Comparative Analysis of Neural Network-Based Techniques for Vehicular Location Prediction

Sina Ebrahimi, *SID: 13207801, Word Count: 3284*
Centre for Future Transport and Cities
Coventry University
Coventry, UK
ebrahimis@coventry.ac.uk

Abstract

Vehicular location prediction is crucial for improving Quality of Service (QoS) in fifth-generation (5G) radio Resource Allocation (RA). This study presents a novel comparison of five Neural Network (NN) models—Recurrent NN (RNN), Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), one-dimensional convolutional NN (Conv1D), and Multi-Layer Perceptron (MLP)—in predicting vehicle positions using a dataset of vehicular mobility traces from Seoul, South Korea. The models are assessed using metrics such as coefficient of determination (R^2 score), Root Mean Square Error (RMSE), loss over epochs, and execution time per epoch. Out of the models tested, the MLP model showed the best performance, with an RMSE of 0.42 meters and an R^2 score of 0.999992. This represents a 32% decrease in RMSE compared to Conv1D and the recurrent models and a significantly higher R^2 score. Moreover, MLP demonstrated the most rapid convergence and the shortest average execution time per epoch, emphasizing its efficiency. The results suggest that using simpler architectures, such as MLP, is quicker and more effective for this particular task. This provides valuable information for designing proactive RA strategies in 5G networks.

Index Terms

Prediction, Vehicular Mobility, Proactive Mobility Prediction, 5G, Handover Management, Radio Resource Management, Neural Networks

I. Introduction

Fifth-generation (5G) wireless technology has enabled fast and responsive communication, which has greatly impacted sectors such as transportation. Especially in metropolitan areas, it is imperative to properly manage network resources for vehicle communications as vehicles are increasingly included in the Internet of Things (IoT) ecosystem. Radio Resource Management (RRM) is crucial in 5G networks as it optimizes network performance and guarantees Quality of Service (QoS) for users. Precise prediction of vehicle location can greatly improve RRM by facilitating proactive Resource Allocation (RA), enabling networks to allocate users to the most optimal base stations in advance of handovers (Taleb et al., 2017). This proactive approach is critical for properly controlling the always-shifting traffic patterns in urban areas because it can significantly reduce delays and improve connectivity.

Several Neural Network (NN) structures, including Recurrent NN (RNN), Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), Convolutional NN (CNN), and Multi-Layer Perceptron (MLP), have demonstrated potential in forecasting intricate patterns (Hochreiter & Schmidhuber, 1997; Wang et al., 2022). Nevertheless, it is necessary to conduct a thorough analysis and evaluation of these architectures specifically in relation to vehicular location prediction for 5G RRM. This study aims to fill this void by assessing the efficacy of these NN architectures using vehicular mobility data obtained from Seoul, South Korea (Kumbhar, 2020). The models are assessed in terms of their computational efficiency, rate of convergence, and capacity to precisely forecast, offering insightful information on their appropriateness for real-time 5G applications. This work is distinctive in that it compares several NN architectures to enhance vehicle location predictions.

The next sections of this paper are organized as follows: Literature is briefly reviewed in Sec. II. The problem and dataset are introduced in Sec. III. The various NN-assisted methods utilized in this paper are succinctly introduced in Sec. IV. Sec. V presents the experimental results. Ultimately, discussion and future work are provided in Sec. VI.

II. Related Work

The ability of vehicle location prediction to improve 5G radio performance has recently gained significant interest. While CNN-based models have also been shown to extract spatial features from this kind of data, LSTM networks have been shown to be successful in capturing temporal relationships in mobility data. (Lin et al., 2022; Zhao et al., 2017)). Within the context of 5G, scientists have looked at how Machine Learning (ML) methods might enhance network performance. Deep reinforcement learning techniques have specifically shown promise in optimizing joint mobility prediction and resource management (Choi et al., 2024; Farooq & Imran, 2017).

ML algorithms have enhanced handover management in 5G networks, reducing service failures and thus greatly improving QoS (Arshad et al., 2016; Priyanka et al., 2023). Recent research suggests that simpler models such as MLP and one-dimensional convolutional NN (Conv1D) can occasionally achieve better performance than more complex recurrent architectures, especially in tasks involving spatio-temporal prediction. Zhang et al., 2023 has demonstrated that MLP networks are successful in forecasting handovers in ultra-dense 5G networks, effectively managing intricate urban settings with exceptional precision and efficiency. Furthermore, Conv1D networks have demonstrated their efficacy in capturing localized patterns in sequential data, thus reinforcing the possibility of employing simpler models for mobility prediction tasks (Zhao et al., 2017).

These studies demonstrate the potential utility of NN-based techniques in determining the location of vehicles in 5G RRM. However, further investigation is required to comprehensively compare the various deep learning-based architectures. The objective of this study is to evaluate different architectures and identify the most suitable approach for proactive RA in 5G networks, thus addressing the existing gap in knowledge.

III. Dataset

The `vehicular mobility trace at Seoul, South Korea` dataset is accessed from IEEE DataPort (Kumbhar, 2020). The dataset is collected by simulating vehicles in urban environment using SUMO, which exports XML files as the raw output[1] (German Aerospace Center (DLR), 2024). The trace location in Seoul, South Korea has an area spanning over 2.5 × 1.5 km where there are are more than 30 intersections. A number of simulated vehicles are moving in the road topology. The dataset is introduced in a study regarding vehicular reliable routing that aims to select the routes with the longest connectivity time to efficiently deliver communication messages (Kumbhar & Shin, 2021), where the authors introduced a heuristic to solve their problem. However, our goal is different, as we aim to predict the location of each vehicle in the next time step. We use the sparse scenario introduced in Kumbhar, 2020 to conduct this experiment.

As part of the preprocessing stage, we initially transform the XML file `OpenStreetMap Trace for a Sparse Traffic.xml` into a CSV file. Then, we remove features `type, lane, slope, pos` as they do not influence vehicle mobility. We also convert the data type of the features `id` and `time` to integer to enhance readability and performance. The preprocessed features are shown in Table I. Moreover, Fig. 1a shows the coordinates of all vehicles across all the time steps in the environment. Overall, there are *6 features and 1,483,385 records (comprising 3592 unique vehicles and 8220 time steps (seconds)*.

Each time step might include some vehicles, and there is time dependency for the mobility of each individual vehicle throughout time steps. In order to predict the behavior of each vehicle, which is moving independently of the others, we generate sequences of length 5 for each vehicle by applying the `group by` method. Each sequence contains 4 features of x, y, angle, and speed for 5 consecutive time step for a unique vehicle followed by its location (x and y coordinates) at the fifth time step. We split the data with 80-20% ratio between train and test datasets (detailed in Sec. B-A).

TABLE I: Dataset Features

Column	Data Type	Range	Unit	Description (Kumbhar, 2020)
time	Integer	0 to 8219	seconds	The time step described by the values within this timestep-element
id	Integer	0 to 4312 (3592 distinct vehicles)	-	The id of the vehicle
x	Float	2.14 to 1420.58	meters	(longitude) The absolute X coordinate of the vehicle (center of front bumper). The value depends on the given geographic projection
y	Float	820.68 to 2212.71	meters	(latitude) The absolute Y coordinate of the vehicle (center of front bumper).
angle	Float	0.0 to 360.0	degrees	The angle of the vehicle in navigational standard (0-360 degrees, going clockwise with 0 at 12'o clock)
speed	Float	0.0 to 37.38	m/s	The speed of the vehicle

*The blue features are the subject of prediction for individual vehicles.

TABLE II: Hyperparameter Grid for Tuning

Hyperparameter	**Values**
Hidden Size	50, 100
No. of Layers	2, 3, 4
Learning Rate	0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005
No. of Epochs	30

[1]Please refer to (German Aerospace Center (DLR), 2024) for the complete set of features.

IV. METHODS

This section explains the fundamental models employed for predicting vehicle movement, focusing on RNN, LSTM, GRU, Conv1D, and MLP. Each model is chosen for its ability to capture temporal and spatial patterns in sequential data.

A. RNN

RNNs are designed for sequential data processing, utilizing a looped structure to retain information from previous time steps. This makes them suitable for tasks like time-series forecasting and vehicle movement prediction, where temporal data sequence is key.

The RNN architecture consists of:

- **Input Layer:** Receives sequential data as input vectors at each time step.
- **Hidden Layers:** Feature recurrent connections, allowing the network to preserve information over time. The hidden state h_t at time step t is calculated as:

$$h_t = \sigma(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b_h)$$

  where x_t is the input at t, W_{xh} and W_{hh} are the weight matrices, b_h is the bias vector, and σ denotes the activation function (e.g., tanh or ReLU[2]).
- **Output Layer:** Generates the output y_t for the current time step:

$$y_t = \sigma(W_{hy} \cdot h_t + b_y)$$

  where W_{hy} represents the weight matrix that connects the hidden state to the output, and b_y is the bias vector.

RNNs are foundational for sequence modeling but can struggle with long-term dependencies due to vanishing gradients, making them better suited for short-term predictions (Ghojogh & Ghodsi, 2023).

B. LSTM

LSTM networks extend RNNs by incorporating memory cells and gating mechanisms to handle long-term dependencies effectively (Hochreiter & Schmidhuber, 1997). This makes them ideal for tasks requiring the retention of information over extended periods, like predicting vehicle trajectories based on historical data.

The LSTM architecture includes:

- **Input Layer:** Sequential data enters the network at each time step.
- **LSTM Cells:** Each cell contains forget (f_t), input (i_t), and output (o_t) gates, which manage the flow of information:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \qquad i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \qquad C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \qquad h_t = o_t \cdot \tanh(C_t)$$

  where C_t is the cell state.
- **Output Layer:** The final hidden state generates the prediction, which represents the next position of the vehicle.

LSTMs are selected for their robustness in modeling complex temporal sequences, making them suitable for long-term vehicle trajectory prediction.

C. GRU

GRUs simplify the LSTM architecture by merging the input and forget gates into a single update gate, reducing complexity and computational cost while still capturing long-term dependencies (Ghojogh & Ghodsi, 2023).

The GRU architecture features:

- **Input Layer:** Sequential data is fed into GRU cells at each time step.
- **GRU Cells:** Comprising reset (r_t) and update (z_t) gates that regulate information flow:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \qquad z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$
$$\tilde{h}_t = \tanh(W_h \cdot [r_t \cdot h_{t-1}, x_t] + b_h) \qquad h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h}_t$$

  where $\tilde{h}_t$ represents the candidate activation, and h_t is the updated hidden state.
- **Output Layer:** Generates the final prediction from h_t.

GRUs are chosen for their efficiency and performance in tasks where both precision and computational resources are critical.

[2]ReLU is chosen in our RNN method due to its flexibility in remembering states.

D. Conv1D

Conv1D models are used to analyze sequential data, utilizing convolutional filters to capture spatial hierarchies, making them effective for identifying local patterns in vehicle trajectory data (Kiranyaz et al., 2021).

The Conv1D architecture includes:

- **Input Layer:** The input data consists of sequences representing time-varying features, which are flattened before being fed into the input layer.
- **Convolutional Layers:** Conv1D layers utilize a collection of convolutional filters to process the input sequence. The filters move across the input, capturing specific characteristics in the surrounding area:

$$z_t = \sum_{i=1}^{k} w_i \cdot x_{t+i-1} + b$$

  where k is the kernel size, w_i are the filter weights, and b is the bias term.
- **Pooling Layers:** Reduce dimensionality, typically using max pooling to retain key features.
- **Fully Connected (Dense) Layer:** Aggregates features to produce the final output.

Conv1D models excel at capturing local dependencies, providing an alternative to recurrent networks by emphasizing local feature extraction.

E. MLP

MLPs are feedforward NNs consisting of multiple layers of interconnected neurons, capable of learning complex, non-linear relationships between input features and output predictions (Zhang et al., 2023).

The MLP architecture comprises:

- **Input Layer:** Receives flattened sequential data.
- **Hidden Layers:** Each layer l performs a weighted sum of inputs followed by a non-linear activation function (i.e. ReLU):

$$h^{(l)} = \sigma(W^{(l)} \cdot h^{(l-1)} + b^{(l)})$$

  where $h^{(l)}$ is the output of the l-th layer, $W^{(l)}$ is the weight matrix, and $b^{(l)}$ is the bias vector.
- **Output Layer:** Provides the predicted output.

Although MLPs do not explicitly incorporate temporal dependencies, they can be utilized for predicting vehicle movement by considering the sequential data as a collection of features. This allows for the learning of non-linear relationships between past movements and future positions. MLPs are employed as a reference model to evaluate the influence of integrating temporal structures in prediction tasks. They offer a direct method for modeling relationships within the data, acting as a benchmark for more intricate models such as RNNs, LSTMs, and Conv1D networks.

Fig. 1b illustrates a generic architecture for all the methods, assuming the presence of three layers, each consisting of 50 neurons. The depicted diagram resembles MLPs and does not illustrate the complexities of other techniques (such as the forget gate in LSTM), despite the fact that all these methods have the same number of layers and neurons. For more detailed information about the models (e.g., utilized activation functions), please refer to step 3 in Sec. B-B.

