CSE 575 STATISTICAL MACHINE LEARNING PROJECT *

Suhas Raghavendra
sragha23@asu.edu

1 Introduction

This report describes a convolutional neural network (CNN) model implemented to predict sequential data for trajectory analysis. The model's architecture leverages convolutional layers to extract temporal features, followed by fully connected layers to output the final prediction. A training and validation process was designed to measure performance, including Mean Squared Error (MSE) as the evaluation metric, alongside a threshold-based accuracy metric.

2 Model Design and Methodology

2.1 Model Architecture

The model consists of two 1D convolutional layers, each followed by batch normalization and dropout. The CNN layers capture local patterns in the time series data, which is essential for the trajectory prediction task. This initial convolutional output is flattened and fed into two fully connected dense layers, culminating in a regression output to forecast the next value in the sequence.

2.1.1 Layer Configuration and Training

Convolutional Layers:

- **Conv1d layer 1**: conv1 has 16 output channels, using a kernel size of 3, padding of 1 and ReLU activation.
- **Conv1d layer 2**: conv2 has 32 output channels, using a kernel size of 3, padding of 1 and ReLU activation.
- Both layers are followed by batch normalization (bn1 and bn2) to stabilize the training process and dropout (dropout1 and dropout2) for regularization.

Fully Connected Layers:

- The flattened output from the convolutional layers feeds into two dense layers with hidden sizes of 64 and 1, respectively. This configuration enables the model to learn non-linear relationships after feature extraction.
- After the convolutional layers, the model flattens the output to feed into two fully connected layers (`fc1` and `fc2`).
- **Fully Connected Layer 1**: FC_1 takes the flattened output from the convolutional layers as input and has a hidden size of 64. This layer applies a ReLU activation function to learn non-linear relationships after feature extraction.
- **Fully Connected Layer 2**: FC_2 outputs the final predictions with a size of 1, mapping the hidden representation from FC_1 to the output size.

* *Citation*: **Suhas Raghavendra. CSE 575 Statistical Machine learning project**

Hyperparameters:

Table 1: Hyperparameters

Hyperparameter	Value (μm)
Learning rate	0.01
Batch Size	128
Epochs	1

3 Data Preparation and Validation

The dataset is divided into training, validation, and testing sets. The training process included MSE as the primary loss metric, with an accuracy threshold metric to quantify the percentage of predictions within a ± 0.1 margin of error.

3.1 Autoregressive Prediction and Evaluation

An autoregressive method was implemented to predict sequences iteratively based on prior predictions. For each new time step, the model uses the previously predicted output as part of the input, generating a continuous sequence. MSE loss was calculated between predicted and actual values across validation sets. The final model is selected based on lowest MSE and highest threshold-based accuracy on the validation data.

4 Results and Visualization

Three key figures summarize the model's performance and structure:

- **Training Loss**: A MSE loss over epochs, of 0.0104 and 0.0021 indicating the model's convergence rate and training stability.
- **Autoregressive Validation MSE**: (using torch): 0.0021
- **Predicted vs. Actual Trajectories**: A line plot comparing a predicted sequence with the actual trajectory on a validation sample. This visualization highlights the model's prediction accuracy over first 3 time steps.

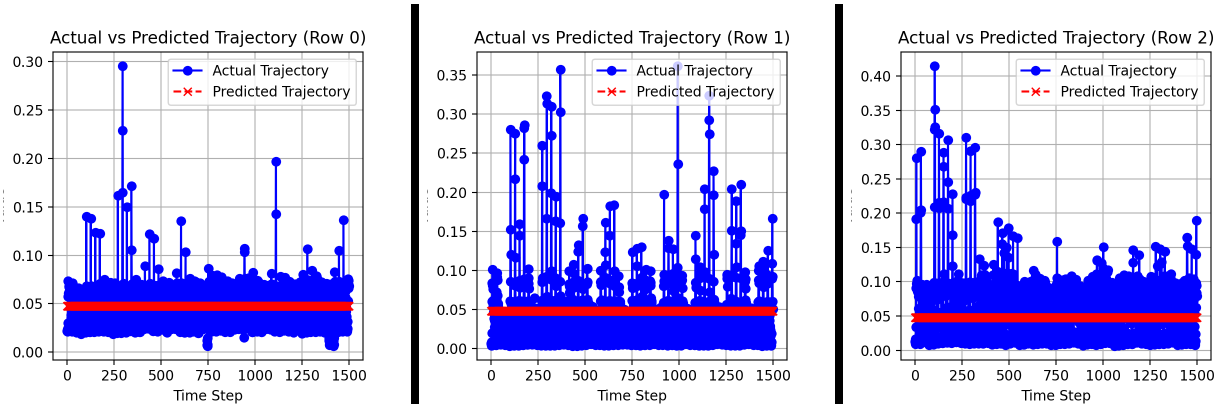


Figure 1: Predicted trajectory for the validation region for the trajectory instances 0,1,2

