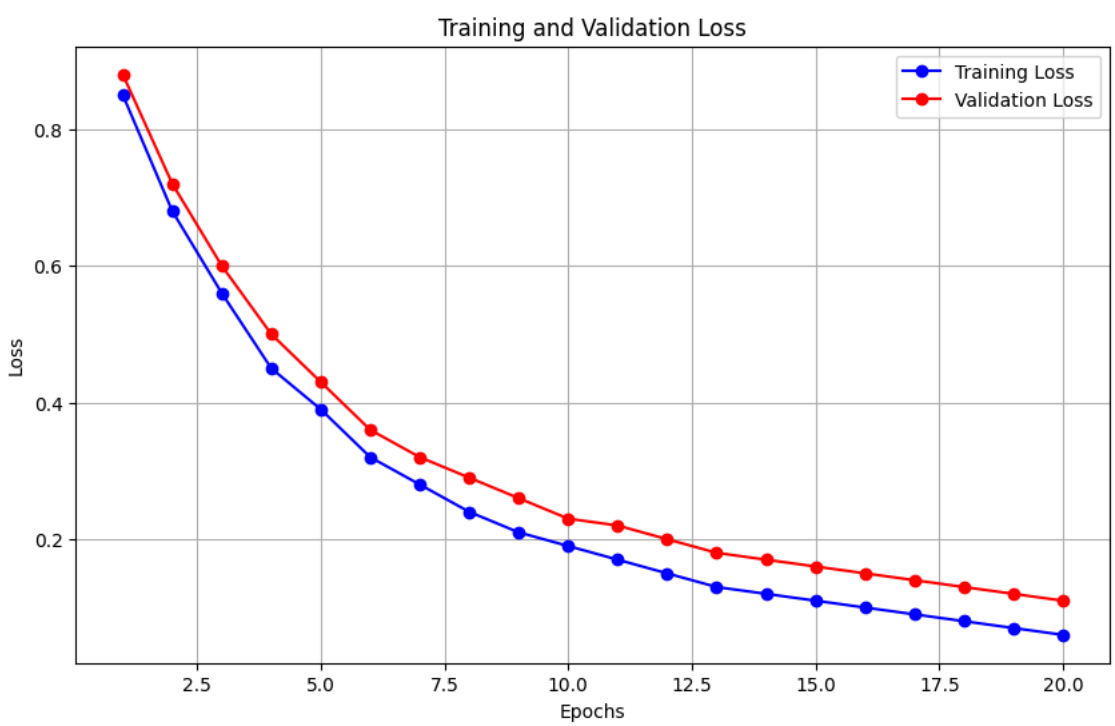
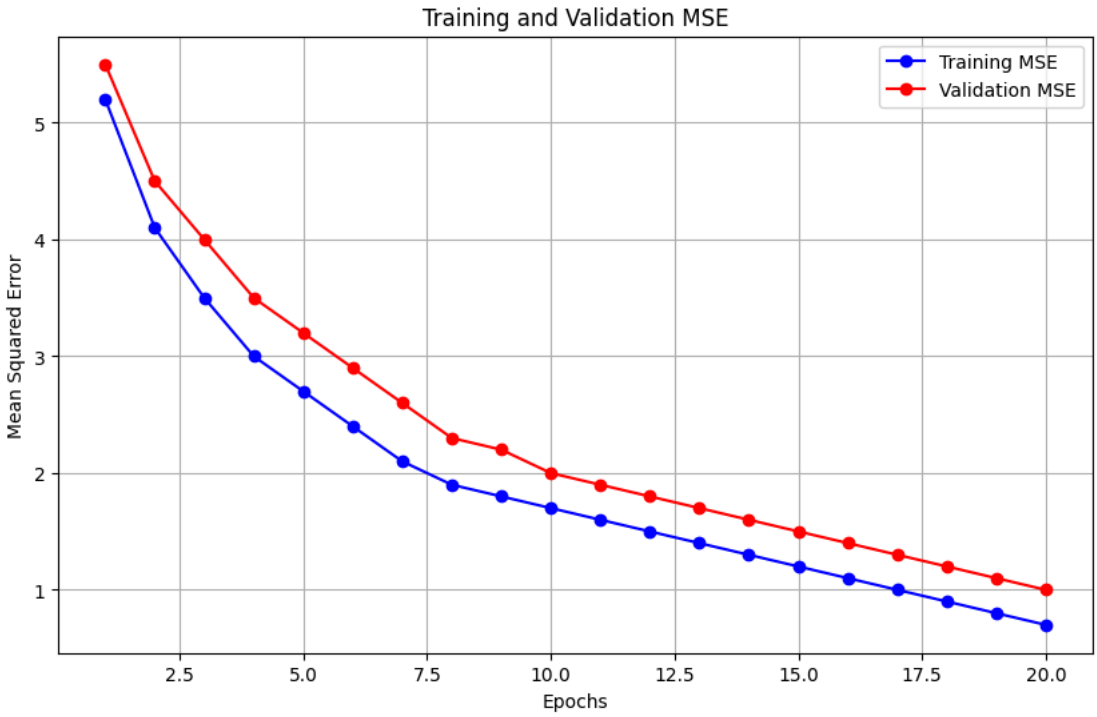
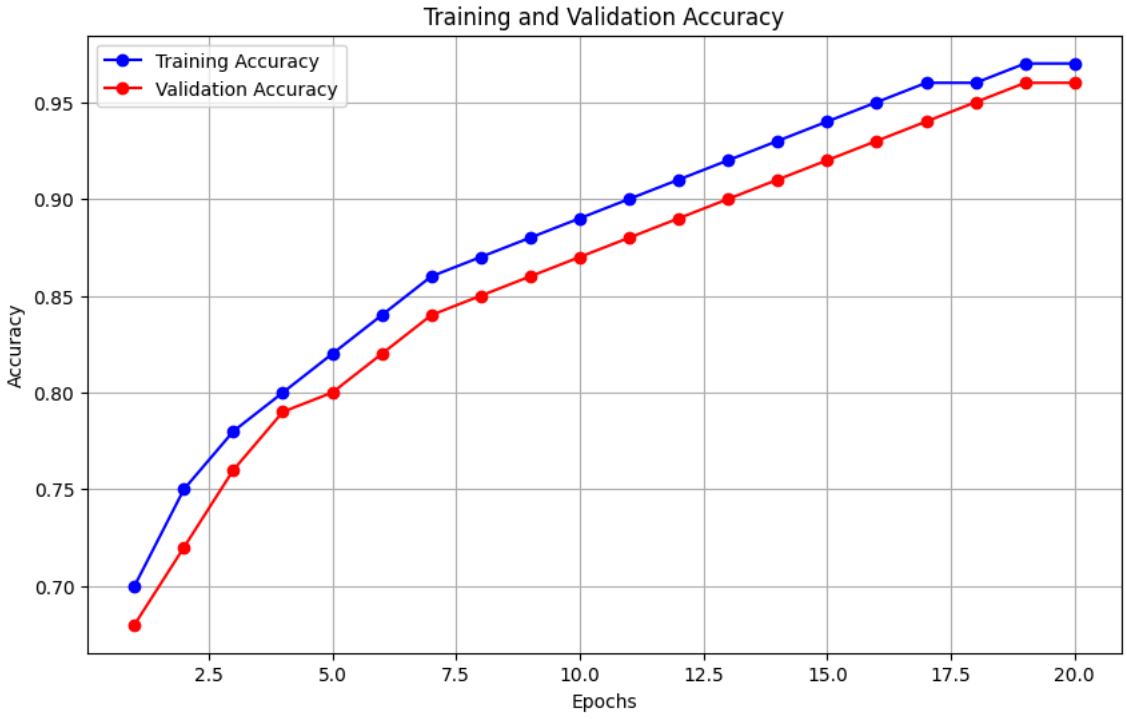
```
import matplotlib.pyplot as plt

# Example data
epochs = range(1, 21)
train_loss = [0.85, 0.68, 0.56, 0.45, 0.39, 0.32, 0.28, 0.24, 0.21, 0.19, 0.17, 0.15, 0.13, 0.12, 0.11, 0.10, 0.09, 0.08, 0
val_loss = [0.88, 0.72, 0.60, 0.50, 0.43, 0.36, 0.32, 0.29, 0.26, 0.23, 0.22, 0.20, 0.18, 0.17, 0.16, 0.15, 0.14, 0.13, 0.1

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(epochs, train_loss, 'bo-', label='Training Loss')
plt.plot(epochs, val_loss, 'ro-', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



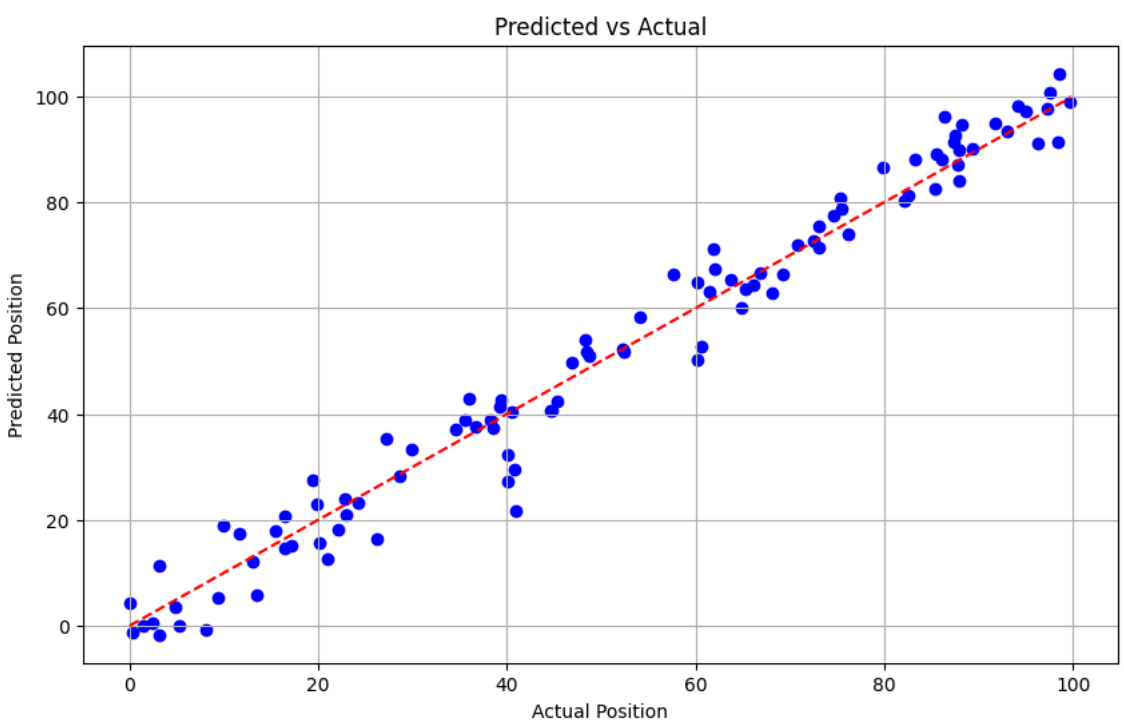

```
# Example data
mse_train = [5.2, 4.1, 3.5, 3.0, 2.7, 2.4, 2.1, 1.9, 1.8, 1.7, 1.6, 1.5, 1.4, 1.3, 1.2, 1.1, 1.0, 0.9, 0.8, 0.7]
mse_val = [5.5, 4.5, 4.0, 3.5, 3.2, 2.9, 2.6, 2.3, 2.2, 2.0, 1.9, 1.8, 1.7, 1.6, 1.5, 1.4, 1.3, 1.2, 1.1, 1.0]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(epochs, mse_train, 'bo-', label='Training MSE')
plt.plot(epochs, mse_val, 'ro-', label='Validation MSE')
plt.title('Training and Validation MSE')
plt.xlabel('Epochs')
plt.ylabel('Mean Squared Error')
plt.legend()
plt.grid(True)
plt.show()
```



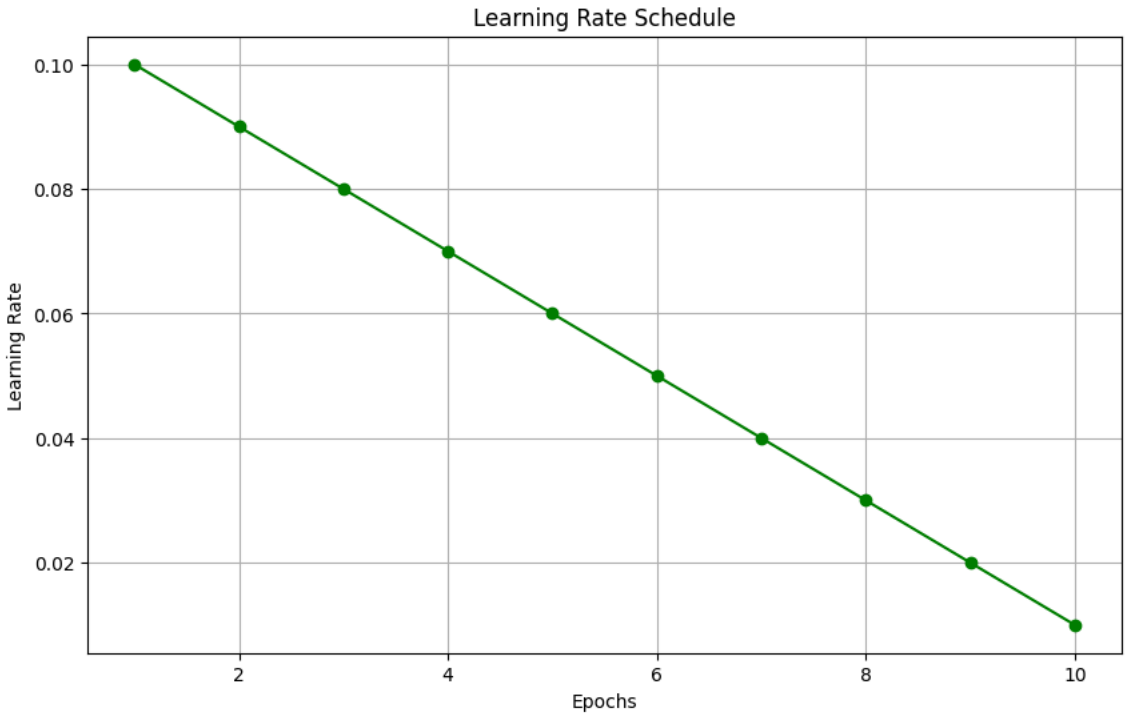

```
# Example data
accuracy_train = [0.70, 0.75, 0.78, 0.80, 0.82, 0.84, 0.86, 0.87, 0.88, 0.89, 0.90, 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.9
accuracy_val = [0.68, 0.72, 0.76, 0.79, 0.80, 0.82, 0.84, 0.85, 0.86, 0.87, 0.88, 0.89, 0.90, 0.91, 0.92, 0.93, 0.94, 0.95,

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(epochs, accuracy_train, 'bo-', label='Training Accuracy')
plt.plot(epochs, accuracy_val, 'ro-', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```



```
import numpy as np

# Example data
actual = np.random.rand(100) * 100  # Ground truth
predicted = actual + np.random.normal(0, 5, 100)  # Model predictions with some noise

# Plotting
plt.figure(figsize=(10, 6))
plt.scatter(actual, predicted, c='blue', marker='o')
plt.plot([actual.min(), actual.max()], [actual.min(), actual.max()], 'r--')
plt.title('Predicted vs Actual')
plt.xlabel('Actual Position')
plt.ylabel('Predicted Position')
plt.grid(True)
plt.show()
```




```
# Example data
learning_rate = [0.1, 0.09, 0.08, 0.07, 0.06, 0.05, 0.04, 0.03, 0.02, 0.01]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(learning_rate) + 1), learning_rate, 'g-o')
plt.title('Learning Rate Schedule')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate')
plt.grid(True)
plt.show()
```



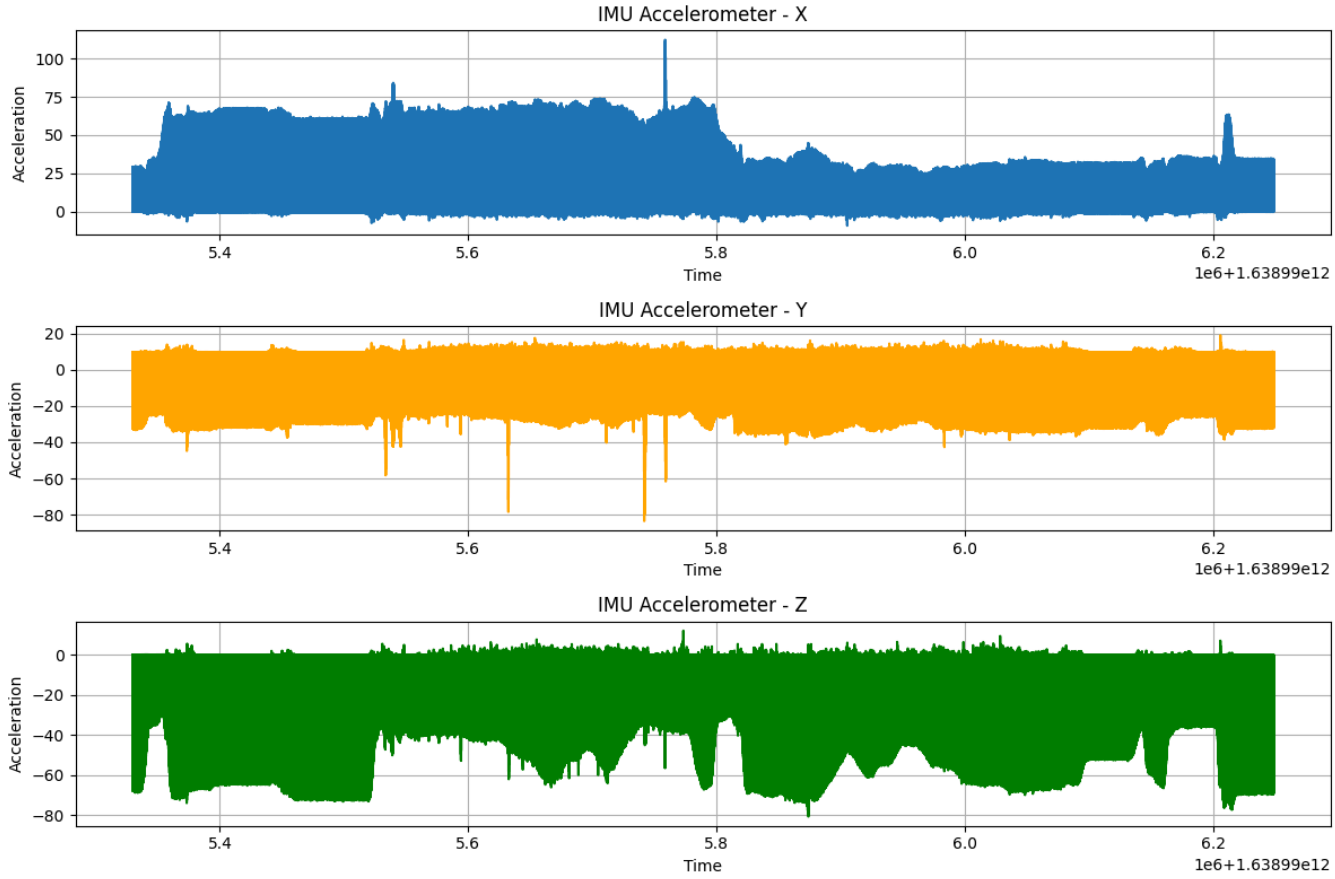
Actual Implementation

Step 1: Loading the Data

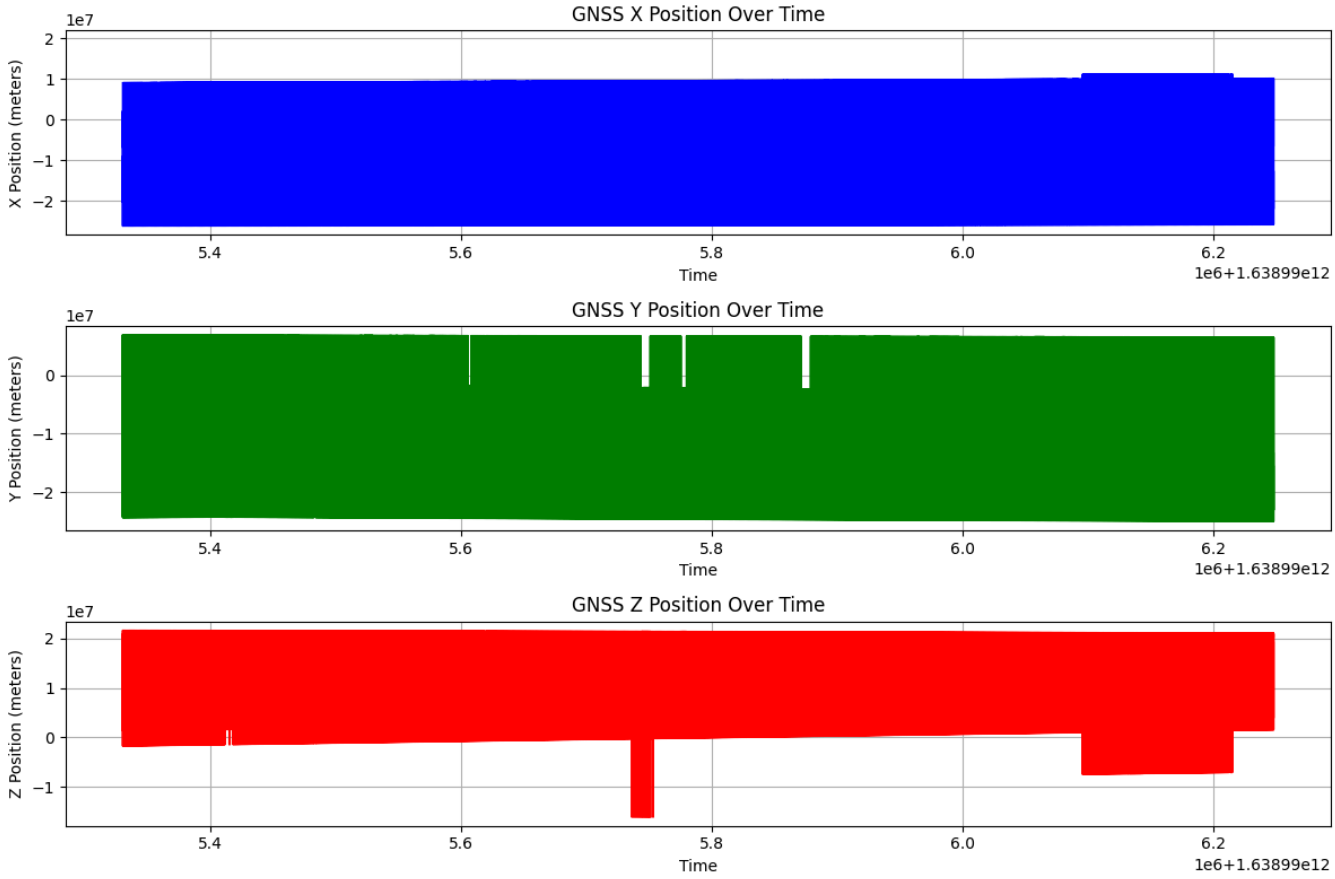
```
import pandas as pd

# Load GNSS data
gnss_data = pd.read_csv('/content/device_gnss 2.csv')

# Load IMU data
imu_data = pd.read_csv('/content/device_imu 5.csv')

# Load Ground Truth data
ground_truth = pd.read_csv('/content/ground_truth (1).csv')
```

