Deep Learning for Time-series Forecasting

Faculty of DS & AI
Spring semester, 2026

Trong-Nghia Nguyen

Business AI Lab

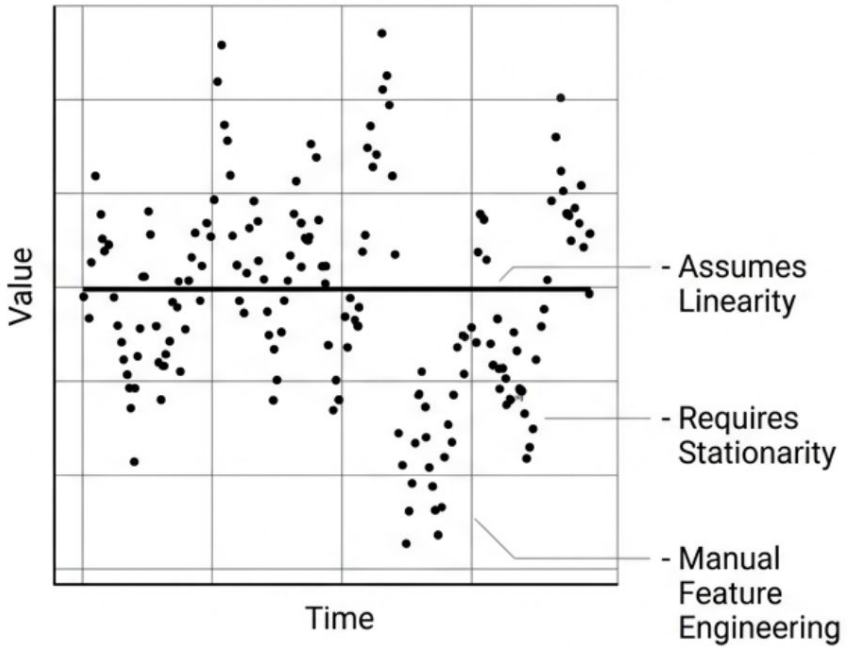
Content

- **Deep Learning Architectures for Time Series**
- Data preprocessing and tensor structure
- Model Implementation
- Advanced libraries

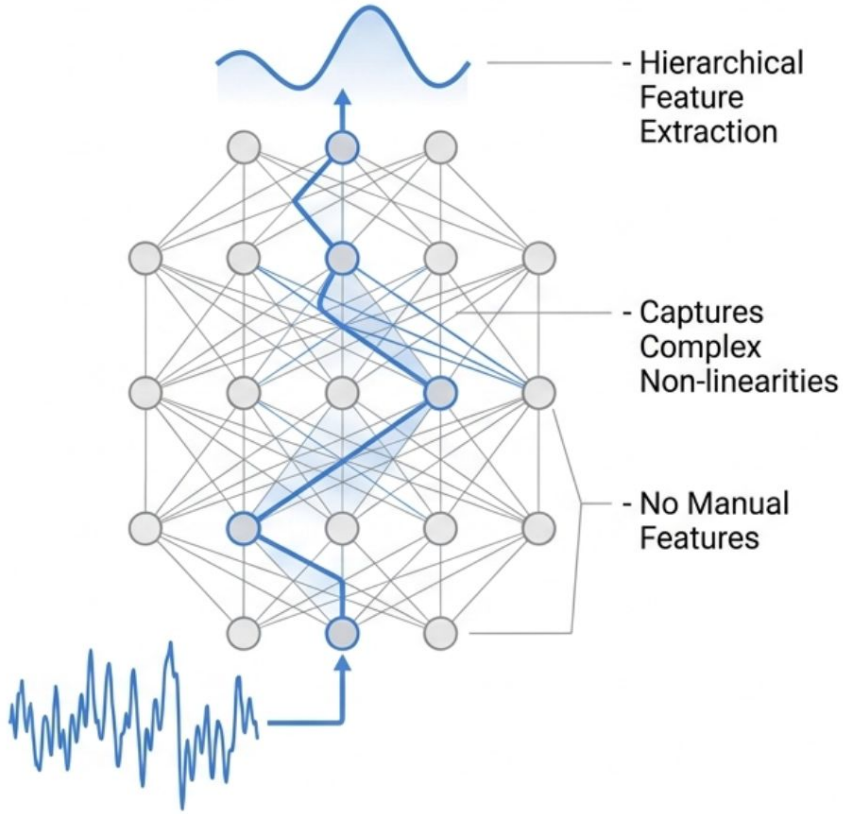
Deep Learning Architectures for Time Series

Motivation

Statistical Baselines

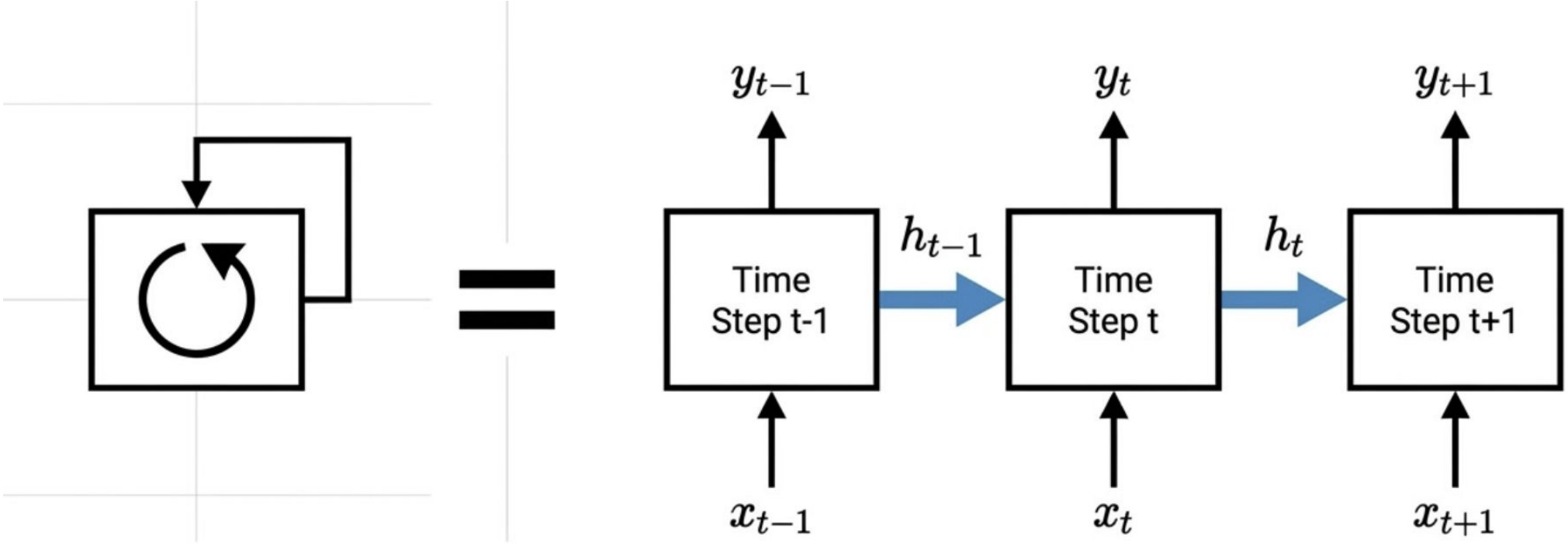


Representation Learning



Deep Learning Architectures for Time Series

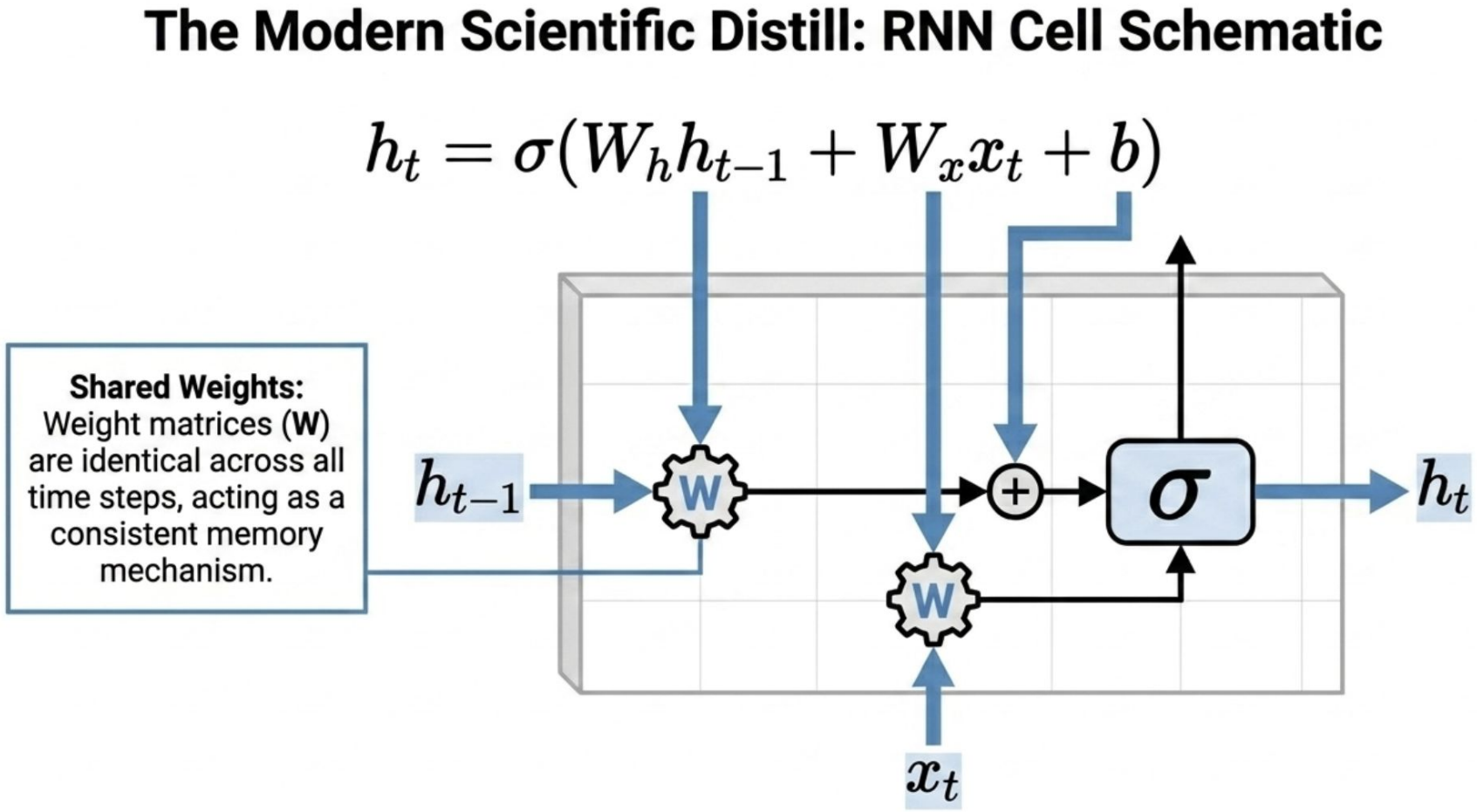
Recurrent Neural Network



RNNs introduce a loop connection. Unlike feedforward networks, this architecture allows information to persist across time steps, making it purpose-built for sequential data.