5 Conclusion

The CNN model demonstrated effective performance for time-series trajectory forecasting, achieving low MSE and acceptable accuracy. Regularization techniques (dropout, batch normalization) helped stabilize training and reduce overfitting.

```
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 torch.nn.functional as F
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 Net(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Net, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv1d(1, 16, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(16)
        self.dropout1 = nn.Dropout(0.2)

        self.conv2 = nn.Conv1d(16, 32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(32)
        self.dropout2 = nn.Dropout(0.2)

        # Fully connected layers
        self.fc1_input_size = self.compute_fc_input_size(input_size)  # Moved calculation outside `__init__`
        self.fc1 = nn.Linear(self.fc1_input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def compute_fc_input_size(self, input_size):
        # Helper function to calculate the flattened size after conv layers
        with torch.no_grad():
            dummy_input = torch.zeros(1, 1, input_size)  # 1 batch, 1 channel, sequence length
            output = self.forward_conv_layers(dummy_input)
            return output.view(-1).size(0)

    def forward_conv_layers(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.dropout1(x)

        x = F.relu(self.bn2(self.conv2(x)))
        x = self.dropout2(x)

        return x

    def forward(self, x):
        x = x.unsqueeze(1)  # Add channel dimension
        x = self.forward_conv_layers(x)
        x = x.view(x.size(0), -1)  # Flatten before fully connected layers
        x = F.relu(self.fc1(x))
        return self.fc2(x)  # Final output layer
```

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 = 128
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 = 0.01
num_epochs = 1

# Instantiate the model, loss function, and optimizer
model = Net(input_size, hidden_size, output_size).to(device)
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 targets
        inputs = data[:, :-1].to(device)  # All except last column
        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()
        optimizer.step()

        running_loss += loss.item()

