Python and AI/ML for Weather, Climate and Environmental Applications

Let us enjoy playing with Python and AI/ML!

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	**Lab:** Python Startup: Basics	**Lab:** Feed-forward and Graph NNs	**Lab:** Graph Learning with PyTorch	**Lab:** Containers and Reproducibility	**Lab:** Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	**Lab:** Work environments, Python everywhere	**Lab:** Simple Transformer and LLM Use	**Lab:** Agent Frameworks	**Lab:** CI/CD Pipelines	**Lab:** Graph-based Assimilation
12:45–13:30			**Lunch Break**		
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	**Lab:** GRIB, NetCDF and Obs Visualisation	**Lab:** RAG Pipeline	**Lab:** DAWID Exploration	**Lab:** Anemoi Training Pipeline	**Lab:** Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	**The AI Transformation**	Learning from Observations Only
16:15–17:00	**Lab:** Torch Tensors and First Neural Net	**Lab:** Radar, SAT and Multimodal Data	**Lab:** MLflow Hands-on	**Lab: How work style could change**	**Lab:** ORIGEN and Open Discussion
17:00–20:00				Joint Dinner	

AI Data Assimilation — Why Does It Matter?

Numerical Weather Prediction (NWP)

Weather forecasts are produced by integrating high-dimensional dynamical models forward in time. However:

- The atmosphere is chaotic
- Small initial errors grow rapidly
- Forecast skill is **dominated by initial conditions**

Core problem

At any analysis time, we have:

- an imperfect model forecast (background)
- sparse, noisy observations

These must be combined optimally.

Data assimilation

Data assimilation provides a statistically consistent framework to merge:

$$\text{model information} \quad + \quad \text{observations}$$

The result is the analysis state:

- best estimate of the atmospheric state
- starting point for forecasts

Data assimilation is the information bottleneck of NWP.

Everything that follows — forecasts, warnings, applications — depends on it.

Classical Data Assimilation: The Analysis Cycle

The classical DA cycle

Operational NWP systems run a repeating assimilation cycle:

1. Start from a background state x_b
2. Assimilate observations y
3. Compute an analysis x_a
4. Run the numerical model forward
5. Use the forecast as next background

This cycle is repeated every few hours as new observations become available.

The analysis step is the only place where observations enter.

Established DA methods

Several algorithmic families are used in practice:

- **3D-Var / 4D-Var**
  - variational optimization
  - adjoint-based
- **Ensemble Kalman Filters (EnKF)**
  - flow-dependent uncertainty
  - ensemble statistics
- **Particle Filters**
  - fully Bayesian
  - now also high-dimensional!

All methods aim at the same goal: a statistically optimal analysis.

Two AI Paths for Using Observations

Path 1: AI-based forecasting

Observations are used directly inside neural networks that produce forecasts.

Typical characteristics:

- observations as additional inputs
- sometimes no explicit model state
- learning focuses on **prediction skill**

This approach bypasses the classical analysis concept.

Observations → forecast directly

Path 2: AI-based data assimilation

Observations are used to compute an analysis state, not a forecast.

Key properties:

- preserves the analysis–forecast separation
- consistent with Bayesian DA theory
- AI replaces the analysis algorithm, not the model

Observations → analysis → forecast

This is the conceptual space of **AI-Var**.

Why Bring AI into Data Assimilation?

Limits of classical DA

State-of-the-art data assimilation systems are:

- computationally expensive
- difficult to scale to higher resolution
- reliant on adjoint models

Operational challenges include:

- complex model development and maintenance
- long wall-clock times
- limited flexibility for new observation types

DA is often the most expensive component of NWP.

What AI can offer

Modern neural networks provide:

- fast inference once trained
- automatic differentiation
- flexible nonlinear mappings

Potential benefits for DA:

- orders-of-magnitude speedup
- end-to-end differentiability
- easier adaptation to new data streams

Goal: replace the solver, not the statistics.

