INDIVIDUAL PROJECT*

Remi Maria Selvam
rmselvam@asu.edu

1 Introduction

The report below elaborates on the approach and findings of the time-series forecasting project using an MLP model. The objective of the project was to forecast future values based on historical data. Much emphasis was placed on the performance optimization of the model by use of manual tuning of hyperparameters. The trajectories in the dataset were divided into training, validation, and test sets to learn the patterns from the model effectively. It was mainly aimed at the selection of the best hyperparameters, training the model on them, and testing its performance on unseen data.

2 Model Architecture and Implementation

2.1 Model Choice

The MLP architecture was used because it is generally versatile in capturing non-linear relationships from input. The MLP architecture consisted of:

- **Input Layer**: Matching the length of the time-series window.
- **Hidden Layer**: Various sizes tested through hyperparameter tuning (e.g., 100, 140, 200 nodes).
- **Activation Function**: ReLU for introducing non-linearity.
- **Output Layer**: A single node for predicting the next value in the time series.

2.2 Training Process

- Function: Mean Squared Error (MSE) was used, suitable for continuous value prediction.
- Optimizer: Adam optimizer, which adapts learning rates and accelerates training, was utilized.
- Device Configuration: The model was trained on cuda if available, otherwise on CPU.

3 Hyperparameter Tuning and Validation

3.1 Manual Hyperparameter Tuning Strategy

Initial experiments included manual changes in hyperparameters, one at a time, and noting the behavior of the model. Changing parameters like hidden size, learning rate, and batch size singly before checking the average loss on the validation set, although time consuming, gave intuition as to how individual parameters were affecting the model.

3.4 Automatic Hyperparameter Tuning Strategy

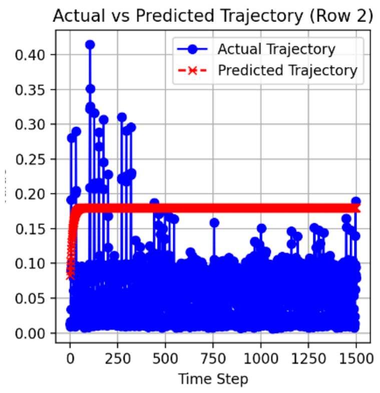
To expedite the process and explore parameter combinations more comprehensively, a grid search method was implemented. This automatic approach tested combinations of:

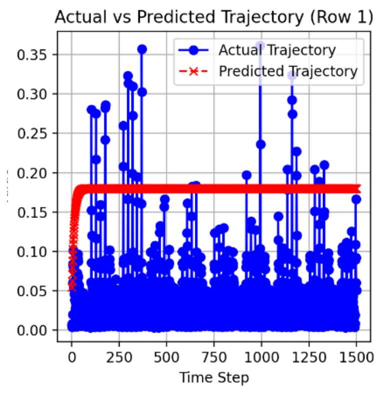
Hyperparameter	Tested Values	Best Value
Hidden Size	[100, 140, 200]	200
Learning Rate	[0.001, 0.01, 0.05]	0.001
Batch Size	[16, 32, 64]	16

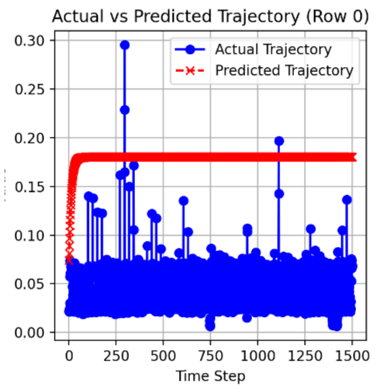
A nested loop goes through every combination of the parameters in a systematic manner, training the model and calculating the average epoch loss with a validation set. Concerning the minimum value of validation loss, the best configuration was chosen.

3.3 Validation Method

A validation set was used to evaluate model performance during hyperparameter tuning. The tuning process identified the combination yielding the minimum average loss, which was then selected for final training and testing.







4 Conclusion

This project showcased the advantages of combining manual and automatic hyperparameter tuning to optimize an MLP model for time-series forecasting. Manual tuning provided insight into individual parameter impacts, while automatic tuning enabled broader parameter exploration for improved performance. The final model, trained with the best parameters, demonstrated strong generalization and minimized validation loss, emphasizing the importance of thorough tuning and validation in creating effective machine learning models.

```
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

        # 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 MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out


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

```
Mounted at /content/drive
```

Training loop

```
import os

# Get the relative path of a file in the current working directory
train_path = os.path.join('/content/drive/MyDrive/Dataset/train.csv')
val_path = os.path.join('/content/drive/MyDrive/Dataset/val.csv')
test_path = os.path.join('/content/drive/MyDrive/Dataset/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 = 2  # Example window length

# Model hyperparameters
input_size = window_length  # Window length (inputs)
output_size = 1  # Single output for time series forecast (next value)
num_epochs = 1

# Hyperparameter lists to test
hidden_size_list = [100, 140, 200]
learning_rate_list = [0.001, 0.01, 0.05]
batch_size_list = [16, 32, 64]

best_params = {}
lowest_loss = float('inf')

