TIME SERIES FORECASTING USING GATED RECURRENT UNIT

Inkita Tewari
itewari1@asu.edu

1 Introduction

For time series forecasting of traffic flows using the given dataset, there are several established models to consider like Autoregressive Integrated Moving Average (ARIMA), Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), Temporal Convolutional Networks (TCNs), Transformers and others.

Given the high frequency nature of the data (traffic flow per hour for a year), it was essential to choose a model that would look back at long term trends and give a prediction. Among the popular choices for this requirement are LSTM and GRU. After weighing their trade-offs and experimenting with both on the training data, GRU demonstrated better performance, achieving a lower Mean Squared Error (MSE) on the validation data, despite comparable loss values on the training data for both models.

1.1 Gated Recurrent Unit

GRUs are a type of recurrent neural network (RNN) that can capture long-term dependencies on the past information which makes it a good choice for time-series forecasting. The GRU model takes a sequence of past time steps as input, and it initializes a hidden state vector that represents the past information. A GRU cell consists of two main components: Reset Gate and Update Gate. These gates control the flow of information that is passed to the next time step, retaining only relevant information. The gating mechanisms in GRUs reduce the vanishing gradient problem encountered in traditional RNNs. This allows the model to propagate useful gradients back over long sequences.

2 Methodology

I started by implementing a basic GRU with just one layer, keeping the model parameters as they were and then passed the output from GRU through a fully connected linear layer. Afterward, I experimented with different parameter values and normalized the data to handle certain values that were significantly larger than others. Through trial and error, I achieved the lowest MSE of 0.0025, with predicted values almost overlapping with actual values in the plot, except for the outliers. The parameters that led to this result were:

- Learning rate = 0.0001
- Number of layers in GRU = 3
- Epochs = 3

To further optimize this model, I considered some regularization techniques. I started by reducing the learning rate after a certain number of batches and, in another iteration, adjusted the learning rate after each epoch. These regularizations did not yield the results I was hoping to achieve. Another popular regularization technique with GRU is adding dropout between layers. Since I used three layers in my model, I added a dropout of 0.3 after each layer. This technique causes the GRU to randomly drop 30% of the cells, helping to prevent the model from overfitting the training data. After applying this, the MSE further reduced to 0.0019, and again the predicted and actual values in the plot seemed to overlap, except for the outliers.

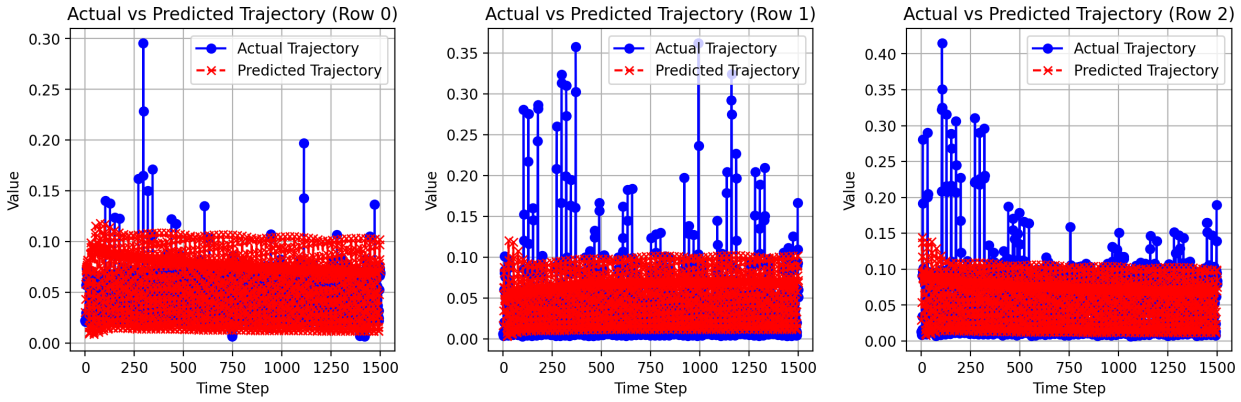


Figure 1: Plots for 3 validation IDs

3 Results

The comparative plots for actual vs predicted values were generated for validation ids: 0, 1 and 2. See Figure 1.

