Ground-Displacement Forecasting from Satellite Image Time Series via a Koopman-Prior Autoencoder

Supplementary Material

6. Dataset

6.1. Train/Val/Test

Figure 6 schematically summarizes the three chronological split strategies (A, B, C) used in our experiments. The top blue bar represents the full, ordered sequence of SBAS frames; the coloured segments beneath it indicate how each strategy partitions those frames into training (blue), validation (orange), and test (green) sets.

Strategy A (30/30/20). This strategy is used for long context length. The first 30 frames are used for training, the next 30 frames for validation, and the final 20 frames for testing.

Strategy B (20/40/20). To investigate a baseline context length, we allocate the first 20 frames to training, the subsequent 40 frames to validation, and keep the same 20-frame test window as in Strategy A. Within this split, we train the model with

Strategy C (10/50/20). Here, the shortest training context remains 10 frames, while the validation horizon is extended to 50 frames; the last 20 frames again serve as a fixed test set.

Across all three strategies, the test portion (right-most green segment) is identical, guaranteeing that every model is evaluated on exactly the same 20 unseen frames. The differing sizes of the training and validation windows allow us to study how context length and prediction horizon affect learning dynamics and generalisation performance.

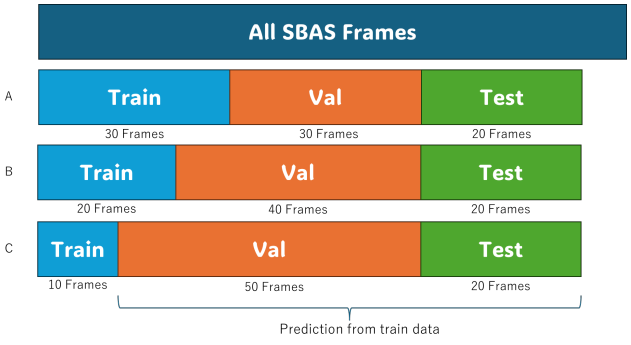

Figure 6. Data splitting strategies for training, validation, and testing. We train a model using only frames of training data, and using that learned model, we predict frames of validation and testing data.

6.2. SBAS dataset in Japan

We evaluate our method using the Small Baseline Subset (SBAS) dataset, which consists of time-series synthetic aperture radar (SAR) interferograms capturing ground deformation over a specified region. The SBAS technique reconstructs surface displacement by minimizing temporal and spatial decorrelation effects, making it well-suited for studying geophysical phenomena such as seismic activity and land subsidence.

For our experiments, we use SBAS-derived displacement maps with a spatial resolution of X meters and a temporal resolution of Y days, covering a total of Z frames. The dataset provides a rich source of spatiotemporal patterns, enabling us to assess the effectiveness of our KPA model in forecasting surface deformation.

6.3. Out of distribution dataset

For training, we utilize the SBAS dataset from Japan. To evaluate the robustness of our method, we test it on SBAS data from regions not included in training, specifically Turkey, Italy, and Hawaii. This cross-region evaluation allows us to assess the generalization capability of our approach in diverse geophysical settings.

Figure 7 illustrates the mean displacement velocity of the SBAS data. The points in the figure represent sampled locations corresponding to the evaluation sites discussed in the text.

7. Method

7.1. Koopman Layer

The latent state $\mathbf{z}_n \in \mathbb{R}^{N_l}$ resides in the bottleneck (cyan); a *single* learnable matrix $\mathbf{K} \in \mathbb{R}^{N_l \times N_l}$ then advances the dynamics:

$$\mathbf{z}_{n+1} = \mathbf{K}\,\mathbf{z}_n, \qquad \mathbf{z}_{n+k} = \mathbf{K}^k\,\mathbf{z}_n \;(k \geq 1). \tag{12}$$

Because $\mathbf{K}$ is declared as a parameter of the network it is updated by back-propagation alongside all convolutional weights. In PyTorch the layer can be written concisely:

```
import torch, torch.nn as nn

class KoopmanLayer(nn.Module):
    def __init__(self, latent_dim: int):
        super().__init__()
        # initialise K as (noisy) identity;
            requires_grad=True by default
        self.K = nn.Parameter(torch.eye(
            latent_dim))
```