From Variational Data Assimilation to AI-Var

Variational DA: main Idea

The analysis is obtained by minimizing:

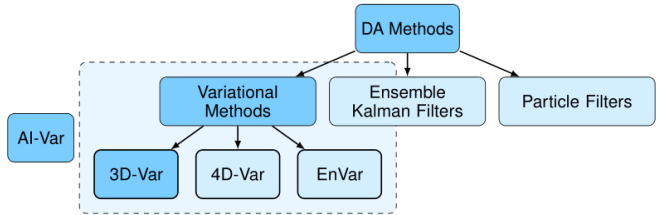
$$J(x) = \frac{1}{2}(x-x_b)^T B^{-1}(x-x_b) + \frac{1}{2}(y-H(x))^T$$

Interpretation

- first term: background constraint
- second term: observation constraint

Classical DA = iterative numerical minimization.

AI-VAR is the AI verion of **3D-Var, 4D-Var or En-VAR** depending on how exactly the minimizer is formulated.



AI-Var sits inside the variational DA family, alongside 3D-Var, 4D-Var, and EnVar.

Core Idea of AI-Var

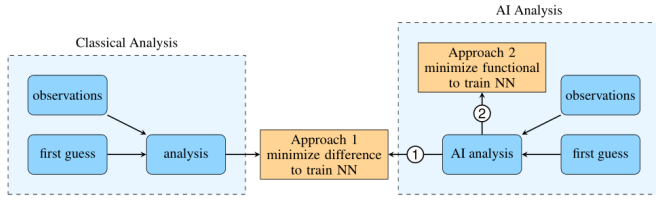
Classical variational DA

- iterative minimization of $J(x)$
- requires adjoints and solvers
- expensive and sequential

Replace the minimization algorithm.

AI-Var

- neural network approximates the minimizer
- outputs analysis x_a directly
- trained using the same cost function



Classical DA workflow (left) versus AI-based inference of the analysis (right).

Paradigm shift

- DA becomes inference, not optimization
- milliseconds instead of iterations
- same statistics, different machinery
- flexible inflow of further information

AI-Var: Architecture, Loss, and Learning

Architecture

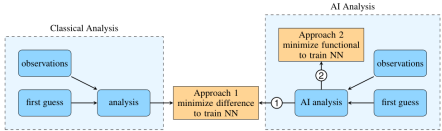
- inputs: background x_b and observations y
- neural network outputs analysis $\hat{x}_a$

Training loss

$$L = (\hat{x}_a - x_b)^T B^{-1} (\hat{x}_a - x_b) + (H(\hat{x}_a) - y)^T R^{-1} (H(\hat{x}_a) - y)$$

- B^{-1} and R^{-1} weight uncertainties
- observation operator H is inside the loss

Statistics and physics are embedded in the loss.



What is learned

- mapping $(x_b, y) \rightarrow x_a$

What is not learned

- no reanalysis targets
- no iterative solvers
- no adjoint code

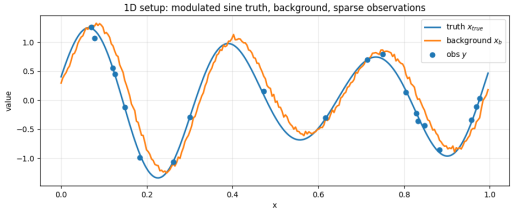
Optimization is replaced by inference.

1D Data Assimilation Setup

Toy problem

We consider a 1D state on a fixed grid, periodic (!).

- truth: modulated sine function
- background x_b: shifted, smoothed, biased
- observations y: sparse, noisy point samples



Truth x_{true}, background x_b, and sparse observations y.

All ingredients are fully controlled:

- known truth
- known error statistics
- explicit observation locations

Ideal testbed to explore data assimilation.

Classical 3D-Var in 1D: Mathematics Made Explicit

3D-Var cost function

The analysis x_a minimizes $J(x)$ from above.

