),$$

where pooling can be sum/mean/max (commutative \Rightarrow order-invariant). The location information enters through (x_m, z_m) inside ψ_{θ} .

Notebook: joint architecture (SAT CNN + RS set encoder + RS query head)

```

1 class JointModel(nn.Module):
2     def forward(self, ysat_t, rs_pts, mask_in, q_xy):
3
4         # 1) encode SAT(t)
5         sat_feat_x, sat_feat_g = self.sat_enc(ysat_t)                      # (B,C,nx), (B,C)
6
7         # 2) encode RS set conditioned on SAT features
8         hr = self.rs_enc(sat_feat_x, rs_pts, mask_in)                      # (B,Hrs)
9
10        # 3) latent
11        h = self.lat_mlp(torch.cat([sat_feat_g, hr], dim=1))            # (B,H)
12
13        # 4) predict SAT(t+1)
14        ysat_pred = self.sat_head(h).view(-1, self.nsat, self.nx)
15
16        # 5) encode predicted SAT(t+1) for RS queries
17        sat1_feat_x, _ = self.sat_enc(ysat_pred)
18
19        # 6) RS query prediction at (x_q, z_q)
20        satq = sample_feat_1d(sat1_feat_x, q_xy[...,0])                  # (B,Nq,C)
21        inp = torch.cat([satq, h[:,None,:].expand(...), q_xy[...,1,None]], dim=-1)
22        yrs_pred = self.rs_head(inp).squeeze(-1)                          # (B,Nq)
23
24    return ysat_pred, yrs_pred

```

Training loss

Training uses a joint loss combining the SAT curtain error and the RS query error:

$$\mathcal{L} = \mathcal{L}_{sat} + \alpha \mathcal{L}_{rs},$$

where

$$\mathcal{L}_{sat} = \|\hat{y}_{t+1}^{sat} - y_{t+1}^{sat}\|_2^2, \quad \mathcal{L}_{rs} = \frac{1}{N_q} \sum_{q=1}^{N_q} (\hat{y}_{t+1}^{rs}(x_q, z_q) - y_{t+1}^{rs}(x_q, z_q))^2.$$

In practice, RS errors must be weighted strongly (large α), because SAT contains many more values per training example. Without re-weighting, the network will prioritize SAT and effectively ignore RS learning.

Notebook: training loop with joint loss

```

1 opt = torch.optim.Adam(model.parameters(), lr=2e-3)
2
3 nepoch = 35
4 w_sat = 1.0
5 w_rs = 30.0 # RS needs stronger weight
6
7 for ep in range(1, nepoch+1):
8     for x_sat, rs_pts, mask_in, q_xy, y_sat, y_rs in dl:
9
10         ysat_pred, yrs_pred = model(x_sat, rs_pts, mask_in, q_xy)
11
12         loss_sat = torch.mean((ysat_pred - y_sat)**2)
13         loss_rs = torch.mean((yrs_pred - y_rs )**2)
14
15         loss = w_sat * loss_sat + w_rs * loss_rs
16         loss.backward()
17         opt.step()
18         opt.zero_grad(set_to_none=True)

```

Evaluation 1: predict an RS profile at an unseen location

A first test of generalization is to predict a radiosonde profile at a new horizontal location x that was not part of the original RS station network. This checks whether the model has learned to use SAT information to drive RS prediction in a physically plausible way.

For evaluation we take RS input strictly from stored observations $yrs[t, :, :]$ (mode B), select a new unseen column index ix_{new} , query all profile heights at that location, and compare to truth $xtrue[t + 1]$.

Notebook: evaluation – RS profile forecast at unseen ix_new

```

1 t = 5
2 ix_new = 111 # manually chosen new x-index (not in rs_ix)
3
4 # build RS input from stored yrs[t]
5 pts = []
6 for irs in range(len(rs_ix)):
7     ix0 = int(rs_ix[irs])
8     for iv in range(len(rs_iz)):
9         iz0 = int(rs_iz[iv])
10        val_phys = float(yrs[t, irs, iv])
11        val_n = (val_phys - xtrue_mu) / xtrue_sd
12        pts.append([ix0/(nx-1), iz0/(nz-1), val_n])
13
14 rs_in = np.asarray(pts, dtype=np.float32)
15

```

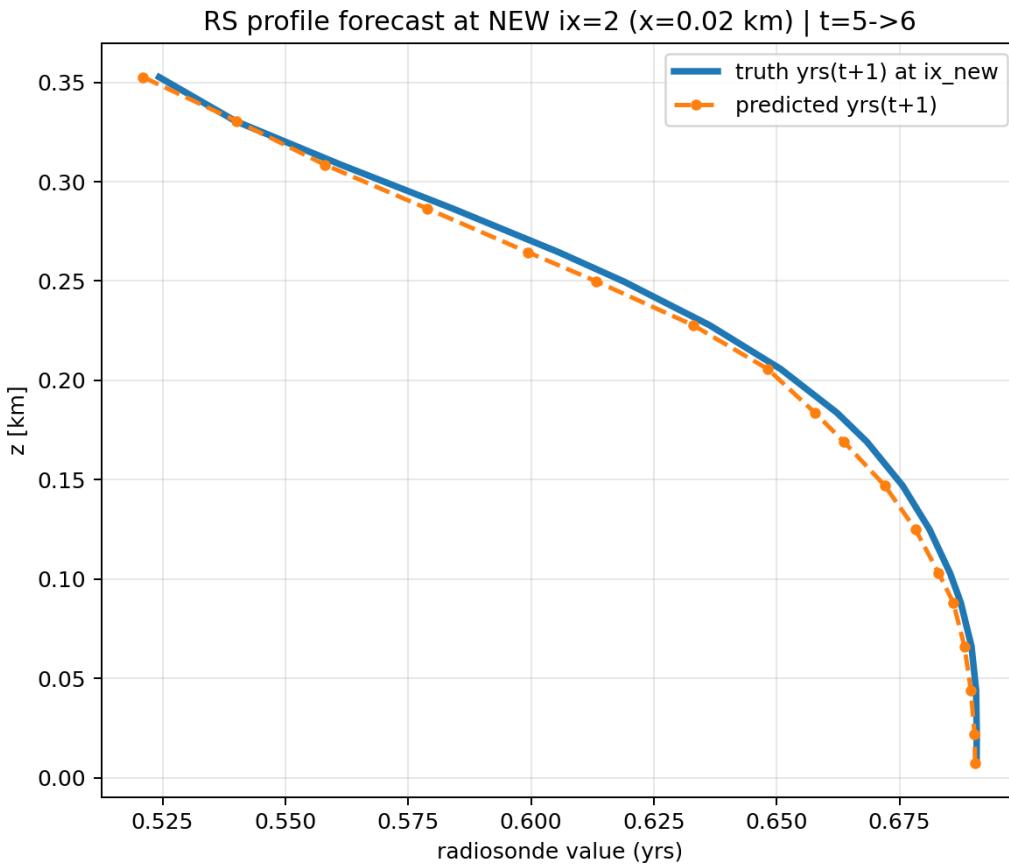


Figure 20.6: Predicted RS profile at an unseen location (dashed) compared to truth at $t + 1$ (solid).

```

16 # query profile at ix_new (same vertical grid as rs_iz)
17 iz_prof = np.array(rs_iz, dtype=int)
18 q_xy = np.stack([
19     np.full(len(iz_prof), ix_new/(nx-1), dtype=np.float32),
20     (iz_prof/(nz-1)).astype(np.float32)
21 ], axis=1)
22
23 # truth target at t+1
24 rs_true_n = xtrue_n[t+1, iz_prof, ix_new]
25
26 # predict
27 _, yrs_pred = model(x_sat, rs_pts, mask_in_t, q_xy_t)

```

Evaluation 2: full-field reconstruction at $t + 1$ by querying everywhere

Finally, the key “state-like” demonstration is full-field prediction. We evaluate the RS query head at *every grid point* (x_i, z_j) and define

$$x_{\text{pred}}(x_i, z_j, t + 1) := \hat{y}_{t+1}^{rs}(x_i, z_j).$$

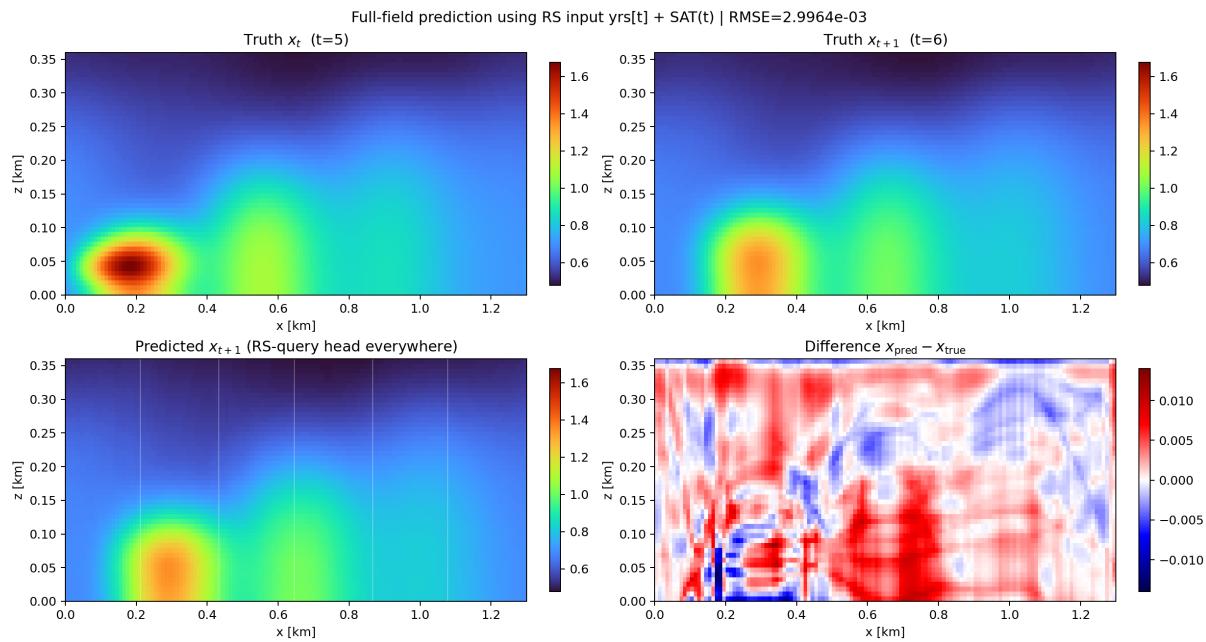


Figure 20.7: Full-field reconstruction at $t + 1$: truth at t , truth at $t + 1$, predicted field, and prediction error.

This produces a full spatial field at time $t + 1$ although the model was trained in observation space only. We then compare x_{pred} to the truth field $x_{\text{true}}[t + 1]$ and visualize both as well as their difference.

Notebook: full-field prediction by RS query head at all grid points

```

1 # build query coords for the entire (x,z) grid
2 Zg, Xg = np.meshgrid(z_norm, x_norm, indexing="ij")
3 q_all = np.stack([Xg.reshape(-1), Zg.reshape(-1)], axis=1).astype(np.float32)
4
5 # chunked forward pass
6 pred_list = []
7 for i0 in range(0, q_all.shape[0], chunk_nq):
8     q_chunk = q_all[i0:i0+chunk_nq]
9     q_xy_t = torch.tensor(q_chunk[None, ...], dtype=torch.float32, device=device)
10    _, yrs_pred = model(x_sat, rs_pts, mask_in_t, q_xy_t)
11    pred_list.append(yrs_pred[0].detach().cpu().numpy())
12
13 yrs_pred_all_n = np.concatenate(pred_list, axis=0)
14 xpred = (yrs_pred_all_n.reshape(nz, nx) * xtrue_sd + xtrue_mu)

```

This experiment closes the loop of the lecture: even though we trained purely on observations, the model can be used as a compact emulator that produces state-like outputs. In more realistic scenarios the “query everywhere” step resembles reconstruction operators that map heterogeneous observations into a gridded analysis field, but here it emerges naturally from the obs-to-obs learning setup.