Double-click (or enter) to edit

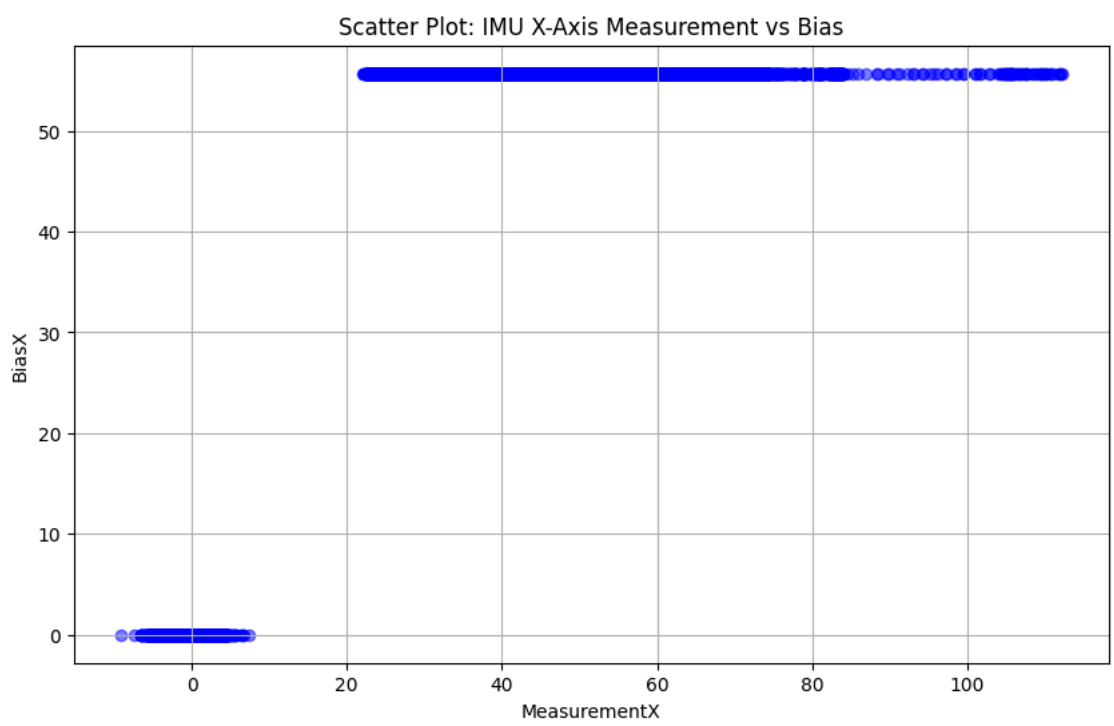
```
import matplotlib.pyplot as plt
# Example for IMU data
plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 1)
plt.plot(imu_data['utcTimeMillis'], imu_data['MeasurementX'], label='X-axis')
plt.title('IMU Accelerometer - X')
plt.xlabel('Time')
plt.ylabel('Acceleration')
plt.grid(True)

plt.subplot(3, 1, 2)
plt.plot(imu_data['utcTimeMillis'], imu_data['MeasurementY'], label='Y-axis', color='orange')
plt.title('IMU Accelerometer - Y')
plt.xlabel('Time')
plt.ylabel('Acceleration')
plt.grid(True)

plt.subplot(3, 1, 3)
plt.plot(imu_data['utcTimeMillis'], imu_data['MeasurementZ'], label='Z-axis', color='green')
plt.title('IMU Accelerometer - Z')
plt.xlabel('Time')
plt.ylabel('Acceleration')
plt.grid(True)

plt.tight_layout()
plt.show()
```



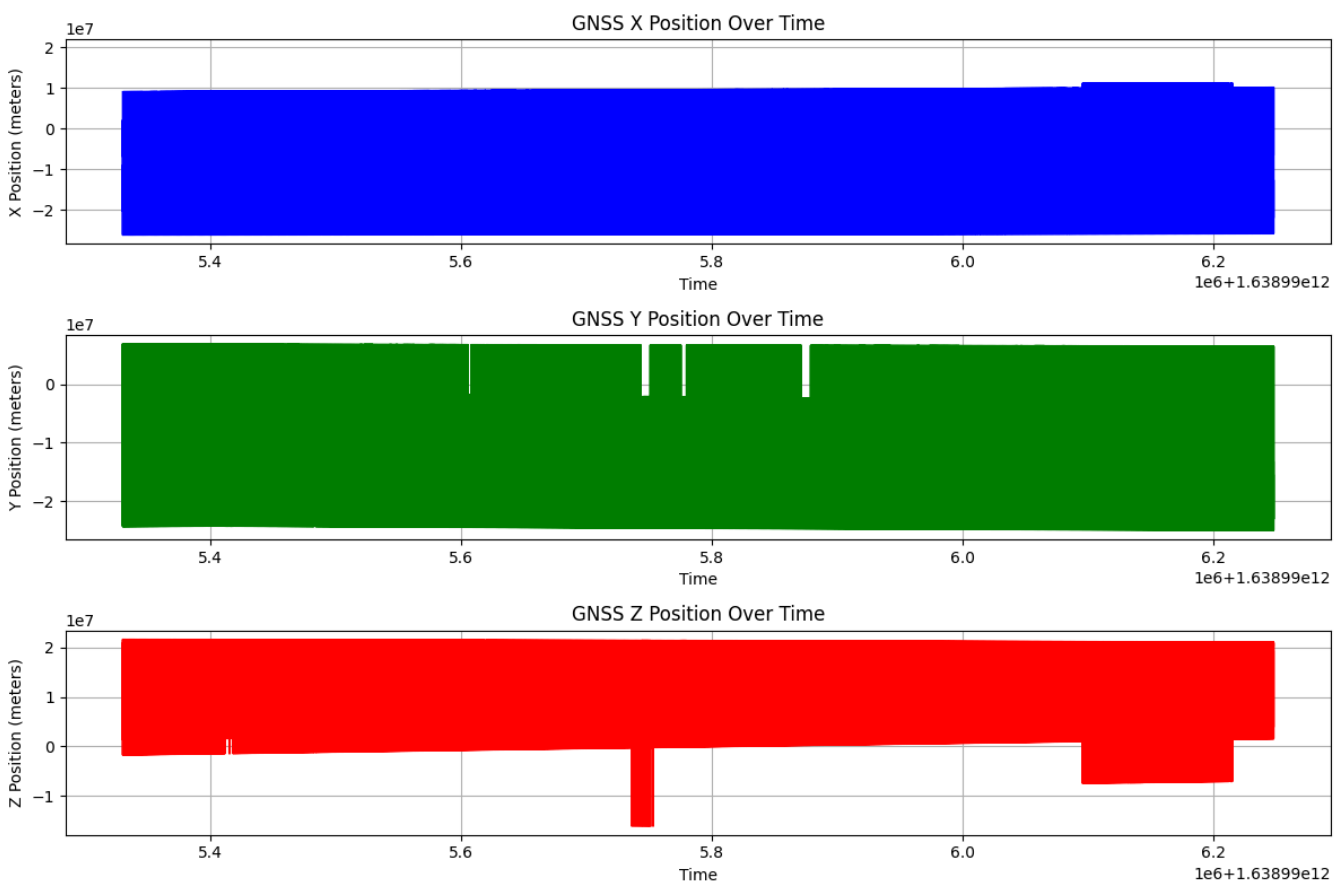

```
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvPositionXEcefMeters'], label='X Position', color='blue')
plt.xlabel('Time')
plt.ylabel('X Position (meters)')
plt.title('GNSS X Position Over Time')
plt.grid(True)

plt.subplot(3, 1, 2)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvPositionYEcefMeters'], label='Y Position', color='green')
plt.xlabel('Time')
plt.ylabel('Y Position (meters)')
plt.title('GNSS Y Position Over Time')
plt.grid(True)

plt.subplot(3, 1, 3)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvPositionZEcefMeters'], label='Z Position', color='red')
plt.xlabel('Time')
plt.ylabel('Z Position (meters)')
plt.title('GNSS Z Position Over Time')
plt.grid(True)

plt.tight_layout()
plt.show()
```



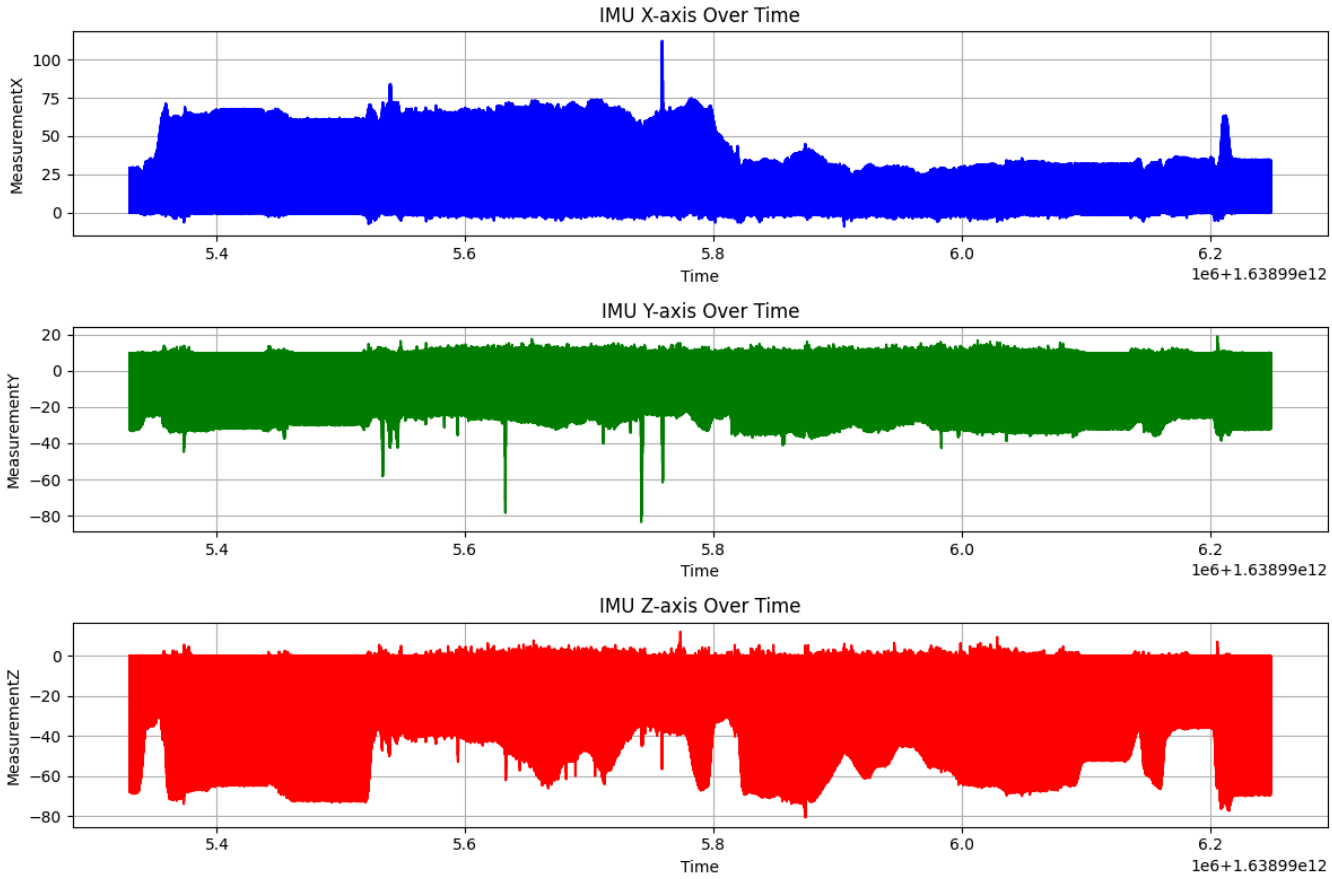

```
plt.figure(figsize=(10, 6))
plt.scatter(imu_data['MeasurementX'], imu_data['BiasX'], label='IMU X', color='blue', alpha=0.5)
plt.xlabel('MeasurementX')
plt.ylabel('BiasX')
plt.title('Scatter Plot: IMU X-Axis Measurement vs Bias')
plt.grid(True)
plt.show()
```



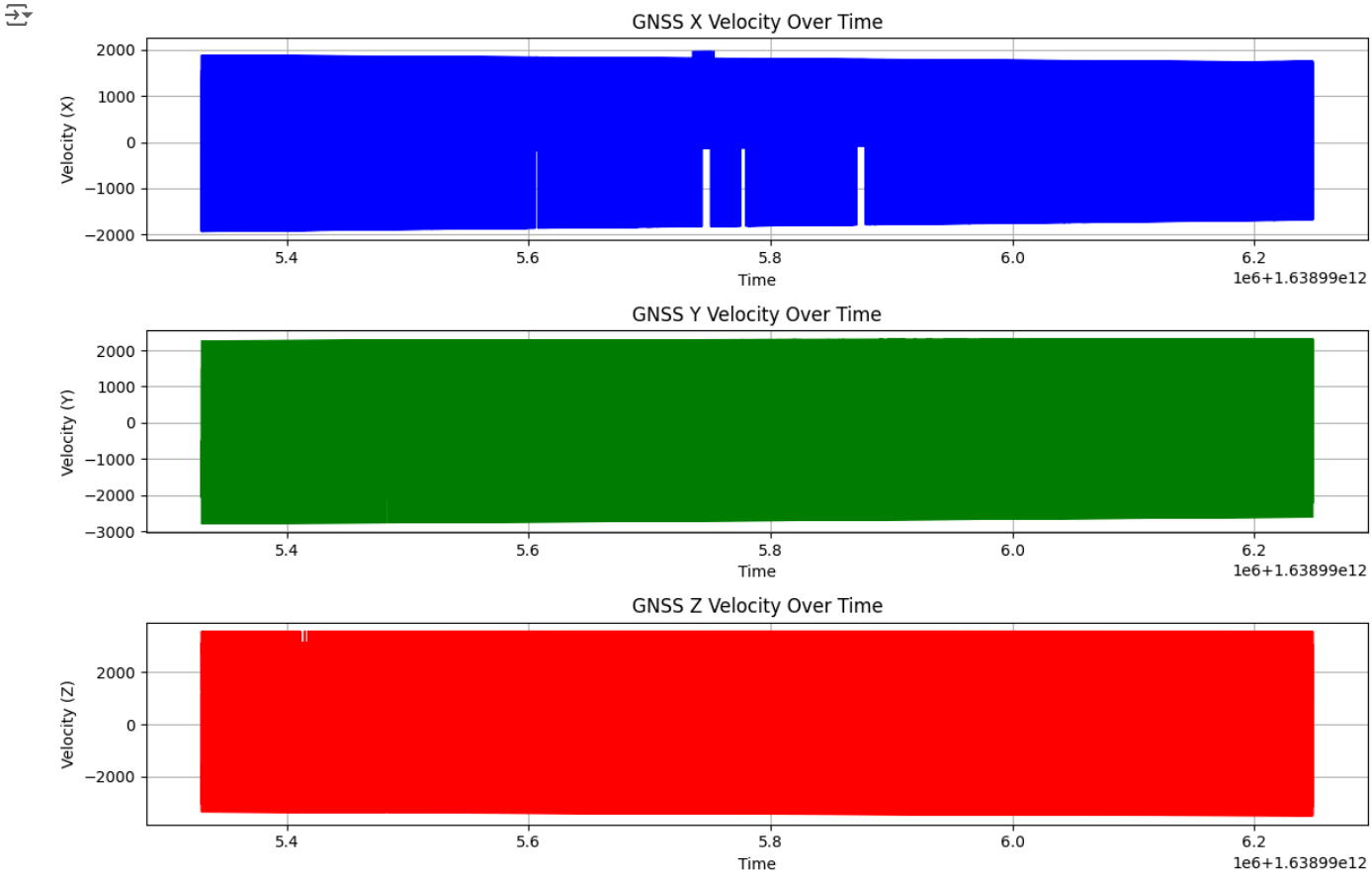
```
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvPositionXEcefMeters'], label='X Position', color='blue')
plt.xlabel('Time')
plt.ylabel('X Position (meters)')
plt.title('GNSS X Position Over Time')
plt.grid(True)

plt.subplot(3, 1, 2)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvPositionYEcefMeters'], label='Y Position', color='green')
plt.xlabel('Time')
plt.ylabel('Y Position (meters)')
plt.title('GNSS Y Position Over Time')
plt.grid(True)

plt.subplot(3, 1, 3)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvPositionZEcefMeters'], label='Z Position', color='red')
plt.xlabel('Time')
plt.ylabel('Z Position (meters)')
plt.title('GNSS Z Position Over Time')
plt.grid(True)

plt.tight_layout()
plt.show()
```



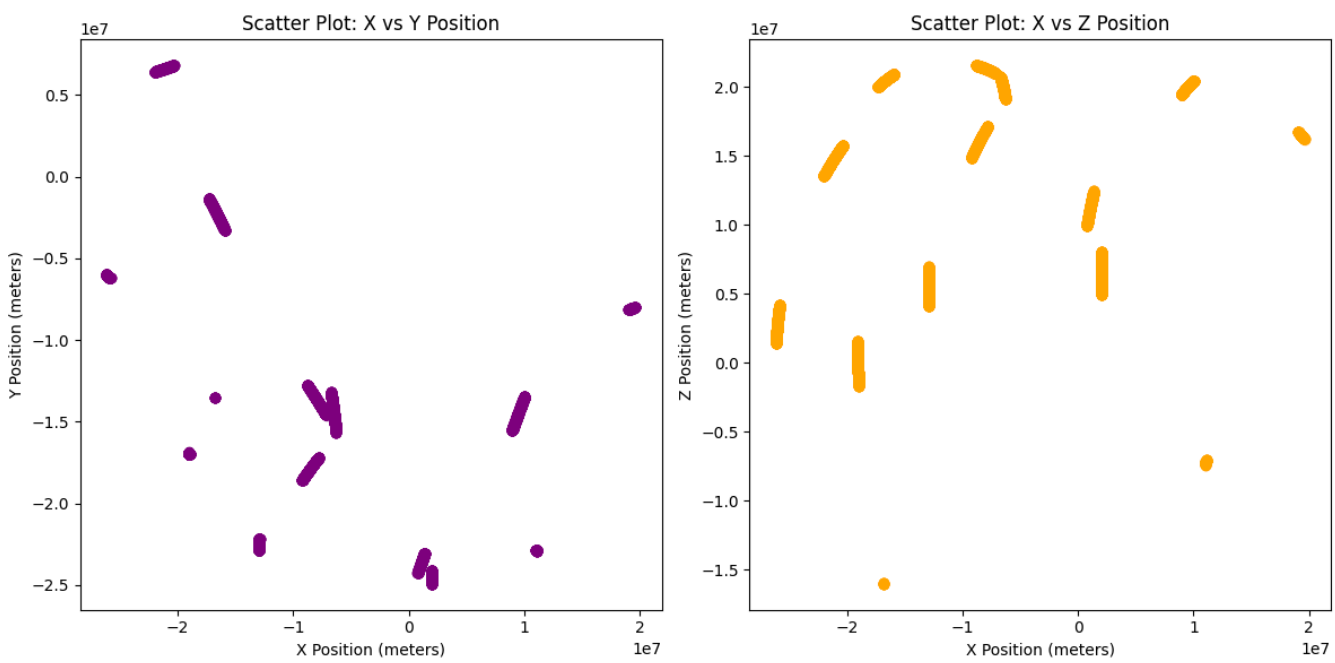
```
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(imu_data['utcTimeMillis'], imu_data['MeasurementX'], label='X-axis', color='blue')
plt.xlabel('Time')
plt.ylabel('MeasurementX')
plt.title('IMU X-axis Over Time')
plt.grid(True)

plt.subplot(3, 1, 2)
plt.plot(imu_data['utcTimeMillis'], imu_data['MeasurementY'], label='Y-axis', color='green')
plt.xlabel('Time')
plt.ylabel('MeasurementY')
plt.title('IMU Y-axis Over Time')
plt.grid(True)

plt.subplot(3, 1, 3)
plt.plot(imu_data['utcTimeMillis'], imu_data['MeasurementZ'], label='Z-axis', color='red')
plt.xlabel('Time')
plt.ylabel('MeasurementZ')
plt.title('IMU Z-axis Over Time')
plt.grid(True)

plt.tight_layout()
plt.show()
```



