

design

February 28, 2026

1 Non-Stationary Spectral Decomposition Network (NS-SDN) For Econometric Forecasting

Nikhil Sunder

Student | ICSRI Research Fellow | Open-Source Developer

B.S.B.A. Quantitative Economics, Finance & Minor in Math

University of Miami Herbert Business School

email: nss106@miami.edu

phone: (305) 409-3108

links: [Github](#) | [PyPI](#) | [Anaconda](#) | [LinkedIn](#) | [ORCID](#) | [Zenodo](#) | [Handshake](#)

2 NS-SDN: Design & Buildout

This notebook implements the **Non-Stationary Spectral Decomposition Network (NS-SDN)**

2.1 Target decomposition

$$\hat{y}(t) = B(t) + \sum_{k=1}^K A_k(t) \sin(\theta_k(t) + \varphi_k(t)),$$

where: - $B(t)$ is **trend** / **low-frequency** component, - $A_k(t) > 0$ is a **time-varying amplitude** (envelope), - $\theta_k(t)$ is **cumulative phase**, with **instantaneous frequency** $\omega_k(t) = \frac{d}{dt}\theta_k(t)$, - $\varphi_k(t)$ is a **phase offset** (horizontal shift).

These are emitted from a **latent state** h_n on a discrete grid $t_0 < \dots < t_N$.

2.2 Revisions and Additions

We implement components motivated by the following reference papers:

- **SIREN**: periodic activations + principled initialization for representing oscillatory functions. (Sitzmann et al., 2020)
- **Fourier Features**: positional encoding to mitigate spectral bias. (Tancik et al., 2020)
- **Hilbert–Huang** / **EMD**: instantaneous frequency as a diagnostic object; optionally used for initialization/diagnostics. (Huang et al., 1998)
- **Neural ODEs**: interpret θ as integral of ω ; allow time-aware latent updates. (Chen et al., 2019)

- **TVP-VAR**: treat $A_k(t)$ and $\omega_k(t)$ as time-varying “parameters” -> smoothness/drift priors. (Lubik & Matthes, 2015)
- **Wavelet Networks**: localization intuition -> envelopes + optional sparsity/gating. (Zhang & Benveniste, 1992)
- **DeepAR / TFT**: forecasting protocol (iterated vs direct multi-horizon), scaling, and probabilistic outputs. (Salinas et al., 2019) (Lim et al., 2020)
- **Spectral density paper**: add frequency-domain evaluation/loss beyond MSE. (Mohammadi et al., 2026)

We proceed *component-by-component* with clean PyTorch modules and a minimal training + diagnostics scaffold.

```
[ ]: # Imports
from dataclasses import dataclass
from typing import Optional, Dict, Any, Literal
import math
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
```

2.3 Data and time grid conventions

We standardize:

- **t**: (N, 1) float tensor, strictly increasing (can be irregular).
- **y**: (N, D) float tensor; default D=1.
- optional **x**: (N, P) exogenous covariates available at each t_n .

```
[2]: @dataclass(slots=True)
class TimeSeriesBatch:
    """A batch of time series data.

    Attributes:
        t (torch.Tensor): Time steps (N, 1)
        y (torch.Tensor): Observations (N, D)
        x (Optional[torch.Tensor]): Optional covariates (N, P)

    Note:
        t should be a column vector of shape (N, 1).
        y should be a matrix of shape (N, D) where D is the number of observed
        ↪ features.
        x is optional and can be used for additional covariates, with shape (N,
        ↪ P) where P is the number of covariate features.
    """

    t: torch.Tensor # (N, 1)
    """Time steps for the batch, expected to be a column vector of shape (N, 1).
    ↪ Each entry represents the time at which the corresponding
```

```

        observation in `y` was recorded."""

    y: torch.Tensor # (N, D)
    """Observations for the batch, expected to be a matrix of shape (N, D)
    ↪where D is the number of observed features."""

    x: Optional[torch.Tensor] = None # (N, P)
    """Optional covariates for the batch, expected to be a matrix of shape (N,
    ↪P) where P is the number of covariate
        features. This can be used to include additional information that may help
    ↪in modeling the time series data."""

def make_time_grid(n: int, *, dt: float = 1.0, start: float = 0.0, device:
    ↪Optional[torch.device] = None, dtype: torch.dtype = torch.float32,) -> torch.
    ↪Tensor:
    """Create a time grid tensor of shape (n, 1) with specified parameters.

    Args:
        n (int): Number of time steps to generate.
        dt (float, optional): Time step size. Default is 1.0.
        start (float, optional): Starting time value. Default is 0.0.
        device (torch.device, optional): Device on which to create the tensor.
    ↪Default is None (CPU).
        dtype (torch.dtype, optional): Data type of the tensor. Default is
    ↪torch.float32.

    Returns:
        torch.Tensor: A tensor of shape (n, 1) representing the time grid.

    Raises:
        ValueError: If n is not a positive integer.

    Note:
        The time grid starts at the specified `start` value and increments by
    ↪`dt` for
        `n` time steps. The resulting tensor is a column vector suitable for
    ↪use as time
        inputs in time series models.
    """

    if n <= 0:
        raise ValueError("n must be a positive integer")

    t = start + dt * torch.arange(n, device=device, dtype=dtype)
    return t.view(-1, 1)

```

```
def ensure_2d(x: torch.Tensor, name: str) -> torch.Tensor:
    """Ensure that a tensor is 2D.

    Args:
        x (torch.Tensor): The input tensor.
        name (str): The name of the tensor (for error messages).

    Returns:
        torch.Tensor: The input tensor if it is 2D.

    Raises:
        ValueError: If the input tensor is not 2D.

    Note:
        This function checks if the input tensor `x` has exactly 2 dimensions.
        ↪ If it does, it returns the tensor unchanged.
        ↪ If not, it raises a ValueError with a message indicating the expected
        ↪ and actual shapes of the tensor. This is useful
        ↪ for validating inputs to functions that require 2D tensors, such as
        ↪ time series data where the first dimension
        ↪ typically represents time steps and the second dimension represents
        ↪ features.
    """
    if x.ndim != 2:
        raise ValueError(f"{name} must be 2D; got shape={tuple(x.shape)}")
    return x
```

2.4 MapMinMax scaler (input/output scaling)

DeepAR explicitly motivates scaling as practically important for stable training and comparability. (Salinas et al., 2019)

We implement a minimal, *tensor-native* min-max scaler with `transform` and `inverse_transform`.

