AlphaEarth Foundations

Explainer & Implementation by Brayden Zhang

Summary

- AEF is a geospatial foundation model that produces embedding fields, which can then be used for downstream geographical/environmental applications
- Key Features
  - Multisource integration
  - Space-Time-Precision Encoder
  - Time-conditional summarization
  - vMF bottleneck for embeddings
  - Multi-term loss: reconstruction, batch uniformity, teacher-student consistency, CLIP alignment.

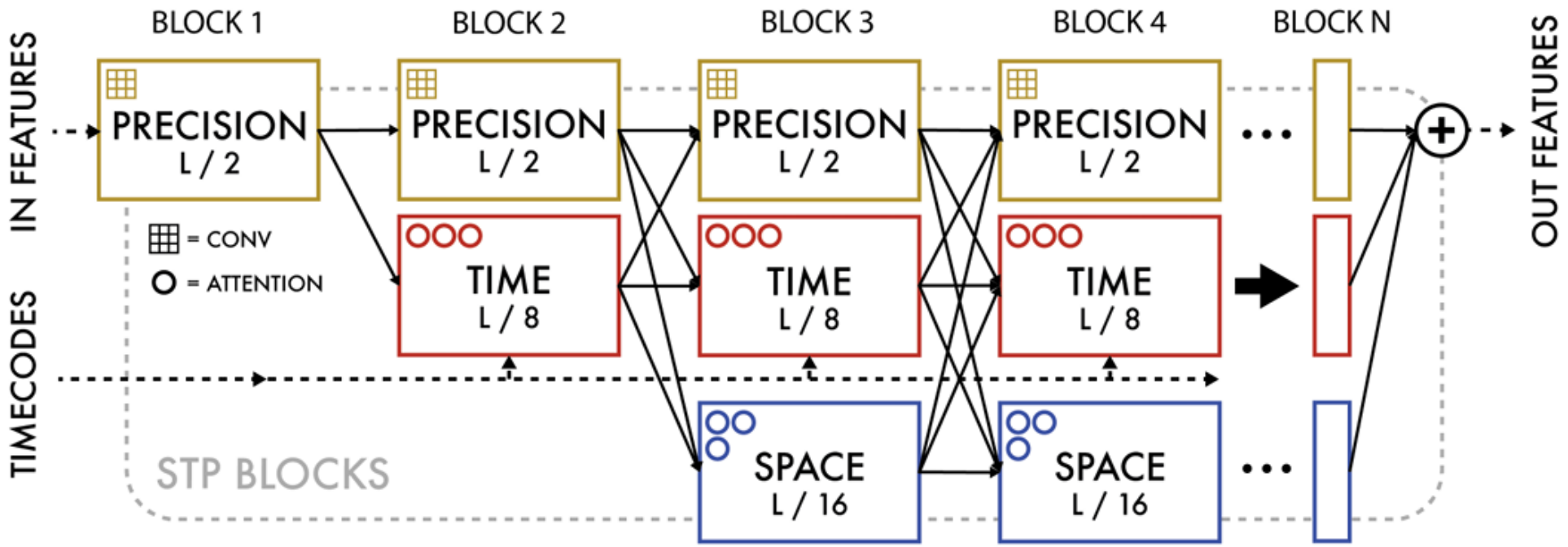
Data

- Data Types
  - Sentinel-2 (optical, 13 bands, 10–60m).
  - Sentinel-1 (radar, dual-pol).
  - Landsat 8/9.
  - ERA5 climate reanalysis.
  - GEDI LiDAR (vertical canopy profiles).
  - Ancillary labels: Wikipedia text.

- Each training sample:
  - Support period: multi-frame inputs (up to 1 year).
  - Valid period: target interval for summarization.
  - Spatial patch: e.g., 128x128 px.
  - Output: per-pixel embeddings (B,H,W,64).