V. Experimental Results

The experiments were carried out using PyTorch and widely used Python libraries on a High-Performance Computing (HPC) node at Coventry University called `zeus400`. This node is equipped with an NVIDIA Tesla K80 GPU. This section provides an overview of the metrics used to compare models, the process of tuning hyperparameters, and the comparative analysis of five NN-based techniques using the selected hyperparameters. The Adam optimizer was consistently employed in all methods to minimize the loss during training Kingma & Ba, 2015, with Mean Square Error (MSE) serving as the loss function in all experiments.

Different activation functions and techniques were utilized in each of the five NN architectures to improve performance. The LSTM and GRU models employ the `tanh` activation function as the default choice for their recurrent units, and the final output is then processed through a linear layer. The RNN model also employs a `tanh` activation function in the recurrent layer, which is then followed by a linear layer. The Conv1D model utilizes the `ReLU` activation function after each convolutional layer, followed by max pooling. The model then concludes with global average pooling before reaching a fully connected layer. The MLP model employs the `ReLU` activation function following each linear layer. Additionally, the input data is flattened to ensure compatibility with the fully connected structure of the network.

A. Metrics

The efficacy of the location prediction models was assessed using fundamental metrics: Root Mean Square Error (RMSE), Mean Absolute Error (MAE), and the coefficient of determination (R^2 score). In addition, we monitored the loss function throughout the epochs and documented the average time it took to complete each epoch.

a) RMSE: is a metric that measures the magnitude of the differences between predicted and actual values. It emphasizes larger errors by squaring the discrepancies. Smaller RMSE values indicate higher model accuracy (Willmott & Matsuura, 2005). The study emphasizes the significance of RMSE as it offers a quantitative assessment of the overall precision of the models' predictions. Smaller RMSE values indicate better performance of the models.

b) MAE: is a metric that calculates the average of the absolute differences between predicted and actual values. It provides a straightforward measure of prediction error without giving more importance to larger errors. It provides additional information to RMSE by giving a better understanding of the accuracy of predictions, especially in terms of the typical sizes of errors (Willmott & Matsuura, 2005).

c) R^2 Score: quantifies the amount of variability in the dependent variable that can be explained by the model. A higher value, with a maximum of 1, indicates a more robust correlation with the data. It is crucial to comprehend the model's ability to explain things effectively (Nagelkerke, 1991).

d) Loss: was monitored throughout all epochs to assess the convergence and stability of the model during training. Lower loss values suggest better model performance in fitting the training data (Goodfellow et al., 2016).

e) Average Execution Time per Epoch: is important for assessing computational efficiency, helping balance accuracy with time constraints in practical applications (Sze et al., 2017).

B. Hyperparameter Tuning

Optimizing NN model performance requires thorough hyperparameter tuning (Goodfellow et al., 2016). We analyzed different arrangements using a hyperparameter grid (Table II).

The grid search resulted in 48 unique combinations of hyperparameters, enabling a thorough examination of the hyperparameter space to determine the most optimal configurations for each model (e.g., LSTM). To accelerate the process, we evaluated each combination using 10% of the training and test datasets. The RMSE and R^2 scores for each method were assessed using the specified hyperparameter combinations, and a ranking system was developed to identify the optimal combination for each method, as detailed in Sec. B-B. The hyperparameter sets selected for final analysis are highlighted in green in Table III.

TABLE III: Top Experiments for Each Method

Method	Hidden Size	No. of Layers	Learning Rate (α)	RMSE	MAE	R^2 Score	Avg Epoch Time	Final Loss	Final Epoch	Combined Rank
RNN	100	2	0.0005	16.26	11.05	0.9890	18.12	115.46	30	7.5
	100	2	0.001	59.52	41.63	0.8712	18.25	3824.85	30	11.9
	100	2	0.05	180.51	126.69	-0.0470	18.2	35930.73	5	53.1
LSTM	100	4	0.0005	10.8	6.68	0.9959	23.69	53.01	30	8.2
	100	2	0.0005	10.76	7.3	0.9956	20.525	67.74	30	10.525
	100	2	0.001	12.49	8.39	0.9947	20.39	213.96	30	18.52
GRU	100	3	0.0005	10.68	6.84	0.9962	22.88	104.95	30	8.07
	100	2	0.0005	14.43	10.81	0.9933	21.95	96.07	30	11.26
	50	3	0.001	16.68	10.71	0.9901	21.16	410.59	30	16.94
Conv1D	50	3	0.0001	0.62	0.37	0.999985	18.44	0.83	30	26.45
	50	3	0.0005	0.525	0.4	0.999991	18.45	1.9	30	28.35
	50	2	0.0001	0.78	0.5	0.999976	16.73	0.67	30	32.2
MLP	100	3	0.00005	0.42	0.23	0.999992	15.78	0.36	30	7.34
	50	2	0.0001	0.525	0.41	0.999990	14.3	0.35	30	16.2
	100	2	0.0001	0.525	0.36	0.999988	14.41	0.38	30	18.0

C. Methods Comparison

Several NN architectures—RNN, LSTM, GRU, Conv1D, and MLP—are compared in this part. Using the previously mentioned metrics helps one evaluate the performance. The further study is based on the ideal hyperparameter configuration for every method, shown by the green highlighting in Table III. It is crucial to mention that when predicting vehicle positions (x and y coordinates), the units for RMSE, MAE, and loss (calculated using MSE) are expressed in meters (m).

a) Visualizing predicted locations: Fig. 2 displays the projected and real positions of all vehicles throughout all time intervals for all the utilized techniques. The predictions made by MLP and Conv1D closely align with the actual locations, whereas RNN fails to accurately detect vehicle locations. Nevertheless, the graph is excessively congested to analyze. Hence, in Figs. 3 and 4, we examine the paths of two random vehicles over a period of time for a more detailed analysis. The forecasting performed by Conv1D and MLP techniques is exceptional. Nevertheless, the LSTM and GRU methods fail to accurately predict the locations of some of the sampled vehicles. Furthermore, the RNN method exhibits significant disparities between actual and predicted locations.

b) RMSE, MAE, and R^2 Score Comparison: As depicted in Fig. 5, the MLP model surpasses all other methods in terms of the important metrics. The achieved RMSE is 0.42 meters, which is roughly 32% lower than the RMSE of the second-best Conv1D model, which is 0.62 meters. In addition, the MLP model achieves an MAE of 0.23 meters, which is 38% superior to the Conv1D model's MAE of 0.37 meters. The coefficient of determination (R^2) score for the MLP model is likewise the highest, reaching 0.999992. This value suggests an almost flawless level of prediction accuracy. This demonstrates the exceptional aptitude of MLP in precisely predicting vehicle positions with remarkable accuracy and a minimal error margin.

Although MLP outperforms Conv1D, Conv1D still shows strong performance with an RMSE of 0.62 meters and an MAE of 0.37 meters. On the other hand, the conventional recurrent models (RNN, LSTM, and GRU) demonstrate considerably higher RMSE and MAE values, with RNN doing the poorest. For example, the RMSE of the RNN is 16.26 meters, which is almost 38.7 times greater than the RMSE of the MLP. The notable disparity underscores the constraints of basic recurrent architectures in comprehending the intricate dynamics of vehicle movements compared to more straightforward models such as MLP and Conv1D.

c) Loss Over Epochs: Convergence pertains to the speed and efficiency at which the loss function of each model reaches a stable minimum value during the training phase. As seen in Fig. 6a, the MLP method shows the fastest and most constant convergence. Its loss from 6248.56 in the first epoch to roughly 0.11 by the 27th epoch clearly shows effective learning with almost no oscillations. Although the Conv1D approach has a slower pace, it exhibits satisfactory convergence, reaching a loss of around 3.45 by the 11th epoch and maintaining stability. These findings indicate that simpler models like MLP and Conv1D may optimize their weights more efficiently, rendering them highly effective for this particular task.

On the other hand, LSTM and GRU exhibit slower and less consistent convergence, reaching final losses of 22.59 and 174.87, respectively, by the 30th epoch. The GRU approach required only 12 epochs to complete and did not require further progression. The RNN model demonstrates the most significant challenge, as evidenced by its highest ultimate loss of 174.87. This indicates difficulties in accurately identifying the fundamental patterns within the data. The RNN model attained its minimum recorded loss of 116.77 meters in the 10th epoch. LSTM and GRU, despite their theoretical advantages for sequential data, necessitate a greater number of epochs for complete optimization and display more volatility during training due to their intricate structures. This investigation demonstrates that MLP and Conv1D not only produce lower loss values, but they also do it more effectively, rendering them more appropriate for this particular location prediction task.

d) Execution Time Per Epoch: With the Conv1D model closely behind, Fig. 6b shows that the MLP model displays the lowest average execution time per epoch. MLP's simplicity—that which includes fully connected layers free from sequential processing—helps to explain its computational efficiency. More effective than other methods is Conv1D, which employs convolutional operations. This results from the slower stride used for pooling layers, which guarantees accuracy while accelerating data processing.

On the other hand, recurrent models, specifically LSTM and GRU, have longer execution times per epoch. The LSTM model, for instance, has an average epoch duration of 23.69 seconds, which is over 50% longer than the MLP model's 15.78 seconds. The longer execution time is due to the complexity of the LSTM and GRU architectures. They have to process sequential data and keep hidden states between time steps, which makes them harder to compute.

VI. Discussion and Future Work

MLP emerged as the best-performing model in all categories, combining high prediction accuracy with excellent computational efficiency. Conv1D also demonstrated strong and consistent performance, making it a viable choice. While RNN, LSTM, and GRU are theoretically advantageous for sequence modeling, they are less suitable for accurate and efficient vehicle location prediction in this context.

The differences in architecture between these models are most likely to explain the findings. The MLP's fully connected layers are ideal for the small input size and specific nature of this prediction task, allowing it to reach high precision quickly. Conv1D's convolutional layers accurately capture spatial relationships, yielding high precision and short wall-clock times. On the other hand, the iterative nature of RNN, LSTM, and GRU, though beneficial in some sequential tasks, adds unnecessary complexity for this particular forecasting application.

Our study indicates that MLP and Conv1D models outperform traditional recurrent models (RNN, LSTM, GRU) in vehicular location prediction for 5G RRM. The MLP model performed particularly well on metrics such as R^2 score, RMSE, MAE, and computational efficiency. These findings call into question the widespread belief that recurrent models are always superior for sequential data handling. Instead, they propose that simpler architectures like MLP and Conv1D might work better for some prediction tasks. These architectures offer faster training and inference times, which are important for real-time 5G applications, while still being able to capture patterns in space and time.

Future research should prioritize the integration of these models into a proactive 5G RRM framework. This integration should make use of their predictive capabilities to optimize network resources by taking into account the expected positions of vehicles. Furthermore, investigating hybrid models that integrate the advantages of MLP, Conv1D, and recurrent architectures could improve prediction accuracy in more intricate settings. Additional research should evaluate the ability of these models to be expanded and applied in larger and denser urban settings, as well as their effectiveness in different traffic situations. Evaluating the models' capacity to adapt to changing mobility patterns and resilience to different data sampling rates would give substantial insight for practical application. Incorporating effective location prediction into 5G RRM enables the development of intelligent, user-centric networks capable of consistently delivering QoS in extremely mobile environments.

References

Arshad, R., Elsawy, H., Sorour, S., Al-Naffouri, T. Y., & Alouini, M.-S. (2016). Handover Management in 5G and Beyond: A Topology-Aware Skipping Approach. *IEEE Access*, *4*, 9073–9081. https://doi.org/10.1109/ACCESS.2016.2642538

Choi, S., Choi, S., Lee, G., Yoon, S.-G., & Bahk, S. (2024). Deep Reinforcement Learning for Scalable Dynamic Bandwidth Allocation in RAN Slicing with Highly Mobile Users. *IEEE Transactions on Vehicular Technology*, *73*(1), 576–590. https://doi.org/10.1109/TVT.2023.3302416

Farooq, H., & Imran, A. (2017). Spatio-Temporal Mobility Prediction in Proactive Self-Organizing Cellular Networks. *IEEE Communications Letters*, *21*(2), 370–373. https://doi.org/10.1109/LCOMM.2016.2623276

German Aerospace Center (DLR). (2024). Simulation of Urban MObility (SUMO) User Documentation: Floating Car Data (FCD) Output. www.sumo.dlr.de/docs/Simulation/Output/FCDOutput.html

Ghojogh, B., & Ghodsi, A. (2023). Recurrent Neural Networks and Long Short-Term Memory Networks: Tutorial and Survey. https://arxiv.org/abs/2304.11461

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning* [http://www.deeplearningbook.org]. MIT Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Kingma, D., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations (ICLR)*.

Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., & Inman, D. J. (2021). 1D convolutional neural networks and applications: A survey. *Mechanical Systems and Signal Processing*, *151*, 107398. https://doi.org/https://doi.org/10.1016/j.ymssp.2020.107398

Kumbhar, F. H. (2020). *Vehicular Mobility Trace at Seoul, South Korea*. https://doi.org/10.21227/kjzd-rk05

Kumbhar, F. H., & Shin, S. Y. (2021). VAR²: Novel Vehicular Ad-Hoc Reliable Routing Approach for Compatible and Trustworthy Paradigm. *IEEE Communications Letters*, *25*(2), 670–674. https://doi.org/10.1109/LCOMM.2020.3032753

LeNail, A. (2019). NN-SVG: Publication-Ready Neural Network Architecture Schematics. *Journal of Open Source Software*, *4*(33), 747. https://doi.org/10.21105/joss.00747

Lin, L., Li, W., Bi, H., & Qin, L. (2022). Vehicle Trajectory Prediction Using LSTMs With Spatial–Temporal Attention Mechanisms. *IEEE Intelligent Transportation Systems Magazine*, *14*(2), 197–208. https://doi.org/10.1109/MITS.2021.3049404

Nagelkerke, N. J. D. (1991). A note on a general definition of the coefficient of determination. *Biometrika*, *78*(3), 691–692. https://doi.org/10.1093/biomet/78.3.691

Priyanka, A., Gauthamarayathirumal, P., & Chandrasekar, C. (2023). Machine learning algorithms in proactive decision making for handover management from 5G & beyond 5G. *Egyptian Informatics Journal*, *24*(3), 100389. https://doi.org/https://doi.org/10.1016/j.eij.2023.100389

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, *105*(12), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

Taleb, T., Samdanis, K., Mada, B., Flinck, H., Dutta, S., & Sabella, D. (2017). On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Communications Surveys & Tutorials*, *19*(3), 1657–1681. https://doi.org/10.1109/COMST.2017.2705720

Wang, S., Cao, J., & Yu, P. S. (2022). Deep Learning for Spatio-Temporal Data Mining: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, *34*(8), 3681–3700. https://doi.org/10.1109/TKDE.2020.3025580

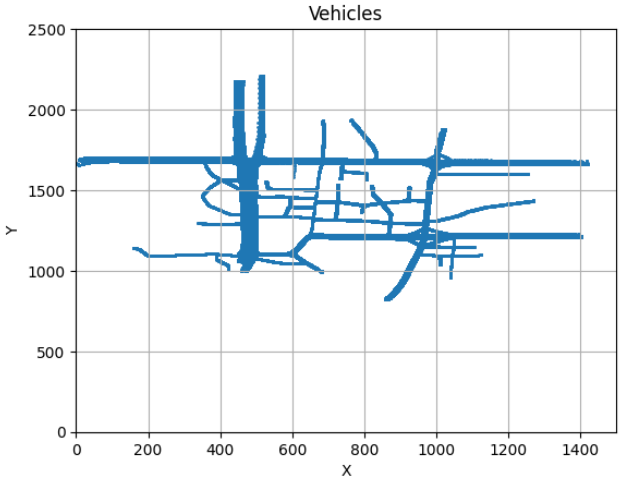
Willmott, C. J., & Matsuura, K. (2005). Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate Research*, *30*(1), 79–82. http://www.jstor.org/stable/24869236

Zhang, Z., Huang, Z., Hu, Z., Zhao, X., Wang, W., Liu, Z., Zhang, J., Qin, S. J., & Zhao, H. (2023). MLPST: MLP is All You Need for Spatio-Temporal Prediction. *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, 3381–3390. https://doi.org/10.1145/3583780.3614969

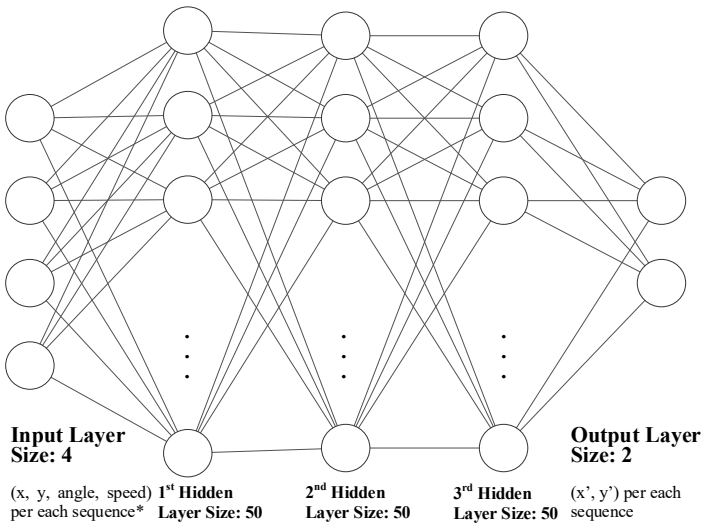
Zhao, B., Lu, H., Chen, S., Liu, J., & Wu, D. (2017). Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics*, *28*(1), 162–169. https://doi.org/10.21629/JSEE.2017.01.18

APPENDIX A
SCREENSHOTS AND STEPS

The graphs from the report are appeared in this section.



(a)



(b)

Fig. 1: a) Location of all vehicles throughout all time steps (Kumbhar, 2020), b) Generic NN architecture (Image generated using (LeNail, 2019)).

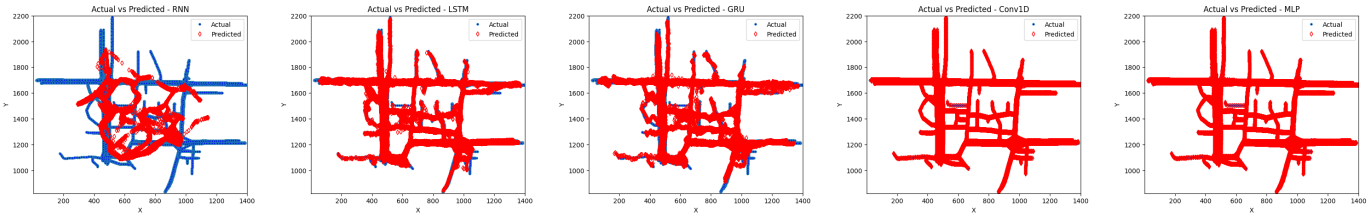


Fig. 2: Visualization of predicted and actual locations for all vehicles across all time steps using five different methods.

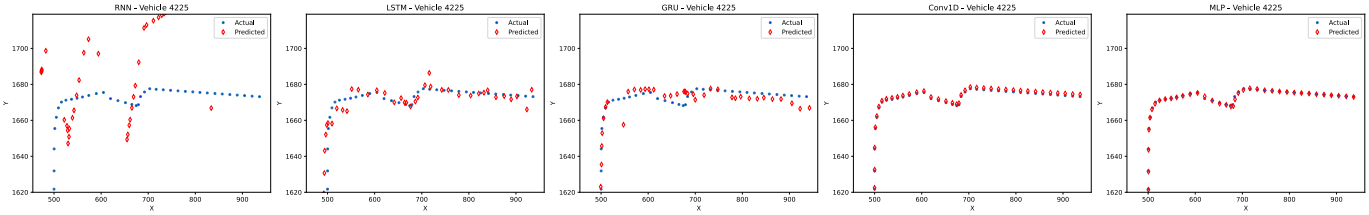


Fig. 3: Visualization of predicted and actual locations for a random vehicle across all time steps using five different methods (1).

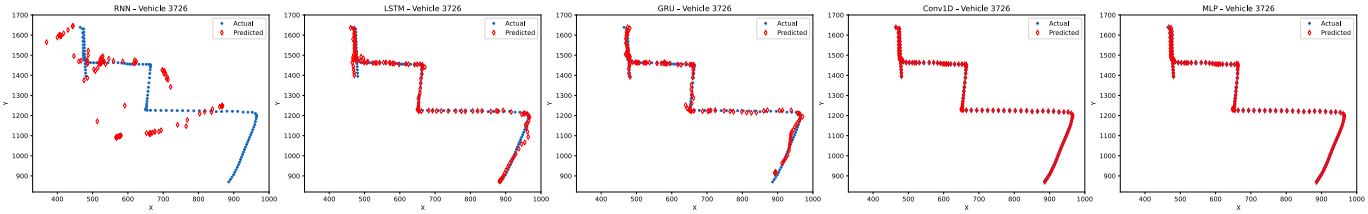


Fig. 4: Visualization of predicted and actual locations for a random vehicle across all time steps using five different methods (2).

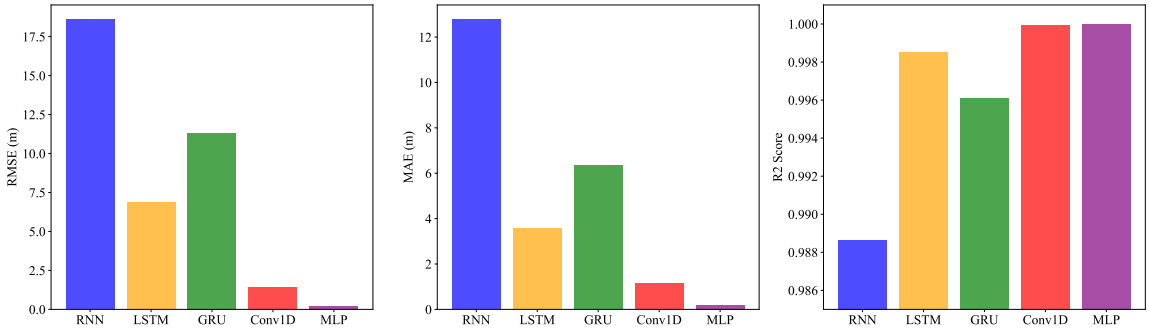
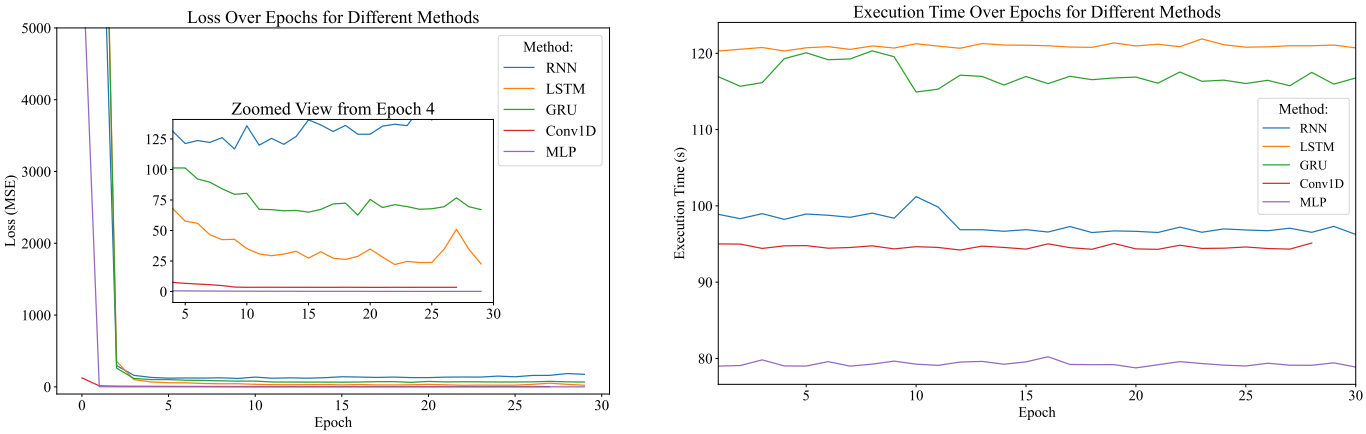


Fig. 5: RMSE, MAE, and R^2 score of the final epoch for various methods.



(a) (b)

Fig. 6: a) Loss (based on MSE) and b) average execution time (s) per epoch for different methods.

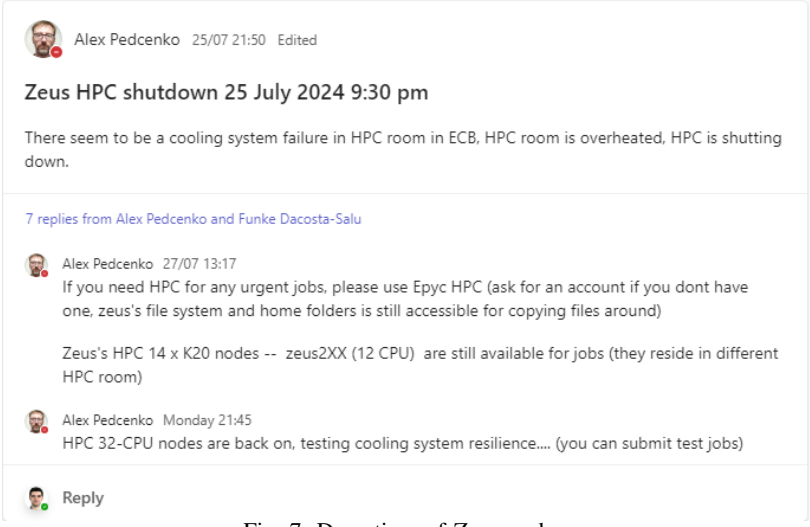


Fig. 7: Downtime of Zeus nodes.

