LabIA
UFBA

IEEE WCCI 2026 · TUTORIAL

FROM STOPS TO GRAPHS

Hands-On Spatio-Temporal Learning for Public Transportation

A practical tour of SUNT (Salvador Urban Network Transportation dataset)

Ricardo A. Rios, Tatiane N. Rios, Felipe Fernandes, Marcos Ferreira
ricardoar@ufba.br · Laboratory of Artificial Intelligence (LabIA), UFBA

TUTORIAL ROADMAP

Five Steps, One Dataset

A full pipeline on a single real-world dataset — from raw boardings in Salvador to spatio-temporal forecasting models you can run yourself.

01

The SUNT Dataset

What it is, where it comes from, and how it compares.

02

Loading SUNT

The `suntdataset` package and its data loaders.

03

Exploring the Data

Network views and time-series views.

04

Forecasting Demand

Statistical, deep, Transformer & foundation models.

05

Putting It Together

Choosing the right model for the right question.

PART 01

The SUNT Dataset

What the data is, where it comes from, and how it compares to the benchmarks the field already knows.

What Is SUNT?

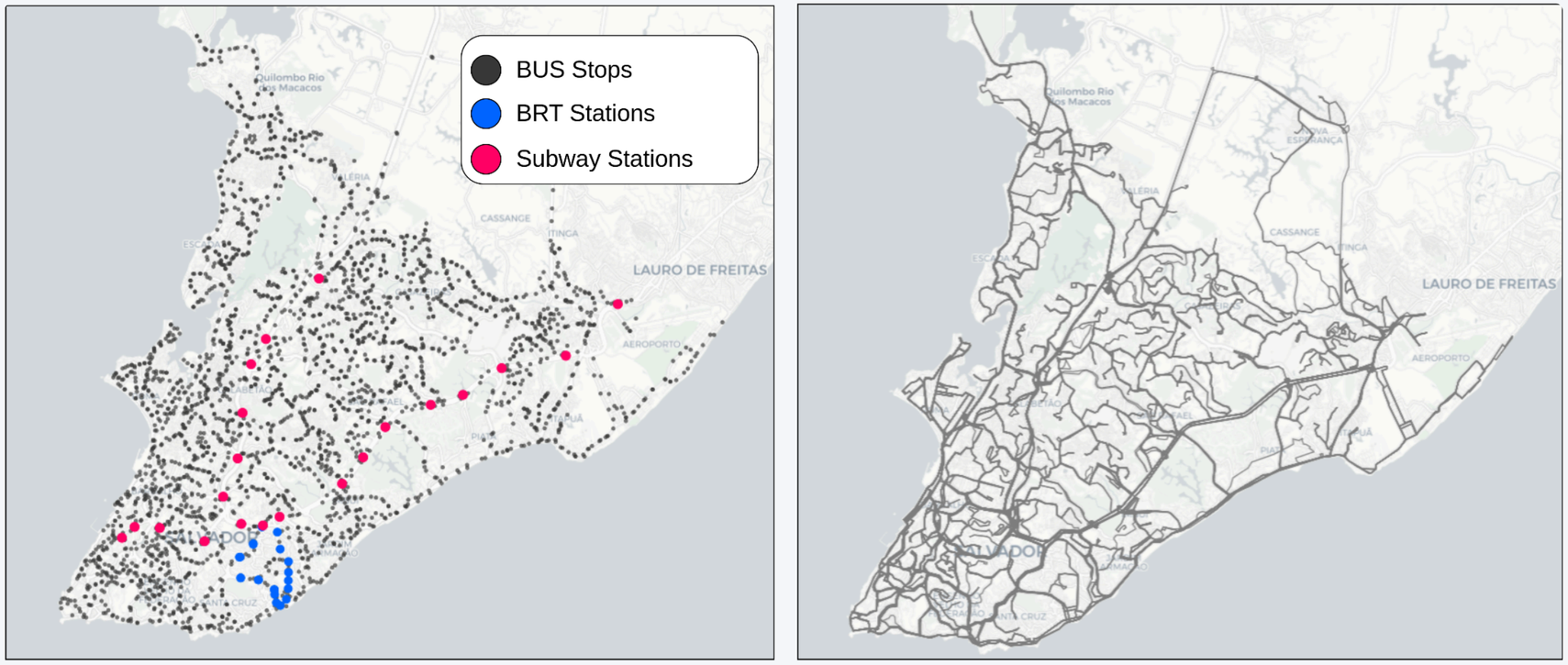
SUNT — **Salvador Urban Network Transportation** — is a landmark spatio-temporal dataset describing one full year of public transit in Salvador, Brazil: one of Latin America's largest bus-and-BRT systems.

- Built by fusing **four operational data sources** — fare collection, vehicle GPS, trip logs and schedules — into a single, analysis-ready dataset.
- Captures demand at **sub-minute resolution** for thousands of stops over an entire year.
- Designed for **both views of mobility**: the network in space and demand over time.

IN ONE SENTENCE

A real, city-scale benchmark for **spatio-temporal learning** — where every stop is a node, every trip a flow, and every minute a new observation.

Where Our Data Come From



a) ~3,000 stops and stations across the Salvador metropolitan area — bus stops, BRT stations and subway stations. **b)** The underlying street network those vehicles travel on. Demand is observed everywhere people board and alight.

Four Data Sources, One Dataset

SUNT integrates four independent operational feeds so that **space, time and demand** line up on a common index.

AFC

Fare Collection

Automated **boarding events** from smart-card validations — who boards, where and when.

AVL

Vehicle Tracking

Timestamped **GPS positions** of every vehicle — the trajectory each trip actually followed.

LTI

Trip Logs

Operational **trip start / end** records that delimit and identify individual trips.

GTFS

Schedules & Topology

Stops, routes, shapes and timetables — the **static structure** of the network.

Fusing AFC + AVL + LTI + GTFS yields integrated **boarding**, **alighting** and **origin–destination** tables, all aligned to stops and trips.

Dataset at a Glance

~700K
PASSENGERS PER DAY

~2,000
VEHICLES · ELECTRIC & DIESEL

~400
LINES

~3,000
STOPS & STATIONS

Sub-minute
TEMPORAL RESOLUTION

Mar 2024 → Mar 2025
ONE FULL YEAR OF COVERAGE

SUNT vs. Classical Datasets

Dataset	#Nodes	#Edges	Period	Shortest interval
METR-LA	207	2,369	Mar – Jun 2012	5 minutes
PeMS-BAY	325	1,515	Jan – May 2017	5 minutes
TaxiBJ	NA	NA	2013 – 2016	30 minutes
BikeNYC	50	NA	Apr – Sep 2014	1 hour
Shanghai Metro	288	958	Jul – Sep 2016	15 minutes
Hangzhou Metro	80	248	Jan 2019	15 minutes
Chongqing Metro	170	NA	Mar 2019	15 minutes
UVDS	104	NA	Three months	5 minutes
SUNT	**2,871**	**4,526**	**Mar 2024 – Mar 2025**	**< 1 minute**

An order of magnitude larger in nodes and edges than common benchmarks — and the only one spanning a **full year** at **sub-minute** resolution.

Where the Dataset Is Published

Scientific Data · **Nature** OPEN ACCESS

Salvador Urban Network Transportation (SUNT): A Landmark Spatiotemporal Dataset for Public Transportation

M. V. Ferreira, M. Souza, T. N. Rios, I. F. C. Fernandes, J. Nery, J. Gama, A. Bifet & R. A. Rios

Scientific Data **12**, Article 1320 (2025) · Published 30 July 2025

DATA DESCRIPTOR

A peer-reviewed **data descriptor** documents the full collection, fusion and validation pipeline — so the dataset is citable, reproducible and openly available.

nature.com/articles/s41597-025-05674-6

Get the Data

SUNT is mirrored across three open platforms — pick whichever fits your workflow.

Mendeley Data

Versioned archive of the raw AFC, AVL and LTI feeds with a citable DOI.

GitHub

The `suntdataset` package — loaders, examples and notebooks.

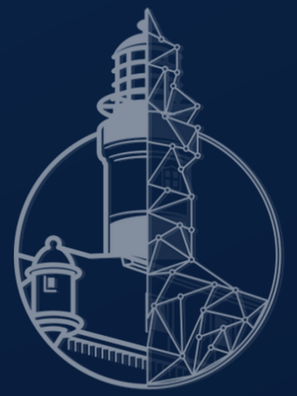
Hugging Face

Streaming-ready tables queried on demand by date, with no full download.

PART 02

Loading SUNT

One small package turns a year of raw transit feeds into tidy tables, time series and graphs — in a few lines of Python.



The suntdataset Package

A thin, well-documented wrapper around the published data. It downloads only what you ask for and returns familiar objects.

- **SUNTLoader** — pulls any table or time series, filtered by date, frequency and day type.
- **SUNTVisualizer** — builds graphs and figures straight from the loaded data.
- Returns **pandas DataFrames** and **networkx graphs** — no custom formats to learn.

INSTALL

```
pip install suntdataset
```

Backed by **Hugging Face** streaming, so you query by date instead of downloading a year of data up front.

pypi.org

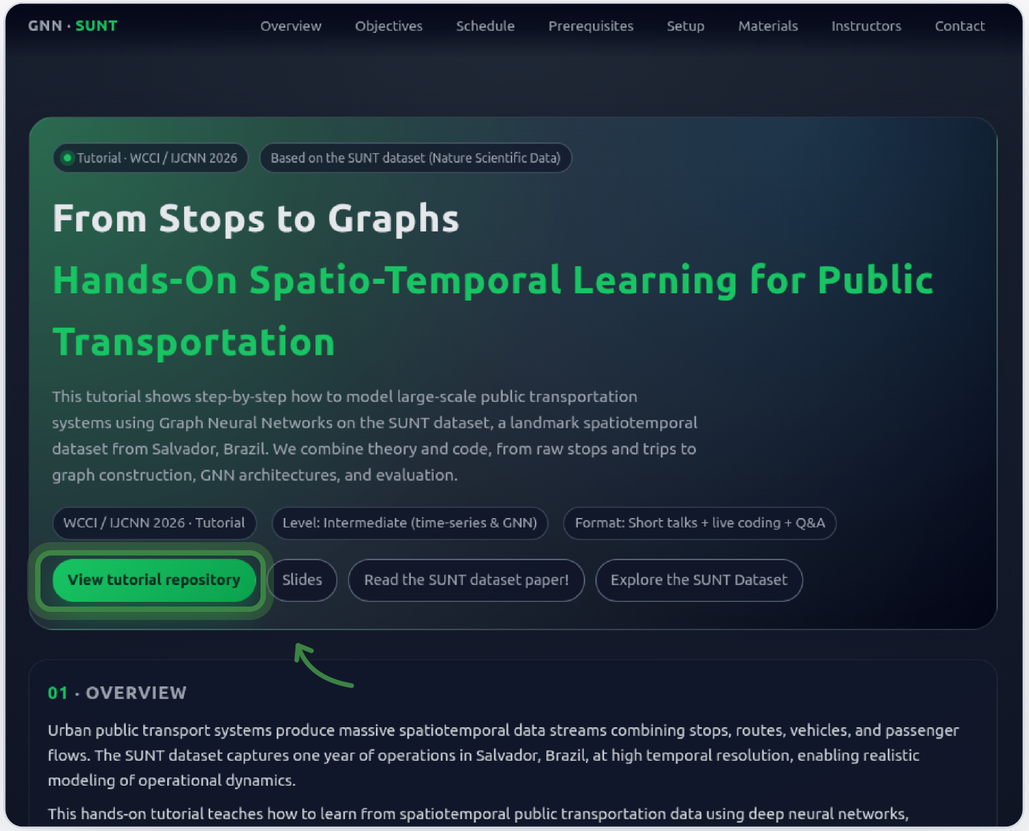
BEFORE WE BEGIN

Code along with us.

Everything in this tutorial — the data loaders and the model code — lives in one place. Open it now and follow along hands-on.

labia-ufba.github.io/
wcci-ijcnn-sunt-gnn

Start at the Tutorial Site



- Click the green **“View tutorial repository”** button to get the code.
- **Slides**, the **SUNT dataset paper** and the **dataset itself** are one click away.
- Everything we run today lives there — clone it and code along.

OPEN IT NOW

`labia-ufba.github.io/wcci-ijcnn-sunt-gnn`

Basic Code

```python
# 1 · import the SUNT data-loader helpers
import utils.data_loader as dl

# 2 · load one month of boardings — weekdays only
df = dl.load_boarding(
    start_date="2024-04-01",
    n_days=30,
    day_type="workdays",   # all | workdays | saturdays | sundays
    source="pip",           # pip (package) | hgface (Hugging Face)
)

# 3 · aggregate raw events → (time × stop) matrix
ts = dl.create_stop_timeseries(df, freq="15min", top_n=50)
```

Every loader takes the same arguments — a **start date**, a number of **days**, a **day type** and a **source** — and returns a ready-to-use `pandas` DataFrame.

Alighting — `load_alighting()`

```python
# where passengers leave the system
df = dl.load_alighting(
    start_date="2024-04-01",
    n_days=30,
    day_type="all",
    source="pip",
)
```

RETURNED COLUMNS

`tripuserid` `stop_time_ali` `stop_id_ali` `walk_dis` `trip_dis` `target_alighting` `date_ref`

Estimated **unboarding** events — when and where each trip ends, plus the walking and in-vehicle distances used to infer them.

Origin–Destination — `load_od()`

```python
# per-stop loading along each trip
df = dl.load_od(
    start_date="2024-04-01",
    n_days=7,
    day_type="all",
    source="pip",
)
# feeds build_graph_from_od(df)
```

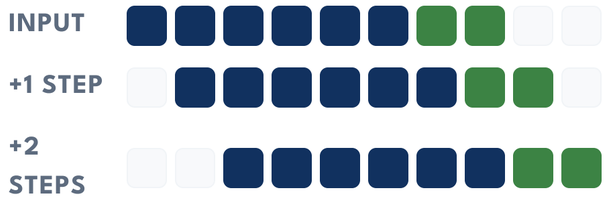
RETURNED COLUMNS

`stop_id` `trip_id` `pt_sequence` `stop_time` `n_boardings`
`n_alighting` `loading` `balance` `route_short_name`
`direction_id` `vehicle` `date_ref`

Tracks **loading** and **balance** at every stop along a trip — the table the network graph is built from.

