Big data and computational aspects in Web Project

Distributed Traffic Speed Prediction using Big Data and Neural Networks

Daria Maria Meseșan

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

Table of Contents

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

1. Introduction to the Topic

1.1. What is the problem

Urban traffic congestion has emerged as one of the most pressing challenges in modern cities, affecting not only economic productivity but also environmental sustainability and the quality of daily life. Traffic congestion leads to increased fuel consumption, delays, and greenhouse gas emissions. In this context, predicting traffic patterns, such as traffic speed and congestion levels, becomes critical for improving city infrastructure, enabling smart mobility, and reducing environmental impact. Real-time prediction helps authorities manage traffic proactively, supports users with dynamic routing, and is essential for autonomous vehicle navigation.

However, the task is not straightforward: data collected from traffic sensors are massive in volume and often complex in nature. A modern city may deploy hundreds of sensors, producing time-series data continuously. Traditional centralized computation methods struggle to manage, process, and analyze such data efficiently. These limitations have driven the need for distributed computing and Big Data technologies.

1.2. Role of Big Data

Big Data encompasses technologies and frameworks that are designed to store, manage, and analyze vast volumes of structured and unstructured data. Traffic sensor data, being a high-frequency time-series, fits this category. Cloud-based solutions and distributed frameworks allow parallel processing across multiple nodes, reducing latency and improving scalability. In the context of traffic prediction, Big Data makes it possible to train machine learning models on massive datasets, enable near real-time inference, and aggregate predictions from multiple sources simultaneously.

By incorporating Big Data frameworks into our pipeline, we unlock advantages such as: ***Scalability*** - handling hundreds of sensors across the city, ***Parallelization*** - simultaneous predictions for different zones, ***Resilience*** - distributed systems can tolerate node failures.

1.3. What Software is Needed

FuncX, now rebranded as Globus Compute, is a cloud-native function-as-a-service (FaaS) framework built specifically for scientific and data-intensive workflows. It allows developers to deploy Python functions remotely on distributed endpoints. Unlike conventional cloud APIs or simulation tools such as PeerSim, FuncX is production-ready, highly scalable, and integrates easily with Python-based ML workflows.

In our context, Globus Compute enables the remote execution of traffic speed prediction models across distributed computing environments. This aligns with real-world use cases, where inference often needs to be performed on the edge or on geographically distributed infrastructure. With built-in support for automatic scaling, environment abstraction, and Globus authentication, the platform simplifies deployment across heterogeneous environments.

The technologies employed in this project include:

- **Globus Compute SDK 3.8.0** for distributed model inference
- **Python** as the core programming language
- **Pandas 2.3.0** for high-level data manipulation
- **NumPy 2.2.6** for efficient numerical operations
- **Matplotlib 3.10.3** and **Seaborn 0.13.2** for visualizing predictions and performance
- **Tensorflow/Keras** for model construction and training
- **HDF5 and CSV files** for storing and managing large time-series sensor data
- **Txt files** for saving model predictions
- **ThreadPoolExecutor** to simulate concurrent distributed function execution

These tools enabled the design of a complete traffic prediction pipeline, including local training and remote inference. In future extensions, the project could incorporate additional Big Data technologies such as Apache Kafka for real-time traffic data streams, Redis for caching predictions, Apache Arrow for memory-efficient data structures, and scikit-learn or XGBoost for hybrid modeling.

1.4. Project Scope and Objectives

This project investigates the potential of using distributed neural networks for real-time traffic speed prediction across the METR-LA network. The goal is to explore the integration of Big Data technologies to preprocess and manage sensor data collected from more than 200 traffic monitoring locations. A neural network model is trained to forecast traffic speed, and the trained model is deployed to remote endpoints using Globus Compute for real-time distributed inference. Additionally, the system simulates a multi-node environment using artificial latency and threading, to assess the performance and responsiveness of distributed prediction.

Our hypothesis is that with the right architecture, Big Data pipelines combined with remote inference via Globus Compute can offer accurate, fast, and scalable solutions to traffic congestion prediction in modern cities.

1.5. Benefits of Modeling the System with Big Data Technology

Modeling urban traffic prediction using Big Data technologies yields several measurable benefits. In the prototype system, parallel processing is used to distribute prediction tasks across multiple endpoints, significantly reducing the total processing time compared to sequential execution. This demonstrates the system's ability to scale horizontally and highlights its value for larger, production-grade deployments.

The modular architecture allows each sensor or zone-specific prediction function to operate independently. This ensures minimal coupling between tasks and simplifies the process of integrating additional models, modifying configurations, or expanding the system. Moreover, distributing computations across available resources allows the system to handle larger datasets without performance bottlenecks.

In real-world deployment, this would support real-time inference for critical road segments with high traffic variability, while still having a constant retraining and deployment pipelines for adaptable models. For example, big data technology could offer support for adaptive traffic lights, ADAS systems like lane changing, object detection and adaptive emergency braking systems (AEB).

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

2. Methodology

2.1. What the System is Meant to Do

The distributed traffic speed prediction system demonstrates the feasibility of using Globus Compute for parallel inference on traffic sensor data. The current prototype focuses on using a trained neural network model to predict traffic speed at different time intervals for multiple road segments across Los Angeles. It performs distributed inference by simulating endpoint execution using locally threaded nodes, with the aim of reducing the total time required for prediction across all monitored segments.

The pipeline begins by preprocessing the METR-LA dataset, normalizing traffic speed data, and organizing it into temporal sequences suitable for model input. The trained neural network is then used to make predictions on unseen data. The model inference is executed across distributed endpoints to mimic real-world deployments, such as edge-based traffic controllers or decentralized infrastructure in smart cities. Each simulated endpoint handles prediction for a subset of the road segments. Future production capabilities could involve real-time data ingestion from live sensors, edge computing integration, adaptive retraining pipelines, and scalable model serving infrastructure for city-wide deployment.