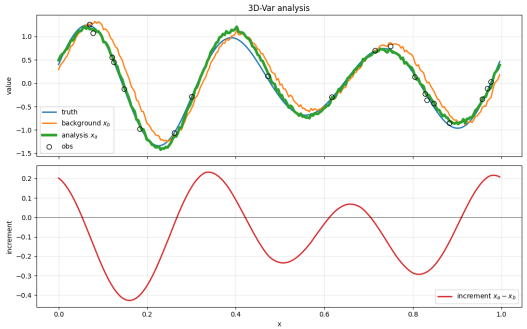
Linear case (this setup)

- H: point-sampling operator
- B: Gaussian covariance
- $R = \sigma_o^2 I$

Closed-form solution

$$x_a = x_b + BH^T(HBH^T + R)^{-1}\big(y - Hx_b\big)$$

This is the optimal Bayesian estimate.



Top: state space — truth, background, analysis.
Bottom: analysis increment

$$\delta x = x_a - x_b$$

Information from sparse observations is spread by B.

AI-Var in 1D: From Mathematics to Code

Neural increment model

The network predicts the analysis increment:

$$\delta x_\theta = \mathcal{N}_\theta(x_b, y)$$

PyTorch implementation (kept simple)

```
class IncrementMLP(nn.Module):
    def __init__(self, n):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3*n, 256),
            nn.Tanh(),
            nn.Linear(256, 256),
            nn.Tanh(),
            nn.Linear(256, n))

    def forward(self, inp):
        return self.net(inp)
```

Input vector:

$$\texttt{inp} = [x_b,\ y_{\text{grid}},\ \text{mask}]$$

Variational loss in code

The 3D-Var cost is used directly:

```
def J_3dvar(delta_x):
    x = xb + delta_x
    innov = y - H @ x

    Jb = 0.5 * (delta_x @ B_inv @ delta_x)
    Jo = 0.5 * (innov @ R_inv @ innov)
    return Jb + Jo
```

No analysis data. No solver. Only the variational objective.

AI-Var in 1D: Training Loop with Background & Observations

Neural input construction

The network receives both background and observations:

```
# observations mapped to grid
y_grid = torch.zeros(n)
mask   = torch.zeros(n)

y_grid[obs_idx] = y
mask[obs_idx]   = 1.0

# NN input
inp = torch.cat([xb_t, y_grid, mask])
```

$$\delta x_\theta = \mathcal{N}_\theta(x_b,\, y)$$

The solver is replaced by learning.

Training loop

```
for ep in range(n_epochs):
    optimizer.zero_grad()

    delta_x = model(inp)

    loss, Jb, Jo = J_3dvar(delta_x)

    loss.backward()
    optimizer.step()
```

Key points

- loss = classical 3D-Var functional
- no analysis targets
- gradients pass through B^{-1} and H

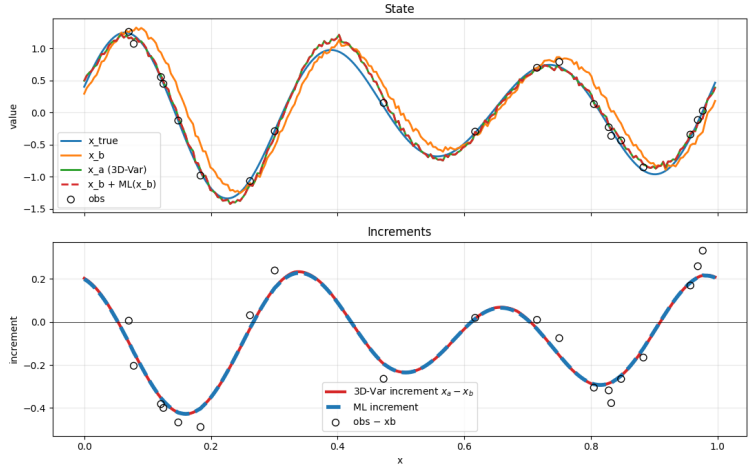
1D Result: AI-Var Analysis vs 3D-Var

What is shown

- true state x_{true}
- background x_b
- 3D-Var analysis x_a
- AI-Var analysis $x_b + \delta x_{\mathrm{ML}}$

Bottom panel

- classical 3D-Var increment
- learned AI-Var increment
- observation increments at obs points



AI-Var reproduces the variational update.

AI-Var in 1D: Training on Many Cases

From single case to ensemble training

Instead of one fixed (x_b, y) pair, we train on many randomly generated cases.

