Privacy-Preserving Traffic Flow Prediction using Federated Learning: A Simplified Implementation of FedGRU Algorithm

Girik Khullar, CS21B2003

I. AIM

Our project aims to replicate and implement the FedGRU algorithm described in the paper [1] for privacy-preserving traffic flow prediction using federated learning. By following the methodology outlined in the paper [1], we aim to demonstrate the feasibility and effectiveness of this approach. The implementation involves combining federated learning with the gated recurrent unit (GRU) neural network, enabling organizations to collaboratively train a global model without sharing raw data to ensure data privacy. Through this implementation, we seek to validate the accuracy of traffic flow prediction while maintaining privacy within the federated learning framework. Our objective is to reproduce the findings of the paper and provide a practical implementation of the FedGRU algorithm for accurately predicting traffic flow in urban areas while safeguarding sensitive information.

II. Apparatus/Software Required

The apparatus required for this project includes a computer equipped with a decent CPU, such as an Intel Core i7-10750H, which serves as the primary computational resource for implementing the algorithm. Optionally, a GPU, such as the GTX 1650Ti, can be utilized for faster training of machine learning models. Additionally, software components include Python as the programming language for algorithm implementation, along with an integrated development environment (IDE) like VSCode or Jupyter Notebooks for code development. Essential machine learning libraries such as PyTorch, NumPy, Pandas, Matplotlib and scikit-learn are utilized for model training and data processing. Access to the internet is necessary for acquiring datasets and reference materials, while presentation software like Google Slides is employed for creating project presentation.

III. Procedure

A. Introduction

The project is based on a paper [1] that addresses the pressing challenge of predicting traffic flow accurately while safeguarding data privacy. In contemporary urban settings, various stakeholders, including residents, taxi drivers, businesses, and government agencies, rely on timely traffic flow information to optimize transportation systems and alleviate congestion. Traffic flow prediction (TFP) is to provide such traffic flow information by using the historical traffic flow data to predict future traffic flow. However, conventional centralized machine learning methods often encounter privacy concerns when dealing with sensitive data. This is because these learning methods typically require sharing data among public agencies and private companies. Also, what is often overlooked is that the data may contain private information which is a potential source of data leakage.

To tackle this issue, the paper [1] uses federated learning (FL), a privacy-preserving technique that enables collaborative model training without sharing raw data. The proposed approach combines FL with the gated recurrent unit (GRU) neural network, known as the FedGRU algorithm, to predict traffic flow while maintaining data privacy. By leveraging FL, organizations can collectively train a global model using their local data, thus mitigating privacy risks associated with data sharing.