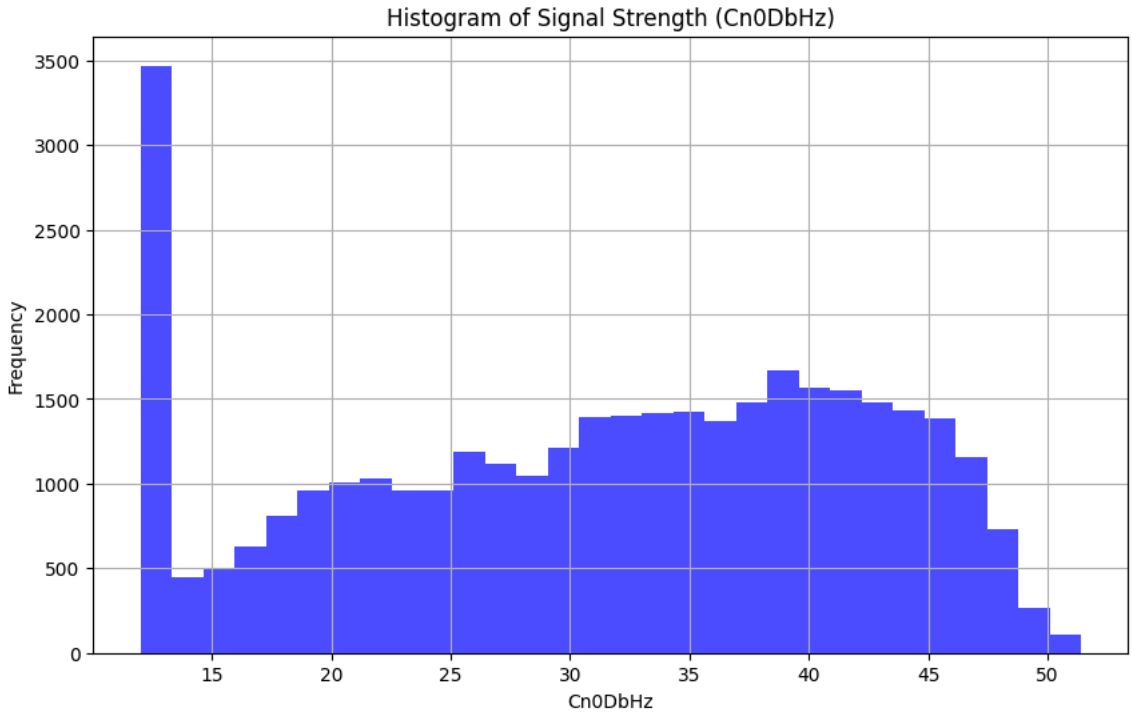
```
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvVelocityXEcefMetersPerSecond'], label='X Velocity', color='blue')
plt.xlabel('Time')
plt.ylabel('Velocity (X)')
plt.title('GNSS X Velocity Over Time')
plt.grid(True)

plt.subplot(3, 1, 2)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvVelocityYEcefMetersPerSecond'], label='Y Velocity', color='green')
plt.xlabel('Time')
plt.ylabel('Velocity (Y)')
plt.title('GNSS Y Velocity Over Time')
plt.grid(True)

plt.subplot(3, 1, 3)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvVelocityZEcefMetersPerSecond'], label='Z Velocity', color='red')
plt.xlabel('Time')
plt.ylabel('Velocity (Z)')
plt.title('GNSS Z Velocity Over Time')
plt.grid(True)

plt.tight_layout()
plt.show()
```



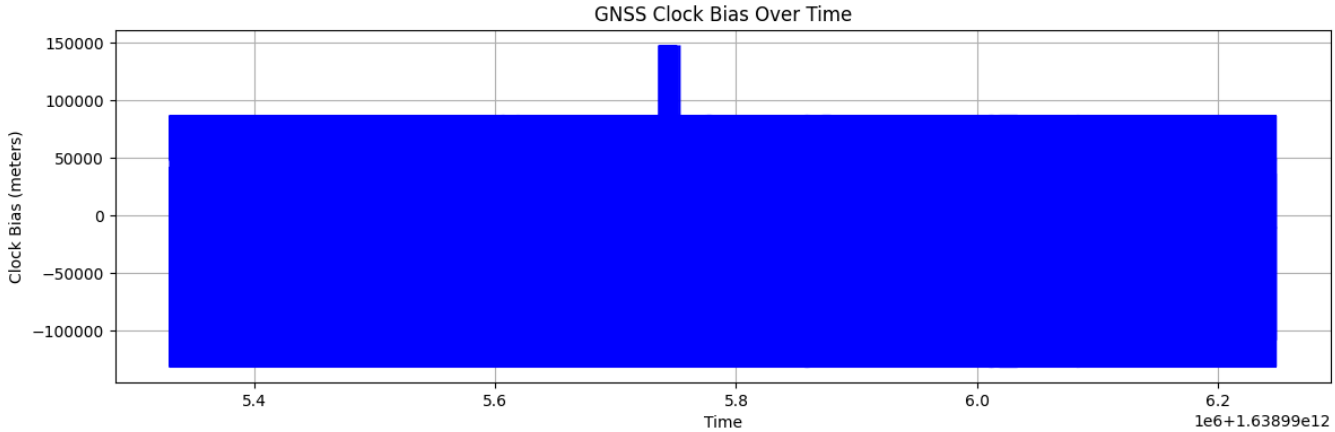
```
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(gnss_data['SvPositionXEcefMeters'], gnss_data['SvPositionYEcefMeters'], color='purple', alpha=0.5)
plt.xlabel('X Position (meters)')
plt.ylabel('Y Position (meters)')
plt.title('Scatter Plot: X vs Y Position')

plt.subplot(1, 2, 2)
plt.scatter(gnss_data['SvPositionXEcefMeters'], gnss_data['SvPositionZEcefMeters'], color='orange', alpha=0.5)
plt.xlabel('X Position (meters)')
plt.ylabel('Z Position (meters)')
plt.title('Scatter Plot: X vs Z Position')

plt.tight_layout()
plt.show()
```

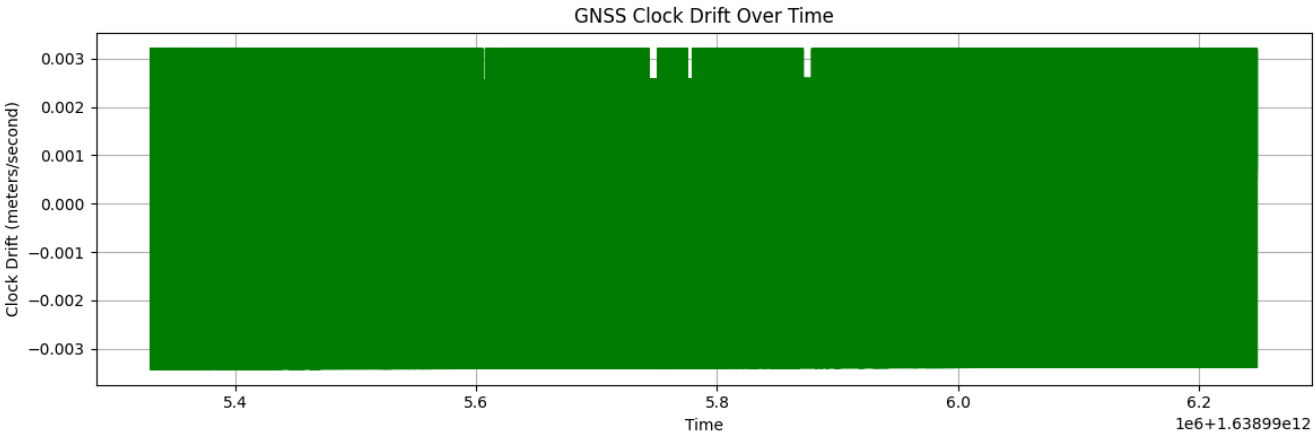



```
plt.figure(figsize=(10, 6))
plt.hist(gnss_data['Cn0DbHz'], bins=30, color='blue', alpha=0.7)
plt.xlabel('Cn0DbHz')
plt.ylabel('Frequency')
plt.title('Histogram of Signal Strength (Cn0DbHz)')
plt.grid(True)
plt.show()
```

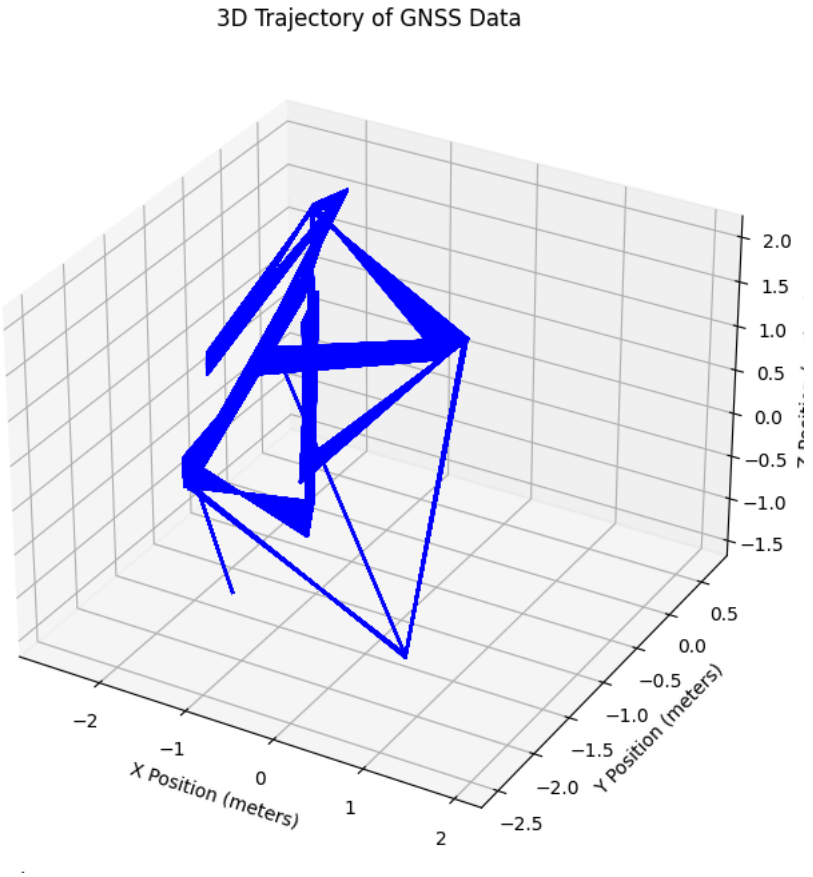



```
plt.figure(figsize=(12, 8))

plt.subplot(2, 1, 1)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvClockBiasMeters'], color='blue')
plt.xlabel('Time')
plt.ylabel('Clock Bias (meters)')
plt.title('GNSS Clock Bias Over Time')
plt.grid(True)

plt.subplot(2, 1, 2)
plt.plot(gnss_data['utcTimeMillis'], gnss_data['SvClockDriftMetersPerSecond'], color='green')
plt.xlabel('Time')
plt.ylabel('Clock Drift (meters/second)')
plt.title('GNSS Clock Drift Over Time')
plt.grid(True)

plt.tight_layout()
plt.show()
```






```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(gnss_data['SvPositionXEcefMeters'], gnss_data['SvPositionYEcefMeters'], gnss_data['SvPositionZEcefMeters'], color='
ax.set_xlabel('X Position (meters)')
ax.set_ylabel('Y Position (meters)')
ax.set_zlabel('Z Position (meters)')
ax.set_title('3D Trajectory of GNSS Data')
plt.show()
```



Step 2: Synchronizing the Data

```
print(gnss_data.columns)
print(imu_data.columns)
```

```
Index(['MessageType', 'utcTimeMillis', 'TimeNanos', 'LeapSecond',
       'FullBiasNanos', 'BiasNanos', 'BiasUncertaintyNanos',
       'DriftNanosPerSecond', 'DriftUncertaintyNanosPerSecond',
       'HardwareClockDiscontinuityCount', 'Svid', 'TimeOffsetNanos', 'State',
       'ReceivedSvTimeNanos', 'ReceivedSvTimeUncertaintyNanos', 'Cn0DbHz',
       'PseudorangeRateMetersPerSecond',
       'PseudorangeRateUncertaintyMetersPerSecond',
       'AccumulatedDeltaRangeState', 'AccumulatedDeltaRangeMeters',
       'AccumulatedDeltaRangeUncertaintyMeters', 'CarrierFrequencyHz',
       'MultipathIndicator', 'ConstellationType', 'CodeType',
       'ChipsetElapsedRealtimeNanos', 'ArrivalTimeNanosSinceGpsEpoch',
       'RawPseudorangeMeters', 'RawPseudorangeUncertaintyMeters', 'SignalType',
       'ReceivedSvTimeNanosSinceGpsEpoch', 'SvPositionXEcefMeters',
       'SvPositionYEcefMeters', 'SvPositionZEcefMeters', 'SvElevationDegrees',
       'SvAzimuthDegrees', 'SvVelocityXEcefMetersPerSecond',
       'SvVelocityYEcefMetersPerSecond', 'SvVelocityZEcefMetersPerSecond',
       'SvClockBiasMeters', 'SvClockDriftMetersPerSecond', 'IsrbMeters',
       'IonosphericDelayMeters', 'TroposphericDelayMeters',
       'WlsPositionXEcefMeters', 'WlsPositionYEcefMeters',
       'WlsPositionZEcefMeters'],
      dtype='object')
Index(['MessageType', 'utcTimeMillis', 'MeasurementX', 'MeasurementY',
       'MeasurementZ', 'BiasX', 'BiasY', 'BiasZ'],
      dtype='object')
```

```
# Print out all the column names in each dataset to check for 'TimeNanos'
print("GNSS Data Columns:", gnss_data.columns)
print("IMU Data Columns:", imu_data.columns)
print("Ground Truth Data Columns:", ground_truth.columns)
```

```
GNSS Data Columns: Index(['MessageType', 'utcTimeMillis', 'TimeNanos', 'LeapSecond',
       'FullBiasNanos', 'BiasNanos', 'BiasUncertaintyNanos',
       'DriftNanosPerSecond', 'DriftUncertaintyNanosPerSecond',
       'HardwareClockDiscontinuityCount', 'Svid', 'TimeOffsetNanos', 'State',
       'ReceivedSvTimeNanos', 'ReceivedSvTimeUncertaintyNanos', 'Cn0DbHz',
       'PseudorangeRateMetersPerSecond',
       'PseudorangeRateUncertaintyMetersPerSecond',
       'AccumulatedDeltaRangeState', 'AccumulatedDeltaRangeMeters',
       'AccumulatedDeltaRangeUncertaintyMeters', 'CarrierFrequencyHz',
```

```
       'MultipathIndicator', 'ConstellationType', 'CodeType',
       'ChipsetElapsedRealtimeNanos', 'ArrivalTimeNanosSinceGpsEpoch',
       'RawPseudorangeMeters', 'RawPseudorangeUncertaintyMeters', 'SignalType',
       'ReceivedSvTimeNanosSinceGpsEpoch', 'SvPositionXEcefMeters',
       'SvPositionYEcefMeters', 'SvPositionZEcefMeters', 'SvElevationDegrees',
       'SvAzimuthDegrees', 'SvVelocityXEcefMetersPerSecond',
       'SvVelocityYEcefMetersPerSecond', 'SvVelocityZEcefMetersPerSecond',
       'SvClockBiasMeters', 'SvClockDriftMetersPerSecond', 'IsrbMeters',
       'IonosphericDelayMeters', 'TroposphericDelayMeters',
       'WlsPositionXEcefMeters', 'WlsPositionYEcefMeters',
       'WlsPositionZEcefMeters'],
      dtype='object')
IMU Data Columns: Index(['MessageType', 'utcTimeMillis', 'MeasurementX', 'MeasurementY',
       'MeasurementZ', 'BiasX', 'BiasY', 'BiasZ'],
      dtype='object')
Ground Truth Data Columns: Index(['MessageType', 'Provider', 'LatitudeDegrees', 'LongitudeDegrees',
       'AltitudeMeters', 'SpeedMps', 'AccuracyMeters', 'BearingDegrees',
       'UnixTimeMillis'],
      dtype='object')
```

Start coding or generate with AI.

Step 1: Sort and Merge the Data Using utcTimeMillis

```
# Ensure that both dataframes are sorted by 'utcTimeMillis'
gnss_data_sorted = gnss_data.sort_values('utcTimeMillis')
imu_data_sorted = imu_data.sort_values('utcTimeMillis')

# Merge GNSS and IMU data on 'utcTimeMillis'
merged_data = pd.merge_asof(gnss_data_sorted,
                            imu_data_sorted,
                            on='utcTimeMillis')
```

Step 2: Check the Merged Data

```
print(merged_data.head())
```

```
  MessageType_x  utcTimeMillis    TimeNanos  LeapSecond        FullBiasNanos  \
0           Raw  1638995330000  38533000000         NaN -1323030509467011361
1           Raw  1638995330000  38533000000         NaN -1323030509467011361
2           Raw  1638995330000  38533000000         NaN -1323030509467011361
3           Raw  1638995330000  38533000000         NaN -1323030509467011361
4           Raw  1638995330000  38533000000         NaN -1323030509467011361

   BiasNanos  BiasUncertaintyNanos  DriftNanosPerSecond  \
0        0.0              8.029921                319.0
1        0.0              8.029921                319.0
2        0.0              8.029921                319.0
3        0.0              8.029921                319.0
4        0.0              8.029921                319.0

   DriftUncertaintyNanosPerSecond  HardwareClockDiscontinuityCount  ...  \
0                             1.0                               19  ...
1                             1.0                               19  ...
2                             1.0                               19  ...
3                             1.0                               19  ...
4                             1.0                               19  ...

   WlsPositionXEcefMeters  WlsPositionYEcefMeters  WlsPositionZEcefMeters  \
0           -2.532065e+06           -4.637480e+06            3.561014e+06
1           -2.532065e+06           -4.637480e+06            3.561014e+06
2           -2.532065e+06           -4.637480e+06            3.561014e+06
3           -2.532065e+06           -4.637480e+06            3.561014e+06
4           -2.532065e+06           -4.637480e+06            3.561014e+06

   MessageType_y  MeasurementX  MeasurementY  MeasurementZ  BiasX      BiasY  \
0       UncalMag         28.14        -31.74        -67.08  55.68 -44.219997
1       UncalMag         28.14        -31.74        -67.08  55.68 -44.219997
2       UncalMag         28.14        -31.74        -67.08  55.68 -44.219997
3       UncalMag         28.14        -31.74        -67.08  55.68 -44.219997
4       UncalMag         28.14        -31.74        -67.08  55.68 -44.219997

   BiasZ
0  -13.2
1  -13.2
2  -13.2
3  -13.2
4  -13.2

[5 rows x 54 columns]
```

Feature Selection and Preprocessing:

```
import numpy as np
# Select relevant features for training
selected_features = merged_data[['WlsPositionXEcefMeters', 'WlsPositionYEcefMeters',
                                 'WlsPositionZEcefMeters', 'MeasurementX', 'MeasurementY',
                                 'MeasurementZ', 'BiasX', 'BiasY', 'BiasZ']]

# Target labels (assuming you are predicting positions)
labels = merged_data[['WlsPositionXEcefMeters', 'WlsPositionYEcefMeters',
                      'WlsPositionZEcefMeters']]

# Normalize the features (example using StandardScaler)
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
normalized_features = scaler.fit_transform(selected_features)

# Prepare data for model training (e.g., applying sliding window for sequences)
def sliding_window(data, window_size):
    return np.array([data[i:i+window_size] for i in range(len(data)-window_size)])

window_size = 50  # Example window size
X = sliding_window(normalized_features, window_size)
y = sliding_window(labels.values, window_size)

# Example shape of the prepared data
print("Input shape:", X.shape)
print("Label shape:", y.shape)
```

```
Input shape: (35105, 50, 9)
Label shape: (35105, 50, 3)
```

Normalization

```
from sklearn.preprocessing import StandardScaler

# Normalize selected features
scaler = StandardScaler()
normalized_features = scaler.fit_transform(selected_features)
```

Sliding Window

```
import numpy as np

def sliding_window(data, window_size):
    return np.array([data[i:i + window_size] for i in range(len(data) - window_size)])

window_size = 50  # Define your window size (this value can be tuned)
X = sliding_window(normalized_features, window_size)
y = sliding_window(labels.values, window_size)
```

```
def sliding_window(data, window_size):
    return np.array([data[i:i+window_size] for i in range(len(data)-window_size)])

window_size = 50
X = sliding_window(normalized_features, window_size)
y = sliding_window(labels.values, window_size)
```

Spatial Encoder

```
import torch
import torch.nn as nn

class SpatialEncoder(nn.Module):
    def __init__(self):
        super(SpatialEncoder, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=9, out_channels=64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(64)
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(128)

    def forward(self, x):
        x = torch.relu(self.bn1(self.conv1(x)))
        x = torch.relu(self.bn2(self.conv2(x)))
        return x
```

Integration with transformer block

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model=256, nhead=8):  # Adjusted d_model to 256
        super(TransformerBlock, self).__init__()
        self.transformer = nn.TransformerEncoderLayer(d_model=d_model, nhead=nhead)

    def forward(self, x):
        x = self.transformer(x)
        return x
```

Temporal Encoder

```
class TemporalEncoder(nn.Module):
    def __init__(self, input_size=128, hidden_size=128, num_layers=2):
        super(TemporalEncoder, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True, bidirectional=True)

    def forward(self, x):
        x, _ = self.lstm(x)
        return x
```

Integration and Final output

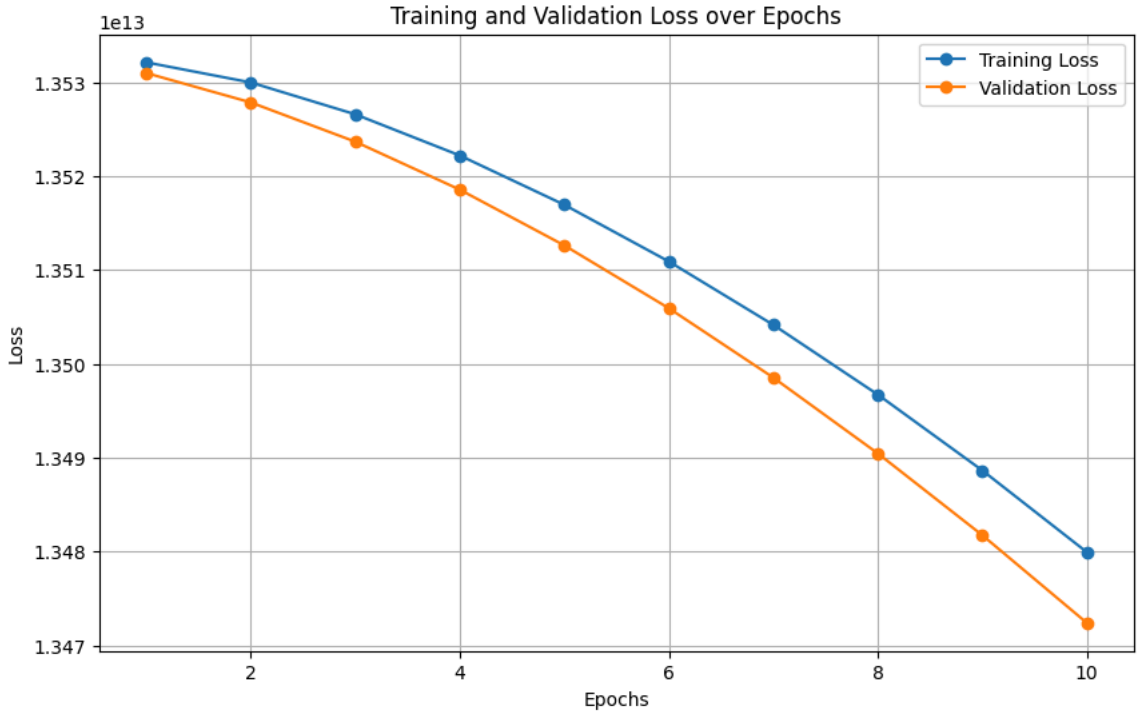
```
class FusionModel(nn.Module):
    def __init__(self):
        super(FusionModel, self).__init__()
        self.spatial_encoder = SpatialEncoder()
        self.temporal_encoder = TemporalEncoder()