Key message: Observation-space forecasting does not prevent state-like prediction — if the network output is query-based, it can generate dense fields by evaluation on a grid.

20.1.8 ORIGEN: iterative reconstruction and learning cycles from partial observations

ORIGEN (Observation-based Reconstruction and Inversion for Generative Emulation of Nonlinear systems) is a conceptual framework that explicitly couples three ingredients:

- **observations** (typically partial and noisy),
- **reconstruction / inversion** (analysis, state estimate),
- **learning of a forecast model / emulator** from reconstructed states.

The defining characteristic is that these components are not executed once, but in a *closed learning cycle*. In each cycle, a current model is used to define a background trajectory, observations are used to produce a reconstructed analysis, and the model is then updated from the reconstructed trajectory. The next cycle uses the updated model, leading to a progressive refinement of both reconstruction quality and forecast model.

In this section we demonstrate ORIGEN mechanics on a deliberately minimal example, chosen such that every update can be inspected analytically and visually.

Minimal testbed: a 2D oscillator on a circle

We define a “truth” trajectory $x_k = (x_{1,k}, x_{2,k})$ with n discrete time steps on the unit circle:

$$x_k = \begin{bmatrix} \cos \theta_k \\ \sin \theta_k \end{bmatrix}, \quad \theta_k = \frac{2\pi k}{n}, \quad k = 0, \dots, n-1.$$

The resulting states form a simple nonlinear trajectory with a known geometry.

ORIGEN notebook: truth trajectory on a circle

```

1 nn = 10
2 theta = np.linspace(0, 2*np.pi, nn, endpoint=False)
3
4 xt = np.zeros((nn, 2))
5 xt[:, 0] = np.cos(theta)
6 xt[:, 1] = np.sin(theta)

```

The purpose of this oscillator example is not to model atmospheric dynamics, but to provide a clean environment where partial observations lead to incomplete state knowledge, and where iterative reconstruction becomes visibly meaningful.

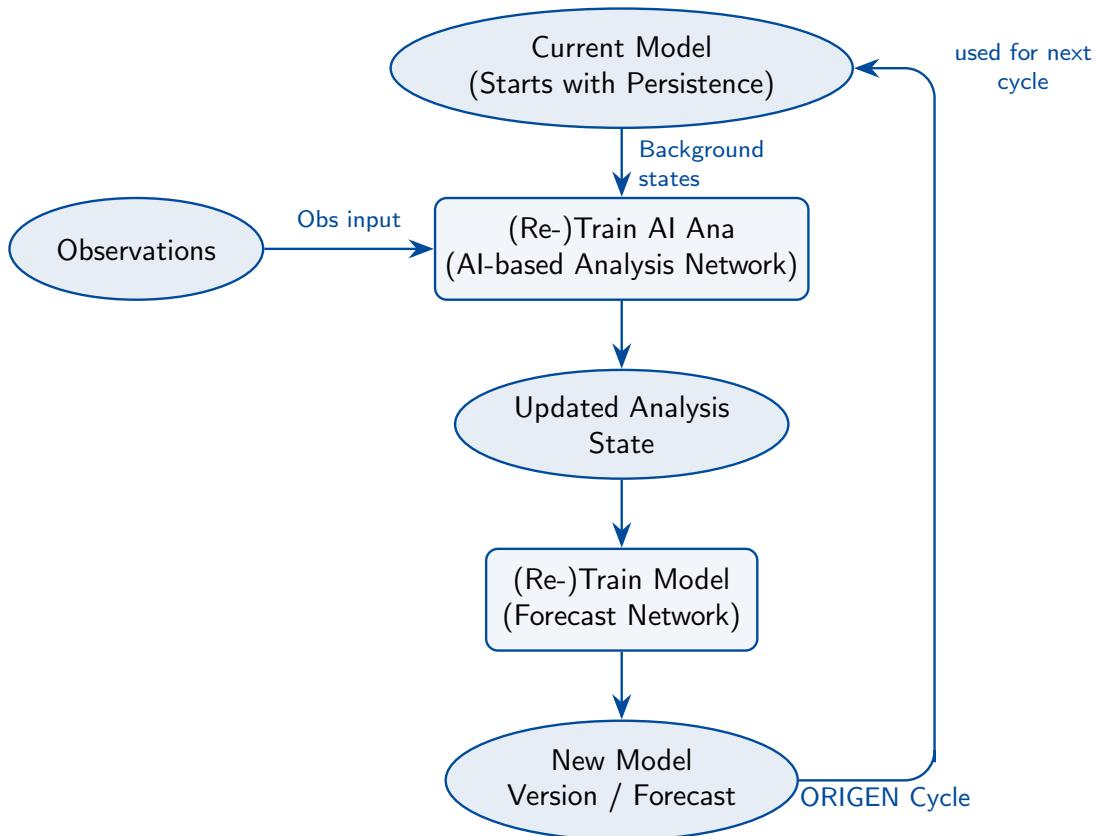


Figure 20.8: ORIGEN cycle: observations → analysis reconstruction/inversion → model learning → next cycle.

Partial observations: time-dependent observation operator

The observation at each time step is scalar and measures only one component: either x_1 or x_2 . We introduce a selector

$$s_k \in \{1, 2\},$$

which determines the time-dependent observation operator

$$H_k = \begin{cases} [1, 0] & s_k = 1 \quad (\text{observe } x_1) \\ [0, 1] & s_k = 2 \quad (\text{observe } x_2). \end{cases}$$

The observation equation is then

$$y_k = H_k x_k + \epsilon_k, \quad \epsilon_k \sim \mathcal{N}(0, R).$$

This setup mimics heterogeneous sensor availability and missing data: the system never observes the full 2D state at any single time step, but only a 1D projection.

ORIGEN notebook: observation operator for x1 or x2

```

1 obs_selector = np.random.choice([1, 2], size=nn, p=[0.5, 0.5])
2 y = np.where(obs_selector == 1, xt[:, 0], xt[:, 1]) + noise
  
```

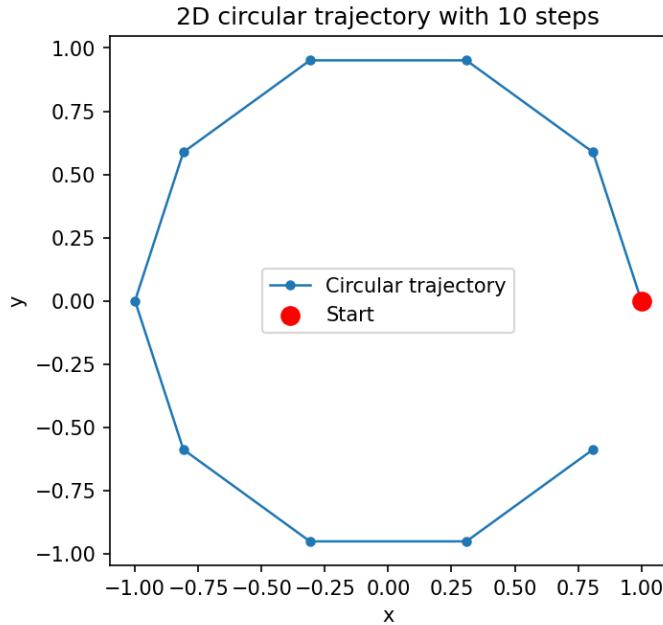


Figure 20.9: Truth trajectory x_k on a circle (minimal ORIGEN example).

Key point: because the observations are incomplete, a single assimilation pass cannot reconstruct the trajectory exactly. Iteration is required to integrate information across time steps.

3D-Var analysis step with cycling (persistence model)

We next implement a minimal 3D-Var cycle. Let x_k^b be the background state at time step k and x_k^a the analysis. For a single scalar observation, the 3D-Var update reads

$$x_k^a = x_k^b + K_k (y_k - H_k x_k^b),$$

where the innovation is

$$d_k = y_k - H_k x_k^b,$$

and the gain is

$$K_k = B H_k^\top \left(H_k B H_k^\top + R \right)^{-1}.$$

We choose for simplicity a diagonal background covariance

$$B = \begin{bmatrix} \sigma_b^2(x_1) & 0 \\ 0 & \sigma_b^2(x_2) \end{bmatrix}, \quad R = \sigma_o^2.$$

The cycling rule is given by a persistence model:

$$x_{k+1}^b = M(x_k^a), \quad M(x) = x.$$

Thus, any improvement in the analysis at time k becomes the background for time $k + 1$. Even though only one component is observed at each step, the repeated cycle allows information to accumulate over time.

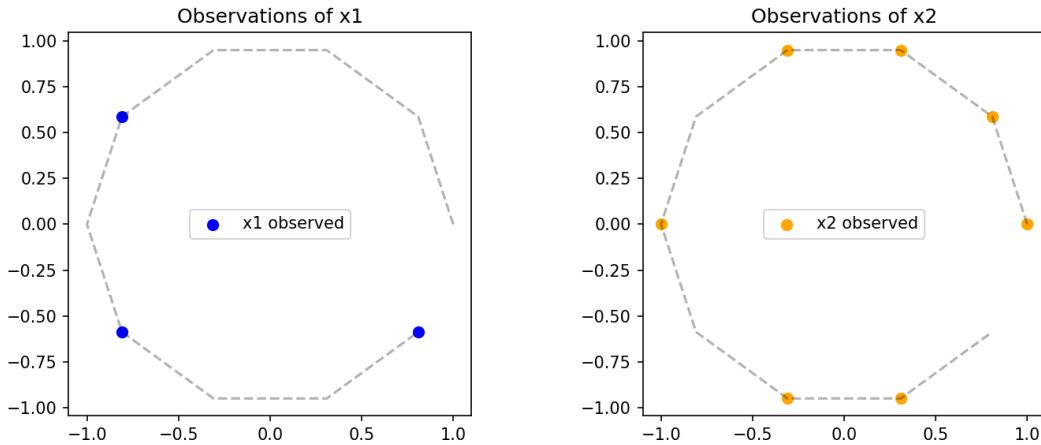


Figure 20.10: Partial observations: at each time step either x_1 (blue) or x_2 (orange) is observed.

ORIGEN notebook: 3D-Var cycle with persistence

```

1 for k in range(nn):
2     xb[k] = xbg
3     h = H_vec(obs_selector[k])           # selects x1 or x2
4
5     HBht = h @ B @ h
6     K = (B @ h) / (HBht + R)
7
8     d = y[k] - (h @ xbg)                 # innovation
9     xa[k] = xbg + K * d                  # analysis
10
11    xbg = xa[k]                         # persistence cycling

```

A useful complementary visualization is obtained in state space: plotting (x_1^a, x_2^a) reveals whether the reconstructed analysis points lie close to the circle.