4 Possible Enhancements

While researching techniques to improve model training and reduce the risk of overfitting—so that predictions handle outliers more effectively—I came across several regularization techniques, including Dropout, Weight Decay, Gradient Clipping, Early Stopping, and others. I was able to implement only Dropout in my model, which improved its performance. However, combining Dropout with other regularization techniques could further enhance the model's performance. Additionally, another potential improvement would be to incorporate feature engineering, such as adding day-of-the-week flags since that information is available in the data. This could help the model account for outliers and produce more accurate predictions for those cases.

5 Conclusion

In this project, I explored various approaches for time series forecasting of traffic flow, focusing on high-frequency data. After evaluating multiple models, Gated Recurrent Unit (GRU) emerged as the most effective for capturing long-term trends, achieving a low Mean Squared Error (MSE) on the validation data. Through parameter optimization and the application of regularization techniques (dropout), the GRU model's performance improved, accurately predicting values with exception of outliers.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm  # Import tqdm for progress tracking


class TrajectoryDataset(Dataset):
    def __init__(self, dataframe, window_length=100):
        # Perform the custom transformation
        sliced_df = self.custom_transformation(dataframe.to_numpy(), window_length=window_length)
        self.data = torch.tensor(sliced_df, dtype=torch.float32)

    def __len__(self):
        # Return the number of trajectories
        return self.data.shape[0]

    def __getitem__(self, idx):
        # Get the trajectory at the given index
        return self.data[idx]

    def custom_transformation(self, dataframe_array, window_length):
        num_rows, num_cols = dataframe_array.shape
        window_length += 1  # get one more column as targets

        # Preallocate memory for the slices
        sliced_data = np.lib.stride_tricks.sliding_window_view(dataframe_array, window_shape=(window_length,), axis=1)

        # Reshape into a flat 2D array for DataFrame-like output
        sliced_data = sliced_data.reshape(-1, window_length)

        return sliced_data

# Implement your model
class GRUModelWithDropout(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, dropout):
        super(GRUModelWithDropout, self).__init__()
        input_size = 1
        output_size = 1

        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True, dropout=dropout)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # x = nn.functional.normalize(x)
        x = x.unsqueeze(-1) #adding a dimension to data before passing it to GRU
        gru_out, _ = self.gru(x)
        last_hidden = gru_out[:, -1, :]
        last_hidden = self.dropout(last_hidden)
        out = self.fc(last_hidden)
        return out
```

Training loop

```
import os

# Get the relative path of a file in the current working directory
train_path = os.path.join('train.csv')
val_path = os.path.join('val.csv')
test_path = os.path.join('test.csv')

train_df = pd.read_csv(train_path, header = 0).drop('ids', axis=1)
val_df = pd.read_csv(val_path, header = 0).drop('ids', axis=1)
test_df = pd.read_csv(test_path, header = 0).drop('ids', axis=1)



# Check if MPS is available and set the device accordingly
device = torch.device('cpu')
```

```
    if torch.cuda.is_available():
        device = torch.device("cuda")

    window_length = 100  # Example window length
    dataset = TrajectoryDataset(dataframe=train_df, window_length=window_length)
    batch_size = 32
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

    # Model hyperparameters
    input_size = window_length  # Window length minus 1 (since the last column is the target)
    hidden_size = 64
    output_size = 1  # Single output for time series forecast (next value)
    learning_rate = 0.0001
    num_epochs = 3
    dropout = 0.3
    num_layers = 3

    # Instantiate the model, loss function, and optimizer
    model = GRUModelWithDropout(input_size=input_size, hidden_size=hidden_size, output_size=output_size, num_layers=num_layers, dropo
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Training loop with tqdm for progress tracking
    for epoch in tqdm(range(num_epochs), desc="Epochs", unit="epoch"):
        model.train()
        running_loss = 0.0
        # Use tqdm to track batch progress within each epoch
        for batch_idx, data in tqdm(enumerate(dataloader), desc=f"Epoch {epoch + 1}", unit="batch", leave=False):
            # Separate inputs and target
            inputs = data[:, :-1].to(device)  # All except last column
            # print(inputs.shape)
            targets = data[:, -1].to(device)  # Last column is the target (next value)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)