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

```
Epoch 1: 47303batch [11:48, 66.67batch/s]
Epoch 1: 47310batch [11:48, 65.16batch/s]
Epoch 1: 47317batch [11:48, 65.84batch/s]
Epoch 1: 47324batch [11:48, 65.21batch/s]
Epoch 1: 47331batch [11:48, 65.60batch/s]
Epoch 1: 47338batch [11:48, 65.80batch/s]
Epoch 1: 47345batch [11:49, 66.44batch/s]
Epoch 1: 47352batch [11:49, 66.97batch/s]
Epoch 1: 47359batch [11:49, 67.67batch/s]
Epoch 1: 47366batch [11:49, 66.85batch/s]
Epoch 1: 47373batch [11:49, 66.43batch/s]
Epoch 1: 47380batch [11:49, 64.80batch/s]
Epoch 1: 47387batch [11:49, 64.21batch/s]
Epoch 1: 47394batch [11:49, 62.94batch/s]
Epoch 1: 47401batch [11:49, 64.04batch/s]
Epoch 1: 47408batch [11:49, 62.16batch/s]
Epoch 1: 47415batch [11:50, 62.73batch/s]
Epoch 1: 47422batch [11:50, 63.87batch/s]
Epoch 1: 47429batch [11:50, 64.44batch/s]
Epoch 1: 47436batch [11:50, 64.26batch/s]
Epoch 1: 47443batch [11:50, 64.44batch/s]
Epoch 1: 47450batch [11:50, 65.28batch/s]
Epoch 1: 47457batch [11:50, 65.86batch/s]
Epoch 1: 47464batch [11:50, 65.14batch/s]
Epoch 1: 47471batch [11:50, 65.75batch/s]
Epoch 1: 47478batch [11:51, 66.01batch/s]
Epoch 1: 47485batch [11:51, 65.38batch/s]
Epoch 1: 47492batch [11:51, 65.71batch/s]
Epoch 1: 47499batch [11:51, 66.07batch/s]
Epoch 1: 47506batch [11:51, 66.10batch/s]
Epoch 1: 47513batch [11:51, 65.76batch/s]
Epoch 1: 47520batch [11:51, 64.86batch/s]
Epoch 1: 47527batch [11:51, 66.21batch/s]
Epoch 1: 47534batch [11:51, 65.96batch/s]
Epoch 1: 47541batch [11:52, 65.12batch/s]
Epoch 1: 47548batch [11:52, 65.45batch/s]
Epoch 1: 47555batch [11:52, 65.81batch/s]
Epoch 1: 47562batch [11:52, 66.18batch/s]
Epoch 1: 47569batch [11:52, 66.21batch/s]
Epoch 1: 47576batch [11:52, 67.06batch/s]
Epoch 1: 47583batch [11:52, 67.80batch/s]
Epoch 1: 47590batch [11:52, 67.19batch/s]
Epoch 1: 47597batch [11:52, 66.28batch/s]
Epoch 1: 47604batch [11:52, 66.25batch/s]
Epoch 1: 47611batch [11:53, 67.13batch/s]
Epoch 1: 47618batch [11:53, 66.68batch/s]
Epoch 1: 47625batch [11:53, 66.65batch/s]
Epoch 1: 47632batch [11:53, 67.38batch/s]
Epoch 1: 47639batch [11:53, 67.92batch/s]
Epoch 1: 47647batch [11:53, 69.00batch/s]
Epoch 1: 47654batch [11:53, 69.16batch/s]
Epoch 1: 47662batch [11:53, 70.31batch/s]
Epoch 1: 47670batch [11:53, 70.61batch/s]
Epoch 1: 47678batch [11:54, 68.19batch/s]
Epoch 1: 47686batch [11:54, 69.25batch/s]
Epoch 1: 47693batch [11:54, 68.92batch/s]
Epoch 1: 47700batch [11:54, 68.23batch/s]
Epoch 1: 47707batch [11:54, 67.55batch/s]
```

Evaluation Loop

```
from torch.nn import MSELoss
```

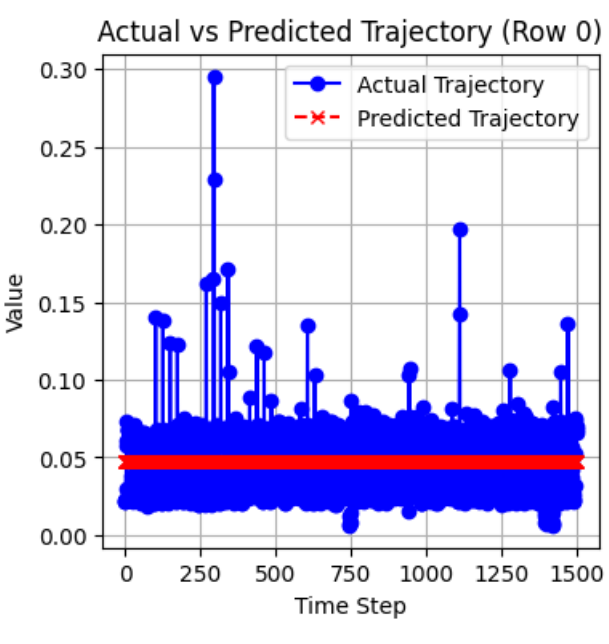
```
train_set = torch.tensor(train_df.values[:,:].astype(np.float32), dtype=torch.float32)
val_set = torch.tensor(val_df.values[:,:].astype(np.float32), dtype=torch.float32)
test_set = torch.tensor(val_df.values[:,:].astype(np.float32), dtype=torch.float32)

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)
    current_input = input_maxtrix

    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:]  #use the last window of training set as initial input
full_trajectories = autoregressive_predict(model, initial_input)


# 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): 0.0021
```

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].numpy()