Experiments were conducted on a Coventry University HPC node (`zeus400`) equipped with an NVIDIA Tesla K80 GPU (24GB GDDR5, comprising two NVIDIA GK210 GPUs), an Intel Xeon E5-2683 v4 CPU @ 2.10GHz with 32 cores, and 128 GB of RAM. The system runs a GNU/Linux operating system with kernel version 4.18, utilizing CUDA version 11.4 and Nvidia driver version 470.199.02. In the folder `ann_project`, a new Python virtualenv is created, and the following packages are installed:

```
[ebrahimis@zeus400 ~]$ module load python/last
[ebrahimis@zeus400 ~]$ module load cuda/last
[ebrahimis@zeus400 ~]$ module load gcc/gcc
[ebrahimis@zeus400 ~]$ cd ann_project/
[ebrahimis@zeus400 ann_project]$ python3 --version
Python 3.8.8
[ebrahimis@zeus400 ann_project]$ python -m venv venv
[ebrahimis@zeus400 ann_project]$ source venv/bin/activate
(venv) [ebrahimis@zeus400 ann_project]$ pip install pandas numpy torch scikit-learn matplotlib
(venv) [ebrahimis@zeus400 ann_project]$ code .
```

Unfortunately, the Zeus nodes have been non-operational since July 25th, 2024, and they remain inactive as of today's date, August 5th, 2024 (see Fig. 7). Although initial experiments with LSTM and some hyperparameter tuning were performed on Zeus nodes (no captured screenshots at the time), the majority of the results were obtained using Google Colab. Displayed below are screenshots depicting the concluding phase of the training process on an MLP model (Figs. 8, 9, 10, and 11). The detailed code of the notebooks can be found in Sec. B. An interested reader can re-run all the experiments using the existing code.

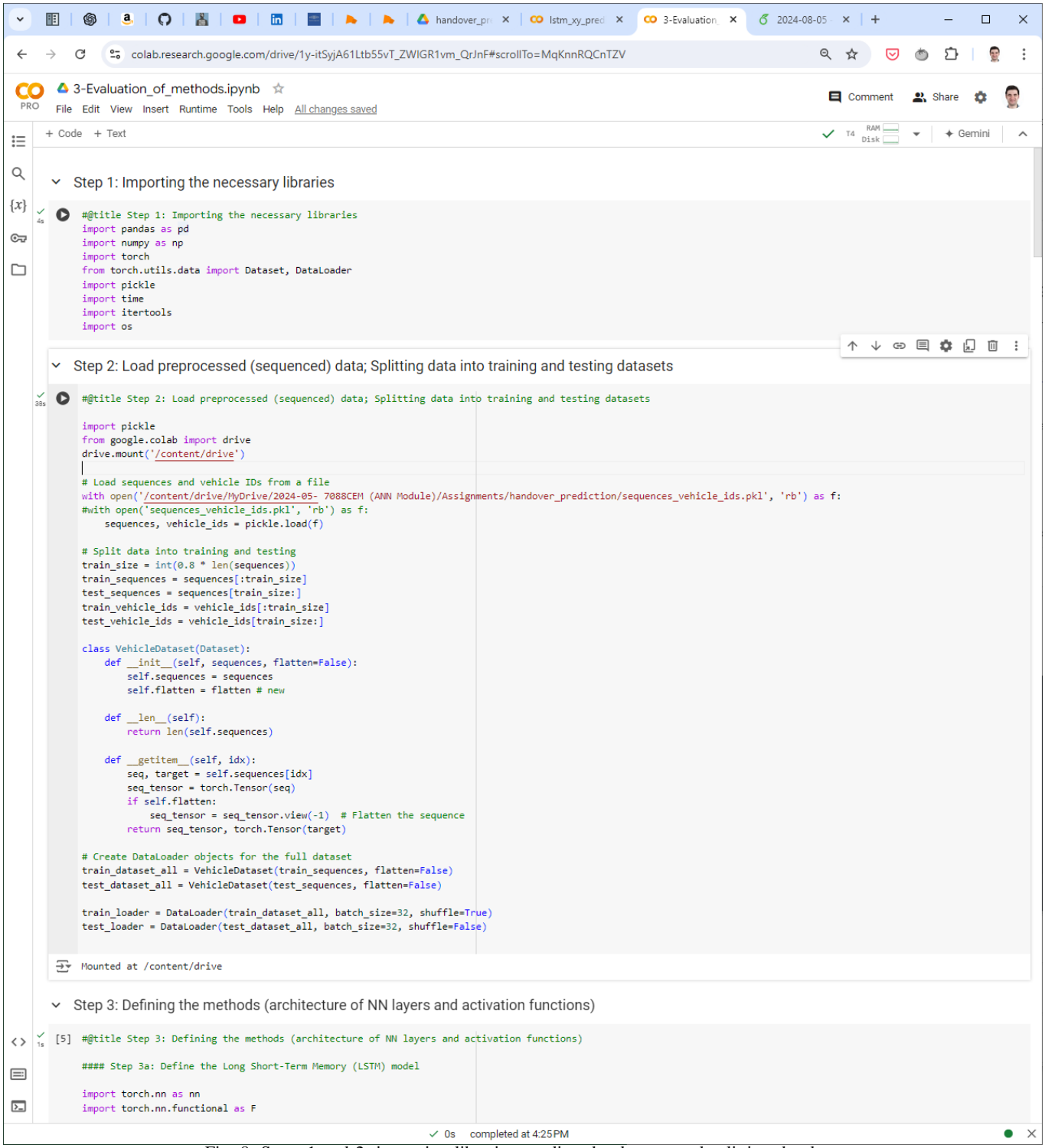


Fig. 8: Steps 1 and 2: importing libraries, reading the dataset, and splitting the data.

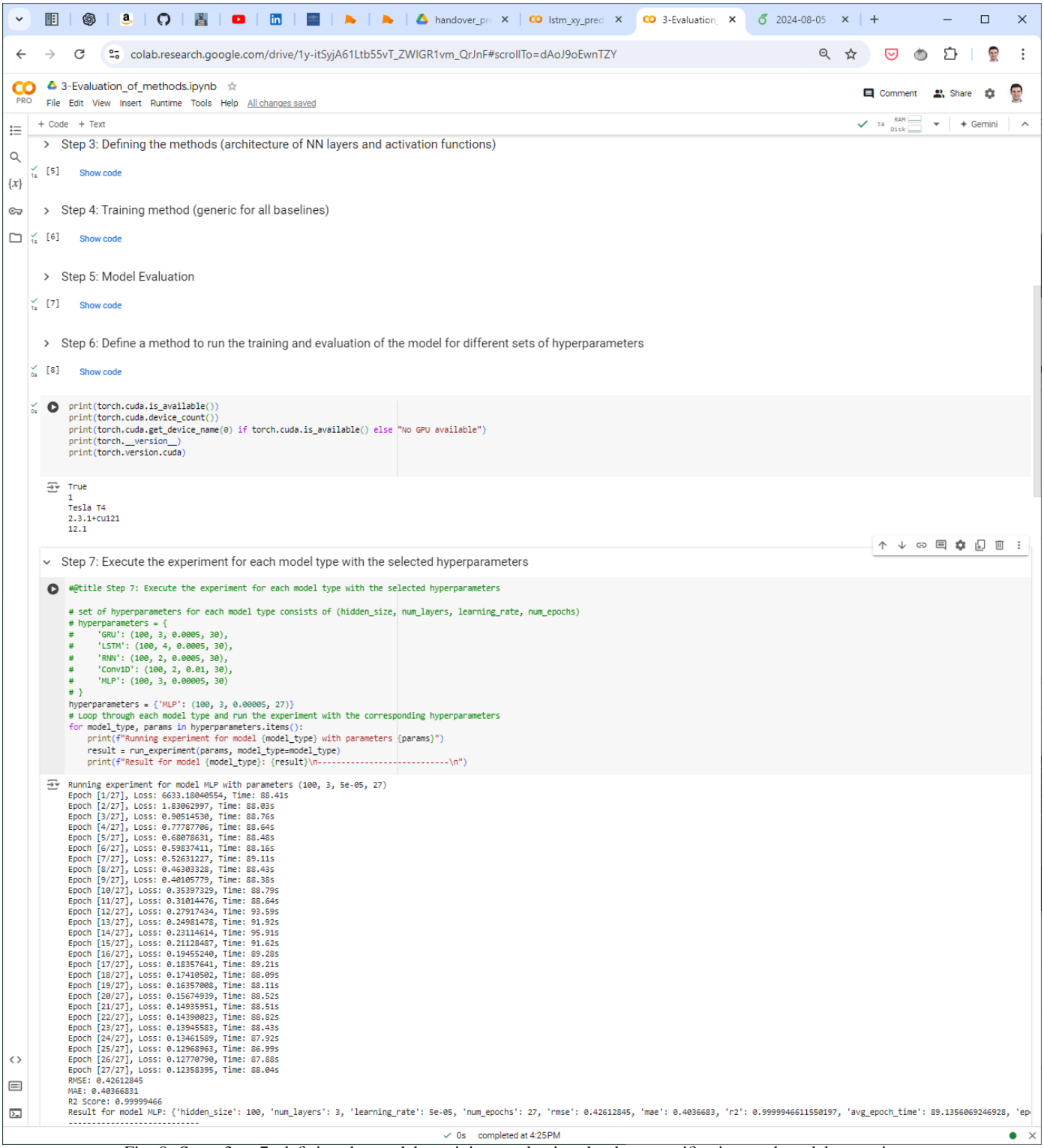


Fig. 9: Steps 3 to 7: defining the models, training, evaluation, hardware verification, and model execution.

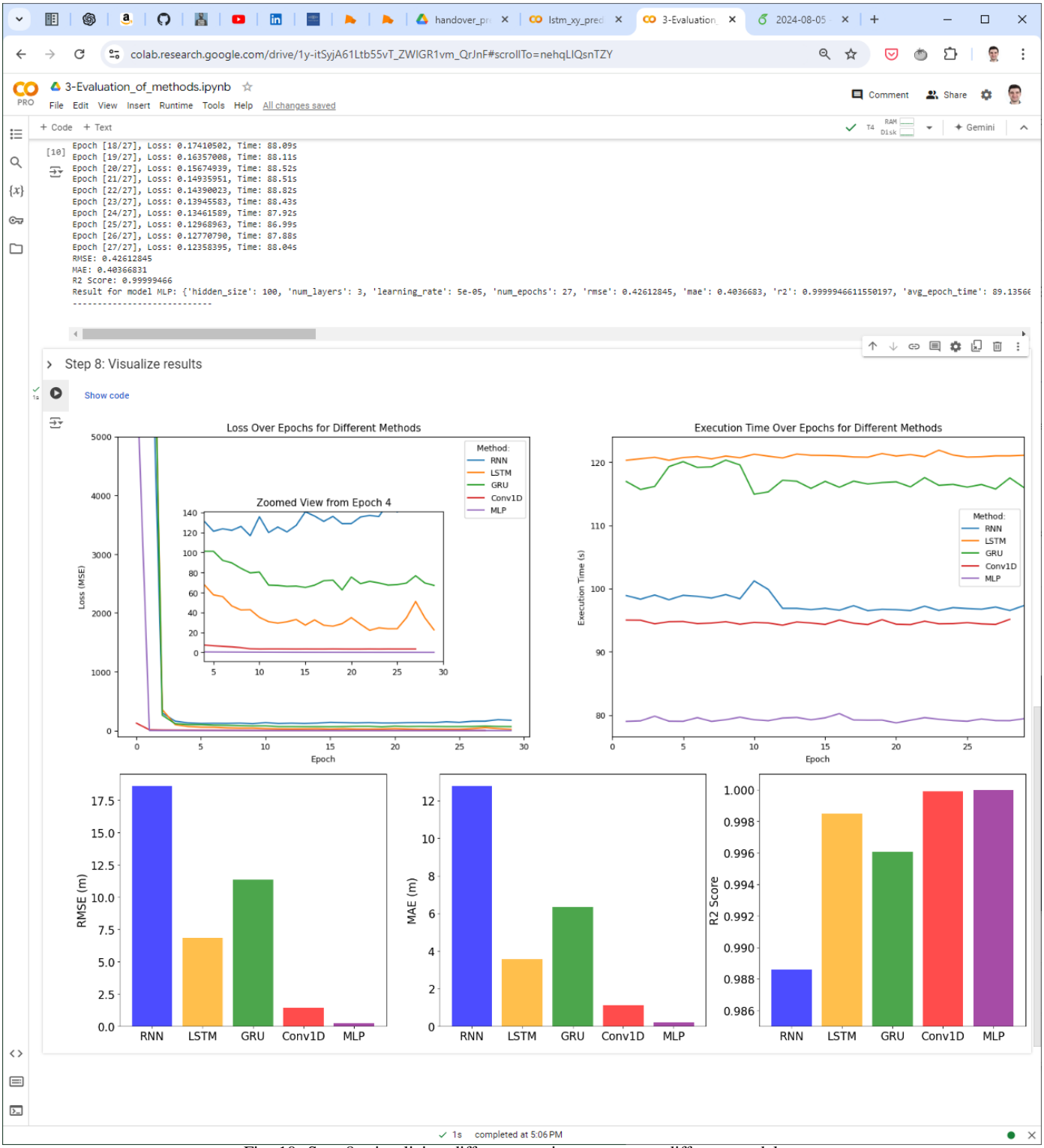


Fig. 10: Step 8: visualizing different metrics to compare different models.