From Series to Sequences

```
# slide a window over the series → (X, y) pairs
data = dl.prepare_sequences(
    ts,
    input_len=96,      # past steps in
    horizon=12,        # future steps out
    train_ratio=0.70,
    val_ratio=0.15,
)
# X_train (samples, 96, N)
# y_train (samples, 12, N)
```



■ input window (**input_len**) ■ forecast horizon (**horizon**)

The window slides one step at a time, producing thousands of training pairs. Values are **MinMax-scaled** and split **70 / 15 / 15** into train / val / test.

Dataset Types via `load_batch()`

dataset_type	What it returns	Source
`boarding`	**AFC + LTI + AVL + GTFS integrated boarding events**	**pip · hgface**
`alighting`	**Estimated alighting (unboarding) events**	**pip · hgface**
`od`	**Origin–Destination — loading and balancing per stop / trip**	**pip · hgface**
`gtfs-stops` · `gtfs-trips`	Bus-stop metadata · trip definitions	pip
`gtfs-stop-times` · `gtfs-routes`	Stop-time sequences per trip · route / line metadata	pip
`gtfs-shapes` · `gtfs-agency`	Geographic shapes · agency info	pip
`afc` · `lti`	Raw fare-collection events · trip start / end info	Mendeley
`avl-lines` · `avl-vehicles`	Static route/stop sequences · timestamped GPS positions	Mendeley

Highlighted rows are the **fused, analysis-ready** tables. The **source** column shows where each is served from — the `pip` package, **Hugging Face** streaming, or the **Mendeley** archive.

The Data Loader at a Glance

Raw data loaders	`load_boarding(start_date, n_days, day_type, source)`	Boarding table (AFC + LTI + AVL + GTFS), filtered by period & day type.
	`load_alighting(…)`	Estimated alighting (unboarding) table.
	`load_od(…)`	Origin–Destination table — boardings, loading and balance by stop / trip.
	`load_gtfs(table)`	Any GTFS table — stops, trips, routes, shapes, etc.
Time Series	`load_timeseries(…)`	**Memory-efficient** — aggregates one day at a time, so peak memory is **O(1 day)** instead of O(n days). Returns a **time × stop** matrix.
	`create_stop_timeseries(df, freq, top_n)`	Same aggregation from a pre-loaded DataFrame (`groupby` + `unstack`).
	`prepare_sequences(ts, input_len, horizon)`	Sliding windows (X, y) — MinMax scaling and train / val / test splits.
Graph	`build_graph_from_od(od)`	Builds an **nx.DiGraph** from the OD table — nodes are stops, edges are passengers between consecutive stops in a trip.
Internal	`get_available_dates(type)`	`@lru_cache` — queries the Hugging Face API for available dates before downloading, so missing days are skipped instead of throwing 404s.

PART 03

Exploring the Data

Before any model, look at the data two ways — as a network in space, and as demand unfolding over time.

Why Visualize SUNT Data?

Two complementary views answer different questions about how the system works — **where** demand concentrates and **when** it happens.

NETWORK VIEW · SPACE

How stops connect

See how passengers flow across the system. Edges represent passengers travelling between consecutive stops on the same trip.

Reveals key corridors, transfer points and highly connected areas.

TIME SERIES VIEW · TIME

How demand evolves

See how boardings change over time at a stop, a line, or the whole system — from minutes to seasons.

Reveals peaks, daily and weekly patterns, and day-type effects.

Together they give the **where** and **when** of demand — the foundation for everything that follows.

From Trips to Graphs

The idea

Each trip visits a sequence of stops. If we draw an arrow between every pair of **consecutive** stops and weight it by how many passengers ride that segment, the whole city becomes one directed graph.

- **Nodes** are stops and stations.
- **Edges** are passenger flows between consecutive stops.
- Built directly from the OD table by `build_graph_from_od()`.

FORMAL DEFINITION

$$G = (V, E)$$

$$V = \{ s_1, s_2, \ldots, s_N \} \quad N \approx 2{,}871 \text{ stops}$$

$$(s_i, s_j) \in E \quad \text{if some trip goes } s_i \to s_j$$

$$w_{ij} = \Sigma \text{ passengers on } s_i \to s_j$$

NETWORK VIEW

Building the Network Graph

```python
import networkx as nx
from utils import load_od, build_graph_from_od, load_timeseries

# 1 · rank stops by mean hourly occupancy
ts = load_timeseries(freq="1h", top_n=200)
top_stops = ts.mean().sort_values(ascending=False).head(20).index

# 2 · build the directed graph from OD flows
od = load_od()
G  = build_graph_from_od(od)   # nodes = stops · edges = flows

# 3 · subgraph of the busiest stops  → figure A
G_top = G.subgraph(top_stops)    # node size ∝ boardings/h

# 4 · most central hubs  → figure B
centrality = nx.degree_centrality(G)
hubs = sorted(centrality, key=centrality.get, reverse=True)[:20]
```

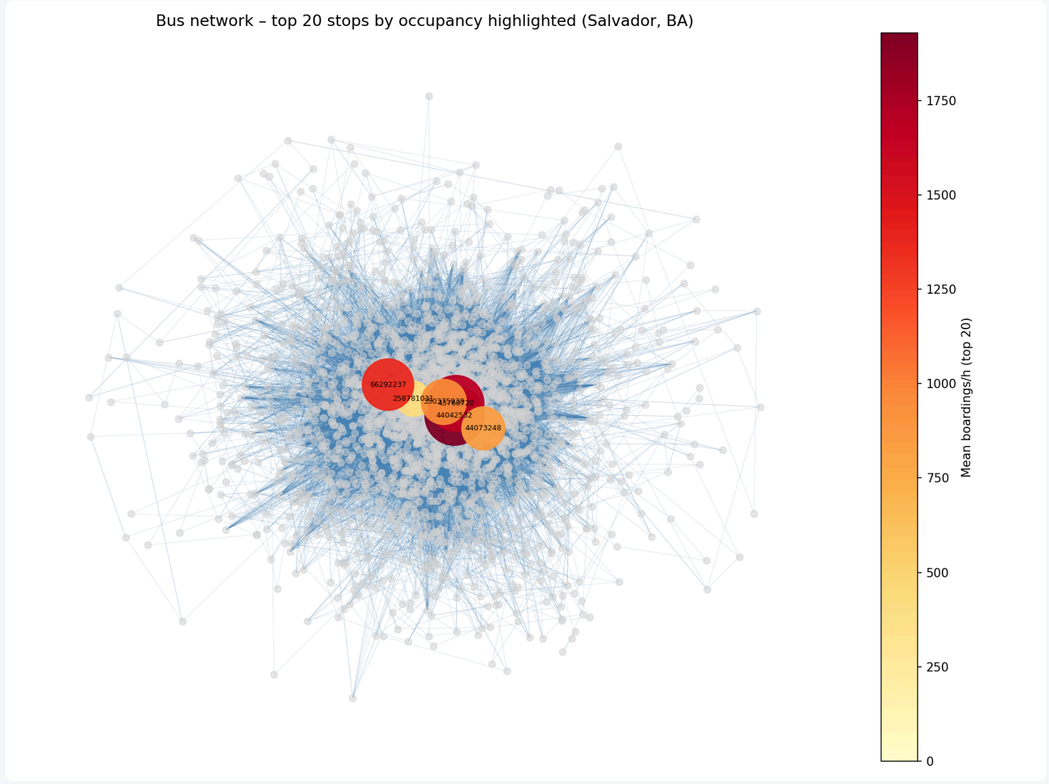
- **load_timeseries()** ranks stops by mean boardings/hour — the busiest 20 become the focus.
- **build_graph_from_od()** turns the OD table into a directed graph: stops are nodes, passenger flows are weighted edges.
- The **subgraph** drives the next slide's figure; **degree centrality** drives the one after.

BONUS

The same graph exports to an interactive **Folium** map — every hub plotted on Salvador with occupancy-scaled markers.

NETWORK VIEW

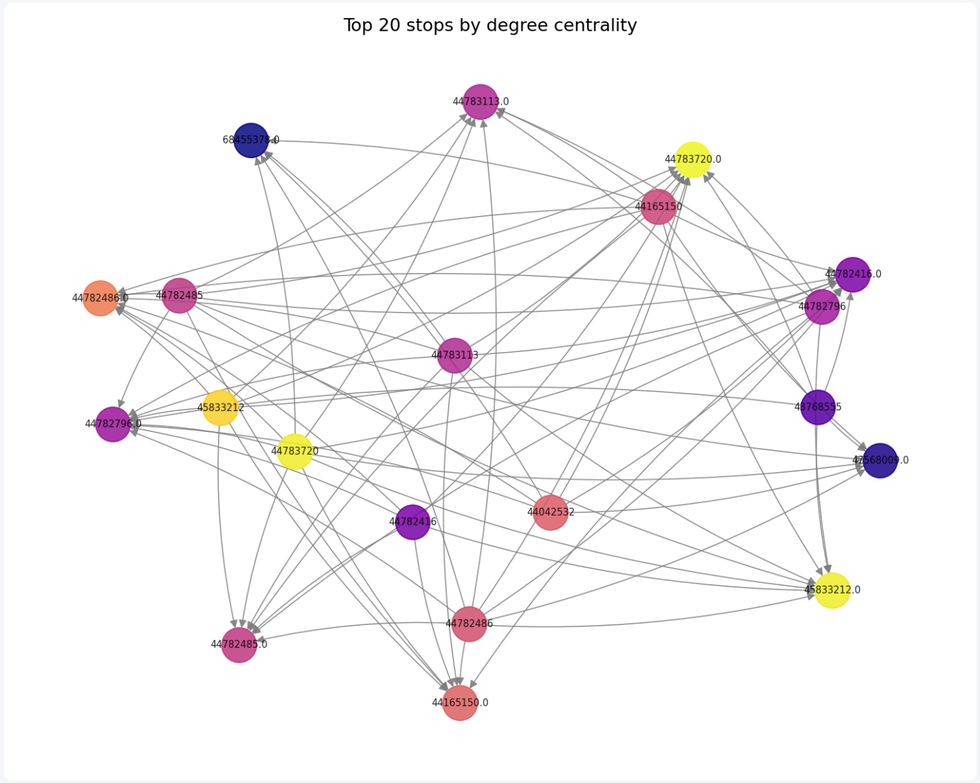
The Salvador Bus Network

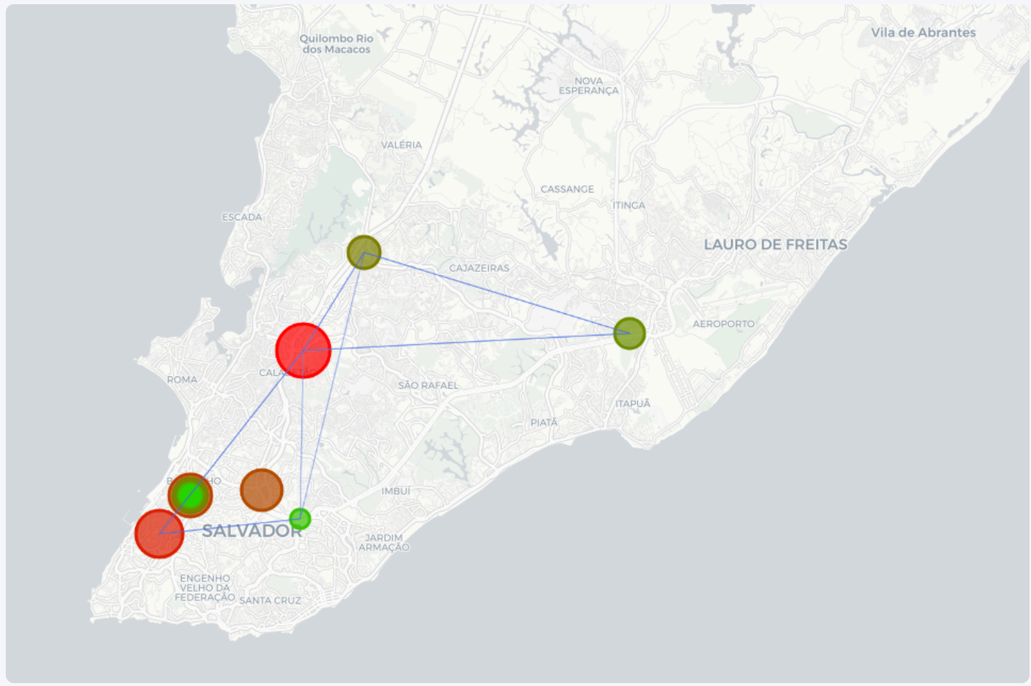


The full bus network drawn as a graph. The **top-20 stops by mean boardings/hour** are highlighted and sized by volume — a dense, scale-free structure with a few dominant hubs at the centre.

NETWORK VIEW

Key Corridors & Centrality





Left: the most central stops by degree — the natural transfer hubs of the system. **Right:** those hubs placed back on the map, revealing the corridors that structure mobility across Salvador.

TIME SERIES VIEW

Demand Over Time

The idea

Count how many passengers board each stop in every time interval. One stop becomes a time series; the whole city becomes a stack of them aligned on the same clock.

- Native resolution is **sub-minute**; resample to any `freq` you need.
- Demand is strongly periodic: daily and weekly cycles.
- This is the signal every forecasting model will learn from.

FORMAL DEFINITION

$$x_t^{(s)} = \text{boardings at stop } s \text{ in interval } t$$

$$X = [\, x_t^{(s)} \,] \in \mathbb{R}^{T \times N}$$

T time intervals × **N** stops: a multivariate time series over the whole network. Each column is one stop's demand; each row is one snapshot of the city.