Encoder

D


BLOCK 1
BLOCK 2
BLOCK 3
BLOCK 4
BLOCK N
IN FEATURES
TIMECODES
OUT FEATURES
PRECISION
L / 2
TIME
L / 8
SPACE
L / 16
= CONV
= ATTENTION
STP BLOCKS

Space Operator

- **Goal**: to capture long-range spatial dependencies inside each frame
- Input: $x \in \mathbb{R}^{B \times T \times H \times W \times C}$
- Flatten Spatial Grid $x_{\text{flat}} \in \mathbb{R}^{(BT) \times (HW) \times C}$

$$[Q, K, V] = x_{\text{flat}} W_{qkv}, \quad W_{qkv} \in \mathbb{R}^{C \times (3 \cdot C)} \qquad Q, K, V \in \mathbb{R}^{(BHW) \times \text{heads} \times T \times d}$$

- Perform (per-frame) MHSA
- Residual + MLP

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V$$

- Output: (B, T, H, W, C) at 1/16L resolution

Time Operator

- Goal: to capture temporal evolution of each pixel over many frames
- Input: $x \in \mathbb{R}^{B \times T \times H \times W \times C}$
- Add Sinusoidal temporal encodings

$$x_{b,t,h,w} \leftarrow x_{b,t,h,w} + \mathrm{TimeEnc}(t)$$

- Flatten along time at each pixel location

$$x_{\mathrm{flat}} \in \mathbb{R}^{(BHW) \times T \times C} \qquad [Q, K, V] = x_{\mathrm{flat}} W_{qkv} \qquad Q, K, V \in \mathbb{R}^{(BHW) \times \mathrm{heads} \times T \times d}$$

- Temporal MHSA
- Residual + MLP

$$z = \mathrm{softmax}\left(\frac{QK^{\top}}{\sqrt{d}}\right) V$$

- Output: (B, T, H, W, C) at 1/8L resolution

Precision Operator

- Goal: to be able to preserve finer local detail with Convolutions
- Input $x \in \mathbb{R}^{B\times T\times H\times W\times C} \Longrightarrow (BT, C, H, W)$
- Apply 3x3 convolutions

$$x' = \mathrm{Conv}_{3\times 3}(\mathrm{GroupNorm}(x))$$

- Expansion + Contraction Conv + Residual

$$x_{\mathrm{exp}} = \mathrm{GELU}\big(\mathrm{Conv}_{3\times 3}^{C\to 4C}(x_{\mathrm{norm}})\big) \qquad x_{\mathrm{contr}} = \mathrm{Conv}_{3\times 3}^{4C\to C}(x_{\mathrm{exp}}) \qquad x' = x + x_{\mathrm{contr}}$$

- Output at L/2 resolution $x' \in \mathbb{R}^{B\times T\times H\times W\times C}$

Pyramid Exchanges

- Purpose: a general way to *share* learnings across S/T/P operators
- At the end of each STP block, each operator has features at its native resolution
  - Precision (B,T,L/2,L/2,Cp)

The *learned Laplacian pyramid resampler* takes each output and resizes it the resolutions of the other operators