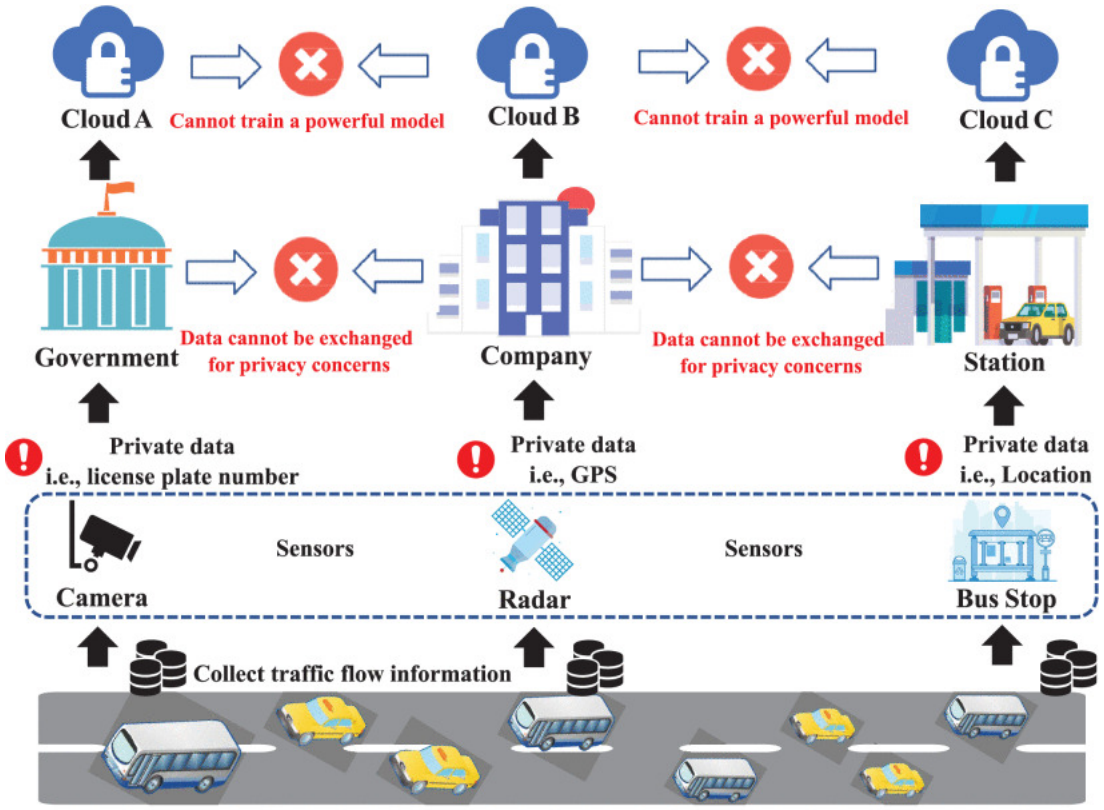

Fig. 1. Privacy and security problems in TFP. (Taken from the paper [1])

B. Methodology/Problem Statement

We have implemented the FedGRU algorithm for privacy-preserving traffic flow prediction using federated learning and we compare it to the standard GRU based traffic flow prediction method that combines all the data gathered from the sensors in the cloud for training the model [1]. The traditional centralized learning methods consist of three main steps: data processing, data fusion, and model building. However, these methods face challenges in sharing sensitive information, especially in scenarios like traffic prediction where privacy concerns are paramount.

To address these challenges, federated learning (FL) is introduced, allowing organizations to collaboratively train a global model without sharing raw data. The proposed FedGRU algorithm combines FL with the gated recurrent unit (GRU) neural network, enabling accurate traffic flow prediction while maintaining data privacy. The following key components are used:

1) Federated Learning and Gated Recurrent Unit (GRU): FL is a distributed ML paradigm designed to train models without compromising privacy. Federated learning is a decentralized machine learning approach that enables training models across multiple devices or servers holding local data samples without exchanging them. Instead of centralizing data for training, federated learning allows models to be trained directly on data stored locally on these devices. Each device independently trains a global model on its local dataset and sends only model updates to a central server or aggregator. These updates are aggregated to update the global model, which becomes the starting point for the next round of training. Federated learning preserves data privacy by keeping raw data on local devices, reduces communication overhead by exchanging only model updates, and enables decentralized learning across distributed data sources. In this project, we utilize FL to train a single, globally predictive model from databases stored in multiple organizations. Specifically, we integrate FL with the gated recurrent unit (GRU) neural network, leveraging its effectiveness in handling time-series data for traffic flow prediction.

2) Privacy-Preserving Traffic Flow Prediction Algorithm: The FedGRU algorithm is developed to address privacy concerns in centralized learning methods. This algorithm incorporates the FedAVG mechanism for secure parameter aggregation, reducing communication overhead and preserving data privacy during model training.

3) Federated-Learning-Based Gated Recurrent Unit Neural Network Algorithm: The FedGRU algorithm follows a series of steps, including model initialization, distribution to organizations, local model training, parameter updates, and global model aggregation. By leveraging FL and GRU, FedGRU achieves accurate and timely traffic flow prediction while maintaining privacy.

Overall, the methodology involves implementing and fine-tuning the FedGRU algorithm to suit the specific requirements of traffic flow prediction in urban areas. Through this approach, we aim to demonstrate the feasibility and effectiveness of privacy-preserving traffic flow prediction using federated learning.

C. Implementation

1) Hardware Design: We have not used any specialized hardware like IoT sensors for data collection purposes or used Edge Devices to simulate clients for the distributed FL system. We have simulated all this on our local laptop and do all the training on a GTX 1650Ti GPU. The way this works is that we split the dataset into n number of clients and then we train individual models, all separately for each client dataset [1]. This simulates the way sensors collect data at each client and then process them on the local edge device. Once this processing is done on each individual model at each client, the weights of the models are communicated to the cloud, which in our case is another model. We then use the FedAvg algorithm to train the global model. And this step repeats after we send this model, trained collectively on all the clients back to all clients for updation.