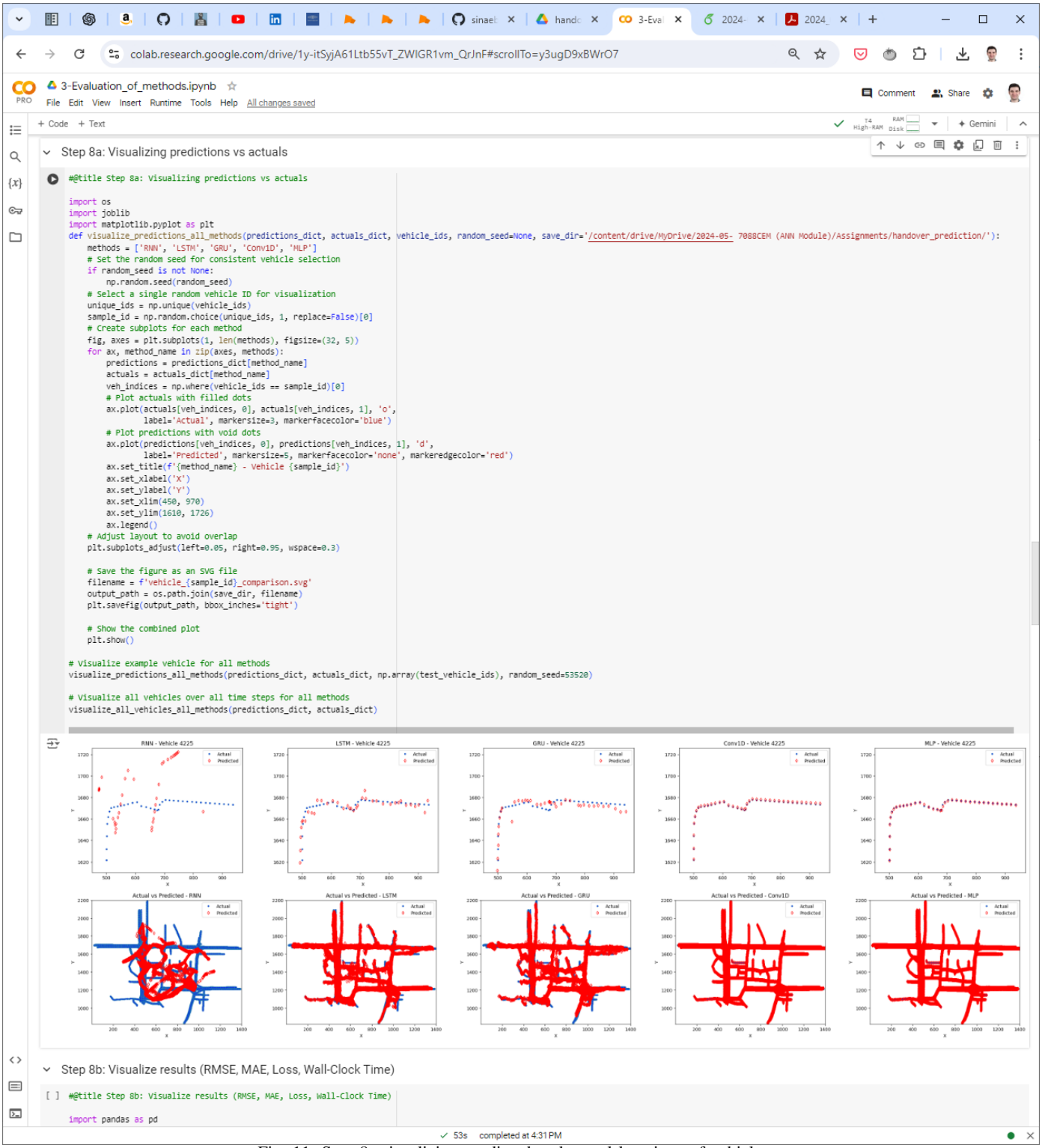


Fig. 11: Step 8: visualizing predicted and actual locations of vehicles.

APPENDIX B
CODE

One can find the code publicly on a Github repository. For future use of interested readers, the entire code—including three stages of data preprocessing, hyperparameter tuning, and evaluation of the several approaches—is conveniently compiled here.

A. Preprocessing

This section presents the steps for preprocessing the file `OpenStreetMap Trace for a Sparse Traffic.xml`.

1) Convert XML to CSV:

```
import xml.etree.ElementTree as ET
import csv

tree = ET.parse('OpenStreetMap Trace for a Sparse Traffic.xml')
root = tree.getroot()

# open a file for writing
with open('sparse.csv', 'w', newline='') as out_csv:
    writer = csv.writer(out_csv)
    writer.writerow(["time", "id", "x", "y", "angle", "type", "speed", "pos", "lane", "slope"])

    for timestep in root.findall('timestep'):
        time = timestep.get('time')
        for vehicle in timestep.findall('vehicle'):
            id = vehicle.get('id')
            x = vehicle.get('x')
            y = vehicle.get('y')
            angle = vehicle.get('angle')
            type = vehicle.get('type')
            speed = vehicle.get('speed')
            pos = vehicle.get('pos')
            lane = vehicle.get('lane')
            slope = vehicle.get('slope')
            writer.writerow([time, id, x, y, angle, type, speed, pos, lane, slope])
```

2) Read the input data from the CSV file

```
import pandas as pd

# from google.colab import drive
# drive.mount('/content/drive')

# Load data
data = pd.read_csv('sparse.csv')
# data = pd.read_csv('/content/drive/MyDrive/2024-05- 7088CEM (ANN
↪  Module)/Assignments/handover_prediction/sparse.csv')
data.head()
data.info()
data.describe()
```

3) Removing the unnecessary columns

```
# Convert the 'time' column to float and then to integer
data['time'] = data['time'].astype(float).astype(int)

# Remove 'veh' from 'id' column and convert to integer
data['id'] = data['id'].str.replace('veh', '').astype(int)

# Remove specified columns
columns_to_remove = ['type', 'lane', 'slope', 'pos']
data = data.drop(columns=columns_to_remove)

# Display the first 10 rows of the processed data
data.head(10)
data.info()
data.describe()

# Save the preprocessed DataFrame to a CSV file for future use
# output_path = '/content/drive/MyDrive/2024-05-7088CEM (ANN
↪  Module)/Assignments/handover_prediction/preprocessed_sparse.csv'
output_path = 'preprocessed_sparse.csv'
data.to_csv(output_path, index=False)
print(f"Preprocessed data saved to {output_path}")
```

4) Preparing the sequenced data In this method, we store a sequence of length 5 for each unique vehicle for the final dataset. This way, the learning algorithm would understand the patterns for the mobility of each vehicle and use it for further prediction. By grouping the vehicles, a total of 1,465,425 sequences were generated from the `preprocessed_sparse.csv` file.

```python
import pickle

sequence_length = 5

def create_sequences(data, sequence_length):
    sequences = [] # each seqence has the properties of one vehicle in different time steps (sequential)
    vehicle_ids = []
    for veh_id, group in data.groupby('id'):
        group = group.sort_values(by='time')
        for i in range(len(group) - sequence_length):
            seq = group.iloc[i:i+sequence_length]
            sequences.append((seq[['x', 'y', 'speed', 'angle']].values, seq[['x', 'y']].values[-1]))
            vehicle_ids.append(veh_id)
    return sequences, vehicle_ids

sequences, vehicle_ids = create_sequences(data, sequence_length)

# Save sequences to a pkl file
# with open('/content/drive/MyDrive/2024-05- 7088CEM (ANN
↪  Module)/Assignments/handover_prediction/sequences_vehicle_ids.pkl', 'wb') as f:
with open('sequences_vehicle_ids.pkl', 'wb') as f:
    pickle.dump((sequences, vehicle_ids), f)
```

5) Splitting the dataset into train and test datasets We split the data with 80-20% ratio between train and test datasets.

```python
import pickle

# Load sequences and vehicle IDs from a file
#with open('/content/drive/MyDrive/2024-05- 7088CEM (ANN
↪  Module)/Assignments/handover_prediction/sequences_vehicle_ids.pkl', 'rb') as f:
with open('sequences_vehicle_ids.pkl', 'rb') as f:
    sequences, vehicle_ids = pickle.load(f)

# Split data into training and testing
train_size = int(0.8 * len(sequences))
train_sequences = sequences[:train_size]
test_sequences = sequences[train_size:]
train_vehicle_ids = vehicle_ids[:train_size]
test_vehicle_ids = vehicle_ids[train_size:]

# Define Dataset class
from torch.utils.data import Dataset, DataLoader

class VehicleDataset(Dataset):
    def __init__(self, sequences, flatten=False):
        self.sequences = sequences
        self.flatten = flatten # new

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        seq, target = self.sequences[idx]
        seq_tensor = torch.Tensor(seq)
        if self.flatten:
            seq_tensor = seq_tensor.view(-1)  # Flatten the sequence
        return seq_tensor, torch.Tensor(target)

# Create DataLoader objects for the full dataset
train_dataset_all = VehicleDataset(train_sequences, flatten=False)
test_dataset_all = VehicleDataset(test_sequences, flatten=False)

train_loader_all = DataLoader(train_dataset_all, batch_size=32, shuffle=True)
test_loader_all = DataLoader(test_dataset_all, batch_size=32, shuffle=False)

# Split train dataset for hyperparameter tuning
tune_size = int(0.1 * len(train_sequences))  # Use 10% of the training data for tuning
tune_sequences = train_sequences[:tune_size]
tune_vehicle_ids = train_vehicle_ids[:tune_size]
```

```
44
45  tune_dataset = VehicleDataset(tune_sequences, flatten=False)
46  tune_test_size = int(0.1 * len(test_sequences))  # Use 10% of the testing data for tuning
47  tune_test_sequences = test_sequences[:tune_test_size]
48
49  tune_test_dataset = VehicleDataset(tune_test_sequences, flatten=False)
50
51  # Create DataLoader objects for tuning set
52  train_loader = DataLoader(tune_dataset, batch_size=32, shuffle=True)
53  test_loader = DataLoader(tune_test_dataset, batch_size=32, shuffle=False)
```

B. Hyperparameter Tuning

In this section, we define our models and hyperparameter grid to execute the experiments in order to choose the best hyperparameters for each method.

1) Import necessary libraries:

```
#@title Step 1: Importing the necessary libraries
import pandas as pd
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
import pickle
import time
import itertools
import os
```

2) Loading tuning train and test datasets: We use only 10% of the data of each datasets for hyperparameter tuning in order to reduce the computation time. For this step, just copy the code from step 5 of Sec. B-A. `train_loader` and `test_loader` will be used in the next steps.

3) Defining the methods: In this section, the architecture of each method and the activation functions between the NN layers are defined.

```
#@title Step 3: Defining the methods (architecture of NN layers and activation functions)

#### Step 3a: Define the Long Short-Term Memory (LSTM) model

import torch.nn as nn
import torch.nn.functional as F

class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out

#### Step 3b: Define the Gated Recurrent Unit (GRU) model

class GRUModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(GRUModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.gru(x, h0)
        out = self.fc(out[:, -1, :])
        return out


#### Step 3c: Define the generic Recurrent Neural Network (RNN) model

class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
```

```
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out


#### Step 3d: Define 1-d CNN (Conv1D) model

class Conv1DModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(Conv1DModel, self).__init__()
        self.conv_layers = nn.ModuleList()
        self.pool_layers = nn.ModuleList()

        # Input layer
        self.conv_layers.append(nn.Conv1d(input_size, hidden_size, kernel_size=3, padding=1))
        self.pool_layers.append(nn.MaxPool1d(kernel_size=2, stride=1))  # Reduced stride

        # Hidden layers
        for _ in range(1, num_layers):
            self.conv_layers.append(nn.Conv1d(hidden_size, hidden_size, kernel_size=3, padding=1))
            self.pool_layers.append(nn.MaxPool1d(kernel_size=2, stride=1))  # Reduced stride

        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.transpose(1, 2)  # Swap dimensions to fit Conv1d input format
        for conv, pool in zip(self.conv_layers, self.pool_layers):
            x = F.relu(conv(x))
            x = pool(x)
        x = x.mean(dim=2)  # Global average pooling
        x = self.fc(x)
        return x

#### Step 3e: Define a simple Multi-layer Perceptron (MLP) model

class MLPModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(MLPModel, self).__init__()
        layers = [nn.Linear(input_size, hidden_size), nn.ReLU()]

        for _ in range(1, num_layers):
            layers.extend([nn.Linear(hidden_size, hidden_size), nn.ReLU()])

        layers.append(nn.Linear(hidden_size, output_size))
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x.view(x.size(0), -1))
```

4) Training: This section provides a comprehensive approach for training the models. Additionally, a premature termination mechanism is established such that if the training loss does not decrease by more than 1% in four consecutive epochs, the execution will be halted to conserve computational resources.

```
#@title Step 4: Training method (generic for all baselines)

def train_model(model, train_loader, criterion, optimizer, num_epochs=10, device='cuda',
↪  early_stopping_rounds=4, min_delta=0.01):
    model.to(device)
    epoch_times = []
    epoch_losses = []

    for epoch in range(num_epochs):
        start_time = time.time()
        model.train()
        total_loss = 0
        for sequences, targets in train_loader:
            sequences, targets = sequences.to(device), targets.to(device)
            outputs = model(sequences)
            loss = criterion(outputs, targets)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        average_loss = total_loss / len(train_loader)
        epoch_time = time.time() - start_time
```

```
        epoch_times.append(epoch_time)
        epoch_losses.append(average_loss)
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {average_loss:.8f}, Time: {epoch_time:.2f}s')

        # Early stopping mechanism
        if epoch >= early_stopping_rounds:
          recent_losses = epoch_losses[-early_stopping_rounds:]
          if all(abs(recent_losses[i] - recent_losses[i-1]) < min_delta * recent_losses[i-1] for i in
          ↪  range(1, early_stopping_rounds)):
              print(f"Early stopping at epoch {epoch+1}")
              break

    return epoch_times, epoch_losses
```