TIME SERIES VIEW

Building the Time-Series Views

```python
from utils import load_timeseries

# aggregate boardings → (time × stop) matrix; rank stops
ts = load_timeseries(freq="1h", top_n=50)
top_stops = ts.sum().nlargest(5).index   # figure A

# hour × day-of-week pivot  → heatmap (figure B)
df = ts[stop].to_frame("boardings")
df["hour"], df["dow"] = df.index.hour, df.index.day_name()
pivot = (df.groupby(["hour", "dow"])["boardings"]
           .mean().unstack("dow"))

# weekly box-plot by day-of-week  → figure C
sns.boxplot(data=df, x="dow", y="boardings")

# raw signal + rolling daily trend  → figure D
roll = ts[stop].rolling(24, center=True).mean()
```

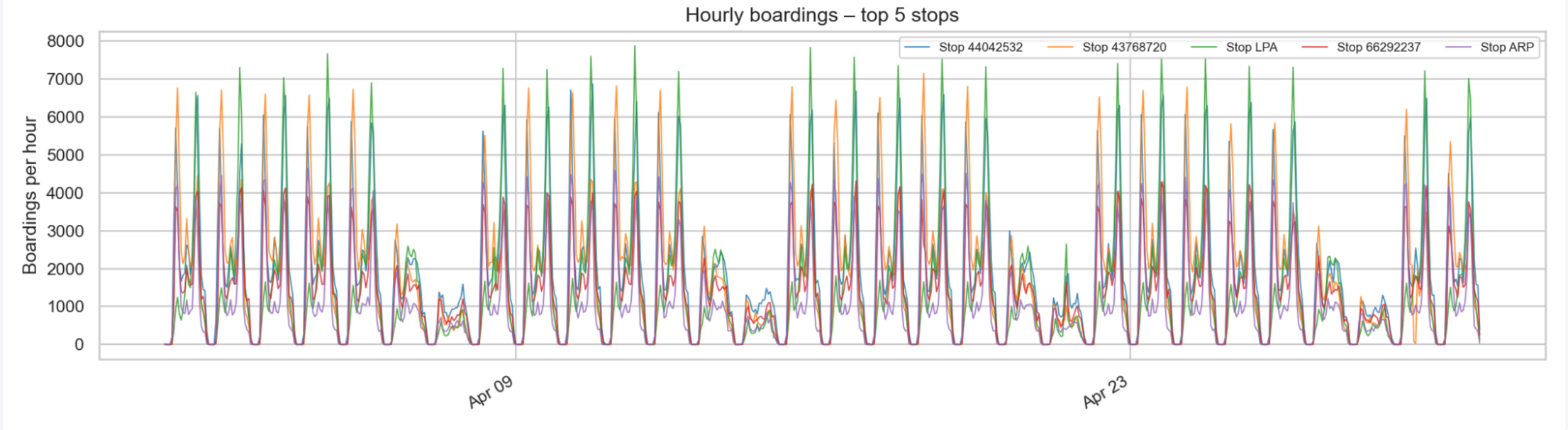
- **load_timeseries()** returns the **time × stop** matrix — `.sum().nlargest(5)` picks the busiest stops.
- The **hour × day-of-week** pivot (`groupby + unstack`) drives the heatmap.
- A **day-of-week box-plot** exposes weekly seasonality.
- `rolling(24, center=True)` overlays the daily trend on the raw signal.

ONE SOURCE, FOUR VIEWS

Every figure that follows comes from the **same** `ts` matrix — only the reshape changes.

TIME SERIES VIEW

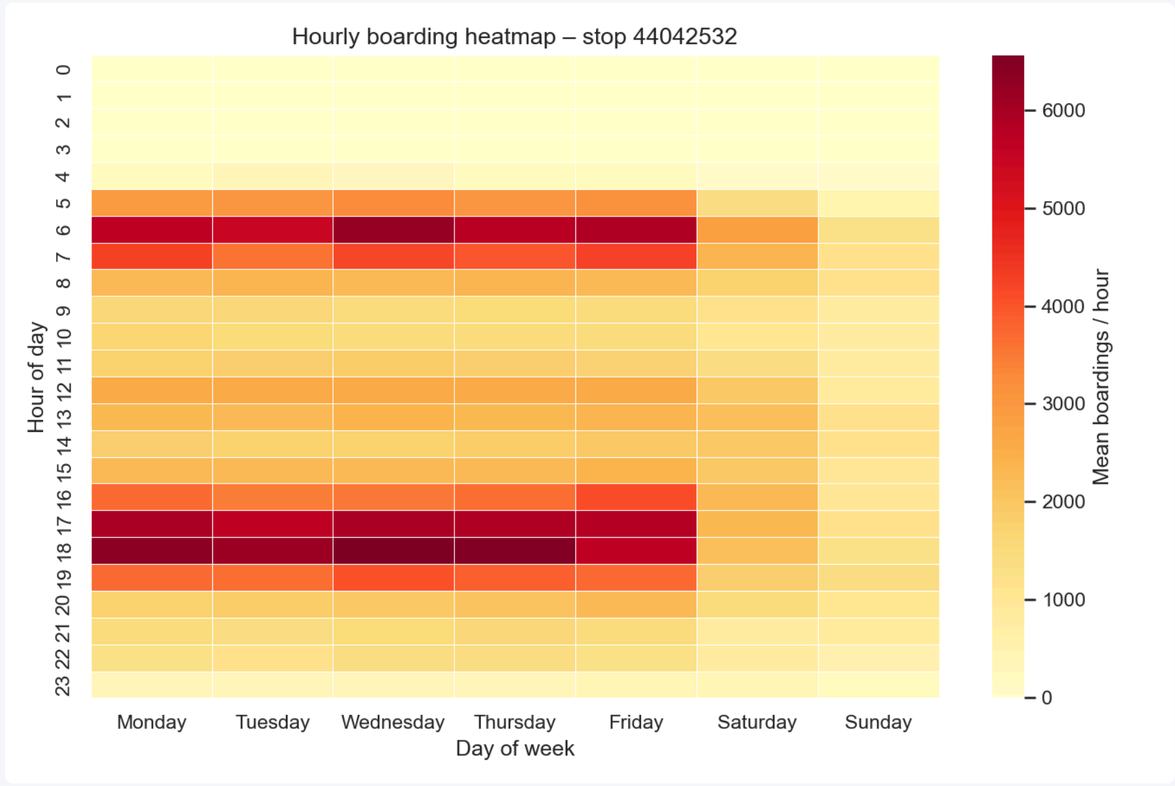
Daily Demand Profile



Hourly boardings at the five busiest stops over April. The signal is sharply periodic: a strong **morning peak**, a softer midday, and an **evening peak** every weekday — with visibly lower weekend demand.

TIME SERIES VIEW

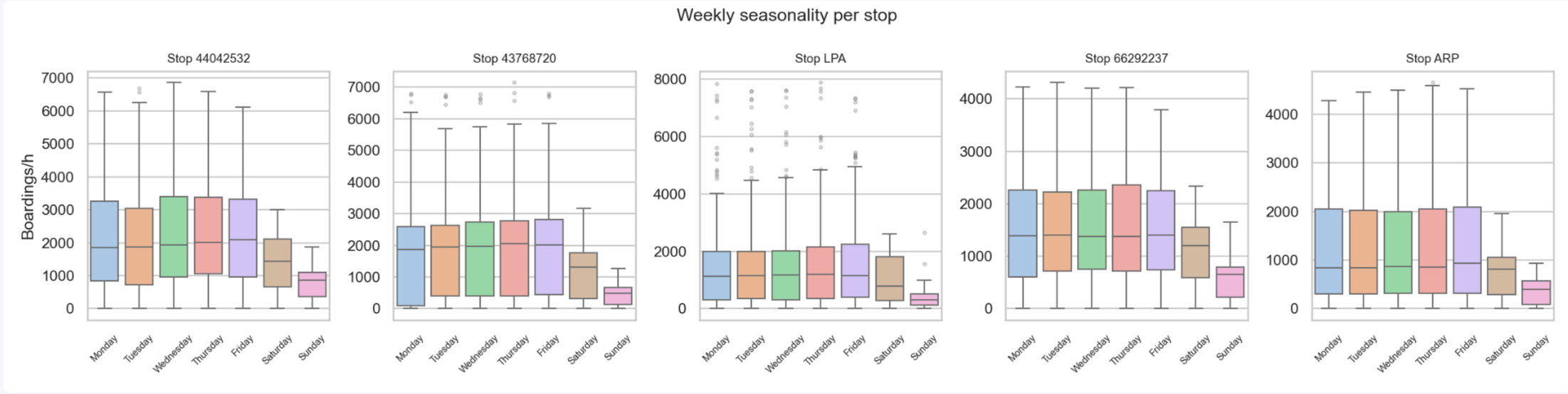
Hourly Boarding Heatmap



Mean boardings per hour for a single stop, by **hour of day** (rows) and **day of week** (columns). The twin weekday bands at ~6–7h and ~17–19h are unmistakable, and they fade clearly on weekends.

TIME SERIES VIEW

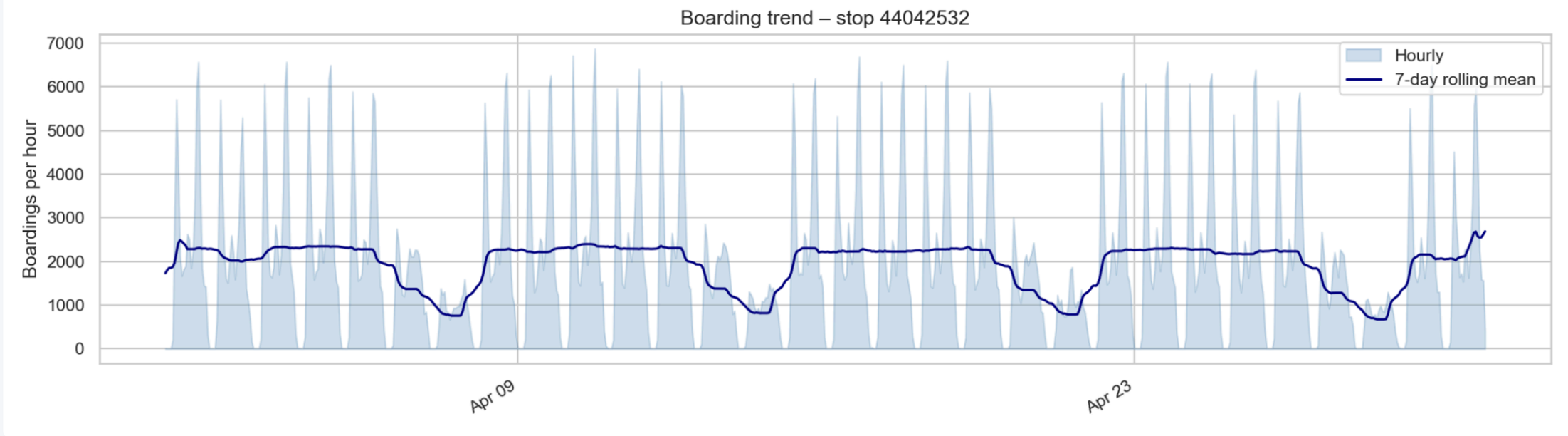
Weekly Seasonality



Distribution of hourly boardings by day of week for five stops. Medians hold steady Monday–Friday and drop on weekends — the **weekly cycle** models must capture, on top of the daily one.

TIME SERIES VIEW

Signal & Rolling Trend

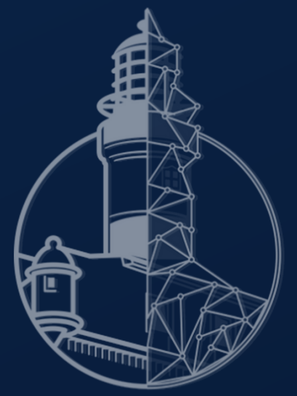


The raw hourly signal for one stop with its **7-day rolling mean**. Beneath the spikes the level is remarkably stable across the month — the periodic component dominates, with only gentle drift in the trend.

PART 04

From Visualization to Prediction

The patterns we just saw (peaks, cycles, corridors) are exactly what forecasting models learn to anticipate.



FROM VISUALIZATION TO PREDICTION

Multiple Modeling Approaches

Five families of models, from interpretable baselines to large pre-trained networks. We walk through each on the same SUNT demand series.

A · Statistical

ARIMA / SARIMA

Captures linear trend and seasonality.

INTERPRETABLE BASELINE

B · Recurrent

RNN · LSTM · GRU

Learn temporal dependencies from sequences.

SEQUENTIAL PATTERNS

C · Attention

Transformers

Capture long-range dependencies and complex dynamics.

LONG-HORIZON FORECASTING

D · Pre-trained

Foundation Models

TimesFM, TEMPO, Chronos — trained across many series.

ZERO / FEW-SHOT

E · Graph

GNN & Spatio-Temporal

Model space and time jointly across stops.

NETWORK-WIDE

The Forecasting Problem

The set-up

Use a window of recent observations to predict the next few. We slide that window across the series to build training examples.

- **Window w** — how much past the model sees.
- **Horizon h** — how far ahead it predicts.
- Scored with **MAE**, **RMSE** and **MAPE** on a held-out test set.
- `prepare_sequences()` handles windows, scaling and splits.

FORMAL DEFINITION

$$\hat{y} = f(\, x_{t-w+1}, \ldots, x_t \,)$$

$$\textit{predict} \quad x_{t+1}, \ldots, x_{t+h}$$

ERROR METRICS

$$MAE = (1/n)\ \Sigma\ |\, x_i - \hat{x}_i \,|$$

$$RMSE = \sqrt{(\ (1/n)\ \Sigma\ (\, x_i - \hat{x}_i \,)^2\ )}$$

$$MAPE = (100/n)\ \Sigma\ |\, (\, x_i - \hat{x}_i \,) / x_i \,|$$

Stationarity & the Random Walk

Why stationarity matters

Classical models assume the series is **stationary** — its mean and variance do not change over time. The random walk (the "drunkard's walk") is the textbook counter-example.

- Each value is the previous one plus a random shock $\boldsymbol{\varepsilon_t}$ (white noise).
- The future depends on a past observation **and** a random term.
- Its mean and variance **grow with t** — so it is **not** stationary.

RANDOM WALK

$$x_t = x_{t-1} + \varepsilon_t$$

MEAN & VARIANCE

$$E(x_t) = t\mu$$

$$Var(x_t) = t\sigma^2$$

Because both mean $t\,\mu$ and variance $t\,\sigma^2$ depend on **t**, the process drifts — the hallmark of non-stationarity. Such series must be differenced before modelling.

Moving Average — MA(q)

The idea

The current value is a weighted combination of the last **q** random shocks — yesterday's noise still echoes today.

- Defined from **q** past white-noise terms.
- **θ** are the model coefficients (roots); **c** is a constant.
- MA(q) is **stationary**: constant mean, time-invariant variance.