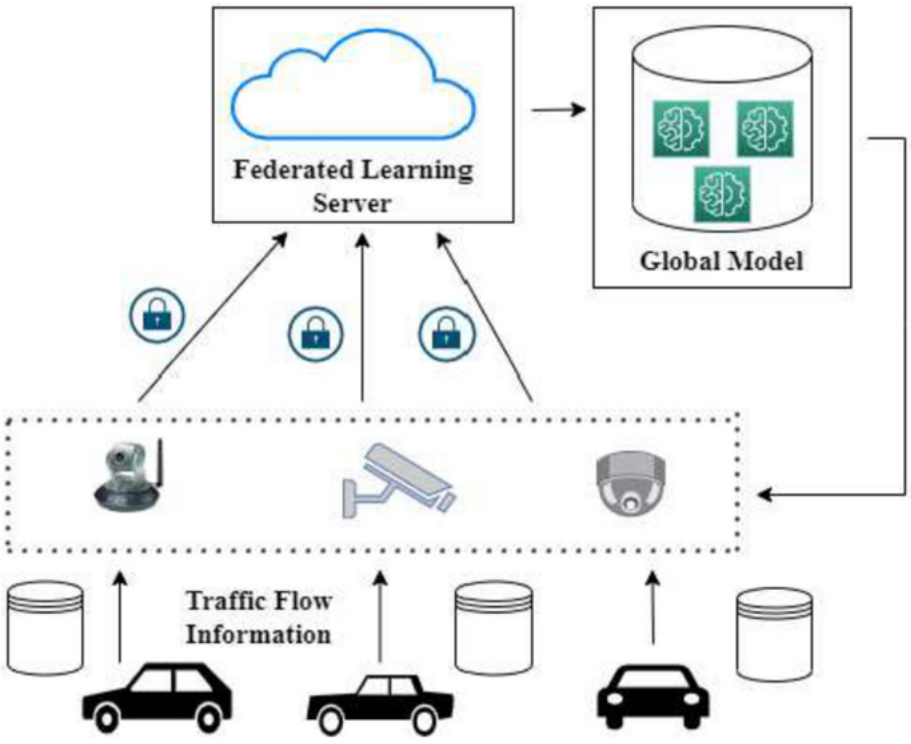

Fig. 2. Data Flow pipeline simulated on the laptop. Each sensor independently collects data on which the local models are trained and then those weights are shared to the learning server. The server utilises those weights to train a global model which is then passed on to the clients.This image was taken from [2]

2) Executable Code: Here is the code for the FedAVG algorithm.

```
def federated_average(weights_list):
```

```
    """
    Perform federated averaging of model weights.

    Args:
        weights_list (list): List of dictionaries containing model weights.

    Returns:
        dict: A dictionary containing the averaged model weights.
    """
    # Deep copy the weights of the first model
    averaged_weights = deepcopy(weights_list[0])

    # Iterate over the keys of the model weights
    for key in averaged_weights.keys():
        # Sum up the corresponding weights from all models
        for i in range(1, len(weights_list)):
            averaged_weights[key] += weights_list[i][key]

        # Calculate the average by dividing by the number of models
        averaged_weights[key] = torch.div(averaged_weights[key], len(weights_list))

    return averaged_weights
```

Here is the code for the dataset creation for each client which is formed in iid manner.

```
def generate_iid_data_indices(dataset, num_users):
    """
    Generate indices for IID (independent and identically distributed) data partitioning.

    Args:
        dataset (list or torch.utils.data.Dataset): The dataset to partition.
        num_users (int): The number of users (clients) to generate data indices for.

    Returns:
        dict: A dictionary where keys are user IDs and values are sets of data indices.
    """
    num_items_per_user = int(len(dataset) / num_users)
    user_data_indices = {}  # Dictionary to store data indices for each user
    all_indices = [i for i in range(len(dataset))]  # List of all indices in the dataset

    # Iterate over the number of users to generate data indices for each user
    for user_id in range(num_users):
        # Randomly select indices for the current user without replacement
        user_data_indices[user_id] = set(np.random.choice(all_indices, num_items_per_user,
    replace=False))

        # Remove the selected indices from the list of all indices
        all_indices = list(set(all_indices) - user_data_indices[user_id])

    return user_data_indices

def generate_client_datasets(dataset, user_data_indices, batch_size=128):
    """
    Generate DataLoader objects for each client based on their data indices.

    Args:
        dataset (list or torch.utils.data.Dataset): The dataset to create client datasets from.
        user_data_indices (dict): A dictionary where keys are user IDs and values are sets of
    data indices.
        batch_size (int, optional): The batch size for DataLoader objects. Defaults to 128.

    Returns:
        dict: A dictionary where keys are user IDs and values are DataLoader objects for each
    client.
    """
    datasets = {}  # Dictionary to store DataLoader objects for each client
```

```
    for user_id in range(len(user_data_indices.keys())):
        # Create DataLoader for the current client's data using DatasetSplit
        datasets[user_id] = DataLoader(DatasetSplit(dataset, user_data_indices[user_id]),
    batch_size=batch_size, shuffle=True)

    return datasets
```