Deep Learning Architectures for Time Series

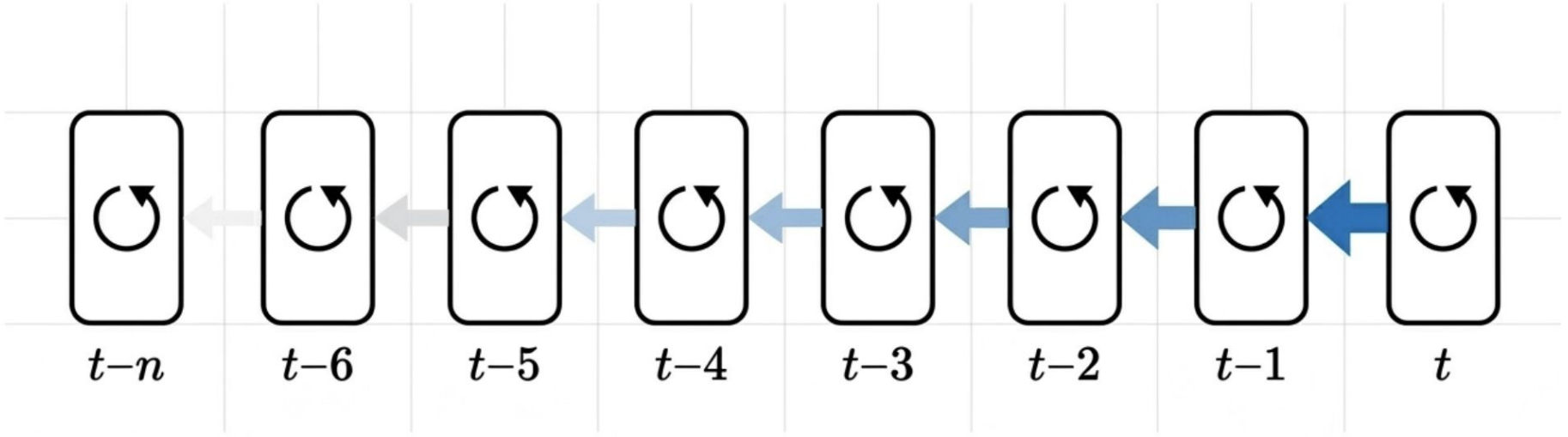
Recurrent Neural Network



Deep Learning Architectures for Time Series

Recurrent Neural Network

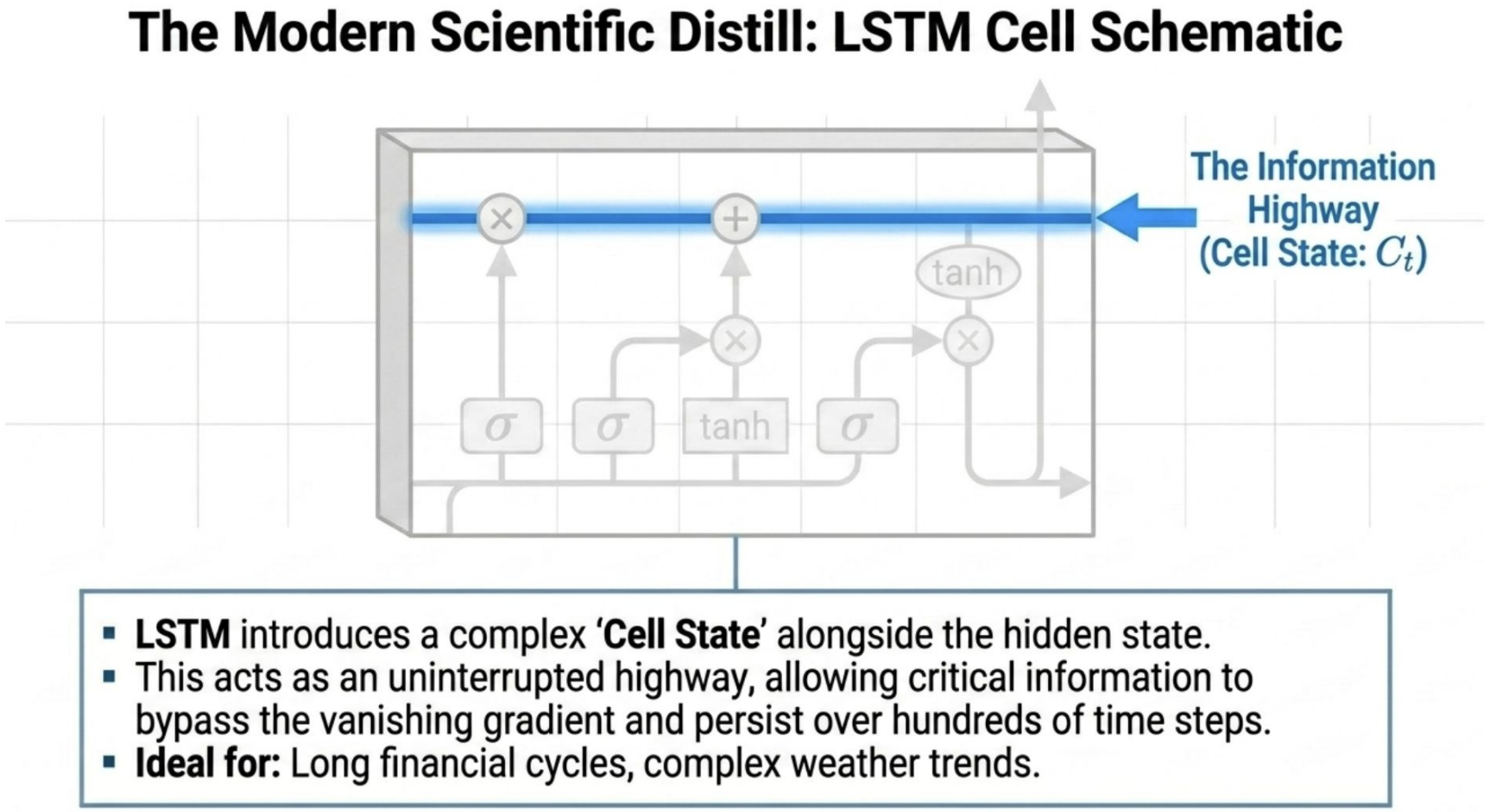
The Bottleneck: The Vanishing Gradient Problem



- **The Flaw:** When sequences are **too long**, information from distant past steps degrades to near zero during backpropagation.
- **The Impact:** The network structurally fails to learn long-term dependencies.

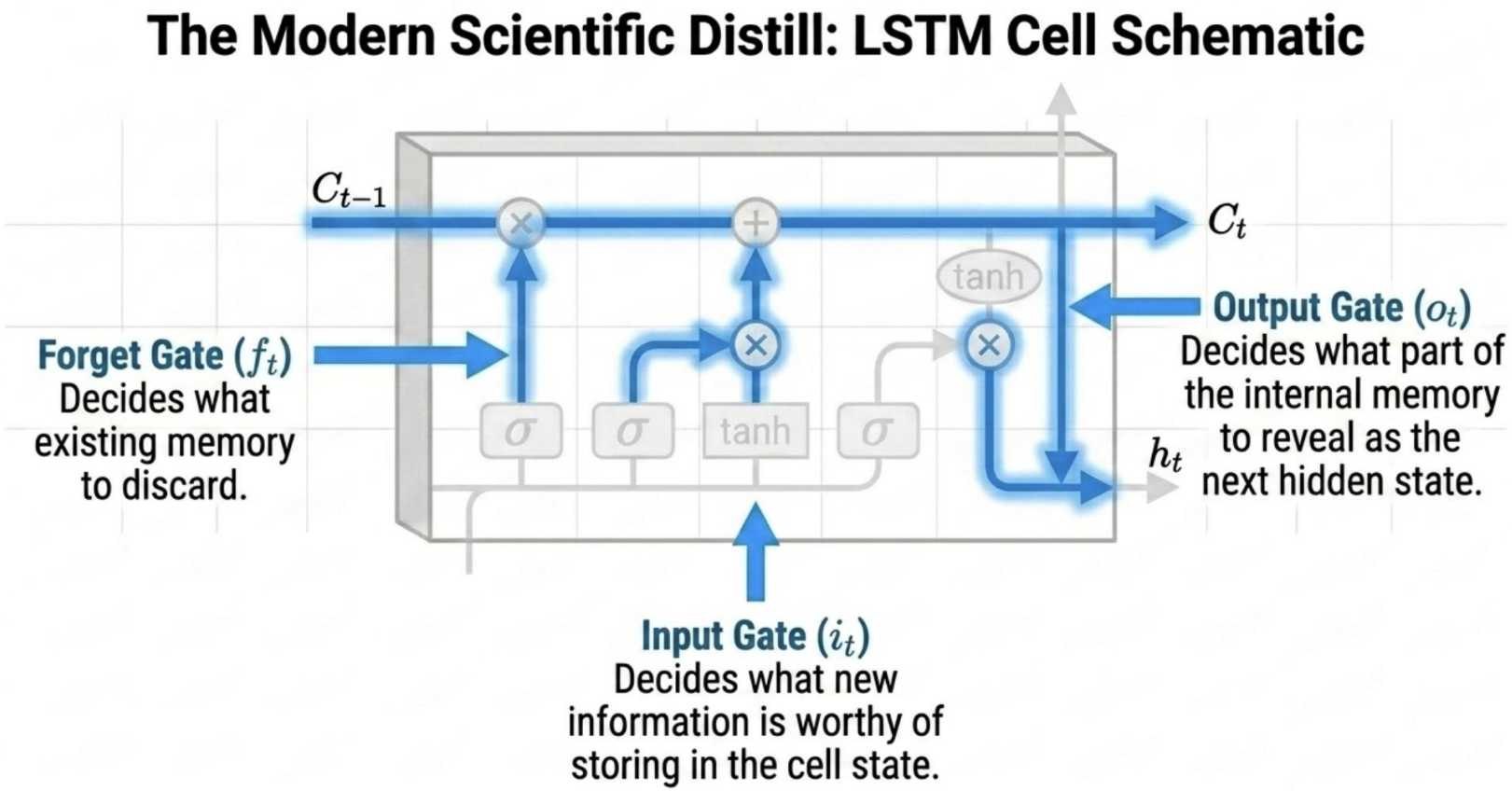
Deep Learning Architectures for Time Series