Sequential rounds: convergence with complete information

To illustrate the role of information completeness, we consider two sequential rounds:

- **Round 1:** observe x_1 only (for all k),

$$y_k^{(1)} = x_{1,k} + \epsilon_k,$$

- **Round 2:** observe x_2 only, while using as background the round-1 analyses at each time step,

$$x_k^{b,(2)} := x_k^{a,(1)}.$$

When observation noise is small, convergence is immediate: after round 1 the trajectory is correct in the x_1 component, and after round 2 it becomes correct in x_2 as well. This demonstrates the

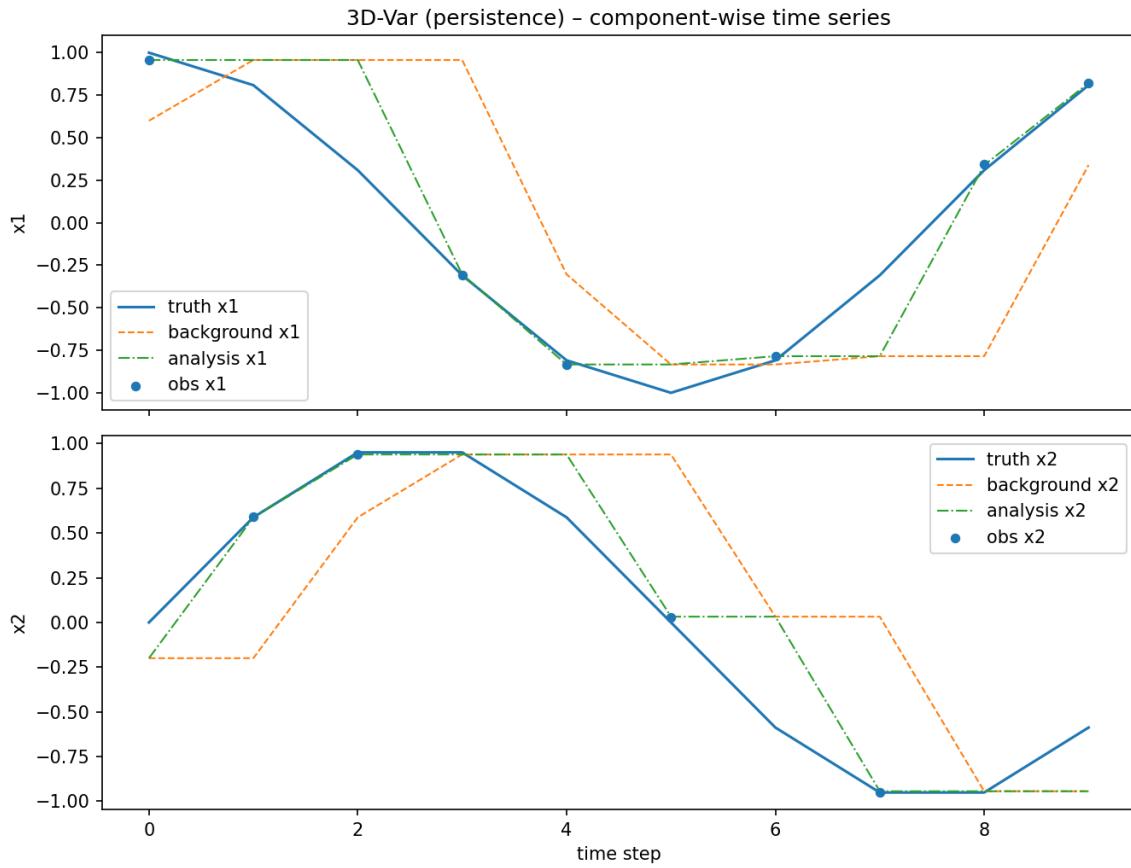


Figure 20.11: 3D-Var cycling result: background vs analysis vs truth for x_1 and x_2 time series.

essential ORIGEN intuition: *cycling accumulates information and reconstructs hidden components as soon as the missing information becomes available in later rounds.*

Iterative reconstruction across many cycles

The more realistic setting is noisy and heterogeneous observations, where the trajectory cannot be perfectly recovered in a single pass. We therefore iterate multiple cycles, each cycle producing a refined estimate of the entire sequence:

$$\{x_k^{a,(c)}\}_{k=0}^{n-1} \Rightarrow \{x_k^{a,(c+1)}\}_{k=0}^{n-1}.$$

In the notebook we demonstrate this by repeating assimilation for many cycles. The cycling mechanism is

$$x_b^{(c+1)} := x_a^{(c)}.$$

A first diagnostic is the RMSE evolution per cycle. It typically decreases as the reconstruction stabilizes.

To interpret the reconstruction geometrically, we plot the 2D trajectories over cycles. Later cycles approach the true circle more closely.

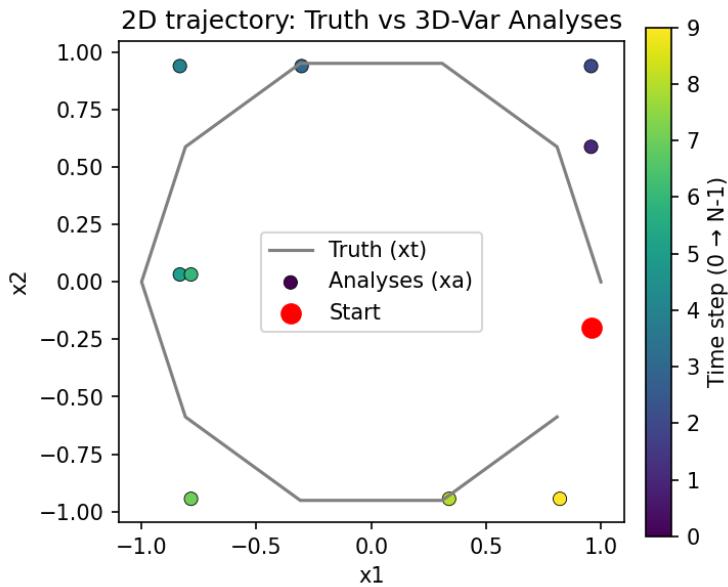


Figure 20.12: State-space view: truth circle and 3D-Var analysis points colored by time step.

Finally, a time series diagnostic of the final cycle illustrates a key property of variational filtering: the analysis does not overfit noisy scalar observations. Instead it remains a weighted compromise controlled by B and R .

Crucial observation: in this minimal setup, repeated 3D-Var cycling reduces error, but cannot achieve full convergence under persistent partial/noisy observation patterns if the uncertainty model remains fixed. In other words: a static B limits how far reconstruction can improve.

Covariance update: Kalman-style uncertainty reduction and convergence

A key step towards convergence is to update the background covariance during cycling. After each analysis update we also update the uncertainty, using the Kalman filter covariance update:

$$A_k = (I - K_k H_k) B_k, \quad B_k \leftarrow A_k.$$

This update reduces uncertainty in observed directions. Over many cycles the gain therefore adapts: it becomes consistent with the observation pattern and stabilizes the iterative reconstruction.

In the notebook this modification yields a dramatic improvement: trajectories converge strongly to the truth circle, RMSE decreases substantially, and the final time series nearly overlays the truth.

How this connects back to ORIGEN learning

So far, the notebook focuses on reconstruction mechanics: repeated cycling of analysis states from partial observations. This corresponds to the “inversion” component of ORIGEN.

The full ORIGEN principle now closes the loop: once a trajectory reconstruction $\{x_k^a\}$ is available, it can be used as training data for a learned forecast map

$$x^a(t) \mapsto x^a(t + \Delta t).$$

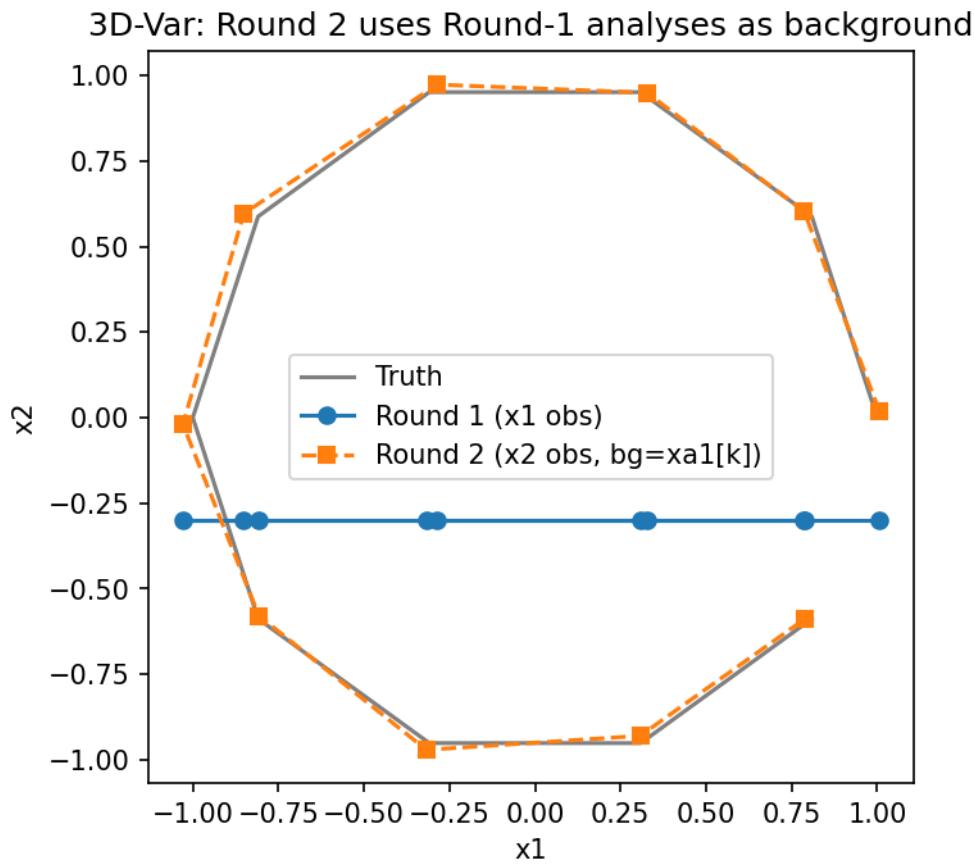


Figure 20.13: Sequential rounds: round 2 uses $x_k^{a,(1)}$ as background and completes the reconstruction.

This yields an improved emulator (learned dynamics), which in turn produces better backgrounds for the next reconstruction cycle. The iterative combination of reconstruction and model learning is the mechanism by which ORIGEN turns sparse partial observation sequences into stable forecasting capability.

Key message: ORIGEN is not a one-shot DA method. It is a closed iterative learning loop:

$$\text{obs} \Rightarrow \text{reconstruction} \Rightarrow \text{model learning} \Rightarrow \text{next cycle.}$$

Even in the minimal oscillator example, we see the essential pattern: cycling integrates information over time, and adaptive uncertainty modeling (covariance update) enables convergence under partial noisy observations.

20.1.9 ORIGEN on Lorenz–63: reconstruction from partial observations + learned dynamics

The circle oscillator example in the previous section isolates the key idea of ORIGEN: *iterative reconstruction from partial observations* becomes possible when we cycle analysis states and

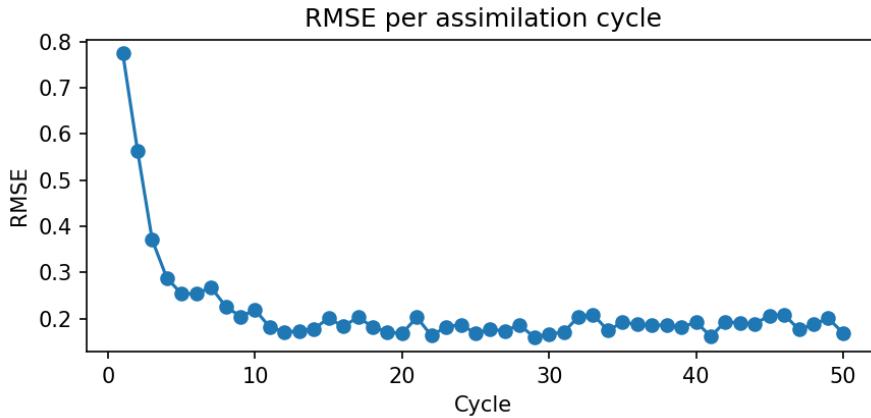


Figure 20.14: Iterative reconstruction: RMSE per cycle (partial noisy observations).

adapt uncertainty. We now move to a more realistic chaotic system: the classical Lorenz–63 model. Here ORIGEN becomes an end-to-end pipeline:

partial observations \Rightarrow iterative reconstruction \Rightarrow learned dynamics \Rightarrow forecast rollouts.

The notebook ORIGEN – Lorenz-63: Observation-based Reconstruction + Learned Dynamics implements this full loop and stores all results in `origen_163/`.