And here is the main training loop of the whole algorithm.

```
net_glob.train()
w_glob = net_glob.state_dict()

learning_rate = 0.001
loss_function = nn.MSELoss()

# Determine which clients are participating based on the specified criteria
if all_clients_participate:
    # If all clients participate, initialize local model weights for each client
    local_model_weights = [net_glob.state_dict() for _ in range(num_clients)]
else:
    # If only a fraction of clients participate, randomly select participating clients
    num_participating_clients = max(int(fraction_of_clients_participating * num_clients), 1)  # Ensure at least 1 client participates
    participating_clients = np.random.choice(range(num_clients), num_participating_clients, replace=False)

local_models = [w_glob for i in range(USERS)]
main_epoch_losses = []
test_losses = []

for iter in range(total_main_epochs):
    print(f"+++++++++++++++Main Epoch: {iter+1}+++++++++++++++")
    total_avg_local_loss = 0

    for client in participating_clients:
        print(f"Client: {client}")

        local_glob, avg_local_loss = Update_Local_Client(local_models[client], user_data[client])
        print(f"Local Loss for Client {client}: {avg_local_loss}")

        if all_clients_participate:
            local_models[client] = deepcopy(local_glob)
        else:
            local_models.append(deepcopy(local_glob))

        total_avg_local_loss += avg_local_loss

    avg_local_loss_over_clients = total_avg_local_loss / len(participating_clients)
    print(f"Avg Local Loss over Clients: {avg_local_loss_over_clients}")
    main_epoch_losses.append(avg_local_loss_over_clients)

    w_glob = federated_average(local_models)
    net_glob.load_state_dict(w_glob)

    # Evaluation on the test set
    net_glob.eval()  # Set model to evaluation mode
    running_test_loss = 0

    with torch.no_grad():
        # Iterate over batches in the test data loader
        for batch_idx, (X_batch, y_batch) in enumerate(test_data_loader):
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            output, _ = net_glob(X_batch)  # Use net_glob for evaluation
            output = output.squeeze(1)
            loss = loss_function(output, y_batch)
```

```
            running_test_loss += loss.item()

    avg_test_loss = running_test_loss / len(test_data_loader)
    print(f"Average Test Loss: {avg_test_loss}")
    print("*****************************************")

    test_losses.append(avg_test_loss)  # Append testing loss for the epoch

    if all_clients_participate:
        # If all clients participate
        continue
    else:
        # If only a fraction of clients participate, randomly select participating clients
        num_participating_clients = max(int(fraction_of_clients_participating * num_clients),
    1)  # Ensure at least 1 client participates
        participating_clients = np.random.choice(range(num_clients), num_participating_clients,
     replace=False)

# Plot the main epoch losses
plt.plot(range(1, total_main_epochs + 1), main_epoch_losses, label='Main Epoch Loss')
plt.plot(range(1, total_main_epochs + 1), test_losses, label='Testing Loss')
plt.xlabel('Main Epoch')
plt.ylabel('Average Local Loss')
plt.title('Main Epoch Losses')
plt.legend()
plt.show()
```

Remaining code is described in the jupyter notebook shared.

3) Software Implementation: These diagrams [2] accurately describes the entire architecture of the implementation.

We use Mean absolute Error (MAE) and Mean squared Error(MSE) to indicate the prediction accuracy.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_p|$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_p)^2$$

where y_i represents the observed traffic flow and $\hat{y}_p$ represents the predicted traffic flow for the ith sample.