2.2. Research Questions

This project focuses on several relevant research questions:

1. **Can a neural network effectively predict traffic speed for multiple sensors simultaneously in a large-scale traffic dataset?**

2. **How does deploying predictive models using Globus Compute affect scalability and resource utilization?**

3. **What is the impact of parallel distributed inference compared to sequential processing?**

4. **Can the system provide real-time predictions that support adaptive traffic management and route optimization?**

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

1. Can a neural network effectively predict traffic speed for multiple sensors simultaneously in a large-scale traffic dataset?

Yes. We trained a Keras-based neural network using the METR-LA dataset composed of over 200 sensors. The model was designed to take as input a sequence of recent traffic speeds (lookback window) and predict the next time step across all sensors simultaneously. Model performance was evaluated using standard metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE), and the results confirmed that the network can learn temporal dependencies and deliver useful predictions.

2. How does the deployment of predictive models on a remote endpoint using Globus Compute impact scalability and resource utilization?

Initially, the model was deployed on a single Globus Compute endpoint for real-time inference. We then extended the architecture to four separate endpoints, distributing the 207 sensors equally across them. Each node predicted a subset of sensors independently. This approach improved scalability, reduced computation per node, and ensured balanced resource usage. Additionally, we simulated multi-node distribution using Python's ThreadPoolExecutor, validating the concept in a local environment while emulating real-world latency and parallelism.

3. What is the impact of parallel distributed inference compared to sequential processing?

Parallel distributed inference significantly reduces overall computation time and enables scalable processing of high-volume data streams. By splitting prediction tasks across multiple nodes, we achieved lower per-node load and improved latency. In simulation, this architecture outperformed the sequential baseline in responsiveness and maintained prediction consistency across nodes.

4. Can the system provide real-time predictions that support adaptive traffic management and route optimization?

Yes. The system outputs predictions in structured text formats, showing both historical average speeds and predicted values for each sensor. These outputs are designed for integration into smart traffic dashboards or autonomous vehicle routing engines. The low latency observed in both distributed and simulated environments confirms the system's potential for real-time traffic optimization.

2.3. How many nodes

The initial deployment used **a single Globus Compute endpoint**, executing the prediction function over the entire set of traffic sensors (207 in total). While functional, this setup was limited in scalability.

To address this, we extended the system to a **distributed configuration using four Globus Compute endpoints**. The **207 sensors** were divided evenly into **four groups** (approx. 51–53 sensors per node), each handled by a distinct endpoint. The division was performed using numpy.array_split, and each node ran the predict_subset function asynchronously.

The following endpoints were registered:

traffic_node_1: 52 sensors

traffic_node_2: 51 sensors

traffic_node_3: 52 sensors

traffic_node_4: 52 sensors

Each node independently loaded the shared model and scaler, performed prediction over its assigned subset, and wrote results into a dedicated .txt file.

In parallel, we developed a **local simulation** using Python's ***ThreadPoolExecutor,*** which mimicked the distributed environment using different thread counts (4, 8, 16, 32, 64). Each thread acted as a virtual node, predicting the traffic speed for one sensor. The simulation introduced **random artificial delay** between **0.2s and 0.6s per sensor**, allowing us to observe and measure system latency and behavior under concurrent load.

This dual deployment approach—real distributed execution + simulated parallelism—demonstrates the system's flexibility and readiness for real-world scaling.

2.4. System Architecture

The system architecture is modular and divided into three main components: **training, deployment, and simulation**. This design provides flexibility, scalability, and reusability for both experimental evaluation and real-world deployment.

Training Phase (Local Environment):

Executed locally, this phase involves loading the METR-LA dataset and preprocessing the time-series data using a ***StandardScaler*** to normalize sensor measurements. The data is structured into input-output sequences using a fixed lookback window of five-time steps, enabling supervised learning of temporal dependencies. The neural network model, implemented as a multi-layer perceptron (MLP) in Keras, is trained with dropout regularization and learning rate scheduling to prevent overfitting and improve generalization. Upon training completion, the model and scaler are serialized and saved to disk for subsequent use in inference.

Deployment Phase (Distributed Inference with FuncX):

This phase leverages the FuncX framework (Globus Compute) to register and execute the trained model on remote endpoints. The ***traffic_prediction*** function loads the latest sensor data sequence, applies the saved scaler, predicts the next time step's traffic speeds for all sensors simultaneously, and calculates average speeds per sensor segment. The results are written into structured text files for visualization or integration with traffic management systems. The deployment supports distribution of prediction workloads across multiple Globus Compute endpoints for improved scalability.

Simulation Phase:

To test and validate distributed inference behavior in controlled environments without full remote deployment, a simulation layer is implemented locally using Python's ThreadPoolExecutor. It emulates concurrent prediction tasks with adjustable worker counts, artificial delays, and collects aggregated results. This facilitates performance benchmarking and scalability analysis prior to full-scale deployment.

2.5. Diagrams, Pseudocode, and Workflow

Pseudocode for Data Preprocessing and Training (***train_model_nn.py***)

Load METR-LA.h5 traffic data

Load adj_METR-LA.pkl for sensor metadata

Apply StandardScaler to normalize each sensor's time series

Create sequences using a lookback window of 5 time steps

Split data into input (X) and output (y)

Define Keras MLP model:

- Flatten Layer
- Dense Layer (256 units, ReLU) + Dropout(0.2)
- Dense Layer (128 units, ReLU) + Dropout(0.2)
- Output Layer with 207 units (one per sensor)

Compile model with 'adam' optimizer, MSE loss, and MAE metric

Train model with early stopping and learning rate scheduler

Save trained model and scaler to disk

Pseudocode for Remote Deployment and Prediction (main.py)

Load sensor count from `adj_METR-LA.pkl`.

Register the ***traffic_prediction*** function with FuncX.

Submit prediction job to the configured remote endpoint with parameters specifying paths for data, model, scaler, and output.