        # Reduce dimensionality to 128 if needed
        self.reduce_dim = nn.Linear(256, 128)  # Add this layer if reducing from 256 to 128

        self.transformer = TransformerBlock(d_model=128)  # Ensure the Transformer expects 128
        self.fc = nn.Linear(128, 3)  # Final output layer

    def forward(self, x):
        x = x.transpose(1, 2)  # Shape [batch_size, 9, 50]
        x = self.spatial_encoder(x)
        x = x.transpose(1, 2)  # Shape [batch_size, 50, 128] or [batch_size, 50, 256] depending on SpatialEncoder

        x = self.temporal_encoder(x)

        # If needed, reduce dimension
        x = self.reduce_dim(x)  # Shape now [batch_size, 50, 128]

        x = self.transformer(x)
        x = self.fc(x)
        return x
```

Model Training and Evaluation

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Assuming X and y are your full datasets from the sliding window process
# X -> Input data with shape (35105, 50, 9)
# y -> Labels with shape (35105, 50, 3)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.float32)

# Initialize model, loss function, and optimizer
model = FusionModel()  # Assuming FusionModel is already defined
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Number of epochs
epochs = 10

# Store loss values
train_losses = []
val_losses = []

# Checkpointing and memory management
checkpoint_path = 'model_checkpoint.pth'

# Training loop
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    # Mini-batch processing to avoid memory issues
    batch_size = 64  # Adjust based on available memory
    num_batches = len(X_train) // batch_size

    epoch_train_loss = 0.0

    for i in range(num_batches):
        start_idx = i * batch_size
        end_idx = start_idx + batch_size

        # Get batch data
        X_batch = X_train[start_idx:end_idx]
        y_batch = y_train[start_idx:end_idx]

        # Forward pass
        outputs = model(X_batch)

        # Compute loss
        loss = criterion(outputs, y_batch)
        epoch_train_loss += loss.item()

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

    # Average training loss for the epoch
    epoch_train_loss /= num_batches
    train_losses.append(epoch_train_loss)

    # Validation step
    model.eval()
    with torch.no_grad():
        val_outputs = model(X_val)
        val_loss = criterion(val_outputs, y_val)
        val_losses.append(val_loss.item())

    print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {epoch_train_loss}, Validation Loss: {val_loss.item()}')

    # Save checkpoint after each epoch
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'train_loss': epoch_train_loss,
```

```
            'val_loss': val_loss.item(),
        }, checkpoint_path)

        # Free up memory
        del X_batch, y_batch, outputs, loss
        torch.cuda.empty_cache()  # If using GPU

    # Training is complete
    print("Training complete.")

    # Plot the training and validation losses
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, epochs+1), train_losses, label='Training Loss', marker='o')
    plt.plot(range(1, epochs+1), val_losses, label='Validation Loss', marker='o')
    plt.title('Training and Validation Loss over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()
```

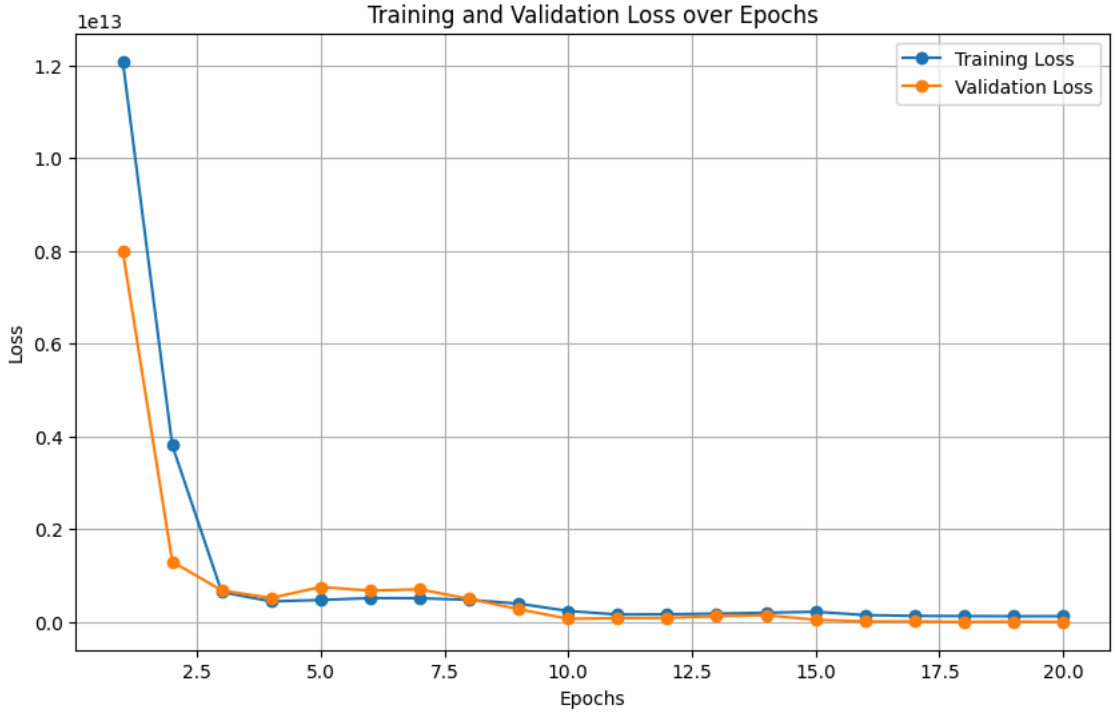
```
Epoch [1/10], Training Loss: 13532134279663.635, Validation Loss: 13530993524736.0
Epoch [2/10], Training Loss: 13529989551520.146, Validation Loss: 13527861428224.0
Epoch [3/10], Training Loss: 13526594906420.604, Validation Loss: 13523676561408.0
Epoch [4/10], Training Loss: 13522207997203.873, Validation Loss: 13518565801984.0
Epoch [5/10], Training Loss: 13516953647506.12, Validation Loss: 13512624570368.0
Epoch [6/10], Training Loss: 13510913351792.219, Validation Loss: 13505931509760.0
Epoch [7/10], Training Loss: 13504155952188.785, Validation Loss: 13498535903232.0
Epoch [8/10], Training Loss: 13496725409885.516, Validation Loss: 13490459770880.0
Epoch [9/10], Training Loss: 13488643926923.105, Validation Loss: 13481741910016.0
Epoch [10/10], Training Loss: 13479939012860.492, Validation Loss: 13472399097856.0
Training complete.
```



```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Assuming X and y are your full datasets from the sliding window process
# X -> Input data with shape (35105, 50, 9)
# y -> Labels with shape (35105, 50, 3)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.float32)

# Define a more complex model architecture
class ImprovedFusionModel(nn.Module):
    def __init__(self):
        super(ImprovedFusionModel, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=9, out_channels=64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(64)
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(128)
        self.fc1 = nn.Linear(128 * 50, 256)  # Flattened feature maps
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, 3 * 50)  # Assuming output size is (50, 3)

    def forward(self, x):
        x = x.transpose(1, 2)  # Transpose to (batch_size, channels, sequence_length)
        x = torch.relu(self.bn1(self.conv1(x)))
        x = torch.relu(self.bn2(self.conv2(x)))
        x = x.view(x.size(0), -1)  # Flatten for fully connected layers
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x.view(-1, 50, 3)  # Reshape to match the output shape

# Initialize model, loss function, and optimizer
model = ImprovedFusionModel()
criterion = nn.MSELoss()
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=3, factor=0.1)

# Number of epochs
epochs = 20

# Store loss values
train_losses = []
val_losses = []

# Checkpointing and memory management
checkpoint_path = 'model_checkpoint.pth'

# Training loop
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    # Mini-batch processing to avoid memory issues
    batch_size = 64  # Adjust based on available memory
    num_batches = len(X_train) // batch_size

    epoch_train_loss = 0.0

    for i in range(num_batches):
        start_idx = i * batch_size
        end_idx = start_idx + batch_size

        # Get batch data
        X_batch = X_train[start_idx:end_idx]
        y_batch = y_train[start_idx:end_idx]

        # Forward pass
        outputs = model(X_batch)

        # Compute loss
```

```
            loss = criterion(outputs, y_batch)
            epoch_train_loss += loss.item()

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

        # Average training loss for the epoch
        epoch_train_loss /= num_batches
        train_losses.append(epoch_train_loss)

        # Validation step
        model.eval()
        with torch.no_grad():
            val_outputs = model(X_val)
            val_loss = criterion(val_outputs, y_val)
            val_losses.append(val_loss.item())

        print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {epoch_train_loss}, Validation Loss: {val_loss.item()}')

        # Learning rate scheduling
        scheduler.step(val_loss)

        # Save checkpoint after each epoch
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'train_loss': epoch_train_loss,
            'val_loss': val_loss.item(),
        }, checkpoint_path)

        # Free up memory
        del X_batch, y_batch, outputs, loss
        torch.cuda.empty_cache()  # If using GPU

    # Training is complete
    print("Training complete.")

    # Plot the training and validation losses
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, epochs+1), train_losses, label='Training Loss', marker='o')
    plt.plot(range(1, epochs+1), val_losses, label='Validation Loss', marker='o')
    plt.title('Training and Validation Loss over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
Epoch [1/20], Training Loss: 12080560140493.734, Validation Loss: 7989561393152.0
Epoch [2/20], Training Loss: 3806379486914.046, Validation Loss: 1298843697152.0
Epoch [3/20], Training Loss: 642535722241.169, Validation Loss: 678483263488.0
Epoch [4/20], Training Loss: 442570363572.01825, Validation Loss: 515854237696.0
Epoch [5/20], Training Loss: 473590321413.8447, Validation Loss: 753975230464.0
Epoch [6/20], Training Loss: 513252247168.5845, Validation Loss: 673696776192.0
Epoch [7/20], Training Loss: 511206494320.2192, Validation Loss: 703490359296.0
Epoch [8/20], Training Loss: 476317745544.76715, Validation Loss: 497112940544.0
Epoch [9/20], Training Loss: 393918024848.94977, Validation Loss: 274772033536.0
Epoch [10/20], Training Loss: 236297266274.19177, Validation Loss: 68221657088.0
Epoch [11/20], Training Loss: 160769512172.12787, Validation Loss: 80880099328.0
Epoch [12/20], Training Loss: 164998581692.20093, Validation Loss: 87681130496.0
Epoch [13/20], Training Loss: 179202850236.20093, Validation Loss: 123447541760.0
Epoch [14/20], Training Loss: 198118386683.3242, Validation Loss: 145150197760.0
Epoch [15/20], Training Loss: 219401790786.63013, Validation Loss: 46029189120.0
Epoch [16/20], Training Loss: 147904297175.08676, Validation Loss: 10189704192.0
Epoch [17/20], Training Loss: 129966864935.7443, Validation Loss: 8493361664.0
Epoch [18/20], Training Loss: 126437665188.82191, Validation Loss: 1279842048.0
Epoch [19/20], Training Loss: 122718496665.13242, Validation Loss: 6059115008.0
Epoch [20/20], Training Loss: 123592771350.21005, Validation Loss: 1387937920.0
Training complete.
```




```
# Model should be in evaluation mode
model.eval()

# Generate predictions for the validation set
with torch.no_grad():
    y_pred = model(X_val)
```

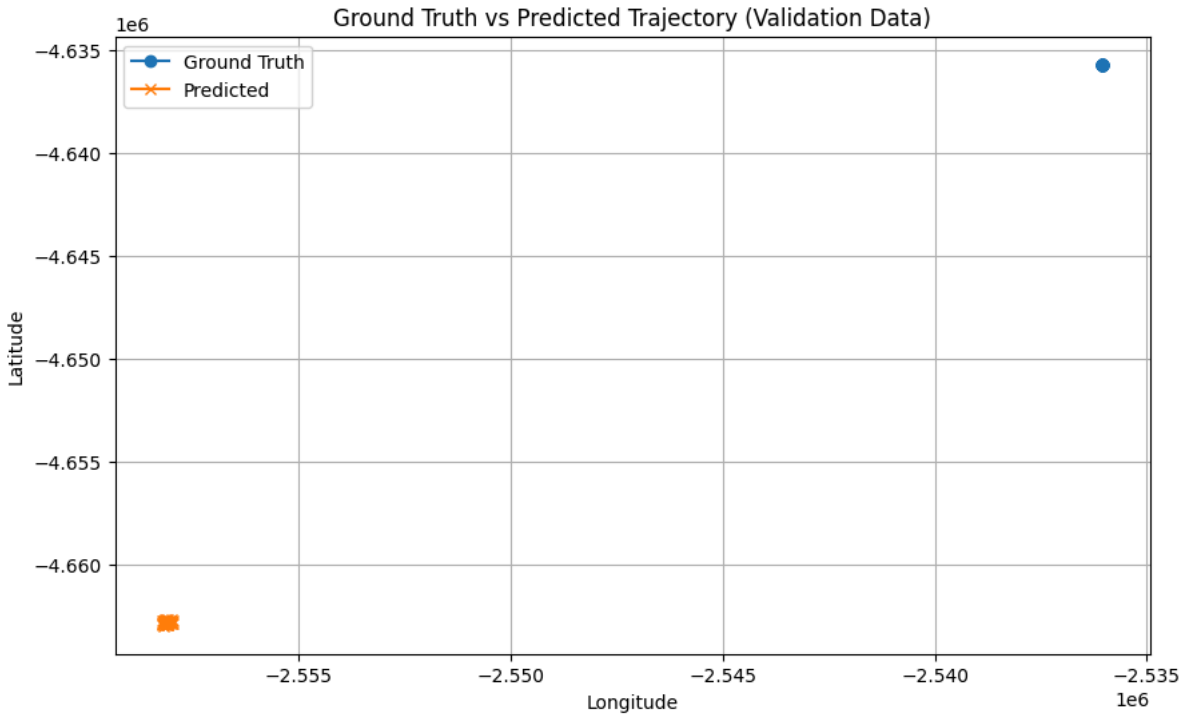
```
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np

# Generate predictions (if not already done)
model.eval()
with torch.no_grad():
    y_pred = model(X_val)

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3)))

# Calculate MAE
mae = mean_absolute_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3))

# Calculate Position Error (PE)
pe = np.linalg.norm(y_val.reshape(-1, 3) - y_pred.reshape(-1, 3), axis=1).mean()

print(f"RMSE: {rmse}")
print(f"MAE: {mae}")
print(f"PE: {pe}")
```

```
RMSE: 37253.5078125
MAE: 30468.158203125
```

```
PE: 54408.4375
```

```
import torch
import torch.nn as nn
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error
import matplotlib.pyplot as plt

# Assuming X_val and y_val are already defined and contain validation data
# Assuming the model has already been trained and is defined as 'model'

# Set model to evaluation mode
model.eval()

# Generate predictions for the validation set
with torch.no_grad():
    y_pred = model(X_val)

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3)))

# Calculate MAE
mae = mean_absolute_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3))

# Calculate Position Error (PE)
pe = np.linalg.norm(y_val.reshape(-1, 3) - y_pred.reshape(-1, 3), axis=1).mean()

# Print the metrics
print(f"RMSE: {rmse}")
print(f"MAE: {mae}")
print(f"PE: {pe}")

# Example for plotting the first trajectory in your validation set
plt.figure(figsize=(10, 6))
plt.plot(y_val[0, :, 0], y_val[0, :, 1], label='Ground Truth', marker='o')
plt.plot(y_pred[0, :, 0], y_pred[0, :, 1], label='Predicted', marker='x')
plt.title('Ground Truth vs Predicted Trajectory (Validation Data)')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()
```

```
RMSE: 37253.5078125
MAE: 30468.158203125
PE: 54408.4375
```



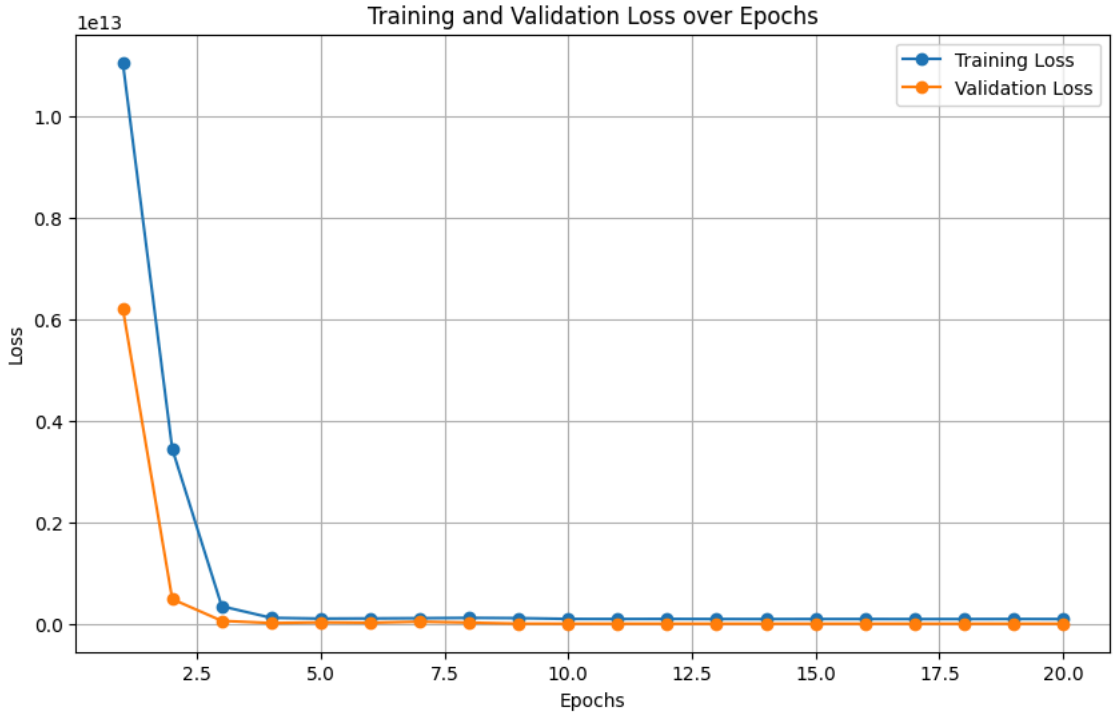

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np

# Assuming X and y are your full datasets from the sliding window process
# X -> Input data with shape (35105, 50, 9)
# y -> Labels with shape (35105, 50, 3)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train.reshape(-1, X_train.shape[-1])).reshape(X_train.shape)
X_val = scaler.transform(X_val.reshape(-1, X_val.shape[-1])).reshape(X_val.shape)

# Convert data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.float32)

# Define a more complex model architecture with LSTM
class ImprovedFusionModel(nn.Module):
    def __init__(self):
        super(ImprovedFusionModel, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=9, out_channels=64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(64)
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(128)
        self.lstm = nn.LSTM(input_size=128, hidden_size=256, batch_first=True)
        self.fc1 = nn.Linear(256 * 50, 256)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, 3 * 50)

    def forward(self, x):
        x = x.transpose(1, 2)  # Transpose to (batch_size, channels, sequence_length)
        x = torch.relu(self.bn1(self.conv1(x)))
        x = torch.relu(self.bn2(self.conv2(x)))
        x, _ = self.lstm(x.transpose(1, 2))
        x = x.reshape(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x.view(-1, 50, 3)

# Initialize model, loss function, and optimizer
model = ImprovedFusionModel()
criterion = nn.MSELoss()
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=3, factor=0.1)

# Number of epochs
epochs = 20

# Store loss values
train_losses = []
val_losses = []

# Checkpointing and memory management
checkpoint_path = 'model_checkpoint.pth'

# Training loop
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    # Mini-batch processing to avoid memory issues
    batch_size = 64  # Adjust based on available memory
    num_batches = len(X_train) // batch_size

    epoch_train_loss = 0.0

    for i in range(num_batches):
        start_idx = i * batch_size
        end_idx = start_idx + batch_size

        # Get batch data
        X_batch = X_train[start_idx:end_idx]
        y_batch = y_train[start_idx:end_idx]
```

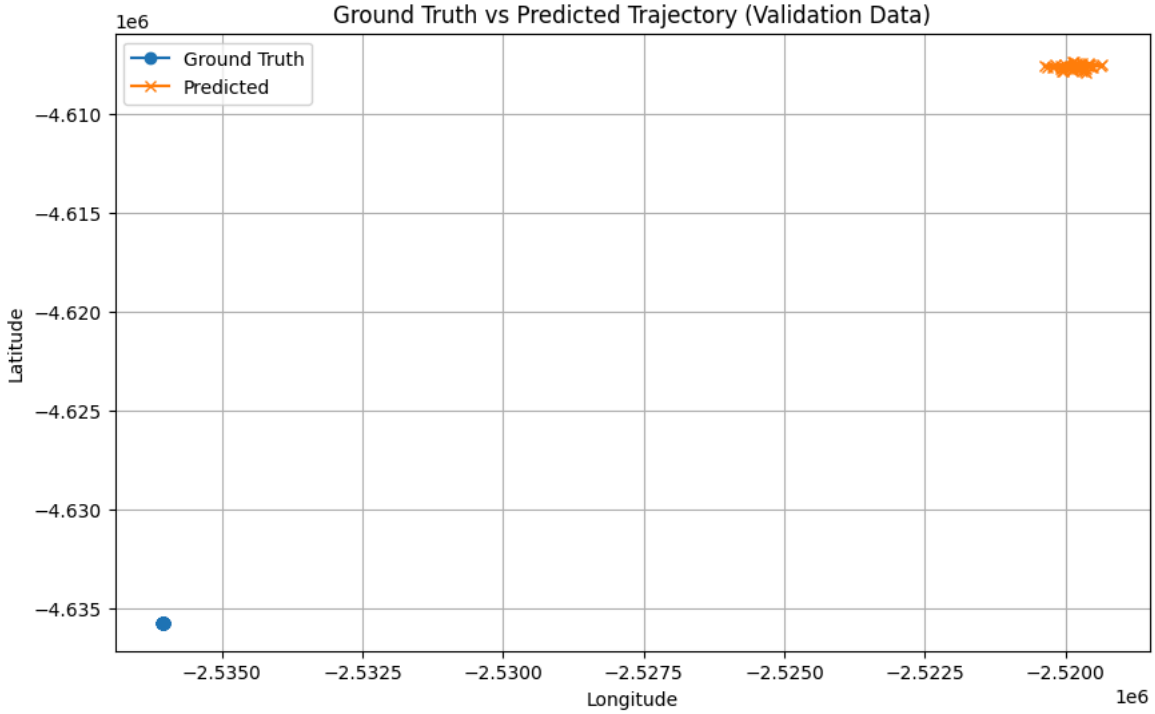
```
            # Forward pass
            outputs = model(X_batch)