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

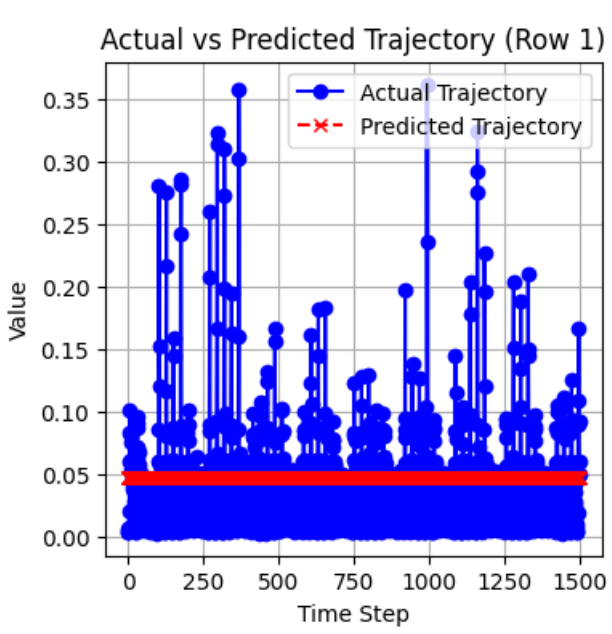



```
# Generate predictions for all the validation dataset
initial_input = train_set[:, -window_length:]
```

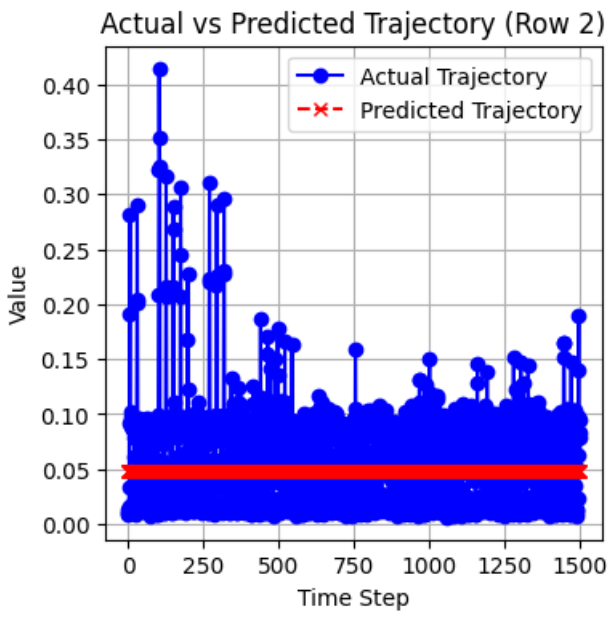
```
val_predictions_tensor = autoregressive_predict(model, initial_input)

# 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)

    # Ensure the shape of pred_val_tensor and pred_test_tensor is correct
    assert pred_val_tensor.shape[0] * pred_val_tensor.shape[1] == original_val_df.shape[0] * (original_val_df.shape[1] - 1)
    assert pred_test_tensor.shape[0] * pred_test_tensor.shape[1] == original_test_df.shape[0] * (original_test_df.shape[1] - 1)

    # Create empty lists to store ids and values
    ids = []
    values = []

    # Process validation set
    for col_idx, col in enumerate(original_val_df.columns[1:]):  # Skip the 'ids' column
        for row_idx, _ in enumerate(original_val_df[col]):
            ids.append(str(f"{col}_traffic_val_{row_idx}"))
            values.append(float(pred_val_tensor[row_idx, col_idx]))

    # Process testing set
    for col_idx, col in enumerate(original_test_df.columns[1:]):  # Skip the 'ids' column
        for row_idx, _ in enumerate(original_test_df[col]):
            ids.append(str(f"{col}_traffic_test_{row_idx}"))
            values.append(float(pred_test_tensor[row_idx, col_idx]))

    # Create the submissions dataframe
    submissions_df = pd.DataFrame({
        "ids": ids,
        "value": values
    })

    # Impute any null values
    submissions_df.fillna(100, inplace=True)

    # Assert the shape of the dataframe
    assert submissions_df.shape[1] == 2
    assert submissions_df.shape[0] == (original_val_df.shape[0] * (original_val_df.shape[1] - 1)) + (original_test_df.shape[0] * (original_test_df.shape[1] - 1))
    assert "ids" in submissions_df.columns
    assert "value" in submissions_df.columns

    # Save to CSV
    submissions_df.to_csv('submissions_v3.csv', index=False)

# Call the function
generate_submissions_v4(val_predictions_tensor, test_predictions_tensor, 'val.csv', 'test.csv')


# 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].numpy()

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

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



Start coding or generate with AI.