FORMAL DEFINITION

$$x_t = c + \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q}$$

ε → white noise · **θ** → MA coefficients · **c** → constant. A finite memory of the last **q** shocks.

Autoregressive — AR(p)

The idea

The current value is a regression on its own **p** previous observations — the recent past predicts the present directly.

- Defined from **p** past observations of the series.
- **φ** are the model coefficients (roots); **c** is a constant.

FORMAL DEFINITION

$$x_t = c + \varphi_1 x_{t-1} + \ldots + \varphi_p x_{t-p} + \varepsilon_t$$

φ → AR coefficients · **c** → constant · **ε** → white noise.

Autocorrelation Function (ACF)

What it measures

How similar the series is to a copy of itself shifted by **k** steps. Strong autocorrelation means the past is predictive of the present.

- Compares the series with its own **lagged** values.
- Peaks expose **seasonality** — e.g. a spike at lag 24 for a daily cycle.
- Guides the **moving-average order q** of ARIMA models.

FORMAL DEFINITION

$$\rho(k) = \gamma(k) / \gamma(0)$$

$$\gamma(k) = Cov(\, x_t, x_{t-k}\,)$$

ρ(k) is the correlation between $\mathbf{x_t}$ and its value **k** steps earlier, normalised to lie in **[−1, 1]**.

Partial Autocorrelation (PACF)

What it measures

The **direct** correlation between the series and its lag-k value, after removing everything explained by the lags in between.

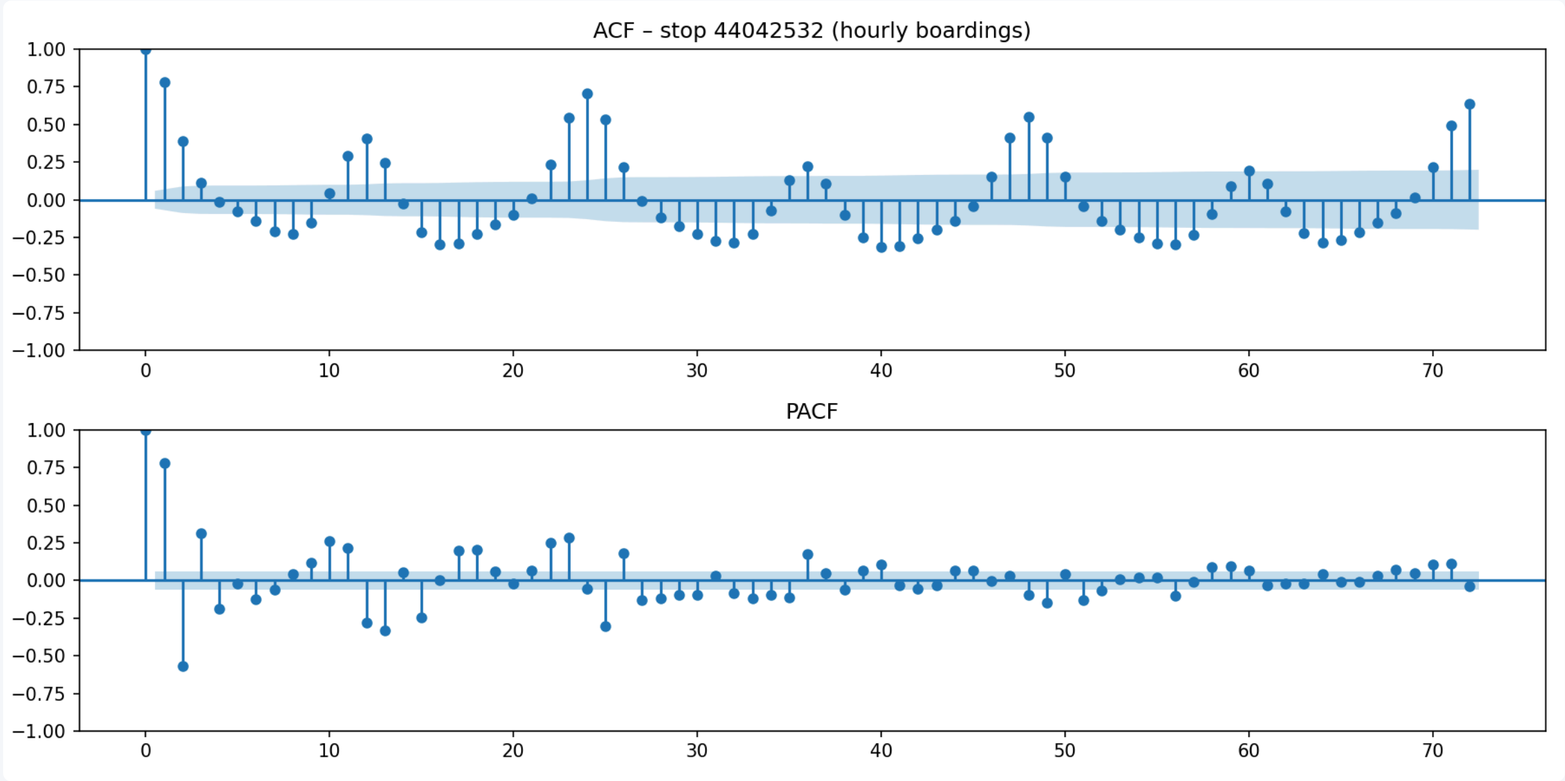
- Strips out the influence of intermediate lags — unlike the ACF.
- Isolates the **genuinely new** information each lag adds.
- Guides the **autoregressive order p** of ARIMA models.

FORMAL DEFINITION

$$\varphi_{kk} = Corr(x_t, x_{t-k} \mid x_{t-1}, \ldots, x_{t-k+1})$$

The correlation at lag **k conditioned** on the intermediate observations — the partial effect of that lag alone.

Reading the Diagnostics



ACF (top) and PACF (bottom) of hourly boardings at one stop. The ACF shows **recurring spikes around multiples of 24h** — clear daily seasonality — while the PACF cuts off quickly, suggesting a **low autoregressive order** plus seasonal terms.

From ARMA to ARIMA

Combine, then integrate

ARMA puts AR and MA together. **ARIMA** adds an integration step that **differences** a non-stationary series until it becomes stationary.

- **ARMA(p, q)** = AR(p) + MA(q).
- **ARIMA(p, d, q)**: difference **d** times, replacing x_t by w_t.
- Differencing removes the non-stationary source of variation.

ARMA(P, Q)

$$x_t = c + \varphi_1 x_{t-1} + \ldots + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots$$

ARIMA(P, D, Q)

$$w_t = x_t - x_{t-1} \quad \text{difference}$$

$$w_t = c + \varphi_1 w_{t-1} + \ldots + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots$$

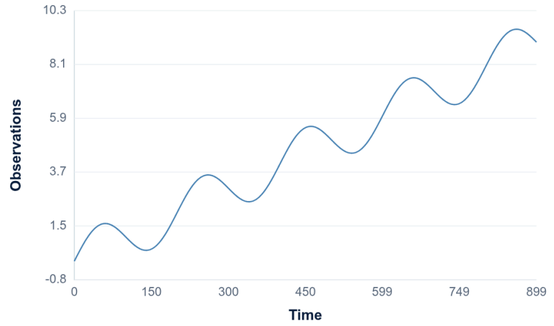
ARMA is applied to the differenced series $\mathbf{w_t}$. **SARIMA** next adds a seasonal copy of the model; **SARIMAX** further allows exogenous variables.

Differencing Removes the Trend

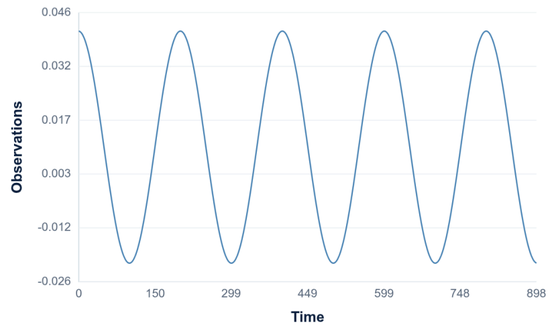
```python
# a signal with trend + seasonality
time = np.arange(0, 9, 0.01)
xt = np.sin(np.pi*time) + time

plt.plot(xt, color="steelblue")

# first difference  ∇x_t = x_t - x_{t-1}
xt_detrend = np.diff(xt)

plt.plot(xt_detrend, color="steelblue")
```

The raw signal **drifts upward** (a linear trend). One **difference** cancels that drift — what remains oscillates around a **constant level**, i.e. it is now stationary.

BEFORE · `xt = sin(πt) + t`



AFTER · `np.diff(xt)` — TREND GONE



Is It Stationary? ADF & KPSS

A trend can be spotted by visual inspection or confirmed objectively by combining two hypothesis tests, ADF and KPSS, which use opposite null hypotheses.

ADF · Augmented Dickey-Fuller

H_0: the series is non-stationary

p < 0.05 → reject H_0 → the series **tends to be stationary**.

p ≥ 0.05 → do not reject → it **may be non-stationary**.

KPSS · Kwiatkowski-Phillips-Schmidt-Shin

H_0: the series is stationary

p < 0.05 → reject H_0 → the series **tends to be non-stationary**.

p ≥ 0.05 → do not reject → it **may be stationary**.

Test	Null hypothesis (H_0)	When p < 0.05
ADF	Series is non-stationary	Evidence of **stationarity**
KPSS	Series is stationary	Evidence of **non-stationarity**

Run both: when ADF says "stationary" **and** KPSS agrees, you can trust it. Disagreement is a hint that the series is only **trend-** or **difference-stationary** — and may need differencing.

Seasonality — the S in SARIMA

A second clock

ARIMA captures short-term dependence between neighbouring steps. But SUNT demand also repeats on a fixed cycle — every **24 h**, every week. **Seasonality** is that regular, period-**s** pattern.

- **Period s** — how many steps before the pattern repeats (s = 24 for a daily cycle on hourly data).
- **Seasonal differencing** removes the cycle, just as ordinary differencing removes trend.
- SARIMA adds AR / MA terms that act at lag **s** — a second model running on the seasonal clock.

SEASONAL DIFFERENCE

$$\nabla_s x_t = x_t - x_{t-s}$$

SEASONAL AR / MA · PERIOD S

$$\Phi_P(B^s) \cdot \Theta_Q(B^s)$$

Compare a value with the one **one season earlier** (x_{t-s}). The seasonal polynomials **Φ**, **Θ** operate on the seasonal lag $\mathbf{B^s}$ — captured by the **(P, D, Q)$_s$** orders.

STATISTICAL · MODEL

SARIMA

The idea

Take **ARIMA** and add a **seasonal** copy of itself. It models trend through differencing and demand cycles through seasonal terms — a transparent, strong baseline.

- **(p, d, q)** — non-seasonal AR, differencing, MA.
- **$(P, D, Q)_s$** — the same, repeated every **s** steps.
- For SUNT, period **s = 24** captures the daily cycle.
- **SARIMAX** extends it further with exogenous variables (X).

FORMAL DEFINITION

$$SARIMA(p, d, q)(P, D, Q)_s$$

$$\Phi_P(B^s)\,\varphi_p(B)\,(1-B)^d(1-B^s)^D\,x_t$$

$$= \Theta_Q(B^s)\,\theta_q(B)\,\varepsilon_t$$

B is the backshift operator ($B\,x_t = x_{t-1}$); φ, θ are AR/MA polynomials and Φ, Θ their seasonal counterparts.

Choosing the Model — AIC & BIC

Fit vs. complexity

Fitting a SARIMAX means comparing many candidate orders $(p,d,q)(P,D,Q)_s$. Two criteria score each one by how well it fits **and** how complex it is — the **lower**, the better.

- **AIC** — Akaike Information Criterion: a relative estimator of quality; a lower value means a better fit to the data.
- **BIC** — Bayesian Information Criterion: penalises adding extra parameters more strongly as the sample grows.
- Choose the orders that **minimise** the criterion — a guard against over-fitting.

AIC · AKAIKE

$$AIC = 2k - 2\ln(\hat{L})$$

BIC · BAYESIAN

$$BIC = k\ln(n) - 2\ln(\hat{L})$$

k → number of parameters · **n** → number of observations · $\hat{L}$ → maximised likelihood. BIC's **k ln(n)** term penalises complexity harder than AIC's **2k**.