Long-short term memory



Deep Learning Architectures for Time Series

Long-short term memory

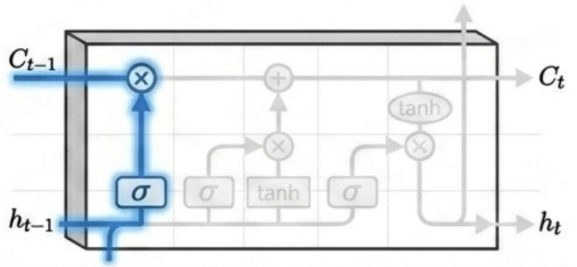


Deep Learning Architectures for Time Series

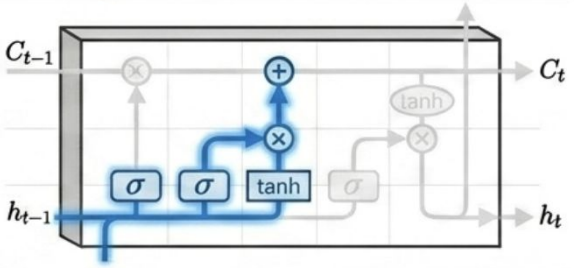
Long-short term memory

The Modern Scientific Distill: LSTM Gate Equations and Micro-Schematics

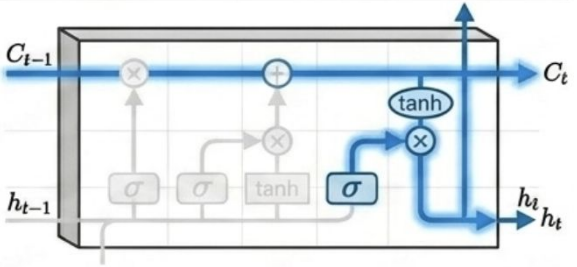
Forget Gate: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$



Input Gate: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

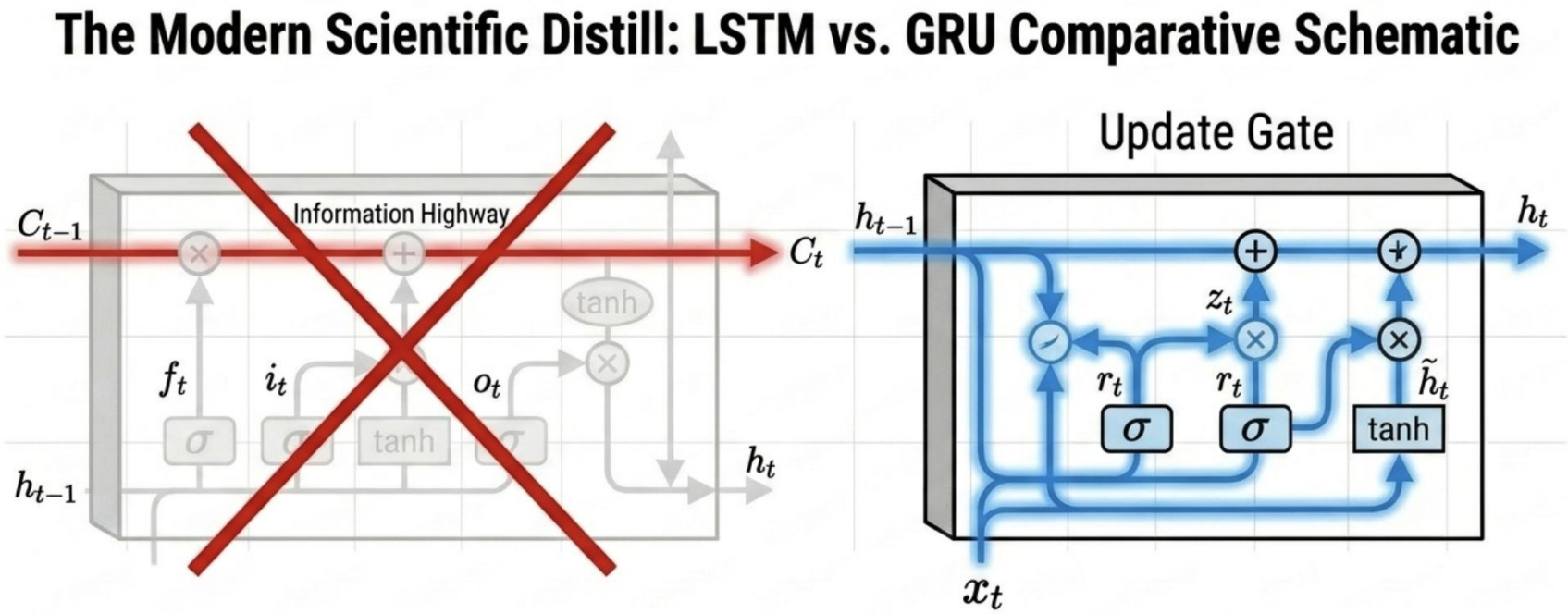


Output Gate: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$



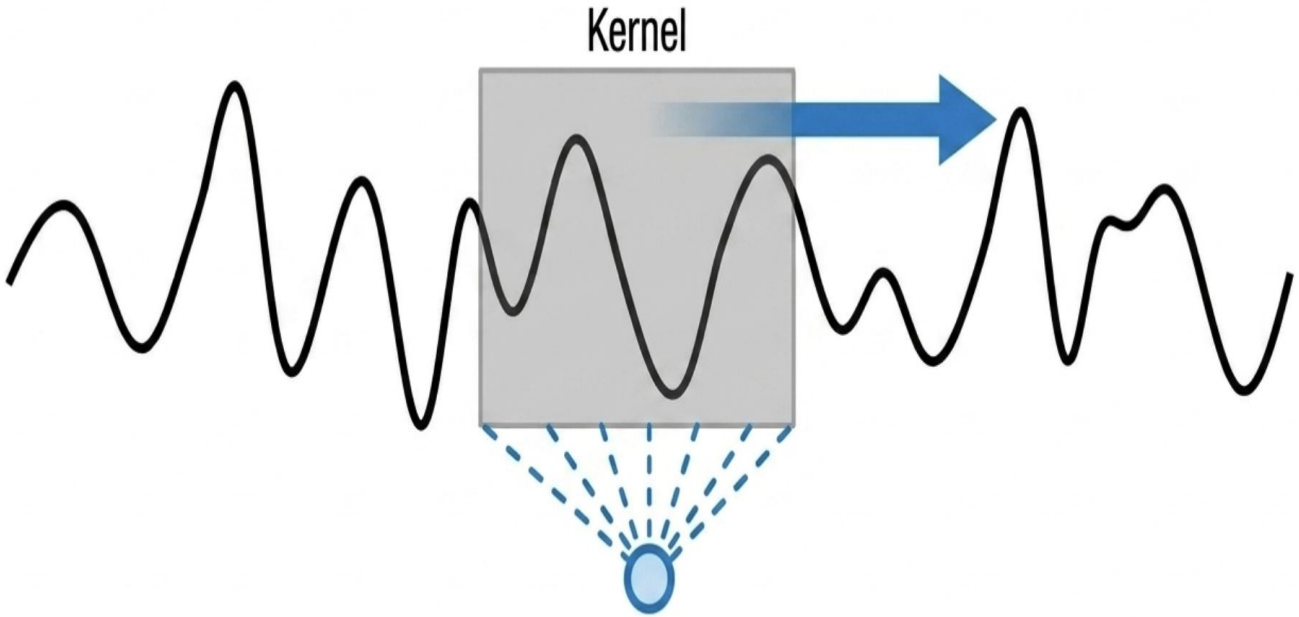
Deep Learning Architectures for Time Series

Gated Recurrent Unit

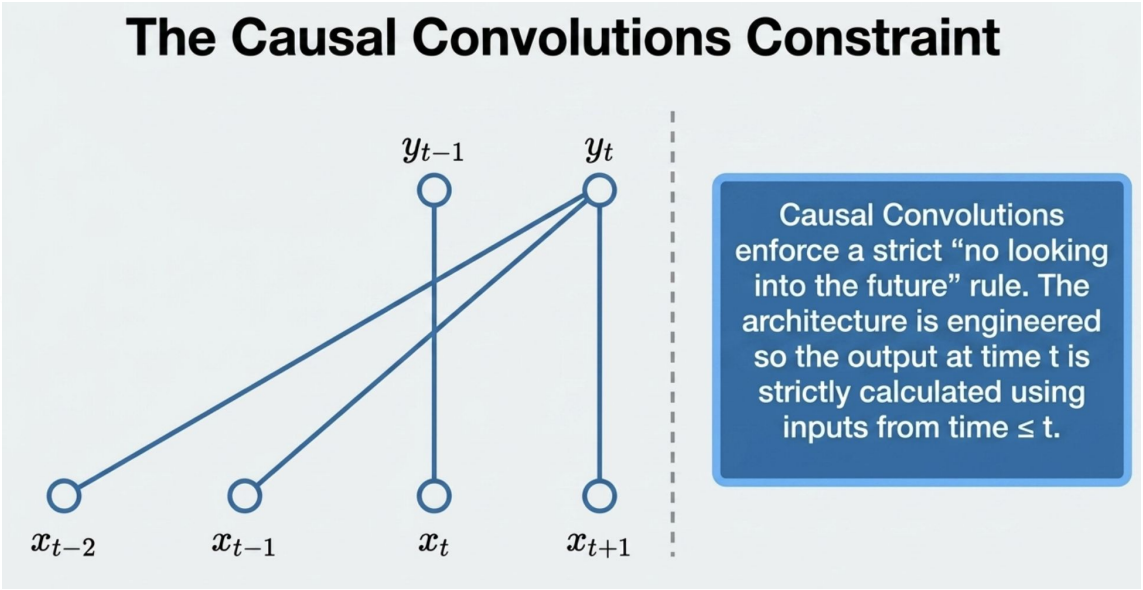


Deep Learning Architectures for Time Series

1-D CNN



- **Paradigm Shift:** Instead of processing sequentially step-by-step, 1D-CNNs use sliding filters to detect local patterns simultaneously.
- **Massive parallelization on GPUs** (bypassing the sequential bottleneck of RNNs).
- **Extreme robustness** to noisy, erratic data.



Deep Learning Architectures for Time Series

Comparison

Architecture	Strengths	Weaknesses	Best Use Cases
RNN	Simple, fast for short sequences.	Vanishing gradient, fails at long dependencies.	Simple problems with very short dependencies.
LSTM	Exceptional at learning long-term dependencies, stable.	Computationally complex, heavy resource cost, slower.	Financial forecasting, weather, complex long-cycle trends.
GRU	Highly efficient, fewer parameters, fast.	Can underperform LSTM on ultra-complex data.	Scenarios requiring optimal speed and memory efficiency.
1D-CNN	Fast feature extraction, GPU parallelization, noise resistant.	Requires careful tuning of window / filter size.	Detecting local patterns, ultra-long sequences, highly noisy data.