- Fit on **training window only** to avoid leakage.
- Apply to y (and optionally to x).

```
[3]: @dataclass(slots=True)
class MapMinMax:
    """A class for performing min-max scaling on time series data. This
    ↪ transformation scales each feature of the input data to a specified target
    ↪ range,
    ↪ typically [-1, 1]. The class provides methods for fitting the scaler to the
    ↪ data, transforming the data, and inverse transforming the scaled data
    ↪ back to the original scale.

    Attributes:
```

`feature_min` (torch.Tensor): Minimum values for each feature in the original data, used for scaling.

`feature_max` (torch.Tensor): Maximum values for each feature in the original data, used for scaling.

`target_min` (float): Minimum value of the target range for scaling. Default is -1.0.

`target_max` (float): Maximum value of the target range for scaling. Default is 1.0.

`eps` (float): A small value added to the denominator during scaling to prevent division by zero. Default is 1e-12.

Note:

The `fit` method computes the minimum and maximum values for each feature in the input data `y` and initializes the scaler. The `transform` method scales the input data to the target range using the computed minimum and maximum values. The `inverse_transform` method can be used to convert the scaled data back to the original scale. The `eps` parameter ensures numerical stability when the feature range is very small or zero, preventing division by zero errors during scaling.

"""

`feature_min`: torch.Tensor

"""Minimum values for each feature in the original data, used for scaling. This tensor should have the same number of elements as the number of features (D) in the input data `y`. Each element represents the minimum value of the corresponding feature across the dataset, which is used to shift the data to a 0-based scale before scaling to the target range."""

`feature_max`: torch.Tensor

"""Maximum values for each feature in the original data, used for scaling. This tensor should have the same number of elements as the number of features (D) in the input data `y`. Each element represents the maximum value of the corresponding feature across the dataset, which is used to scale the data to the target range."""

`target_min`: float = -1.0

"""Minimum value of the target range for scaling. This is the lower bound of the range to which the original data will be scaled.

The default value is -1.0, meaning that after transformation, the minimum value of each feature will be mapped to -1.0."""

`target_max`: float = 1.0

```

    """Maximum value of the target range for scaling. This is the upper bound
    of the range to which the original data will be scaled.
    The default value is 1.0, meaning that after transformation, the maximum
    value of each feature will be mapped to 1.0."""

    eps: float = 1e-12
    """A small value added to the denominator during scaling to prevent
    division by zero. This helps to ensure numerical stability
    when the feature range is very small or zero."""

    @classmethod
    def fit(cls, y: torch.Tensor, *, target_min: float = -1.0, target_max:
float = 1.0) -> "MapMinMax":
    """Fit the MapMinMax scaler to the input data `y` by computing the
    minimum and maximum values for each feature.

    Args:
        y (torch.Tensor): The input data to fit the scaler, expected to be
        a 2D tensor of shape (N, D) where N is the number of samples and D is the
        number of features.
        target_min (float, optional): Minimum value of the target range for
        scaling. Default is -1.0.
        target_max (float, optional): Maximum value of the target range for
        scaling. Default is 1.0.

    Returns:
        MapMinMax: An instance of the MapMinMax class initialized with the
        computed feature minimum and maximum values.

    Note:
        The `fit` method computes the minimum and maximum values for each
        feature in the input data `y` and initializes the MapMinMax scaler with
        these values.
        The method ensures that the input data is 2D and then calculates
        the minimum and maximum values along the first dimension (across samples)
        for each
        feature. These values are stored in the `feature_min` and
        `feature_max` attributes of the scaler instance.
    """

    y = ensure_2d(y, "y")
    fmin = y.min(dim=0).values
    fmax = y.max(dim=0).values
    return cls(fmin, fmax, target_min=target_min, target_max=target_max)

    def transform(self, y: torch.Tensor) -> torch.Tensor:

```

```

        """Transform the input data `y` to the target range using the fitted
        ↪ scaler parameters.

        Args:
            y (torch.Tensor): The input data to be transformed, expected to be
            ↪ a 2D tensor of shape (N, D) where N is the number of samples and D is the
            ↪ number of features.

        Returns:
            torch.Tensor: The transformed data, scaled to the target range.
        """

        y = ensure_2d(y, "y")
        denom = (self.feature_max - self.feature_min).clamp_min(self.eps)
        y01 = (y - self.feature_min) / denom
        return y01 * (self.target_max - self.target_min) + self.target_min

    def inverse_transform(self, ys: torch.Tensor) -> torch.Tensor:
        """Inverse transform the scaled data back to the original range.

        Args:
            ys (torch.Tensor): The scaled data to be inverse transformed,
            ↪ expected to be a 2D tensor of shape (N, D) where N is the number of samples
            ↪ and D is the number of features.

        Returns:
            torch.Tensor: The data transformed back to the original range.
        """

        ys = ensure_2d(ys, "ys")
        y01 = (ys - self.target_min) / (self.target_max - self.target_min)
        return y01 * (self.feature_max - self.feature_min) + self.feature_min

```

2.5 Feature maps: identity vs Fourier Features

Fourier feature embeddings are a “clean alternative to SIREN” and mitigate spectral bias by exposing sinusoidal bases at the input. (Tancik et al., 2020)

We implement:

- `FourierFeatures(mode="random")`: random Gaussian frequencies (paper-style).
- `FourierFeatures(mode="grid")`: deterministic frequency grid (more interpretable).