Each training sample contains:

- a new truth x_{true}
- a new background x_b
- new observation locations and values y

What stays fixed

- background covariance B
- observation error R
- variational cost J

The network learns a general DA operator.

Training logic (conceptual)

```
for sample in training_set:
    xb, y = sample
    inp = build_input(xb, y)

    delta_x = model(inp)
    loss = J_3dvar(delta_x)

    loss.backward()
    optimizer.step()
```

Same loss, many realizations.

1D Generalization: Many Unseen Test Cases

Test phase

The trained AI-Var network is applied to previously unseen cases.

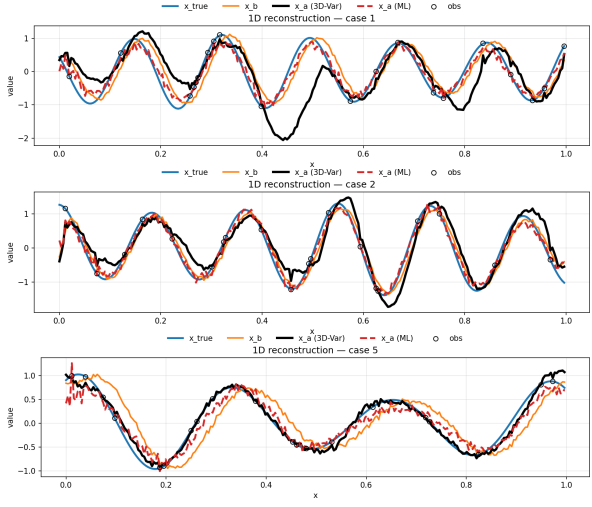
For each case:

- different truth
- different background
- different observations

Observation

- consistent increments
- smooth, physical updates
- no retraining required

One network, many analyses.



2D Data Assimilation Setup

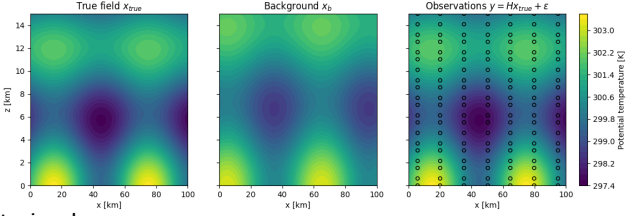
2D atmospheric toy problem

- horizontal–vertical grid
- structured background errors
- sparse column observations

State variables

- truth $x_{true}(x, z)$
- background $x_b(x, z)$
- analysis $x_a(x, z)$

This already resembles NWP geometry.

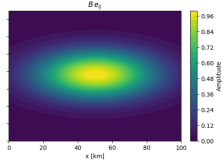


2D setup in code

```
# 2D grid (x,z)
nx, nz = 120, 40
x = np.linspace(0, Lx, nx)
z = np.linspace(0, Lz, nz)

# background covariance (separable)
B = Bx x Bz

# sparse vertical profiles
H : (n_obs × n_state)
```



B matrix in 2d, Be_{ij}

Same DA ingredients as in 1D, now on a 2D grid.

2D AI-Var: Explicit Spectral B^{-1} in Flattened Control Space

2D control vector

The 2D field is represented as a 1D control vector:

$$x \in \mathbb{R}^n, \qquad n = n_x n_z$$

```
nz, nx = xb.shape
n = nz * nx

X_flat = X.reshape(-1)
Z_flat = Z.reshape(-1)
```

Full Gaussian background covariance

```
dx2 = (X_flat[:,None]-X_flat[None,:])**2
dz2 = (Z_flat[:,None]-Z_flat[None,:])**2

B = sigma_b**2 * np.exp(
   -0.5*(dx2/Lx**2 + dz2/Lz**2)
)
B = 0.5*(B + B.T)
```

Spectral regularization

```
lam, U = np.linalg.eigh(B)

lam_floor = alpha * lam.max()
lam_reg   = np.maximum(lam, lam_floor)

Breg_inv = (U * (1.0/lam_reg)) @ U.T
Breg_inv = 0.5*(Breg_inv + Breg_inv.T)
```

Background term in the loss

$$J_b(\delta x) = \frac{1}{2}\,\delta x^T\, B^{-1}\,\delta x$$

```
Jb = 0.5 * dx @ Breg_inv @ dx
```

Key point: B^{-1} is built explicitly in this tutorial.