  - Precision → downsampled to L/8, L/16

- The resampled outputs are fused and sent into the next block

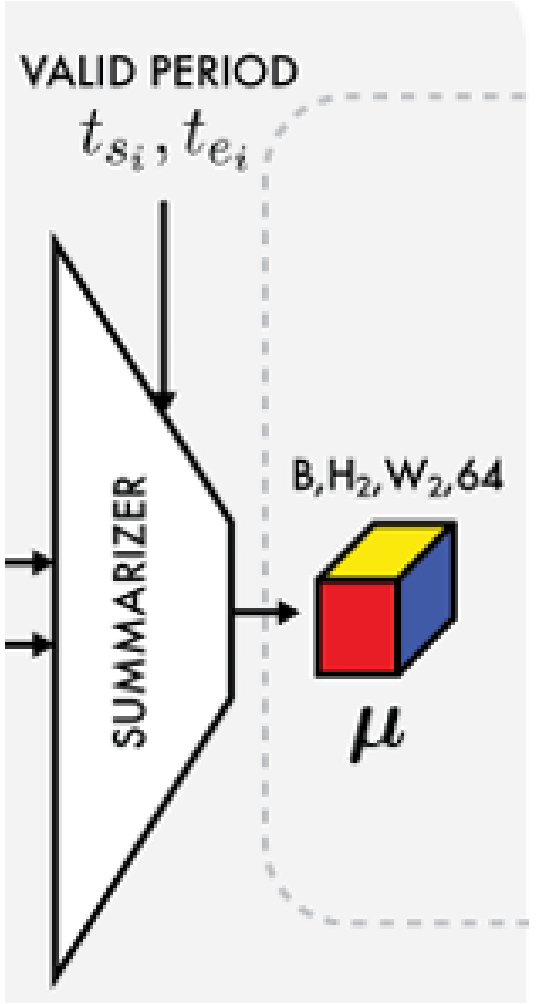
Implementation

```python
class LearnedSpatialResampling(nn.Module):
    """Learned Laplacian pyramid rescaling for spatial pyramid exchanges."""

    def __init__(self, in_channels: int, out_channels: int, scale_factor: float):
        super().__init__()
        self.scale_factor = scale_factor

        if scale_factor > 1:
            # Upsampling: scale_factor > 1 means output is larger
            # For scale_factor = 2.0, output size = input_size * 2
            self.conv = nn.ConvTranspose2d(
                in_channels, out_channels,
                kernel_size=4, stride=2, padding=1
            )
        elif scale_factor < 1:
            # Downsampling: scale_factor < 1 mea
            # For scale_factor = 0.5, output siz
            stride = int(1 / scale_factor)
            self.conv = nn.Conv2d(
                in_channels, out_channels,
                kernel_size=stride * 2 - 1,
                stride=stride,
                padding=stride - 1
            )
        else:
            # Same resolution
            self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.conv(x)
```

```python
self.space_to_time = LearnedSpatialResampling(self.space_dim, self.time_dim, 2.0)
self.space_to_precision = LearnedSpatialResampling(self.space_dim, self.precision_dim, 8.0)
self.time_to_space = LearnedSpatialResampling(self.time_dim, self.space_dim, 0.5)
self.time_to_precision = LearnedSpatialResampling(self.time_dim, self.precision_dim, 4.0)
self.precision_to_space = LearnedSpatialResampling(self.precision_dim, self.space_dim, 0.125)
self.precision_to_time = LearnedSpatialResampling(self.precision_dim, self.time_dim, 0.25)
```

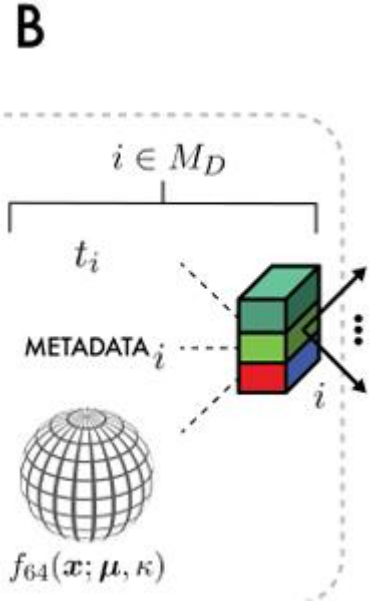
Summary Period Encoder

- After the STP encoder, we still have a time-series of features per pixel
- Objective: to reduce the sequence of features to a single embedding per pixel that summarizes a chosen valid period $[t_s, t_e]$
- Valid Period Encoding: encode start, end, and duration with sinusoidal + MLP features
- Query Vector: learned query conditioned on valid-period encoding
- Attention pooling: have the query attend over temporal features at each pixel location
- Outputs: Embedding Grid (B, H, W, 64)