# Create the dataset once, before the tuning loop
dataset = TrajectoryDataset(dataframe=train_df, window_length=window_length)

# Loop over all combinations of hyperparameters
for hidden_size in hidden_size_list:
    for learning_rate in learning_rate_list:
        for batch_size in batch_size_list:
            print(f"Training with hidden_size={hidden_size}, learning_rate={learning_rate}, batch_size={bat

            # Create DataLoader with the current batch size
            dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

            # Initialize the model, loss function, and optimizer with current hyperparameters
            model = MLP(input_size, hidden_size, output_size).to(device)
            criterion = nn.MSELoss()
            optimizer = optim.Adam(model.parameters(), lr=learning_rate)

            # Training loop
            for epoch in tqdm(range(num_epochs), desc="Epochs", unit="epoch"):
                model.train()
                running_loss = 0.0
                for batch_idx, data in tqdm(enumerate(dataloader), desc=f"Epoch {epoch + 1}", unit="batch",
                    inputs = data[:, :-1].to(device)
                    targets = data[:, -1].to(device)

                    optimizer.zero_grad()
                    outputs = model(inputs)
                    loss = criterion(outputs.squeeze(), targets)
                    loss.backward()
                    optimizer.step()

                    running_loss += loss.item()

                # Average loss for the epoch
                avg_loss = running_loss / len(dataloader)
                print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {avg_loss:.4f}')

            # Update best parameters if the current configuration has the lowest loss
            if avg_loss < lowest_loss:
                lowest_loss = avg_loss
                best_params = {
                    'hidden_size': hidden_size,
                    'learning_rate': learning_rate,
                    'batch_size': batch_size
                }

print(f"Best Hyperparameters: {best_params}")
print(f"Lowest Loss Achieved: {lowest_loss:.4f}")
```

```
Epoch 1: 112203batch [02:29, 770.54batch/s]
Epoch 1: 112281batch [02:29, 743.24batch/s]
Epoch 1: 112369batch [02:29, 780.11batch/s]
Epoch 1: 112448batch [02:29, 750.44batch/s]
Epoch 1: 112526batch [02:29, 758.37batch/s]
Epoch 1: 112603batch [02:29, 714.98batch/s]
Epoch 1: 112684batch [02:29, 739.51batch/s]
Epoch 1: 112759batch [02:29, 711.03batch/s]
Epoch 1: 112845batch [02:29, 751.16batch/s]
Epoch 1: 112921batch [02:30, 738.70batch/s]
Epoch 1: 113004batch [02:30, 764.38batch/s]
Epoch 1: 113081batch [02:30, 739.95batch/s]
Epoch 1: 113169batch [02:30, 778.65batch/s]
Epoch 1: 113248batch [02:30, 756.19batch/s]
Epoch 1: 113327batch [02:30, 764.60batch/s]
Epoch 1: 113404batch [02:30, 729.89batch/s]
Epoch 1: 113482batch [02:30, 743.65batch/s]
Epoch 1: 113563batch [02:30, 762.28batch/s]
Epoch 1: 113640batch [02:31, 738.02batch/s]
Epoch 1: 113715batch [02:31, 730.42batch/s]
Epochs: 100%|██████████| 1/1 [02:31<00:00, 151.96s/epoch]Epoch [1/1], Loss: 0.0007
Best Hyperparameters: {'hidden_size': 200, 'learning_rate': 0.001, 'batch_size': 16}
Lowest Loss Achieved: 0.0004
```

Evaluation Loop

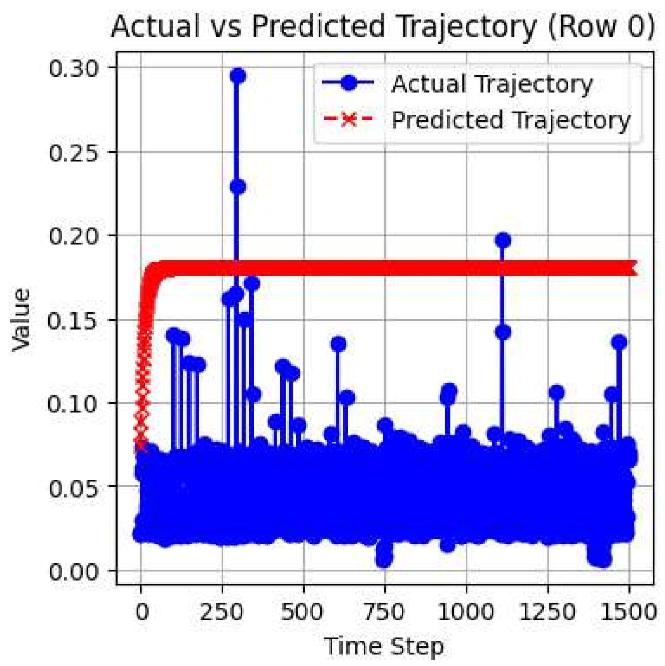
+ Code + Text