Lorenz–63 truth trajectory and partial observations

Lorenz–63 is defined by the nonlinear ODE

$$\dot{x} = \sigma(y - x), \quad \dot{y} = x(\rho - z) - y, \quad \dot{z} = xy - \beta z,$$

with standard parameters $(\sigma, \rho, \beta) = (10, 28, 8/3)$. A “truth” trajectory $\mathbf{x}(t) = (x(t), y(t), z(t))^\top$ is generated by numerical integration from a fixed initial condition.

Observations are formed by selecting only a subset of the three components. For a given choice (e.g. measured= xz), the observation equation reads

$$\mathbf{y}(t_k) = H \mathbf{x}(t_k) + \epsilon_k, \quad \epsilon_k \sim \mathcal{N}(0, R),$$

where H is a (time-independent) selector matrix, e.g.

$$H_{xz} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The key difficulty is that the observed information is incomplete: reconstructing the full 3D chaotic state from partial noisy measurements requires a robust regularization and (in ORIGEN) iteration.

3D-Var reconstruction with Gaussian background covariance

We reconstruct the full state at each observation time using a linear 3D-Var update. For a given background \mathbf{x}_k^b and observation \mathbf{y}_k we compute the analysis

$$\mathbf{x}_k^a = \mathbf{x}_k^b + K_k (\mathbf{y}_k - H \mathbf{x}_k^b),$$

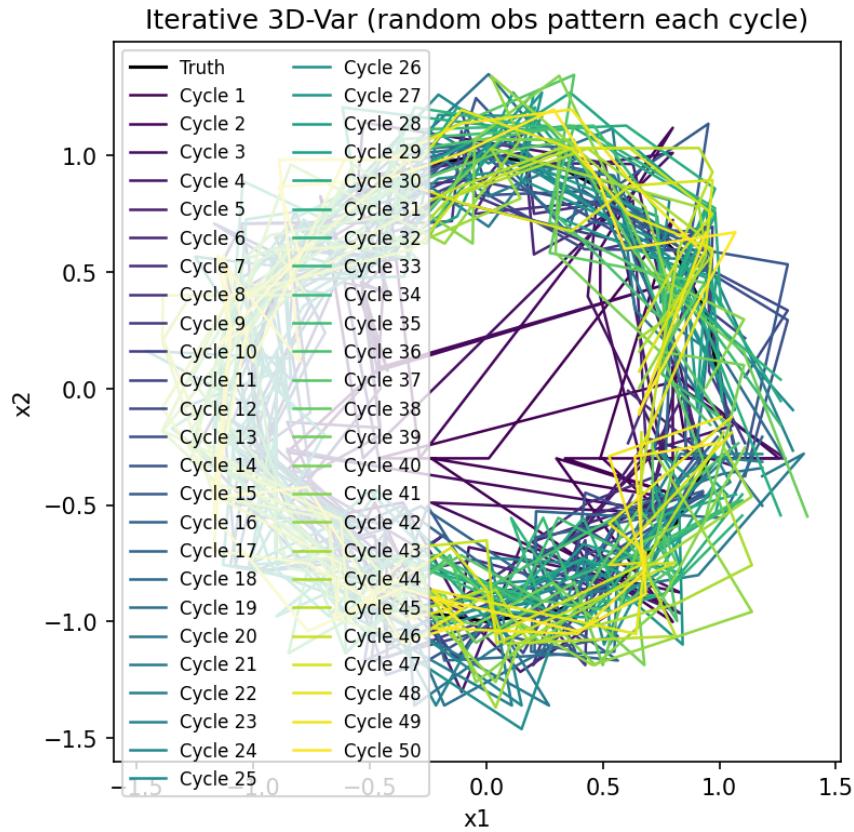


Figure 20.15: Iterative 3D-Var reconstruction in state space: selected cycles approach the truth trajectory.

with gain matrix

$$K_k = B_k H^\top (H B_k H^\top + R)^{-1}.$$

As initial uncertainty model, a symmetric Gaussian background covariance B is used, chosen here as a smooth correlation across the three components (i.e. coupling x, y, z),

$$(B_0)_{ij} = \sigma_b^2 \exp\left(-\frac{(i-j)^2}{2\ell^2}\right), \quad i, j \in \{1, 2, 3\}.$$

This makes the reconstruction non-trivial: even when only x and z are observed, the coupling encoded in B allows information to leak into the unobserved component y .

The simplest diagnostic is a single-pass sequential reconstruction with a fixed background (e.g. $x_k^b = 0$), which already pulls the analysis onto a plausible trajectory shape but cannot fully recover the truth in chaotic regimes.

Iterative ORIGEN reconstruction with covariance updates

ORIGEN introduces iteration and uncertainty adaptation. Instead of assimilating once, we cycle multiple reconstruction iterations. In iteration i , we:

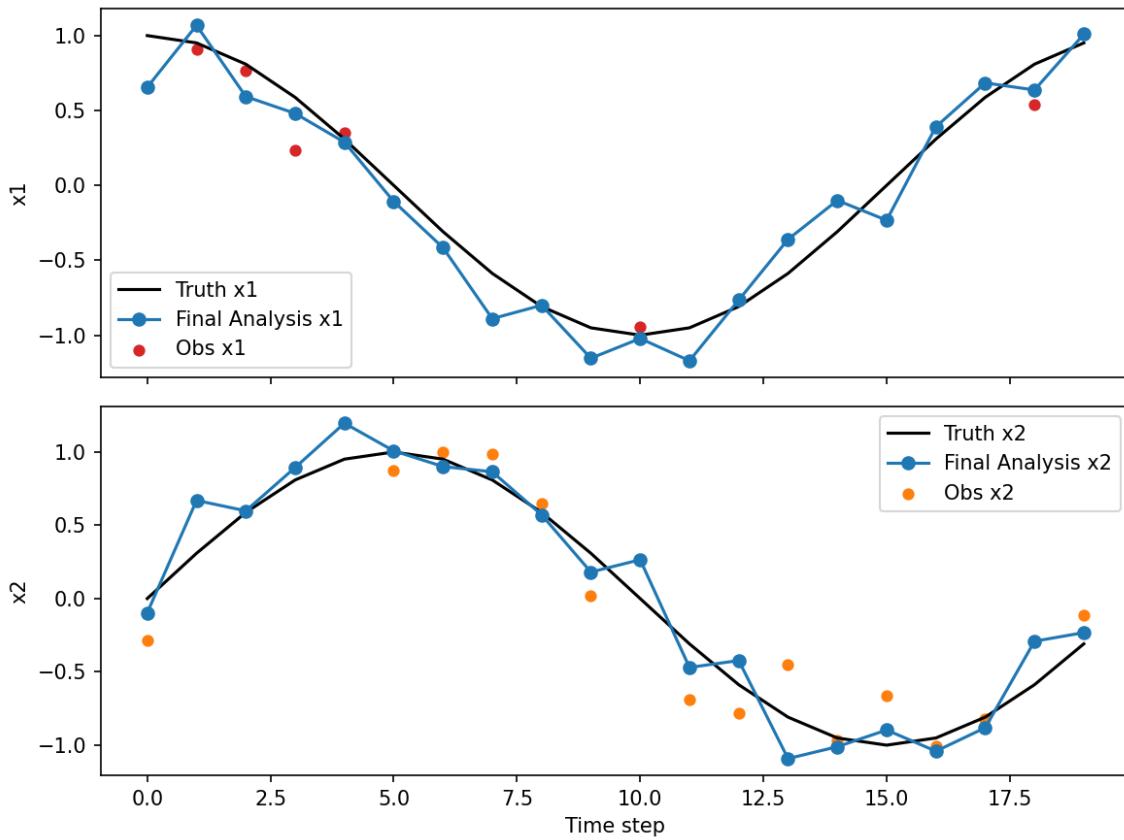


Figure 20.16: Final-cycle time series: truth vs reconstructed analysis, with noisy partial observations.

1. draw a random observation subset (e.g. x, y, z, xy, xz, yz),
2. reconstruct an analysis trajectory $\mathbf{x}^{a,(i)}(t_k)$ with 3D-Var,
3. update the background: $\mathbf{x}^{b,(i+1)} \leftarrow \mathbf{x}^{a,(i)}$,
4. update the covariance (Kalman-style): $B^{(i+1)} \leftarrow P^{a,(i)}$.

The covariance update is central. Using the (stable) Joseph form, the analysis covariance is

$$P_k^a = (I - K_k H) B_k (I - K_k H)^\top + K_k R K_k^\top,$$

and the cycle reduces background uncertainty in directions repeatedly informed by observations. This stabilizes the reconstruction and makes later iterations increasingly consistent.

The notebook stores the reconstructed trajectories for all iterations:

$$\mathbf{xa_all}[i, k, :] \approx \mathbf{x}^{a,(i)}(t_k),$$

which later becomes the training set for learning dynamics.

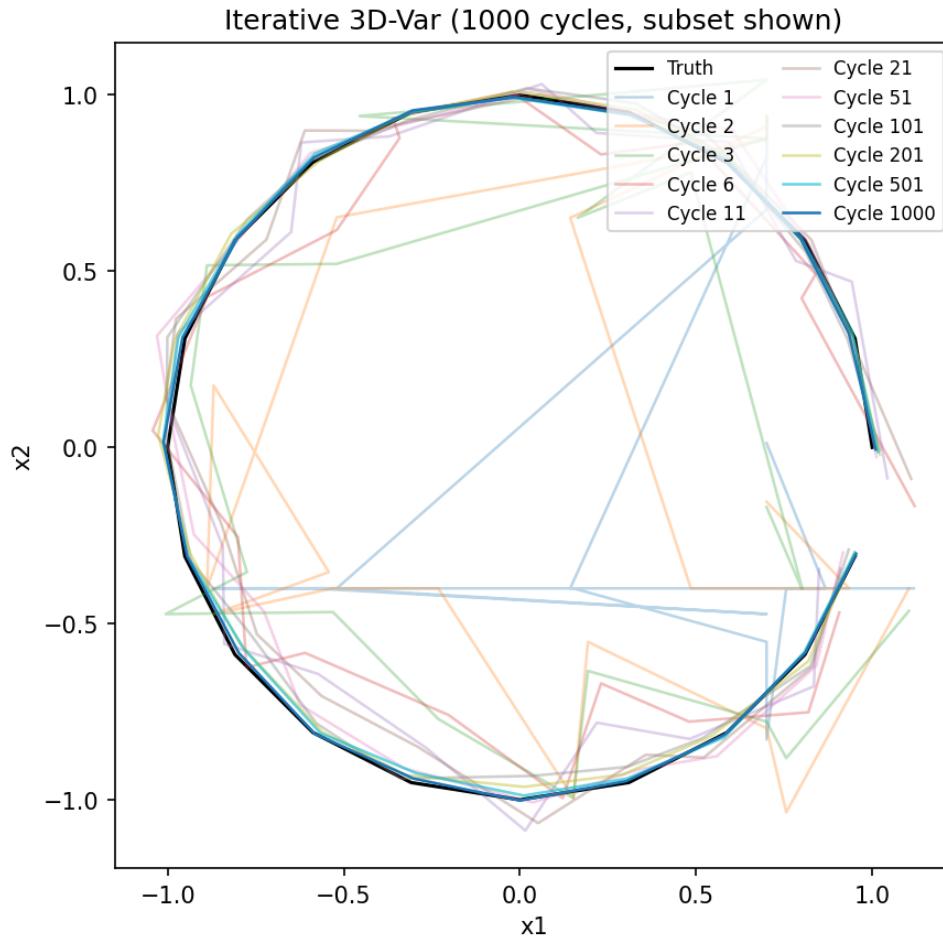


Figure 20.17: With covariance update: selected cycles show strong convergence towards the truth trajectory.

Sequential two-stage illustration. A particularly intuitive bridge is a two-stage scheme: first assimilate one subset (e.g. x, z) and obtain P^a , then assimilate another subset (e.g. x, y) using the updated covariance $B_k \equiv P_k^a$ as uncertainty model. This already demonstrates how uncertainty reduction enables stronger convergence.