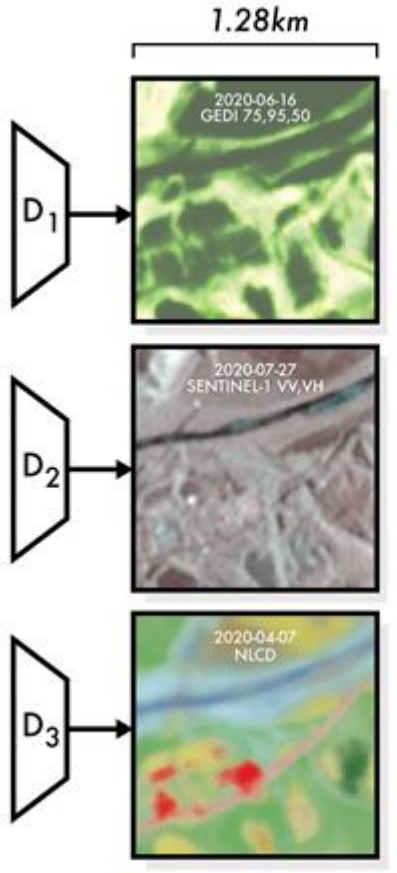
Von Mises-Fisher Bottleneck

- Constrain embeddings to the unit hypersphere
- Each embedding is modeled as the mean direction of a von Mises-Fisher distribution (a probability distribution on the hypersphere $S^{d-1}, in\ our\ case\ d = 64$)
- Project STP features → 64D vector
- Apply L2norm
- Output: per-pixel unit vector $\mu \in S^{63}$



Source Decoders

- Purpose: to reconstruct raw observations from embeddings; used for training
- Lightweight decoders per source
- Using MLPs
- Input (B, H, W, 64)
- Outputs:
  - Continuous values or Categorical classes



Loss Function

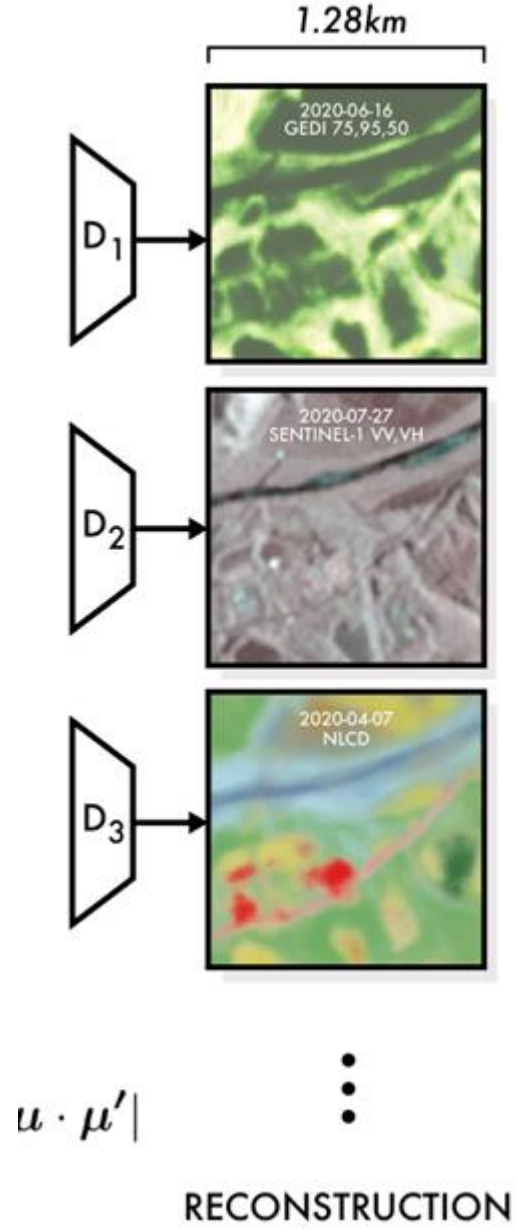
Loss = Reconstruction Loss + Batch Uniformity Loss + Teacher Student Consistency Loss + Text Contrastive (CLIP) Loss

$$l = \frac{a}{M} \sum_{i \in M} f_i(\mathbf{y}_i, \mathbf{y}_i') w_i + b \sum_{i=1}^{64} |u_i \cdot u_i'| + c \left(\frac{1 - \boldsymbol{u} \cdot \boldsymbol{u}_s}{2} \right) + d f_{\text{CLIP}}(\boldsymbol{u}, \boldsymbol{u}_t) \quad (3)$$