We also allow `include_input=True` to concatenate raw t .

```

[4]: class FourierFeatures(nn.Module):
    """A PyTorch module for generating Fourier features from time inputs. This
    ↪ module takes a 1D time input and projects it into a higher-dimensional space
    ↪ using sinusoidal functions

```

of different frequencies. The frequencies can be either randomly sampled or arranged in a grid, and the original input can optionally be included in the output.

Attributes:

num_frequencies (int): The number of Fourier frequencies to use.
scale (float): A scaling factor for the frequencies.
mode (Literal["random", "grid"]): The mode for generating frequencies, either "random" or "grid".
include_input (bool): Whether to include the original input in the output.
"""

```
def __init__(self, num_frequencies: int, *, scale: float = 10.0, mode: Literal["random", "grid"] = "random", include_input: bool = True,) -> None:
    """Initialize the FourierFeatures module with the specified parameters.
```

Args:

num_frequencies (int): The number of Fourier frequencies to use. Must be a positive integer.
scale (float, optional): A scaling factor for the frequencies. Default is 10.0.
mode (Literal["random", "grid"], optional): The mode for generating frequencies, either "random" or "grid". Default is "random".
include_input (bool, optional): Whether to include the original input in the output. Default is True.

Raises:

ValueError: If num_frequencies is not a positive integer or if mode is not "random" or "grid".
"""

```
super().__init__()
```

```
if num_frequencies <= 0:
    raise ValueError("num_frequencies must be positive")
if mode not in {"random", "grid"}:
    raise ValueError("mode must be 'random' or 'grid'")
```

```
self.num_frequencies = int(num_frequencies)
self.scale = float(scale)
self.mode = mode
self.include_input = bool(include_input)
```

```
if mode == "random":
    B = torch.randn(self.num_frequencies, 1) * self.scale
```

```

        elif mode == "grid":
            freqs = torch.arange(1, self.num_frequencies + 1, dtype=torch.
↪float32).view(-1, 1)
            B = freqs * self.scale
        else:
            raise ValueError("mode must be 'random' or 'grid'")

        self.register_buffer("B", B)

    @property
    def out_dim(self) -> int:
        """Calculate the output dimension of the Fourier features based on the_
↪number of frequencies and whether the original input is included."""

        base = 2 * self.num_frequencies
        return base + (1 if self.include_input else 0)

    def forward(self, t: torch.Tensor) -> torch.Tensor:
        """Compute the Fourier features for the given input tensor.

        Args:
            t (torch.Tensor): A tensor of shape (N, 1) representing the input_
↪time values.

        Returns:
            torch.Tensor: A tensor of shape (N, out_dim) containing the Fourier_
↪features.

        Raises:
            ValueError: If the input tensor `t` is not 2D or if its second_
↪dimension is not 1.
        """

        t = ensure_2d(t, "t")
        if t.shape[1] != 1:
            raise ValueError(f"t must be (N,1); got {tuple(t.shape)}")
        proj = (2.0 * math.pi) * (t @ self.B.T) # (N, M)
        parts = [torch.sin(proj), torch.cos(proj)]
        if self.include_input:
            parts.insert(0, t)
        return torch.cat(parts, dim=1)

```

2.6 SIREN block and initialization

SIREN uses sine activations throughout and proposes an initialization that preserves signal statistics and gradient flow for oscillatory functions. (Sitzmann et al., 2020)

We implement: - Sine(ω_0) activation, - siren_init_ for Linear layers, - SirenMLP as a

reusable module.

This ensures $B(t)$, $A_k(t)$, $\omega_k(t)$, etc. will be represented with periodic structure directly.

```
[5]: class Sine(nn.Module):
    """A PyTorch module that applies the sine activation function with a
    ↪ learnable frequency parameter omega0.

    Attributes:
        omega0 (float): The frequency parameter for the sine function.
    """

    def __init__(self, omega0: float = 1.0) -> None:
        """Initialize the Sine activation module with the specified frequency
        ↪ parameter omega0.

        Args:
            omega0 (float, optional): The frequency parameter for the sine
            ↪ function. Default is 1.0.
        """

        super().__init__()
        self.omega0 = float(omega0)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Apply the sine activation function to the input tensor `x` using the
        ↪ frequency parameter `omega0`.

        Args:
            x (torch.Tensor): The input tensor to which the sine activation
            ↪ function will be applied.

        Returns:
            torch.Tensor: The output tensor after applying the sine activation
            ↪ function.
        """

        return torch.sin(self.omega0 * x)

def siren_init_(layer: nn.Linear, *, is_first: bool, omega0: float) -> None:
    """Practical SIREN initialization: scale weights depending on layer
    ↪ position.

    Args:
        layer (nn.Linear): The linear layer to initialize.
        is_first (bool): Whether this is the first layer in the network.
```

```

        omega0 (float): The frequency parameter for the sine activation_
↪function.
        """

        with torch.no_grad():
            in_features = layer.in_features
            if is_first:
                bound = 1.0 / in_features
            else:
                bound = math.sqrt(6.0 / in_features) / omega0
            layer.weight.uniform_(-bound, bound)
            if layer.bias is not None:
                layer.bias.uniform_(-bound, bound)

class SirenMLP(nn.Module):
    def __init__(self, in_dim: int, out_dim: int, *, hidden_dim: int = 64,
↪depth: int = 3, omega0: float = 30.0, omega: float = 1.0) -> None:
        """Initialize a SIREN (Sinusoidal Representation Network) MLP.

        Args:
            in_dim (int): The input dimension.
            out_dim (int): The output dimension.
            hidden_dim (int, optional): The hidden layer dimension. Default is_
↪64.
            depth (int, optional): The number of hidden layers. Default is 3.
            omega0 (float, optional): The frequency parameter for the first_
↪layer. Default is 30.0.
            omega (float, optional): The frequency parameter for the hidden_
↪layers. Default is 1.0.
        """

        super().__init__()
        if depth < 1:
            raise ValueError("depth must be >= 1")

        layers: list[nn.Module] = []

        first = nn.Linear(in_dim, hidden_dim)
        siren_init_(first, is_first=True, omega0=omega0)
        layers += [first, Sine(omega0)]

        for _ in range(depth - 1):
            lin = nn.Linear(hidden_dim, hidden_dim)
            siren_init_(lin, is_first=False, omega0=omega)
            layers += [lin, Sine(omega)]

        final = nn.Linear(hidden_dim, out_dim)