5) Model evaluation: In this section, we evaluate the trained models with the test dataset with MAE, RMSE, and R^2 score metrics.

```
#@title Step 5: Model Evaluation

from sklearn.metrics import mean_absolute_error, r2_score

def evaluate_model(model, test_loader, device='cuda'):
    model.to(device)
    model.eval()
    predictions = []
    actuals = []

    with torch.no_grad():
        for sequences, targets in test_loader:
            sequences, targets = sequences.to(device), targets.to(device)
            outputs = model(sequences)
            predictions.extend(outputs.cpu().numpy())
            actuals.extend(targets.cpu().numpy())

    predictions = np.array(predictions)
    actuals = np.array(actuals)

    # Calculate the metrics
    rmse = np.sqrt(np.mean((predictions - actuals) ** 2))
    mae = mean_absolute_error(actuals, predictions)
    r2 = r2_score(actuals, predictions)

    print(f'RMSE: {rmse:.8f}')
    print(f'MAE: {mae:.8f}')
    print(f'R2 Score: {r2:.8f}')

    return predictions, actuals, rmse, mae, r2
```

6) Specify the general procedure for doing each experiment using a single set of hyperparameters: The MSE is selected as the loss function criterion, and the Adam optimizer is chosen for training optimization.

```
#@title Step 6: Define a method to run the training and evaluation of the model for different sets of
↪  hyperparameters

def run_experiment(params, input_size=4, output_size=2, model_type='LSTM',
↪  save_dir='/content/drive/MyDrive/2024-05- 7088CEM (ANN Module)/Assignments/handover_prediction/'):
    hidden_size, num_layers, learning_rate, num_epochs = params

    if model_type == 'LSTM':
        model = LSTMModel(input_size, hidden_size, num_layers, output_size)
        save_path = 'experiment_results_LSTM.csv'
    elif model_type == 'GRU':
        model = GRUModel(input_size, hidden_size, num_layers, output_size)
        save_path = 'experiment_results_GRU.csv'
    elif model_type == 'RNN':
        model = RNNModel(input_size, hidden_size, num_layers, output_size)
        save_path = 'experiment_results_RNN.csv'
    elif model_type == 'Conv1D':
        model = Conv1DModel(input_size, hidden_size, num_layers, output_size)
        save_path = 'experiment_results_Conv1D.csv'
    elif model_type == 'MLP':
        example_input, _ = next(iter(train_loader))
        flattened_input_size = example_input.view(example_input.size(0), -1).size(1)
        model = MLPModel(flattened_input_size, hidden_size, num_layers, output_size)
```

```
        save_path = 'experiment_results_MLP.csv'
    else:
        raise ValueError(f"Unsupported model type: {model_type}")

    # Initialize criterion and optimizer
    criterion = nn.MSELoss() # mean squared error (MSE) is the criterion for evaluating loss in each epoch
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) # Using Adam optimizer

    # Train the model and measure time
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    #%%%%%%% TRAIN %%%%%%
    # Use the provided train_model function
    epoch_times, epoch_losses = train_model(model, train_loader, criterion, optimizer,
    ↪  num_epochs=num_epochs, device=device)

    # Save the trained model
    model_save_path = os.path.join(save_dir, f'{model_type}_model.pth')
    torch.save(model.state_dict(), model_save_path)
    print(f"Saved {model_type} model to {model_save_path}")

    #%%%%%%% TEST %%%%%%
    # Evaluate the model
    predictions, actuals, rmse, mae, r2 = evaluate_model(model, test_loader, device=device)

    # Save predictions and actuals for each method
    data_save_path = os.path.join(save_dir, f'{model_type}_predictions_actuals.pkl')
    with open(data_save_path, 'wb') as f:
        joblib.dump((predictions, actuals), f)

    # Calculate average epoch time
    avg_epoch_time = np.mean(epoch_times)

    result = {
        'hidden_size': hidden_size,
        'num_layers': num_layers,
        'learning_rate': learning_rate,
        'num_epochs': num_epochs,
        'rmse': rmse,
        'mae': mae,
        'r2': r2,
        'avg_epoch_time': avg_epoch_time,
        'epoch_times': epoch_times,
        'epoch_losses': epoch_losses
    }

    results_df = pd.DataFrame([result])
    if save_path:
        if os.path.exists(save_path):
            results_df.to_csv(save_path, mode='a', header=False, index=False)
        else:
            results_df.to_csv(save_path, index=False)

    return result, predictions, actuals
```

7) Define hyperparameter grid and the generic method to run a specific experiment: The `run_specific_experiment` method allows for the execution of a specific set of hyperparameters, providing flexibility in case the execution is interrupted and needs to be resumed from a previous point. For instance, if the previous execution went up to the 25th set, we would opt to commence the execution of `run_specific_experiment` from the 26th index when we invoke it again.

```
#@title Step 7: Define hyperparameter sets

# Define the hyperparameter grid (2*3*8*1 = 48 total possibilities)
param_grid = {
    'hidden_size': [50, 100],
    'num_layers': [2, 3, 4],
    'learning_rate': [0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005],
    'num_epochs': [30]
}

# Create a list of hyperparameter combinations
param_combinations = list(itertools.product(
    param_grid['hidden_size'],
    param_grid['num_layers'],
```

```
    param_grid['learning_rate'],
    param_grid['num_epochs']
))

def run_specific_experiment(index, input_size=4, output_size=2, model_type='LSTM'): # allowed number for index: 0-47
    if index < 0 or index >= len(param_combinations):
        raise ValueError("Index out of range")

    params = param_combinations[index]

    print('model type: ', model_type)
    print('index of experiment / total number of experiments: ', str(index+1), ' / ',str(len(param_combinations)))
    print('Hyperparameters set (hidden_size, num_layers, learning_rate, num_epochs): ', params)
    result = run_experiment(params, input_size, output_size, model_type)
    print('---------')
    return result
```

8) Execute each method separately:

```
#@title Step 8: Execute each method (flexible for continuing unfinished experiments)

# We have 48 sets of experiments due to the different values for hyperparameters in param_grid (defined in Step 7)
# I wanted to run each method sequentially. You can run all of them in one loop (it does not matter).
for i in range(48):
    result = run_specific_experiment(index=i, model_type='LSTM')
for i in range(48):
    result = run_specific_experiment(index=i, model_type='GRU')
for i in range(48):
    result = run_specific_experiment(index=i, model_type='RNN')
for i in range(48):
    result = run_specific_experiment(index=i, model_type='Conv1D')
for i in range(48):
    result = run_specific_experiment(index=i, model_type='MLP')
```

9) Choose the best set of hyperparameters for each method: In this section, we rank sets of hyperparameters for each of the methods according to their RMSE, MAE, R^2 score, average epoch execution time, and loss slope (based on MSE).

```
# Methods to analyze
methods = ["GRU", "LSTM", "RNN", "Conv1D", "MLP"]

# Function to calculate the loss slope (rate of convergence)
def calculate_loss_slope(losses):
    if isinstance(losses, str):
        losses = eval(losses)  # Convert string representation of list back to list
    # Fit a linear model to the loss data (epochs vs loss)
    epochs = np.arange(1, len(losses) + 1)
    slope, _ = np.polyfit(epochs, losses, 1)
    return slope  # Slope of the loss curve

# Function to rank and select the top 3 experiments
def select_top_experiments(df, epoch_time_weight=0.1):
    # Rank based on RMSE (lower is better)
    df['rank_rmse'] = df['rmse'].rank(ascending=True)
    # Rank based on MAE (lower is better)
    df['rank_mae'] = df['mae'].rank(ascending=True)
    # Rank based on R2 (higher is better)
    df['rank_r2'] = df['r2'].rank(ascending=False)
    # Normalize avg_epoch_time to a 0-1 scale
    df['norm_epoch_time'] = (df['avg_epoch_time'] - df['avg_epoch_time'].min()) / (df['avg_epoch_time'].max() - df['avg_epoch_time'].min())
    # Apply a weight to the normalized epoch time
    df['weighted_epoch_time'] = df['norm_epoch_time'] * epoch_time_weight
    # Calculate and rank based on loss slope (more negative slope is better)
    df['loss_slope'] = df['epoch_losses'].apply(calculate_loss_slope)
    df['rank_loss_slope'] = df['loss_slope'].rank(ascending=True) / 10

    # Sum the ranks to get a combined score
    df['combined_rank'] = (df['rank_rmse'] + df['rank_mae'] + df['rank_r2'] +
                           df['weighted_epoch_time'] + df['rank_loss_slope'])

    # Sort by the combined rank and select the top 3
```

```
    top_experiments = df.sort_values('combined_rank').head(10)

    return top_experiments
# Dictionary to store results
results = {}

# Iterate over each method and process the corresponding file
for method in methods:
    filepath = f'experiment_results_{method}.csv'

    if os.path.exists(filepath):
        df = pd.read_csv(filepath)
        top_experiments = select_top_experiments(df)
        results[method] = top_experiments
    else:
        print(f"File {filepath} not found in the directory.")

# Display the results for each method
for method, top_experiments in results.items():
    print(f"Top 3 experiments for {method}:")
    display(top_experiments)
    print("\n")

# Saving the best results to new CSV files
for method, top_experiments in results.items():
    top_experiments.to_csv(f'top_3_experiments_{method}.csv', index=False)

```

10) Visualize results to evaluate different hyperparameters:

```
#@title Step 9: Visualize Results to evalate different hyperparameters

import matplotlib.pyplot as plt
import pandas as pd
from mpl_toolkits.axes_grid1.inset_locator import inset_axes

#%%% # Define methods and their specific zoom parameters
methods = {
    'RNN': {},
    'LSTM': {'xlim_rmse': (4.95e-4, 1.05e-3), 'ylim_rmse': (10, 60), 'xlim_r2': (4.95e-4, 1.05e-3),
    ↪  'ylim_r2': (0.86, 1)},
    'GRU': {'xlim_rmse': (4.95e-4, 1.05e-3), 'ylim_rmse': (10, 28), 'xlim_r2': (4.9e-4, 1.05e-3),
    ↪  'ylim_r2': (0.975, 0.997)},
    'Conv1D': {'xlim_rmse': (4.9e-5, 1.1e-2), 'ylim_rmse': (0.3, 2.1), 'xlim_r2': (4.9e-5, 1.1e-2),
    ↪  'ylim_r2': (0.99986, 1)},
    'MLP': {'xlim_rmse': (4.7e-5, 1.3e-2), 'ylim_rmse': (0.4, 2.3), 'xlim_r2': (4.7e-5, 1.3e-2),
    ↪  'ylim_r2': (0.99987, 1)},
}

#%%% # Set global font
plt.rcParams['font.family'] = 'Times New Roman'
plt.rcParams['font.size'] = 14

#%%% # Loop through methods
for method, zoom_params in methods.items():
    save_path = f'experiment_results_{method}.csv'
    results_df = pd.read_csv(save_path)

    #%%% #------------------- RMSE -------------------#

    # Plot RMSE for different hyperparameters
    plt.figure(figsize=(14, 7))
    for key, grp in results_df.groupby(['hidden_size', 'num_layers']):
        plt.plot(grp['learning_rate'], grp['rmse'], label=f"hidden_size={key[0]}, num_layers={key[1]}")
    plt.xlabel('Learning Rate')
    plt.ylabel('RMSE')
    plt.xscale('log')
    plt.title(f'RMSE for Different Hyperparameter Combinations ({method})')
    plt.legend()

    # Add inset of zoomed region if not RNN
    if method != 'RNN':
        ax_inset = inset_axes(plt.gca(), width='40%', height='27%', loc='center', borderpad=2)
        for key, grp in results_df.groupby(['hidden_size', 'num_layers']):
            ax_inset.plot(grp['learning_rate'], grp['rmse'], label=f"hidden_size={key[0]},
            ↪  num_layers={key[1]}")
```