Selection depends entirely on the Inductive Bias required by the specific dataset and computational constraints.

Content

- Deep Learning Architectures for Time Series
- **Data preprocessing and tensor structure**
- Model Implementation
- Advanced libraries

Data preprocessing and tensor structure

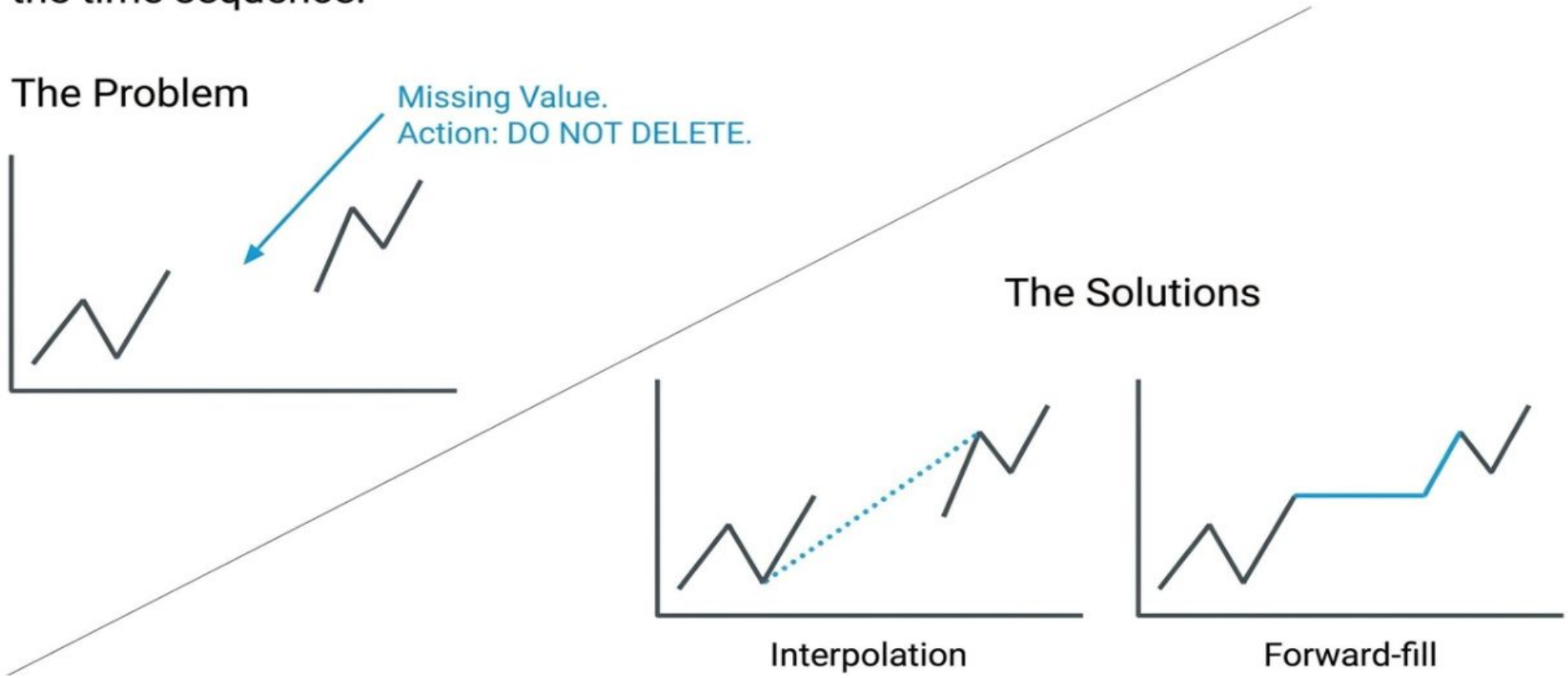
Definition

Deep learning models cannot process raw sequential data like traditional statistics. They demand strict temporal logic, scaled numerical stability, and specific geometric tensor shapes to function.

Data preprocessing and tensor structure

Clean

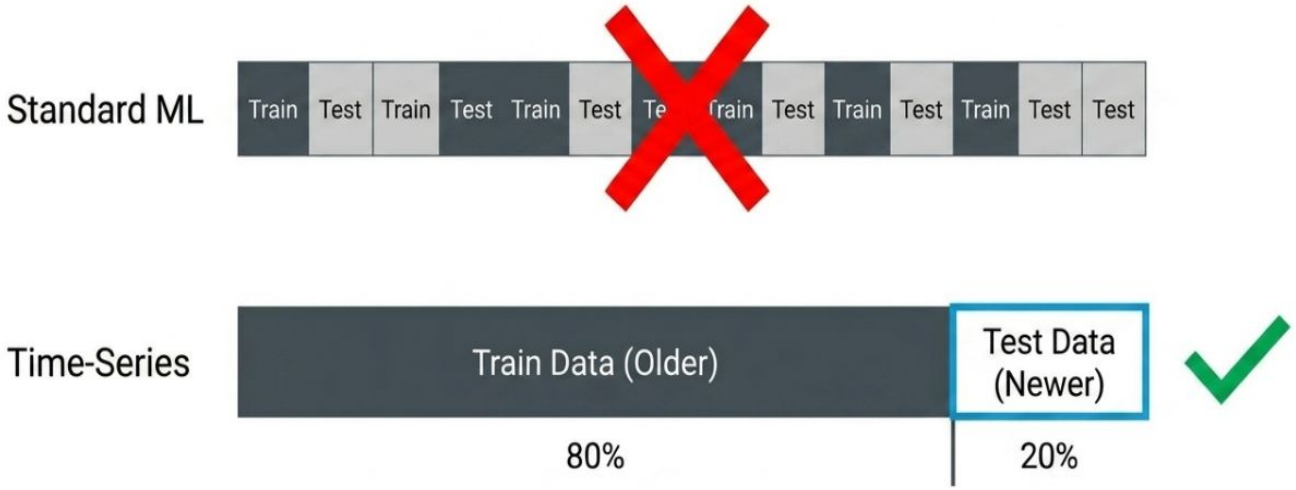
Handling missing values requires temporal continuity. Deleting rows breaks the time sequence.



Data preprocessing and tensor structure

Split

Time-series splitting fundamentally differs from image classification. You absolutely must preserve chronological order.



Validation sets must be extracted from the latest period of the Train set to tune hyperparameters before facing the final, unseen Test set. TimeSeriesSplit (Walk-forward validation) allows evaluation across multiple timeframes.

Data preprocessing and tensor structure

Scale

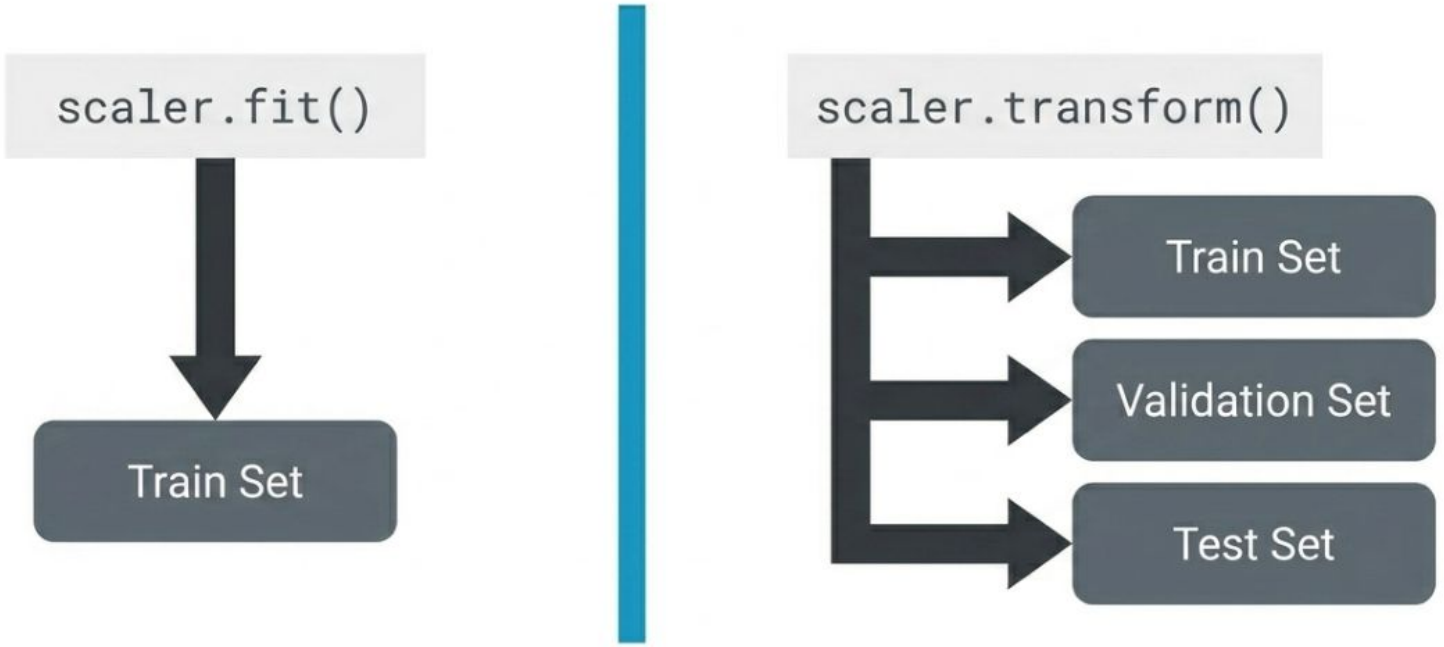
Neural networks update weights via Gradient Descent, making them highly sensitive to input data scales.

