TrafficLens: Desktop Traffic Viewer and Benchmark for Taiwan Highways

LI Mingzhao, 21229013 XU Chenrui, 21272589 ZHANG zijian, 21230933

Contents

1 Introduction

`TrafficLens` is a Python-based desktop application for analyzing, visualizing, and modeling traffic data. It features a GUI built with Tkinter, data processing with pandas, visualization via Matplotlib, and spatio-temporal traffic flow forecasting using PyTorch. The complete source code is openly available at: https://github.com/akatsuky999/TrafficLens.

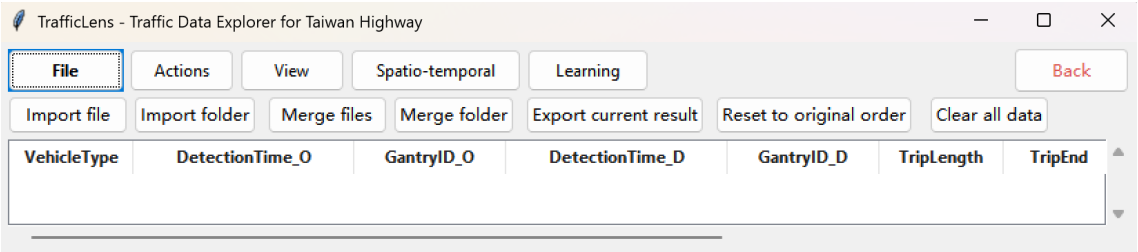


Figure 1: Initial interface example

`TrafficLens` provides a comprehensive set of functionalities for traffic data management and analysis, including:

- **Importing** and **exporting** traffic data files, including **batch(merge)** file import for handling multiple datasets simultaneously.

- **Sorting** and **searching** data by specific fields, and displaying data within user-defined ranges for efficient exploration.
- Data **visualization** and statistical analysis, offering charts, plots, and summary statistics to reveal traffic patterns.
- Generation of **spatio-temporal (ST) traffic flow data** from raw logs, with 2D and 3D visualization capabilities for deeper insights.
- Spatio-temporal traffic modeling based on ST data, including training and **benchmarking** deep learning algorithms, and visualizing predictions.

Table 1: Key Directories of TrafficLens

Directory	Description
trafficlens/	Core GUI and data processing logic. `gui.py` is the main application controller.
Learning/	Deep learning module with training, prediction, and model definitions. Models are pluggable via `model_config.py`.
data/	Input CSV files.
checkpoint/	Model checkpoints saved during training.
output_data/	Prediction outputs saved after inference.

2 UML Design

This section introduces the UML diagrams used in the TrafficLens project. The main interface is built with Tkinter, while the backend relies on pandas for data handling and PyTorch for deep learning tasks.

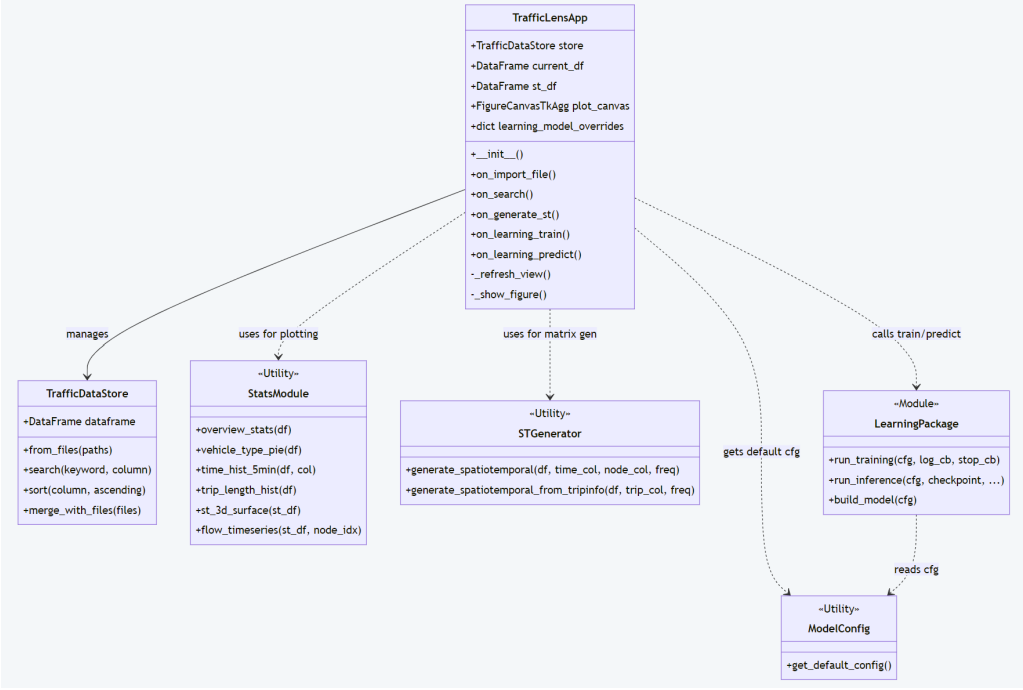


Figure 2: System Architecture Map (main function examples)

2.1 Data Workflow

This sequence diagram presents the essential data operations: Import, Sorting, Searching, and Filtering, which form the foundation of user interaction with the dataset.

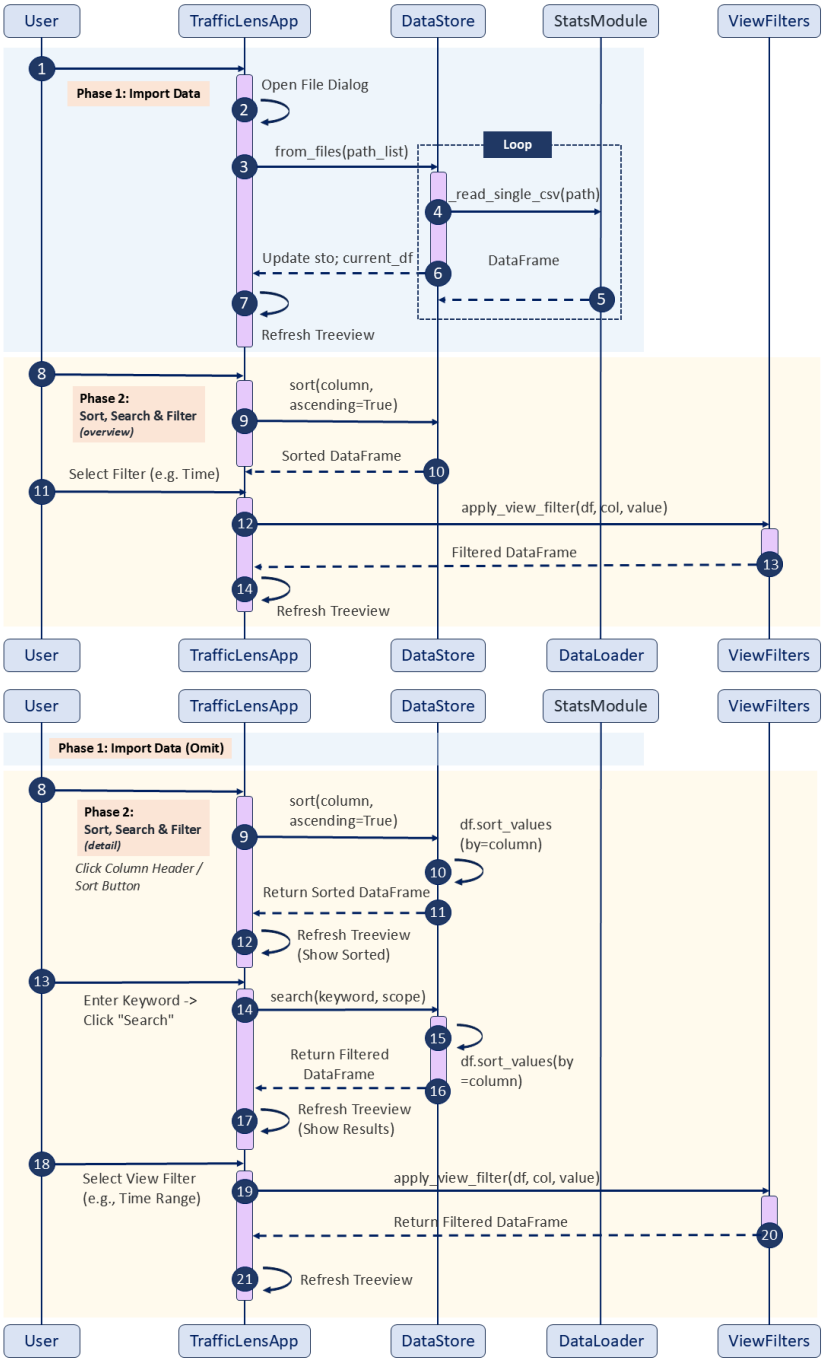


Figure 3: Data Workflow UML Overview & Details (Search, Sort)

- **Data Import**: When the user clicks “Import File”, the GUI calls `TrafficDataStore.from_files`. Internally, this method uses a `ThreadPoolExecutor` to parse multiple CSV files concurrently, ensuring the interface stays responsive even for large datasets.
- **Sorting**: When users click a column header, the GUI triggers

`TrafficDataStore.sort`. The method performs sorting directly on the underlying pandas DataFrame and updates the Treeview display.

- **Searching**: The search function is handled through `TrafficDataStore.search`, which performs a keyword-based query across selected columns (or all columns) and returns a filtered view.
- **Filtering**: The `ViewFilters` module offers additional filtering logic, such as time-range selection or filtering by vehicle type, allowing more focused data exploration.

2.2 ST Workflow

The Spatio-Temporal (ST) Workflow explains how raw detection logs are transformed into structured matrices for visualization and later modeling.

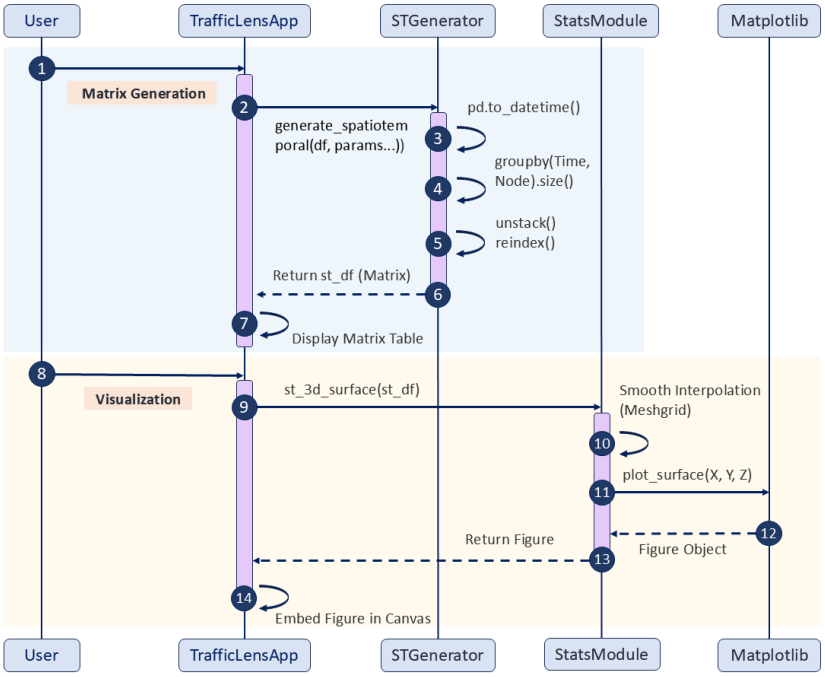


Figure 4: Spatial-Temporal Workflow overview

- **Matrix Generation**: When requested by the user, the `STGenerator` module processes the raw DataFrame by grouping traffic records into fixed time intervals (e.g., 5-minute bins) and sensor nodes. This produces a dense time–node matrix, where missing values are filled appropriately.
- **Visualization**: The generated matrix is passed to the `StatsModule`. The `st_3d_surface` function creates a 3D surface plot using Matplotlib, showing how traffic intensity changes across both time and space. The generated figure is embedded directly into the Tkinter canvas for interactive viewing.

2.3 AI Workflow

This section outlines the asynchronous workflow designed for deep learning tasks, ensuring that training does not block the GUI.

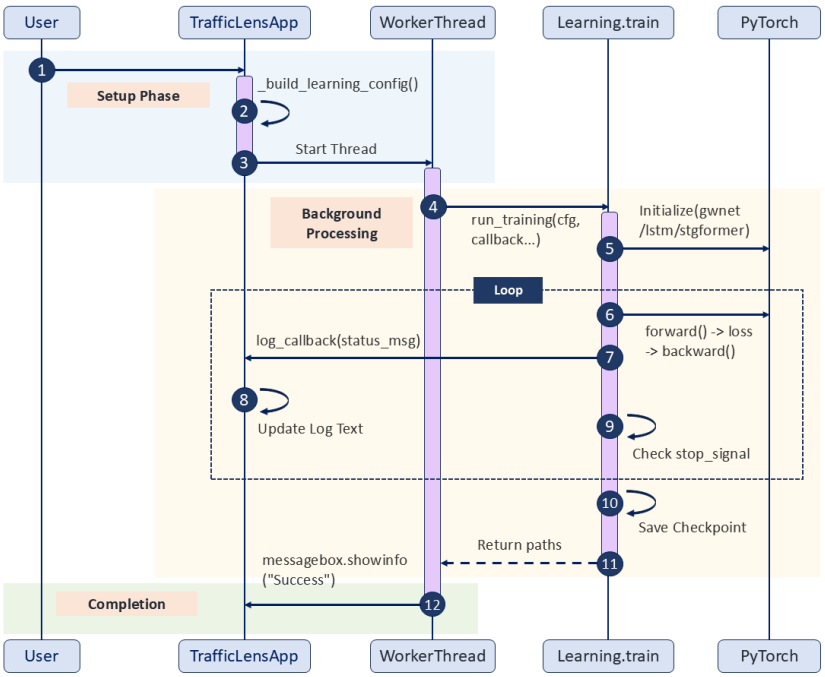


Figure 5: AI Workflow overview

- **Initialization**: After the user sets hyperparameters and selects "Train", the GUI builds a configuration dictionary using `ModelConfig` and starts a background `WorkerThread`.
- **Asynchronous Training**: The training logic is implemented in `Learning.train`. The `run_training` function initializes the selected model (GWNet, LSTM, or STGformer) and runs the training loop.
- **Real-time Feedback**: During training, the backend sends messages to the GUI through a callback function `log_callback`. This updates the training log and loss curve in real time without interrupting user interaction.
- **Checkpointing**: When training finishes or stops early, the model is saved as a `.pth` checkpoint file, which can later be loaded for inference or prediction.

3 GUI Design and Tutorial

Based on the overall design philosophy, we have drawn inspiration from the interface architecture of Microsoft Excel, incorporating five primary functional modules in the navigation bar, each of which contains systematically organized sub-functions.

3.1 *File*: File Management

This section demonstrates the application's file import, export, and merging functions. For the import function, `TrafficLens` uses the "Import file" button to import data; a notification will pop up upon successful import. `TrafficLens` also includes a folder import function, which requires that the CSV files within the folder contain data of the same type, and automatically merges the internal files in order after folder import.

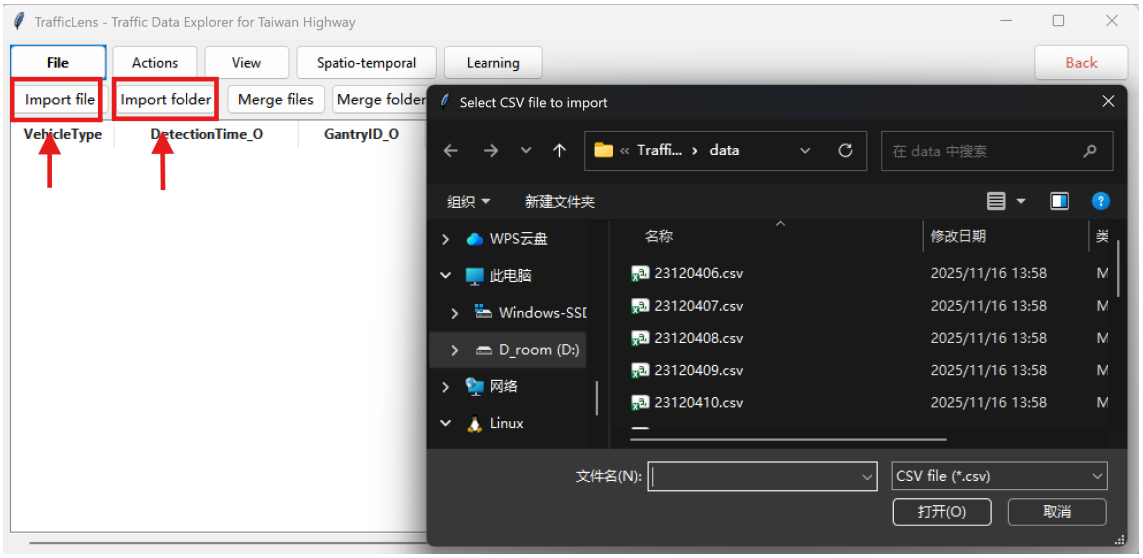


Figure 6: Import files and folders

After the data import is successful, if there is a need to merge data, `TrafficLens` still provides the function of merging files and folders, and other operations are the same as above.

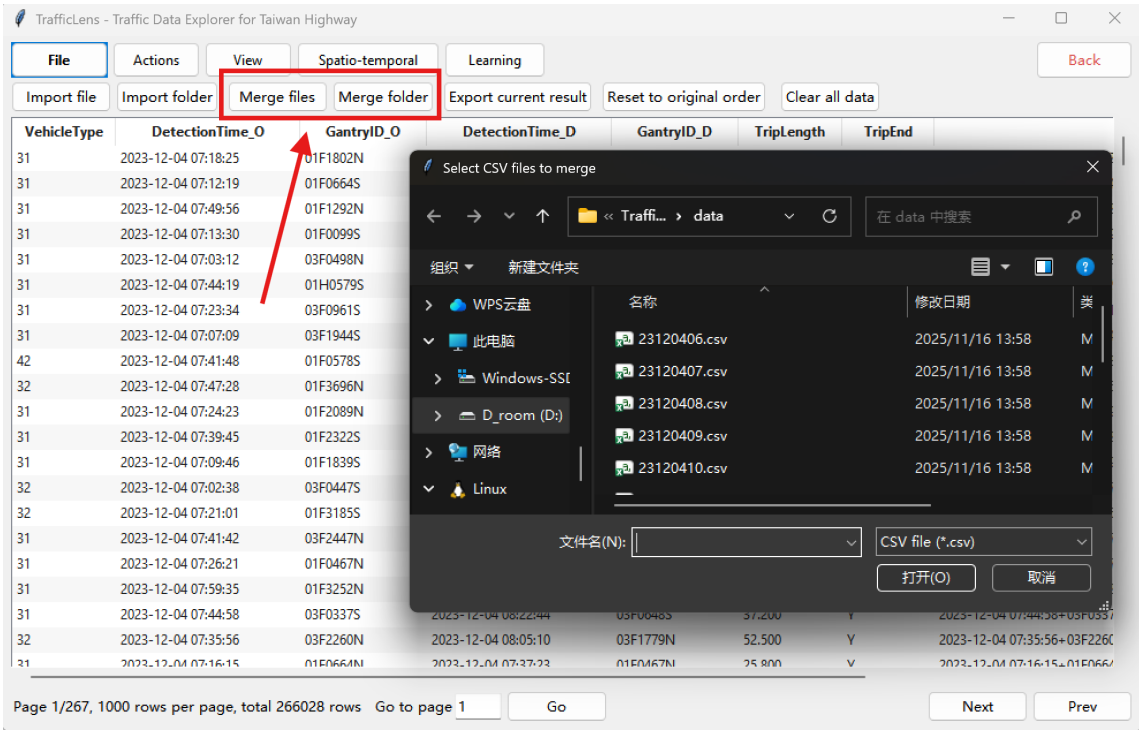


Figure 7: Merge files and folders

Upon importing data, `TrafficLens` further allows users to clear existing data from the current view. This functionality ensures efficient re-import operations in cases of incorrect file submissions without compromising data integrity.

For file export functionality, `TrafficLens` allows export of three file formats: `CSV`, `Xlsx`, and `Npy`.

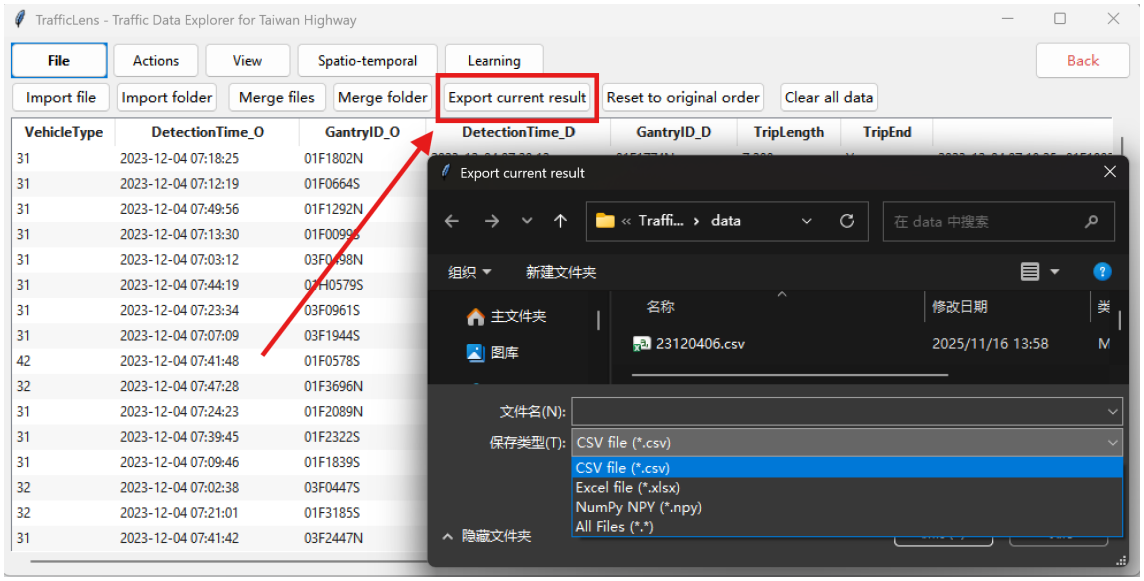


Figure 8: Export file or data

3.2 *Actions*: Search and Sort

In the ”Action” interface of `TrafficLens`, we have implemented two core functions: ”Sorting” and ”Searching” for DataFrames. Based on experimental findings from our open-source project SortTester[1], we found that the quicksort algorithm consistently demonstrates optimal time efficiency across all data fields. Therefore, `TrafficLens` utilizes Python's built-in quicksort function as the backend foundation for its sorting feature. For the search functionality, the system supports both exact matches and fuzzy searches within specified columns.

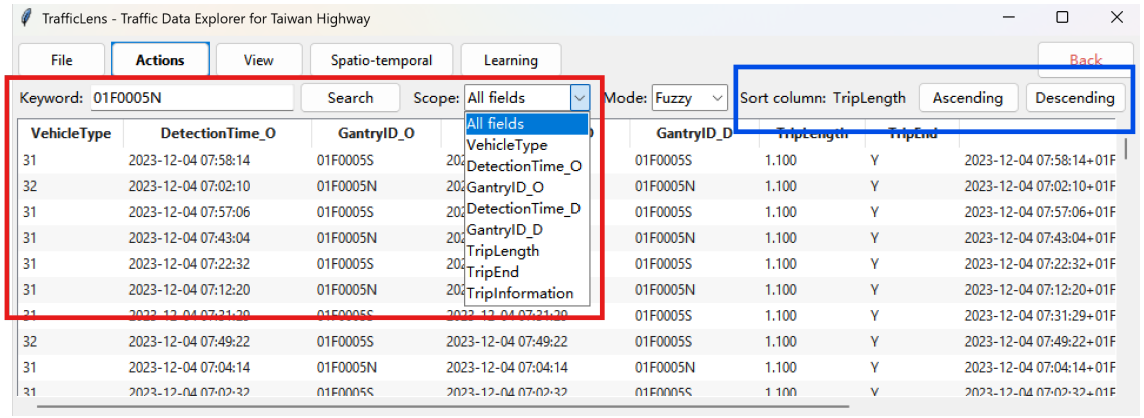


Figure 9: Search and Sort

Unlike the search function, which requires manual column selection, the sorting feature allows users to activate sorting by simply clicking on any column header directly.

[1] https://github.com/akatsuky999/SortTester.

Upon a successful search, the current view will automatically filter to display only the entries containing the target content. Simultaneously, search logs and result statistics are printed in the lower section of the GUI. To return to the original view, users can simply click the red ”Back” button located in the upper-right corner.

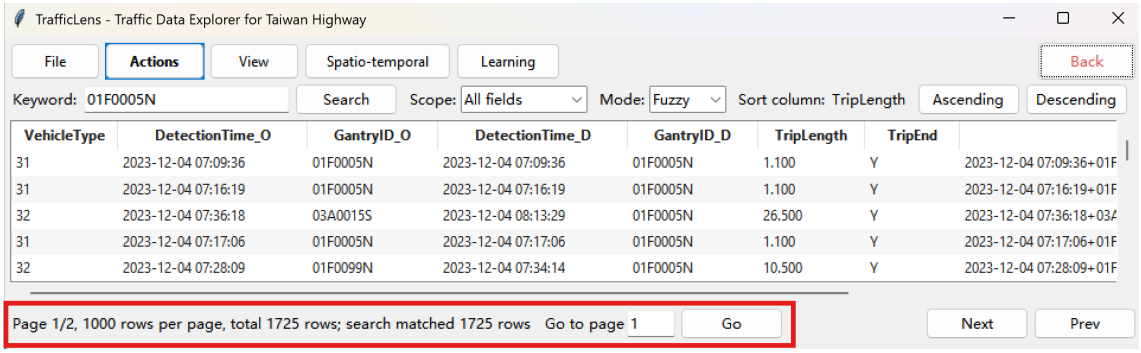


Figure 10: Search Example

3.3 *View*: Statistics and Visualization

The most fundamental function of the view module is ”Filtering,” as shown in the red area in the figure below. Users can freely select a field, and upon selection, the blue area will display the value range of that field. Based on the displayed range, users can then input their desired range in the red section. After clicking the ”Apply Filter” button, the main interface will only show the filtered sample data, while the blue section will report the filtering statistics.

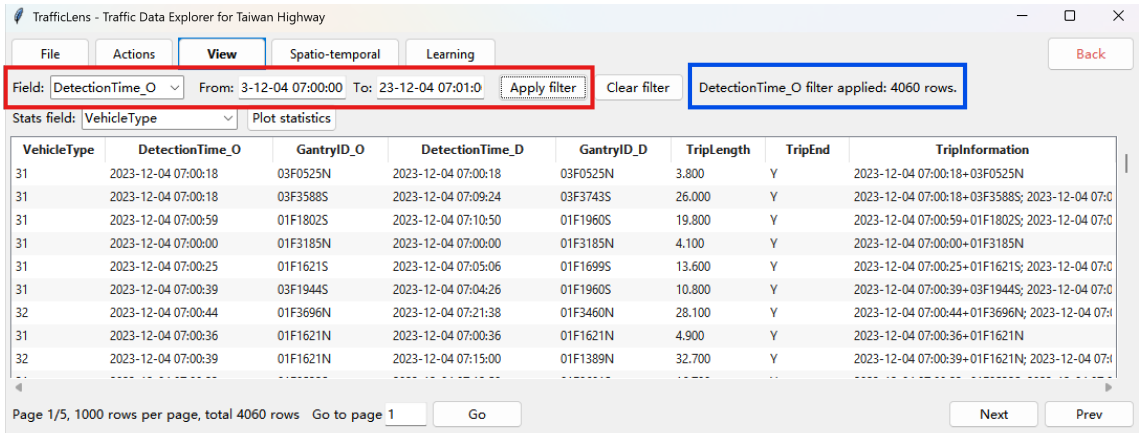


Figure 11: Filter Example

The statistical plotting feature supports five key fields: ”VehicleType”, ”DetectionTime_O”, ”GantryID_O”, ”DetectionTime_D”, ”GantryID_D”, and ”TripLength”. For categorical data like ”VehicleType”, a pie chart visualizes the composition percentages. Time-based ”DetectionTime” fields and numerical ”TripLength” generate frequency histograms to display distribution patterns. For gateway identification fields (”GantryID_O/D”), the system automatically plots the top 30 most frequent sensors using horizontal bar charts, enabling quick identification of high-traffic locations.

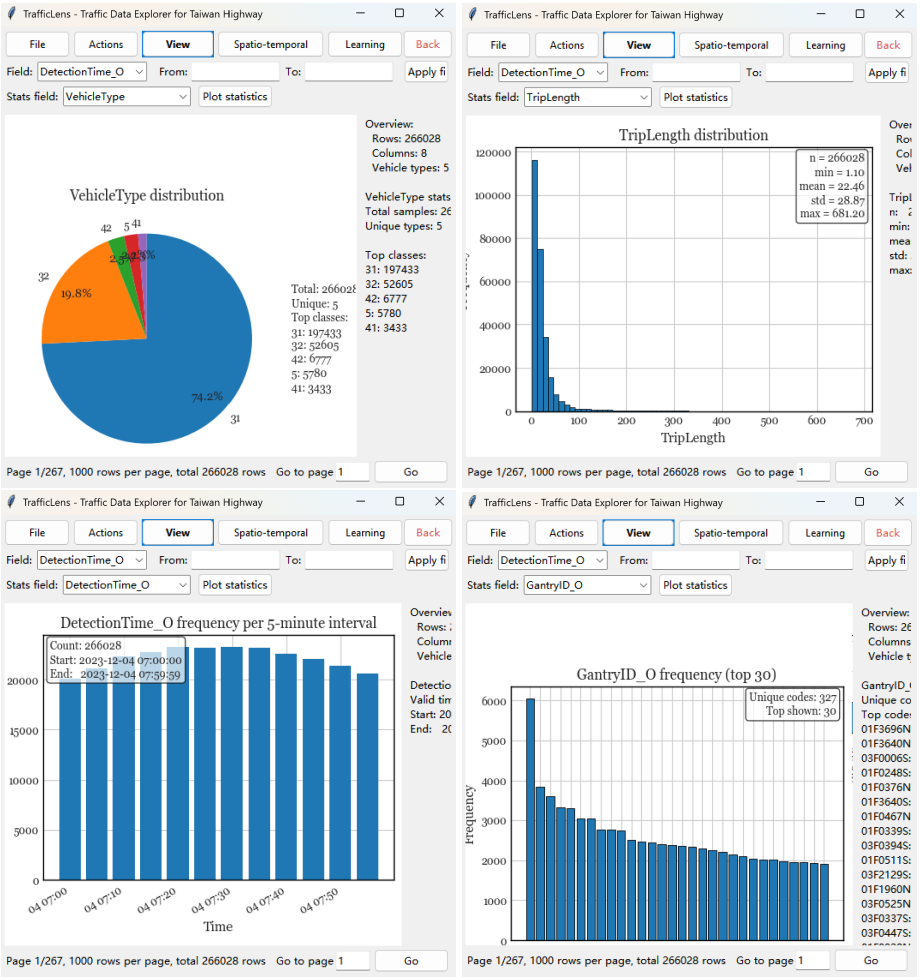


Figure 12: Plot Statistics Function Examples

3.4 *Spatial-Temproral*: Traffic flow statistics

Based on our analysis of the dataset, we observe that Taiwan's highway data represents typical vehicular trajectory records. Building upon this characteristic, we propose a traffic statistics method specifically designed for such trajectory data, with the objective of generating spatiotemporal traffic flow data, also referred to as the spatiotemporal flow matrix.

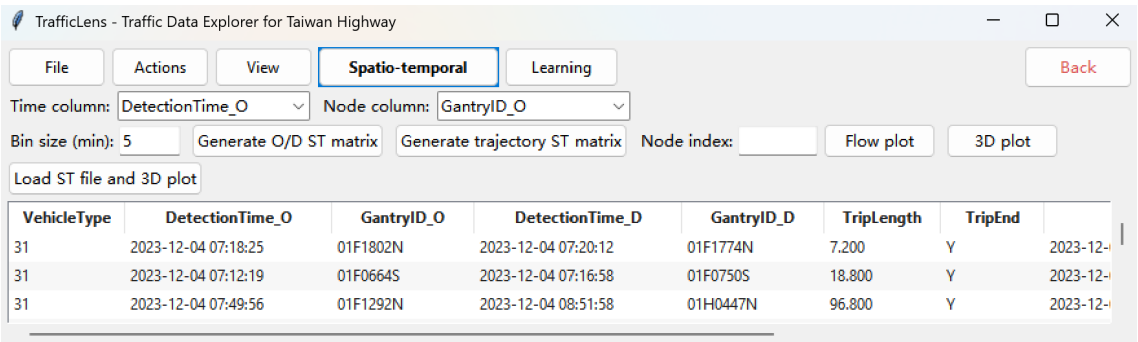
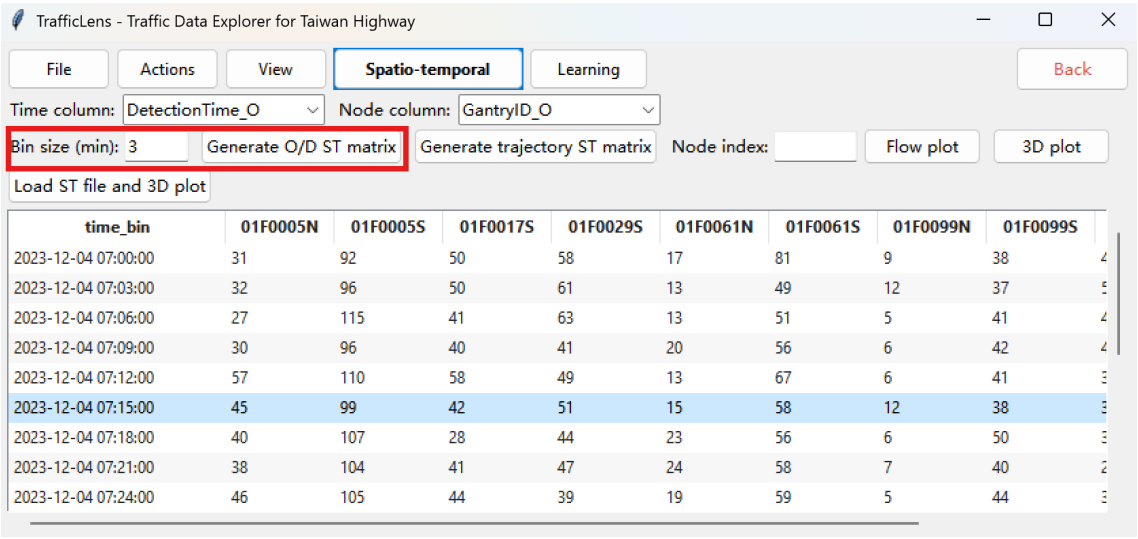


Figure 13: Spatiotemporal statistics interface

Given the timestamp column T and gantry column N, the spatiotemporal (ST) data gener-

ation process is shown in Table 2. Unlike ST matrices built only from origin or destination points, the `TripInformation` field keeps full vehicle trajectories, which gives us richer information and lets us capture more complete spatiotemporal traffic patterns.

For each trajectory with ordered timestamp–gantry pairs (e.g., "$t_1 + n_1; t_2 + n_2; \ldots$"), we extract all events (t_i, n_i) and aggregate them into time windows and gantry locations, as summarized in Table 3. After loading the data, users can generate the O/D-based ST matrix by clicking the red button and can also set their preferred time granularity.



time_bin	01F0005N	01F0005S	01F0017S	01F0029S	01F0061N	01F0061S	01F0099N	01F0099S
2023-12-04 07:00:00	31	92	50	58	17	81	9	38
2023-12-04 07:03:00	32	96	50	61	13	49	12	37
2023-12-04 07:06:00	27	115	41	63	13	51	5	41
2023-12-04 07:09:00	30	96	40	41	20	56	6	42
2023-12-04 07:12:00	57	110	58	49	13	67	6	41
2023-12-04 07:15:00	45	99	42	51	15	58	12	38
2023-12-04 07:18:00	40	107	28	44	23	56	6	50
2023-12-04 07:21:00	38	104	41	47	24	58	7	40
2023-12-04 07:24:00	46	105	44	39	19	59	5	44

Figure 14: Example of generating a spacetime matrix

The system enables users to visualize the processed spatiotemporal data in two key ways. First, it allows for the generation of line charts depicting traffic flow trends for any individual gantry node. Second, it offers a 3D plotting feature that presents a bird's-eye view of how traffic flow evolves over both time and space, providing an intuitive and immediate grasp of the overall traffic situation.

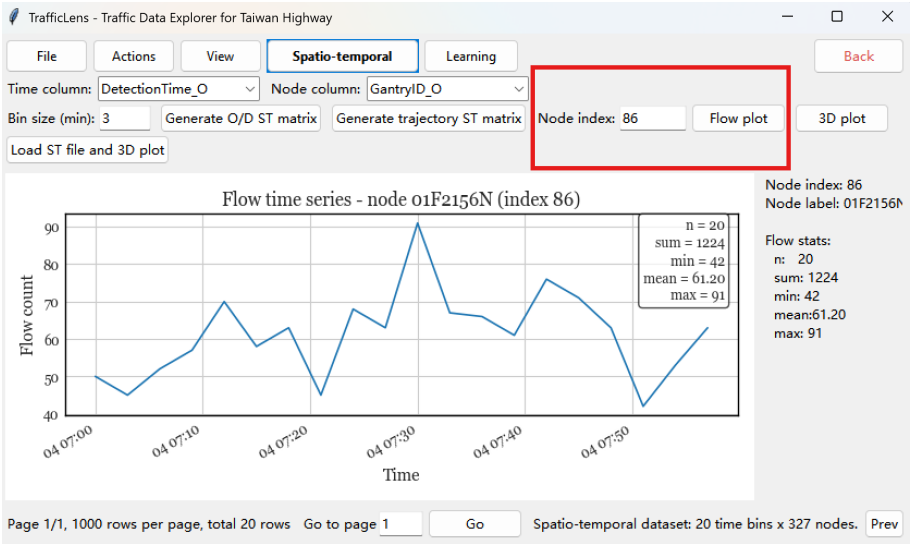


Figure 15: Example of traffic flow trends in node 86

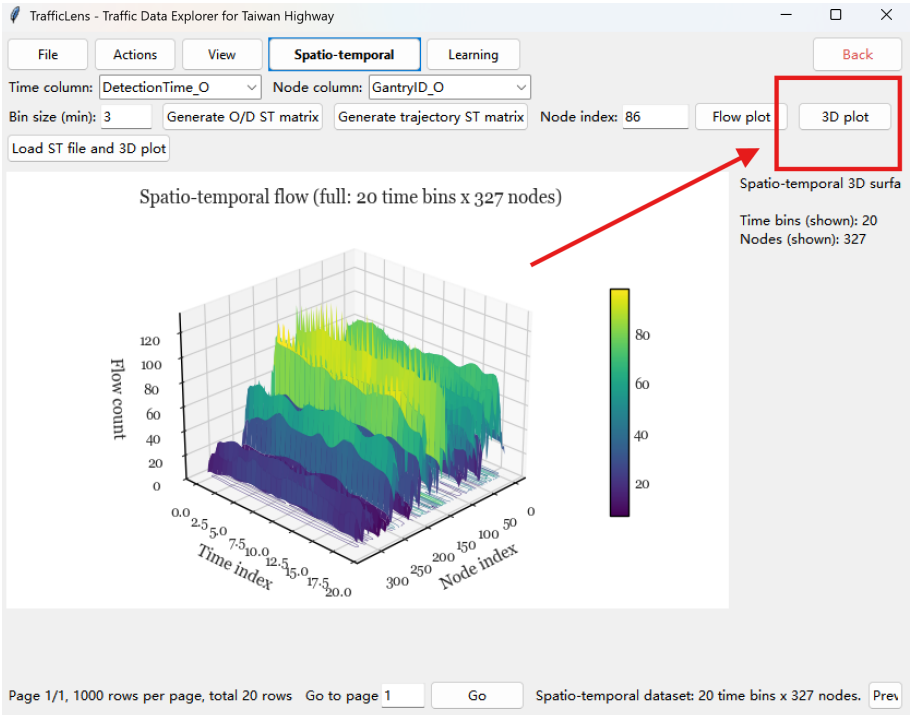


Figure 16: Example of 3D ST Matrix

3.5 *Learning*: Deep learning benchmarks

Building on the spatiotemporal traffic flow data, we also developed built-in deep learning modeling functions. Following a data-driven approach, our goal is to provide an easy-to-use benchmark tool for spatiotemporal traffic prediction, allowing researchers to test the performance of different algorithms on Taiwan's highway network.

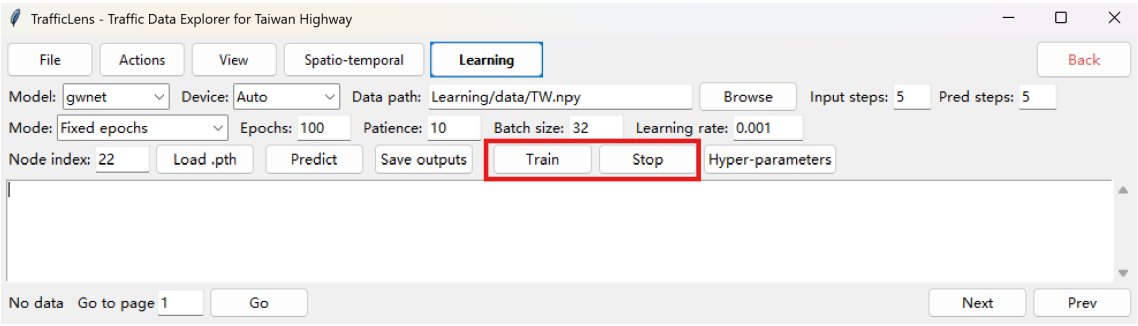


Figure 17: Deep learning interface

The deep learning interface includes basic modeling functions. At the moment, it provides three baseline models: GWNet[2], LSTM, and STGformer[3]. Users can freely choose any of these models for training, select the computing device (GPU or CPU), set the length of the sliding window, and decide whether to enable the early-stopping mechanism. After adjusting these parameters, users can simply click the "Train" button to start the training process.

[2]https://arxiv.org/abs/1906.00121.

[3]https://arxiv.org/abs/2410.00385. Unlike the original implementation, we made a minor improvement here by introducing an adaptive adjacency matrix functionality.

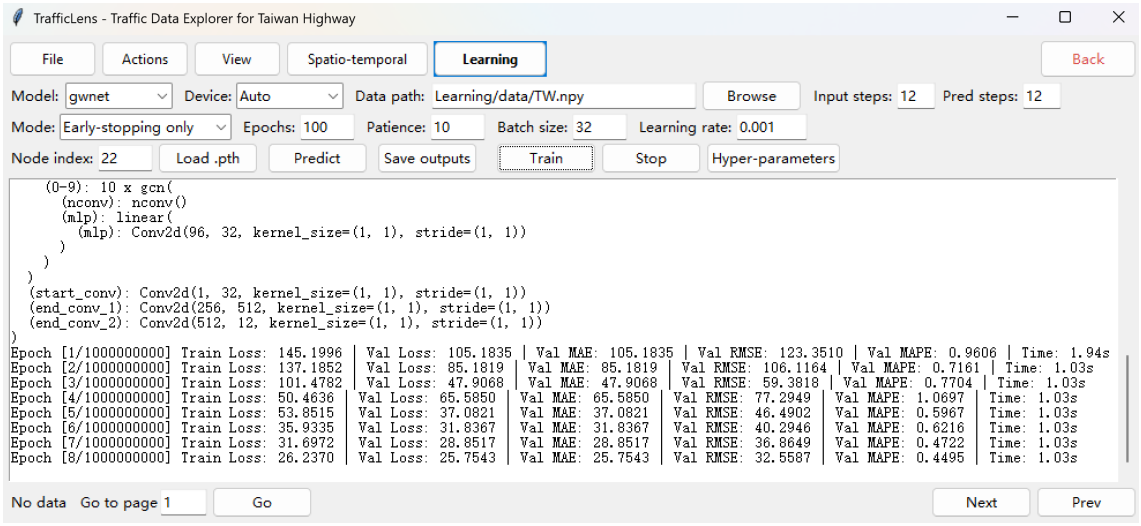


Figure 18: Training Example (GWNet)

We also added visualization functions. By loading the trained `.pth` file, users can visualize the predictions for specific nodes. However, it is important to make sure that the structural parameters, such as the input size and prediction size, are the same as those used during training.

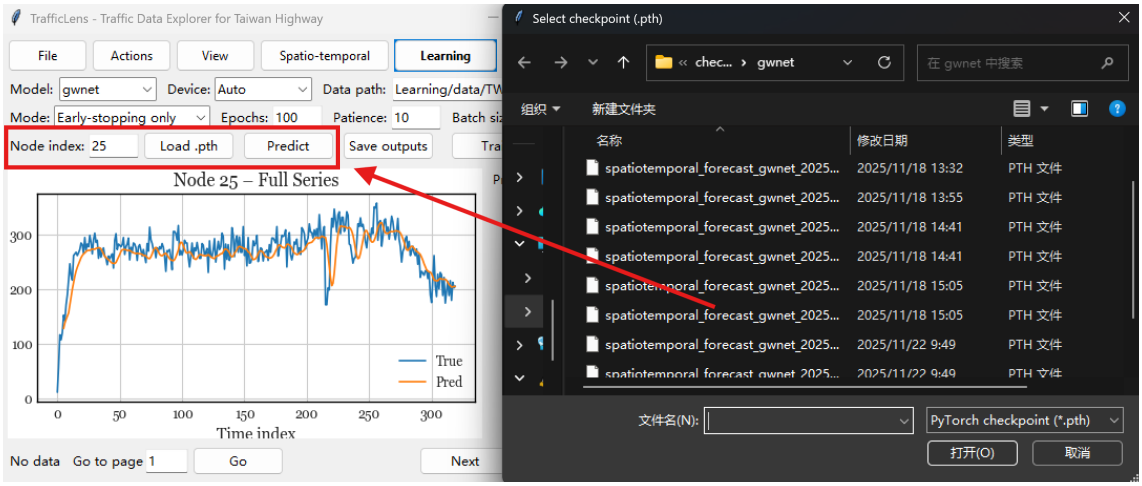


Figure 19: Prediction Example (GWNet) in node 25

Appendix

Spatiotemporal data processing algorithms

We provide detailed explanations here on how we convert the raw data into spatiotemporal data, along with a clear description of how the `TripInformation` field is processed.

Table 2: Spatiotemporal Matrix Generation Algorithm

Algorithm1: Spatiotemporal Matrix Generation

Input: Original event set $\{(t_i, n_i)\}_{i=1}^{M}$ where t_i is timestamp and n_i is gantry ID
Output: ST matrix representing traffic flow distribution

1. Define time window function: $\tau(t) = \lfloor \frac{t}{\Delta} \rfloor \cdot \Delta$ (Δ: fixed time step, e.g., 5 minutes)
2. Initialize empty ST matrix with dimensions $K \times J$
3. **For each** event (t_i, n_i) in the dataset:
   $k \leftarrow \tau(t_i)$ // Map to discrete time window
   $j \leftarrow$ index of n_i // Map to gantry index
   $\mathrm{ST}[k, j] \leftarrow \mathrm{ST}[k, j] + 1$
4. **Return** ST matrix

Table 3: Trajectory-based Spatiotemporal Matrix Generation Algorithm

Algorithm2: Trajectory-based Spatiotemporal Matrix Generation

Input: Trajectory set $\{s_p\}_{p=1}^{P}$, where each s_p is a trajectory string containing ordered points $t_{p,q} + n_{p,q}$ (timestamp + gantry ID)
Output: ST matrix representing trajectory-based traffic flow distribution

1. Initialize empty event set $E \leftarrow \varnothing$
2. **For each** trajectory s_p in the dataset:
   2.1 Split s_p by ";" into segments $\{seg_{p,q}\}$
   2.2 **For each** segment $seg_{p,q}$:
      Parse $seg_{p,q}$ into $(t_{p,q}, n_{p,q})$ by splitting with "+"
      Convert $t_{p,q}$ to timestamp, $n_{p,q}$ to gantry ID
      Add event $(t_{p,q}, n_{p,q})$ into E
3. Define time window function: $\tau(t) = \lfloor \frac{t}{\Delta} \rfloor \cdot \Delta$ (Δ: fixed time step, e.g., 5 minutes)
4. Determine discrete time windows $\{T_k\}_{k=1}^{K}$ and gantry index set $\{N_j\}_{j=1}^{J}$ from E
5. The same as **Spatiotemporal Matrix Generation** before
5. Initialize empty ST matrix with dimensions $K \times J$
6. **For each** event (t_i, n_i) in E:
   $k \leftarrow$ index of $\tau(t_i)$ // Map to discrete time window
   $j \leftarrow$ index of n_i // Map to gantry index
   $\mathrm{ST}[k, j] \leftarrow \mathrm{ST}[k, j] + 1$
7. **Return** ST matrix

Main Functions View

We provides a comprehensive overview of the main functions and components in the system, with detailed explanations of their purposes and functionalities presented in the table below.

Table 4: Main Functions Overview

Name	Type	Description
trafficlens.gui (UI Controller)		
TrafficLensApp	Class	The main Tkinter application class. Coordinates all View updates and Model interactions.
_build_widgets	Func	Initializes the main window layout, tabs, and Treeview tables.
on_import_file	Func	Handler for "File→Import". Opens file dialog and loads CSV into store.
on_search	Func	Filters the displayed data based on user keyword and selected column.
on_sort_button	Func	Handles table sorting (Ascending/Descending) based on the currently selected column.
on_generate_st	Func	Handler for generating O/D-based Spatio-Temporal matrices.
on_learning_train	Func	Spawns a background thread to run the deep learning training loop.
on_learning_predict	Func	Spawns a background thread to run model inference and plot results.
trafficlens.data_loader (Data Access)		
TrafficDataStore	Class	In-memory data manager wrapper around pandas DataFrame.
TrafficDataStore.from_files	Method	Factory method to create a store from multiple CSV paths (uses ThreadPool).
TrafficDataStore.search	Method	Performs keyword search across one or all columns.
TrafficDataStore.sort	Method	Sorts the underlying DataFrame by a specified column (Asc/Desc).
TrafficDataStore.merge_with_files	Method	Returns a new store instance containing merged data from new files.
_read_single_csv	Func	Internal helper to robustly read CSVs (handles missing headers).
trafficlens.stats (Visualization & Analysis)		
overview_stats	Func	Computes basic metrics: total rows, columns, and unique vehicle types.
vehicle_type_pie	Func	Generates a Pie Chart figure for Vehicle-Type distribution.
time_hist_5min	Func	Generates a Bar Chart for event frequency aggregated by 5-minute bins.

Table 4: Main Functions Overview (Continued)

Name	Type	Description
trip_length_hist	Func	Generates a Histogram for the TripLength distribution.
st_3d_surface	Func	Creates a 3D Surface Plot.
flow_timeseries	Func	Creates a 2D Line Plot for a single node's traffic flow over time.
trafficlens.ST_generator (Matrix Transformation)		
generate_spatiotemporal	Func	Converts raw O/D logs into a dense (Time, Node) count matrix.
generate_spatiotemporal_from_tripinfo	Func	Parses complex trajectory strings (Time+Node sequence) into a flow matrix.
trafficlens.view_filters (Query Logic)		
get_field_kind	Func	Returns the semantic type ('time', 'category', 'numeric') of a column.
apply_view_filter	Func	Applies range or equality filters to a DataFrame based on column type.
Learning (Training & Inference)		
run_training	Func	Main training loop. Builds model, data loaders, and executes epochs.
train_one_epoch	Func	Executes one pass of forward/backward propagation.
evaluate	Func	Computes loss metrics (MAE, RMSE, MAPE) on validation/test sets.
run_inference	Func	Loads a saved checkpoint and runs prediction on the dataset.
Learning.model (Neural Networks)		
GWNet	Class	Graph WaveNet implementation. Uses dilated convolutions and adaptive graph adjacency.
LSTM	Class	Standard LSTM-based forecaster with linear output head.
STGformer	Class	Spatio-Temporal Graph Transformer. Uses attention mechanisms for traffic forecasting.
Learning.utils (Helpers)		
build_dataloaders_single_feature	Func	Creates PyTorch DataLoaders from raw .npy files. Handles sliding window generation.
_load_raw_array	Func	Robustly loads .npy or .npz files into a numpy array.
EarlyStopping	Class	Monitors validation loss to stop training early if no improvement occurs.
get_default_config	Func	Returns the default hyperparameter dictionary for all models.

Table 4: Main Functions Overview (Continued)

Name	Type	Description
build_model	Func	Factory function that instantiates a model (GWNet/...) based on config.

Non-blocking GUI and Asynchronous Deep Learning Training

We designed the GUI to run smoothly even when deep learning tasks take a long time. By using a threaded setup, the training or prediction runs in the background, so the interface stays responsive and users can interact without waiting.

- **Background Threads:** We create worker threads with `threading.Thread(target=worker, daemon=True)` to handle training or prediction tasks without freezing the GUI.
- **Callbacks:** Communication between threads is handled via `log_callback` and `stop_callback`. The GUI safely updates itself using `self.after(0, callback)`.
- **Stopping Training:** Users can stop the training anytime using the `self.learning_stop_training` flag, giving them full control.

This approach avoids the usual GUI freezes when computations take a long time. It makes the program more user-friendly while keeping the code simple and easy to maintain.

O/D Time Asymmetry and our Solution

We noticed an issue when generating spatiotemporal matrices based on destination detection times (`DetectionTime_D`). Since vehicles take some time to travel from origin to destination, using the destination times can lead to more time windows than those based on origin times. For example, if the origin windows are $[T_0, T_1, ..., T_n]$, the destination windows might look like $[T_0+\Delta t, T_1+\Delta t, ..., T_{n+1}+\Delta t]$, where Δt is the average travel time. This mismatch makes it tricky to directly compare or combine matrices based on origin and destination times. Also, if we use the destination windows directly, we might include vehicles that fall outside the origin window range, which could mess up the analysis.

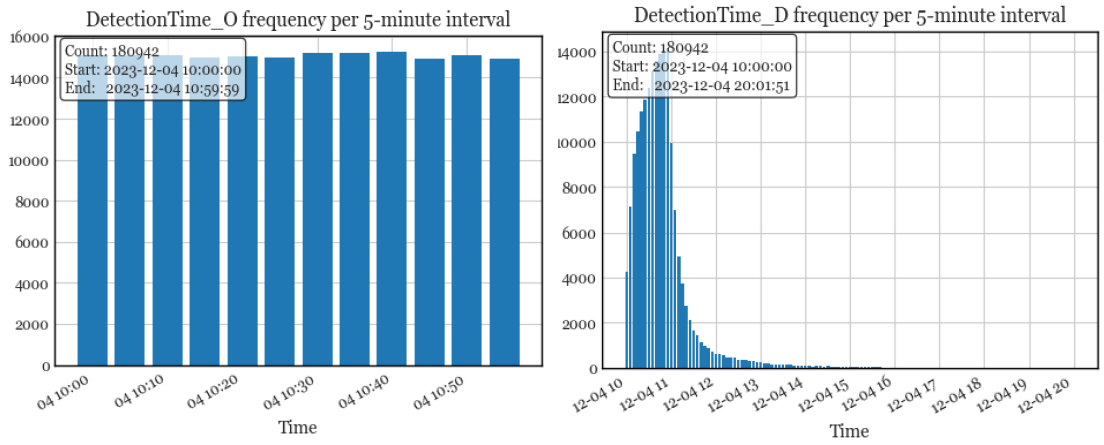


Figure 20: O/D asymmetry in the same dataset

To fix this, we came up with an automatic truncation method using the origin windows as a reference. Basically, we cut the destination-based matrix so it matches the number of origin windows. In practice, we first discretize the origin time column (`DetectionTime_O`) to get the unique windows and count. If the destination matrix has more rows, we just slice it using `df_grouped.iloc[:n_o]`. This way, the timeline stays consistent whether we use origin, destination, or full trajectory times, which makes comparisons and model training much easier.

```
if time_col == "DetectionTime_D" and "DetectionTime_O" in df.
    columns:
     time_o = pd.to_datetime(df["DetectionTime_O"], errors="
         coerce")
     mask_o = time_o.notna()
     if mask_o.any():
         bins_o = time_o.loc[mask_o].dt.floor(freq)
         unique_o = pd.Index(sorted(bins_o.unique()))
         n_o = len(unique_o)
         if n_o and len(df_grouped) > n_o:
             df_grouped = df_grouped.iloc[:n_o]
```

Adaptive Parallel Loading Strategy

We noticed that reading many CSV files serially is slow, but too many threads can also be wasteful. So we implemented an adaptive strategy: for fewer than 8 files, read them one by one; for 8 or more, use up to 8 threads in parallel. Files are processed as they finish to improve responsiveness.

```
if len(paths) < 8:
    # Serial reading
    for p in paths:
        frames.append(_read_single_csv(p))
else:
    # Parallel reading
    max_workers = min(8, len(paths))
    with ThreadPoolExecutor(max_workers=max_workers) as executor
        :
        future_map = {executor.submit(_read_single_csv, p): p
            for p in paths}
        for fut in as_completed(future_map):
            try:
                frames.append(fut.result())
            except Exception:
                continue
```

All CSVs are read as strings to avoid type inference overhead:

```
df = pd.read_csv(path, header=None, dtype=str)
```

Headers are detected automatically and skipped if needed, then all frames are concatenated in one batch for efficiency:

```
combined = pd.concat(frames, ignore_index=True)
```

This approach keeps the timeline and memory consistent, handles exceptions gracefully, and significantly speeds up multi-file merging, especially for large batches.

Advanced usage

For users who want more control over modeling, we provide an external interface to set the neural network's structure parameters. By default, all parameters use preset values. If a user changes them, the new settings will override the defaults, but once the system restarts, the parameters return to their original default values. Different models have different sets of adjustable parameters; for details, please refer to the original text or source code.

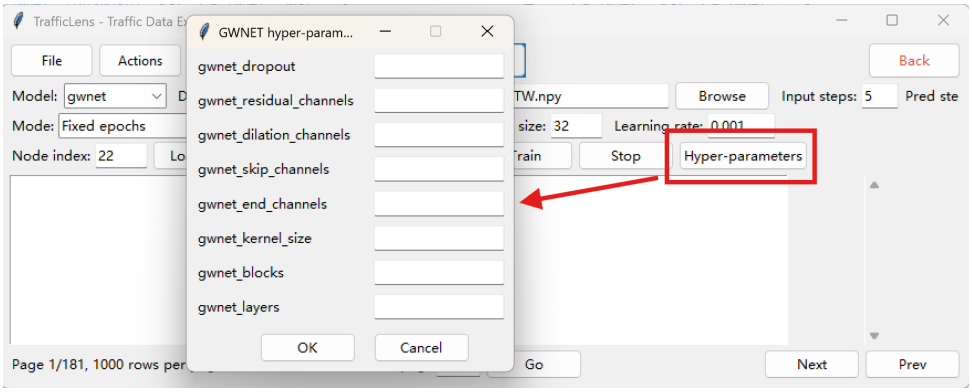


Figure 21: Model parameter adjustment Example (GWNet)

In addition, with the **Spatial-Temporal** feature, users can read the spatiotemporal matrix in `.npy` format generated by the prediction and plot it in 3D:

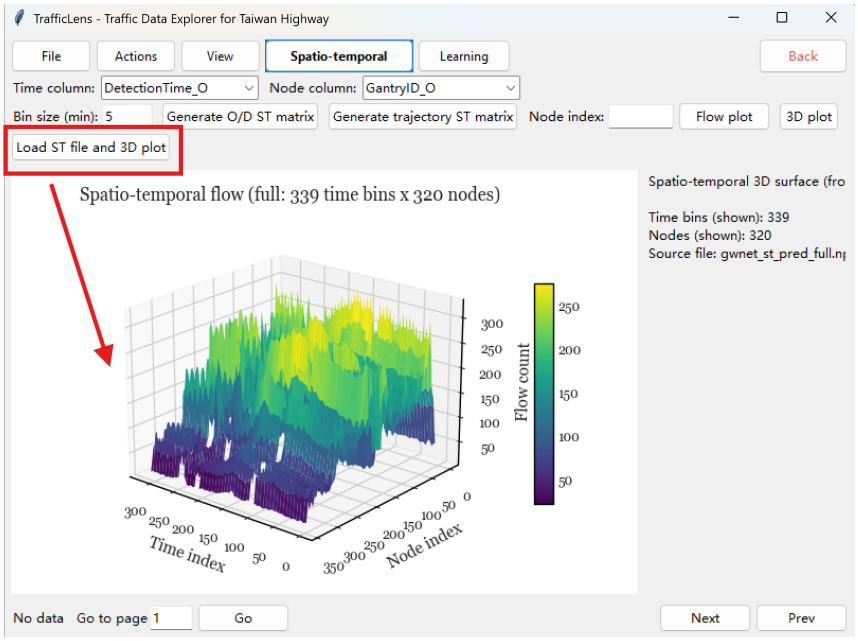


Figure 22: Predicting 3D plots Example