```

```

with torch.no_grad():
    bound = math.sqrt(6.0 / hidden_dim) / omega
    final.weight.uniform_(-bound, bound)
    if final.bias is not None:
        final.bias.uniform_(-bound, bound)
    layers.append(final)

self.net = nn.Sequential(*layers)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    """Forward pass through the SIREN MLP.

    Args:
        x (torch.Tensor): The input tensor.

    Returns:
        torch.Tensor: The output tensor.
    """

    return self.net(x)

```

2.7 State-space NS-SDN in discrete time

We define a **latent state** h_n with transition:

$$h_n = F_\psi(h_{n-1}, x_n, \Delta t_n),$$

and an emission producing $B_n, A_n, \omega_n, \theta_n, \varphi_n$, then:

$$\hat{y}_n = B_n + \sum_{k=1}^K A_{n,k} \sin(\theta_{n,k} + \varphi_{n,k})$$

We now implement this as: 1) a transition cell, 2) parameter heads emitting $B_n, A_n, \omega_n, \varphi_n$, 3) a loop over time that integrates θ_n .

This matches our discrete-time state-space formulation.

```

[6]: class TimeAwareTransition(nn.Module):
    """A time-aware transition module that updates the hidden state based on
    ↪ the previous hidden state, current input, and time step.
    The transition is computed using a feedforward network that takes the
    ↪ concatenation of the previous hidden state and current input,
    and produces a change in the hidden state which is then scaled by the time
    ↪ step `dt_n` before being added to the previous hidden
    state to produce the new hidden state.

    Attributes:
        h_dim (int): The dimension of the hidden state.

```

```

        g (nn.Sequential): The feedforward network used to compute the change
        ↪ in hidden state.
        """

    def __init__(self, in_dim: int, h_dim: int, *, hidden_dim: int = 64) ->
    ↪ None:
        """Initialize the TimeAwareTransition module.

        Args:
            in_dim (int): The dimension of the input.
            h_dim (int): The dimension of the hidden state.
            hidden_dim (int, optional): The dimension of the hidden layer in
            ↪ the feedforward network. Defaults to 64.
        """

        super().__init__()

        self.h_dim = int(h_dim)
        self.g = nn.Sequential(
            nn.Linear(h_dim + in_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, h_dim),
        )

    def forward(self, h_prev: torch.Tensor, x_n: torch.Tensor, dt_n: torch.
    ↪ Tensor) -> torch.Tensor:
        """Compute the new hidden state based on the previous hidden state,
        ↪ current input, and time step.

        Args:
            h_prev (torch.Tensor): The previous hidden state, expected to be a
            ↪ tensor of shape (N, h_dim).
            x_n (torch.Tensor): The current input, expected to be a tensor of
            ↪ shape (N, in_dim).
            dt_n (torch.Tensor): The time step, expected to be a tensor of
            ↪ shape (N, 1) or a scalar.

        Returns:
            torch.Tensor: The new hidden state, computed as  $h_{prev} + dt_n *
            ↪ g(\text{concat}([h_{prev}, x_n], \text{dim}=-1))$ .
        """

        z = torch.cat([h_prev, x_n], dim=-1)
        dh = self.g(z)
        return h_prev + dt_n * dh

```

```

class GRUTransition(nn.Module):
    """This module implements a GRU-based transition function that updates the
    ↪ hidden state based on the previous hidden
    state and current input.

    Attributes:
        cell (nn.GRUCell): The GRU cell used for the transition.
    """

    def __init__(self, in_dim: int, h_dim: int) -> None:
        """Initialize the GRUTransition module.

        Args:
            in_dim (int): The dimension of the input.
            h_dim (int): The dimension of the hidden state.
        """

        super().__init__()
        self.cell = nn.GRUCell(input_size=in_dim, hidden_size=h_dim)

    def forward(self, h_prev: torch.Tensor, x_n: torch.Tensor, dt_n: torch.
    ↪ Tensor) -> torch.Tensor:
        """Forward pass of the GRUTransition module.

        Args:
            h_prev (torch.Tensor): The previous hidden state, expected to be a
            ↪ tensor of shape (N, h_dim).
            x_n (torch.Tensor): The current input, expected to be a tensor of
            ↪ shape (N, in_dim).
            dt_n (torch.Tensor): The time step, expected to be a tensor of
            ↪ shape (N, 1) or a scalar.

        Returns:
            torch.Tensor: The new hidden state, computed using the GRU cell.
        """

        return self.cell(x_n, h_prev)

class EmissionHeads(nn.Module):
    """This module implements the emission heads that produce the parameters
    ↪ for the output distribution from the hidden state.

    Attributes:
        B (nn.Linear): Linear layer to produce the baseline output.
        A (nn.Linear): Linear layer to produce the amplitude of the sinusoidal
        ↪ component.

```

```

        omega (nn.Linear): Linear layer to produce the frequency of the
        ↪ sinusoidal component.
        phi (nn.Linear): Linear layer to produce the phase of the sinusoidal
        ↪ component.
        omega_base (torch.Tensor): Base frequency for the sinusoidal component.
        positive_omega (bool): Whether to enforce positive frequencies.
    """

    def __init__(self, h_dim: int, *, y_dim: int, k: int, omega_base:
    ↪ Optional[torch.Tensor] = None, positive_omega: bool = False) -> None:
        """Initialize the EmissionHeads module.

        Args:
            h_dim (int): The dimension of the hidden state.
            y_dim (int): The dimension of the output.
            k (int): The number of sinusoidal components.
            omega_base (Optional[torch.Tensor], optional): Base frequency for
            ↪ the sinusoidal component. Defaults to None.
            positive_omega (bool, optional): Whether to enforce positive
            ↪ frequencies. Defaults to False.
        """

        super().__init__()
        self.y_dim = int(y_dim)
        self.k = int(k)
        self.positive_omega = bool(positive_omega)

        self.B = nn.Linear(h_dim, y_dim)
        self.A = nn.Linear(h_dim, k)
        self.omega = nn.Linear(h_dim, k)
        self.phi = nn.Linear(h_dim, k)

        if omega_base is None:
            omega_base = torch.zeros(k)
        self.register_buffer("omega_base", omega_base.view(1, k))

    def forward(self, h: torch.Tensor) -> Dict[str, torch.Tensor]:
        """
        Forward pass of the EmissionHeads module.

        Args:
            h (torch.Tensor): The hidden state, expected to be a tensor of
            ↪ shape (N, h_dim).

        Returns:

```

```

        Dict[str, torch.Tensor]: A dictionary containing the parameters for
        the output distribution:
        - "B": Baseline output.
        - "A": Amplitude of the sinusoidal component.
        - "omega": Frequency of the sinusoidal component.
        - "phi": Phase of the sinusoidal component.
    """

    B = self.B(h)
    A = F.softplus(self.A(h)) + 1e-4
    omega_raw = self.omega_base + self.omega(h)
    omega = F.softplus(omega_raw) + 1e-4 if self.positive_omega else
    omega_raw
    phi = self.phi(h)
    return {"B": B, "A": A, "omega": omega, "phi": phi}

```

2.8 Gated emissions

TFT uses gating/variable selection to adaptively modulate contributions. (Lim et al., 2020)

This will be implemented as `use_gating=True` in `NSSDNStateSpace`.

```

[7]: class GatedEmissionHeads(EmissionHeads):
    """This module implements the gated emission heads that produce the
    parameters for the output distribution from the hidden state,
    with an additional gating mechanism to modulate the amplitude of the
    sinusoidal components.

    Attributes:
        gate (nn.Linear): Linear layer to produce the gating values for the
        amplitude of the sinusoidal components.
    """

    def __init__(self, h_dim: int, *, y_dim: int, k: int, omega_base:
    Optional[torch.Tensor] = None, positive_omega: bool = False) -> None:
        """Initialize the GatedEmissionHeads module.

        Args:
            h_dim (int): The dimension of the hidden state.
            y_dim (int): The dimension of the output.
            k (int): The number of sinusoidal components.
            omega_base (Optional[torch.Tensor], optional): Base frequency for
            the sinusoidal component. Defaults to None.
            positive_omega (bool, optional): Whether to enforce positive
            frequencies. Defaults to False.
        """

```

```

        super().__init__(h_dim=h_dim, y_dim=y_dim, k=k, omega_base=omega_base,
        ↪positive_omega=positive_omega)
        self.gate = nn.Linear(h_dim, k)

    def forward(self, h: torch.Tensor) -> Dict[str, torch.Tensor]:
        """Forward pass of the GatedEmissionHeads module.

        Args:
            h (torch.Tensor): The hidden state, expected to be a tensor of
            ↪shape (N, h_dim).

        Returns:
            Dict[str, torch.Tensor]: A dictionary containing the parameters for
            ↪the output distribution, including the gating values:
            - "B": Baseline output.
            - "A": Amplitude of the sinusoidal component, modulated by the
            ↪gating values
            - "omega": Frequency of the sinusoidal component.
            - "phi": Phase of the sinusoidal component.
            - "gate": The gating values applied to the amplitude of the
            ↪sinusoidal components, with values in the range (0, 1) due to the sigmoid
            ↪activation.
            """

        out = super().forward(h)
        g = torch.sigmoid(self.gate(h))
        out["gate"] = g
        out["A"] = out["A"] * g
        return out

```

2.9 NS-SDN state-space module

We implement `forward_sequence(...)` that: - computes Δt_n , - updates h_n , - emits A_n, ω_n, ϕ_n , - integrates θ_n , - synthesizes $\hat{y}_n$.

Forecasting protocol note: - proposal uses iterated forecasting (roll state forward), - DeepAR is a canonical reference for this discipline. (Salinas et al., 2019)

```

[8]: class NSSDNStateSpace(nn.Module):
    """This module implements the state-space model for the Non-Stationary
    ↪Spectral Decomposition Network (NSSDN). It consists of a transition module
    ↪that updates the hidden state based on the previous hidden state, current
    ↪input, and time step, and an emission module that produces the parameters
    ↪for the output distribution from the hidden state. The model can be
    ↪configured to use either a time-aware transition or a GRU-based transition,
    ↪and can optionally include gated emission heads to modulate the amplitude of
    ↪the sinusoidal components in the output distribution.
    """

```

```

    Attributes:
        y_dim (int): The dimension of the output.
        x_dim (int): The dimension of the exogenous input.
        h_dim (int): The dimension of the hidden state.
        k (int): The number of sinusoidal components in the output distribution.
        transition (nn.Module): The transition module used to update the hidden_
        ↪state.
        emit (nn.Module): The emission module used to produce the parameters_
        ↪for the output distribution.
        mix (nn.Linear): A linear layer used to mix the sinusoidal components_
        ↪into the final output.
        h0 (nn.Parameter): The initial hidden state, learned during training.
        theta0 (nn.Parameter): The initial phase of the sinusoidal components,_
        ↪learned during training.
        """

    def __init__(self, *, y_dim: int = 1, x_dim: int = 0, h_dim: int = 32, k:
    ↪int = 6, transition: Literal["timeaware", "gru"] = "timeaware",
        positive_omega: bool = False, omega_base: Optional[torch.
    ↪Tensor] = None, use_gating: bool = False) -> None:
        """Initialize the NSSDNStateSpace module.

    Args:
        y_dim (int, optional): The dimension of the output. Default is 1.
        x_dim (int, optional): The dimension of the exogenous input.
    ↪Default is 0.
        h_dim (int, optional): The dimension of the hidden state. Default_
    ↪is 32.
        k (int, optional): The number of sinusoidal components. Default is_
    ↪6.
        transition (Literal["timeaware", "gru"], optional): The type of_
    ↪transition to use, either "timeaware" or "gru". Default is "timeaware".
        positive_omega (bool, optional): Whether to enforce positive_
    ↪frequencies. Default is False.
        omega_base (Optional[torch.Tensor], optional): Base frequency for_
    ↪the sinusoidal component. Defaults to None.
        use_gating (bool, optional): Whether to use gated emission heads.
    ↪Default is False.

    Raises:
        ValueError: If transition is not "timeaware" or "gru".
        """

    super().__init__()
    self.y_dim = int(y_dim)