Reconstruction Loss

- Compares the predicted observation $y_i{}'$ to the true observation y_i for each source i
- Objective is to force the embeddings to carry enough info to be able to reconstruct the raw inputs
- Using L1 loss
- Weights w_i allow scaling importance by source

$$a \cdot \sum_i f_i(y_i, y_i') \cdot w_i$$

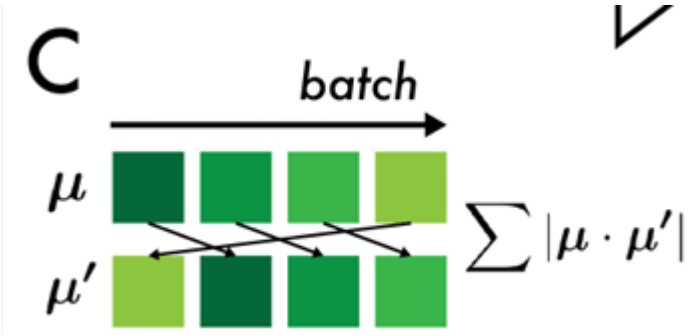


Batch Uniformity Loss

$$b \cdot \text{BatchUniformity}(\mu)$$

- Objective: to make the embeddings (64D) to be uniformly distributed
- Take an embedding batch $\mu_1, \mu_2, \ldots, \mu_B$
- Rotate it by 1 along the batch dimension:
- $\mu_{rot} = [\mu_B, \mu_1, \mu_2, \ldots \mu_{B-1}]$
- Pair each original vector with the rotated one:

$$(\mu_1, \mu_B), (\mu_2, \mu_1), \ldots$$



Compare the dot product for each pair:

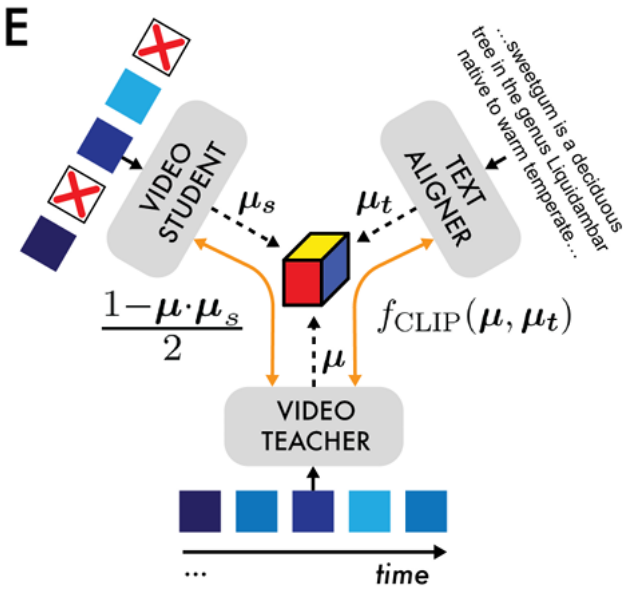
$$L_{uniform} = \frac{1}{B} \Sigma_{i=1}^{B} \left| \mu_i \cdot \mu_{rot,i} \right|$$

- If the embeddings are aligned (dot product closer to 1) → loss is larger
- If the embeddings are further apart (dot product closer to 0) → loss is smaller

```
x = torch.nn.functional.normalize(x, p=2, dim=-1)
# Rotate (roll) sample pairs to approximate u' in the paper
x_prime = torch.roll(x, shifts=1, dims=0)
dots = (x * x_prime).sum(dim=-1).abs()  # |u · u'|
```

Teacher-Student Consistency Loss

$$c \cdot \frac{1 - \mu \cdot \mu_s}{2}$$



μ = teacher embedding (all data sources)

μ_s = student embedding (only a few data sources)

Dot product measures cosine similarity so if the student matches the teacher:

dot product = 1

loss = 0

Text Contrastive Loss

$$d \cdot f_{\text{CLIP}}(\mu, u_t)$$

```python
def clip_loss(self, image_embeddings: torch.Tensor,
              text_embeddings: torch.Tensor) -> torch.Tensor:
    """Compute CLIP-style contrastive loss."""
    # Expect (B, D) vs (B, D)
    img = torch.nn.functional.normalize(image_embeddings, p=2, dim=-1)
    txt = torch.nn.functional.normalize(text_embeddings, p=2, dim=-1)
    logits = img @ txt.t()  # (B, B)
    targets = torch.arange(img.size(0), device=img.device)
    loss_i = torch.nn.functional.cross_entropy(logits, targets)
    loss_t = torch.nn.functional.cross_entropy(logits.t(), targets)
    return 0.5 * (loss_i + loss_t)
```