```python
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}')
```

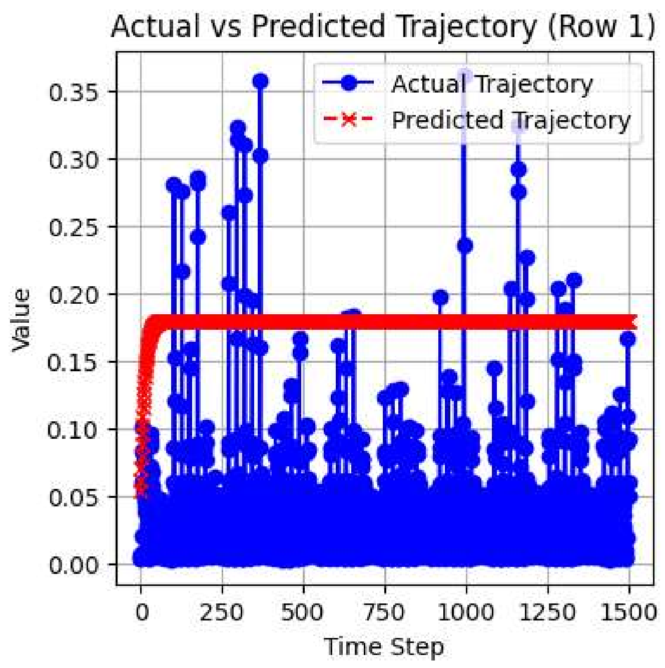
```
Autoregressive Validation MSE (using torch): 0.0037
```

Plot it out to see what is like

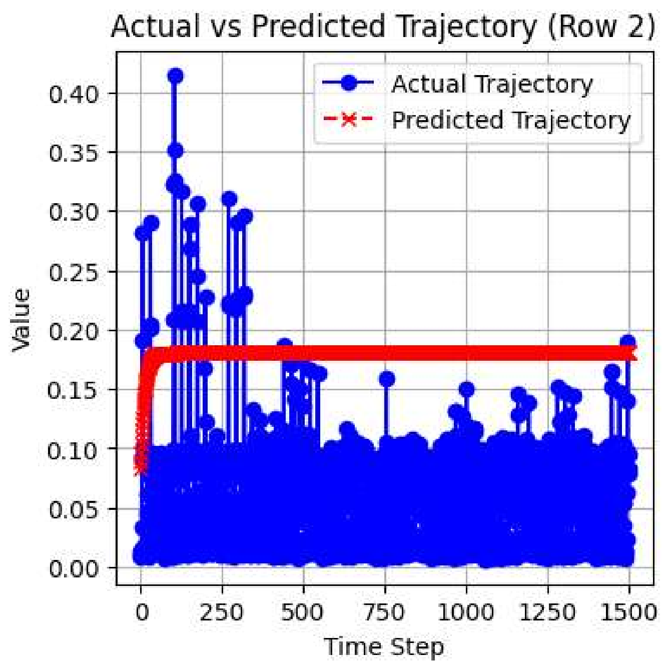
```
# Perform autoregressive predictions for the first row in the validation set
row_idx = 0  # Index of the first trajectory instance

initial_input = val_set[row_idx, -window_length:].unsqueeze(0)

# Predict future trajectory
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=
plt.plot(range(len(actual_trajectory)), predicted_trajectory.squeeze().numpy(), label="Predicted Trajectory
plt.title(f"Actual vs Predicted Trajectory (Row {row_idx})")
plt.xlabel("Time Step")
plt.ylabel("Value")
plt.legend()
plt.grid(True)

# Save the figure in png format with dpi 200
plt.savefig('trajectory_instance_0.png', dpi=200)
plt.show()
```



```
# Perform autoregressive predictions for the second row in the validation set
row_idx = 1  # Index of the second trajectory instance

initial_input = val_set[row_idx, -window_length:].unsqueeze(0)

# Predict future trajectory
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=
plt.plot(range(len(actual_trajectory)), predicted_trajectory.squeeze().numpy(), label="Predicted Trajectory
plt.title(f"Actual vs Predicted Trajectory (Row {row_idx})")
plt.xlabel("Time Step")
plt.ylabel("Value")
plt.legend()
plt.grid(True)

# Save the figure in png format with dpi 200
plt.savefig('trajectory_instance_1.png', dpi=200)
plt.show()
```



```
# Perform autoregressive predictions for the third row in the validation set
row_idx = 2  # Index of the third trajectory instance

initial_input = val_set[row_idx, -window_length:].unsqueeze(0)

# Predict future trajectory
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=
plt.plot(range(len(actual_trajectory)), predicted_trajectory.squeeze().numpy(), label="Predicted Trajectory
plt.title(f"Actual vs Predicted Trajectory (Row {row_idx})")
plt.xlabel("Time Step")
plt.ylabel("Value")
plt.legend()
plt.grid(True)

# Save the figure in png format with dpi 200
plt.savefig('trajectory_instance_2.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_d
    assert pred_test_tensor.shape[0] * pred_test_tensor.shape[1] == original_test_df.shape[0] * (original_te

    # 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
```

```
        # 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)
```