```

```

self.x_dim = int(x_dim)
self.h_dim = int(h_dim)
self.k = int(k)

in_dim = self.y_dim + self.x_dim

if transition == "timeaware":
    self.trans = TimeAwareTransition(in_dim=in_dim, h_dim=h_dim)
elif transition == "gru":
    self.trans = GRUTransition(in_dim=in_dim, h_dim=h_dim)
else:
    raise ValueError("transition must be 'timeaware' or 'gru'")

heads_cls = GatedEmissionHeads if use_gating else EmissionHeads
self.emit = heads_cls(h_dim=h_dim, y_dim=y_dim, k=k,
    ↪omega_base=omega_base, positive_omega=positive_omega)

self.mix = nn.Linear(k, y_dim, bias=False)

self.h0 = nn.Parameter(torch.zeros(h_dim))
self.theta0 = nn.Parameter(torch.zeros(k))

def forward_sequence(self, t: torch.Tensor, y_hist: torch.Tensor, x_exog:
    ↪Optional[torch.Tensor] = None, *, teacher_forcing: bool = True) -> Dict[str,
    ↪torch.Tensor]:
    """
    Forward pass through the NSSDNStateSpace model for a sequence of time
    ↪steps.

    Args:
        t (torch.Tensor): A tensor of shape (N, 1) representing the time
        ↪steps for the sequence.
        y_hist (torch.Tensor): A tensor of shape (N, D) representing the
        ↪historical observations for the sequence, where D is the dimension of the
        ↪output.
        x_exog (Optional[torch.Tensor], optional): A tensor of shape (N, P)
        ↪representing the exogenous inputs for the sequence, where P is the dimension
        ↪of the exogenous input. Defaults to None.
        teacher_forcing (bool, optional): Whether to use teacher forcing
        ↪during the forward pass. If True, the model will use the actual historical
        ↪observations from `y_hist` as input at each time step. If False, the model
        ↪will use its own predictions from the previous time step as input. Default
        ↪is True.

    Raises:

```

*ValueError: If the shapes of `t`, `y\_hist`, or `x\_exog` do not
match the expected dimensions based on the model's configuration.*

Returns:

*Dict[str, torch.Tensor]: A dictionary containing the outputs of the
model for each time step, including:*

- "yhat": The predicted output at each time step.*
- "B": The baseline output from the emission heads at each time
step.*
- "A": The amplitude of the sinusoidal component from the
emission heads at each time step.*
- "omega": The frequency of the sinusoidal component from the
emission heads at each time step.*
- "theta": The phase of the sinusoidal component at each time
step.*
- "phi": The phase shift of the sinusoidal component from the
emission heads at each time step.*
- "h": The hidden state at each time step.*
- "dt": The time step differences between consecutive time
steps.*
- "gate": (optional) The gating values from the emission heads
at each time step, included only if gating is used in the emission heads.*

```
"""  
  
t = ensure_2d(t, "t")  
y_hist = ensure_2d(y_hist, "y_hist")  
if y_hist.shape[1] != self.y_dim:  
    raise ValueError(f"y_hist must have D={self.y_dim}; got {y_hist.  
shape[1]}")  
if x_exog is not None:  
    x_exog = ensure_2d(x_exog, "x_exog")  
    if x_exog.shape[0] != t.shape[0]:  
        raise ValueError("x_exog length must match t")  
  
N = t.shape[0]  
dt = torch.zeros(N, 1, device=t.device, dtype=t.dtype)  
dt[1:] = t[1:] - t[:-1]  
  
h = torch.zeros(N, self.h_dim, device=t.device, dtype=t.dtype)  
theta = torch.zeros(N, self.k, device=t.device, dtype=t.dtype)  
omega = torch.zeros(N, self.k, device=t.device, dtype=t.dtype)  
A = torch.zeros(N, self.k, device=t.device, dtype=t.dtype)  
phi = torch.zeros(N, self.k, device=t.device, dtype=t.dtype)  
B = torch.zeros(N, self.y_dim, device=t.device, dtype=t.dtype)  
yhat = torch.zeros(N, self.y_dim, device=t.device, dtype=t.dtype)  
gate = torch.zeros(N, self.k, device=t.device, dtype=t.dtype)
```

```

h_prev = self.h0.view(1, -1)
theta_prev = self.theta0.view(1, -1)
y_prev = y_hist[0:1]

for n in range(N):
    y_in = y_hist[n:n+1] if teacher_forcing else y_prev

    if x_exog is None:
        x_in = y_in
    else:
        x_in = torch.cat([y_in, x_exog[n:n+1]], dim=1)

    h_n = self.trans(h_prev, x_in, dt[n:n+1])
    params = self.emit(h_n)

    theta_n = theta_prev + params["omega"] * dt[n:n+1]

    s = torch.sin(theta_n + params["phi"])
    osc = self.mix(params["A"] * s)
    y_n = params["B"] + osc

    h[n] = h_n.squeeze(0)
    theta[n] = theta_n.squeeze(0)
    omega[n] = params["omega"].squeeze(0)
    A[n] = params["A"].squeeze(0)
    phi[n] = params["phi"].squeeze(0)
    B[n] = params["B"].squeeze(0)
    yhat[n] = y_n.squeeze(0)
    if "gate" in params:
        gate[n] = params["gate"].squeeze(0)

    h_prev = h_n
    theta_prev = theta_n
    y_prev = y_n.detach()

out = {
    "yhat": yhat,
    "B": B,
    "A": A,
    "omega": omega,
    "theta": theta,
    "phi": phi,
    "h": h,
    "dt": dt
}