            # Compute loss
            loss = criterion(outputs, y_batch)
            epoch_train_loss += loss.item()

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

        # Average training loss for the epoch
        epoch_train_loss /= num_batches
        train_losses.append(epoch_train_loss)

        # Validation step
        model.eval()
        with torch.no_grad():
            val_outputs = model(X_val)
            val_loss = criterion(val_outputs, y_val)
            val_losses.append(val_loss.item())

        print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {epoch_train_loss}, Validation Loss: {val_loss.item()}')

        # Learning rate scheduling
        scheduler.step(val_loss)

        # Save checkpoint after each epoch
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'train_loss': epoch_train_loss,
            'val_loss': val_loss.item(),
        }, checkpoint_path)

        # Free up memory
        del X_batch, y_batch, outputs, loss
        torch.cuda.empty_cache()  # If using GPU

    # Training is complete
    print("Training complete.")

    # Plot the training and validation losses
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, epochs+1), train_losses, label='Training Loss', marker='o')
    plt.plot(range(1, epochs+1), val_losses, label='Validation Loss', marker='o')
    plt.title('Training and Validation Loss over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Evaluate the model on the validation set
    model.eval()
    with torch.no_grad():
        y_pred = model(X_val)

    # Calculate RMSE
    rmse = np.sqrt(mean_squared_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3)))

    # Calculate MAE
    mae = mean_absolute_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3))

    # Calculate Position Error (PE)
    pe = np.linalg.norm(y_val.reshape(-1, 3) - y_pred.reshape(-1, 3), axis=1).mean()

    # Print the metrics
    print(f"RMSE: {rmse}")
    print(f"MAE: {mae}")
    print(f"PE: {pe}")

    # Example for plotting the first trajectory in your validation set
    plt.figure(figsize=(10, 6))
    plt.plot(y_val[0, :, 0], y_val[0, :, 1], label='Ground Truth', marker='o')
    plt.plot(y_pred[0, :, 0], y_pred[0, :, 1], label='Predicted', marker='x')
    plt.title('Ground Truth vs Predicted Trajectory (Validation Data)')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
plt.show()
```

```
Epoch [1/20], Training Loss: 11047316619862.502, Validation Loss: 6205891674112.0
Epoch [2/20], Training Loss: 3440963658214.283, Validation Loss: 485291720704.0
Epoch [3/20], Training Loss: 344731365188.968, Validation Loss: 57478664192.0
Epoch [4/20], Training Loss: 116868824835.50685, Validation Loss: 14342035456.0
Epoch [5/20], Training Loss: 102029327542.35617, Validation Loss: 27155085312.0
Epoch [6/20], Training Loss: 104784796788.89497, Validation Loss: 19926450176.0
Epoch [7/20], Training Loss: 109747329912.40182, Validation Loss: 40948944896.0
Epoch [8/20], Training Loss: 115950356629.62556, Validation Loss: 22923132928.0
Epoch [9/20], Training Loss: 111143980513.6073, Validation Loss: 2009389312.0
Epoch [10/20], Training Loss: 95471642829.73515, Validation Loss: 947683328.0
Epoch [11/20], Training Loss: 94264679877.5525, Validation Loss: 769556736.0
Epoch [12/20], Training Loss: 95695080031.85388, Validation Loss: 673350336.0
Epoch [13/20], Training Loss: 93716788537.27853, Validation Loss: 826363648.0
Epoch [14/20], Training Loss: 92595176555.54338, Validation Loss: 602680128.0
Epoch [15/20], Training Loss: 93578350021.5525, Validation Loss: 534531456.0
Epoch [16/20], Training Loss: 93807531807.56165, Validation Loss: 509185792.0
Epoch [17/20], Training Loss: 92471949611.25114, Validation Loss: 793395712.0
Epoch [18/20], Training Loss: 92897961507.0685, Validation Loss: 511191008.0
Epoch [19/20], Training Loss: 94130625091.79909, Validation Loss: 880722176.0
Epoch [20/20], Training Loss: 93240454868.74886, Validation Loss: 482468832.0
Training complete.
```




```
RMSE: 21964.3203125
MAE: 21102.904296875
PE: 38044.3203125
```

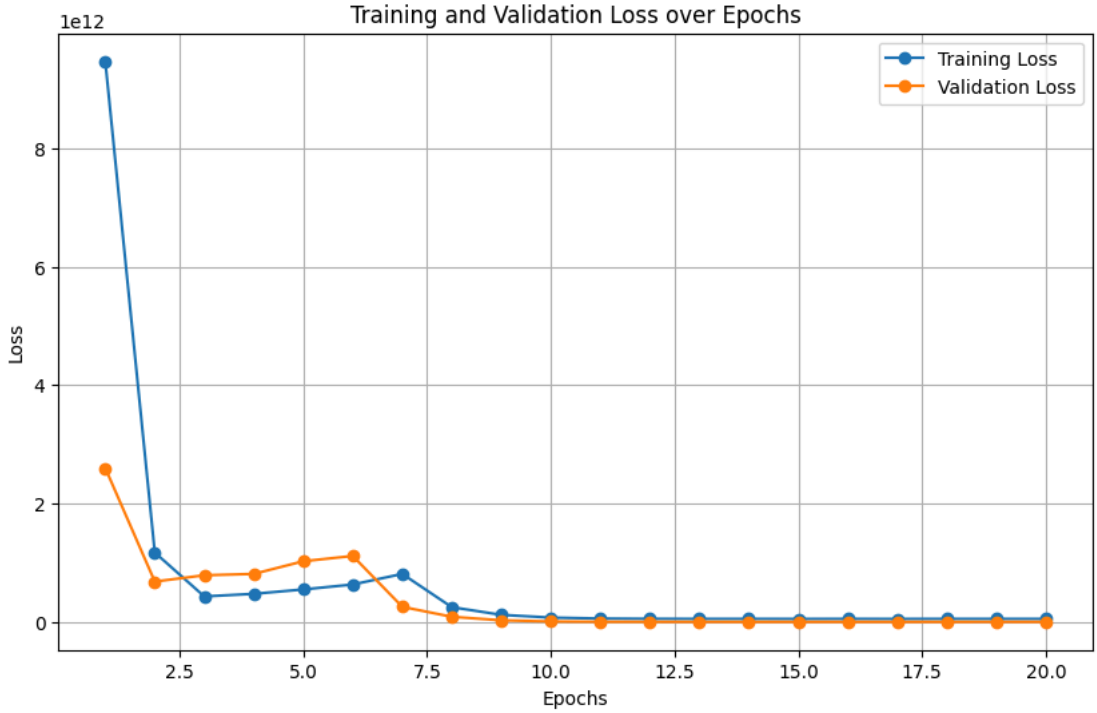


```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np

# Assuming X and y are your full datasets from the sliding window process
# X -> Input data with shape (35105, 50, 9)
# y -> Labels with shape (35105, 50, 3)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train.reshape(-1, X_train.shape[-1])).reshape(X_train.shape)
X_val = scaler.transform(X_val.reshape(-1, X_val.shape[-1])).reshape(X_val.shape)

# Convert data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.float32)

# Define a more complex model architecture with LSTM
class ImprovedFusionModel(nn.Module):
    def __init__(self):
        super(ImprovedFusionModel, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=9, out_channels=64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(64)
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(128)
        self.lstm1 = nn.LSTM(input_size=128, hidden_size=256, batch_first=True)
        self.lstm2 = nn.LSTM(input_size=256, hidden_size=256, batch_first=True)
        self.fc1 = nn.Linear(256 * 50, 512)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 3 * 50)

    def forward(self, x):
        x = x.transpose(1, 2)  # Transpose to (batch_size, channels, sequence_length)
        x = torch.relu(self.bn1(self.conv1(x)))
        x = torch.relu(self.bn2(self.conv2(x)))
        x, _ = self.lstm1(x.transpose(1, 2))
        x, _ = self.lstm2(x)
        x = x.reshape(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x.view(-1, 50, 3)

# Initialize model, loss function, and optimizer
model = ImprovedFusionModel()
criterion = nn.MSELoss()
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=3, factor=0.1)

# Number of epochs
epochs = 20

# Store loss values
train_losses = []
val_losses = []

# Checkpointing and memory management
checkpoint_path = 'model_checkpoint.pth'

# Training loop
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    # Mini-batch processing to avoid memory issues
    batch_size = 64  # Adjust based on available memory
    num_batches = len(X_train) // batch_size

    epoch_train_loss = 0.0
```

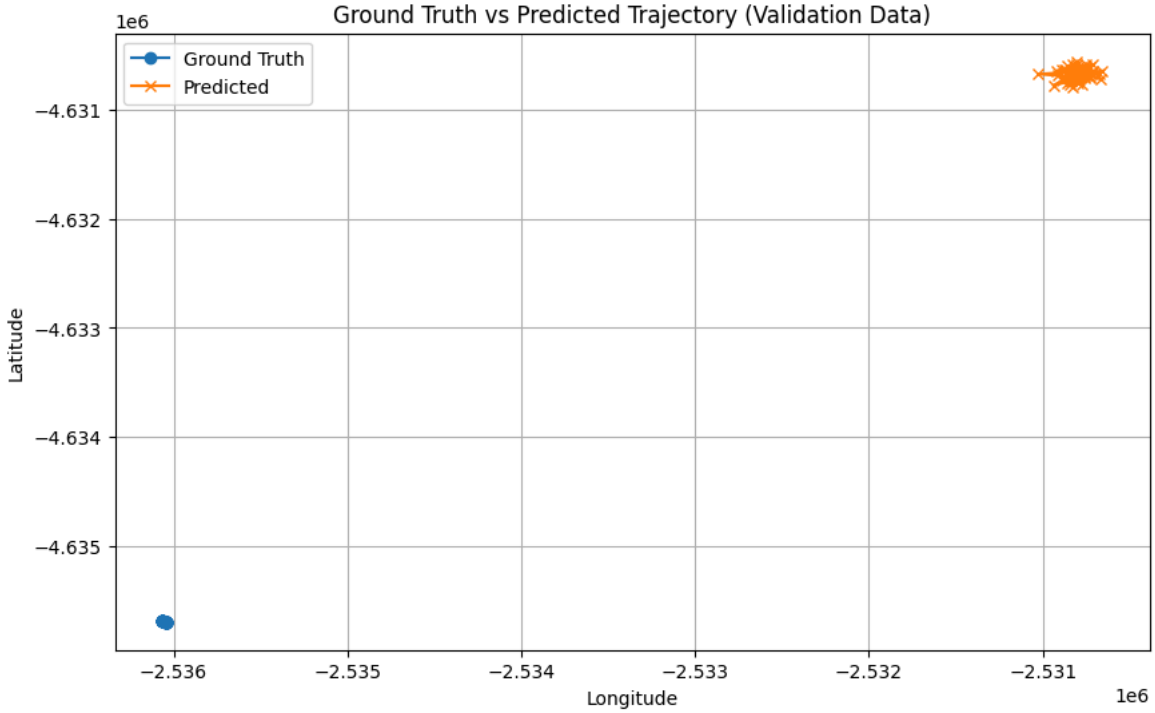
```
        for i in range(num_batches):
            start_idx = i * batch_size
            end_idx = start_idx + batch_size

            # Get batch data
            X_batch = X_train[start_idx:end_idx]
            y_batch = y_train[start_idx:end_idx]

            # Forward pass
            outputs = model(X_batch)

            # Compute loss
            loss = criterion(outputs, y_batch)
            epoch_train_loss += loss.item()

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

        # Average training loss for the epoch
        epoch_train_loss /= num_batches
        train_losses.append(epoch_train_loss)

        # Validation step
        model.eval()
        with torch.no_grad():
            val_outputs = model(X_val)
            val_loss = criterion(val_outputs, y_val)
            val_losses.append(val_loss.item())

        print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {epoch_train_loss}, Validation Loss: {val_loss.item()}')

        # Learning rate scheduling
        scheduler.step(val_loss)

        # Save checkpoint after each epoch
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'train_loss': epoch_train_loss,
            'val_loss': val_loss.item(),
        }, checkpoint_path)

        # Free up memory
        del X_batch, y_batch, outputs, loss
        torch.cuda.empty_cache()  # If using GPU

    # Training is complete
    print("Training complete.")

    # Plot the training and validation losses
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, epochs+1), train_losses, label='Training Loss', marker='o')
    plt.plot(range(1, epochs+1), val_losses, label='Validation Loss', marker='o')
    plt.title('Training and Validation Loss over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Evaluate the model on the validation set
    model.eval()
    with torch.no_grad():
        y_pred = model(X_val)

    # Calculate RMSE
    rmse = np.sqrt(mean_squared_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3)))

    # Calculate MAE
    mae = mean_absolute_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3))

    # Calculate Position Error (PE)
    pe = np.linalg.norm(y_val.reshape(-1, 3) - y_pred.reshape(-1, 3), axis=1).mean()

    # Print the metrics
    print(f"RMSE: {rmse}")
    print(f"MAE: {mae}")
    print(f"PE: {pe}")

    # Example for plotting the first trajectory in your validation set
```

```
    plt.figure(figsize=(10, 6))
    plt.plot(y_val[0, :, 0], y_val[0, :, 1], label='Ground Truth', marker='o')
    plt.plot(y_pred[0, :, 0], y_pred[0, :, 1], label='Predicted', marker='x')
    plt.title('Ground Truth vs Predicted Trajectory (Validation Data)')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
Epoch [1/20], Training Loss: 9472463054632.914, Validation Loss: 2598332792832.0
Epoch [2/20], Training Loss: 1171462847539.4338, Validation Loss: 680994471936.0
Epoch [3/20], Training Loss: 429965912288.43835, Validation Loss: 787641794560.0
Epoch [4/20], Training Loss: 473886936648.4749, Validation Loss: 810321641472.0
Epoch [5/20], Training Loss: 548055489185.31506, Validation Loss: 1024811008000.0
Epoch [6/20], Training Loss: 631778423700.4567, Validation Loss: 1114005569536.0
Epoch [7/20], Training Loss: 810431230382.1735, Validation Loss: 254228627456.0
Epoch [8/20], Training Loss: 248578205013.33334, Validation Loss: 85878734848.0
Epoch [9/20], Training Loss: 118602908143.6347, Validation Loss: 26637717504.0
Epoch [10/20], Training Loss: 72526117799.15982, Validation Loss: 6970107904.0
Epoch [11/20], Training Loss: 56161625443.36073, Validation Loss: 1878298624.0
Epoch [12/20], Training Loss: 51949779748.23744, Validation Loss: 752240192.0
Epoch [13/20], Training Loss: 50704225934.61187, Validation Loss: 383862880.0
Epoch [14/20], Training Loss: 50921777095.89041, Validation Loss: 291100352.0
Epoch [15/20], Training Loss: 50050681753.13242, Validation Loss: 230404896.0
Epoch [16/20], Training Loss: 50666482594.48402, Validation Loss: 277361440.0
Epoch [17/20], Training Loss: 49993106301.07763, Validation Loss: 275707808.0
Epoch [18/20], Training Loss: 50841579159.96347, Validation Loss: 135943808.0
Epoch [19/20], Training Loss: 50772654860.858444, Validation Loss: 502214272.0
Epoch [20/20], Training Loss: 50325258614.06393, Validation Loss: 23816222.0
Training complete.
```




```
RMSE: 4880.1845703125
MAE: 4557.84375
PE: 8445.033203125
```

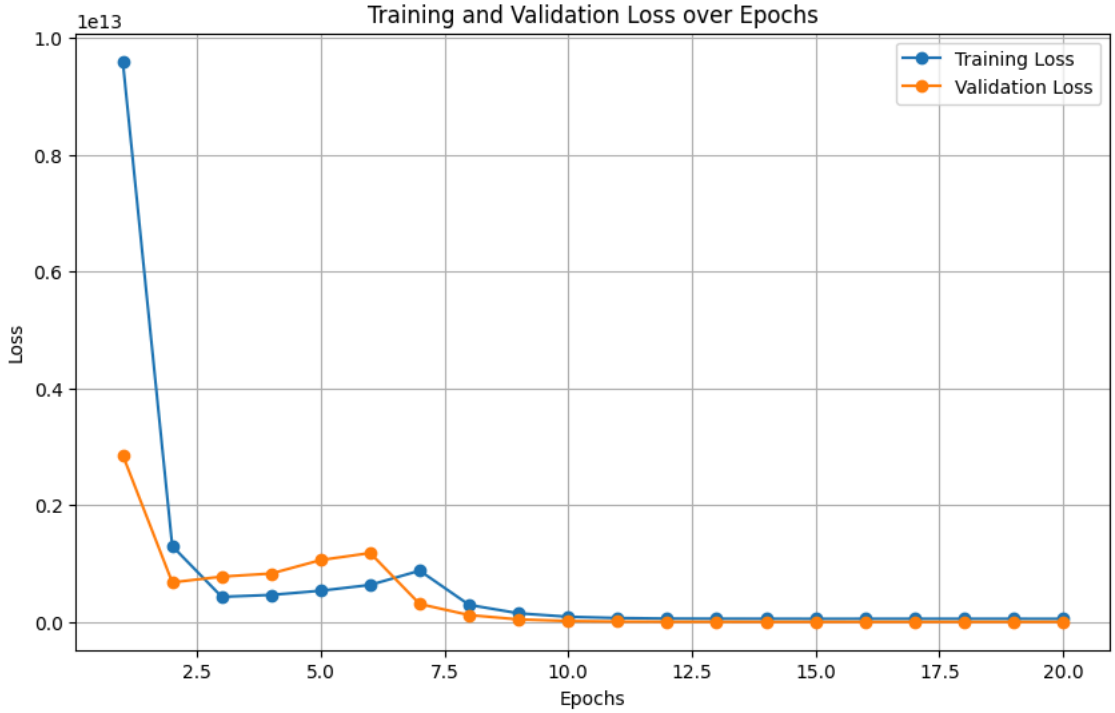


```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np

# Assuming X and y are your full datasets from the sliding window process
# X -> Input data with shape (35105, 50, 9)
# y -> Labels with shape (35105, 50, 3)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train.reshape(-1, X_train.shape[-1])).reshape(X_train.shape)
X_val = scaler.transform(X_val.reshape(-1, X_val.shape[-1])).reshape(X_val.shape)

# Convert data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.float32)

class ResidualLSTM(nn.Module):
    def __init__(self):
        super(ResidualLSTM, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=9, out_channels=64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(64)
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(128)
        self.lstm1 = nn.LSTM(input_size=128, hidden_size=256, batch_first=True)
        self.lstm2 = nn.LSTM(input_size=256, hidden_size=256, batch_first=True)
        self.fc1 = nn.Linear(256 * 50, 512)
        self.fc2 = nn.Linear(512, 256)
        self.dropout = nn.Dropout(0.5)
        self.fc3 = nn.Linear(256, 3 * 50)

    def forward(self, x):
        residual = x
        x = x.transpose(1, 2)  # Transpose to (batch_size, channels, sequence_length)
        x = torch.relu(self.bn1(self.conv1(x)))
        x = torch.relu(self.bn2(self.conv2(x)))
        x, _ = self.lstm1(x.transpose(1, 2))
        x, _ = self.lstm2(x)
        x = x.reshape(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        x = x.view(-1, 50, 3)
        return x + residual  # Add the residual connection

# Initialize model, loss function, and optimizer
model = ImprovedFusionModel()
criterion = nn.MSELoss()
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=3, factor=0.1)

# Number of epochs
epochs = 20

# Store loss values
train_losses = []
val_losses = []

# Checkpointing and memory management
checkpoint_path = 'model_checkpoint.pth'

# Training loop
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    # Mini-batch processing to avoid memory issues
    batch_size = 64  # Adjust based on available memory
    num_batches = len(X_train) // batch_size
```

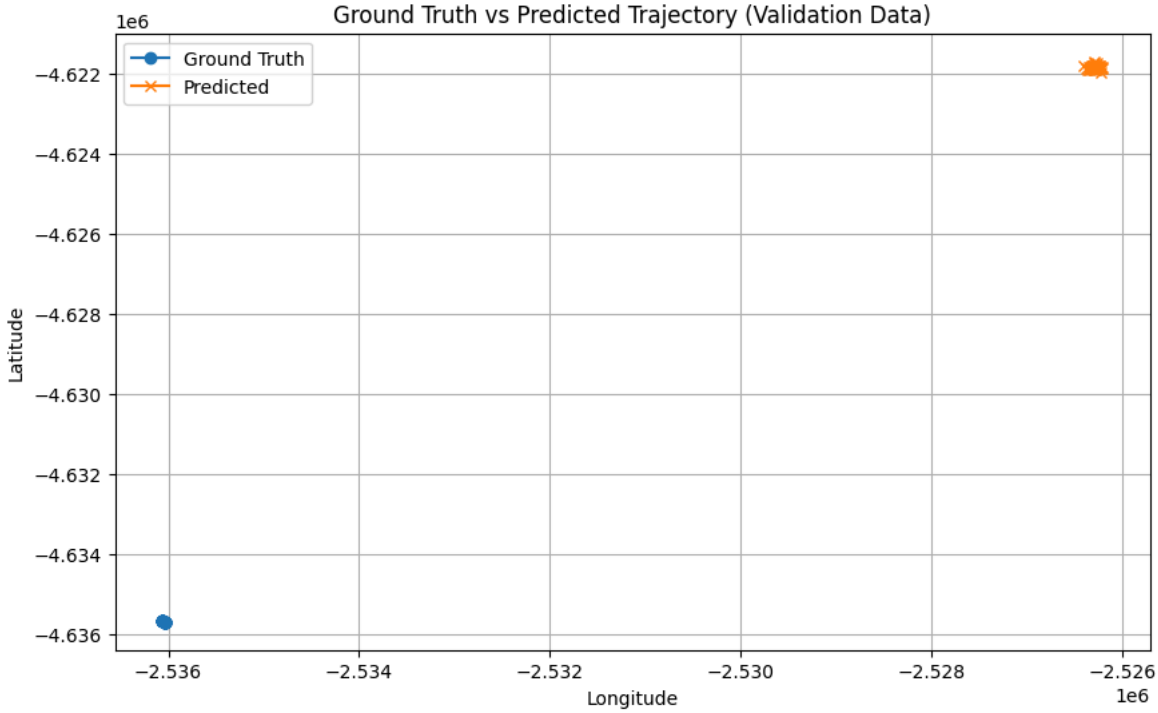
```
        epoch_train_loss = 0.0

        for i in range(num_batches):
            start_idx = i * batch_size
            end_idx = start_idx + batch_size