Feature	MinMaxScaler	StandardScaler
Method	Normalization	Standardization
Formula	$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$	$z = \frac{x - \mu}{\sigma}$
Target Range	[0, 1]	μ=0, σ=1
Ideal Use Case	RNN / LSTM architectures	Data with normal distribution or high noise

Data preprocessing and tensor structure

Scale

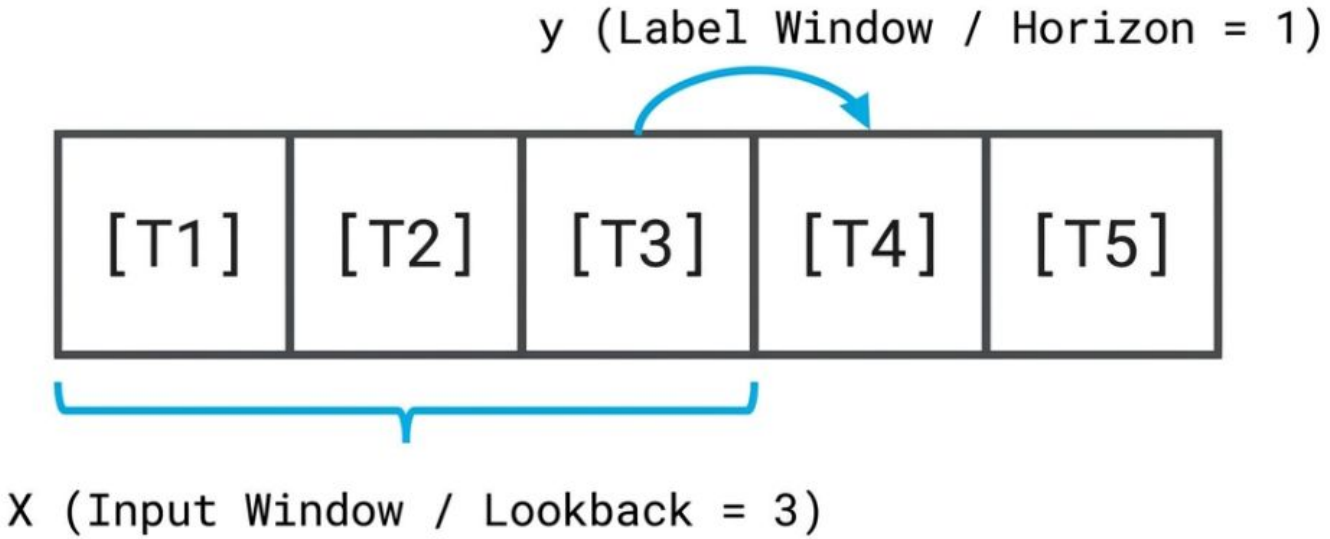
The Golden Rule of Scaling. Fitting the scaler on the entire dataset allows the model to peek at future minimum/maximum values, artificially inflating performance. Fit ONLY on the Train set.



Data preprocessing and tensor structure

Window

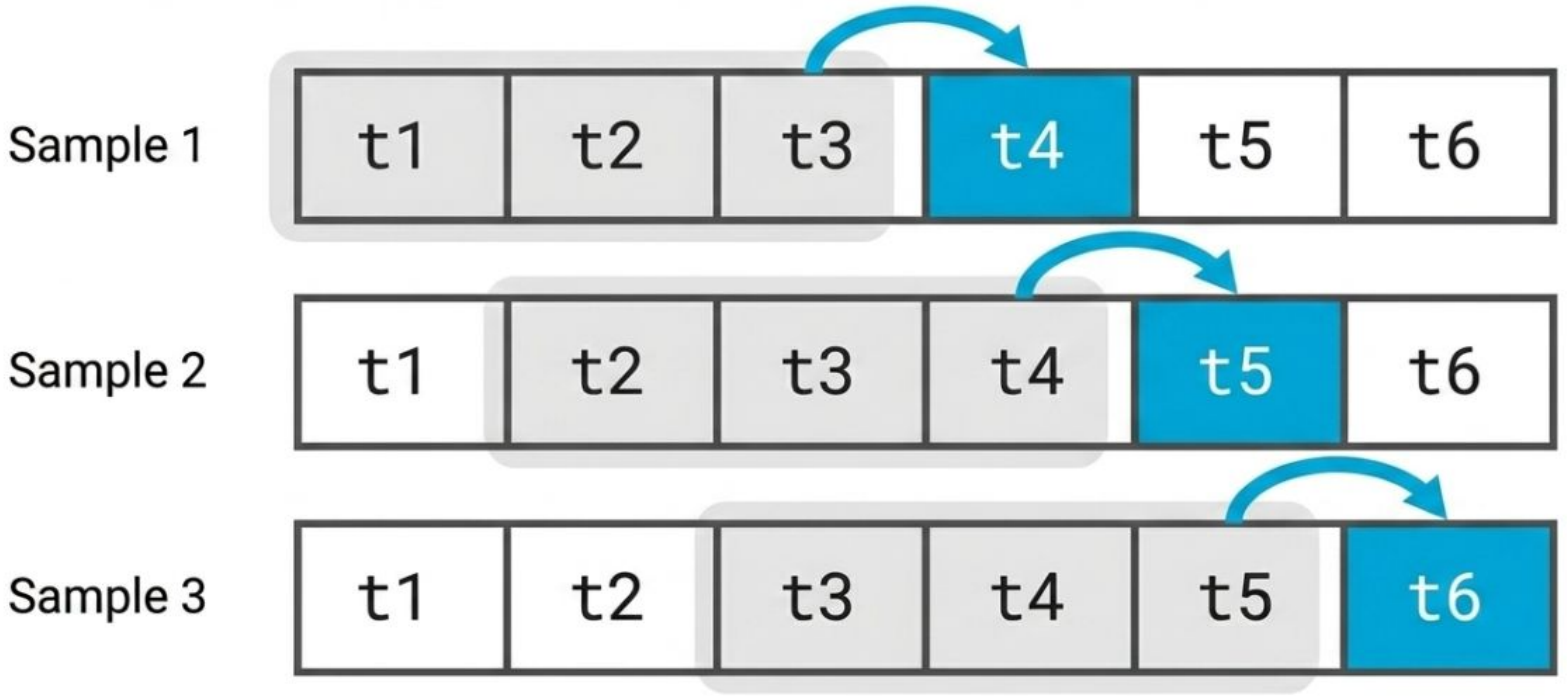
Supervised learning requires (X, y) pairs. The sliding window technique converts raw 1D time-series into bounded past observations (Input) to predict future targets (Label).



Data preprocessing and tensor structure

Window

A Stride (step size) dictates how far the window shifts after each sample. A stride of 1 maximizes data utility, generating a new training sample for every single time step.

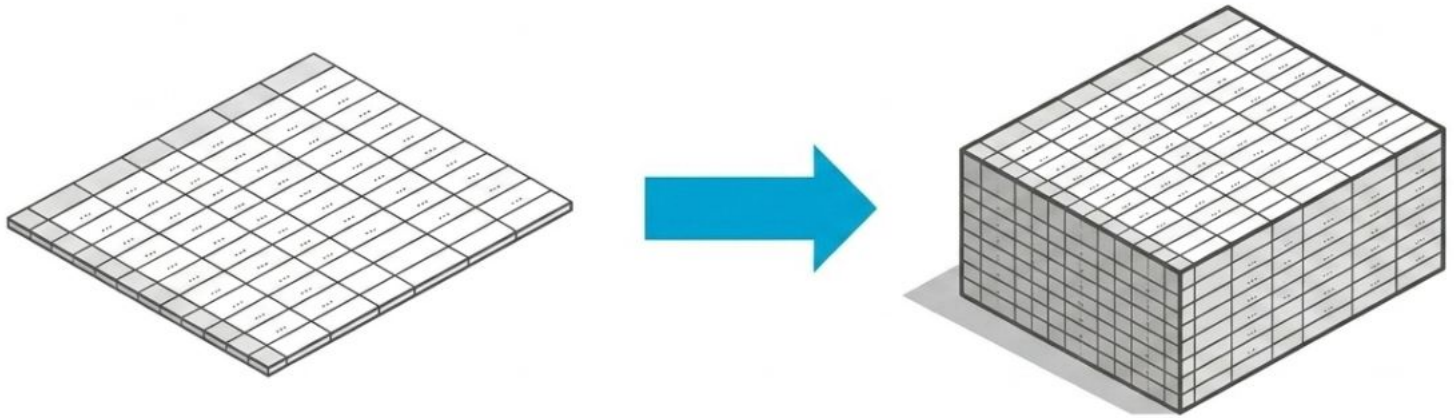


Data preprocessing and tensor structure

Batch

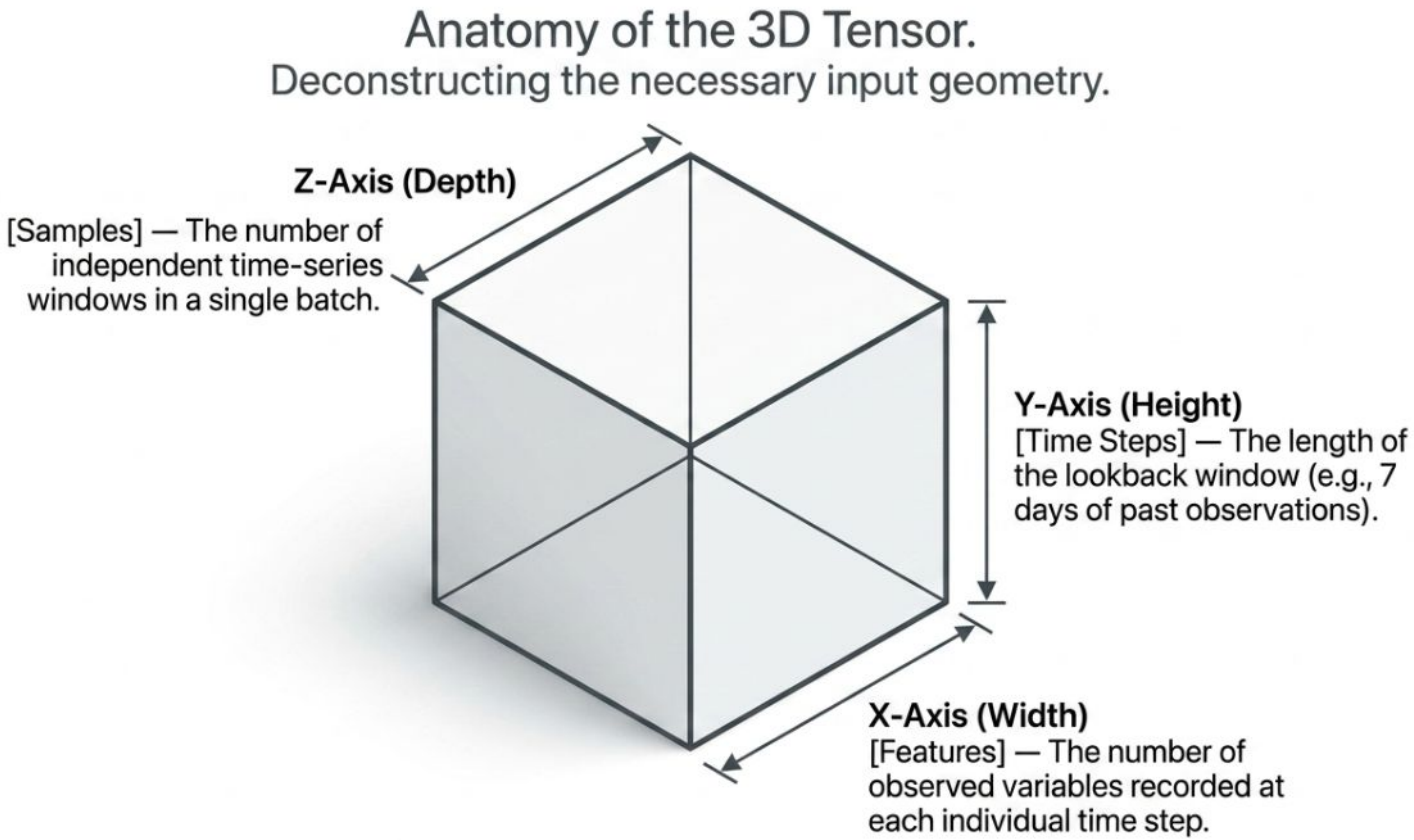
Recurrent and Convolutional architectures (SimpleRNN, LSTM, GRU, Conv1D) cannot ingest 1D sequences or 2D tables. The data must be structurally reshaped into a 3-Dimensional Tensor.

```
Shape = [Samples, Time_Steps, Features]
```



Data preprocessing and tensor structure

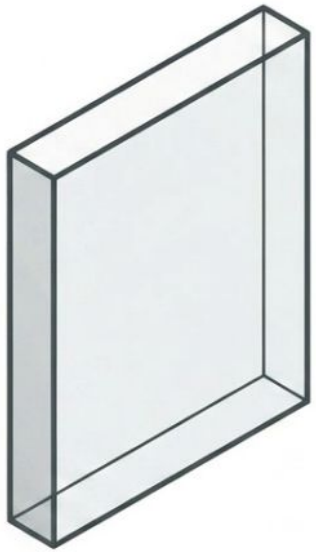
Batch



Data preprocessing and tensor structure

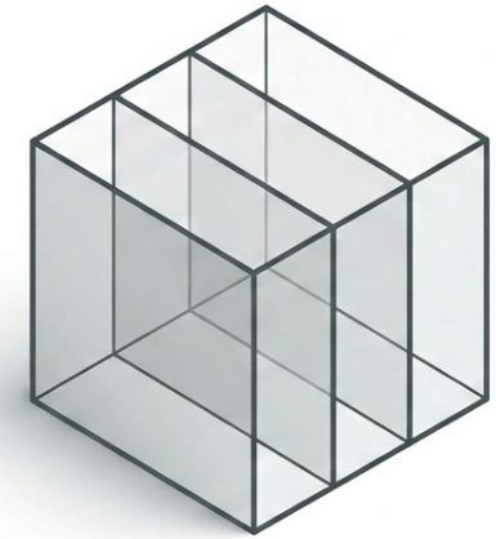
Batch

Tensor depth depends on the complexity
of the recorded variables at each time step.



Univariate (Features = 1)

e.g., Stock Price only.



Multivariate (Features = 3)

e.g., Stock Price, Trading Volume, Sentiment Index.

Data preprocessing and tensor structure

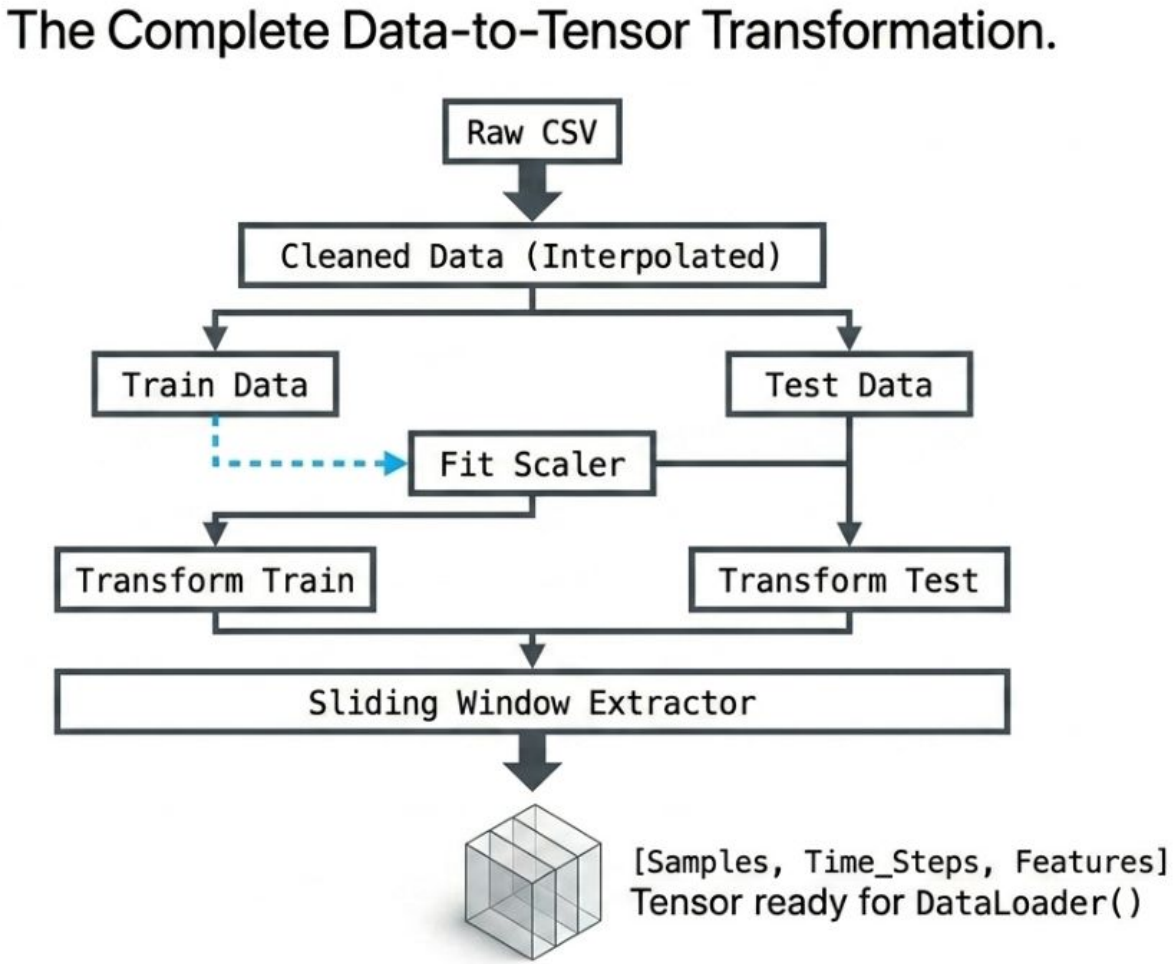
Pipeline

Translating architectural constraints
into Python code.

```
# 1. Chronological Split
X_train, X_test = train_test_split(data, test_size=0.2, shuffle=False)

# 2. Prevent Data Leakage
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 3. Construct 3D Tensor
X_tensor = np.array(X_windows).reshape(samples, time_steps, features)
```

Data preprocessing and tensor structure

Pipeline

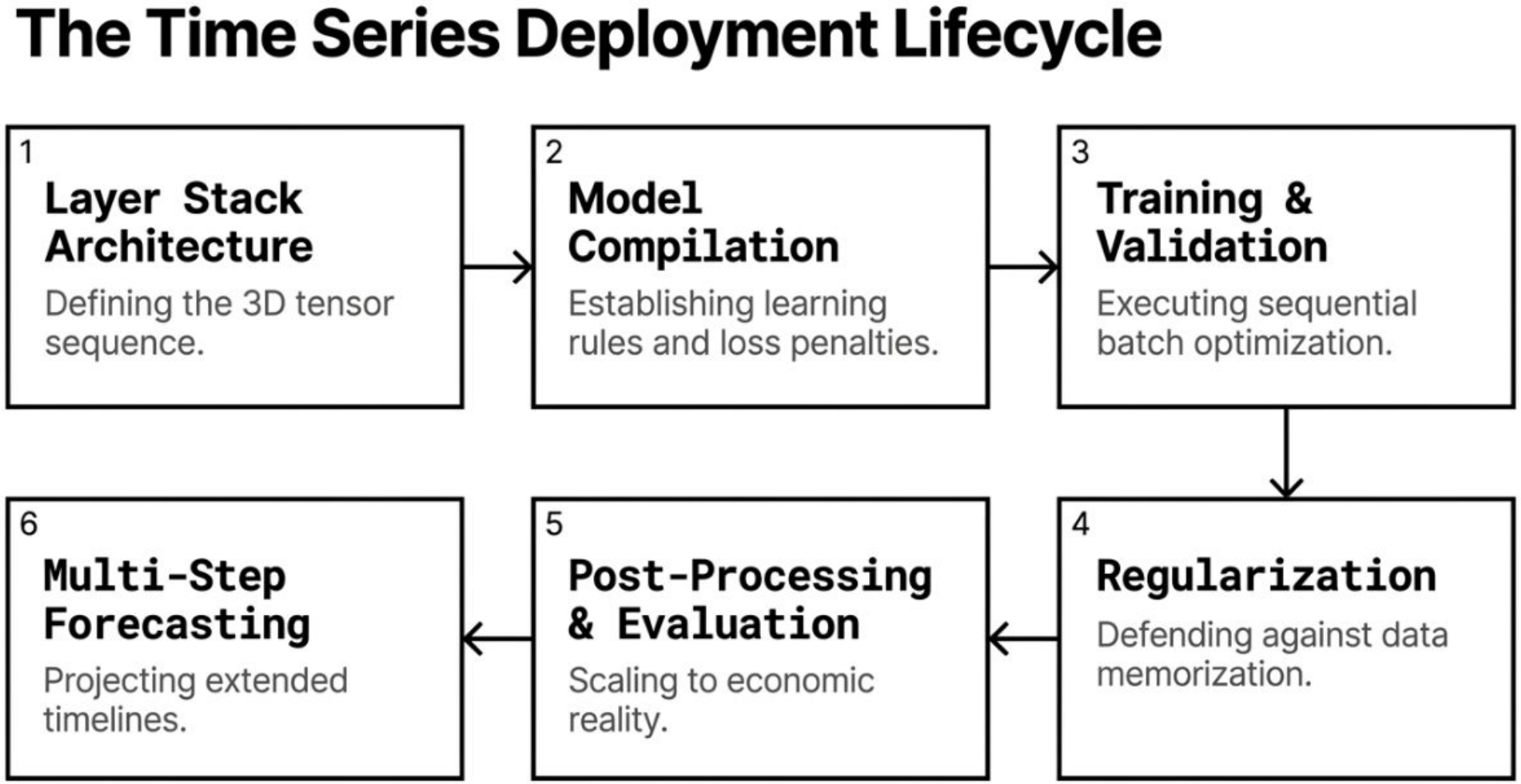


Content

- Deep Learning Architectures for Time Series
- Data preprocessing and tensor structure
- **Model Implementation**
- Advanced libraries

Model Implement

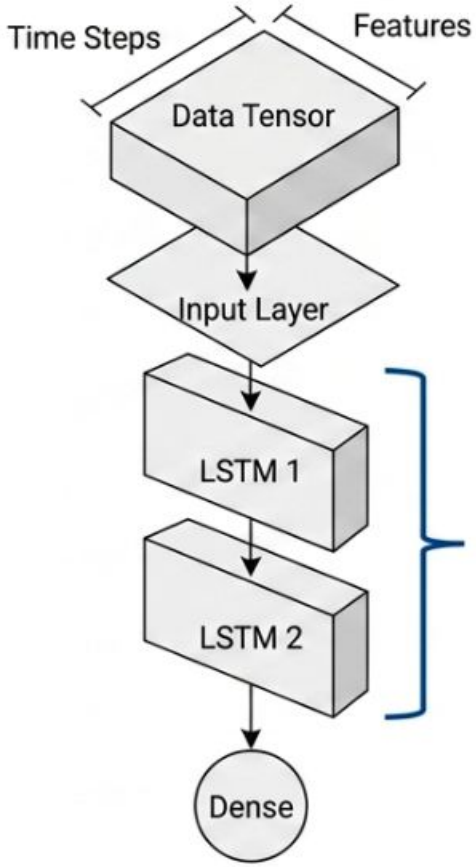
Deployment



Model Implement

Architecture

Constructing the Sequential Network



- `input_shape`
  Mandatory definition of past window size and input variables. Note: Exclude Batch Size at this layer.

- `return_sequences`
  - **`True`**
    Returns the full hidden state sequence. Required when stacking an LSTM into another LSTM.
  - **`False`** `(Default)`
    Returns only the final hidden state. Executed before passing to the Dense output layer.

- `Dense(1)`
  Final output layer applying linear activation for single-value regression predictions.

Model Implement

Compilation

Compilation defines how the model evaluates its mistakes and updates its weights.

Loss Functions (The Penalty)	Optimizer (The Engine)	Metrics (The Tracker)
`MSE` - Mean Squared Error: Penalizes large errors heavily.	`Adam` - Adaptive Moment Estimation: The industry standard. Adaptively adjusts learning rates for individual network weights.	`MAE` `or` `RMSE` - Human-readable performance tracking during training iterations.
`MAE` - Mean Absolute Error: Less sensitive to noise and outliers.		
`Huber` - Hybrid approach; stabilizes extreme data volatility.		

Model Implement

Training

Training Execution & The Sequence Mandate

Hyperparameter Grid

Epochs

Full dataset traversals. Warning: High epochs risk overfitting historical noise.

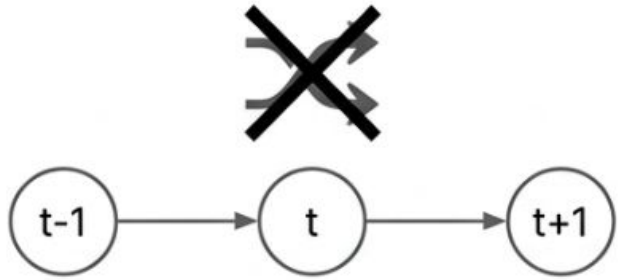
Batch Size

Samples processed **before** weight updates. Smaller batches (32, 64) equal better convergence but slower computation.

Validation Split

Reserving the final **10-20% of chronological data** as a pristine testing ground.

The Sequence Mandate



`shuffle=False`

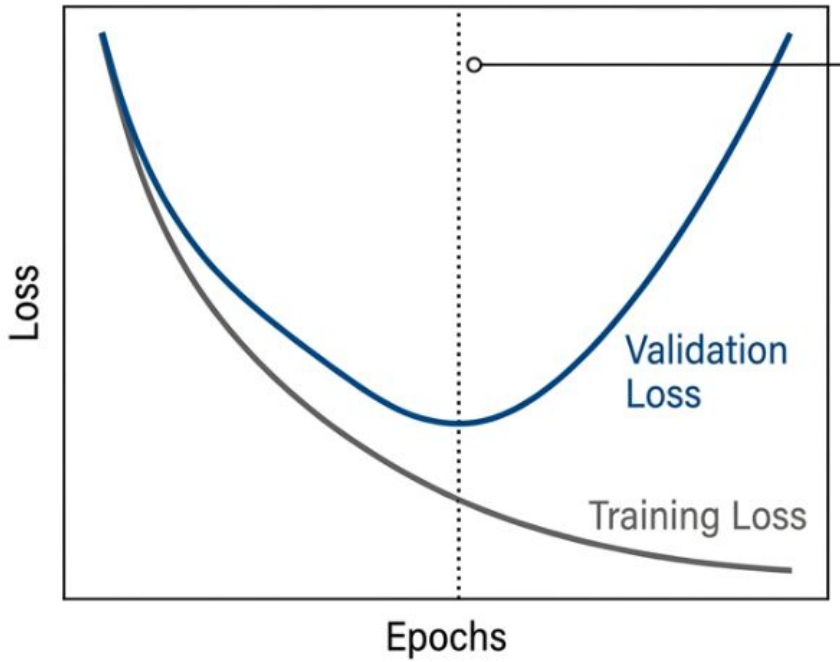
Unlike image or text classification, **temporal order** is the **primary feature**. Shuffling **destroys** the sequential dependencies that Recurrent/LSTM networks rely on.

Model Implement

Regularization

Controlling Overfitting in Small Datasets

Time series data is often small compared to computer vision datasets, leaving Deep Learning highly susceptible to memorizing historical noise rather than underlying patterns.



`EarlyStopping`

Halts training when validation loss stops improving.

`Dropout Layer`

Randomly severs neuron connections to force independent feature learning.

`Learning Rate Scheduler`

Decays the learning rate as the model approaches optimal minimums to prevent oscillation.

Model Implement

Implement

Implementation: Keras / TensorFlow API

```
model = Sequential()
model.add(LSTM(64, input_shape=(X_train.shape[1],
    X_train.shape[2]), return_sequences=True}))
model.add(LSTM(32, return_sequences=False))
model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

early_stop = EarlyStopping(patience=10,
      restore_best_weights=True})

history = model.fit(X_train,  y_train, epochs=100,
      batch_size=32, validation_split=0.2,
      shuffle=False}, callbacks=[early_stop])
```

Passes full temporal sequence to next LSTM layer.

Reverts model to optimal epoch minimum rather than final overfitted state.

Maintains strict chronological integrity.

Model Implement

Post-processing & eval

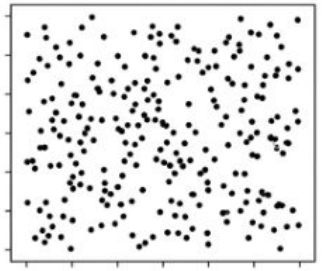
Inverse Scaling & Final Evaluation

Inverse Transform

[0.45] ⟶ [4,500 Passengers]

Neural networks output normalized fractions [0, 1]. Applying `inverse_transform` is mandatory to convert outputs back to business metrics for stakeholder reporting.

Residual Analysis



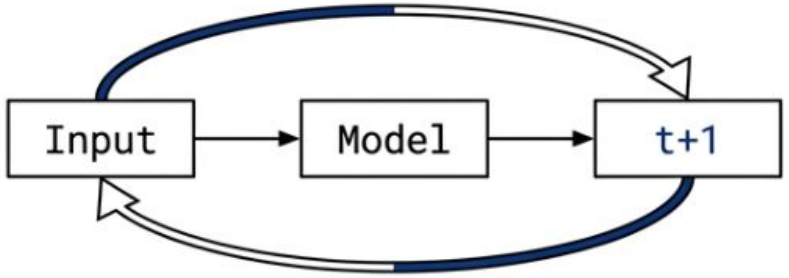
Evaluating the remaining error (Predictions vs. Actuals). If the model has learned perfectly, the residual error must be **White Noise**—completely random, with no remaining patterns or autocorrelation.

Model Implement

Multi-step & one-step

Two primary architectures for forecasting multiple time steps into the future.

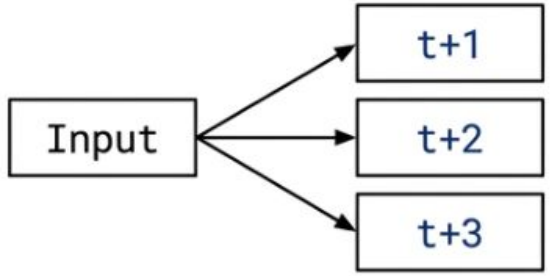
Recursive (Iterative) Strategy



Mechanism
Predict t+1, then feed that prediction back into the model to predict t+2.

Pros/Cons
Simple architecture Dense(1), but highly susceptible to rapid error accumulation.

Direct Strategy



Mechanism
Modify output layer to Dense(n) to predict n future steps simultaneously.

Pros/Cons
Far more stable and prevents compounding errors, but requires significantly larger datasets to train.

Content

- Deep Learning Architectures for Time Series
- Data preprocessing and tensor structure
- Model Implementation
- **Advanced libraries**

Advanced libraries

Time-series libraries

The specialized library ecosystem offers distinct tools for distinct architectural needs

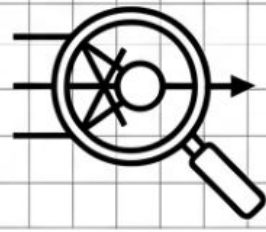
Prototyping

Darts

Simplicity & architectural variety

Interpretability

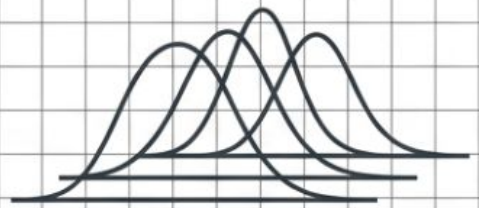
NeuralProphet



Explainable DL via PyTorch

Scale & Probability

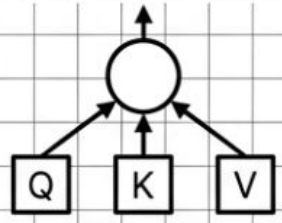
GluonTS



Massive scale probabilistic forecasting

Transformer SOTA

PyTorch Forecasting



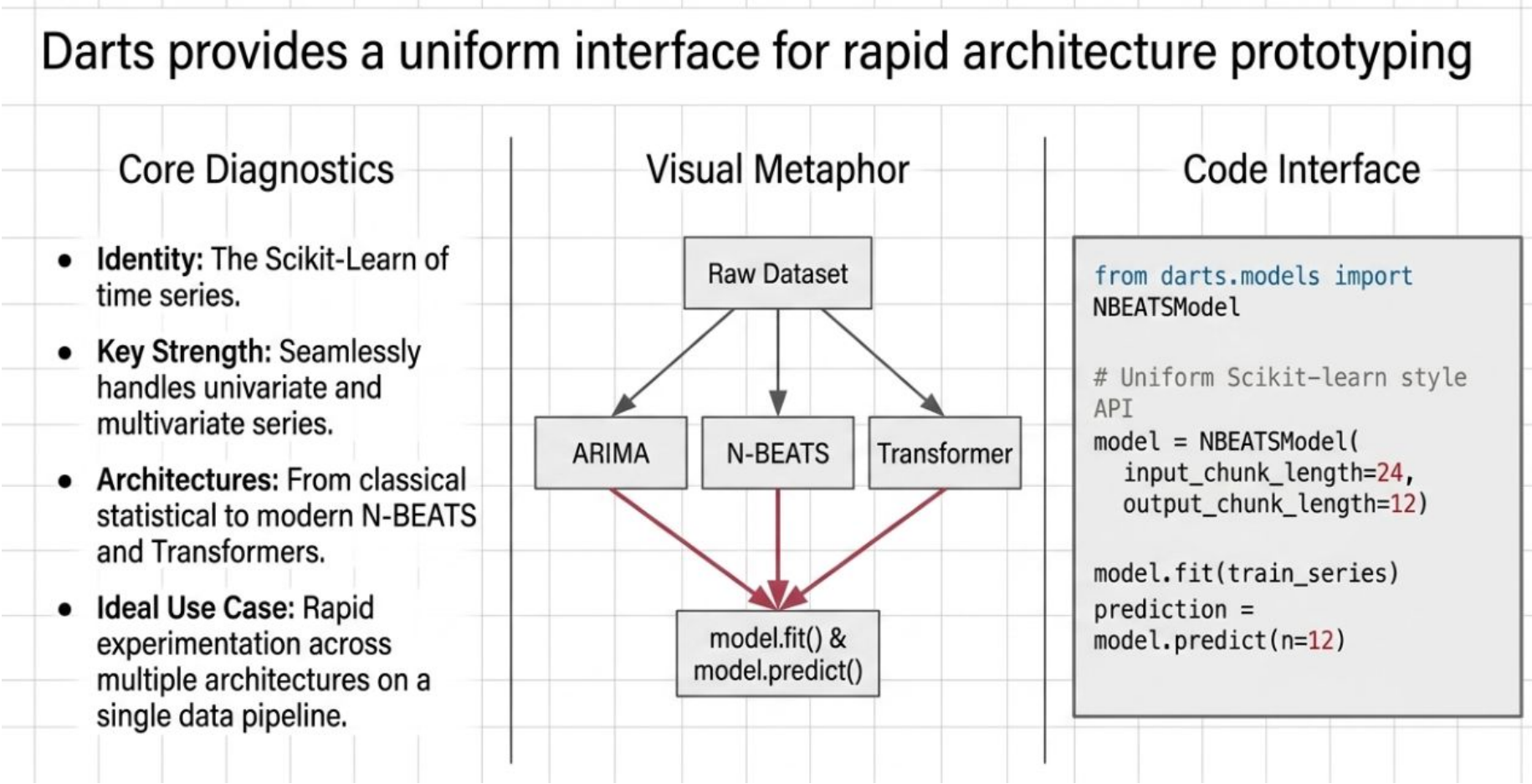
Temporal Fusion Transformers & exogenous variables

Advanced libraries

Darts

Darts

Explore the library



Advanced libraries

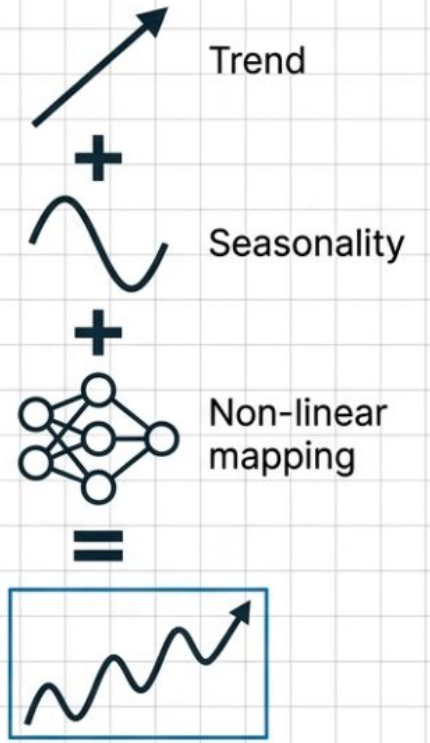
NeuralProphet

Explore the library

NeuralProphet bridges deep learning performance with high interpretability

Core Diagnostics

- **Identity:** PyTorch-based successor to Facebook Prophet.
- **Key Strength:** Drastically higher interpretability than traditional black-box neural networks.
- **Ideal Use Case:** Datasets exhbiting strong cyclical patterns and overarching trends.




```
from neuralprophet import
NeuralProphet

# Combines classical components
with Auto-Regressive NNs
m = NeuralProphet(n_lags=10,
epochs=50)

metrics = m.fit(df, freq="D")
forecast = m.predict(df)
```

Advanced libraries

GluonTS

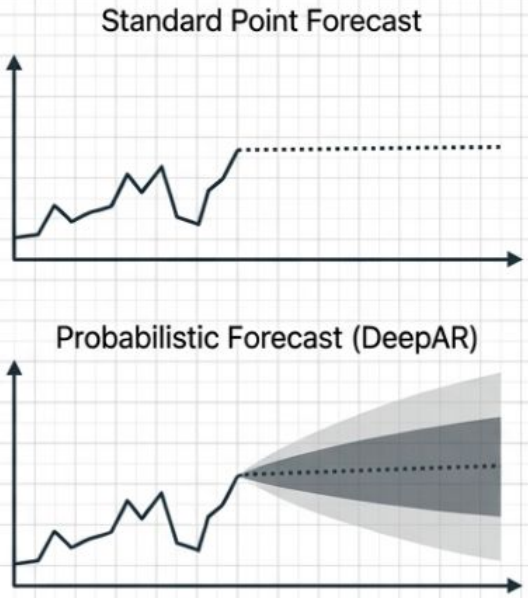
Explore the library

GluonTS specializes in quantifying **uncertainty** across massive datasets

Core Diagnostics

- **Identity:** Amazon-developed framework for large-scale operations.
- **Core Architecture:** DeepAR (RNN-based probabilistic forecasting).
- **Key Strength:** Optimized for concurrent training on millions of distinct time series.
- **Ideal Use Case:** Supply chain and e-commerce applications where risk and margin of error must be strictly quantified.

Visual Metaphor



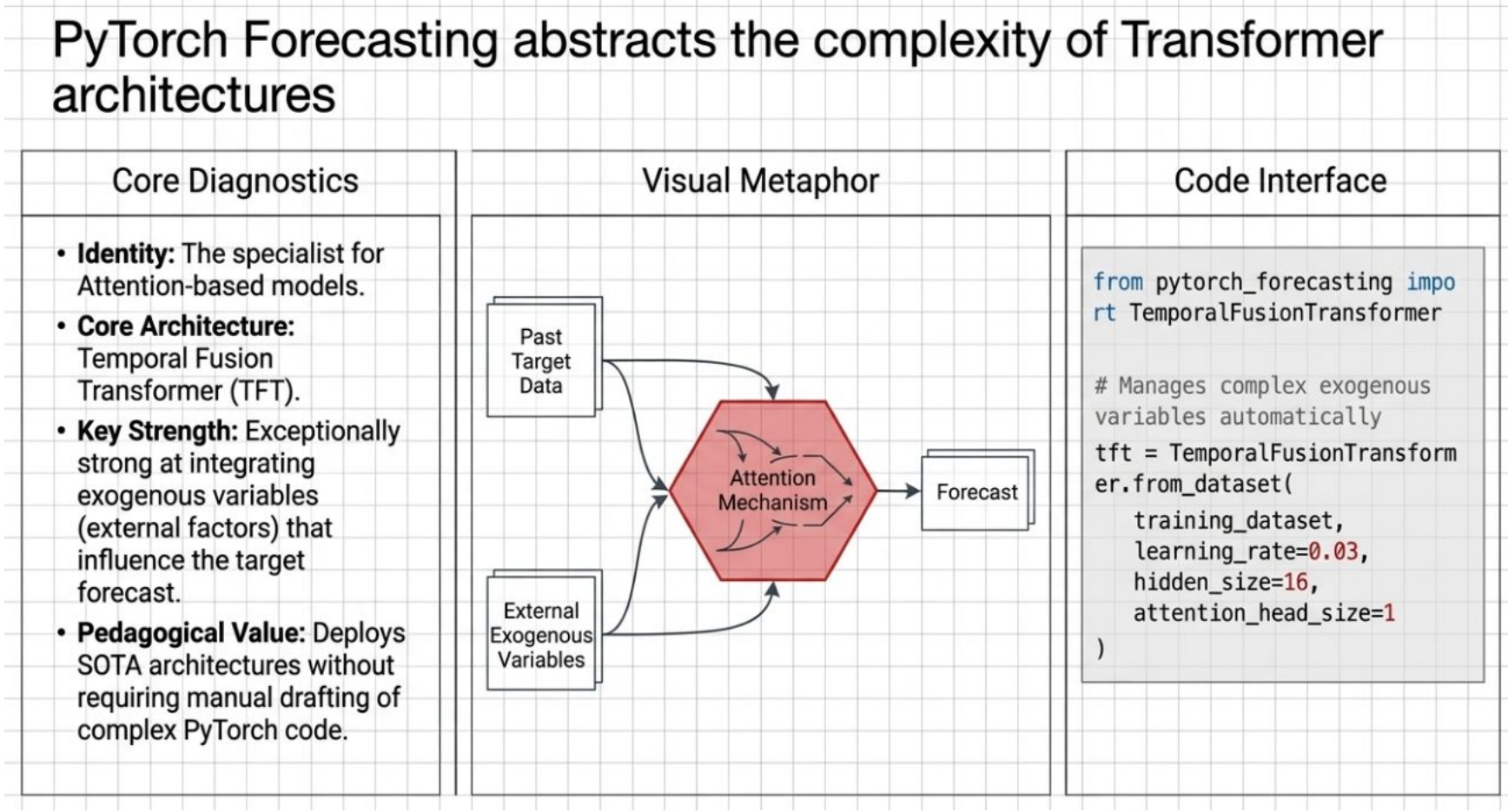
Code Interface

```
from gluonts.torch.model.deepar
import DeepAREstimator

# Focuses on probabilistic
distribution outputs
estimator = DeepAREstimator
(prediction_length=24,
freq="H")

predictor =
estimator.train(training_data)
```

Advanced libraries

PyTorch Forecasting

Explore the library



Thank you