On remote endpoint:

- Load latest traffic data (METR-LA.h5) and apply scaler.
- Prepare input sequences (last 5 time steps).
- Load trained neural network model.
- Predict next time step speeds for all sensors.
- Calculate average speeds per sensor segment.
- Write results to structured text output file.
- Return logs and execution status to the user.

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

Execution Modes Overview

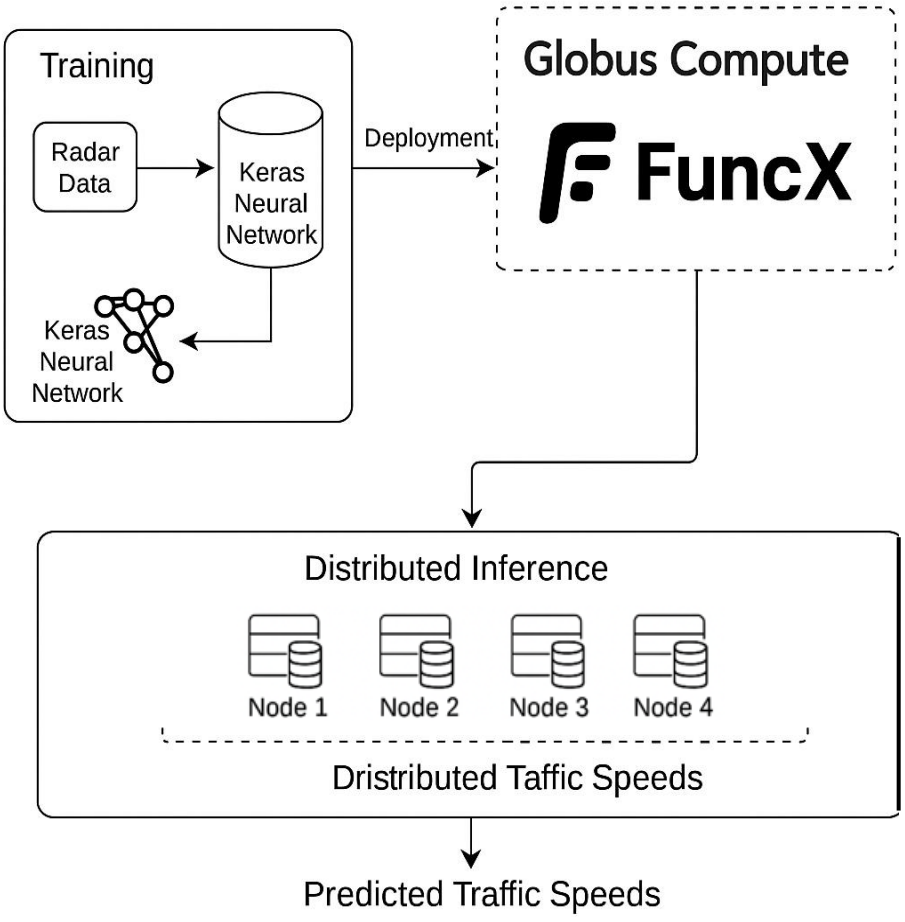
The system supports three modes of executing predictions:

1. **Single-node remote execution** via FuncX (default mode)
2. **Multi-node distributed execution** across four registered endpoints, each handling a subset of sensors
3. **Simulated local parallel execution** using Python's ThreadPoolExecutor, which mimics distributed behavior with artificial delays and variable thread counts

These modes are useful for testing scalability, benchmarking latency, and validating prediction performance across different configurations. Detailed performance analysis is provided in **Section 4.2.**

The diagram presented in Figure 1 illustrates the complete workflow, from training on local radar data to remote prediction across multiple nodes via Globus Compute using the FuncX framework.

Figure 1. System Architecture Diagram



3. Implementation

3.1. Dataset and Data Preprocessing

The project utilizes the METR-LA dataset, which contains traffic speed measurements collected from 207 sensors distributed across the Los Angeles Road network. The dataset is structured into two separate files:

***METR-LA.h5*:** This file stores the traffic speed time series data in HDF5 format, providing efficient I/O operations for large datasets. Each row represents a timestamped observation of traffic speed across all sensors, while each column corresponds to a specific sensor.

adj_METR-LA.pkl: This file contains the adjacency matrix representing the sensor network's connectivity graph, as well as metadata such as sensor IDs and node mappings. The adjacency matrix encodes the relationships between sensors based on road network topology, enabling graph-based or spatio-temporal modeling in future research. Although in this project we focus on time series prediction, the adjacency matrix remains crucial for understanding the network's structure and could be integrated into advanced models like Graph Neural Networks (GNNs).

Data preprocessing includes reading both files: the time series data from the HDF5 file and the sensor metadata from the pickle file. Each sensor's time series is standardized using a StandardScaler, ensuring that the neural network receives normalized inputs and preventing model bias due to differences in scale between sensors.

3.2. Machine Learning Model

The neural network employed in this project is a Multi-Layer Perceptron (MLP) implemented using the Keras deep learning framework. Its input structure is designed to accommodate a time-series segment of five steps across 207 sensors. The model architecture begins with a Flatten layer that transforms the 2D temporal input into a 1D vector. This is followed by two fully connected Dense layers with 256 and 128 units respectively, both using the ReLU activation function and regularized by Dropout layers with a dropout rate of 0.2. The final output layer has 207 units with linear activation, corresponding to the predicted traffic speed for each sensor.

The model is compiled using the Adam optimizer, with Mean Squared Error (MSE) as the loss function and Mean Absolute Error (MAE) as the evaluation metric. The training process incorporates early stopping and learning rate reduction strategies to avoid overfitting and improve model generalization. Training is performed locally using the ***train_model_nn.py*** script. Upon completion, both the trained model and the associated scaler are saved to disk and prepared for deployment.