            # Get batch data
            X_batch = X_train[start_idx:end_idx]
            y_batch = y_train[start_idx:end_idx]

            # Forward pass
            outputs = model(X_batch)

            # Compute loss
            loss = criterion(outputs, y_batch)
            epoch_train_loss += loss.item()

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

        # Average training loss for the epoch
        epoch_train_loss /= num_batches
        train_losses.append(epoch_train_loss)

        # Validation step
        model.eval()
        with torch.no_grad():
            val_outputs = model(X_val)
            val_loss = criterion(val_outputs, y_val)
            val_losses.append(val_loss.item())

        print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {epoch_train_loss}, Validation Loss: {val_loss.item()}')

        # Learning rate scheduling
        scheduler.step(val_loss)

        # Save checkpoint after each epoch
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'train_loss': epoch_train_loss,
            'val_loss': val_loss.item(),
        }, checkpoint_path)

        # Free up memory
        del X_batch, y_batch, outputs, loss
        torch.cuda.empty_cache()  # If using GPU

    # Training is complete
    print("Training complete.")

    # Plot the training and validation losses
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, epochs+1), train_losses, label='Training Loss', marker='o')
    plt.plot(range(1, epochs+1), val_losses, label='Validation Loss', marker='o')
    plt.title('Training and Validation Loss over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Evaluate the model on the validation set
    model.eval()
    with torch.no_grad():
        y_pred = model(X_val)

    # Calculate RMSE
    rmse = np.sqrt(mean_squared_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3)))

    # Calculate MAE
    mae = mean_absolute_error(y_val.reshape(-1, 3), y_pred.reshape(-1, 3))

    # Calculate Position Error (PE)
    pe = np.linalg.norm(y_val.reshape(-1, 3) - y_pred.reshape(-1, 3), axis=1).mean()

    # Print the metrics
    print(f"RMSE: {rmse}")
    print(f"MAE: {mae}")
```

```
print(f"PE: {pe}")

# Example for plotting the first trajectory in your validation set
plt.figure(figsize=(10, 6))
plt.plot(y_val[0, :, 0], y_val[0, :, 1], label='Ground Truth', marker='o')
plt.plot(y_pred[0, :, 0], y_pred[0, :, 1], label='Predicted', marker='x')
plt.title('Ground Truth vs Predicted Trajectory (Validation Data)')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()
```

```
Epoch [1/20], Training Loss: 9594995804805.26, Validation Loss: 2854027788288.0
Epoch [2/20], Training Loss: 1297420470505.79, Validation Loss: 675955671040.0
Epoch [3/20], Training Loss: 428702861358.758, Validation Loss: 774725435392.0
Epoch [4/20], Training Loss: 461612656808.3288, Validation Loss: 827539914752.0
Epoch [5/20], Training Loss: 533510040711.59814, Validation Loss: 1058372452352.0
Epoch [6/20], Training Loss: 632558571669.6256, Validation Loss: 1179080458240.0
Epoch [7/20], Training Loss: 876714248762.4475, Validation Loss: 306025922560.0
Epoch [8/20], Training Loss: 290600173044.3105, Validation Loss: 116341751808.0
Epoch [9/20], Training Loss: 146546781590.79453, Validation Loss: 42184241152.0
Epoch [10/20], Training Loss: 87923339451.03197, Validation Loss: 14565805056.0
Epoch [11/20], Training Loss: 66063751607.525116, Validation Loss: 5295752192.0
Epoch [12/20], Training Loss: 56677674671.34247, Validation Loss: 2317069568.0
Epoch [13/20], Training Loss: 54147895698.11872, Validation Loss: 1029935488.0
Epoch [14/20], Training Loss: 53483391312.65753, Validation Loss: 484929600.0
Epoch [15/20], Training Loss: 52386609325.00457, Validation Loss: 380918176.0
Epoch [16/20], Training Loss: 52813571698.557076, Validation Loss: 246180624.0
Epoch [17/20], Training Loss: 53363118248.328766, Validation Loss: 182394560.0
Epoch [18/20], Training Loss: 52869655687.598175, Validation Loss: 287684224.0
Epoch [19/20], Training Loss: 52885285757.07763, Validation Loss: 171219824.0
Epoch [20/20], Training Loss: 52184060801.753426, Validation Loss: 138125984.0
Training complete.
```

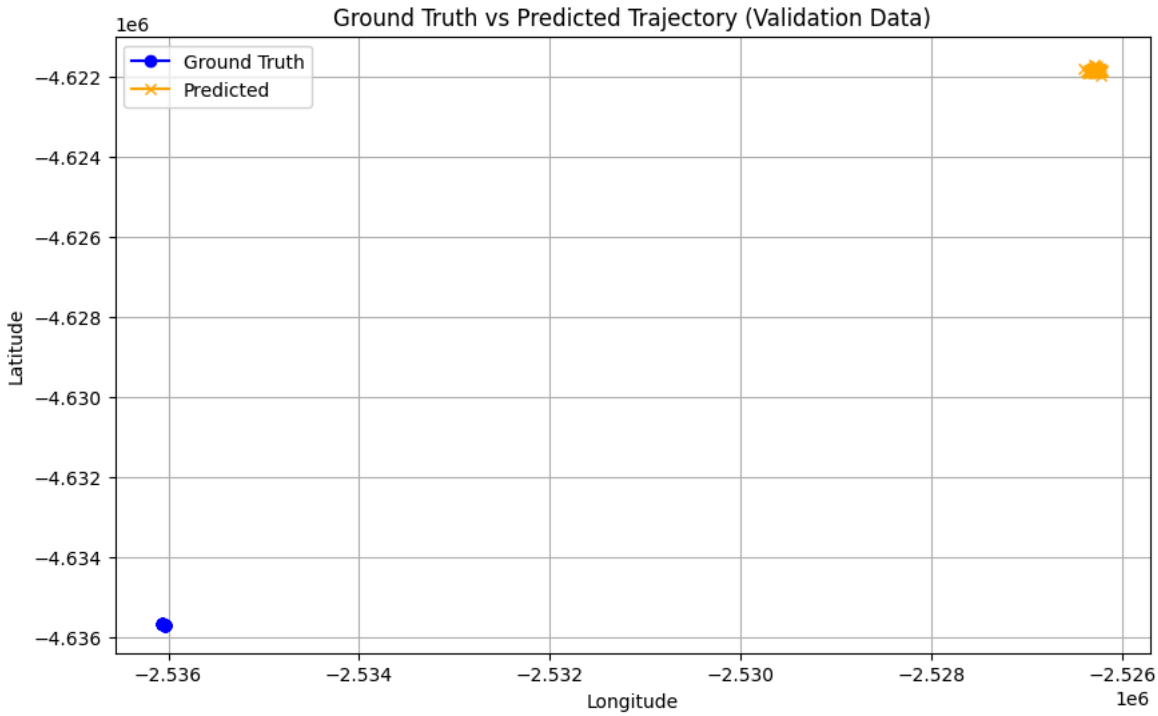


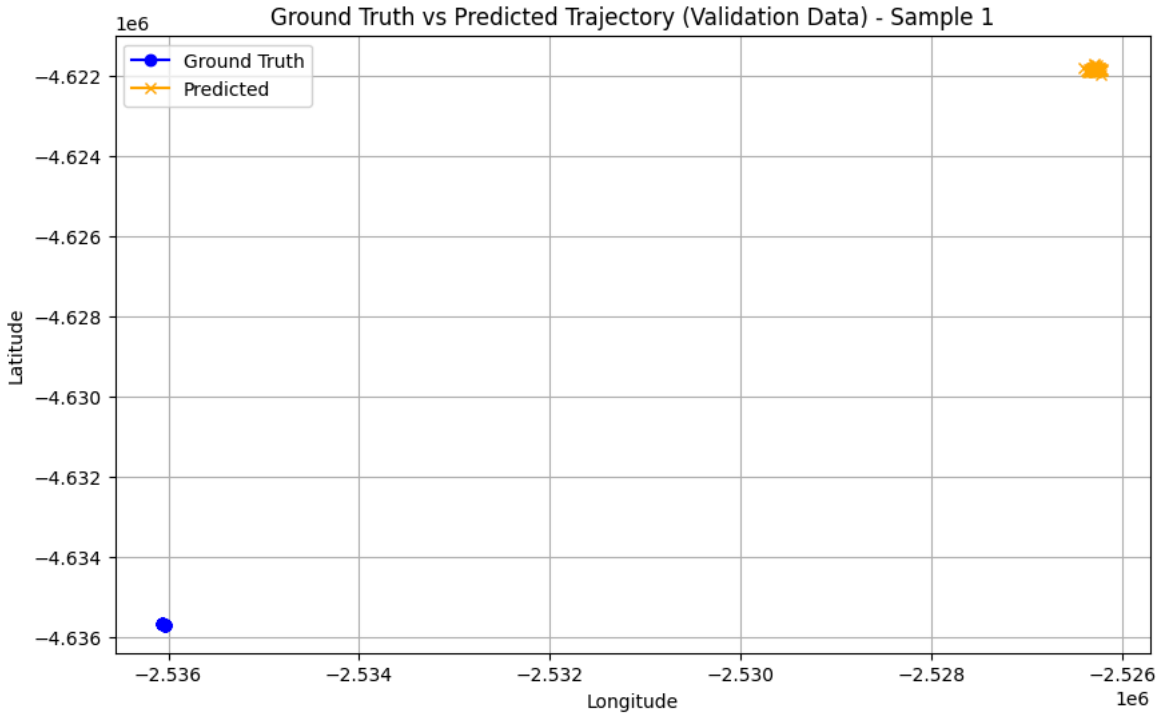

```
RMSE: 11752.4794921875
MAE: 11342.4560546875
PE: 20355.904296875
```

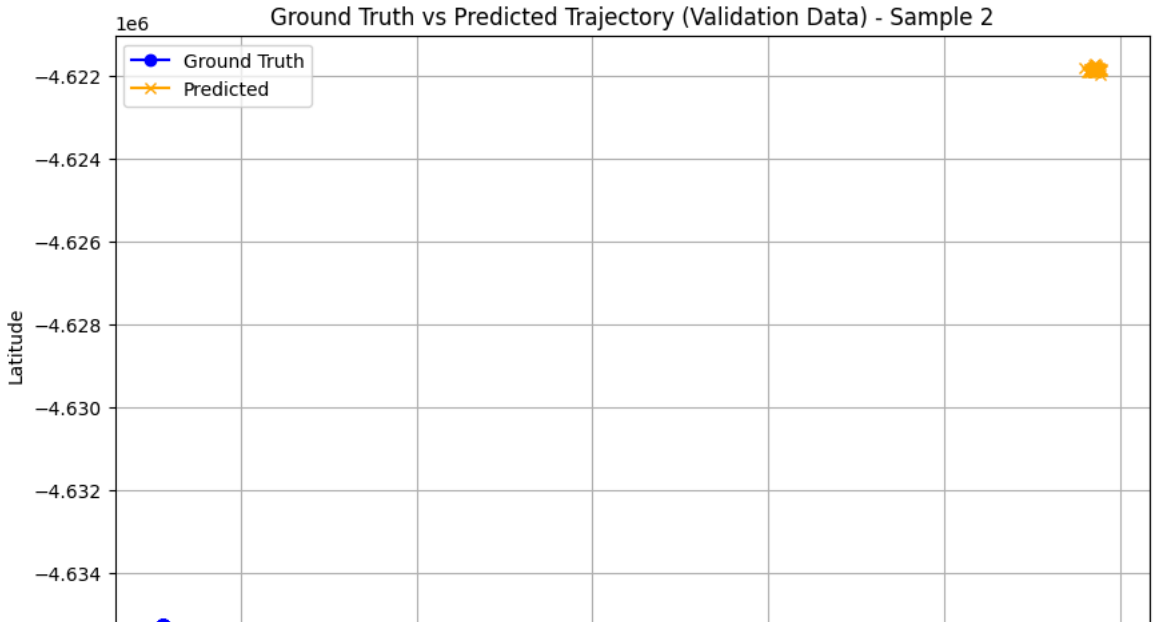


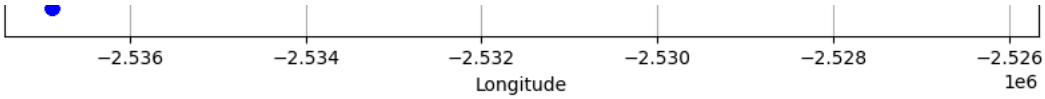
```
import matplotlib.pyplot as plt

# Evaluate the model on the validation set
model.eval()
with torch.no_grad():
    y_pred = model(X_val)

# Assuming y_val and y_pred are already defined and contain the ground truth and predicted data, respectively

# Plot the first trajectory in the validation set
plt.figure(figsize=(10, 6))
plt.plot(y_val[0, :, 0], y_val[0, :, 1], label='Ground Truth', marker='o', linestyle='-', color='blue')
plt.plot(y_pred[0, :, 0], y_pred[0, :, 1], label='Predicted', marker='x', linestyle='-', color='orange')
plt.title('Ground Truth vs Predicted Trajectory (Validation Data)')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()

# Optionally, if you want to compare multiple trajectories, you can loop through a few examples
for i in range(3):  # Plotting the first three trajectories
    plt.figure(figsize=(10, 6))
    plt.plot(y_val[i, :, 0], y_val[i, :, 1], label='Ground Truth', marker='o', linestyle='-', color='blue')
    plt.plot(y_pred[i, :, 0], y_pred[i, :, 1], label='Predicted', marker='x', linestyle='-', color='orange')
    plt.title(f'Ground Truth vs Predicted Trajectory (Validation Data) - Sample {i+1}')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.legend()
    plt.grid(True)
    plt.show()
```

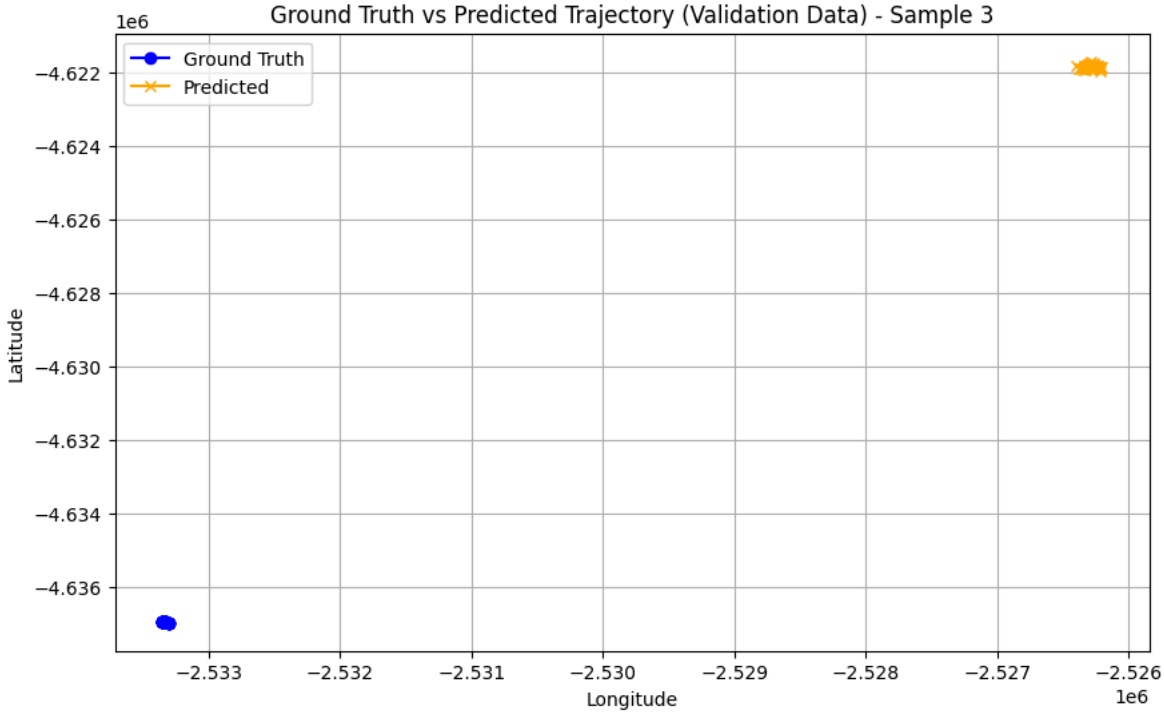
[+]


Ground Truth vs Predicted Trajectory (Validation Data)
1e6
Ground Truth
Predicted
−4.622
−4.624
−4.626
−4.628
−4.630
−4.632
−4.634
−4.636
Latitude
−2.536
−2.534
−2.532
−2.530
−2.528
−2.526
1e6
Longitude

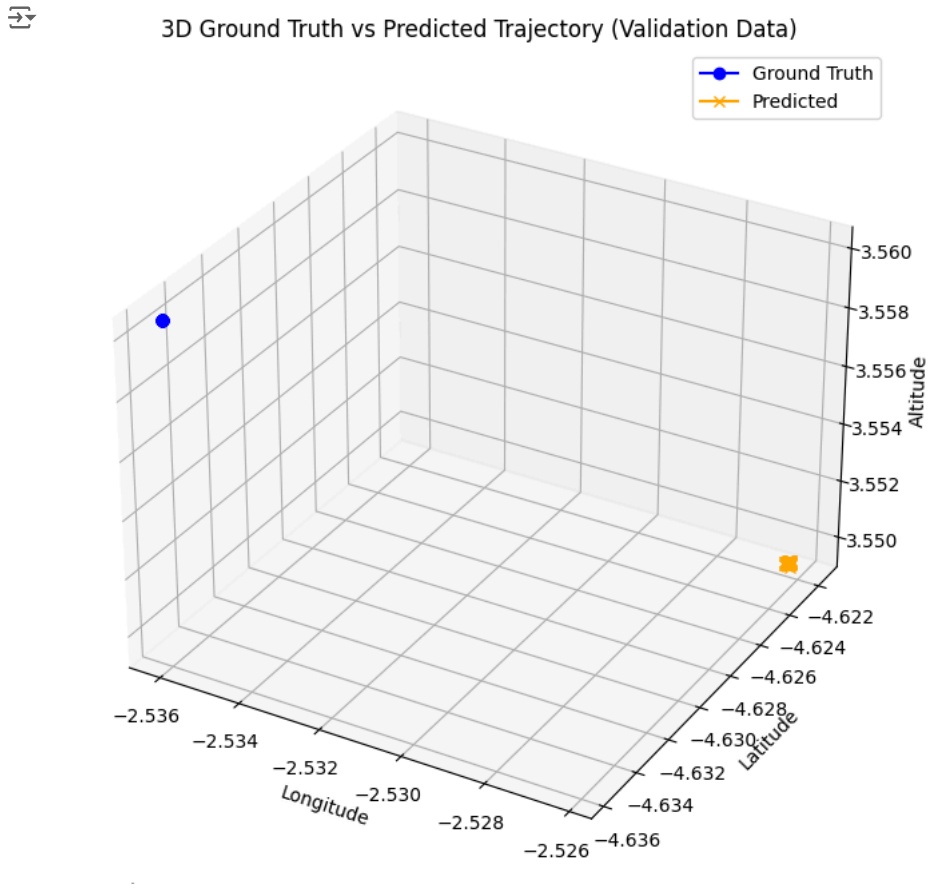

Ground Truth vs Predicted Trajectory (Validation Data) - Sample 1
1e6
Ground Truth
Predicted
−4.622
−4.624
−4.626
−4.628
−4.630
−4.632
−4.634
−4.636
Latitude
−2.536
−2.534
−2.532
−2.530
−2.528
−2.526
1e6
Longitude


Ground Truth vs Predicted Trajectory (Validation Data) - Sample 2
1e6
Ground Truth
Predicted
−4.622
−4.624
−4.626
−4.628
−4.630
−4.632
−4.634
Latitude



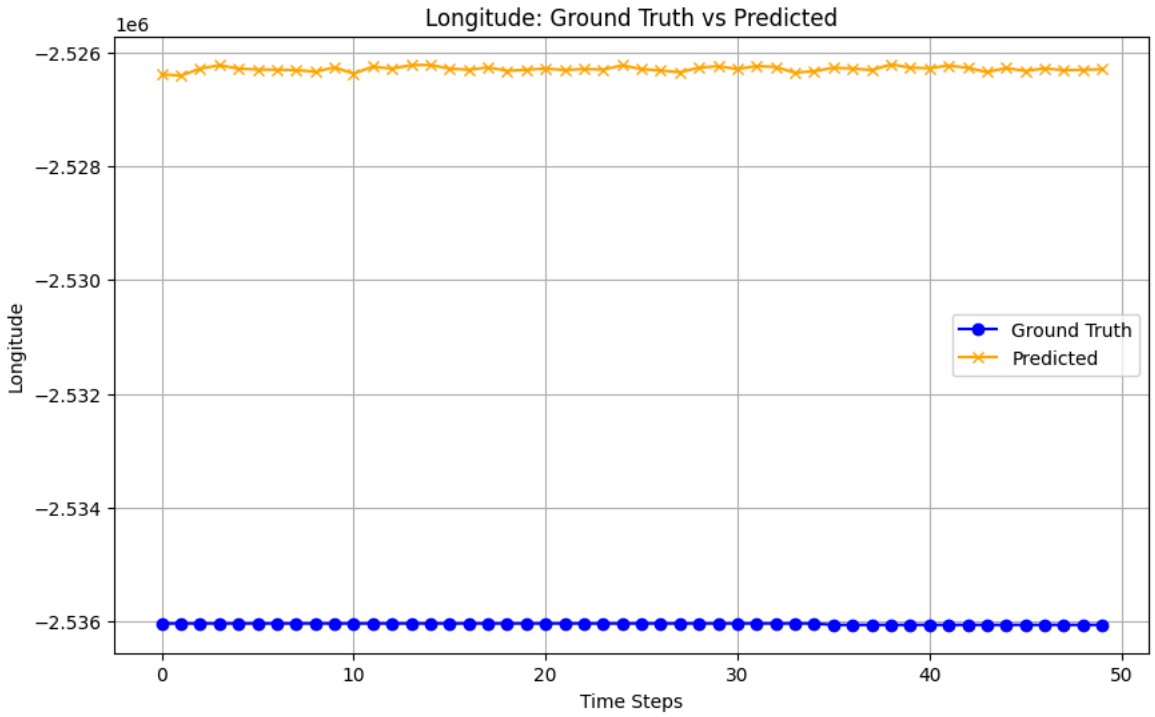


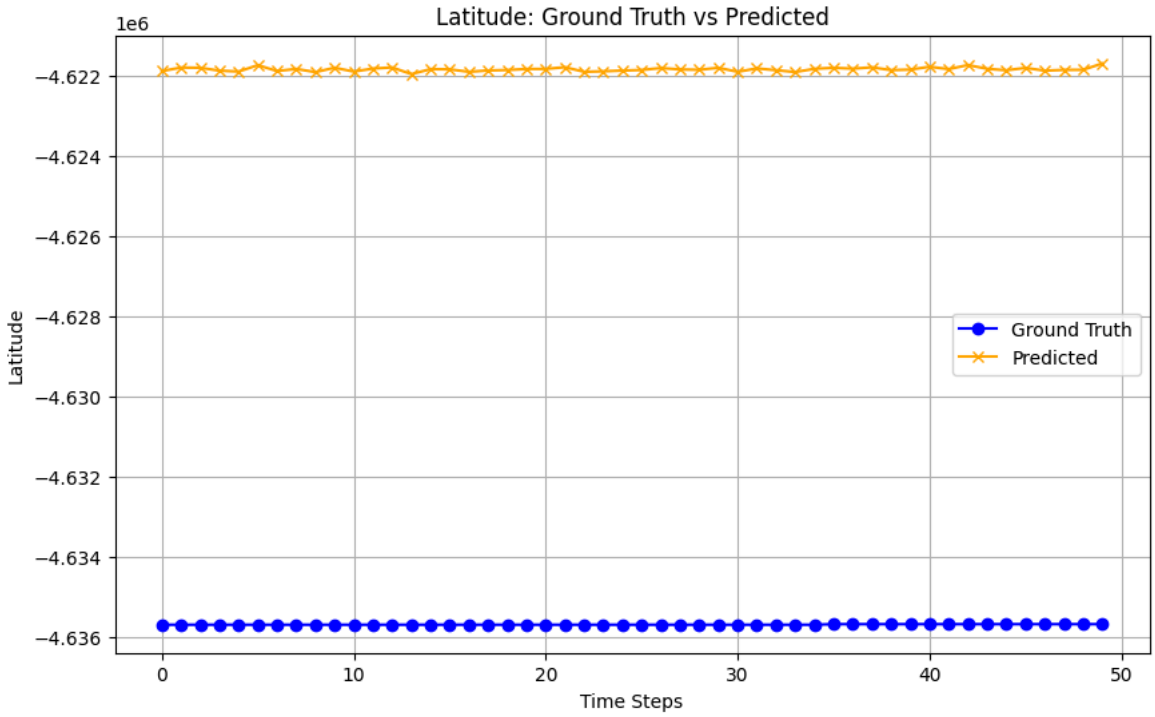

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Evaluate the model on the validation set
model.eval()
with torch.no_grad():
    y_pred = model(X_val)

# Plotting the first trajectory in 3D
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Ground Truth
ax.plot(y_val[0, :, 0], y_val[0, :, 1], y_val[0, :, 2], label='Ground Truth', marker='o', color='blue')

# Predicted
ax.plot(y_pred[0, :, 0], y_pred[0, :, 1], y_pred[0, :, 2], label='Predicted', marker='x', color='orange')

ax.set_title('3D Ground Truth vs Predicted Trajectory (Validation Data)')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.set_zlabel('Altitude')
ax.legend()
plt.show()
```