2D AI-Var (Tutorial Code): Flattened Control + Obs Encoding + 3D-Var Loss

Inputs: background + obs on grid + mask

```
# flatten 2D -> 1D control
nz, nx = xb.shape
n = nz * nx
xb_t = torch.tensor(xb.reshape(-1), dtype=dtype, device=device)

# obs indices (iz-major, ix-minor)
obs_indices = iz_idx * nx + ix_idx
y_vec = y_field.reshape(-1)[obs_indices]
y_t   = torch.tensor(y_vec, dtype=dtype, device=device)

# obs on grid + mask (length n)
y_grid = np.zeros(n);  mask = np.zeros(n)
y_grid[obs_indices] = y_vec
mask[obs_indices]   = 1.0

inp_t = torch.cat([xb_t,
       torch.tensor(y_grid, dtype=dtype, device=device),
       torch.tensor(mask,   dtype=dtype, device=device)], dim=0)
```

Input is $[x_b,\ y_{\text{grid}},\ \text{mask}]$ in control space.

Increment MLP + variational loss

```
class IncrementMLP(nn.Module):
    def __init__(self, n):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3*n, 256), nn.Tanh(),
            nn.Linear(256, 256), nn.Tanh(),
            nn.Linear(256, n),
        )

def quadform(A, v): return torch.dot(v, A @ v)

def J_3dvar(dx):
    x = xb_t + dx
    innov = y_t - (H_t @ x)
    Jb = 0.5 * quadform(B_inv_t, dx)
    Jo = 0.5 * quadform(R_inv_t, innov)
    return Jb + Jo
```

Exactly the classical 3D-Var objective, now differentiable.

2D Result: Variational Reference vs Learned Analysis

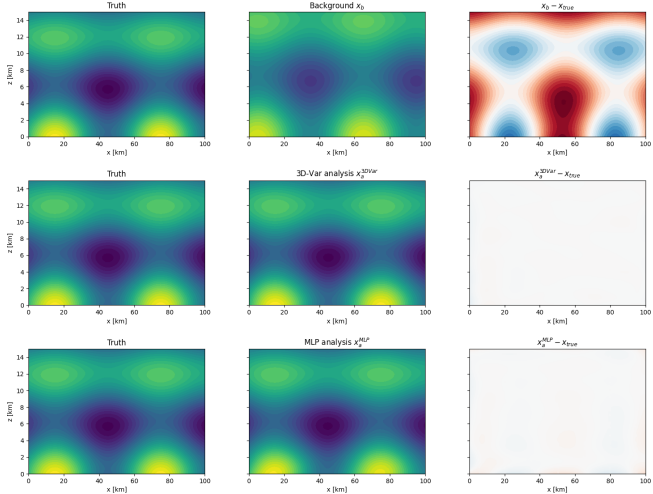
What is compared

- background x_b
- 3D-Var analysis x_a
- AI-Var analysis $\hat{x}_a$

Key observation

- smooth spatial increments
- correct information spreading
- close to variational solution

Learned analysis = inference of the minimizer.



AI Particle Filter (Gaussian Mixture): Core Idea

Filtering problem
Sequential posterior:

$$p(x_n \mid y_{1:n})$$

Represent the distribution by particles

$$X_n = \{x_n^{(i)}\}_{i=1}^N$$

AI Particle Filter (this tutorial)
Forecast ensemble X_n^b is transformed into analysis ensemble:

$$X_n^a = \mathcal{N}_\theta(X_n^b,\ y_n)$$

Gaussian mixture view
Both prior and posterior are approximated by mixtures:

$$q^b(x) \approx \frac{1}{N}\sum_i \mathcal{N}(x|x^{b,(i)}, \Sigma), \qquad q^a(x) \approx \frac{1}{N}\sum_i \mathcal{N}(x|x^{a,(i)}, \Sigma)$$

Model problem: Lorenz-63

- state $x = (x_1, x_2, x_3) \in \mathbb{R}^3$
- partial observations $y = (x_1, x_2) +$ noise
- wrong forecast model (intentional)

Key message
Instead of resampling / MCMC moves, we learn a distribution transform that fits the ensemble to the posterior.