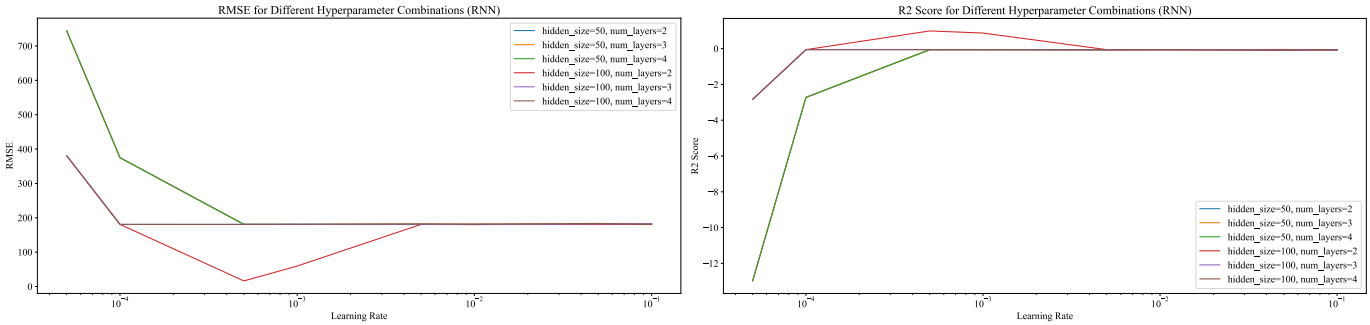
```
        ax_inset.set_xlim(zoom_params['xlim_rmse'])  # Set the limits for x-axis
        ax_inset.set_ylim(zoom_params['ylim_rmse'])  # Set the limits for y-axis
        ax_inset.set_xscale('log')
        ax_inset.set_title('Zoomed View')

    plt.subplots_adjust(left=0.05, right=0.95)
    plt.savefig(f'graphs/rmse-{method.lower()}.svg', bbox_inches='tight')
    plt.show()

    #%%% #------------------- R2 Score -------------------#

    # Plot R2 Score for different hyperparameters
    plt.figure(figsize=(14, 7))
    for key, grp in results_df.groupby(['hidden_size', 'num_layers']):
        plt.plot(grp['learning_rate'], grp['r2'], label=f"hidden_size={key[0]}, num_layers={key[1]}")
    plt.xlabel('Learning Rate')
    plt.ylabel('R2 Score')
    plt.xscale('log')
    plt.title(f'R2 Score for Different Hyperparameter Combinations ({method})')
    plt.legend()

    # Add inset of zoomed region if not RNN
    if method != 'RNN':
        ax_inset = inset_axes(plt.gca(), width='40%', height='27%', loc='center', borderpad=2)
        for key, grp in results_df.groupby(['hidden_size', 'num_layers']):
            ax_inset.plot(grp['learning_rate'], grp['r2'], label=f"hidden_size={key[0]},
            ↪  num_layers={key[1]}")
        ax_inset.set_xlim(zoom_params['xlim_r2'])  # Set the limits for x-axis
        ax_inset.set_ylim(zoom_params['ylim_r2'])  # Set the limits for y-axis
        ax_inset.set_xscale('log')
        ax_inset.set_title('Zoomed View')

    plt.subplots_adjust(left=0.05, right=0.95)
    plt.savefig(f'graphs/r2-{method.lower()}.svg', bbox_inches='tight')
    plt.show()

    #%%% #------------------- MAE -------------------#

    # # Plot MAE for different hyperparameters
    plt.figure(figsize=(14, 7))
    for key, grp in results_df.groupby(['hidden_size', 'num_layers']):
        plt.plot(grp['learning_rate'], grp['mae'], label=f"hidden_size={key[0]}, num_layers={key[1]}")
    plt.xlabel('Learning Rate')
    plt.ylabel('MAE')
    plt.xscale('log')
    plt.title(f'MAE for Different Hyperparameter Combinations ({method})')
    plt.legend()
    plt.subplots_adjust(left=0.05, right=0.95)
    plt.savefig(f'graphs/mae-{method.lower()}.svg', bbox_inches='tight')
    plt.show()
```

11) Analyzing the outcomes for each approach: Within this section, we will provide the RMSE and R^2 score outcomes for each method. Additionally, we will determine the optimal hyperparameter set for each method.

a) RNN: Fig. 12a displays the RMSE metric for the RNN approach using various hyperparameters. The results indicate that selecting a learning rate (α) of 0.0005 and employing 2 hidden layers, each containing 100 neurons, yields the minimum RMSE value. Furthermore, Fig. 12b shows the coefficient of determination (R^2 score) for the RNN technique using various hyperparameters. The findings indicate that employing a RNN with two hidden layers and 100 neurons yields favorable outcomes when utilizing 0.0005 and 0.001 as α. Hence, the hyperparameter set for the RNN method that is deemed optimal is $\alpha = 0.0005$ with 2 hidden layers, each consisting of 100 neurons.

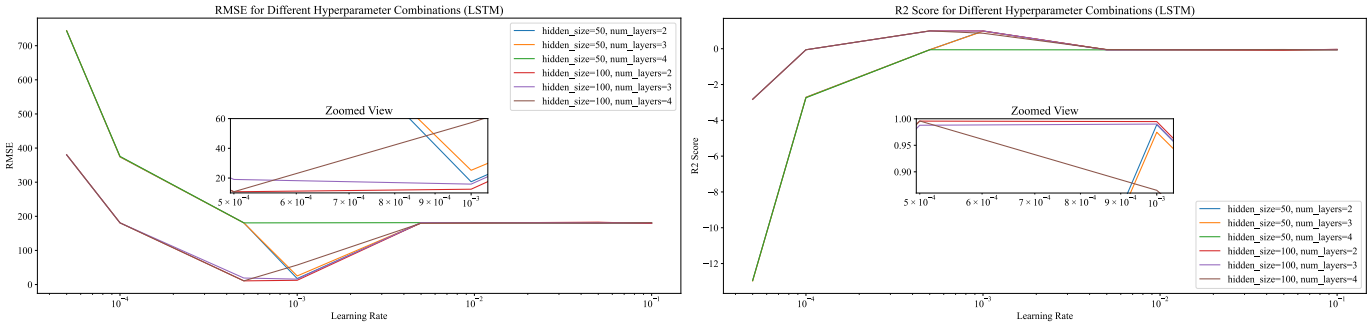
b) LSTM: Fig. 13a illustrates the RMSE metric for the LSTM approach with different hyperparameters. The findings suggest that choosing 0.0005 and 0.001 as the values for α and utilizing either 2 hidden layers, each consisting of 100 neurons, results in the lowest RMSE value. Furthermore, Fig. 13b displays the R^2 score of the LSTM technique using various hyperparameters. The findings indicate that using a LSTM model with two hidden layers and 100 neurons leads to favorable results when using 0.0005 as α. Therefore, the most effective hyperparameter configurations for the LSTM technique includes two layers, with each layer comprising 100 neurons, and a α value of 0.0005.

c) GRU: Fig. 14a displays the RMSE metric for the GRU method across different hyperparameters. The results suggest that selecting 0.0005 as the value of α and using 3 hidden layers, each comprising of 100 neurons, yields the minimum RMSE value. Moreover, Fig. 14b illustrates the R^2 score for the GRU technique with various hyperparameters. The results suggest that using a GRU with three hidden layers and 100 neurons produces acceptable results when α is set to 0.0005. Therefore,



(a) (b)

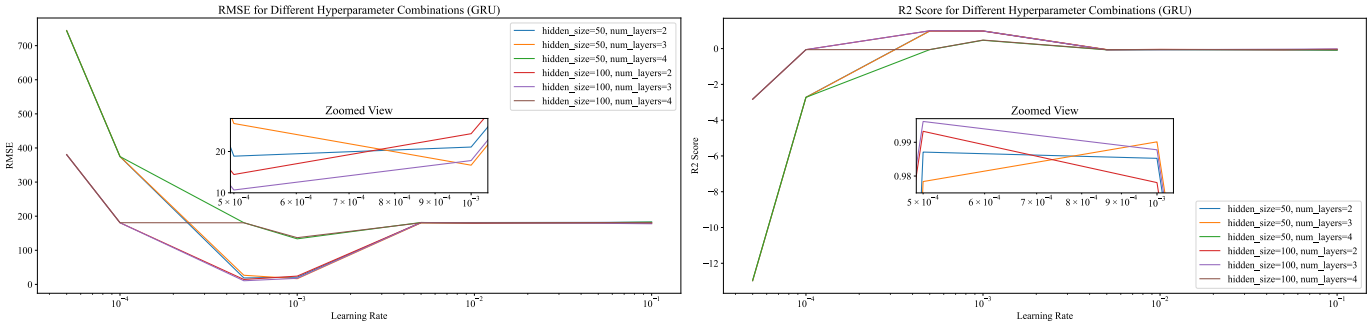
Fig. 12: a) RMSE and b) R^2 score of the RNN method with different hyperparameters.



(a) (b)

Fig. 13: a) RMSE and b) R^2 score of the LSTM method with different hyperparameters.

the hyperparameter configuration considered to be the best for the GRU method is $\alpha = 0.0005$ with 3 hidden layers, each containing 100 neurons.

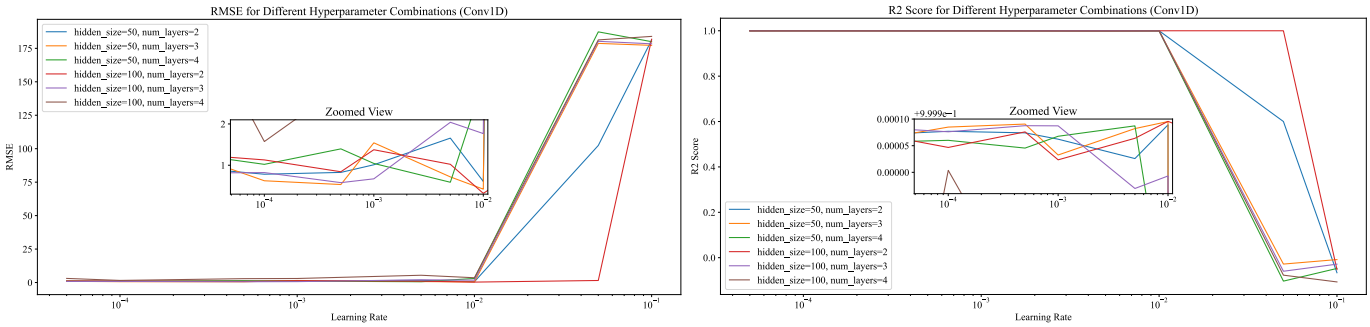


(a) (b)

Fig. 14: a) RMSE and b) R^2 score of the GRU method with different hyperparameters.

d) Conv1D: Fig. 15a displays the RMSE metric for the Conv1D method, showcasing its performance across various hyperparameters. The findings indicate that selecting any combination of layers and neurons can yield positive outcomes within the range of $0.00005 \leq \alpha \leq 0.01$. However, despite the RMSE being fairly similar within this range, the smallest RMSE is observed when α is set to 0.0005, and there are three hidden layers, each consisting of 50 neurons. Furthermore, Fig. 15b illustrates the R^2 score for the Conv1D technique using different hyperparameters. These results also indicate the same conclusion as the RMSE metric, although the experiments (hidden size = 100, no. of layers = 3, α = 0.001), (hidden size = 50, no. of layers = 3, α = 0.01), (hidden size = 100, no. of layers = 2, α = 0.01), and (hidden size = 50, no. of layers = 4, α = 0.005) are nearly identical to the selected set using the RMSE metric. Therefore, we employ a method that utilizes three hidden layers, each consisting of 50 neurons, and a learning rate of $\alpha = 0.0005$.

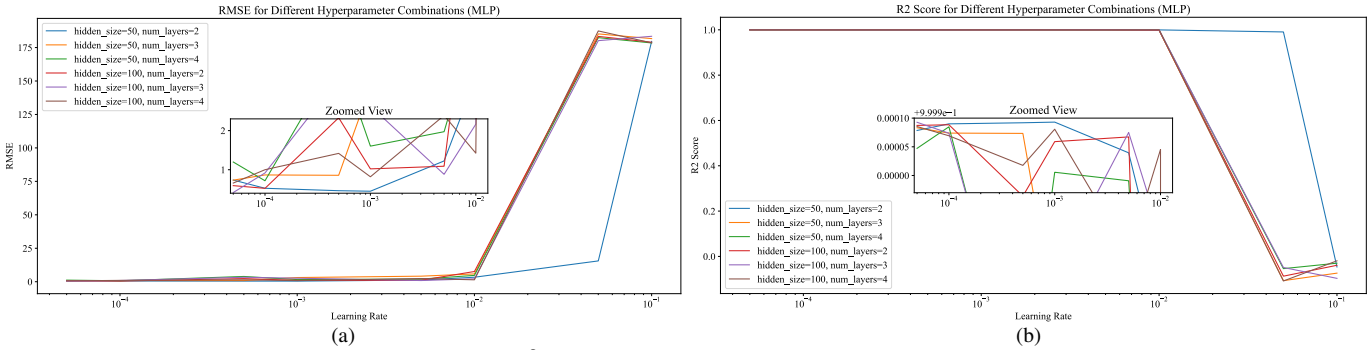
e) MLP: Fig. 16a illustrates the RMSE metric for the MLP method across several hyperparameters. The findings indicate that favorable results can be attained by selecting any configuration of layers and neurons within the specified range of $0.00005 \leq \alpha \leq 0.01$. However, upon closer examination, it is evident that the optimal RMSE may be attained by utilizing only 2 layers, with each layer consisting of 50 neurons, when α is set to 0.001. Furthermore, Fig. 16b displays the R^2 score for the MLP technique with various hyperparameters. These results support the identical conclusion as the RMSE metric. Hence, a learning rate of 0.001, along with 2 hidden layers and 50 neurons, is considered acceptable. However, other configurations,



(a) (b)

Fig. 15: a) RMSE and b) R^2 score of the Conv1D method with different hyperparameters.

such as those with learning rates of 0.00005 and 0.0001, regardless of their layer count and neuron size, are also very similar to the chosen hyperparameter set.



(a) (b)

Fig. 16: a) RMSE and b) R^2 score of the MLP method with different hyperparameters.

C. Evaluation of the Various Methods

In this section, we train the models with 80% of the dataset and evaluate them by testing 20% of the data. We use the best hyperparameter set for each of the methods using the ranking in Sec. B-B (see the green rows in Table II).