Now, we describe the way the whole implementation works. We first initialize the global model weights randomly. Then, these weights are transferred to all the clients who then create their own models with the weights sent by the central global server. Once the clients get the models, they train their individual models on their individual datasets without communicating with each other in any manner possible. This is depicted in Fig. 3.

Once the local models are trained, each client sends the weights of its model to the global model in cloud (Fig. 4.) which then uses the FedAVG algorithm as described in the Executable Code section. Once this weight aggregation is completed, the global model in the cloud then sends its weights back to all the clients and the process continues. (Fig. 5.)

IV. Results/Output

In this experiment, the proposed FedGRU algorithm is applied to the real-world data collected from the Caltrans Performance Measurement System (PeMS) database for performance demonstration [3]. The traffic flow data in the PeMS database were collected from over 39,000 individual detectors in real time. These sensors span the freeway system across all major metropolitan areas of the State of California . In this article, traffic flow data collected during the first three months of 2013 are used for experiments. We select the traffic flow data in the first two months as the training dataset and the third month as the

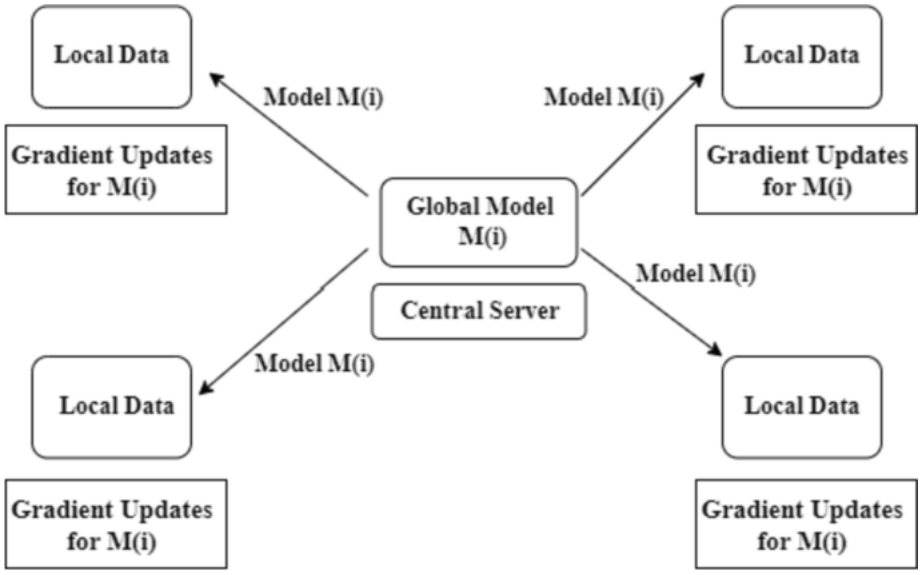

Fig. 3. Globel model M is initialized and its weights are shared to each client model. [2]

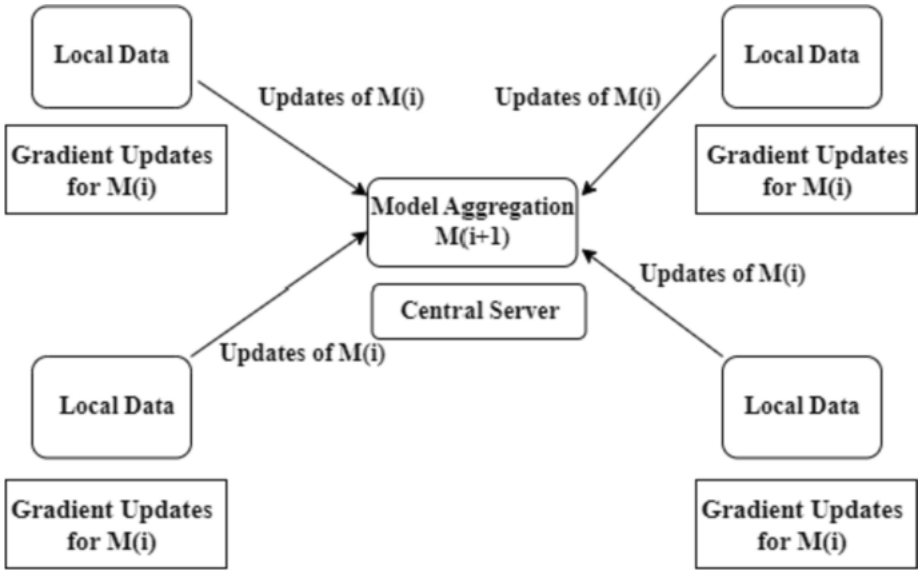

Fig. 4. After clients are trained on their local datasets, they send their data to the global model for aggregation via the FedAVG algorithm. [2]