Many ORIGEN iterations. In the full ORIGEN reconstruction, the observation subset varies between iterations. This emulates realistic heterogeneous sensing and missing data, but now embedded into a controlled cycling loop. The output is a sequence of refined trajectories, progressively constrained by accumulated information.

Learning dynamics from reconstructed trajectories

Once a sufficiently consistent reconstructed trajectory is available, ORIGEN turns the reconstruction output into training data for dynamics learning. We train a small neural network as a one-step map

$$\mathbf{x}^a(k) \mapsto \mathbf{x}^a(k+1),$$

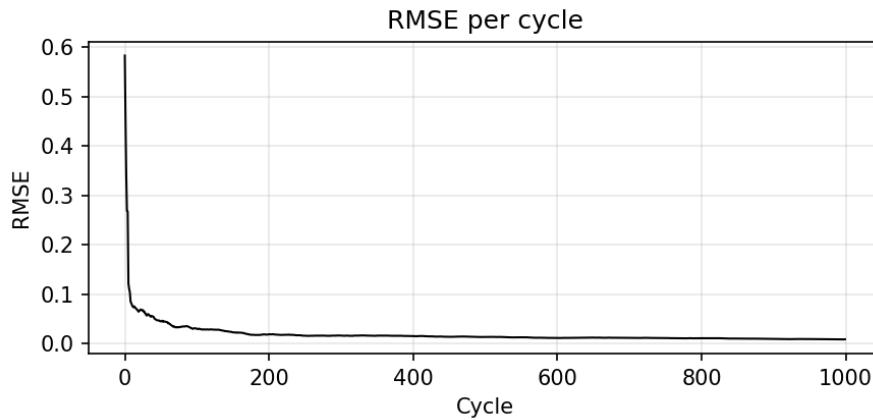


Figure 20.18: RMSE per cycle with covariance update: uncertainty reduction stabilizes the reconstruction.

i.e. it emulates the discrete-time flow map on the reconstructed manifold.

In the notebook, the training data are drawn from a selected reconstruction iteration (e.g. iteration 20), and a compact MLP is trained with standard MSE loss. This yields a learned emulator that can be used for forecasting by rollout.

Forecast evaluation: rollouts and short forecasts

The ultimate test of a learned dynamical model is not the one-step error, but whether it remains stable and consistent under repeated application. We therefore evaluate by:

- **long rollout:** forecast the full trajectory from one initial state,
- **many short forecasts:** start short rollouts from multiple analysis states along the reconstructed trajectory.

The short-forecast evaluation is particularly informative: it illustrates both stability and local accuracy across the attractor, and it provides a simple visual narrative for the ORIGEN concept: *a reconstructed trajectory becomes a training signal for a forecast model, which is then validated by many rollouts.*

Rollout-aware fine-tuning. A known issue in learned dynamical systems is error accumulation: a one-step trained model can drift under rollout because it is evaluated on its own predictions. To mitigate this, the notebook includes an optional fine-tuning stage that adds a short-rollout loss term. This explicitly encourages stability of the forecast map under repeated application, which substantially improves the qualitative behavior of rollouts.

ORIGEN summary on Lorenz–63

The Lorenz–63 experiment demonstrates the full ORIGEN principle in a compact setting:

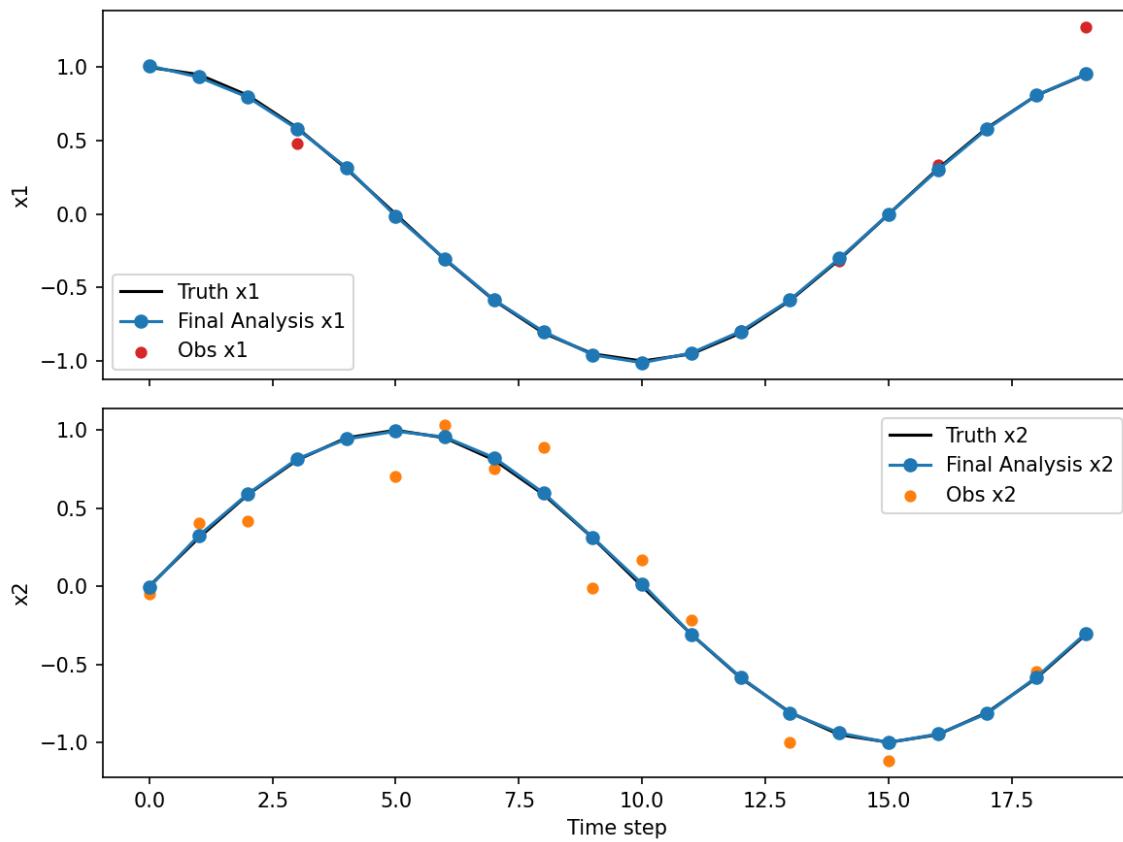


Figure 20.19: Final cycle with covariance update: x_1 and x_2 trajectories converge.

- **Reconstruction:** repeated 3D-Var with adaptive covariance integrates partial noisy observations into a consistent full-state trajectory.
- **Learning:** reconstructed trajectories provide training data for an emulator of nonlinear dynamics.
- **Forecasting:** the emulator is validated by rollout-based tests, including many short forecasts.

Key message: ORIGEN provides a closed learning cycle that transforms incomplete observations into a learned forecast model:

observations \Rightarrow iterative reconstruction + uncertainty update \Rightarrow learned dynamics \Rightarrow forecast rollouts.

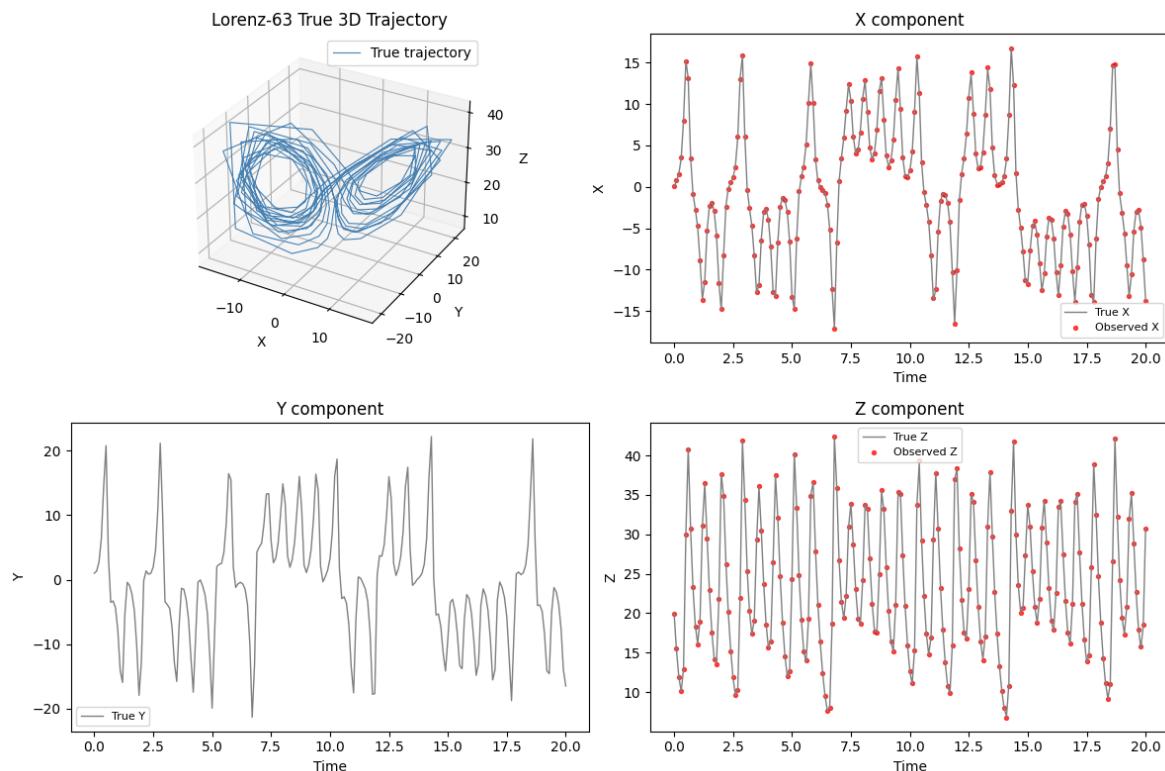


Figure 20.20: Lorenz–63 truth trajectory and partial noisy observations (example shown for a measured subset such as x, z).

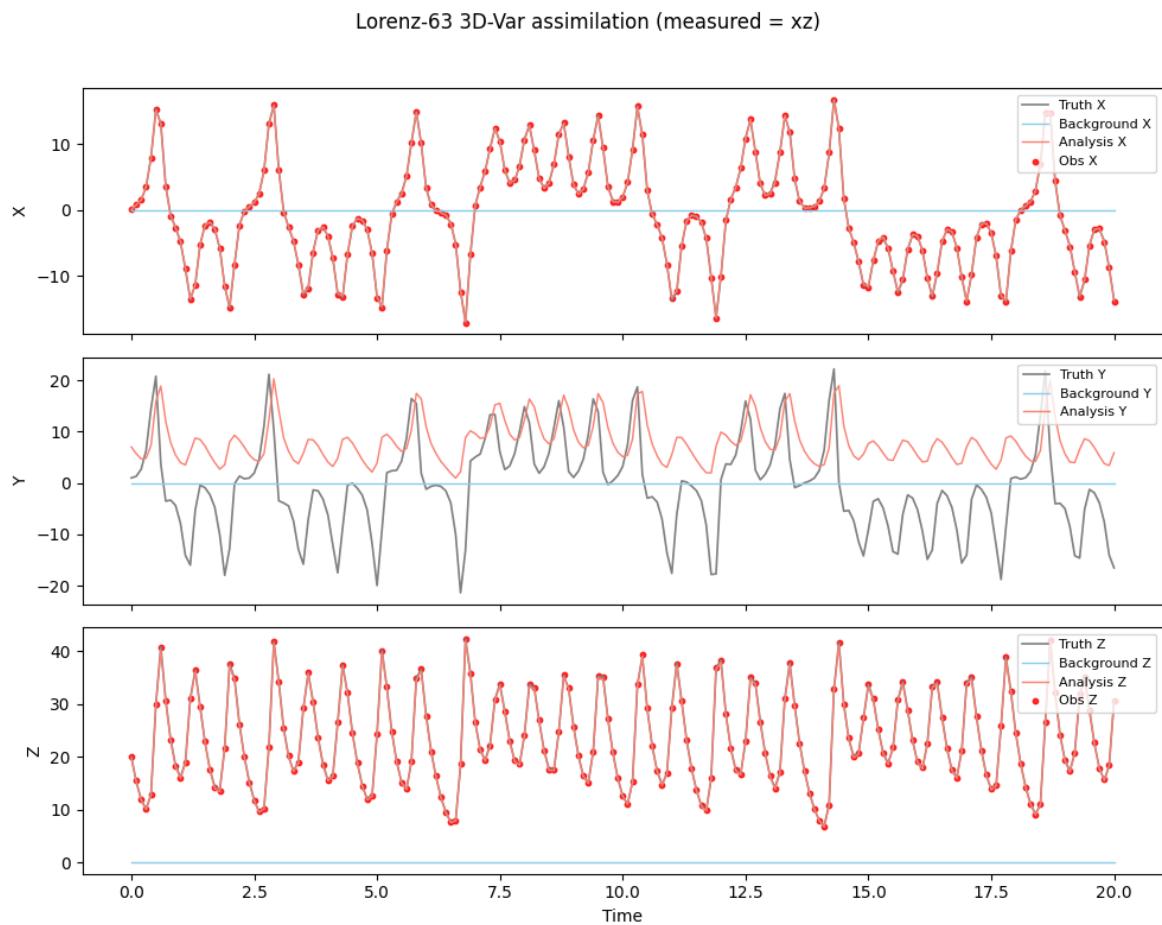


Figure 20.21: Single-pass 3D-Var reconstruction for all time steps (example with measured subset).