Network output is a posterior particle cloud.

Neural Particle Update: DeepSets (Permutation Invariance)

Particle set input

Forecast ensemble is an unordered set:

$$X^b = \{x_i^b\}_{i=1}^N$$

Update must be permutation invariant:

$$\mathcal{N}_\theta(\pi X^b,\ y) = \pi\, \mathcal{N}_\theta(X^b,\ y)$$

DeepSets structure

$$\phi(x_i, y) \ \rightarrow\ \text{mean pool} \ \rightarrow\ \rho(\cdot) \ \rightarrow\ \psi(x_i, \text{context}, y)$$

Each particle sees local state + global context + obs.

```
class ParticleUpdateNN(nn.Module):
  def forward(self, Xb, y):
    N = Xb.shape[0]
    y_rep = y.expand(N, -1)

    emb = phi(torch.cat([Xb, y_rep], 1))
    pooled = emb.mean(0, keepdim=True)
    ctx = rho(pooled).expand(N, -1)

    dX = psi(torch.cat([Xb, ctx, y_rep], 1))
    return Xb + dX
```

Output: updated ensemble X^a.

Training Loss: Fit a Gaussian-Mixture Posterior

Posterior fitting objective
Train the particle update network such that the analysis ensemble represents the posterior:

$$q^a(x) \approx p(x \mid y)$$

Gaussian mixture model
Particles define a mixture density:

$$q(x \mid X) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{N}(x \mid x^{(i)}, \Sigma)$$

Loss = likelihood + KL fit

$$L = \underbrace{-\log\left(\frac{1}{N}\sum_i p(y \mid x_i^a)\right)}_{L_{\text{obs}}} + \lambda_{\text{bg}} \underbrace{\mathrm{KL}\Big(q_{\text{target}}(\cdot) \,\|\, q^a(\cdot)\Big)}_{L_{\text{GM}}}$$

```
# evaluation points (single Kalman update)
Z = kalman_eval_points(Xb, y)

# target posterior weights on Z
log_t = log_mix(Z|Xb) + log_like(y|Z)
w_t = softmax(log_t)

# model mixture log-density on Z
log_m = log_mix(Z|Xa)

# mixture KL term
L_GM = sum_z w_t(z) * (log w_t(z)
- log softmax(log_m(z)))
loss = L_obs + lambda_bg * L_GM
```

Target uses prior mixture + obs likelihood.

Learn the posterior distribution, not only the mean.

Gaussian-Mixture PF: Comparing Two Distributions (Math)

1) Ensembles as Gaussian mixtures

Forecast particles $X^b = \{x_i^b\}_{i=1}^N$ and analysis particles $X^a = \{x_i^a\}_{i=1}^N$ define mixture densities, e.g.:

$$q_\theta^a(x) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{N}(x \mid x_i^a, \Sigma)$$

2) Discretize comparison

Choose evaluation points $Z = \{z_k\}_{k=1}^K$ (from a Kalman-type proposal step in the notebook).

Target posterior on Z

$$w_k^\star = \frac{q^b(z_k)\, p(y \mid z_k)}{\sum_{\ell=1}^K q^b(z_\ell)\, p(y \mid z_\ell)}$$

Prior mixture + obs likelihood $\Rightarrow$ posterior weights.

3) Model distribution on Z

Evaluate the analysis mixture on the same points and normalize:

$$\pi_{\theta,k} = \frac{q_\theta^a(z_k)}{\sum_{\ell=1}^K q_\theta^a(z_\ell)} = \mathrm{softmax}\big(\log q_\theta^a(z_k)\big)$$

4) Fit posterior mass distribution

Training minimizes the discrete KL divergence:

$$L_{\mathrm{GM}} = \mathrm{KL}(w^\star \parallel \pi_\theta) = \sum_{k=1}^{K} w_k^\star \, \log \frac{w_k^\star}{\pi_{\theta,k}}$$

Network learns to move particles toward posterior mass.