testing dataset. Furthermore, since the traffic flow data are time- series data, we need to use them at the previous time interval, i.e., $x_{t-1}, x_{t-2}, \ldots, x_{t-r}$, to predict the traffic flow at time interval t, where r is the length of the history data window.

Without loss of generality, we assume that the detector stations are distributed and independent and the data cannot be exchanged arbitrarily among them.

For the cloud and each client, we use minibatch ADAMs for model optimization. The PeMS data set is split equally and assigned to 7 clients. During the simulation, learning rate α = 0.001 , minibatch size m=256 is used . All experiments are conducted using PyTorch.

We use the model architecture as it was described in the paper [1] since they properly tuned the hyperparameters. We use the optimal length of history data window which was found to be r=12 . We took number of hidden layers as 2 and the number of hidden units in each layer to be 100.

We also randomize the number of clients chosen each iteration with a random factor which lies between

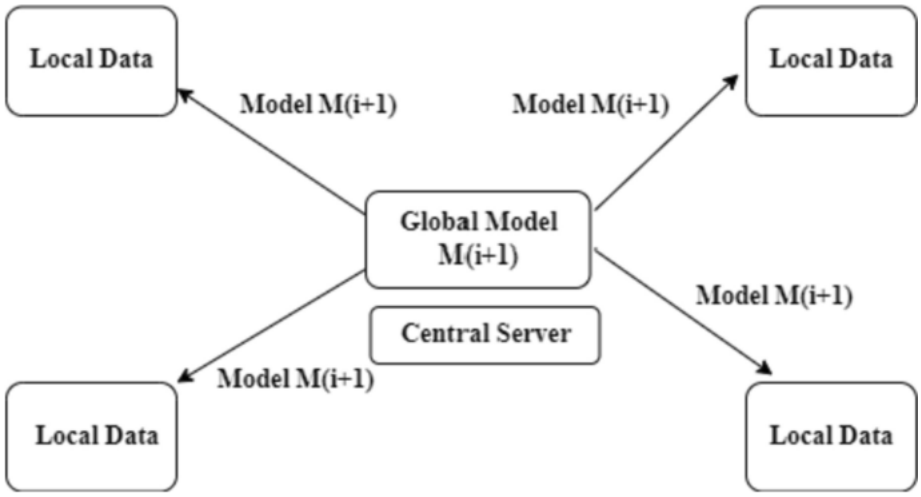

Fig. 5. Once the global model is aggregated, its weights are again sent back to the clients and the process continues. [2]

0 and 1. This was referred in paper as the Joint-Announcement Protocol but what it basically does is to randomly take some fraction of total clients available each time. In our case, we take that fraction to be 0.5. This technique is useful when we have large scale organizations. In such cases, FedAVG is no longer suitable and thus we train the global model on a subset of organizations.

We evaluate the model by predicting a 5 min interval as was shown in the paper and the model is strong enough to predict the traffic flow of the entire test set with a very high accuracy as is described in the paper.

Metrics	MAE	MSE	RMSE	MAPE
FedGRU (default setting)	7.96	101.49	11.04	17.82%
GRU [22]	7.20	99.32	9.97	17.78%
SAE [1]	8.26	99.82	11.60	19.80%
LSTM [20]	8.28	107.16	11.45	20.32%
SVM [51]	8.68	115.52	13.24	22.73%

Fig. 6. Results as presented in the paper. We compare our results to the one presented by the authors. We only implemented the GRU and the FedGRU algorithms. [1]

We trained GRU and FedGRU models with hyperparameters as described in the paper since they tuned hyperparameters extensively. Our results are similar to the ones described in the paper.

TABLE I
MODEL EVALUATION METRICS

Model	**MAE**	**MSE**
GRU	7.21	96.67
FedGRU	7.27	97.74

This table presents the evaluation metrics (MSE and MAE) for the GRU and FedGRU models. MSE stands for Mean Squared Error, and MAE stands for Mean Absolute Error. Lower values indicate better performance. You can observe visually that our implementation of the same models gives slightly better performance than the ones given by the original authors. Additionally, notice that GRU and FedGRU have almost the same performance thus we get the performance of GRU with data protection in FedGRU.