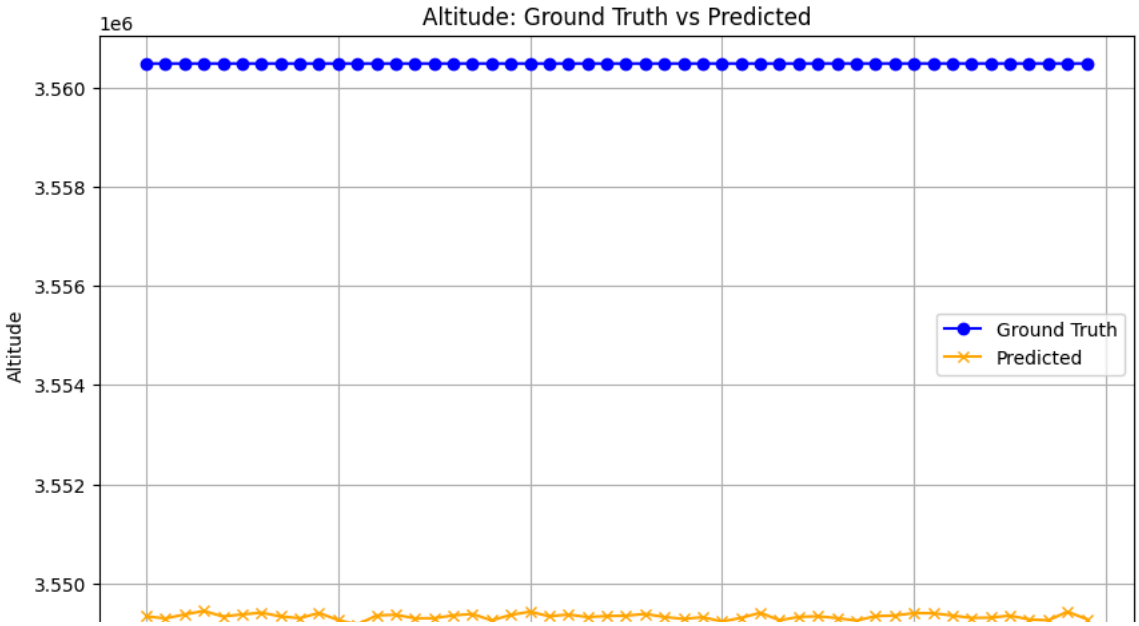



```
# Longitude Plot
plt.figure(figsize=(10, 6))
plt.plot(y_val[0, :, 0], label='Ground Truth', marker='o', linestyle='-', color='blue')
plt.plot(y_pred[0, :, 0], label='Predicted', marker='x', linestyle='-', color='orange')
plt.title('Longitude: Ground Truth vs Predicted')
plt.xlabel('Time Steps')
plt.ylabel('Longitude')
plt.legend()
plt.grid(True)
plt.show()

# Latitude Plot
plt.figure(figsize=(10, 6))
plt.plot(y_val[0, :, 1], label='Ground Truth', marker='o', linestyle='-', color='blue')
plt.plot(y_pred[0, :, 1], label='Predicted', marker='x', linestyle='-', color='orange')
plt.title('Latitude: Ground Truth vs Predicted')
plt.xlabel('Time Steps')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()

# Altitude Plot
plt.figure(figsize=(10, 6))
plt.plot(y_val[0, :, 2], label='Ground Truth', marker='o', linestyle='-', color='blue')
plt.plot(y_pred[0, :, 2], label='Predicted', marker='x', linestyle='-', color='orange')
plt.title('Altitude: Ground Truth vs Predicted')
plt.xlabel('Time Steps')
plt.ylabel('Altitude')
plt.legend()
plt.grid(True)
plt.show()
```

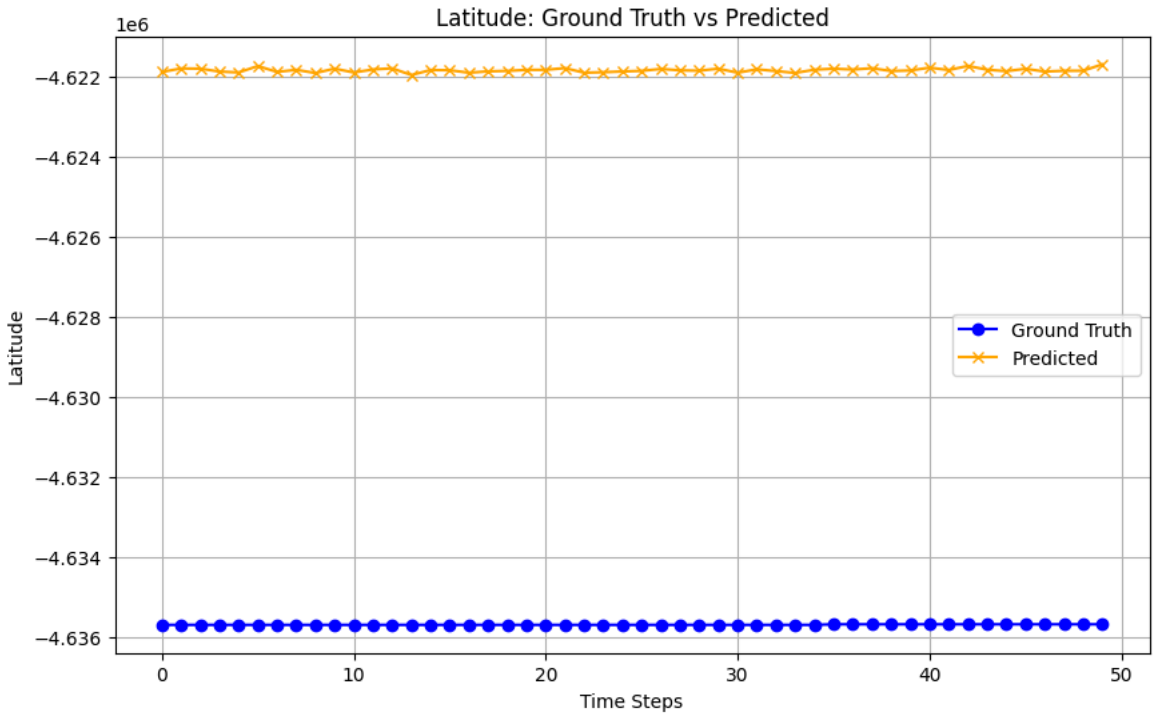
[+]

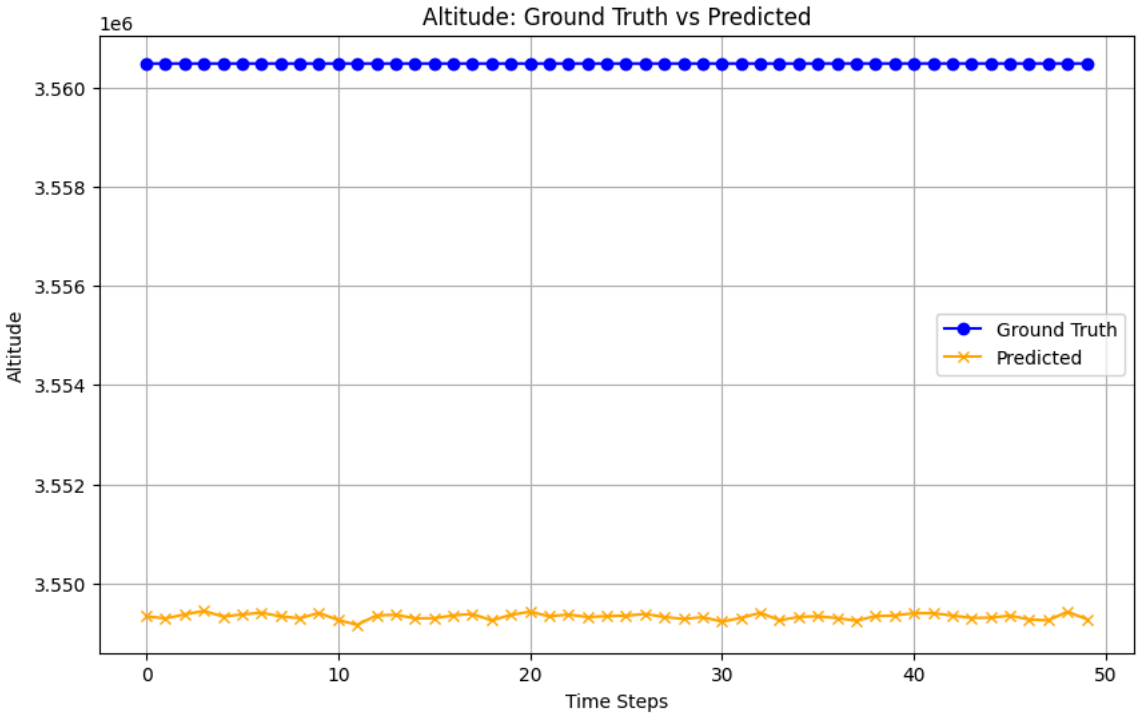

Longitude: Ground Truth vs Predicted
1e6
−2.526
−2.528
−2.530
−2.532
−2.534
−2.536
Longitude
0
10
20
30
40
50
Time Steps
Ground Truth
Predicted


Latitude: Ground Truth vs Predicted
1e6
−4.622
−4.624
−4.626
−4.628
−4.630
−4.632
−4.634
−4.636
Latitude
0
10
20
30
40
50
Time Steps
Ground Truth
Predicted


Altitude: Ground Truth vs Predicted
1e6
3.560
3.558
3.556
3.554
3.552
3.550
Altitude
Ground Truth
Predicted


```
# Latitude Plot
plt.figure(figsize=(10, 6))
plt.plot(y_val[0, :, 1], label='Ground Truth', marker='o', linestyle='-', color='blue')
plt.plot(y_pred[0, :, 1], label='Predicted', marker='x', linestyle='-', color='orange')
plt.title('Latitude: Ground Truth vs Predicted')
plt.xlabel('Time Steps')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()

# Altitude Plot
plt.figure(figsize=(10, 6))
plt.plot(y_val[0, :, 2], label='Ground Truth', marker='o', linestyle='-', color='blue')
plt.plot(y_pred[0, :, 2], label='Predicted', marker='x', linestyle='-', color='orange')
plt.title('Altitude: Ground Truth vs Predicted')
plt.xlabel('Time Steps')
plt.ylabel('Altitude')
plt.legend()
plt.grid(True)
plt.show()
```

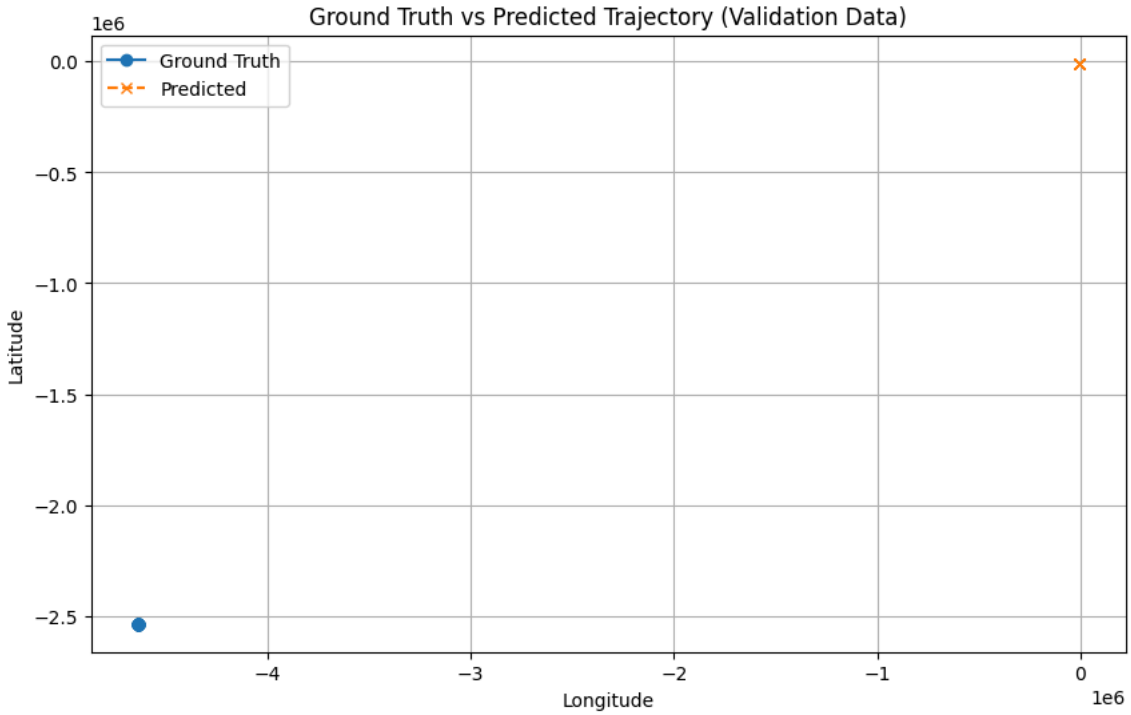


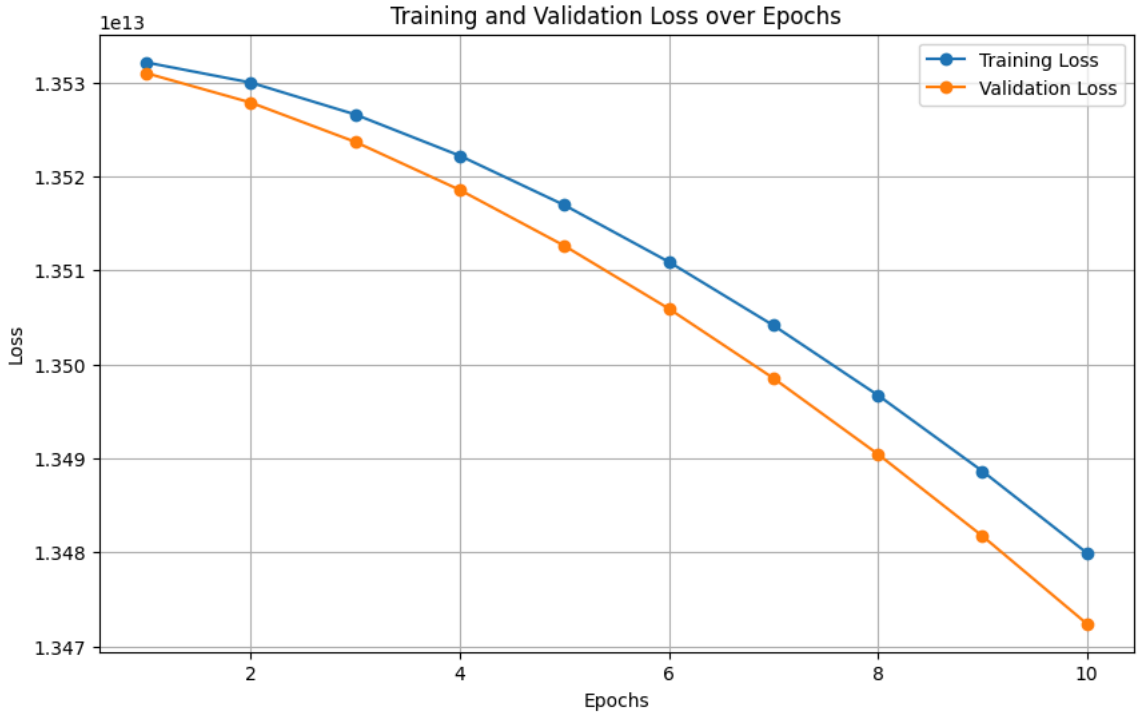


Code for Calculating Metrics and Plotting Trajectories

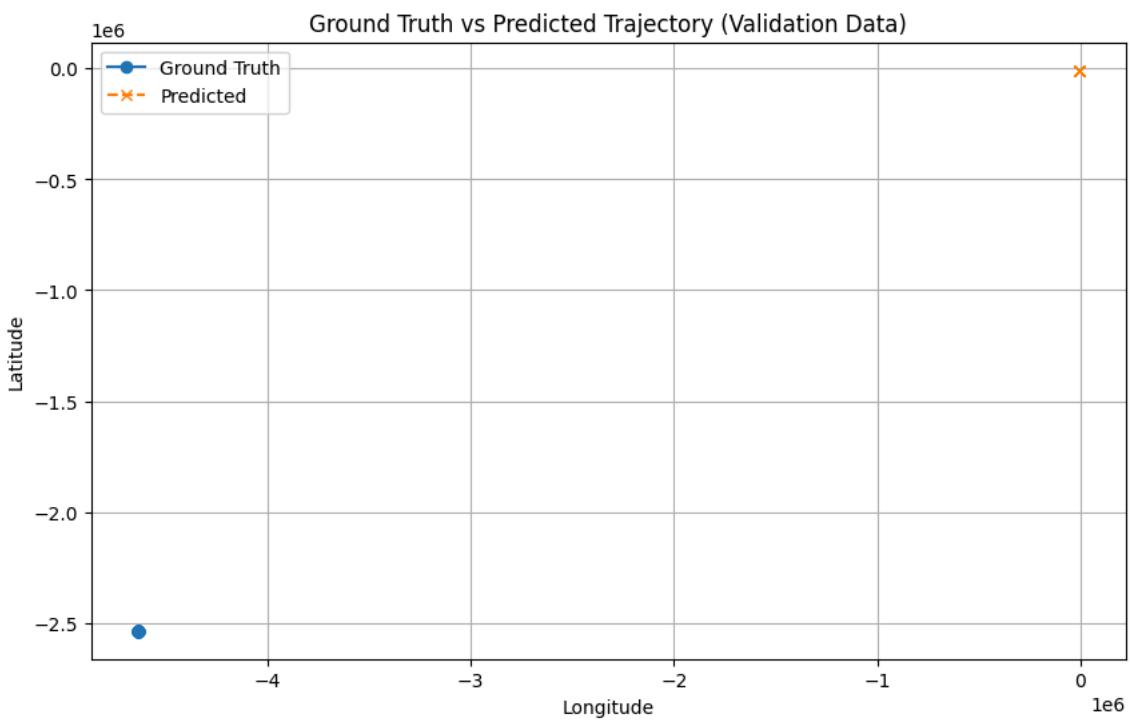
```
import torch
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error, mean_absolute_error

# Ensure the validation data is already a PyTorch tensor
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.float32)

# Set the model to evaluation mode
model.eval()

# Disable gradient computation for inference
with torch.no_grad():
    val_position_pred = model(X_val)

# Convert predictions and labels to numpy arrays for metric calculation
val_position_pred_np = val_position_pred.numpy()
y_val_np = y_val.numpy()

# Flatten the arrays along the sequence length and features dimensions
val_position_pred_np_flat = val_position_pred_np.reshape(-1, val_position_pred_np.shape[-1])
y_val_np_flat = y_val_np.reshape(-1, y_val_np.shape[-1])

# Calculate metrics
rmse = np.sqrt(mean_squared_error(y_val_np_flat, val_position_pred_np_flat))
mae = mean_absolute_error(y_val_np_flat, val_position_pred_np_flat)
pe = np.mean(np.linalg.norm(y_val_np_flat - val_position_pred_np_flat, axis=1))

print(f'RMSE: {rmse}')
print(f'MAE: {mae}')
print(f'Position Error (PE): {pe}')

# Plotting the results - Example for the first trajectory in the validation set
plt.figure(figsize=(10, 6))
plt.plot(y_val_np[0, :, 1], y_val_np[0, :, 0], label='Ground Truth', marker='o')
plt.plot(val_position_pred_np[0, :, 1], val_position_pred_np[0, :, 0], label='Predicted', marker='x', linestyle='--')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Ground Truth vs Predicted Trajectory (Validation Data)')
plt.legend()
plt.grid(True)
plt.show()

# Plot training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(train_losses) + 1), train_losses, label='Training Loss', marker='o')
plt.plot(range(1, len(val_losses) + 1), val_losses, label='Validation Loss', marker='o')
plt.title('Training and Validation Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
<ipython-input-56-ab027d1cbad1>:7: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.c
  X_val = torch.tensor(X_val, dtype=torch.float32)
<ipython-input-56-ab027d1cbad1>:8: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.c
  y_val = torch.tensor(y_val, dtype=torch.float32)
RMSE: 3671889.25
MAE: 3566780.0
Position Error (PE): 6357454.0
```






```
# Use a smaller subset for visualization
subset_idx = 0  # Choose a specific trajectory or subset of data

plt.figure(figsize=(10, 6))
plt.plot(y_val_np[subset_idx, :, 1], y_val_np[subset_idx, :, 0], label='Ground Truth', marker='o')
plt.plot(val_position_pred_np[subset_idx, :, 1], val_position_pred_np[subset_idx, :, 0], label='Predicted', marker='x', lin
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Ground Truth vs Predicted Trajectory (Validation Data)')
plt.legend()
plt.grid(True)
plt.show()
```




```
# Convert validation data to tensors if not already done
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.float32)

# Evaluate the model on validation data
model.eval()
with torch.no_grad():
    val_position_pred = model(X_val)
```