Two-stage 3D-Var with Kalman B update between stages

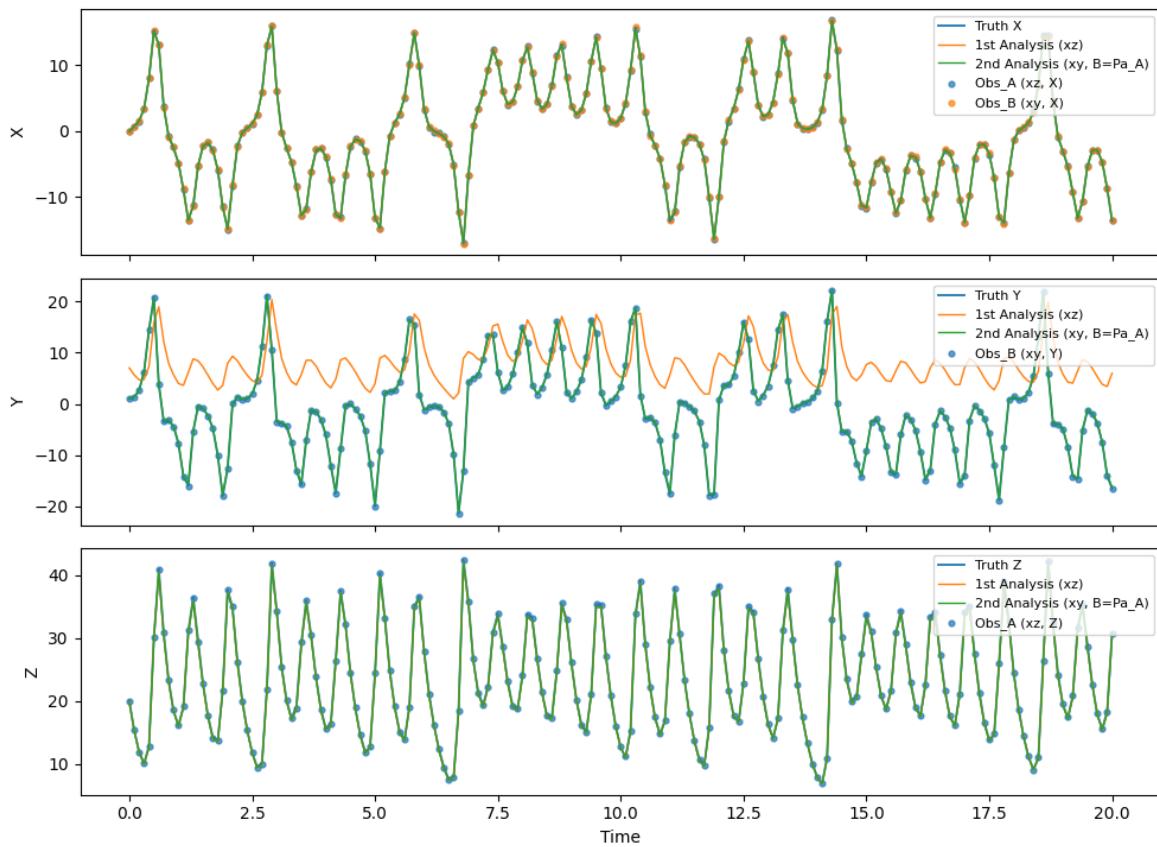


Figure 20.22: Two-stage 3D-Var reconstruction with Kalman-style covariance propagation between stages.

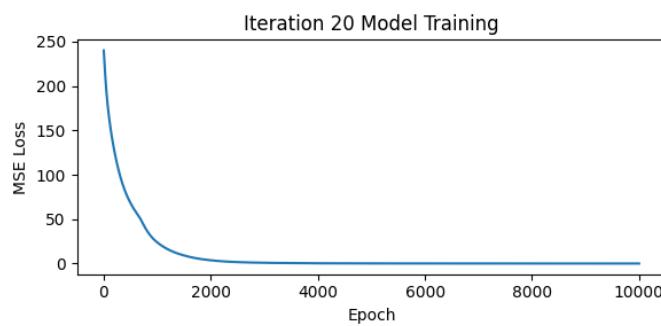


Figure 20.23: Training diagnostics of the learned one-step map on reconstructed Lorenz–63 states.

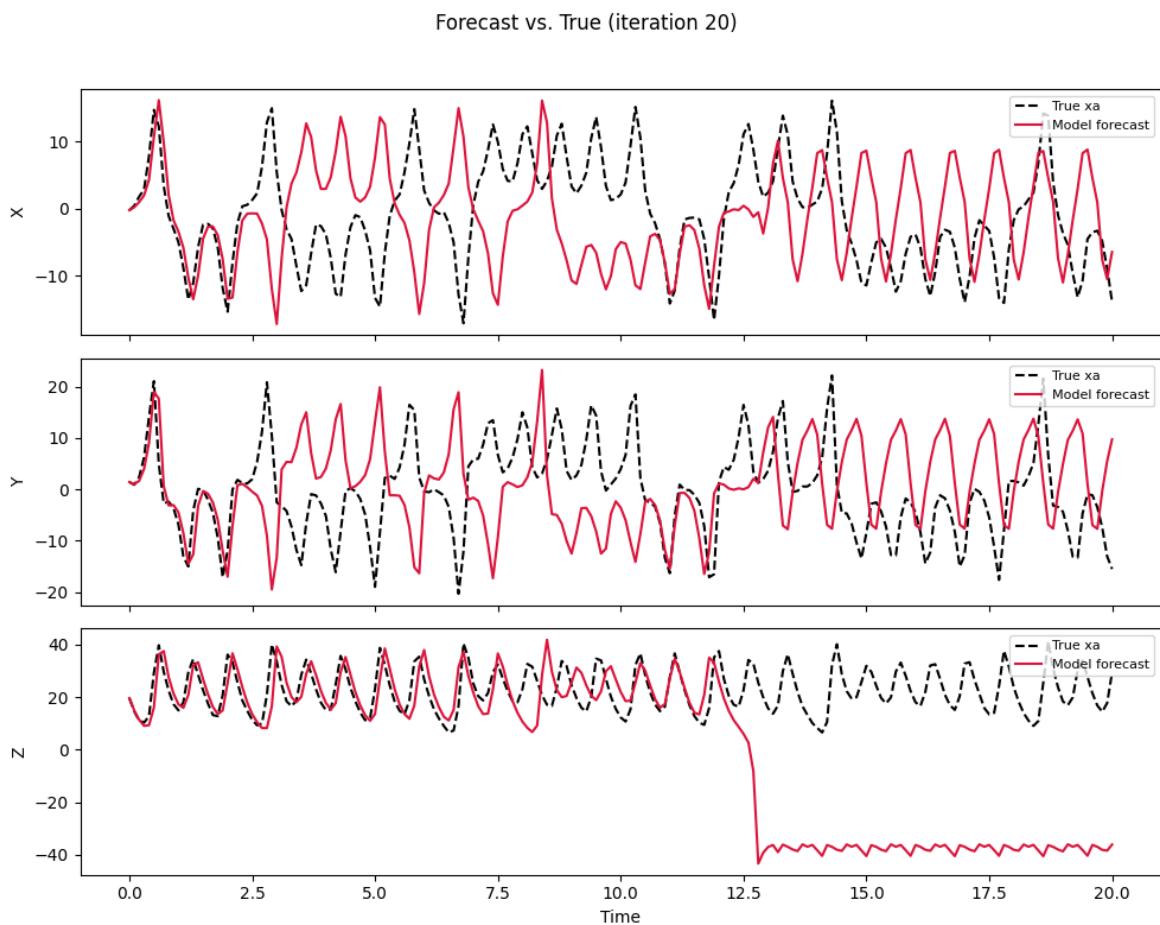


Figure 20.24: Rollout forecast: learned emulator (red) compared to reconstructed reference trajectory (black dashed).

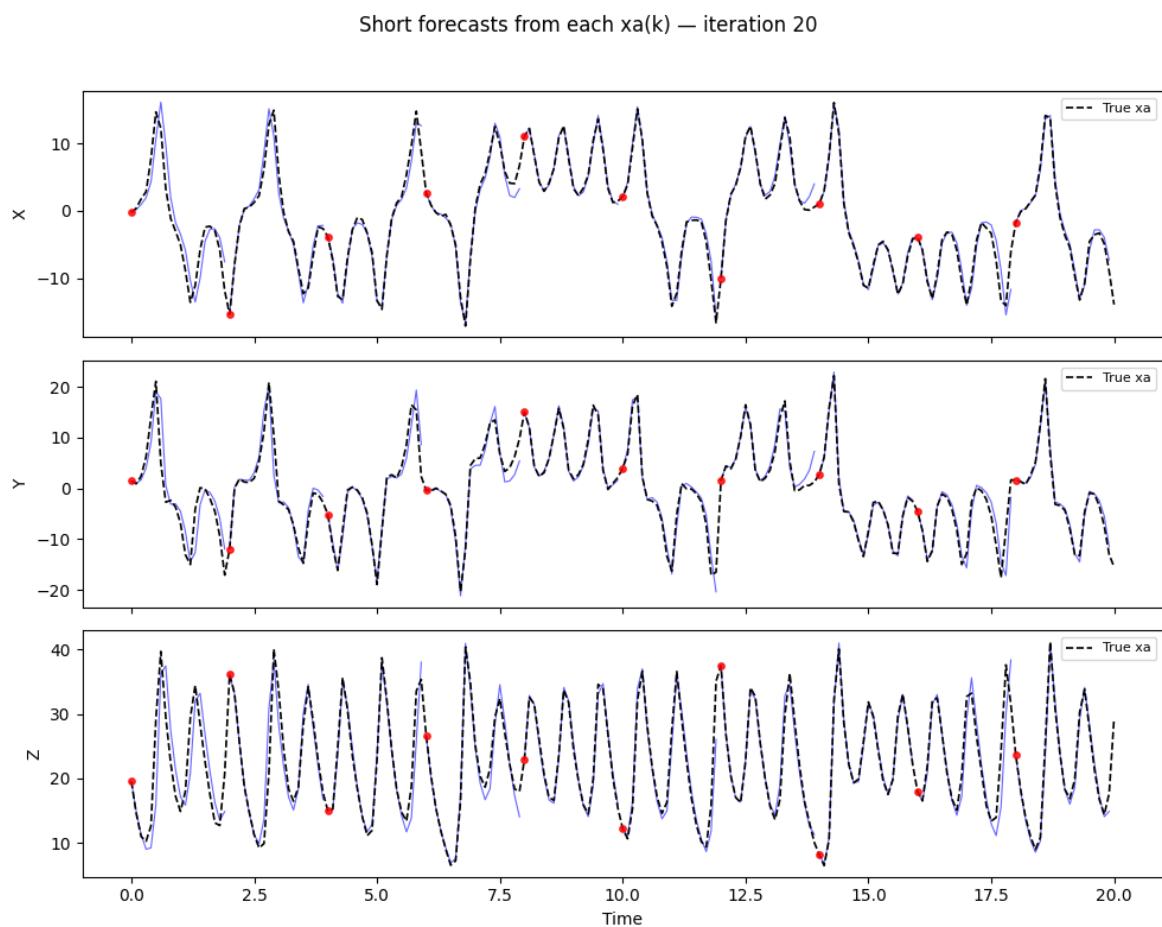


Figure 20.25: Short forecasts (blue) starting from reconstructed analysis states (red markers), compared to the reconstructed reference trajectory (black dashed).