Here are the loss curves for both GRU and FedGRU models on train set as well the test set. This resembles the curves that were published in the original paper. Infact, you can observe that both GRU and FedGRU are able to learn properly and to the same amount even though FedGRU is mostly trained on

subset of dataset each time and the global model is not even trained on the data; just the client weights. This demonstrates that FedGRU is a successful and better method when data privacy is a must.

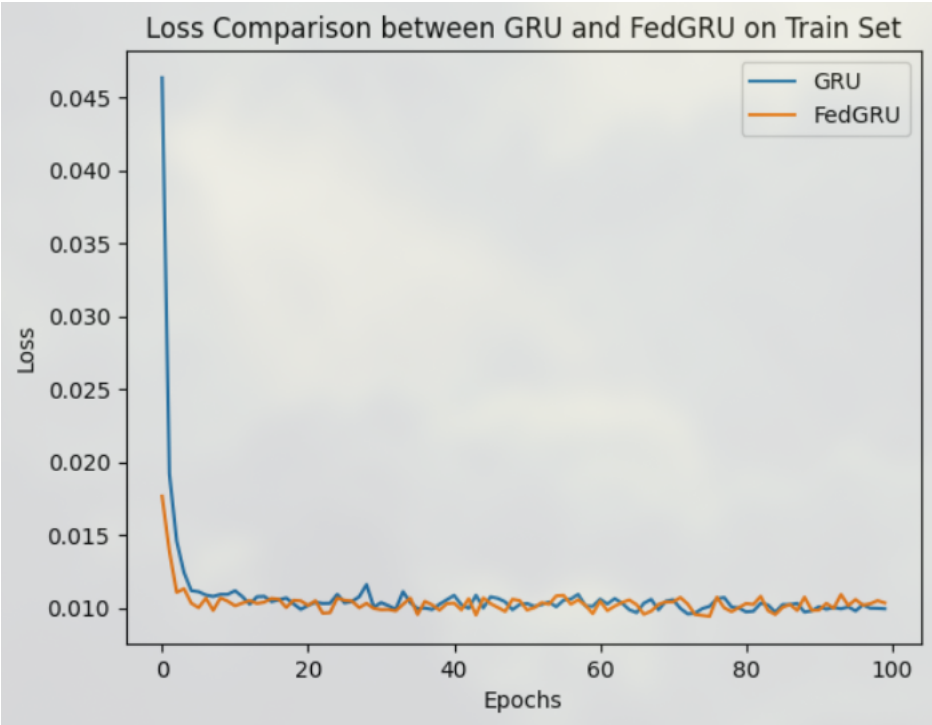

Fig. 7. Loss comparison between GRU and FedGRU on the training set.