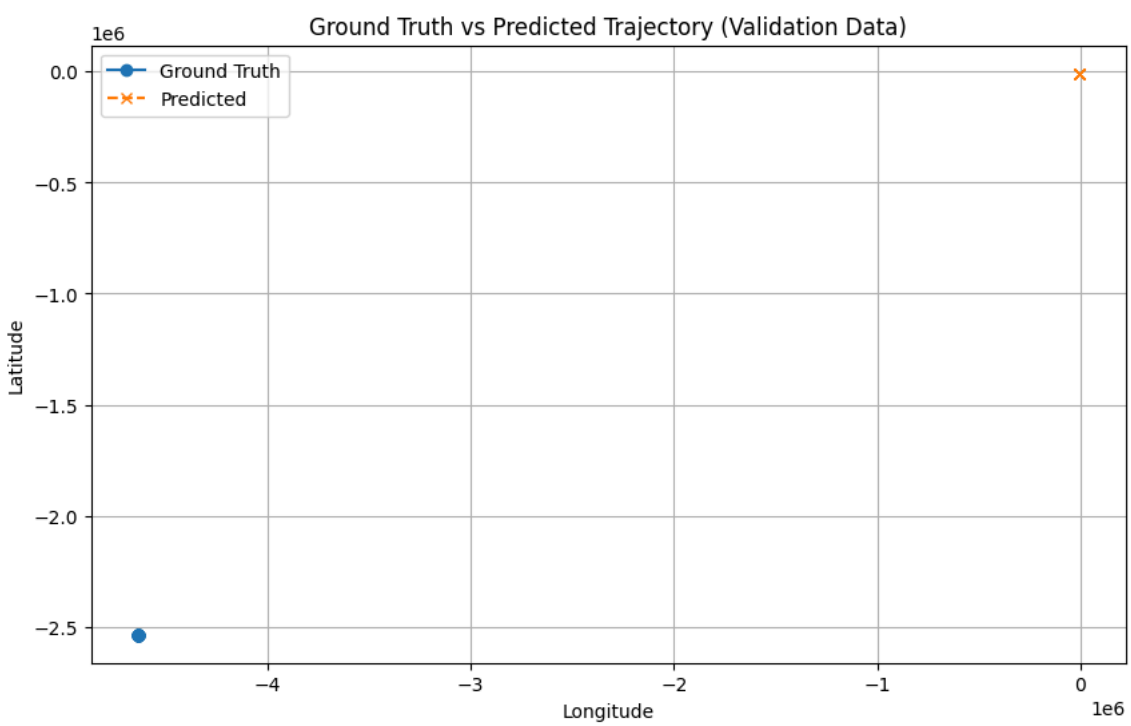
```
<ipython-input-72-f08ef29868eb>:2: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.c
  X_val = torch.tensor(X_val, dtype=torch.float32)
<ipython-input-72-f08ef29868eb>:3: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.c
  y_val = torch.tensor(y_val, dtype=torch.float32)
```

```
# Convert predictions and labels to numpy arrays
val_position_pred_np = val_position_pred.numpy()
y_val_np = y_val.numpy()

# Flatten arrays for metric calculation
val_position_pred_np_flat = val_position_pred_np.reshape(-1, val_position_pred_np.shape[-1])
y_val_np_flat = y_val_np.reshape(-1, y_val_np.shape[-1])
```

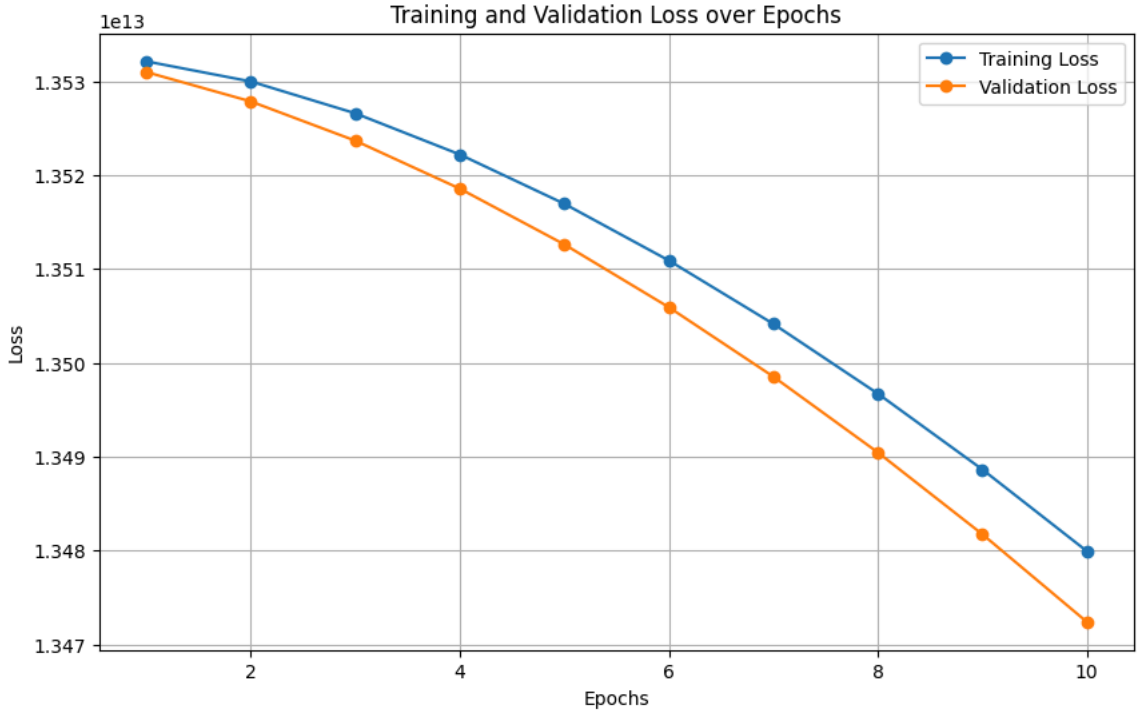
```
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np

# Calculate metrics
rmse = np.sqrt(mean_squared_error(y_val_np_flat, val_position_pred_np_flat))
mae = mean_absolute_error(y_val_np_flat, val_position_pred_np_flat)
pe = np.mean(np.linalg.norm(y_val_np_flat - val_position_pred_np_flat, axis=1))

print(f'RMSE: {rmse}')
print(f'MAE: {mae}')
print(f'Position Error (PE): {pe}')
```

```
RMSE: 3671889.25
MAE: 3566780.0
Position Error (PE): 6357454.0
```

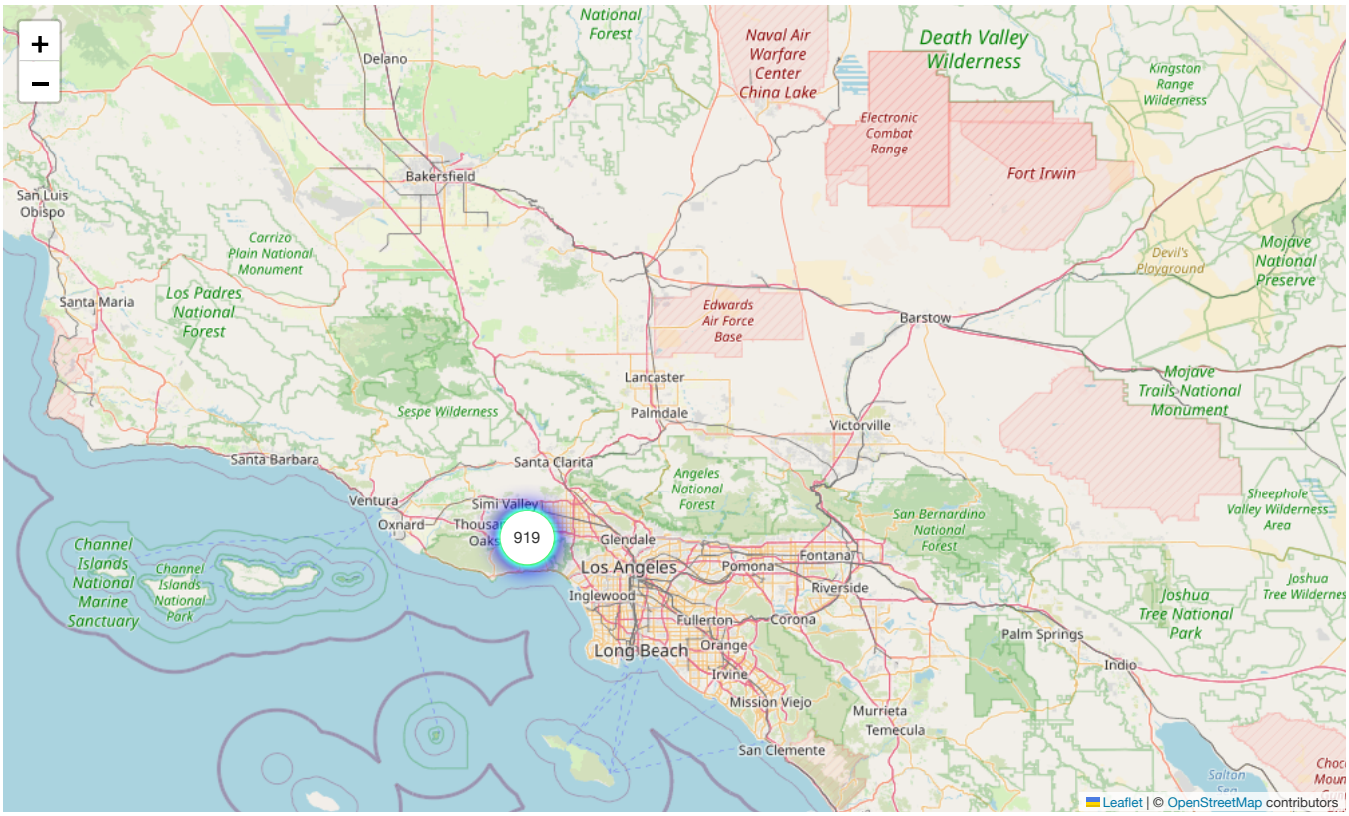
```
import matplotlib.pyplot as plt

# Plotting Ground Truth vs Predicted Trajectory
plt.figure(figsize=(10, 6))
plt.plot(y_val_np[0, :, 1], y_val_np[0, :, 0], label='Ground Truth', marker='o')
plt.plot(val_position_pred_np[0, :, 1], val_position_pred_np[0, :, 0], label='Predicted', marker='x', linestyle='--')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Ground Truth vs Predicted Trajectory (Validation Data)')
plt.legend()
plt.grid(True)
plt.show()
```




```
# Assuming train_losses and val_losses are stored during training
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(train_losses) + 1), train_losses, label='Training Loss', marker='o')
plt.plot(range(1, len(val_losses) + 1), val_losses, label='Validation Loss', marker='o')
plt.title('Training and Validation Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```




```
import folium
from folium import plugins

# Assuming 'ground_truth' DataFrame has 'LatitudeDegrees' and 'LongitudeDegrees' columns

# Extract locations from ground truth data
df_locs = list(ground_truth[['LatitudeDegrees', 'LongitudeDegrees']].values)

# Initialize the map centered around the median latitude and longitude
fol_map = folium.Map(location=[ground_truth['LatitudeDegrees'].median(),
                               ground_truth['LongitudeDegrees'].median()],
                     zoom_start=11)

# Add heatmap layer (optional)
heat_map = plugins.HeatMap(df_locs)
fol_map.add_child(heat_map)

# Add marker cluster to map
markers = plugins.MarkerCluster(locations=df_locs)
fol_map.add_child(markers)

# Display the map
fol_map
```



Method 2

```
import pandas as pd

# Load GNSS data
gnss_data = pd.read_csv('/content/device_gnss 2.csv')

# Load IMU data
imu_data = pd.read_csv('/content/device_imu 5.csv')

# Load Ground Truth data
ground_truth = pd.read_csv('/content/ground_truth (1).csv')

# Display the first few rows to understand the structure
print(gnss_data.head())
print(imu_data.head())
print(ground_truth.head())
```

```
  MessageType  utcTimeMillis    TimeNanos  LeapSecond        FullBiasNanos  \
0         Raw  1638995330000  38533000000         NaN -1323030509467011361
1         Raw  1638995330000  38533000000         NaN -1323030509467011361
2         Raw  1638995330000  38533000000         NaN -1323030509467011361
3         Raw  1638995330000  38533000000         NaN -1323030509467011361
4         Raw  1638995330000  38533000000         NaN -1323030509467011361

   BiasNanos  BiasUncertaintyNanos  DriftNanosPerSecond  \
0        0.0              8.029921                319.0
1        0.0              8.029921                319.0
2        0.0              8.029921                319.0
3        0.0              8.029921                319.0
4        0.0              8.029921                319.0

   DriftUncertaintyNanosPerSecond  HardwareClockDiscontinuityCount  ...  \
0                             1.0                               19  ...
1                             1.0                               19  ...
2                             1.0                               19  ...
3                             1.0                               19  ...
```

```
4                           1.0                                19  ...

   SvVelocityYEcefMetersPerSecond  SvVelocityZEcefMetersPerSecond  \
0                             NaN                             NaN
1                     -918.967610                    -3028.184129
2                    -1996.536510                     -420.561083
3                     -500.485764                    -2268.480805
4                     -239.856045                     3115.450967

   SvClockBiasMeters  SvClockDriftMetersPerSecond  IsrbMeters  \
0                NaN                          NaN         NaN
1     -10654.452279                    -0.003417         0.0
2     -59856.843278                     0.000356         0.0
3      41268.098680                     0.003205         0.0
4      86469.396304                     0.002502         0.0

   IonosphericDelayMeters  TroposphericDelayMeters  WlsPositionXEcefMeters  \
0                     NaN                      NaN           -2.532065e+06
1                2.866442                 2.751123           -2.532065e+06
2                2.810502                 2.800855           -2.532065e+06
3                6.039320                10.711802           -2.532065e+06
4                5.464260                 6.465938           -2.532065e+06

   WlsPositionYEcefMeters  WlsPositionZEcefMeters
0           -4.637480e+06            3.561014e+06
1           -4.637480e+06            3.561014e+06
2           -4.637480e+06            3.561014e+06
3           -4.637480e+06            3.561014e+06
4           -4.637480e+06            3.561014e+06

[5 rows x 47 columns]
  MessageType  utcTimeMillis  MeasurementX  MeasurementY  MeasurementZ  BiasX  \
0  UncalAccel  1638995329555      0.007183      9.636961     -1.704672   0.00
1    UncalMag  1638995329555     28.619999    -32.399998    -66.360000  55.68
2  UncalAccel  1638995329565      0.011971      9.620202     -1.714248   0.00
3   UncalGyro  1638995329565     -0.006567      0.000000     -0.000764   0.00
4    UncalMag  1638995329565     29.039999    -32.040000    -66.540000  55.68

      BiasY  BiasZ
```

```
# Merge GNSS and IMU data based on a common timestamp
merged_data = pd.merge_asof(gnss_data.sort_values('utcTimeMillis'),
                            imu_data.sort_values('utcTimeMillis'),
                            on='utcTimeMillis')

# Merge with ground truth data
merged_data = pd.merge_asof(merged_data.sort_values('utcTimeMillis'),
                            ground_truth.sort_values('UnixTimeMillis'),
                            left_on='utcTimeMillis',
                            right_on='UnixTimeMillis')

# Display the structure of the merged data
print(merged_data.head())
```

```
  MessageType_x  utcTimeMillis    TimeNanos  LeapSecond        FullBiasNanos  \
0           Raw  1638995330000  38533000000         NaN -1323030509467011361
1           Raw  1638995330000  38533000000         NaN -1323030509467011361
2           Raw  1638995330000  38533000000         NaN -1323030509467011361
3           Raw  1638995330000  38533000000         NaN -1323030509467011361
4           Raw  1638995330000  38533000000         NaN -1323030509467011361

   BiasNanos  BiasUncertaintyNanos  DriftNanosPerSecond  \
0        0.0              8.029921                319.0
1        0.0              8.029921                319.0
2        0.0              8.029921                319.0
3        0.0              8.029921                319.0
4        0.0              8.029921                319.0

   DriftUncertaintyNanosPerSecond  HardwareClockDiscontinuityCount  ...  \
0                             1.0                               19  ...
1                             1.0                               19  ...
2                             1.0                               19  ...
3                             1.0                               19  ...
4                             1.0                               19  ...

   BiasZ  MessageType  Provider  LatitudeDegrees  LongitudeDegrees  \
0  -13.2          Fix        GT        34.157033       -118.634522
1  -13.2          Fix        GT        34.157033       -118.634522
2  -13.2          Fix        GT        34.157033       -118.634522
3  -13.2          Fix        GT        34.157033       -118.634522
4  -13.2          Fix        GT        34.157033       -118.634522

   AltitudeMeters  SpeedMps  AccuracyMeters  BearingDegrees  UnixTimeMillis
0      250.124001     0.007             0.1        55.88382   1638995330000
1      250.124001     0.007             0.1        55.88382   1638995330000
2      250.124001     0.007             0.1        55.88382   1638995330000
3      250.124001     0.007             0.1        55.88382   1638995330000
```

```
4      250.124001     0.007             0.1        55.88382   1638995330000

[5 rows x 63 columns]
```

Double-click (or enter) to edit

```
from sklearn.preprocessing import StandardScaler
import numpy as np

# Select relevant features
features = merged_data[['MeasurementX', 'MeasurementY', 'MeasurementZ',
                        'Cn0DbHz', 'PseudorangeRateMetersPerSecond',
                        'AccumulatedDeltaRangeMeters']]

# Labels (ground truth positions)
labels = merged_data[['LatitudeDegrees', 'LongitudeDegrees', 'AltitudeMeters']]

# Normalization
scaler = StandardScaler()
features = scaler.fit_transform(features)

# Sliding window
def sliding_window(data, labels, window_size):
    X = np.array([data[i:i+window_size] for i in range(len(data)-window_size)])
    y = np.array([labels[i:i+window_size] for i in range(len(labels)-window_size)])
    return X, y

# Apply sliding window
window_size = 50  # Example window size
X, y = sliding_window(features, labels.values, window_size)

print(X.shape, y.shape)  # Check shapes
```

```
(35105, 50, 6) (35105, 50, 3)
```

```
import torch
import torch.nn as nn
import torch.optim as optim
class SpatialEncoder(nn.Module):
    def __init__(self):
        super(SpatialEncoder, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(64)
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(128)
        self.fc = nn.Linear(128, 256)

    def forward(self, x):
        x = x.transpose(1, 2)  # Transpose to make the channels dimension the second dimension
        x = self.conv1(x)
        x = self.bn1(x)
        x = nn.ReLU()(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = nn.ReLU()(x)
        x = torch.mean(x, dim=-1)  # Global average pooling
        x = self.fc(x)
        return x

# Example usage
spatial_encoder = SpatialEncoder()
imu_data_tensor = torch.tensor(X[:, :, :3], dtype=torch.float32)  # Assuming the first 3 channels are IMU data
imu_encoded_features = spatial_encoder(imu_data_tensor)
```

```
class TemporalEncoder(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers):
        super(TemporalEncoder, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True, bidirectional=True)

    def forward(self, x):
        out, _ = self.lstm(x)
        return out

# Example usage
temporal_encoder = TemporalEncoder(input_dim=256, hidden_dim=128, num_layers=2)
temporal_features = temporal_encoder(imu_encoded_features)



from sklearn.model_selection import train_test_split
import torch

# Assuming X and y are your full datasets from the sliding window process
# X -> Input data with shape (n_samples, sequence_length, n_features)
# y -> Labels with shape (n_samples, sequence_length, n_targets)

# Split data into 70% train, 15% validation, 15% test
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)  # 70% train, 30% temp
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)  # 15% val, 15% test

# Convert data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
y_val = torch.tensor(y_val, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)



import torch.nn as nn

class TransformerFusionModel(nn.Module):
    def __init__(self):
        super(TransformerFusionModel, self).__init__()
        self.transformer = nn.Transformer(d_model=128, nhead=4, num_encoder_layers=3, batch_first=True)
        self.fc_fusion = nn.Linear(128, 64)
        self.fc_output = nn.Linear(64, 3)

    def forward(self, src, tgt):
        transformer_output = self.transformer(src, tgt)
        fused_features = torch.relu(self.fc_fusion(transformer_output))
        output = self.fc_output(fused_features)
        return output
```

```
import torch.optim as optim
import matplotlib.pyplot as plt

# Initialize the model, loss function, and optimizer
model = TransformerFusionModel()
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_losses = []
val_losses = []

# Training loop
epochs = 10
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    # Forward pass
    tgt = torch.zeros_like(X_train)  # