Preliminary Thesis Report
Probabilistic and Geometry-Aware Operator Learning for Scrape-Off Layer Plasma Modeling

Gabriele Gianuzzo

February 25, 2026

1 Adding Geometry to the Input Tensor

When you enable geometry (the `--include_geometry` option in the tensor builder), you modify the *input* tensor X by appending spatial coordinate channels computed from the fixed SOL grid geometry (`crx.npy`, `cry.npy`). **The output tensor Y is unchanged.**

Baseline Tensor Without Geometry

For each simulation n, the global input tensor has shape:

$$X^{(n)} \in \mathbb{R}^{C_{\text{in}} \times H_g \times W_g}$$

and follows this channel definition:

- Channel 0: global valid-cell mask $m \in \{0,1\}^{H_g \times W_g}$ (1 on mapped cells, 0 in gaps).
- Channels $1 \dots 8$: the 8 scalar conditioning parameters from `X_tmp.npy`, broadcast to every valid pixel:
$$X^{(n)}_{k,i,j} = x^{(n)}_k \quad \text{for } k \in \{1, \dots, 8\} \text{ and } m_{i,j} = 1$$
Values in gap pixels ($m_{i,j} = 0$) are kept at zero.

So, without geometry:

$$C_{\text{in}} = 1 + 8 = 9$$

With geometry enabled, you append 2 additional channels that encode the *spatial location* of each valid pixel in the global canvas:

- `centroid_x`(i, j): the physical x-coordinate of the cell centroid mapped to global pixel (i, j)
- `centroid_y`(i, j): the physical y-coordinate of the cell centroid mapped to global pixel (i, j)

So, with geometry:

$$C_{\text{in}} = 1 + 8 + 2 = 11$$

How the Geometry Channels Are Computed

Each original logical cell (i_x, i_y) is a quadrilateral with 4 corners (x_k, y_k), stored in `crx[ix,iy,k]` and `cry[ix,iy,k]`. You compute the centroid by averaging the 4 corners:

$$x_c(i_x, i_y) = \frac{1}{4}\sum_{k=1}^{4} x_k(i_x, i_y), \qquad y_c(i_x, i_y) = \frac{1}{4}\sum_{k=1}^{4} y_k(i_x, i_y)$$

Then, using the unrolled-strip mapping (pixel → logical cell index), you paste these centroid values into the global rectangular canvas at the corresponding pixel positions, exactly like you paste physical fields. Gap pixels remain zero and are excluded by the mask.

2 Architecture

```
UNetSmall(c_in, c_out, base=32)

# Encoder blocks (each block: Conv3x3 + ReLU + Conv3x3 + ReLU)
enc1 = ConvBlock(c_in        -> base)
enc2 = ConvBlock(base        -> 2*base)
enc3 = ConvBlock(2*base      -> 4*base)

pool = MaxPool2D(kernel=2, stride=2)

# Bottleneck
bottleneck = ConvBlock(4*base -> 8*base)

# Decoder: upsample + 1x1 conv (channel reduce) + concat skip + ConvBlock
up3  = Conv1x1(8*base -> 4*base)
dec3 = ConvBlock((4*base + 4*base) -> 4*base)

up2  = Conv1x1(4*base -> 2*base)
dec2 = ConvBlock((2*base + 2*base) -> 2*base)

up1  = Conv1x1(2*base -> base)
dec1 = ConvBlock((base + base) -> base)

# Output projection
out = Conv1x1(base -> c_out)
```

3 Differences Between `train_cnn.py` and `train_unet.py`

Both scripts use the same `UNetSmall` architecture. The differences are entirely in the *objective formulation*, *target transformation*, and *normalization strategy*.