SARIMA in Practice

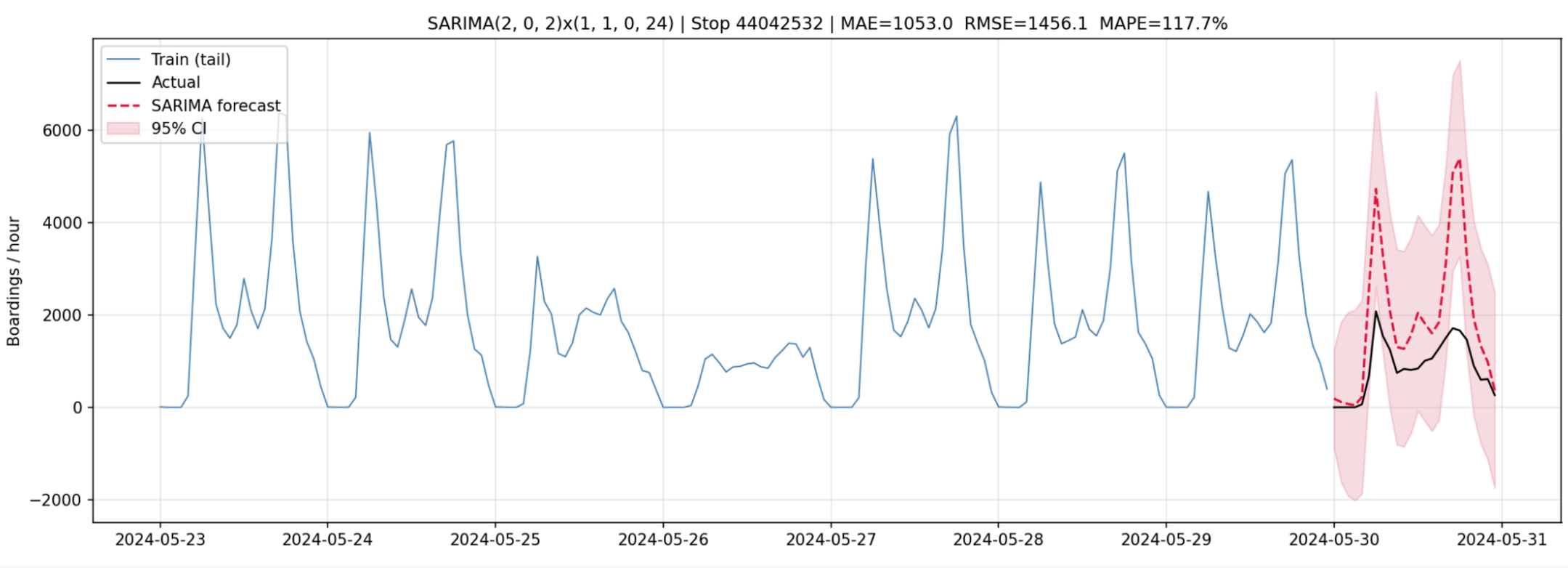
```
# 1 · is the series stationary? (complementary tests)
adfuller(train)   # H0: non-stationary → p < .05 ⇒ stationary
kpss(train)       # H0: stationary     → p > .05 ⇒ stationary

# 2 · inspect ACF / PACF to bound the orders
plot_acf(train, lags=72); plot_pacf(train, lags=72)

# 3 · auto-select orders by minimising AIC
auto = pm.auto_arima(train, seasonal=True, m=24,
                     information_criterion="aic",
                     max_p=2, max_q=2, max_P=1, max_Q=1, D=1)
# → SARIMA(2,0,2)(1,1,0,24)

# 4 · fit, forecast 24h ahead, with 95% interval
fc = SARIMAX(train, order=auto.order,
             seasonal_order=auto.seasonal_order).fit()
out = fc.get_forecast(steps=24)
out.predicted_mean; out.conf_int(alpha=0.05)
```

- **Stationarity** — ADF and KPSS have **opposite null hypotheses**; agreeing on "stationary" is strong evidence.
- **ACF / PACF** bound the search — daily spikes at lag 24 justify **m = 24**.
- **auto_arima** minimises **AIC** over candidate orders — this is the AIC/BIC criterion put to work, and it returns the `(2,0,2)(1,1,0,24)` on the next slide.
- **get_forecast** yields a mean **and** a 95% confidence band.

SAME TEST WINDOW

A **24 h** horizon — identical to the LSTM, GRU, Transformer and Chronos splits — so every model is compared fairly.

SARIMA — Forecast



SARIMA forecast against observed boardings. The model tracks the dominant daily rhythm well and gives an **interpretable baseline**, but smooths over sharp irregular peaks — motivating the learned models that follow.

Recurrent Neural Networks

The idea

Process the series one step at a time, carrying a **hidden state** that summarises everything seen so far. The same weights are reused at every step.

- The hidden state $\mathbf{h_t}$ is the network's memory.
- Naturally handles **variable-length** sequences.
- Plain RNNs forget long-range context — **vanishing gradients** motivate LSTM and GRU.

FORMAL DEFINITION

$$h_t = \sigma(W_x x_t + W_h h_{t-1} + b)$$

$$\hat{y}_t = W_y h_t$$

The state $\mathbf{h_t}$ depends on the current input **and** the previous state — a recurrence that threads information through time.

Long Short-Term Memory (LSTM)

The idea

Add a protected **cell state** and three **gates** that decide what to forget, what to store, and what to output — so the network can hold information for many steps.

- The input gate controls the importance of the new information.
- The **forget gate** controls how much of the past cell state to keep.
- The **candidate memory** $\tilde{C}_t$ is a new vector of potential information built from the current input and the previous hidden state — a "draft of ideas" that may or may not be written into the network's long-term memory.
- If forget → 1 and input → 0, the past dominates and is **carried forward** to future states.

FORMAL DEFINITION

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t]) \text{ forget}$$

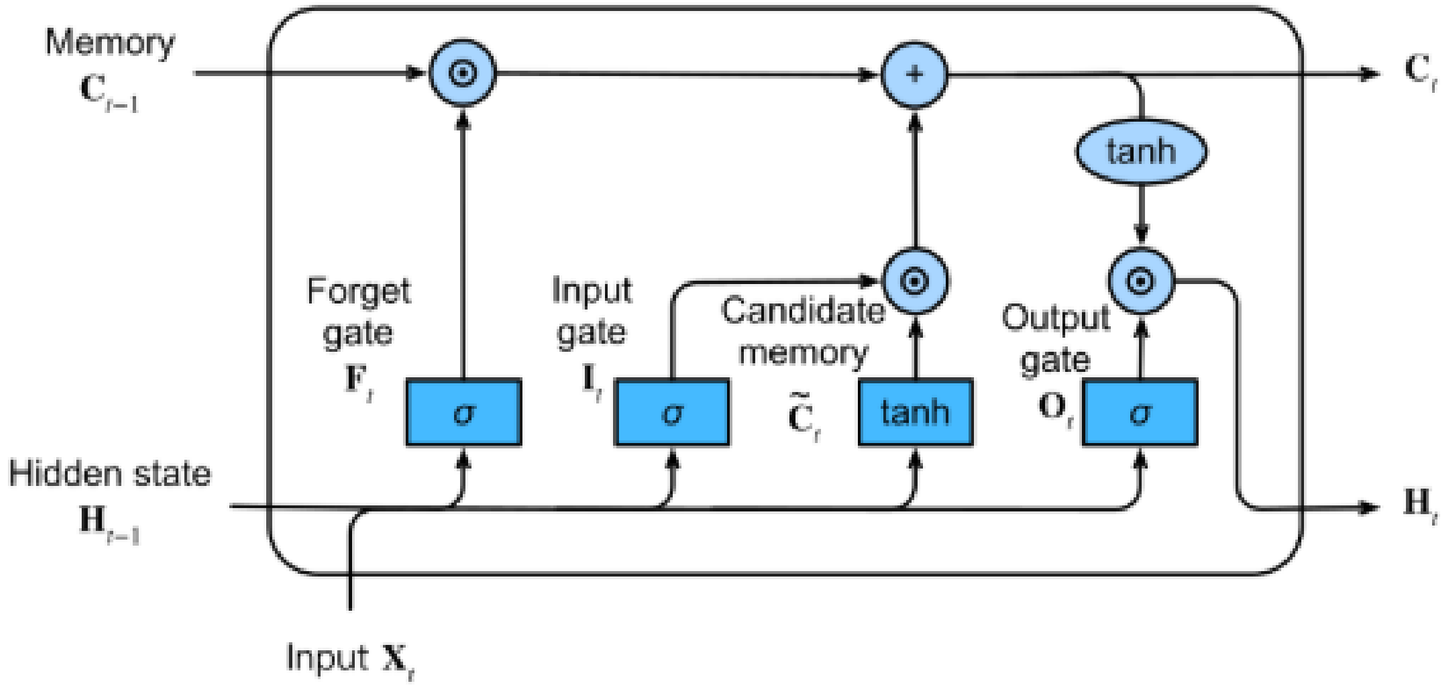
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t]) \text{ input}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t]) \text{ output}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot tanh(W_c \cdot [h_{t-1}, x_t])$$

$$h_t = o_t \odot tanh(c_t)$$

LSTM — Inside the Memory Cell



The cell state $\mathbf{C}_t$ runs straight across the top — a protected memory highway. The **forget gate $\mathbf{F}_t$** erases part of it, the **input gate $\mathbf{I}_t$** writes the **candidate memory $\tilde{\mathbf{C}}_t$** (tanh), and the **output gate $\mathbf{O}_t$** reads it out as the hidden state $\mathbf{H}_t$. Each blue box is a fully-connected layer with a σ or tanh activation.

LSTM in Practice

```python
# multivariate: all 20 stops at once
ts = load_timeseries(freq="1h", top_n=20)
splits = prepare_sequences(ts, input_len=72, horizon=24)

# stacked LSTM → linear head
class LSTMForecaster(nn.Module):
    def __init__(self, N):
        self.lstm = nn.LSTM(N, 128, num_layers=2,
                            batch_first=True, dropout=0.2)
        self.head = nn.Linear(128, N * 24)
    def forward(self, x):           # x: (B, 72, N)
        out, _ = self.lstm(x)
        return self.head(out[:, -1]).view(-1, 24, N)

# train: Adam · MSE · grad-clip · best-val checkpoint
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

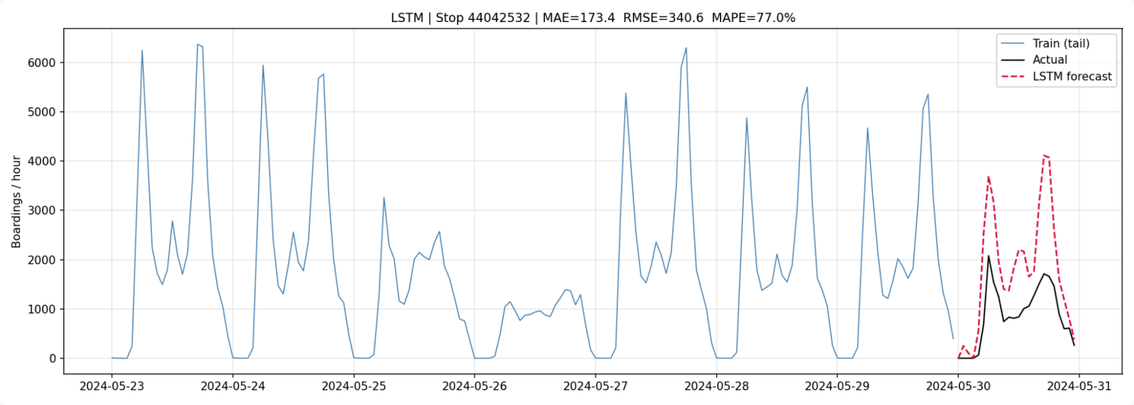
- **Multivariate** — all 20 stops are input features **and** outputs, so the model shares signal across the network.
- **Windows** — `input_len=72` (3 days) → `horizon=24` (1 day), built by **prepare_sequences()**.
- **Architecture** — 2 stacked LSTM layers (hidden 128, dropout 0.2) → a linear head emitting **N × 24** values.
- **Training** — Adam, MSE loss, **gradient clipping**, LR scheduling, and a **best-validation checkpoint**.

SAME WINDOW, FAIR TEST

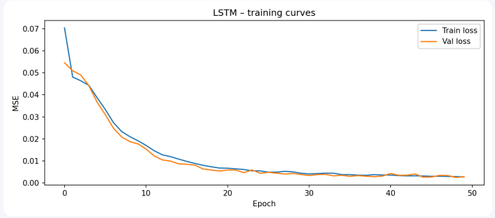
Identical **72 h → 24 h** split feeds the GRU and Transformer too — only the model class changes.

LSTM — Training & Forecast

FORECAST VS. ACTUAL



TRAINING & VALIDATION LOSS



The LSTM follows both the height and the timing of the peaks far more closely than SARIMA; the loss curve converges smoothly without over-fitting.

Gated Recurrent Unit (GRU)

The idea

A lighter cousin of the LSTM: it merges the cell and hidden state and uses just two gates — usually **faster to train** with comparable accuracy.

- **Update** gate balances old state vs. new candidate.
- **Reset** gate decides how much past to ignore.
- Fewer parameters — handy for **many stops** at once.

FORMAL DEFINITION

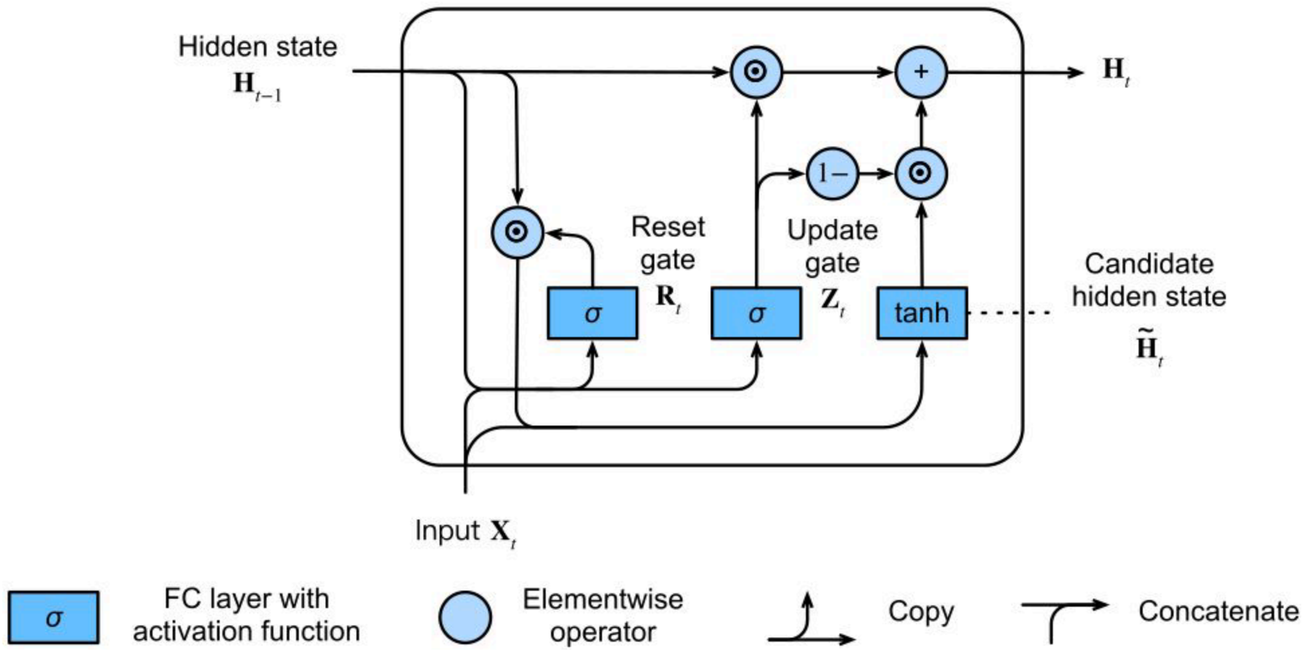
$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \text{ update}$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \text{ reset}$$

$$\tilde{h}_t = tanh(W \cdot [r_t \odot h_{t-1}, x_t])$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

GRU — Inside the Hidden State



No separate memory — the GRU carries only the hidden state $\mathbf{H}_t$. The **reset gate $\mathbf{R}_t$** decides how much past to forget when forming the **candidate $\tilde{\mathbf{H}}_t$** (tanh), and the **update gate $\mathbf{Z}_t$** blends old and new via the **$(1-\mathbf{Z}_t)$ / $\mathbf{Z}_t$** mix — two gates instead of three, fewer parameters.

GRU in Practice