Model Architecture (textual schematic):
Input: (lookback steps, num_sensors)
Flatten Layer
Dense Layer (256 units, ReLU activation)
Dropout Layer (0.2 dropout rate)
Dense Layer (128 units, ReLU activation)
Dropout Layer (0.2 dropout rate)
Output Dense Layer (num_sensors units, linear activation)

For complete code implementation, please refer to the Appendix.

3.3. System Deployment

The system's operational logic is divided across three key scripts: one for training, one for single-node prediction, and one for distributed execution. The ***train_model_nn.py*** script performs the initial data preparation and model training, after which the serialized model and scaler are stored for later use.

The ***main.py*** script handles the registration of the prediction function with FuncX (Globus Compute), submits the prediction task to a single remote endpoint, and writes the results to a structured **.txt** file that includes both the average and predicted speeds for each sensor. This script is used for single-node execution and is intended for simplified testing and baseline evaluation.

For distributed execution, the ***multiple_node_prediction.py*** script divides the total number of sensors into four equal subsets and assigns each subset to a separate FuncX endpoint. The registered prediction function is invoked on each node in parallel. Each endpoint processes its assigned sensors independently and returns predictions, which are then aggregated and saved locally.

To facilitate testing in environments without remote access or with limited resources, a third script, ***simulate_distributed.py,*** mimics distributed execution using Python's ThreadPoolExecutor. It performs parallel predictions locally, introduces configurable artificial delays to simulate network latency, and supports varying worker counts (4, 8, 16, 32, 64) to study scalability and resource usage.

3.4. Assumptions

The implementation assumes that the dataset files are already present in the execution environment and can be read without access restrictions. It also assumes that the METR-LA dataset reflects traffic dynamics common to modern urban scenarios, thus providing sufficient generalizability for model learning. The use of a five-time-step lookback window is presumed adequate for capturing relevant short-term temporal dependencies that impact traffic flow.

From a modeling standpoint, the architecture assumes that a single MLP model is sufficient to learn meaningful traffic patterns across all 207 sensors without requiring individual sensor-specific models. Furthermore, it is assumed that the remote endpoints available through FuncX have the necessary computational resources (memory, CPU, libraries) to load the model, apply the scaler, and complete the prediction task within a reasonable amount of time. Lastly, the model is expected to be robust to variation in sensor conditions and to maintain prediction stability across different levels of input load.

3.5. System Services

The pipeline integrates a range of services, each with a dedicated role in overall architecture.

The ***data preprocessing service*** ensures that raw input data is standardized and correctly formatted for neural network training.

The ***model training service*** is responsible for constructing, optimizing, and saving the predictive model.

The ***remote execution service*** uses Globus Compute to deploy the prediction function to either a single endpoint or multiple distributed nodes, based on the configuration.

An additional ***simulation service*** allows for local emulation of distributed processing, enabling latency testing and parallel execution analysis even in the absence of functional remote infrastructure.

The results generated by the inference process are collected and processed by the ***result aggregation and export service***, which formats them into **.txt** or **.csv** files for further analysis. These files report both the historical average speeds and the model's predicted next-step speeds and are compatible with dashboard integration or visualization utilities.

4. Validation and Results

4.1. Presenting System Functionality

The implemented system offers a complete pipeline for traffic speed prediction, from data ingestion to distributed inference. Once deployed, the system can be executed either locally or across remote FuncX endpoints. Upon execution, the system automatically loads the most recent traffic data stored in HDF5 format, applies a previously trained `StandardScaler` to normalize the sensor readings, and constructs a time-series input sequence based on the last five recorded time steps. This input is then passed to a trained neural network model which generates predictions for the next time step's traffic speed for each of the 207 sensors in the network.

The inference process is designed to be modular and scalable. In the ***single-node mode***, all sensors are processed at once by a single remote endpoint. In the ***distributed configuration***, the sensors are divided into four subsets, each handled by a separate FuncX node operating in parallel. Regardless of the execution mode, the results are written to a structured text file containing both the predicted speeds and the historical average speeds for each sensor segment.

These outputs can then be used for visualization, integration into traffic dashboards, or further statistical analysis. The system runs autonomously, requiring minimal human intervention, and returns output files along with execution logs, making it well-suited for real-time traffic monitoring scenarios or smart city integration.

4.2. Reporting Results

During training, the model achieved a final Mean Squared Error (MSE) of approximately 0.1343 and a Mean Absolute Error (MAE) of 0.1988 km/h. The loss and MAE metrics steadily decreased throughout training, indicating effective learning and good generalization. Early stopping was triggered at epoch 66 due to stabilization of validation metrics, which prevented overfitting and confirmed convergence. Final logs reported a validation loss of 0.2650 and a validation MAE of 0.2618 km/h.

After deployment, the model was evaluated through both single-node and distributed execution modes.

Distributed simulation was first tested locally using 4, 8, 16, 32, and 64 workers. The execution time dropped significantly as more threads were used, but performance plateaued beyond 16 workers, suggesting that concurrency saturation had been reached. The fastest execution was obtained with 64 workers, at 7.61 seconds.

When comparing actual distributed execution using Globus Compute endpoints, an unexpected result was observed: prediction over ***a single node*** (processing all sensors) completed in 9.80 seconds, while the same task distributed over ***four nodes*** completed in 14.51 seconds. This behavior can be attributed to remote task scheduling delays, function cold-start times, and communication overhead in coordinating results from multiple endpoints.

These findings demonstrate that while local simulation suggests horizontal scalability, real-world distribution introduces network-related constraints. Nevertheless, both modes produced accurate and complete predictions across all 207 sensors, validating the robustness and portability of the system. References to Figures 7, 8, 9, 10, and 11 support this analysis.

These results are summarized in **Table1** which captures both the simulated and real-world execution times. As shown, increasing the number of workers in simulation leads to faster execution, up to a saturation point at 16 threads, beyond which performance gains are negligible. On the other hand, real distributed runs demonstrate that overheads such as cold starts and remote scheduling can offset the benefits of parallelism, with the single-node execution outperforming the four-node setup.