```

```

if gate.abs().sum().item() > 0:
    out["gate"] = gate
return out

```

2.10 Distributional residual head

DeepAR models a predictive distribution (not only point forecasts). (Salinas et al., 2019)

We add a residual head predicting $\log \sigma_n$ from h_n and train under Gaussian NLL.

```

[9]: class ResidualScaleHead(nn.Module):
    """A simple feedforward network that takes the hidden state as input and
    produces a residual output to be added to the main output of the model.

    Attributes:
        net (nn.Sequential): The feedforward network used to compute the
        residual output from the hidden state.
    """

    def __init__(self, h_dim: int, y_dim: int) -> None:
        """Initialize the ResidualScaleHead module.

        Args:
            h_dim (int): The dimension of the hidden state.
            y_dim (int): The dimension of the output.
        """

        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(h_dim, 32),
            nn.Tanh(),
            nn.Linear(32, y_dim),
        )

    def forward(self, h: torch.Tensor) -> torch.Tensor:
        """Forward pass through the ResidualScaleHead module.

        Args:
            h (torch.Tensor): The hidden state tensor of shape (N, h_dim).

        Returns:
            torch.Tensor: The residual output tensor of shape (N, y_dim).
        """

        return self.net(h)

def gaussian_nll(y: torch.Tensor, mu: torch.Tensor, log_sigma: torch.Tensor) ->
torch.Tensor:

```

```

"""Compute the Gaussian negative log-likelihood loss.

Args:
    y (torch.Tensor): The target tensor of shape (N, y_dim).
    mu (torch.Tensor): The predicted mean tensor of shape (N, y_dim).
    log_sigma (torch.Tensor): The predicted log standard deviation tensor
    of shape (N, y_dim).

Returns:
    torch.Tensor: The mean Gaussian negative log-likelihood loss.
    """

sigma = torch.exp(log_sigma).clamp_min(1e-6)
return (0.5 * math.log(2.0 * math.pi) + log_sigma + 0.5 * ((y - mu) /
sigma).pow(2)).mean()

```

2.11 Regularizers: smoothness and drift priors

TVP-VARs motivate our desire to restrict how quickly parameters evolve. (Lubik & Matthes, 2015)
We implement: - envelope drift $\|\Delta A\|^2$, - frequency drift $\|\Delta \omega\|^2$, - optional phase curvature $\|\Delta^2 \theta\|^2$.

```

[10]: def diff1(x: torch.Tensor) -> torch.Tensor:
    """Compute the first-order difference of the input tensor `x` along the
    first dimension.

    Args:
        x (torch.Tensor): The input tensor of shape (N, ...), where N is the
        number of time steps.

    Returns:
        torch.Tensor: The first-order difference of `x`, computed as x[1:] - x[:
        -1], resulting in a tensor of shape (N-1, ...).
    """
    return x[1:] - x[:-1]

def diff2(x: torch.Tensor) -> torch.Tensor:
    """Compute the second-order difference of the input tensor `x` along the
    first dimension.

    Args:
        x (torch.Tensor): The input tensor of shape (N, ...), where N is the
        number of time steps.

    Returns:
        torch.Tensor: The second-order difference of `x`, computed as x[2:] - 2
        * x[1:-1] + x[:-2], resulting in a tensor of shape (N-2, ...).

```

```

    """

    return x[2:] - 2.0 * x[1:-1] + x[: -2]

def smoothness_losses(out: Dict[str, torch.Tensor]) -> Dict[str, torch.Tensor]:
    """Compute smoothness losses for the parameters of the output distribution.

    Args:
        out (Dict[str, torch.Tensor]): A dictionary containing the output
        ↪ parameters "A", "omega", and "theta".

    Returns:
        Dict[str, torch.Tensor]: A dictionary containing the smoothness losses
        ↪ "pA", "pOmega", and "pTheta2".
    """

    A = out["A"]
    omega = out["omega"]
    theta = out["theta"]

    pA = diff1(A).pow(2).mean()
    pOmega = diff1(omega).pow(2).mean()
    pTheta2 = diff2(theta).pow(2).mean() if theta.shape[0] >= 3 else theta.
    ↪ new_tensor(0.0)
    return {"pA": pA, "pOmega": pOmega, "pTheta2": pTheta2}

```

2.12 Frequency-domain auxiliary loss and evaluation

Spectral-density estimation work motivates evaluating in the frequency domain, not only MSE. (Mohammadi et al., 2026)

We implement a lightweight PSD loss via rFFT power spectra.

```

[11]: def psd_loss(y: torch.Tensor, yhat: torch.Tensor, *, n_fft: int = 256) -> torch.
    ↪ Tensor:
    """Compute the power spectral density (PSD) loss between the target and
    ↪ predicted signals.

    Args:
        y (torch.Tensor): The target signal tensor of shape (N, D), where N is
        ↪ the number of time steps and D is the number of channels.
        yhat (torch.Tensor): The predicted signal tensor of shape (N, D), where
        ↪ N is the number of time steps and D is the number of channels.
        n_fft (int, optional): The number of FFT points. Defaults to 256.

    Returns:

```

torch.Tensor: The mean squared error loss between the PSDs of the
target and predicted signals.

"""

```
y = ensure_2d(y, "y")
yhat = ensure_2d(yhat, "yhat")
losses = []
for d in range(y.shape[1]):
    Y = torch.fft.rfft(y[:, d], n=n_fft)
    Yh = torch.fft.rfft(yhat[:, d], n=n_fft)
    psd = (Y.abs() ** 2)
    psdh = (Yh.abs() ** 2)
    losses.append(F.mse_loss(psdh, psd))
return torch.stack(losses).mean()
```

2.13 Deterministic Training loop and optional NLL

Combines: - MSE or Gaussian NLL, - smoothness losses, - optional PSD loss.