Chapter 21

History of Large Language Models

21.1 The History of Large Language Models

21.1.1 The Beginnings: From Vision to the First Machine Translation

The idea of machines that can understand and use language dates back a long way. As early as the 1950s, Alan Turing laid the groundwork for computational linguistics with his vision of “thinking machines”. He developed the famous Turing Test to determine whether a machine could communicate so convincingly that it was indistinguishable from a human. This concept inspired many early chatbot systems.

A practical example of machine language processing was the Georgetown-IBM experiment in 1954, which could automatically translate simple sentences. It soon became clear that language is not just words and grammar, but also context, meaning, and nuance—a major challenge for machines.

21.1.2 The 1970s and 1980s: Rule-Based Systems and Symbolic AI

In the following decades, researchers relied on rule-based systems that analyzed language using fixed patterns. ELIZA (1966) was one of the most well-known early programs of this type. It simulated therapeutic conversations by recognizing keywords and returning pre-defined responses—without any real language understanding.

Another milestone was SHRDLU (1970), a system that interpreted simple verbal commands in a simulated block world. These early efforts showed that while AI could process language, it was still heavily dependent on manually crafted rules and responses.

21.1.3 The 1990s: Statistics Over Rules — The Rise of Probabilistic Models

With the increasing availability of large text corpora, researchers began to use statistical methods instead of fixed rules. N-gram models analyzed word sequence probabilities, producing text that appeared more natural.

IBM's work in statistical machine translation was especially groundbreaking. These systems

performed much better than earlier rule-based approaches and laid the foundation for modern translation technologies. However, they computed only probabilities, without a deeper understanding of language.

21.1.4 The 2000s: The Rise of Neural Networks and Deep Learning

Advances in artificial neural networks marked a major breakthrough in the 2000s. Recurrent Neural Networks (RNNs) and especially Long Short-Term Memory (LSTM) networks greatly improved the processing of language sequences.

A revolutionary step was the introduction of word embeddings. While older systems treated language as a mere sequence of words, models like word2vec (2013) mapped words into a multidimensional space, making it possible to compute semantic similarities (e.g., “king” and “queen” are related, or “Paris” belongs to “France”).

21.1.5 The 2010s: Transformers — The Revolution in Language Processing

In 2017 the breakthrough came with the Transformer architecture by Vaswani et al. This model employed self-attention to analyze a word’s context across the entire sentence rather than only its neighbors. This approach quickly became the standard for nearly all modern language models.

In 2018, Google introduced BERT, a model that processed text bidirectionally. BERT revolutionized many NLP tasks by understanding words in the context of their surroundings.

21.1.6 The 2020s: Generative AI and the Breakthrough of Large Language Models

In 2020, OpenAI’s GPT-3 made headlines. With 175 billion parameters, it was the most powerful language model of its time, capable of generating fluent text, writing code, and answering questions impressively.

Then, with ChatGPT (2022), AI became truly accessible. Suddenly, anyone could chat with an AI that responded in natural language, explained complex topics, and even wrote creative texts. The introduction of GPT-4 (2023) and other multimodal models, which can process text alongside images and other data, expanded AI’s versatility.

21.1.7 Current Trends: The Future of Language AI

Today, in 2025, AI is evolving rapidly:

- Multimodal models can analyze not only text but also images, videos, and audio.
- AI agents are taking on complex tasks autonomously and interacting with other systems.
- Ethical challenges are coming into focus to prevent misuse, biases, and misinformation.

Key Factors Driving Progress

Two essential factors have made the development of language models possible:

1. **Large Data Sets:** AI models require enormous text corpora to learn language effectively.
2. **Computing Power:** Advances in hardware, especially high-performance GPUs and TPUs, have enabled the training of ever larger models.

Ethical and Societal Considerations

As AI grows more powerful, new challenges arise:

- **Risks:** The spread of misinformation, biases in models, and potential misuse by fraudsters or manipulators.
- **Opportunities:** Enhanced communication, task automation, and entirely new possibilities for research, education, and creativity.

The impact of these developments on our worldview and understanding of humanity will be explored further in this book.

21.2 The Georgetown-IBM Experiment of 1954

The Georgetown-IBM Experiment of 1954 marked a significant milestone in the history of machine translation. On January 7, 1954, scientists from Georgetown University in collaboration with IBM demonstrated a system that could automatically translate Russian sentences into English.

21.2.1 Background and Objectives

The main goal of the experiment was to showcase the potential of machine translation and to attract public and governmental support for further research. Although the system was limited in scope, it impressively demonstrated the capabilities of computer technology in language processing.

21.2.2 Technical Details

The translation system was based on the IBM 701, one of IBM's first commercial scientific computers. It had a vocabulary of about 250 words and six grammar rules. The vocabulary covered fields such as politics, law, mathematics, chemistry, metallurgy, communications, and military affairs. Words were stored on punch cards and processed by the computer. The translation was fully automatic, with no human intervention during the process.

21.2.3 Experiment Process

During the demonstration, more than 60 Russian sentences (transliterated into Latin) were entered into the computer by an operator who did not understand Russian. The IBM 701 produced the corresponding English translations within seconds. The selected sentences covered topics from politics and law to mathematics and natural sciences.

21.2.4 Examples of Translated Sentences

Some examples of the translated sentences are:

- **Russian (transliterated):** Mi pyeryedayem mislyi posryedstvom ryechyi.
English: We transmit thoughts by means of speech.
- **Russian (transliterated):** Vyelyichyina ugla opryedyelyayetsya otnoshenyiyem dlyini dugi k radyusu.
English: The magnitude of an angle is determined by the ratio of the arc length to the radius.
- **Russian (transliterated):** Myezhdunarodnoye ponyimaniye yavlyayetsya vazhnim faktorom v ryeshenyi polyticheskix voprosov.
English: International understanding is an important factor in resolving political issues.

21.2.5 Reception and Impact

The demonstration received widespread media attention and was hailed as a great success. It raised high expectations for automatic translation systems and led to increased investments in research. Although the developers were optimistic that machine translation could be solved within three to five years, the actual progress turned out to be far more complex and time-consuming.

Overall, the Georgetown-IBM Experiment laid the groundwork for future research and development in machine translation and greatly influenced the field of computational linguistics.

21.3 ELIZA — The First Chatbot in History

ELIZA is one of the most well-known early programs for natural language processing and is often regarded as the first chatbot in history. Developed between 1964 and 1966 by Joseph Weizenbaum at MIT, ELIZA was designed to show that machines could simulate human-like interactions using simple linguistic tricks.

21.3.1 How ELIZA Worked

ELIZA used a pattern-based approach, recognizing keywords in user inputs and returning pre-formulated responses by repeating or rephrasing parts of the input. It did not possess true language understanding but operated using simple rules and scripted patterns.

Its best-known script, the “DOCTOR” script, simulated a psychotherapist using a Rogerian approach, for example:

- **User:** I have problems with my mother.
- ELIZA:** Tell me more about your mother.

This led many users to believe that ELIZA truly “understood” them, even though it was simply performing rule-based text transformations.

21.3.2 The Significance of ELIZA

Although technically simple, ELIZA was the first program to show that people tend to attribute human characteristics to machine interactions—a tendency Weizenbaum warned against. He argued that humans should not overly personalize machines since they lack true understanding.

ELIZA had a lasting impact on the development of artificial intelligence and language processing, inspiring later chatbots such as PARRY (1972), ALICE (1995), and ultimately modern systems like ChatGPT.

21.3.3 Limitations and Constraints

ELIZA had several limitations:

- It could only manage very limited conversations.
- It did not understand context or meaning.
- Its responses were generated solely through simple text patterns without any actual analysis.

21.3.4 Publicly Available Literature on ELIZA

For further information on ELIZA, see:

1. The Wikipedia article on ELIZA for an overview of its history, functionality, and significance.
2. Joseph Weizenbaum’s original 1966 paper on ELIZA for a detailed technical description.
3. The original publication “ELIZA — A Computer Program for the Study of Natural Language Communication between Man and Machine” (1966).
4. Articles and historical overviews on the evolution from ELIZA to modern chatbots.
5. Additional background available on MIT’s website regarding early AI research.

21.4 Probabilistic Models in Language Processing in the 1990s

21.4.1 The Paradigm Shift: From Rules to Probabilities

Until the 1980s, rule-based methods dominated machine language processing. Systems like ELIZA (1966) and SHRDLU (1970) used pre-defined rules and patterns, making them inflexible to new expressions and requiring extensive manual adjustments.

In the 1990s, a fundamental shift occurred: statistical probabilistic models were seen as a more powerful alternative. Instead of manually specifying rules, these models analyzed large text datasets to identify word and sentence patterns based on probabilities.

21.4.2 N-Gram Models: The Foundation of Modern Language Processing

One of the key developments of this era was the introduction of N-gram models. An N-gram is a sequence of N consecutive words:

- A unigram considers a single word.
- A bigram analyzes word pairs (e.g., “good morning”).
- A trigram considers three consecutive words (e.g., “I am going today”).

These models estimated the probability of a word based on the preceding words. For example, a bigram model might calculate that “morning” is highly likely to follow “good.”

21.4.3 The N-Gram Formula

In a bigram model, the probability of a sentence is given by:

$$P(W_1, W_2, \dots, W_n) = P(W_1) P(W_2 | W_1) P(W_3 | W_2) \dots P(W_n | W_{n-1})$$

Here, $P(W_n | W_{n-1})$ is the probability of a word given the previous word.

This simple method allowed for the calculation of sentence probabilities and significantly improved machine translation, speech recognition, and autocomplete features.

21.4.4 Statistical Machine Translation (IBM Models)

IBM developed a series of models in the 1990s that used statistical methods for translating texts:

- Parallel texts (e.g., English-French) were analyzed.
- Probabilities were computed to determine which word in one language corresponds to a specific word in another.
- The more frequently a word pair occurred in the training data, the more likely it was to be used in future translations.

IBM Models 1–5 (1990–1993) introduced alignment methods to calculate word-to-word correspondences. Models 4 and 5 further improved these ideas by considering word order.

21.4.5 Hidden Markov Models (HMMs) for Speech Recognition

While N-gram models were used for text, automatic speech recognition systems increasingly relied on Hidden Markov Models (HMMs):

- HMMs are probabilistic models that represent a sequence of states (e.g., spoken sounds).
- They were employed in systems such as Dragon Dictate and early versions of Google Voice.
- HMMs improved speech recognition by considering both the probability of individual words and their phonetic variations.

21.4.6 Influence and Limitations

Advantages:

- Automated learning from large datasets without explicit rules.
- High scalability and flexibility across various languages.
- Applications in machine translation, speech recognition, and spell checking.