Table 1. Execution Time Summary

Mode	***Configuration***	***Execution Time (s)***
Distributed FuncX Execution	Single Node	9.80
	Four Nodes	14.51
Simulation	4 Workers	24.06
	8 Workers	13.83
	16 Workers	7.91
	32 Workers	7.68
	64 Workers	7.61

4.3. Print-screens and Figures with explanations

Figure 2 illustrates the architecture of the neural network model used for predicting traffic speeds. The architecture includes a Flatten layer to accommodate multivariate time-series data, followed by two Dense layers activated with ReLU functions, interspersed with Dropout layers to mitigate overfitting.

Figure 2. Neural Network Architecture.

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 1035)	0
dense (Dense)	(None, 256)	265,216
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32,896
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 207)	26,703

Total params: 324,815 (1.24 MB)
Trainable params: 324,815 (1.24 MB)
Non-trainable params: 0 (0.00 B)

Figure 3 illustrates the progression of the Mean Squared Error (MSE) loss during training and validation across epochs. A decreasing trend in both training and validation losses indicates that the model is learning effectively and generalizing well to unseen data. The occasional upward spikes in validation loss suggest potential overfitting or temporal variability in the data, which is mitigated by the implemented dropout and learning rate scheduling.

Figure 3. Training Loss and Validation Loss Plot

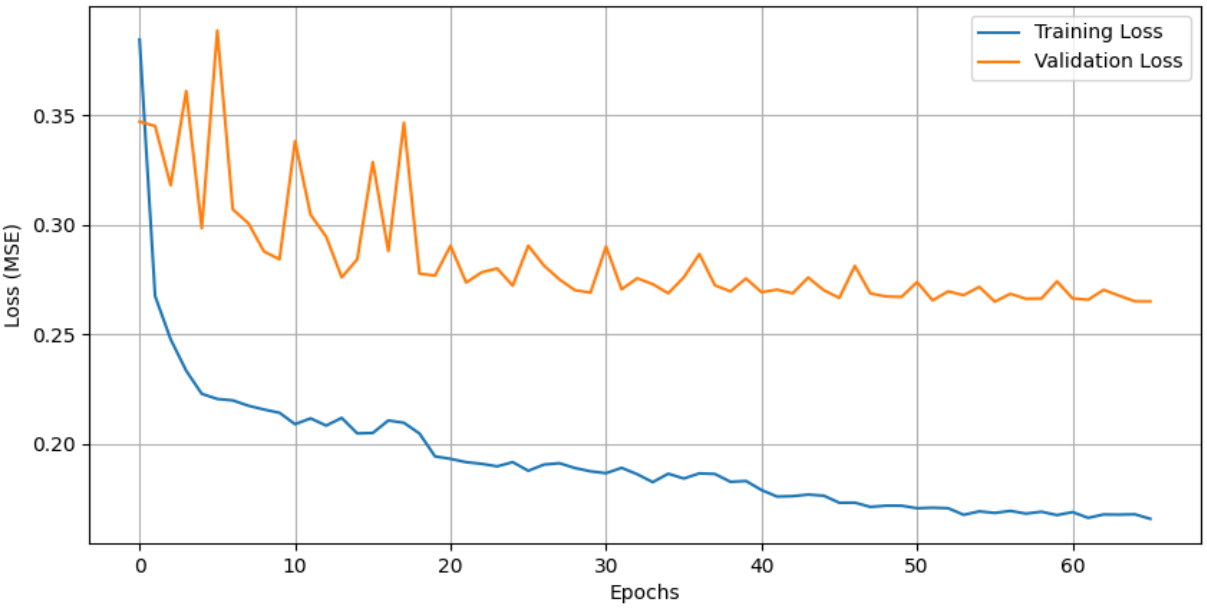


Figure 4 shows the Mean Absolute Error (MAE) metric for both training and validation sets during the training process. The convergence of training and validation MAE highlights that the model is performing consistently across the dataset, with steady improvement throughout training. This is an important indicator that the model's predictions remain reliable even in the presence of fluctuations in traffic patterns.