```python
# same 72h → 24h windows as the LSTM
splits = prepare_sequences(ts, input_len=72, horizon=24)

# bidirectional GRU → LayerNorm → MLP head
class GRUForecaster(nn.Module):
    def __init__(self, N):
        self.gru = nn.GRU(N, 128, num_layers=2, batch_first=True,
                          dropout=0.2, bidirectional=True)
        self.norm = nn.LayerNorm(128 * 2)
        self.head = nn.Sequential(
            nn.Linear(256, 128), nn.ReLU(), nn.Dropout(0.2),
            nn.Linear(128, N * 24))

# robust loss + decoupled weight decay + cosine LR
opt   = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
sched = CosineAnnealingLR(opt, T_max=50)
loss  = nn.HuberLoss(delta=1.0)
```

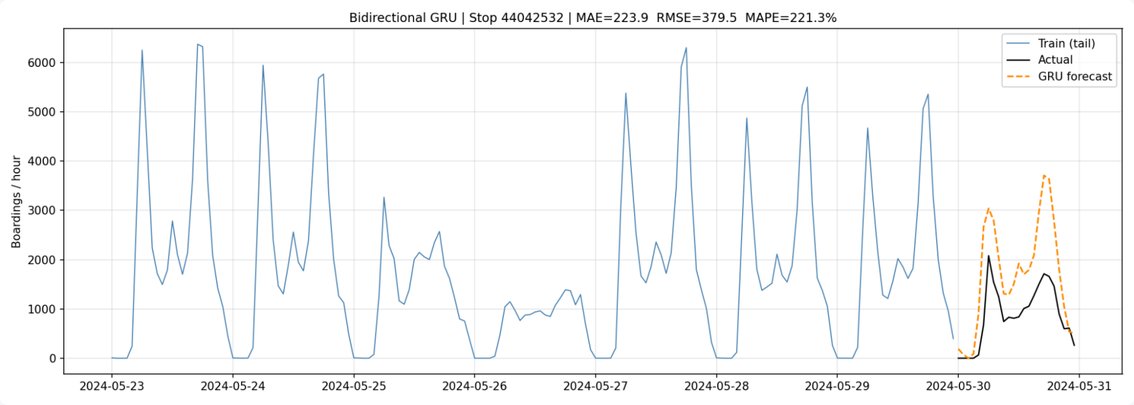
- **Same data & windows** as the LSTM — only the model class changes, so results are directly comparable.
- **Bidirectional GRU** reads each window forwards **and** backwards, then a **LayerNorm + MLP** head maps to the forecast.
- **HuberLoss** is robust to sharp boarding spikes — less outlier-sensitive than MSE.
- **AdamW** + **cosine** LR schedule for smoother convergence.

LIGHTER, OFTEN AS GOOD

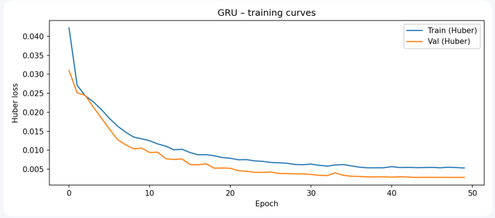
Two gates instead of three — **fewer parameters** than the LSTM, with comparable accuracy.

GRU — Training & Forecast

FORECAST VS. ACTUAL



TRAINING & VALIDATION LOSS



The GRU reaches accuracy on par with the LSTM while training faster — a practical default when forecasting demand at many stops simultaneously.

Transformers

The idea

Drop recurrence entirely. Every time step looks at **every other** directly through self-attention, and learns how much weight to give each one.

- Captures **long-range** dependencies in one hop.
- Fully **parallel** — no step-by-step bottleneck.
- Strong for **long-horizon** and complex dynamics.

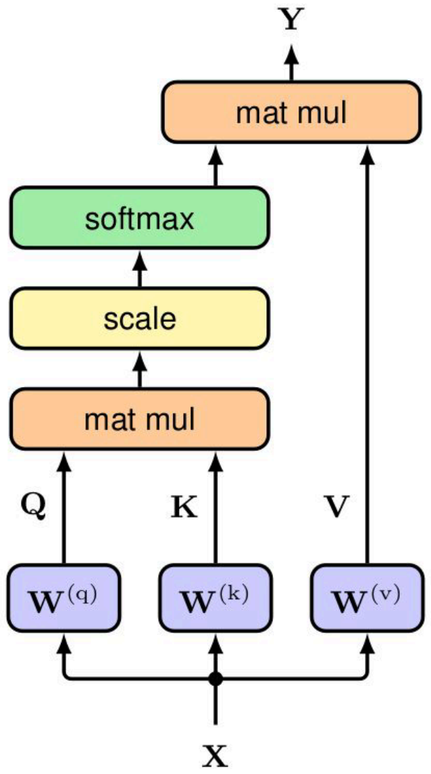
FORMAL DEFINITION

$$Attention(Q, K, V) =$$

$$softmax(\, Q\, K^{\top} / \sqrt{d_k}\,)\, V$$

Queries **Q**, keys **K** and values **V** are linear projections of the input. The softmax turns step-to-step similarities into attention weights over the whole sequence.

Inside a Self-Attention Head



Each input **X** is projected three ways by trainable weight matrices:

QUERY · KEY · VALUE

$$Q = X\,W^{(q)}$$

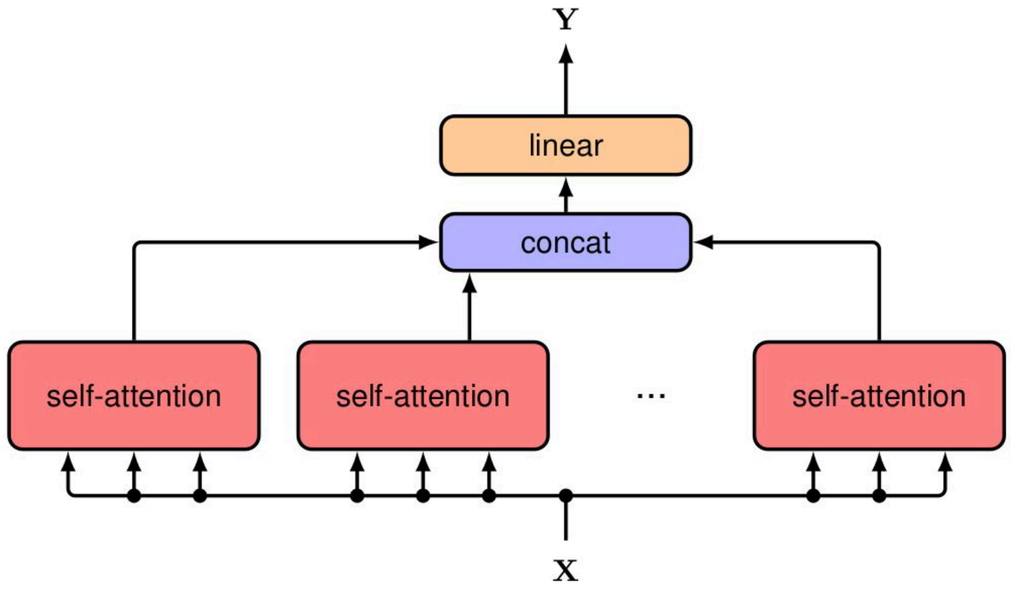
$$K = X\,W^{(k)}$$

$$V = X\,W^{(v)}$$

$W^{(q)}$, $W^{(k)}$, $W^{(v)}$ are learned. Scores QK^{T} are **scaled** by $\sqrt{d_k}$, passed through **softmax**, then used to weight **V** — the output **Y**.

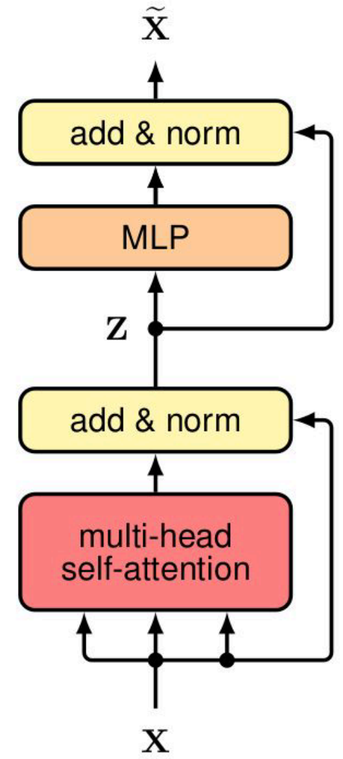
Multi-Head Attention & the Transformer Layer

MULTI-HEAD ATTENTION



Heads run in **parallel**, then **concat** + **linear**.

TRANSFORMER LAYER



Attention → **add & norm** → **MLP** → **add & norm**, with residuals.

Transformer in Practice

```python
# each time step = one token of dim = #stops
splits = prepare_sequences(ts, input_len=72, horizon=24)

class TransformerForecaster(nn.Module):
    def __init__(self, N):
        self.input_proj = nn.Linear(N, 64)        # d_model
        self.pos_enc    = SinusoidalPositionalEncoding(64)
        layer = nn.TransformerEncoderLayer(d_model=64,
            nhead=4, dim_feedforward=256,
            batch_first=True, norm_first=True)   # Pre-LN
        self.encoder = nn.TransformerEncoder(layer, 3)

    def forward(self, x):              # x: (B, 72, N)
        h = self.pos_enc(self.input_proj(x))
        h = self.encoder(h).mean(dim=1)    # avg-pool over time
        return self.head(h).view(-1, 24, N)

# AdamW + linear warm-up → cosine decay
opt = torch.optim.AdamW(model.parameters(), lr=5e-4)
```

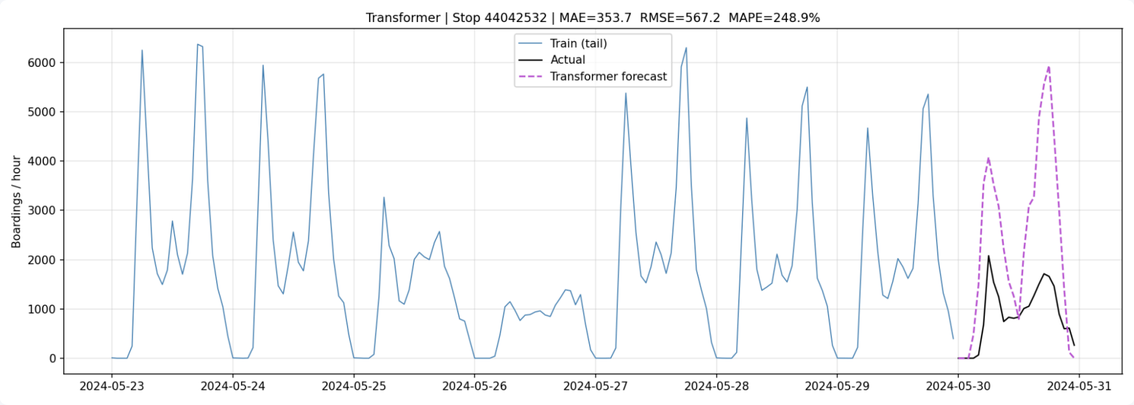
- **Tokens over time** — each hourly step is a token of dimension **#stops**, linearly projected to `d_model=64`.
- **Positional encoding** — sinusoidal, so the model knows the order of the sequence.
- **3 encoder layers**, **4 heads**, **Pre-LN** (norm_first) for stable training; **avg-pool** over time → forecast head.
- **Warm-up → cosine** LR with AdamW and HuberLoss — the standard recipe for training attention models.

SAME 72 H → 24 H WINDOW

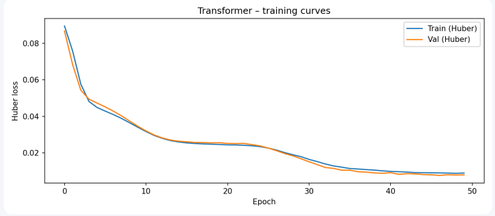
Identical data split to SARIMA, LSTM and GRU — so the numbers are directly comparable.

Transformer — Training & Forecast

FORECAST VS. ACTUAL



TRAINING & VALIDATION LOSS



Attention holds accuracy further into the horizon, where recurrent models tend to drift — at the cost of more data and compute to train.

Time-Series Foundation Models

The idea

Large models pre-trained on huge, heterogeneous collections of time series. They forecast a **new** series with little or no task-specific training.

- **Zero-shot** — apply directly, no fitting.
- **Few-shot** — light fine-tuning on a little SUNT data.
- Examples: **TimesFM**, **TEMPO**, **Chronos**.

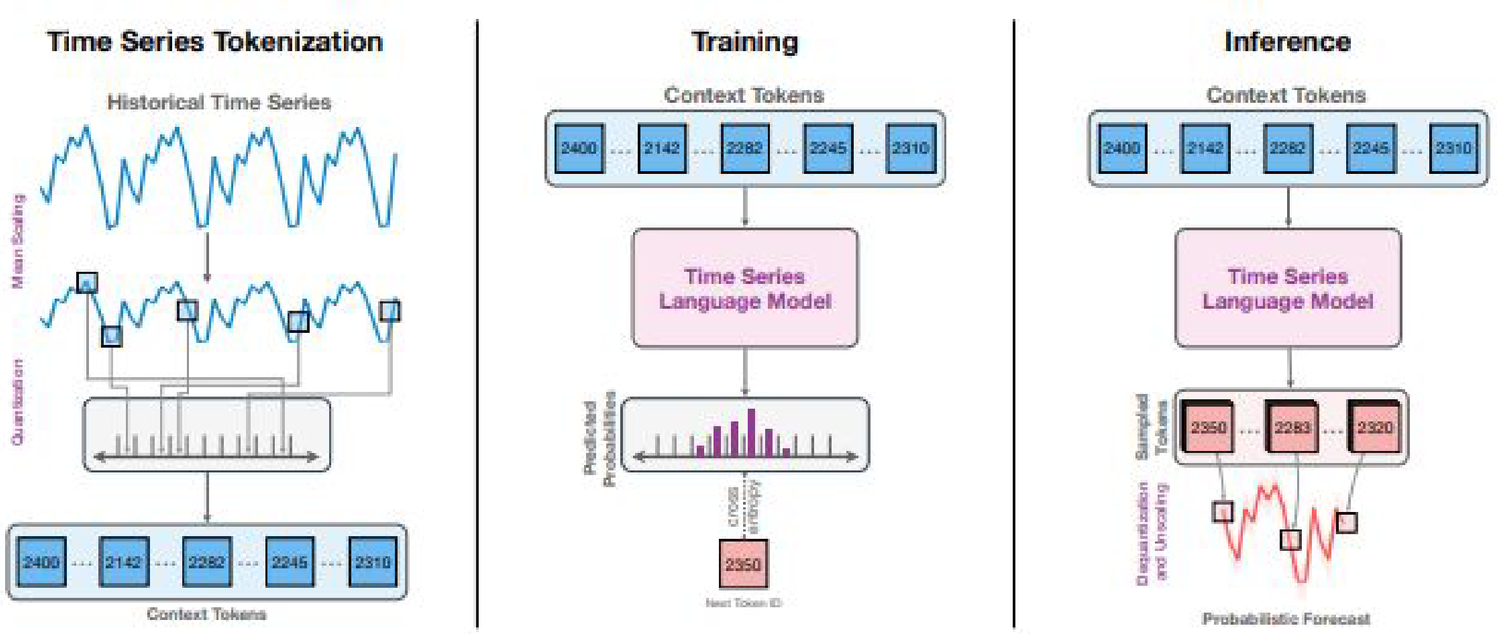
FORMAL DEFINITION

$$f_\theta : x_{1:t} \mapsto x_{t+1:t+h}$$

θ is pre-trained once on many datasets and then either **frozen** (zero-shot) or lightly adapted (few-shot) — robust forecasts with minimal SUNT-specific effort.

Chronos — Language Modeling for Time Series

Chronos (TMLR, 2024) **tokenizes** a series into discrete bins, then trains a standard **language model** to predict the next token.



Tokenization — mean-scale then quantize into token IDs. **Training** — an LM learns next-token probabilities via cross-entropy. **Inference** — sample tokens autoregressively, then unscale to a **probabilistic forecast**.

The Chronos Pipeline, Step by Step

From real values to tokens

- **1 · Scaling** — mean-scale the series (m = 0) by its mean absolute value **s**, so magnitudes are comparable across datasets.
- **2 · Quantization** — map each scaled value into one of **B** bins via edges $\mathbf{b_1 < \ldots < b_{B-1}}$, giving an integer token.
- **3 · Vocabulary** — the **B** bin tokens plus special **EOS** and **PAD** symbols form $\mathcal{V}_{\text{ts}}$.
- **4 · Train · 5 · Forecast** — train as a language model, then sample tokens and unscale back to values.

1 · MEAN SCALING

$$s = (1/t)\,\Sigma_{i=1}^{t} \mid x_i \mid$$

2 · QUANTIZATION

$$q(x) = 1 \text{ if } x < b_1;\ \ldots;\ B \text{ if } x \geq b_{B-1}$$

4 · TRAINING LOSS

$$\ell(\theta) = -\,\Sigma_h\,\Sigma_i\, 1_{(z=i)}\, \log p_\theta(\, z_{C+h+1} = i \mid z_{1:C+h}\,)$$

Standard **cross-entropy** over the token vocabulary — exactly a language-model objective, applied to quantized time-series tokens.

Chronos in Practice

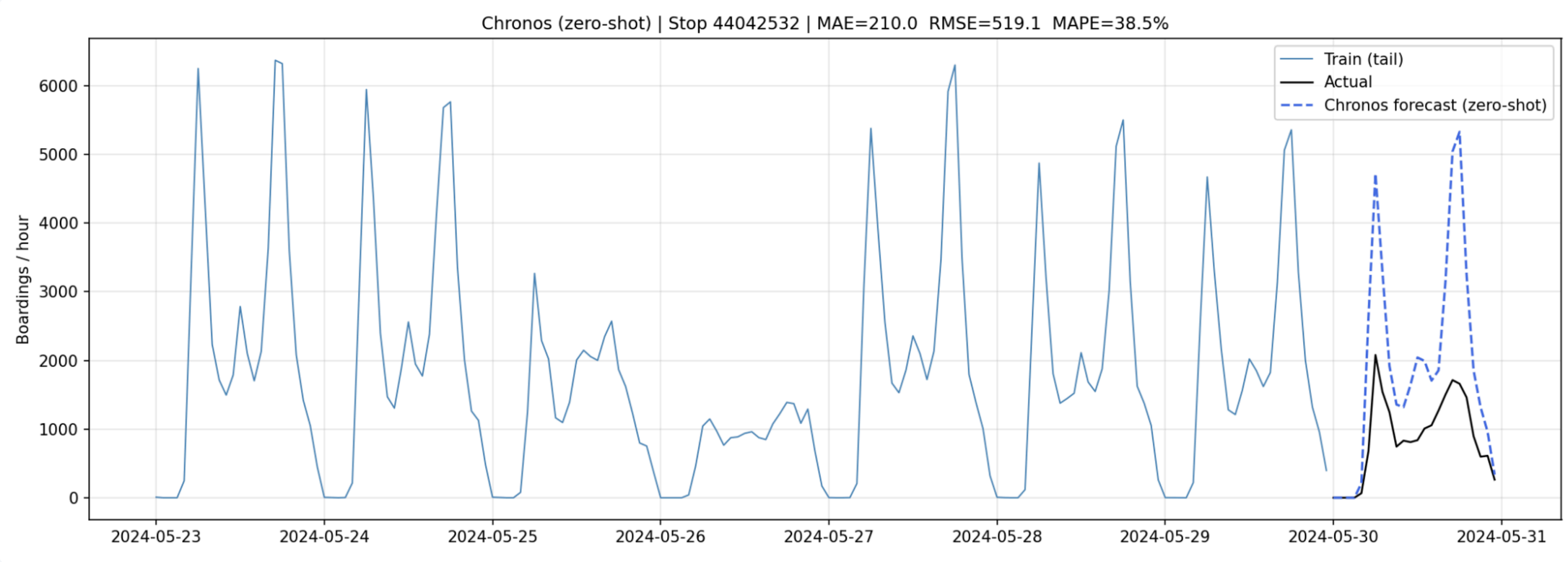
```python
# zero-shot: load pre-trained weights, no training
from chronos import ChronosPipeline

pipe = ChronosPipeline.from_pretrained(
    "amazon/chronos-t5-small",   # T5 encoder-decoder
    device_map="cpu", torch_dtype=torch.bfloat16)

# foundation models normalise internally →
# feed RAW counts, not the MinMax-scaled values
context = torch.tensor(X_test_raw[:, :, stop])   # (B, 128)

# probabilistic forecast — median as point estimate
forecast = pipe.predict(context,
    prediction_length=24, num_samples=10)
y_hat = forecast.median(dim=1).values        # (B, 24)
```

- **Zero-shot** — pre-trained T5 weights (≈46 M params) loaded from Hugging Face, applied with **no fine-tuning**.
- **Univariate** — one stop at a time; context of `input_len=128` (~5 days).
- **Feed raw counts** — Chronos scales internally, so MinMax-scaled input would distort its learned priors.
- **Probabilistic** — draw **10 samples**, take the **median** as the point forecast.

NO TRAINING, SAME TEST

Evaluated on the identical **24 h** horizon and metrics as every other model — a true out-of-the-box baseline.

Foundation Model — Forecast



A pre-trained foundation model applied to SUNT demand with minimal tuning. It produces credible **multi-horizon** forecasts out of the box — a strong starting point when little task data or training budget is available.

Graph Neural Networks

The idea

The models so far treat each stop on its own. GNNs let neighbouring stops **share information** along the network graph — then combine that with a temporal model.

- Each node aggregates features from its **neighbours**.
- Pair with RNN / Transformer for **spatio-temporal** forecasting.
- Best for **network-wide, location-aware** demand.

FORMAL DEFINITION

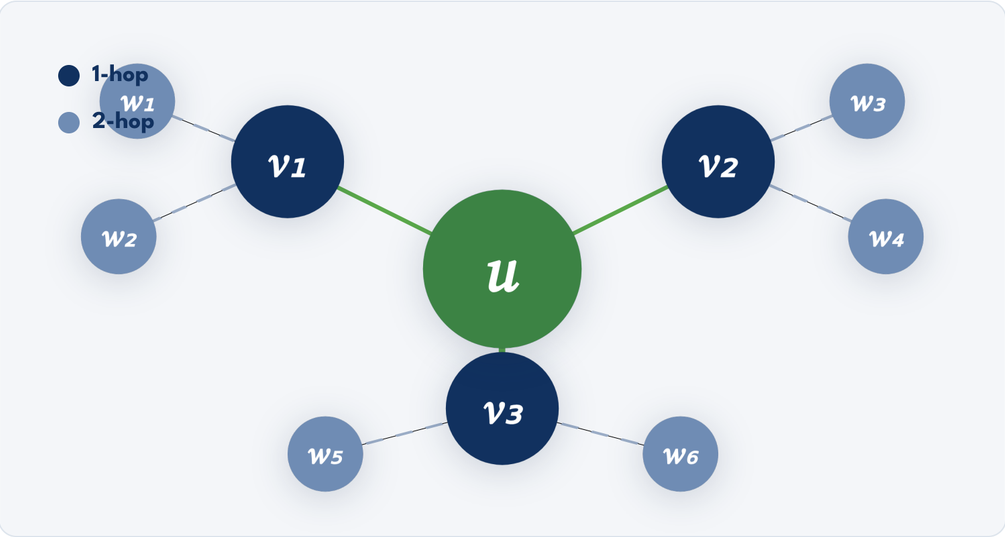
$$h_v^{(l+1)} = \sigma\left(\Sigma_{u \in \mathcal{N}(v)} \; c_{uv} \, W^{(l)} \, h_u^{(l)} \right)$$

A node's new representation mixes its neighbours' features, normalised by c_{uv}. Stacking layers widens the spatial reach across the network.

Neural Message Passing

The core operation of a GNN. At each iteration **k**, every node's hidden embedding $\mathbf{h_u}$ is updated from the **aggregated information of its neighbours** $\mathcal{N}(u)$.

- **AGGREGATE** — gathers neighbour embeddings into one message $\mathbf{m}_{\mathcal{N}(u)}$.
- **UPDATE** — combines that message with the node's own previous embedding.
- Both are **any differentiable function**; stacking **k** iterations widens each node's reach across the graph.



UPDATE PROCESS · ITERATION K

$$h_u^{(k+1)} = UPDATE(\, h_u^{(k)},\ m_{\mathcal{N}(u)}^{(k)} \,)$$

$$m_{\mathcal{N}(u)}^{(k)} = AGGREGATE^{(k)}(\, h_v^{(k)},\ \forall v \in \mathcal{N}(u) \,)$$

A Basic GNN Layer → GCN

Make it concrete

The simplest choice: **sum** the neighbour embeddings, then pass a learned linear mix through a non-linearity.

- $\mathbf{W}_{self}$, $\mathbf{W}_{neigh}$ are trainable weight matrices; **b** is a bias.
- **σ** is a non-linearity (tanh or ReLU).
- **GCN** generalises AGGREGATE by **normalising** the neighbourhood — mean, or symmetric normalisation — to keep scales stable across high- and low-degree stops.

BASIC GNN

$$m_{\mathcal{N}(u)} = \Sigma_{v \in \mathcal{N}(u)} h_v$$

$$h_u = \sigma(W_{self} h_u + W_{neigh} m_{\mathcal{N}(u)} + b)$$

GCN · SYMMETRIC NORMALISATION

$$h_u = \sigma(W \Sigma_{v \in \mathcal{N}(u) \cup \{u\}} h_v / \sqrt{(|\mathcal{N}(u)||\mathcal{N}(v)|)})$$

Dividing by node degrees stops high-degree hubs from dominating the sum — the key idea behind the **Graph Convolutional Network**.

GCN in Practice

```python
# build graph → symmetric-normalised adjacency
G   = build_graph_from_od(load_od())
A   = adjacency_matrix(G, top_stops)      # (N, N)
Â   = D**-.5 @ A @ D**-.5 + np.eye(N)   # + self-loops

# one spectral GCN layer:  H' = σ( Â · H · W )
class GraphConvolution(nn.Module):
    def forward(self, x, adj):       # x: (B, N, f_in)
        return adj @ (x @ self.weight)   # (B, N, f_out)

# stack k layers → linear head over the horizon
class GCNForecaster(nn.Module):
    def forward(self, x, adj):       # x: (B, N, 72)
        for gcn in self.layers:
            x = self.dropout(torch.relu(gcn(x, adj)))
        return self.head(x)             # (B, N, 24)
```

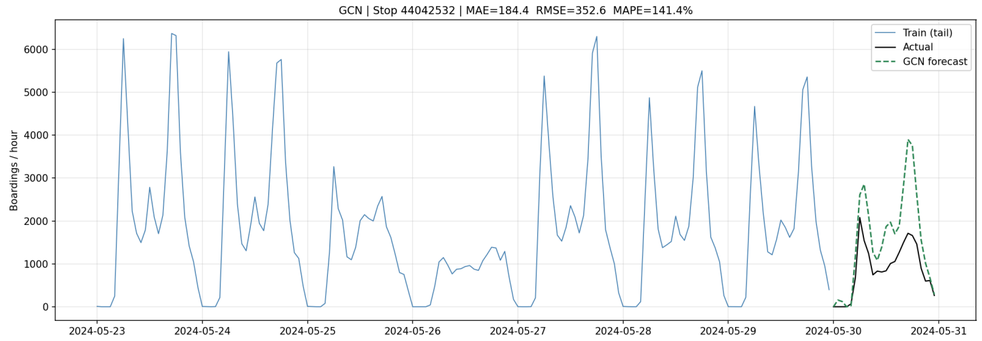
- **Pure spatial** — the 72 past hours are per-node features; **no recurrence**, all graph convolution.
- **Normalised adjacency** $\hat{A} = D^{-½} A D^{-½} + I$ — exactly the GCN rule from the previous slide, with self-loops.
- Each layer is two matmuls: **Â · H · W** → fast, since Â is fixed.
- 3 GCN layers (hidden 64) → linear head; Adam + MSE, **step** LR, gradient clipping.

KIPF & WELLING, ICLR 2017

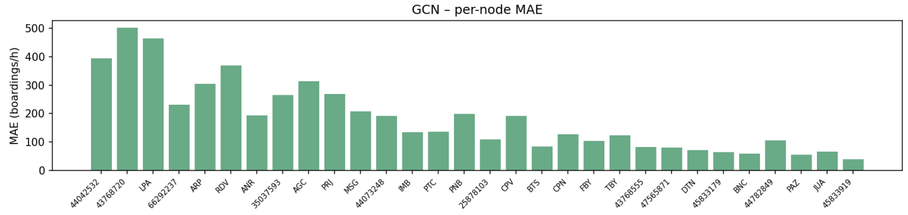
Isolating graph convolution makes the role of the **adjacency matrix** easy to inspect.

GCN — Forecast & Per-Node Error

FORECAST VS. ACTUAL · BUSIEST STOP



PER-NODE MAE · WHICH STOPS BENEFIT FROM SPATIAL CONTEXT



The GCN tracks the daily peaks by borrowing signal from neighbouring stops. **Per-node MAE** shows error concentrates on a few very high-volume hubs, while most stops forecast tightly — training converged smoothly over 60 epochs.

From GCN to T-GCN: Why Add Time?

The pure GCN captures **where** demand sits on the network — but it folds the whole 72-hour window into a flat feature vector, so the **order of the hours is lost**.

What the GCN gives us

SPACE

- Each stop **borrows signal** from its graph neighbours.
- Hubs and corridors inform nearby stops.
- One shared, normalised adjacency Â.

What is still missing

TIME

- The 72 input hours enter as **unordered features**.
- No notion of **sequence** — morning vs. evening looks the same.
- Daily and weekly **dynamics** go unmodelled.

Demand is a **sequence**, not a bag of numbers. The fix: keep the GCN's spatial aggregation, but process the resulting embeddings **through time** — that is exactly what T-GCN does.

Composing Space and Time

T-GCN stacks the two modules we already know: a **GCN** for space at every step, then a **GRU** for time across steps.

PER TIME STEP

Graph snapshot

Boardings at all stops at hour *t*.

$x_t \in \mathbb{R}^N$

→

SPACE

GCN layer

Aggregate from graph neighbours.

$h_t = \sigma(\hat{A}\, x_t\, W)$

→

TIME

GRU over t = 1…T

Model the sequence of spatial embeddings.

$z_T = GRU(h_{1:T})$

→

OUTPUT

Forecast head

Next 24 hours, per stop.

$\hat{y} \in \mathbb{R}^{N \times 24}$

Space first, then time — the GCN answers **“where”** at each instant, the GRU answers **“when”** across the window. Same Â and the same 72 h → 24 h split as every other model.

T-GCN in Practice

```python
# T-GCN = GCN (space) + GRU (time)
class TGCN(nn.Module):
    def forward(self, x, adj):       # x: (B, N, 72)
        # 1 · GCN at every time step → spatial embedding
        seq = []
        for t in range(T):
            h = x[:, :, t].unsqueeze(-1)   # (B, N, 1)
            for gcn in self.gcn_layers:
                h = torch.relu(gcn(h, adj))
            seq.append(h)
        seq = torch.stack(seq, dim=2)       # (B, N, T, H)

        # 2 · GRU over the spatial embeddings → time
        out, _ = self.gru(seq.reshape(B*N, T, -1))
        return self.head(out[:, -1]).view(B, N, 24)
```

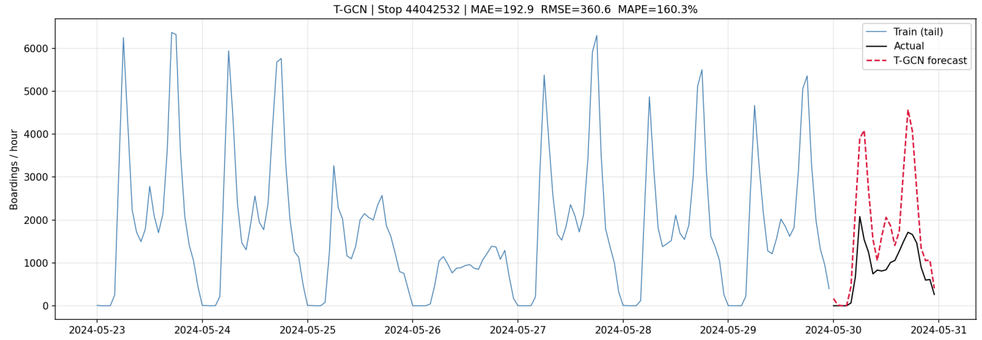
- **Space + time together** — a GCN encodes neighbours at **each** time step, then a GRU models the temporal sequence of those embeddings.
- Same **normalised adjacency** Â and 72 h → 24 h windows as the pure GCN.
- GCN hidden 64 → GRU hidden 64 → linear head; Adam + MSE, step LR.

ZHAO ET AL., 2020

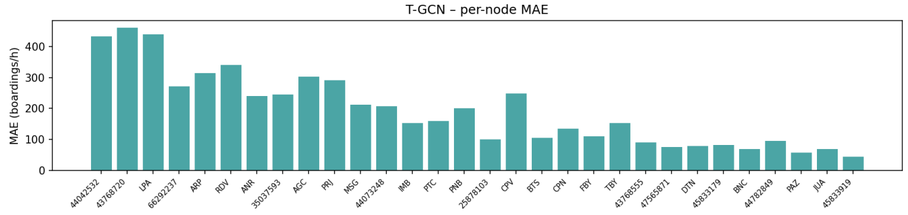
The spatio-temporal model: where graph convolution and recurrence finally meet on one network.

T-GCN — Forecast & Per-Node Error

FORECAST VS. ACTUAL · BUSIEST STOP



PER-NODE MAE ACROSS THE NETWORK



Adding temporal recurrence lets T-GCN react more sharply to the demand peaks than the pure GCN. As before, error concentrates on the highest-volume hubs — these are the stops where richer models pay off most.

GCN, T-GCN & GAT: What Changes?

All three work on the same graph — they differ in **how they treat time** and **how much each neighbour counts**.

Model	Uses the graph	Models time	Learns neighbour weights
GCN	Yes	No	No — fixed by Â
T-GCN	Yes	Yes	No — fixed by Â
GAT	**Yes**	**No**	**Yes — via attention**

Each adds one idea to the last: T-GCN adds **time** to the GCN; GAT adds **learned importance** to the neighbours. The next slides take them one at a time.

GCN: Learn from Your Neighbours

The intuition

A GCN updates each node by **averaging information from the stops connected to it**. A stop's representation depends on its neighbours in the network.

- All neighbours contribute through a **fixed**, normalised adjacency Â.
- Great at capturing **spatial** relations in the graph.
- But it does **not** model time on its own.

ONE GRAPH-CONVOLUTION LAYER

$$H^{(l+1)} = \sigma(\hat{A}\, H^{(l)}\, W^{(l)})$$

$\hat{A}$ → normalised adjacency (with self-loops) · $H^{(l)}$ → node features at layer l · $W^{(l)}$ → learned weights.

T-GCN: Graph + Time

The intuition

T-GCN looks at the neighbours **and** at the temporal history. It runs a GCN at each step, then a recurrent unit over those embeddings.

- Combines **GCN** (space) with a **GRU / LSTM** (time).
- Best fit for **time series on graphs** — traffic, sensors, transit.
- Neighbour weights are still fixed by Â.

SPACE THEN TIME

$$GCN \; + \; GRU \,/\, LSTM$$

$$x_{t-3},\, x_{t-2},\, x_{t-1} \; + \; graph \; \longrightarrow \; \hat{x}_t$$

A GCN encodes neighbours at each step; the recurrent unit carries those spatial embeddings **through time** to forecast the next step.

GAT: Not Every Neighbour Counts Equally

The intuition

A GAT uses **attention** to learn how important each neighbour is to a given node — instead of weighting them all the same.

- Each edge gets a learned weight α_{ij}.
- Aggregation becomes **flexible and data-driven**.
- More expressive, but can be **more costly** to train.

ATTENTION-WEIGHTED AGGREGATION

$$h_i' = \sigma\left(\Sigma_{j \in \mathcal{N}(i)} \alpha_{ij} W h_j \right)$$

α_{ij} → learned attention weight of neighbour *j* for node *i*. High α = strong influence; low α = weak — the model decides per edge.

GAT in Practice

```python
# T-GAT = GAT (attention over space) + GRU (time)
from torch_geometric.nn import GATConv

class GraphAttentionLayer(nn.Module):
    def forward(self, x, adj):        # x: (B, N, f_in)
        Wh = self.W(x)                    # shared projection
        # score every edge:  e_ij = LeakyReLU(aᵀ[Wh_i‖Wh_j])
        e = self.leaky_relu((cat * self.a).sum(-1))
        e = e.masked_fill(adj == 0, -inf)   # neighbours only
        α = F.softmax(e, dim=2)             # learned weights
        return F.elu((α @ Wh).mean(dim=heads))

class TGAT(nn.Module):
    def forward(self, x, adj):        # x: (B, N, 72)
        seq = [self.gat(x[:,:,t], adj) for t in range(T)]
        out, _ = self.gru(torch.stack(seq, 2))   # temporal
        return self.head(out[:, -1])           # (B, N, 24)
```

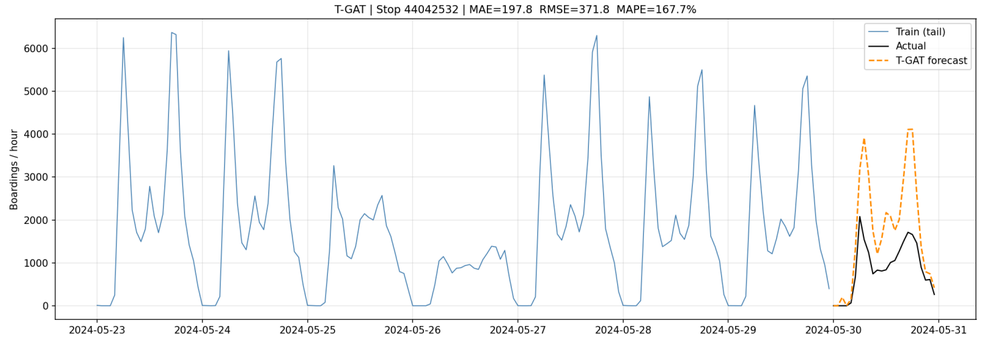
- **Learned edge weights** — instead of the fixed Â, GAT scores each neighbour with $\alpha_{ij} = \mathbf{softmax(LeakyReLU(a^T[Wh_i \| Wh_j]))}$.
- **Multi-head** attention (4 heads) — several views of "who matters", then averaged.
- Same recipe as T-GCN, swapping the GCN for **GAT layers** → still GRU over time.
- Uses **torch-geometric**'s `GATConv`, with a dense-attention fallback.

VELIČKOVIĆ ET AL., ICLR 2018

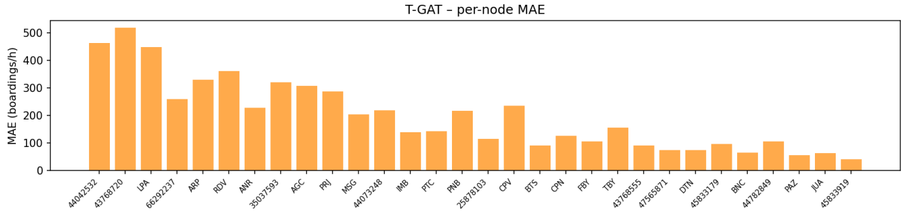
The attention map is **inspectable**: it shows which stops each hub actually listens to.

T-GAT — Forecast & Per-Node Error

FORECAST VS. ACTUAL · BUSIEST STOP



PER-NODE MAE ACROSS THE NETWORK



Learned attention catches the **timing** of the peaks, but here it **overshoots their height** — driving a high MAPE on this very spiky stop. As with the other graph models, error concentrates on a handful of **high-volume hubs**; most stops stay well below 150 boardings/h MAE.

In One Line Each

GCN

Who are my neighbours?

Learns each stop's representation **from the stops connected to it**.

T-GCN

How do my neighbours evolve in time?

Learns from the neighbours **across the temporal history**.

GAT

Which neighbours matter most?

Learns **how much each neighbour counts** through attention.

Which Model for Which Question?

Model	Captures	Best for	Cost to train
SARIMA	Linear trend & seasonality	Interpretable baseline, single series	Low
LSTM / GRU	Non-linear sequential patterns	Per-stop & multi-stop demand	Medium
Transformer	Long-range dependencies	Long-horizon forecasting	High
Foundation model	Cross-dataset priors	Zero / few-shot, little data	None → low
GCN	Spatial structure of the network	Location-aware, neighbour-sharing	High
T-GCN	Space **and** time jointly	Network-wide spatio-temporal demand	High
GAT	Learned neighbour importance	Graphs where some links matter more	High

No single winner — start with the **SARIMA baseline**, reach for **recurrent or attention** models as patterns get richer, and use **foundation or graph** models when data is scarce or space matters.

Key Takeaways

01 · THE DATA

A real, city-scale benchmark

SUNT brings a full year of sub-minute Salvador transit — far larger and finer than classical spatio-temporal datasets, and openly available.

02 · TWO VIEWS

Space and time, together

Network views expose corridors and hubs; time-series views expose daily and weekly cycles. Both are one `pip install` away.

03 · MANY MODELS

From baselines to foundations

The same demand series feeds SARIMA, LSTM/GRU, Transformers, foundation and graph models — a ready-made playground for comparison.

04 · YOUR TURN

Reproducible by design

Published code, data and a peer-reviewed descriptor mean every result here is something you can run, extend and build on today.

THANK YOU

Acknowledgements

SUNT is supported by the institutions and funding agencies that make open, reproducible research possible.

THANK YOU

Now go build with SUNT.

Data, code and the paper are open — questions, collaborations and pull requests are all welcome.

Ricardo A. Rios · ricardoar@ufba.br
Tatiane N. Rios · Felipe Fernandes · Marcos Ferreira
nature.com/articles/s41597-025-05674-6