Disadvantages:

- Limited context: N-gram models consider only a fixed number of words.
- High data requirements: Large text corpora are needed.
- Lack of semantic understanding: These models compute probabilities without capturing meaning.

Despite these drawbacks, probabilistic models were a crucial step towards modern language AI, paving the way for neural networks and deep learning.

21.5 The Rise of Neural Networks from 2000

21.5.1 From Statistics to Deep Learning: A Turning Point in AI

Up to the 1990s, statistical models dominated language processing. By the 2000s, it became clear that these methods were limited—they struggled with complex semantics and required massive datasets.

During this period, neural networks experienced a resurgence. Although the concept of artificial neurons dates back to the 1950s, technological advances in the 2000s made large-scale neural networks practical.

21.5.2 The Renaissance of Neural Networks

1. **Hardware Advances:** Powerful GPUs made training large networks feasible.
2. **Robust Architectures:**
 - Multi-Layer Perceptrons (MLPs) were initially too shallow to capture deep patterns.
 - Recurrent Neural Networks (RNNs), especially LSTMs (introduced in 1997), improved sequence learning for language.
3. **New Training Methods:**
 - Backpropagation with improved optimization algorithms (e.g., Adam from 2014) enabled faster, more stable learning.
 - Dropout techniques (introduced in 2012) helped reduce overfitting.

21.5.3 Deep Learning Takes Over

- **2006:** Geoffrey Hinton and colleagues introduced Deep Belief Networks, showing that deep architectures could outperform traditional methods.
- **2010s:** Deep learning surpassed classical statistical models in almost every AI domain, from image recognition to language processing.

21.5.4 Breakthroughs in Speech Recognition and Machine Translation

- **2011:** Apple introduced Siri, one of the first voice assistants using neural network recognition.
- **2012:** Google Voice began using neural networks, achieving drastic improvements in speech recognition.
- **2016:** AlphaGo defeated the world champion in Go, a major milestone based on deep neural networks.
- **2016:** Google Translate shifted from statistical to neural machine translation, dramatically improving translation quality.

21.5.5 Challenges and Limitations

Neural networks also faced significant challenges:

- High computational costs for training large models.
- Dependence on vast amounts of training data.
- Limited interpretability: Neural networks often act as “black boxes” with unclear decision processes.

These challenges spurred the development of new models, such as Transformers, which set a new standard for language AI from 2017 onward.

21.6 The Vector Representation of Language and Its Significance

21.6.1 From Text to Vectors: A Revolution in Language Processing

Traditionally, language in AI systems was treated as a mere sequence of characters or words. Simple statistical models and early neural networks could not capture complex semantic relationships.

The breakthrough came with the vector representation of language. Words, sentences, or even entire documents are now represented as mathematical vectors in a multidimensional space, fundamentally changing how machines understand and store language.

21.6.2 Word Embeddings: The Foundation

The first major progress in vector representation came with word embeddings:

- **Word2Vec (2013, Google):** The first widely used model to convert words into dense vectors, enabling calculations such as:

$$\text{"king"} - \text{"man"} + \text{"woman"} \approx \text{"queen"}$$
- **GloVe (2014, Stanford):** An alternative based on word co-occurrence statistics, yielding accurate vector representations for semantic similarity.

21.6.3 Representing Sentences and Documents

While word embeddings capture individual words, later models represented whole sentences or paragraphs as vectors. Models such as BERT (2018) or Sentence Transformers emerged:

- **BERT & Transformers:** Provide contextual word vectors that account for a word's meaning within a sentence.
- **Sentence-BERT (SBERT, 2019):** Converts entire sentences into vectors for faster, more accurate semantic search.

21.6.4 Vector Databases: A New Infrastructure

The advent of word and sentence embeddings created the need for efficient databases to store and search high-dimensional vectors:

- **Definition:** Vector databases store data in specialized index structures rather than traditional tables.
- **Notable Examples:**
 - FAISS (Facebook AI Similarity Search)
 - Pinecone
 - Weaviate
 - Milvus

21.6.5 Applications and Importance

Vector representations enable:

- **Semantic Search:** Comparing meanings instead of exact keywords.
- **Recommendation Systems:** Used by platforms like Netflix and Spotify.
- **Chatbots and AI Assistants:** Faster retrieval of responses, as seen in ChatGPT.
- **Plagiarism Detection:** Automated text comparison in academia and publishing.

The combination of neural networks and vector databases now forms the backbone of modern AI language processing.

21.7 The Transformer Revolution (2010–2020): The Rise of Modern Language Models

21.7.1 A Decade of Transformation

Between 2010 and 2020, AI-driven language processing underwent a radical transformation. Early models such as RNNs and LSTMs had their limitations, and the breakthrough came in 2017 with the Transformer architecture, which laid the foundation for modern large language models.

21.7.2 Challenges Before Transformers

- RNNs processed sequences but suffered from vanishing gradients.
- LSTMs, with gating mechanisms, improved on RNNs but still struggled with long-range dependencies.
- Both required sequential processing, which slowed down training.

Researchers needed a solution that preserved long context and allowed for efficient training.

21.7.3 2017: “Attention Is All You Need” — The Birth of the Transformer

In 2017, a Google research team published the groundbreaking paper “Attention Is All You Need”. The Transformer introduced:

- **Self-Attention:** Every word in a sentence relates directly to every other word.
- **Parallel Processing:** Eliminating the need for sequential steps.
- **Scalability:** Efficient training on GPUs allowed for much larger models.

Self-Attention

Instead of looking only at the few preceding words, a Transformer considers all words at once, determining the relevance between each pair via:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q (Query), K (Key), and V (Value) encode word relationships.

21.7.4 Transformers in NLP (2018–2020)

Following the Transformer paper, the architecture was quickly adopted:

- **BERT (2018, Google):** Used Transformers for bidirectional contextual language understanding.
- **XLNet (2019, Google/CMU):** Enhanced BERT with permutation-based training.
- **T5 (2019, Google):** Treated all NLP tasks as text-to-text transformations.
- **RoBERTa (2019, Facebook AI):** An optimized BERT variant with improved pretraining.

21.7.5 Bridging to GPT

While models like BERT were designed as encoders for understanding, OpenAI took a different path:

- **2018:** GPT-1 — The first generative Transformer model.
- **2019:** GPT-2 — Improved autoregressive text generation.
- **2020:** GPT-3 — Revolutionized generative AI with human-like text output.

21.8 The GPT Revolution from 2020: Artificial Intelligence at the Next Level

21.8.1 A Paradigm Shift

From 2020 onward, large generative language models based on Transformers triggered an AI revolution. While the 2010s saw improved language understanding via models like BERT, OpenAI's GPT-3 (2020) introduced a new quality in text generation.

Key differences include:

- GPT models are autoregressive, meaning they generate text as well as understand it.
- They are based on the decoder part of the Transformer, unlike encoder-based models.

This breakthrough spurred new applications in chatbots, automated content creation, AI-assisted programming, and more.

21.8.2 GPT-3: The Breakthrough in Generative AI (2020)

Released in June 2020, GPT-3:

- Boasted 175 billion parameters, an unprecedented scale.
- Performed a wide variety of tasks from translation to code generation.
- Produced text so human-like that it was often indistinguishable from human writing.

Its success was driven by “few-shot learning,” where the model needed only a few examples to perform a task.

21.8.3 ChatGPT: Making AI Interactive (2022)

In December 2022, OpenAI launched ChatGPT, enabling dialogue with large language models. Key features included:

- Reinforcement Learning from Human Feedback (RLHF) to improve safety and utility.
- The ability to remember conversational context.
- Wide accessibility to the public.

Within five days, ChatGPT reached one million users and sparked debates on automation, ethics, and job displacement.

21.8.4 GPT-4: Multimodality and Enhanced Intelligence (2023)

In March 2023, GPT-4 was released with significant improvements:

- **Multimodality:** The ability to understand and describe images.
- **Larger Context Windows:** Enhanced capacity to analyze longer texts.
- **Improved Accuracy:** Fewer factual errors through refined tuning.

Its applications range from coding and medical research to educational tools.

21.8.5 Open-Source and New Competitors (2023–2024)

Alongside proprietary models, the open-source AI community grew:

- **Meta’s LLaMA (2023):** A powerful model accessible for research.

- **Mistral AI (2023):** Smaller but highly efficient models.
- **DeepSeek AI (2024):** New competitors from China offering alternative systems.

Companies such as Google (Gemini), Microsoft (Copilot), and Anthropic (Claude) also expanded their generative AI offerings.

21.8.6 The Next Step: Agents and Multimodal AI (2025)

Development is accelerating:

- **Autonomous AI Agents:** Systems that plan and execute complex tasks independently.
- **Enhanced Multimodality:** Integration of text, images, video, and audio in real time.
- **Personal AI Assistants:** Digital assistants capable of long-term, interactive engagement.

The GPT revolution has transformed our world, and the coming decade promises even greater advancements.

21.9 AI Agents: Autonomous Systems of the Future

21.9.1 From Language Models to Autonomous Agents

While large language models like GPT-3 and GPT-4 generate impressive text, the next step is to develop autonomous AI agents. These agents go beyond simple Q&A systems and can plan, execute, and optimize complex tasks independently.

21.9.2 What is an AI Agent?

An AI agent is a system that:

- Pursues specific goals (e.g., research, coding, process optimization).
- Makes independent decisions based on available information.
- Interacts with external systems via tools and APIs.

They typically include:

- A large language model as the “brain” for processing and decision-making.
- Tools and APIs to access external systems.
- Long-term memory for recalling past tasks or user preferences.
- Planning mechanisms to think several steps ahead.

21.9.3 Existing AI Agents Today

Examples include:

- **Autonomous Research and Writing Agents:**
 - **Auto-GPT (2023)**: The first agent capable of setting its own goals and working autonomously.
 - **BabyAGI (2023)**: An agent that creates and refines long-term plans.
 - **AgentGPT (2023)**: A web-based agent that can independently undertake tasks.
- **AI Agents for Software Development:**
 - **GitHub Copilot X (2023–2024)**: An agent integrated into IDEs for code generation.
 - **Devin (2024, Cognition AI)**: An agent that autonomously programs and debugs.
- **Agents for Research and Science:**
 - **Google DeepMind AlphaFold (2021–2023)**: An agent that predicts protein structures and advances biological research.
 - **Elicit AI (2023)**: An agent that analyzes and summarizes scientific literature.
- **Autonomous Business and Automation Agents:**
 - **Microsoft Copilot for Office (2023–2024)**: AI-powered automation for business processes.
 - **Adept ACT-1 (2023–2024)**: A multimodal agent that interacts with user interfaces.

21.9.4 The Future of AI Agents

Looking ahead, we can expect:

- **Multimodal AI Agents**: Systems processing text, images, audio, and video simultaneously.
- **Agents with True Long-Term Memory**: Models that remember contexts and previous interactions over long periods.
- **Physical AI Agents**: Integration of language models with robotics to perform complex, real-world tasks.
- **Autonomous Economic Systems**: AI agents managing supply chains, marketing strategies, or trading.
- **Fully AI-Driven Developer Teams**: Software agents handling complete projects from planning to implementation.

The coming years will reveal how much autonomy AI agents can truly assume.

21.9.5 Publicly Available Literature on AI Agents

For more on AI agents, consider these resources:

1. Auto-GPT: An introduction to autonomous AI agents (<https://github.com/Torantulino/Auto-GPT>).
2. BabyAGI: A simple implementation of an AI agent with long-term planning (<https://github.com/yoheinakajima/babyagi>).
3. Devin: The first AI agent for software development (<https://www.cognition-labs.com/devin>).
4. AlphaFold: Google's AI agent for protein research (<https://www.nature.com/articles/s41586-021-03819-2>).
5. Elicit AI: Autonomous agents for scientific research (<https://elicit.org/>).
6. Adept ACT-1: Multimodal interaction agents (<https://www.adept.ai/blog/act-1>).
7. GitHub Copilot X: The future of AI-assisted software development (<https://github.com/features/copilot-x>).