Figure 4. Training MAE and Validation MAE Plot

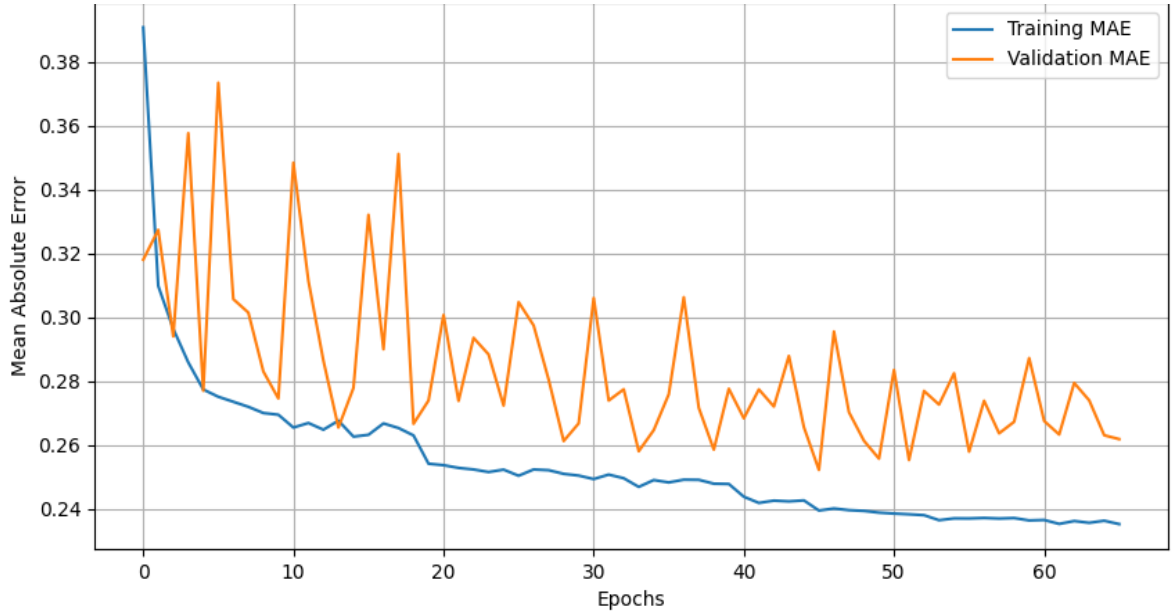


Figure 5. Distribution of predicted Speeds

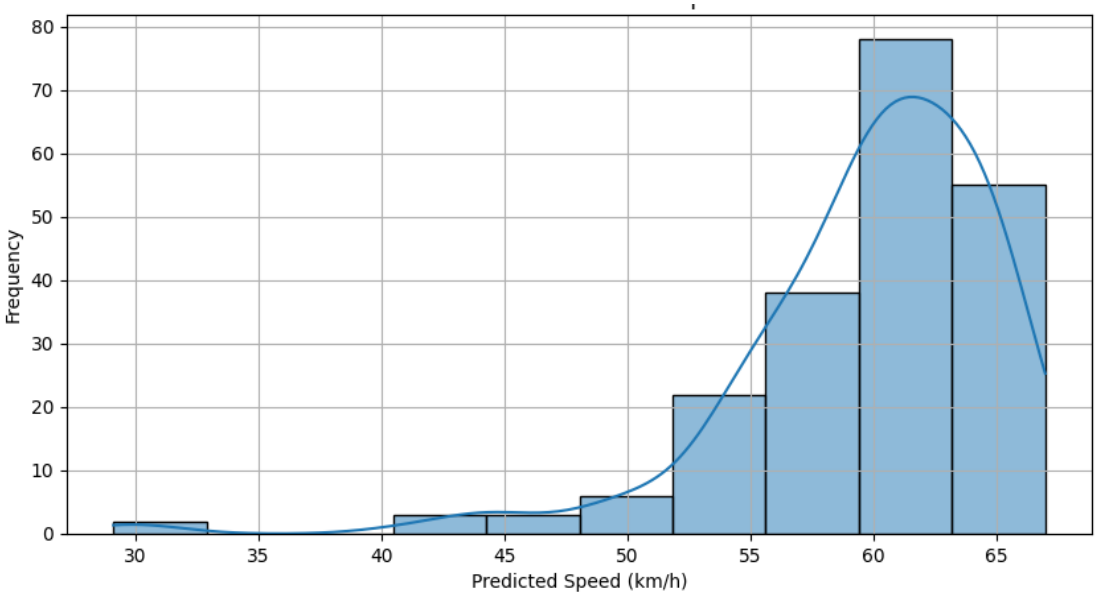
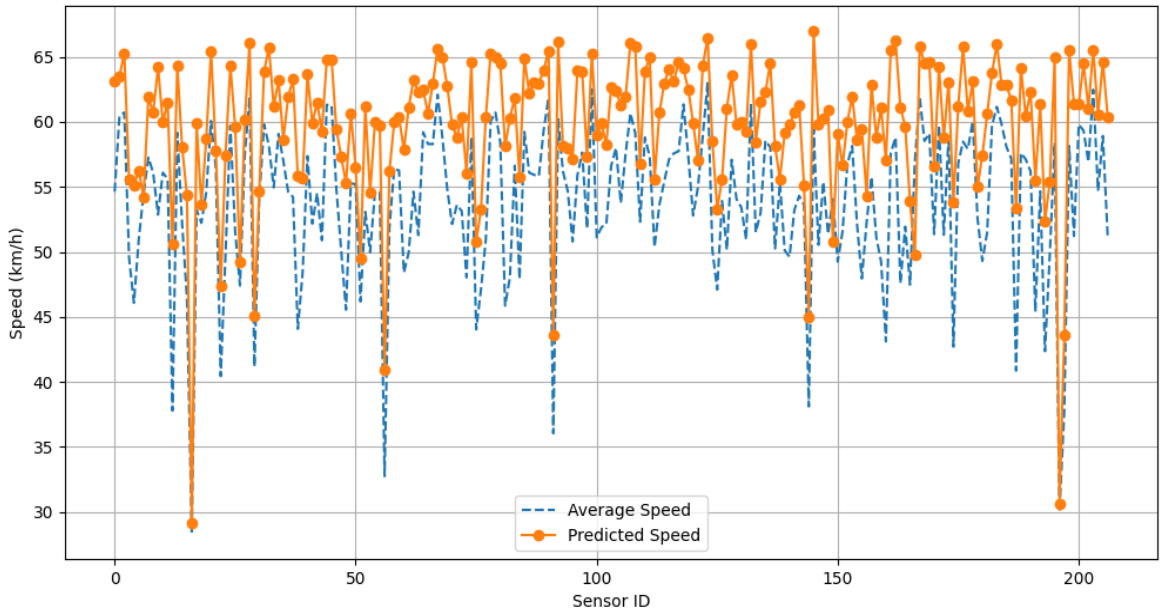


Figure 6. Predicted vs Average Speed per sensor



The histogram in Figure 5 demonstrates that most predicted speeds fall between 45 and 65 km/h, which aligns well with typical urban driving conditions. This supports the validity of the model's outputs and its applicability for urban traffic prediction tasks.

Figure 6 shows that the predicted traffic speeds for each sensor align closely with the historical average values. The clustering along the diagonal line reflects high prediction accuracy and confirms that the model captures the underlying traffic patterns effectively.

5. Conclusions

The developed system offers a complete and modular approach to real-time traffic speed prediction by leveraging neural networks and distributed computing. Through careful design, training, and deployment, the project demonstrates that time-series data from hundreds of sensors can be efficiently processed to yield accurate next-step predictions. The training process confirmed strong model convergence and generalization, as evidenced by low MSE and MAE scores and early stopping behavior.

On the simulation side, scalability was successfully tested across increasing thread counts, though saturation was observed beyond 16 workers. This confirms the computational potential of the system on multi-core machines. However, real-world performance revealed that overhead from distributed systems can reduce speed, a limitation that should be addressed in future iterations. The findings underscore the trade-off between concurrency and orchestration in distributed environments.

The model also performed reliably in terms of output distribution, error variance across sensors, and predictive alignment with historical averages. These indicators support the model's stability and its capacity to generalize across different urban traffic conditions.

From an architectural standpoint, the integration with FuncX ensures portability and reproducibility. The codebase supports reusability, the system scales modularly, and the data pipeline is well-structured for continuous integration with larger traffic infrastructure platforms. Given these properties, the system is well-positioned for future enhancements, including real-time streaming, integration of spatial graph information, or deployment within smart city control centers.

In conclusion, this project not only addresses a practical problem in traffic analytics but also showcases the value of combining Big Data techniques with machine learning and distributed computing. It lays the groundwork for continued academic exploration, potential industry adoption, and future innovation in urban mobility systems.

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

Appendix

A1. Source Code

Main.py – traffic prediction on one endpoint

```
# ----------------------------------------------------
# Main function to analyze traffic and predict speed
# ----------------------------------------------------
def traffic_prediction(params):
    import numpy as np
    import pandas as pd
    import pickle
    import tensorflow as tf

    logs = []

    try:
        adj_pickle_path = params['adj_pickle_path']
        h5_path = params['h5_path']
        model_path = params['model_path']
        scaler_path = params['scaler_path']
        output_path = params['output_path']

        lookback = 5

        # Load data
        logs.append(f"[INFO] Loading traffic data from: {h5_path}")
        df = pd.read_hdf(h5_path, key='df')
        data = df.values  # shape: (num_timesteps, num_sensors)

        # Load scaler
        logs.append(f"[INFO] Loading scaler from: {scaler_path}")
        with open(scaler_path, 'rb') as f:
            scaler = pickle.load(f)
        data_scaled = scaler.transform(data)

        if data_scaled.shape[0] < lookback:
            logs.append("[ERROR] Not enough data for lookback.")
            return logs

        # Prepare input sequence
        X_pred = np.expand_dims(data_scaled[-lookback:, :], axis=0)
```

```
    # Load model
    logs.append(f"[INFO] Loading model from: {model_path}")
    model = tf.keras.models.load_model(model_path, compile=False)
    logs.append("[INFO] Model loaded successfully.")

    # Predict traffic speed based on input sequence
    y_pred_scaled = model.predict(X_pred)[0]
    y_pred = scaler.inverse_transform(y_pred_scaled.reshape(1, -1))[0]

    logs.append("[INFO] Prediction completed.")

    # Write results in txt file
    with open(output_path, 'a') as f:
        f.write("===== Traffic Prediction Results =====\n")
        for idx, speed in enumerate(y_pred):
            avg_speed = np.mean(data[:, idx])
            line = (f"🚗 Driving segment from sensor {idx}:\n"
                    f"   Average speed: {avg_speed:.2f} km/h\n"
                    f"   Predicted next: {speed:.2f} km/h\n")
            logs.append(line.strip())
            f.write(line)
        f.write("\n")

    logs.append(f"[INFO] Results written to: {output_path}")
    return logs

except Exception as e:
    logs.append(f"[ERROR] Exception occurred: {e}")
    return logs
```

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

Multiple_node_prediction.py – prediction on 4 endpoints

```
def predict_subset (h5_path, model_path, scaler_path, sensor_indices, lookback, output_txt_path):
    logs = []
    try:
        # Load data
        logs.append(f"[INFO] Loading traffic data from: {h5_path}")
        df = pd.read_hdf(h5_path, key='df')
        data = df.values

        logs.append(f"[INFO] Loading scaler from: {scaler_path}")
        with open(scaler_path, 'rb') as f:
            scaler = pickle.load(f)
        data_scaled = scaler.transform(data)

        if data_scaled.shape[0] < lookback:
            logs.append("[ERROR] Not enough data for lookback.")
            return logs

        logs.append("[INFO] Preparing input for prediction.")
        X_pred = np.expand_dims(data_scaled[-lookback:, :], axis=0)

        logs.append(f"[INFO] Loading model from: {model_path}")
        model = tf.keras.models.load_model(model_path, compile=False)
        y_pred_scaled = model.predict(X_pred)[0]
        y_pred = scaler.inverse_transform(y_pred_scaled.reshape(1, -1))[0]
        logs.append("[INFO] Prediction completed.")
        os.makedirs(os.path.dirname(output_txt_path), exist_ok=True)
        with open(output_txt_path, 'a') as f:
            f.write("===== Traffic Prediction Results =====\n")
            for idx in sensor_indices:
                avg_speed = np.mean(data[:, idx])
                pred_speed = y_pred[idx]
                line = (f"🚗 Driving segment from sensor {idx}:\n"
                        f"   Average speed: {avg_speed:.2f} km/h\n"
                        f"   Predicted next: {pred_speed:.2f} km/h\n")
                f.write(line)
                logs.append(line.strip())
            f.write("\n")

        logs.append(f"[INFO] Results written to: {output_txt_path}")
        return logs

    except Exception as e:
        logs.append(f"[ERROR] Exception occurred: {str(e)}")
        return logs
```