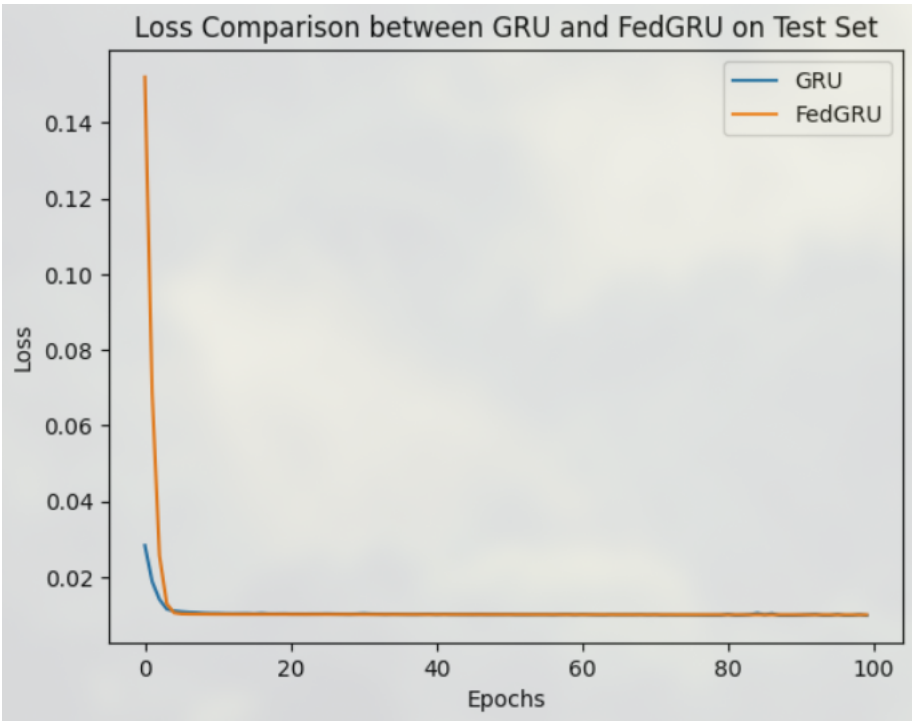

Fig. 8. Loss comparison between GRU and FedGRU on the testing set.

Below are the predictions made by both GRU and FedGRU models on the testing set.

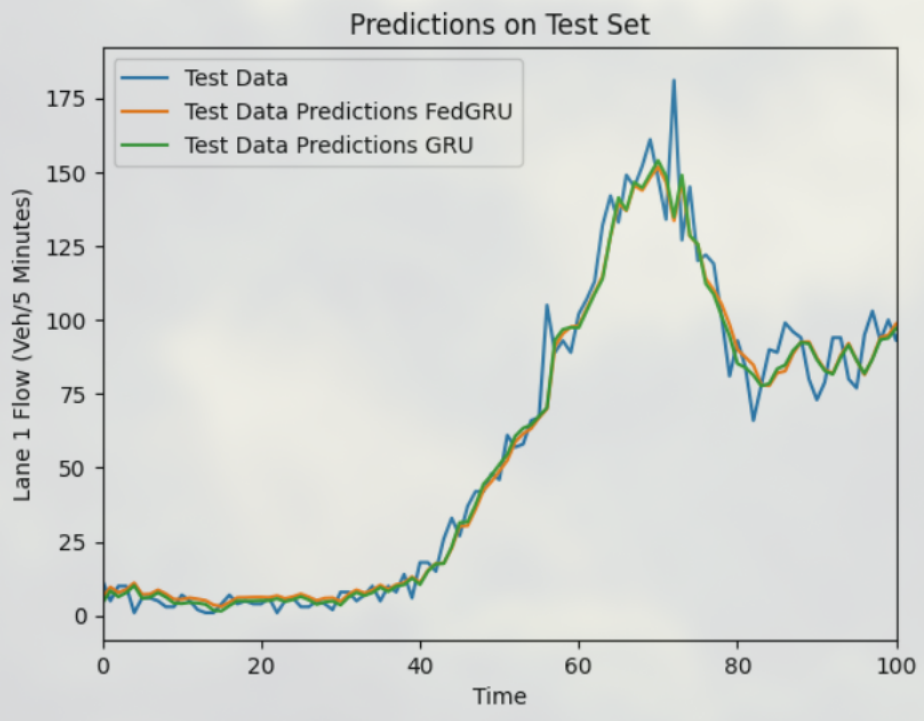

Fig. 9. Predictions made by models GRU and FedGRU on train and test set.FedGRU does implement Joint Announcement Protocol in its training.

V. Conclusion and Future Directions

We implemented the FedGRU algorithm for traffic flow prediction with FL for privacy preservation. FedGRU does not have direct access to the organizational data. It aggregates the gradient information uploaded by all the locally trained models and aggregates them to construct a single global model for the traffic flow prediction problem.

We tested our implementation on the PeMS dataset and compared algorithms GRU and FedGRU. Results show that the FedGRU algorithm performs comparably to the GRU algorithm with very minuscule losses in performance.

One thing we did not implement from the paper was the ensemble clustering based FedGRU for TFP to improve model performance further due to time and computation constraints. We would like to do this in the future and try out graph convolutional networks as well.

References

[1] Y. Liu, J. J. Q. Yu, J. Kang, D. Niyato, and S. Zhang, "Privacy-preserving traffic flow prediction: A federated learning approach," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7751–7763, 2020.

[2] G. Nidhi and J, "Federated learning analysis for vehicular traffic flow prediction: evaluation of learning algorithms and aggregation approaches," *Cluster Computing*, 2024.

[3] C. Chao, "Freeway performance measurement system (pems)," 2003.