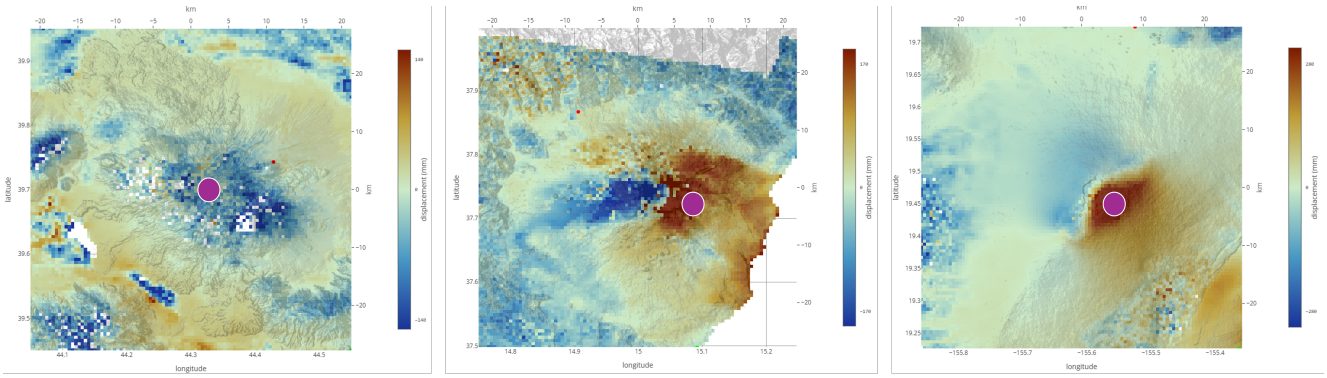

Figure 7. SBAS data for evaluating the robustness of our trained model.

```
    def forward(self, z, steps: int = 1):
        """Propagate latent code z forward
           by `steps`."""
        if steps == 1:
            return z @ self.K.T
        Kk = torch.matrix_power(self.K,
           steps)
        return z @ Kk.T
```

Here `self.K` is a fully learnable weight matrix; the call `z @ self.K.T` realises the update $\mathbf{z}_{n+1} = \mathbf{K}\mathbf{z}_n$. Multi-step propagation uses `torch.matrix_power` to obtain $\mathbf{K}^k$ without explicit loops, keeping the operation both GPU-friendly and differentiable.

8. Experimental Results

8.1. Model Efficiency Analysis

Comparison of Model Architectures for Time Series Image Prediction. In this supplementary material, we provide a detailed analysis of the computational efficiency of our proposed model compared to conventional ConvLSTM and Transformer-based approaches for the task of predicting the next frame ($64 \times 64 \times 1$) from time series image data ($64 \times 64 \times 20$ch).

Parameter Count Comparison Table 4 shows the parameter count for each model architecture with default hyperparameter settings. Our model achieves a significant reduction in parameter count compared to traditional approaches. Specifically, our model requires only 0.2 million parameters, which is approximately 50 times fewer than ConvLSTM (10M) and 140 times fewer than Transformer-based architectures (28M). This dramatic reduction in model size is primarily attributed to the efficient Koopman operator representation and the utilization of spectral methods through FFT, which enables capturing complex dynamics with fewer parameters.

Table 4. Parameter Count Comparison

Model	Parameter Count	Relative Size
Ours	0.2M	1×
ConvLSTM	10M	50×
Transformer	28M	140×

Computational Complexity Analysis. Table 5 presents the theoretical computational complexity in terms of floating point operations (FLOPs) required for a single forward pass with batch size 1. The computational advantage of our approach is particularly evident when compared to Transformer-based models, requiring approximately 1,647 times fewer operations. Our model also offers a 4-fold reduction in computational requirements compared to ConvLSTM architectures, making it significantly more efficient for real-time applications and deployment on resource-constrained devices.

Table 5. Computational Complexity Comparison

Model	FLOPs	Relative Computation
Ours	19M	1×
ConvLSTM	76.5M	4×
Transformer	31.3G	1,647×

Inference Time Estimation. Table 6 provides estimated inference times on modern hardware for a single input sample. These measurements demonstrate that Ours not only has theoretical efficiency advantages but also translates to practical performance benefits, with approximately 5-10× faster inference times on both GPU and CPU hardware compared to Transformer models.

Table 6. Estimated Inference Time Comparison

Model	GPU Time (RTX 3080)	CPU Time
Ours	1-3 ms	10-30 ms
ConvLSTM	5-10 ms	50-100 ms
Transformer	15-30 ms	200-400 ms