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

Simulate distributed – on multiple threads

```
# Config
lookback = 5
num_workers = 64

# Load model and scaler
model = tf.keras.models.load_model(model_path, compile=False)
with open(scaler_path, 'rb') as f:
    scaler = pickle.load(f)

# Load data

# Simulate distributed nodes – each one makes inference on one section
def predict_on_sensor(sensor_id):
    time.sleep(random.uniform(0.2, 0.6))  # simulare delay
    single_sensor_input = X_pred[0, :, sensor_id].reshape(1, lookback, 1)
    repeated_input = np.repeat(single_sensor_input, num_sensors, axis=2)  # for
model input shape
    y_pred_scaled = model.predict(repeated_input)[0]
    y_pred = scaler.inverse_transform(y_pred_scaled.reshape(1, -1))[0]
    avg_speed = np.mean(data[:, sensor_id])
    return {
        "sensor": sensor_id,
        "prediction": y_pred[sensor_id],
        "avg_speed": avg_speed,
        "latency": round(random.uniform(0.2, 0.6), 3)
    }


start_time = time.time()
# Run simulation in parallel
with ThreadPoolExecutor(max_workers=num_workers) as executor:
    results = list(executor.map(predict_on_sensor, range(num_sensors)))

# Save results for analysis
df_res = pd.DataFrame(results)
os.makedirs("/home/darime/outputs", exist_ok=True)
df_res.to_csv("/home/darime/outputs/simulated_predictions.csv",
index=False)
print(f"Execution time: {time.time() - start_time:.2f} seconds")
```

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

A2. Training code

train_model_nn.py – only relevant code

```
# Paths
data_path = '/home/darime/data/METR-LA.h5'
model_path = '/home/darime/models/keras_model_all_sensors.h5'
scaler_path = '/home/darime/models/scaler_all_sensors.pkl'
loss_plot_path = '/home/darime/outputs/loss_plot.png'
mae_plot_path = '/home/darime/outputs/mae_plot.png'

# Hyperparameters
lookback = 5
epochs = 100
batch_size = 64

# Load data
print(f"[INFO] Loading data from {data_path}...")
df = pd.read_hdf(data_path, key='df')
data = df.values
print(f"[INFO] Raw data shape: {data.shape}")

# Scale data
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)
print("[INFO] Data scaled.")
```

```
os.makedirs(os.path.dirname(scaler_path), exist_ok=True)
with open(scaler_path, 'wb') as f:
    pickle.dump(scaler, f)
print(f"[INFO] Scaler saved at {scaler_path}")


# Prepare sequences
X, y = [], []
T, num_sensors = data_scaled.shape
for i in range(T - lookback):
    X.append(data_scaled[i:i + lookback, :])
    y.append(data_scaled[i + lookback, :])
X = np.array(X)
y = np.array(y)
print(f"[INFO] Training data shape: X={X.shape}, y={y.shape}")


# Build model
model = Sequential([
    InputLayer(input_shape=(lookback, num_sensors)),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.2),
    Dense(128, activation='relu'),
    Dropout(0.2),
    Dense(num_sensors)
])
```

```
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()


# Callbacks
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-5, verbose=1)


# Train
print("[INFO] Training model...")
history = model.fit(X, y, epochs=epochs, batch_size=batch_size,
                    validation_split=0.1,
                    callbacks=[early_stop, reduce_lr],
                    verbose=2)
# Evaluate
final_loss, final_mae = model.evaluate(X, y, verbose=0)
print(f"[INFO] Final Loss (MSE): {final_loss:.4f}")
print(f"[INFO] Final MAE: {final_mae:.4f} km/h")


# Save model
os.makedirs(os.path.dirname(model_path), exist_ok=True)
model.save(model_path)
print(f"[INFO] Model saved at {model_path}")
```

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

Training Logs - sample

[INFO] Training model...

Epoch 1/100

482/482 - 3s - 6ms/step - loss: 0.3845 - mae: 0.3909 - val_loss: 0.3470 - val_mae: 0.3180 - learning_rate: 1.0000e-03

Epoch 2/100

482/482 - 2s - 4ms/step - loss: 0.2675 - mae: 0.3098 - val_loss: 0.3450 - val_mae: 0.3274 - learning_rate: 1.0000e-03

Epoch 61/100

Epoch 61: ReduceLROnPlateau reducing learning rate to 3.125000148429535e-05.

482/482 - 2s - 5ms/step - loss: 0.1678 - mae: 0.2355 - val_loss: 0.2676 - val_mae: 0.2739 - learning_rate: 3.1250e-05

Epoch 65/100

482/482 - 2s - 4ms/step - loss: 0.1680 - mae: 0.2362 - val_loss: 0.2651 - val_mae: 0.2630 - learning_rate: 3.1250e-05

Epoch 66/100

Epoch 66: ReduceLROnPlateau reducing learning rate to 1.5625000742147677e-05.

482/482 - 2s - 4ms/step - loss: 0.1659 - mae: 0.2351 - val_loss: 0.2650 - val_mae: 0.2618 - learning_rate: 3.1250e-05

[INFO] Final Loss (MSE): 0.1343

[INFO] Final MAE: 0.1988 km/h

[INFO] Loss plot saved at /home/darime/outputs/loss_plot.png

[INFO] MAE plot saved at /home/darime/outputs/mae_plot.png

Meseșan Daria-Maria
Business Modelling and Distributed Computing Master

A3. Globus Compute setup – one node

```
gc = Client()

gc.serializer = ComputeSerializer(strategy_code=CombinedCode())

# Define endpoint ID

endpoint_id = 'ac9f12d1-f1f7-44d3-a40f-b0ee9ea6618b'

# Register the function

func_id = gc.register_function( function=traffic_prediction,

   description="Analyze traffic and predict next speed for all sensors using METR-LA dataset.")

print(f"Function registered with ID: {func_id}")
```

A4. Analysis Sample Output - traffic_analysis_with_nn.txt

```
===== Traffic Prediction Results =====

🚗 Driving segment from sensor 0:

  Average speed: 54.63 km/h

  Predicted next: 63.12 km/h

🚗 Driving segment from sensor 1:

  Average speed: 60.45 km/h

  Predicted next: 63.50 km/h

🚗 Driving segment from sensor 2:

  Average speed: 60.73 km/h

  Predicted next: 65.29 km/h

🚗 Driving segment from sensor 20:

  Average speed: 60.08 km/h

  Predicted next: 59.44 km/h
```