Optimizer: Adam baseline.

```
[12]: def train_state_space_nssdn(model: NSSDNStateSpace, batch: TimeSeriesBatch, *,
    y_scaler: Optional[MapMinMax] = None,
    x_scaler: Optional[MapMinMax] = None, lr: float =
1e-3, steps: int = 2000, teacher_forcing: bool = True,
    use_nll: bool = False, lam_A: float = 1e-3,
    lam_omega: float = 1e-3, lam_theta2: float = 0.0,
    lam_psd: float = 0.0, n_fft: int = 256, device:
Optional[torch.device] = None) -> Dict[str, Any]:
    """Train the NSSDNStateSpace model on a given batch of time series data.

    Args:
        model (NSSDNStateSpace): The NSSDNStateSpace model to be trained.
        batch (TimeSeriesBatch): A batch of time series data containing time
        steps, target values, and optional exogenous inputs.
        y_scaler (Optional[MapMinMax], optional): An optional MapMinMax scaler
        for the target values. If provided, the target values will be scaled before
        training. Defaults to None.
        x_scaler (Optional[MapMinMax], optional): An optional MapMinMax scaler
        for the exogenous inputs. If provided, the exogenous inputs will be scaled
        before training. Defaults to None.
        lr (float, optional): The learning rate for the optimizer. Default is
1e-3.
        steps (int, optional): The number of training steps to perform. Default
is 2000.
```

```

    teacher_forcing (bool, optional): Whether to use teacher forcing during
    ↪ training. If True, the model will use the actual target values from the
    ↪ batch as input at each time step. If False, the model will use its own
    ↪ predictions from the previous time step as input. Default is True.

    use_nll (bool, optional): Whether to use the Gaussian negative
    ↪ log-likelihood loss instead of mean squared error. If True, the model will
    ↪ predict both the mean and log standard deviation of the output distribution,
    ↪ and the loss will be computed using the Gaussian NLL. If False, the loss
    ↪ will be computed using mean squared error. Default is False.

    lam_A (float, optional): The regularization weight for the amplitude
    ↪ smoothness loss. Default is 1e-3.

    lam_omega (float, optional): The regularization weight for the
    ↪ frequency smoothness loss. Default is 1e-3.

    lam_theta2 (float, optional): The regularization weight for the phase
    ↪ smoothness loss. Default is 0.0.

    lam_psd (float, optional): The regularization weight for the power
    ↪ spectral density loss. Default is 0.0.

    n_fft (int, optional): The number of FFT points for the PSD loss.
    ↪ Default is 256.

    device (Optional[torch.device], optional): The device to run the
    ↪ training on. If None, will use CUDA if available, else CPU. Default is None.
    """

    device = device or (torch.device("cuda") if torch.cuda.is_available() else
    ↪ torch.device("cpu"))
    model = model.to(device)

    t = batch.t.to(device)
    y = batch.y.to(device)
    x = batch.x.to(device) if batch.x is not None else None

    y_train = y_scaler.transform(y) if y_scaler is not None else y
    x_train = x_scaler.transform(x) if (x is not None and x_scaler is not None)
    ↪ else x

    resid_head = ResidualScaleHead(model.h_dim, model.y_dim).to(device) if
    ↪ use_nll else None
    params = list(model.parameters()) + ([ if resid_head is None else
    ↪ list(resid_head.parameters())])
    opt = torch.optim.Adam(params, lr=lr)

    hist = {"loss": []}

    for step in range(steps):
        opt.zero_grad(set_to_none=True)

```

```

        out = model.forward_sequence(t, y_train, x_train,
↪teacher_forcing=teacher_forcing)
        yhat = out["yhat"]

        if use_nll:
            log_sigma = resid_head(out["h"])
            loss_main = gaussian_nll(y_train, yhat, log_sigma)
        else:
            loss_main = F.mse_loss(yhat, y_train)

        regs = smoothness_losses(out)
        loss = loss_main + lam_A * regs["pA"] + lam_omega * regs["pOmega"] +
↪lam_theta2 * regs["pTheta2"]

        if lam_psd > 0.0:
            loss = loss + lam_psd * psd_loss(y_train, yhat, n_fft=n_fft)

        loss.backward()
        opt.step()

        if step % 100 == 0 or step == steps - 1:
            hist["loss"].append((step, float(loss.detach().cpu())))

    return {"model": model, "resid_head": resid_head, "history": hist}

```

2.14 Minimal synthetic sanity-check

Verifies: - time-aware state update, - emission, - phase integration, - learning dynamics.

Replace with macro data after validation.

```

[14]: torch.manual_seed(7)
device = torch.device("cuda") if torch.cuda.is_available() else torch.
↪device("cpu")

N = 400
t = make_time_grid(N, dt=1.0, device=device)

A_true = 0.3 + 0.25 * torch.sigmoid((t - 200) / 30)
omega_true = 0.03 + 0.00005 * t
theta_true = torch.cumsum(omega_true.squeeze(1), dim=0).view(-1, 1)
trend_true = 0.001 * (t - 200)

y = trend_true + A_true * torch.sin(theta_true)
y = y + 0.02 * torch.randn_like(y)

batch = TimeSeriesBatch(t=t.cpu(), y=y.cpu())
y_scaler = MapMinMax.fit(batch.y)

```

```

model = NSSDNStateSpace(y_dim=1, x_dim=0, h_dim=32, k=6,
    ↳transition="timeaware", positive_omega=False, use_gating=True)

res = train_state_space_nssdn(
    model,
    batch,
    y_scaler=y_scaler,
    steps=1000,
    lr=1e-3,
    teacher_forcing=True,
    use_nll=False,
    lam_A=1e-3,
    lam_omega=1e-3,
    lam_theta2=0.0,
    lam_psd=0.0,
)

out = res["model"].forward_sequence(batch.t.to(device), y_scaler.
    ↳transform(batch.y.to(device)), teacher_forcing=True)
float(out["yhat"].mean().detach().cpu()), float(out["yhat"].std().detach().
    ↳cpu())

```

[14]: (3.217433214187622, 6.2181396484375)