            # Compute the loss
            loss = criterion(outputs.squeeze(), targets)

            # Backward pass and optimize
            loss.backward()

            # if batch_idx % 10000 == 0:
            #     for param_group in optimizer.param_groups:
            #         param_group['lr'] *= 0.9

            optimizer.step()

            running_loss += loss.item()

        # optimizer.param_groups[0]['lr'] *= 0.1

        # Print the average loss per epoch
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {running_loss / len(dataloader):.4f}')
```

```
Epoch 1: 13055batch [00:20, 662.53batch/s]
Epoch 1: 13122batch [00:20, 661.95batch/s]
Epoch 1: 13189batch [00:20, 661.01batch/s]
Epoch 1: 13256batch [00:20, 661.69batch/s]
Epoch 1: 13323batch [00:20, 661.61batch/s]
Epoch 1: 13390batch [00:20, 662.78batch/s]
Epoch 1: 13457batch [00:20, 662.91batch/s]
Epoch 1: 13524batch [00:20, 662.20batch/s]
Epoch 1: 13591batch [00:20, 661.90batch/s]
Epoch 1: 13658batch [00:20, 661.46batch/s]
Epoch 1: 13725batch [00:21, 661.99batch/s]
Epoch 1: 13792batch [00:21, 661.49batch/s]
Epoch 1: 13859batch [00:21, 661.76batch/s]
Epoch 1: 13926batch [00:21, 661.23batch/s]
Epoch 1: 13993batch [00:21, 660.84batch/s]
Epoch 1: 14060batch [00:21, 659.93batch/s]
Epoch 1: 14127batch [00:21, 661.43batch/s]
Epoch 1: 14194batch [00:21, 660.67batch/s]
Epoch 1: 14261batch [00:21, 661.48batch/s]
Epoch 1: 14328batch [00:21, 661.21batch/s]
Epoch 1: 14395batch [00:22, 660.32batch/s]
Epoch 1: 14462batch [00:22, 660.56batch/s]
Epoch 1: 14529batch [00:22, 660.78batch/s]
Epoch 1: 14596batch [00:22, 661.17batch/s]
Epoch 1: 14663batch [00:22, 662.18batch/s]
Epoch 1: 14730batch [00:22, 662.08batch/s]
Epoch 1: 14797batch [00:22, 662.87batch/s]
Epoch 1: 14864batch [00:22, 662.67batch/s]
Epoch 1: 14931batch [00:22, 662.53batch/s]
Epoch 1: 14998batch [00:22, 663.06batch/s]
Epoch 1: 15065batch [00:23, 663.70batch/s]
Epoch 1: 15132batch [00:23, 663.12batch/s]
Epoch 1: 15199batch [00:23, 662.16batch/s]
Epoch 1: 15266batch [00:23, 661.60batch/s]
Epoch 1: 15333batch [00:23, 660.74batch/s]
Epoch 1: 15400batch [00:23, 660.68batch/s]
Epoch 1: 15467batch [00:23, 661.14batch/s]
```

```
torch.save(model.state_dict(), "model.pth")
```

Evaluation Loop

```
from torch.nn import MSELoss


train_set = torch.tensor(train_df.values[:,:].astype(np.float32), dtype=torch.float32)
val_set = torch.tensor(val_df.values[:,:].astype(np.float32), dtype=torch.float32).to(device)
test_set = torch.tensor(val_df.values[:,:].astype(np.float32), dtype=torch.float32)

points_to_predict = val_set.shape[1]

# Autoregressive prediction function
def autoregressive_predict(model, input_maxtrix, prediction_length=points_to_predict):
    """
    Perform autoregressive prediction using the learned model.

    Args:
    - model: The trained PyTorch model.
    - input_maxtrix: A matrix of initial time steps (e.g., shape (963, window_length)).
    - prediction_length: The length of the future trajectory to predict.

    Returns:
    - output_matrix: A tensor of the predicted future trajectory of the same length as `prediction_length`.
    """
    model.eval()  # Set model to evaluation mode
    output_matrix = torch.empty(input_maxtrix.shape[0],0).to(device)
    current_input = input_maxtrix.to(device)

    with torch.no_grad():  # No need to calculate gradients for prediction
        for idx in range(prediction_length):
            # Predict the next time step
            next_pred = model(current_input)

            # Concatenating the new column along dimension 1 (columns)
            output_matrix = torch.cat((output_matrix, next_pred), dim=1)

            # Use the predicted value as part of the next input
            current_input = torch.cat((current_input[:, 1:],next_pred),dim=1)

    return output_matrix


initial_input = train_set[:, -window_length:].to(device)  #use the last window of training set as initial input
full_trajectories = autoregressive_predict(model, initial_input).to(device)


# Calculate MSE between predicted trajectories and actual validation trajectories using torch
mse_loss = MSELoss()

# Compute MSE
mse = mse_loss(full_trajectories, val_set)

# Print MSE
print(f'Autoregressive Validation MSE (using torch): {mse.item():.4f}')
```

```
Autoregressive Validation MSE (using torch): 0.0019
```

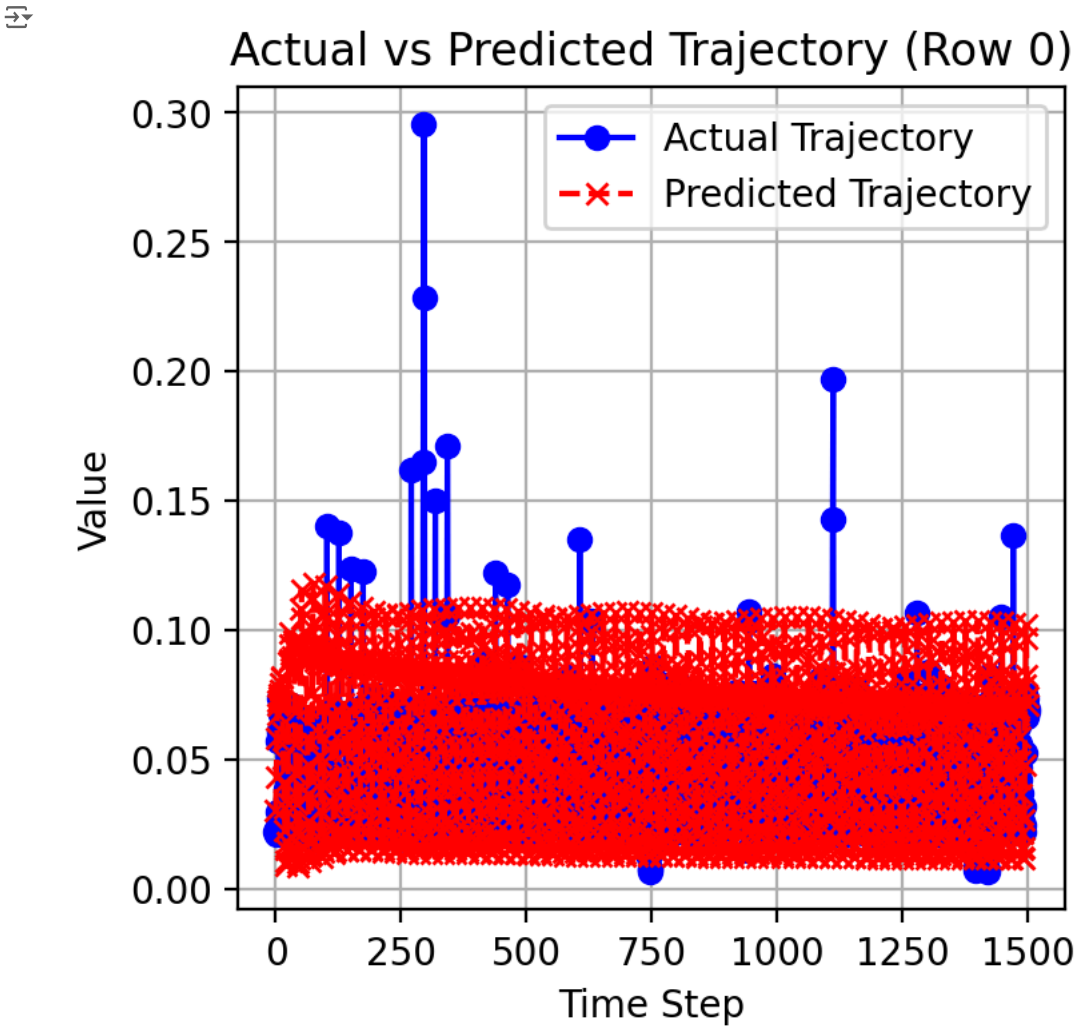
Plot it out to see what is like

```
# Perform autoregressive predictions for one row in the validation set
# We can pick a specific row (e.g., row 0) to visualize
row_idx = 0  # You can change this to visualize predictions for different rows
initial_input = val_set[row_idx, :window_length].unsqueeze(0)

# Predict future trajectory of length 100
predicted_trajectory = autoregressive_predict(model, initial_input)

# Get the actual trajectory for comparison
actual_trajectory = val_set[row_idx].cpu().numpy()

# Plot the actual vs predicted trajectory
plt.figure(figsize=(4, 4), dpi=200)
plt.plot(range(len(actual_trajectory)), actual_trajectory, label="Actual Trajectory", color='blue', marker='o')
plt.plot(range(len(actual_trajectory)), predicted_trajectory.cpu().squeeze().numpy(), label="Predicted Trajectory", color='red',
plt.title(f"Actual vs Predicted Trajectory (Row {row_idx})")
plt.xlabel("Time Step")
plt.ylabel("Value")
plt.legend()
plt.grid(True)
plt.show()
```

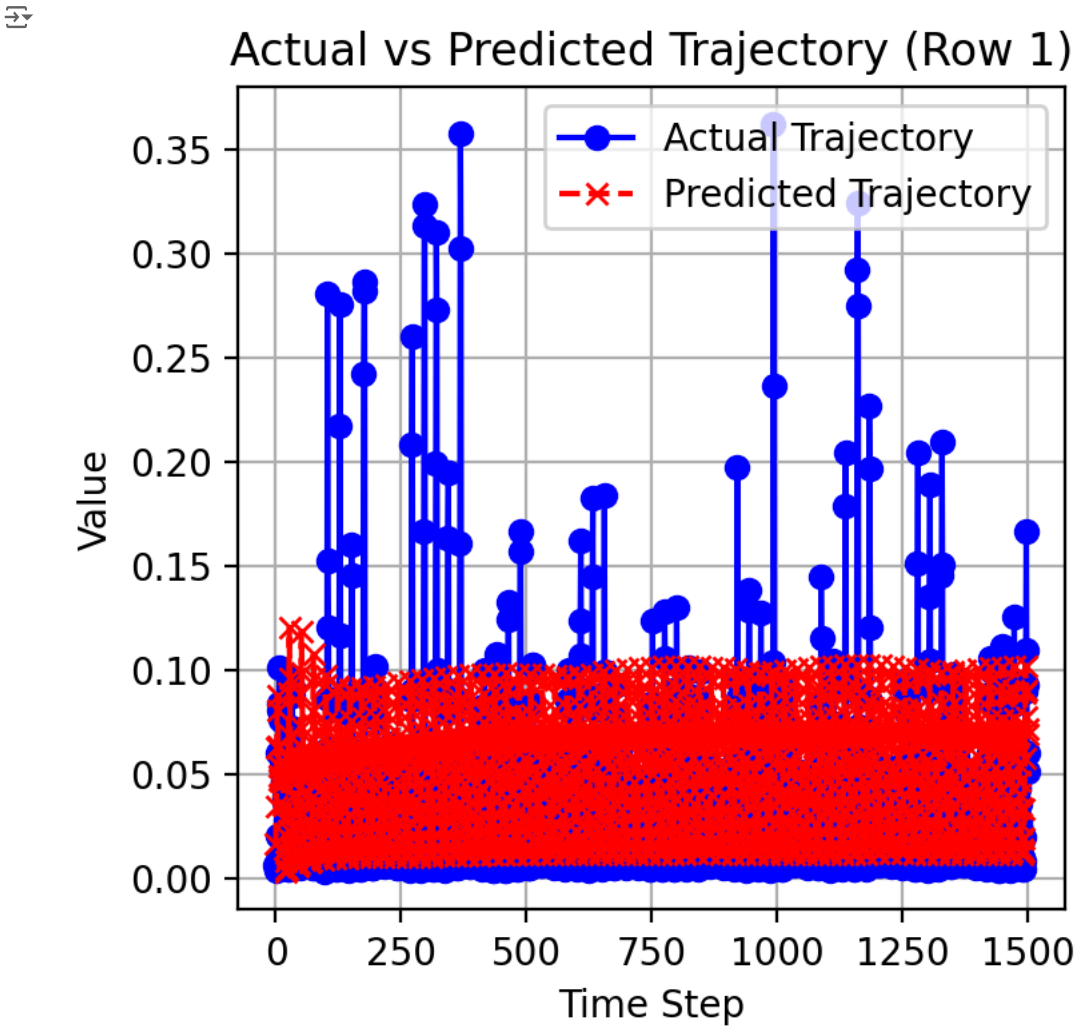


```
# Perform autoregressive predictions for one row in the validation set
# We can pick a specific row (e.g., row 0) to visualize
row_idx = 1  # You can change this to visualize predictions for different rows
initial_input = val_set[row_idx, :window_length].unsqueeze(0)

# Predict future trajectory of length 100
predicted_trajectory = autoregressive_predict(model, initial_input)

# Get the actual trajectory for comparison
actual_trajectory = val_set[row_idx].cpu().numpy()

# Plot the actual vs predicted trajectory
plt.figure(figsize=(4, 4), dpi=200)
plt.plot(range(len(actual_trajectory)), actual_trajectory, label="Actual Trajectory", color='blue', marker='o')
plt.plot(range(len(actual_trajectory)), predicted_trajectory.cpu().squeeze().numpy(), label="Predicted Trajectory", color='red',
plt.title(f"Actual vs Predicted Trajectory (Row {row_idx})")
plt.xlabel("Time Step")
plt.ylabel("Value")
plt.legend()
plt.grid(True)
plt.show()
```



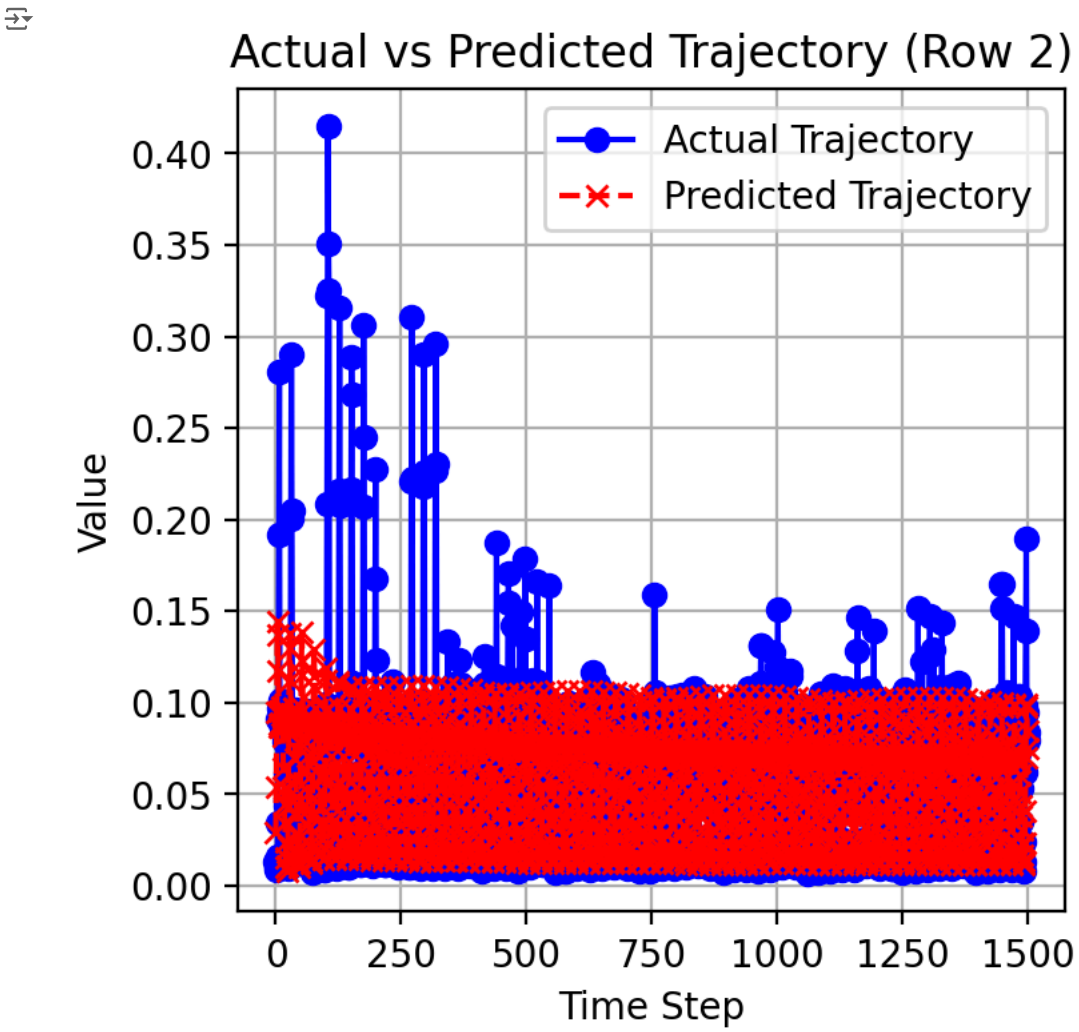

```
# Perform autoregressive predictions for one row in the validation set
# We can pick a specific row (e.g., row 0) to visualize
row_idx = 2  # You can change this to visualize predictions for different rows
initial_input = val_set[row_idx, :window_length].unsqueeze(0)

# Predict future trajectory of length 100
predicted_trajectory = autoregressive_predict(model, initial_input)

# Get the actual trajectory for comparison
actual_trajectory = val_set[row_idx].cpu().numpy()

# Plot the actual vs predicted trajectory
plt.figure(figsize=(4, 4), dpi=200)
```

```
plt.figure(figsize=(4, 4), dpi=200)
plt.plot(range(len(actual_trajectory)), actual_trajectory, label="Actual Trajectory", color='blue', marker='o')
plt.plot(range(len(actual_trajectory)), predicted_trajectory.cpu().squeeze().numpy(), label="Predicted Trajectory", color='red',
plt.title(f"Actual vs Predicted Trajectory (Row {row_idx})")
plt.xlabel("Time Step")
plt.ylabel("Value")
plt.legend()
plt.grid(True)
plt.show()
```




```
# Generate predictions for all the validation dataset
initial_input = train_set[:, -window_length:]
val_predictions_tensor = autoregressive_predict(model, initial_input)

# Generate predictions for all the test dataset
initial_input = val_predictions_tensor[:, -window_length:]
test_predictions_tensor = autoregressive_predict(model, initial_input)


# Print their shapes
print(f'Validation Predictions Tensor Shape: {val_predictions_tensor.shape}')
print(f'Test Predictions Tensor Shape: {test_predictions_tensor.shape}')
```

```
Validation Predictions Tensor Shape: torch.Size([963, 1500])
Test Predictions Tensor Shape: torch.Size([963, 1500])
```

```
def generate_submissions_v4(pred_val_tensor, pred_test_tensor, original_val_path, original_test_path):
    # Read the original validation and testing datasets
    original_val_df = pd.read_csv(original_val_path)
    original_test_df = pd.read_csv(original_test_path)
```