1) Import necessary libraries:

```
#@title Step 1: Importing the necessary libraries
import pandas as pd
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
import pickle
import time
import itertools
import os
```

2) Load preprocessed (sequenced) data; Splitting data into training and testing datasets

```
#@title Step 2: Load preprocessed (sequenced) data; Splitting data into training and testing datasets

# Load sequences and vehicle IDs from a file
with open('sequences_vehicle_ids.pkl', 'rb') as f:
#with open('sequences_vehicle_ids.pkl', 'rb') as f:
    sequences, vehicle_ids = pickle.load(f)

# Split data into training and testing
train_size = int(0.8 * len(sequences))
train_sequences = sequences[:train_size]
test_sequences = sequences[train_size:]
train_vehicle_ids = vehicle_ids[:train_size]
test_vehicle_ids = vehicle_ids[train_size:]

class VehicleDataset(Dataset):
    def __init__(self, sequences, flatten=False):
        self.sequences = sequences
        self.flatten = flatten # new

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        seq, target = self.sequences[idx]
        seq_tensor = torch.Tensor(seq)
        if self.flatten:
            seq_tensor = seq_tensor.view(-1)  # Flatten the sequence
        return seq_tensor, torch.Tensor(target)

# Create DataLoader objects for the full dataset
train_dataset_all = VehicleDataset(train_sequences, flatten=False)
test_dataset_all = VehicleDataset(test_sequences, flatten=False)

train_loader = DataLoader(train_dataset_all, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset_all, batch_size=32, shuffle=False)
```

3) Defining the methods (models): We replicate the exact code from step 3 in Sec. B-B.

4) Train the model: We replicate the exact code from step 4 in Sec. B-B.

5) Model evaluation: We replicate the exact code from step 5 in Sec. B-B.

6) Specify the general procedure for doing each experiment using a single set of hyperparameters: We replicate the exact code from step 6 in Sec. B-B.

7) Execute each method separately for the selected hyperparameters:

```
#@title Step 7: Execute the experiment for each model type with the selected hyperparameters

# set of hyperparameters for each model type consists of (hidden_size, num_layers, learning_rate,
↪  num_epochs)
hyperparameters = {
    'RNN': (100, 2, 0.0005, 10),
    'LSTM': (100, 4, 0.0005, 30),
    'GRU': (100, 3, 0.0005, 15),
    'Conv1D': (100, 2, 0.01, 10),
    'MLP': (100, 3, 0.00005, 10)
}
```

```

# Create dictionaries to store predictions and actuals for each model
predictions_dict = {}
actuals_dict = {}

# Loop through each model type and run the experiment with the corresponding hyperparameters
for model_type, params in hyperparameters.items():
    print(f"Running experiment for model {model_type} with parameters {params}")
    result, predictions, actuals = run_experiment(params, model_type=model_type)

    # Store the predictions, actuals, and model save path
    predictions_dict[model_type] = predictions
    actuals_dict[model_type] = actuals

    print(f"Result for model {model_type}: {result}\n------------------------------\n")
```

8) Visualize episodic loss and execution time of the methods, along with RMSE, MAE, and R^2 score of their final epoch:

```
import pandas as pd
import matplotlib.pyplot as plt
import ast
from mpl_toolkits.axes_grid1.inset_locator import inset_axes

# Load the CSV file
file_path = 'experiment_methods.csv'
df = pd.read_csv(file_path)

# Convert the 'epoch_losses' column from a string representation of a list to an actual list
df['epoch_losses'] = df['epoch_losses'].apply(ast.literal_eval)

# Set global font to Times New Roman and size to 18
plt.rcParams['font.family'] = 'Times New Roman'
plt.rcParams['font.size'] = 18

# Main plot
plt.figure(figsize=(12, 8))
for index, row in df.iterrows():
    plt.plot(row['epoch_losses'], label=row['method_type'])

plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.ylim(-100, 5000)
plt.title('Loss Over Epochs for Different Methods')
plt.legend(title="Method: ", prop={'size': 18})

# Add inset of zoomed region
ax_inset = inset_axes(plt.gca(), width='58%', height='50%', loc='center', borderpad=6)
for index, row in df.iterrows():
    # Starting from epoch 4
    ax_inset.plot(range(4, len(row['epoch_losses'])), row['epoch_losses'][4:], label=row['method_type'])

ax_inset.set_xlim(4, len(df.iloc[0]['epoch_losses']) )
ax_inset.set_ylim(-9, 141)
ax_inset.set_title('Zoomed View from Epoch 4')

plt.savefig('graphs/loss-over-epochs.svg', bbox_inches='tight')
plt.show()

#%%% Execution time:

# Manually extracted execution times from notebook output (from the printed output (it wasn't in the CSV
↪  files (only the avg exec time is in the CSV output)))
execution_times = {
    'RNN': [98.89, 98.32, 98.98, 98.23, 98.93, 98.77, 98.50, 99.05, 98.38, 101.21,
            99.85, 96.87, 96.87, 96.67, 96.88, 96.57, 97.29, 96.50, 96.72, 96.66,
            96.51, 97.21, 96.54, 96.98, 96.84, 96.74, 97.07, 96.53, 97.31, 96.26],
    'LSTM': [120.30, 120.53, 120.76, 120.30, 120.72, 120.87, 120.51, 120.96, 120.69, 121.25,
             120.94, 120.66, 121.28, 121.08, 121.06, 120.99, 120.82, 120.78, 121.36, 120.96,
             121.19, 120.86, 121.89, 121.12, 120.80, 120.85, 120.99, 120.99, 121.08, 120.71],
    'GRU': [116.93, 115.68, 116.15, 119.29, 120.07, 119.16, 119.26, 120.32, 119.55, 114.92,
            115.30, 117.13, 116.97, 115.84, 116.96, 116.02, 116.99, 116.54, 116.77, 116.88,
            116.08, 117.55, 116.33, 116.48, 116.04, 116.46, 115.74, 117.50, 115.96, 116.76],
    'Conv1D': [95.01, 94.99, 94.42, 94.76, 94.79, 94.45, 94.54, 94.76, 94.36, 94.65,
               94.55, 94.21, 94.72, 94.55, 94.33, 95.02, 94.52, 94.31, 95.07, 94.36,
```

```
                94.30, 94.84, 94.42, 94.45, 94.61, 94.41, 94.33, 95.13],
    'MLP': [78.99, 79.07, 79.81, 79.02, 79.00, 79.58, 78.99, 79.25, 79.65, 79.25,
            79.09, 79.52, 79.61, 79.24, 79.55, 80.22, 79.21, 79.18, 79.19, 78.76,
            79.17, 79.58, 79.33, 79.12, 79.01, 79.37, 79.11, 79.10, 79.42, 78.87]
}

# Plotting Execution Time for Each Epoch
plt.figure(figsize=(14, 8))

for method, times in execution_times.items():
    plt.plot(range(1, len(times) + 1), times, label=method)

plt.xlabel('Epoch')
plt.xlim(1, 30)
plt.ylabel('Execution Time (s)')
plt.title('Execution Time Over Epochs for Different Methods')
plt.legend(title="Method:", prop={'size': 16}, loc='center right', bbox_to_anchor=(1, 0.63))
# plt.grid(True)
plt.savefig('graphs/exec-time-per-epoch.svg')
plt.show()

#%% RMSE, MAE, and R2 score of the final epoch

# Load the CSV file
file_path = 'experiment_methods.csv'
df = pd.read_csv(file_path)

# Extract the metrics
methods = df['method_type']
rmse_values = df['rmse']
mae_values = df['mae']
r2_values = df['r2']

# Define the original color mapping
color_mapping = {
    'RNN': 'blue',
    'LSTM': 'orange',
    'GRU': 'green',
    'Conv1D': 'red',
    'MLP': 'purple'
}

# Lighten the colors
def lighten_color(color, amount=0.7):
    return to_rgba(color, alpha=amount)

# Apply lighter colors
light_colors = [lighten_color(color_mapping[method], amount=0.7) for method in methods]

# Plotting RMSE, MAE, R2 for each method as a bar plot
plt.rcParams['font.size'] = 17
fig, ax = plt.subplots(1, 3, figsize=(25, 7))

ax[0].bar(methods, rmse_values, color=light_colors)
ax[0].set_ylabel('RMSE (m)')

ax[1].bar(methods, mae_values, color=light_colors)
ax[1].set_ylabel('MAE (m)')

ax[2].bar(methods, r2_values, color=light_colors)
ax[2].set_ylabel('R2 Score')
ax[2].set_ylim(0.985, 1.001)  # Limit the y-axis from 0.985 to 1.001

plt.savefig('graphs/rmse-mae-r2score-methods.svg')
plt.show()
```

9) Visualizing predictions vs actuals

```
#@title Step 8a: Visualizing predictions vs actuals

import os
import joblib
import matplotlib.pyplot as plt
import numpy as np

```

```
def load_predictions_actuals(method_name, save_dir):
    data_save_path = os.path.join(save_dir, f'{method_name}_predictions_actuals.pkl')
    if os.path.exists(data_save_path):
        with open(data_save_path, 'rb') as f:
            predictions, actuals = joblib.load(f)
        return predictions, actuals
    else:
        raise FileNotFoundError(f"Data for {method_name} not found in {save_dir}.")

def visualize_predictions_all_methods(predictions_dict, actuals_dict, vehicle_ids, random_seed=None,
↪  save_dir='/content/drive/MyDrive/2024-05- 7088CEM (ANN Module)/Assignments/handover_prediction/'):
    methods = ['RNN', 'LSTM', 'GRU', 'Conv1D', 'MLP']

    # Load data if predictions_dict and actuals_dict are empty
    if not predictions_dict or not actuals_dict:
        for method in methods:
            try:
                predictions_dict[method], actuals_dict[method] = load_predictions_actuals(method,
                ↪  save_dir)
            except FileNotFoundError as e:
                print(e)
                return

    # Set the random seed for consistent vehicle selection
    if random_seed is not None:
        np.random.seed(random_seed)

    # Select a single random vehicle ID for visualization
    unique_ids = np.unique(vehicle_ids)
    sample_id = np.random.choice(unique_ids, 1, replace=False)[0]

    # Create subplots for each method
    fig, axes = plt.subplots(1, len(methods), figsize=(25, 5))

    for ax, method_name in zip(axes, methods):
        predictions = predictions_dict[method_name]
        actuals = actuals_dict[method_name]

        veh_indices = np.where(vehicle_ids == sample_id)[0]

        # Plot actuals with filled dots
        ax.plot(actuals[veh_indices, 0], actuals[veh_indices, 1], 'o',
                label='Actual', markersize=3, markerfacecolor='blue')

        # Plot predictions with void dots
        ax.plot(predictions[veh_indices, 0], predictions[veh_indices, 1], 'd',
                label='Predicted', markersize=5, markerfacecolor='none', markeredgecolor='red')

        ax.set_title(f'{method_name} - Vehicle {sample_id}')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_xlim(470, 700)
        ax.set_ylim(840, 1670)
        ax.legend()

    # Adjust layout to avoid overlap
    plt.subplots_adjust(left=0.05, right=0.95, wspace=0.3)

    # Save the figure as an SVG file
    filename = f'vehicle_{sample_id}_comparison.svg'
    output_path = os.path.join(save_dir, filename)
    plt.savefig(output_path, bbox_inches='tight')

    # Show the combined plot
    plt.show()

def visualize_all_vehicles_all_methods(predictions_dict, actuals_dict,
↪  save_dir='/content/drive/MyDrive/2024-05- 7088CEM (ANN Module)/Assignments/handover_prediction/'):
    methods = ['RNN', 'LSTM', 'GRU', 'Conv1D', 'MLP']

    # Load data if predictions_dict and actuals_dict are empty
    if not predictions_dict or not actuals_dict:
        for method in methods:
            try:
                predictions_dict[method], actuals_dict[method] = load_predictions_actuals(method,
                ↪  save_dir)
```

```
            except FileNotFoundError as e:
                print(e)
                return

    # Create subplots for each method
    fig, axes = plt.subplots(1, len(methods), figsize=(20, 5))

    for ax, method_name in zip(axes, methods):
        predictions = predictions_dict[method_name]
        actuals = actuals_dict[method_name]

        # Plot actuals with filled dots
        ax.plot(actuals[:, 0], actuals[:, 1], 'o',
                label='Actual', markersize=3, markerfacecolor='blue')

        # Plot predictions with void dots
        ax.plot(predictions[:, 0], predictions[:, 1], 'd',
                label='Predicted', markersize=5, markerfacecolor='none', markeredgecolor='red')

        ax.set_title(f'Actual vs Predicted - {method_name}')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_xlim(2, 1400)
        ax.set_ylim(820, 2200)
        ax.legend()

    # Adjust layout to avoid overlap
    plt.subplots_adjust(left=0.05, right=0.95, wspace=0.3)

    # Save the figure as an SVG file
    filename = 'all_vehicles_comparison.svg'
    output_path = os.path.join(save_dir, filename)
    plt.savefig(output_path, bbox_inches='tight', dpi=100)

    # Show the combined plot
    plt.show()

# Visualize example vehicle for all methods
visualize_predictions_all_methods(predictions_dict, actuals_dict, np.array(test_vehicle_ids),
↪  random_seed=3520)

# Visualize all vehicles over all time steps for all methods
visualize_all_vehicles_all_methods(predictions_dict, actuals_dict)
```