Ensemble Geometry: Prior → Analysis in a 2D Slice

What the plot shows
2D slice of the state space (e.g. x_1–x_2 plane).

Prior (forecast ensemble)

- particles X^b (cloud)
- prior mean $\bar{x}^b$

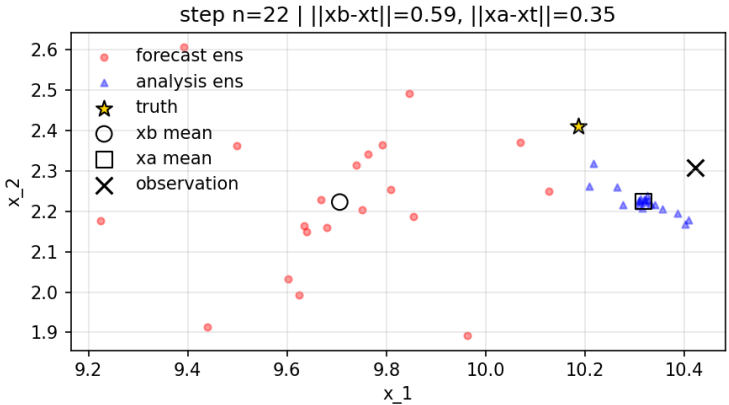
Observation / truth

- observation y (marker)
- truth x_{true} (marker)

Analysis ensemble

- updated particles X^a
- analysis mean $\bar{x}^a$

Ensemble scatter in x_1–x_2 plane



Prior ensemble (forecast) and analysis ensemble after the learned update.

Ablation: Why the Background / Posterior-Fit Term Matters

Two trained networks

- **net** : full AI-PF loss
- **net_obs** : obs term only

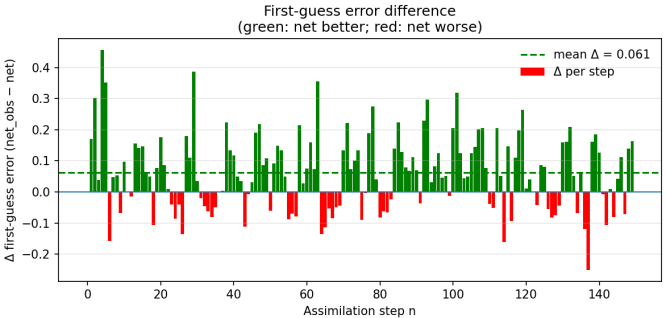
Metric shown

Difference of first-guess error:

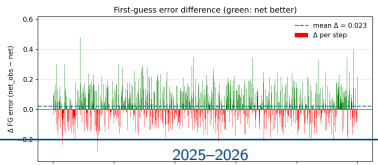
$$\Delta = \mathrm{FG_{err}}(\texttt{net\_obs}) - \mathrm{FG_{err}}(\texttt{net})$$

Interpretation

- $\Delta > 0$ (green) : net is better
- $\Delta < 0$ (red): net_obs is better



Evaluation over 10 000 assimilation cycles after short training.



Summary: Two AI Paths for Data Assimilation

AI-Var (Keller & Potthast)

What it learns:

$$(x_b,\ y) \mapsto x_a$$

- neural network approximates the minimizer
- trained by variational cost $J(x)$
- output = one deterministic analysis field

Strengths

- stable, structured increments
- scalable in dimension (1D $\rightarrow$ 2D demo)
- direct link to operational 3D/4D-Var

AI Particle Filter (Gaussian mixture)

What it learns:

$$(X^b,\ y) \mapsto X^a$$

- neural update transforms particle cloud
- trained to fit posterior mass distribution
- output = analysis distribution (ensemble)

Strengths

- non-Gaussian posteriors (multi-modal)
- distribution-aware filtering
- background-term ablation shows skill gain

Take-home: AI-Var learns the analysis; AI-PF learns a posterior distribution.