1. Target Transformation

train_cnn.py

All output channels are transformed uniformly:

$$y_{\text{trans}} = \log_{10}\big(\max(y, 0) + 10^{-3}\big)$$

This is applied to:

- Electron temperature
- Ion temperature
- All densities
- All velocities

Limitation: Velocities are signed quantities. Applying $\log_{10}$ destroys sign information and distorts their distribution.

train_unet.py

Channels are separated into two categories:

Positive channels (temperature + densities)

$$y_{\text{trans}} = \log_{10}(\max(y, 0) + \varepsilon)$$

Signed channels (velocities)

$$y_{\text{trans}} = \operatorname{asinh}\left(\frac{y}{s_c}\right)$$

where

$$s_c = \text{masked standard deviation of channel } c$$

Advantage:

- Preserves velocity sign
- Symmetric handling of positive and negative values
- More stable gradients

2. Target Normalization

train_cnn.py

- Only input X is normalized.
- Targets Y are transformed but *not normalized per channel.*

Consequence: Channels with larger variance dominate the loss.

train_unet.py

After transformation, each channel is normalized:

$$\tilde{y}_c = \frac{y_{\text{trans},c} - \mu_c}{\sigma_c}$$

where μ_c and σ_c are computed in masked space.
Advantage:

- Equal scale across channels
- Prevents domination by large-variance outputs
- Improves optimization stability

3. Loss Function Formulation

train_cnn.py

Masked MAE aggregated over all pixels and channels:

$$\mathcal{L} = \frac{\sum_{b,c,i,j} |\hat{y}_{b,c,i,j} - y_{b,c,i,j}| \, m_{b,i,j}}{\sum_{b,c,i,j} m_{b,i,j}}$$

Effect: Channels with more active pixels or larger magnitude influence the loss more strongly.

train_unet.py

Loss computed per channel, then averaged:

$$\mathcal{L} = \frac{1}{C} \sum_{c=1}^{C} \left(\frac{\sum_{b,i,j} |\hat{y}_{b,c,i,j} - y_{b,c,i,j}| \, m_{b,i,j}}{\sum_{b,i,j} m_{b,i,j}} \right)$$

Advantage:

- Each channel contributes equally
- Prevents temperature from dominating velocity training
- More balanced multi-task learning

4. Evaluation Metrics

train_cnn.py

- Reports global MAE and RMSE
- Aggregated across all channels

train_unet.py

- Reports per-channel MAE and RMSE
- Also reports global averages

Advantage: Enables detailed error diagnostics for each physical quantity.

5. Physical Consistency

train_cnn.py

- Deterministic surrogate
- No distinction between signed and positive physics
- Loss biased toward large-scale fields

train_unet.py

- Physically aware transformation
- Balanced multi-channel objective
- Correct handling of signed velocities
- More stable training across regimes

Summary

Feature	train_cnn	train_unet
Architecture	UNetSmall	UNetSmall
Velocity handling	log transform	asinh transform
Per-channel normalization	No	Yes
Channel-balanced loss	No	Yes
Per-channel metrics	No	Yes
Physics awareness	Low	High

Follow-Up Questions for Discussion

1. Architecture and Model Design

- Is the current architecture too weak? should I focus on improving architecture or improving normalization, adding geometries and thing like that?
- Does the rectangular unrolled representation introduce geometric distortions?
- Is joint prediction of all 22 channels optimal, or would separating temperature, density, and velocity improve stability?

2. Loss Function and Physical Consistency

- Should physical constraints (e.g. positivity of density, symmetry, conservation properties) be explicitly enforced in the loss?
- Would adding gradient regularization improve boundary layer behavior near the divertor?
- Is equal channel weighting physically justified, or should weights reflect physical relevance (e.g. heat flux importance)?

3. Geometry Representation

- Is adding centroid coordinates sufficient to encode geometry, or should geometry be represented in a more structured way (e.g. metric tensors or field-aligned coordinates)?

4. Evaluation Strategy

- Are MAE and RMSE sufficient, or should I change metrics? if yes what metrics should I approach?

5. Strategic Question

- Given the current deterministic U-Net baseline, what would be the most scientifically meaningful next step?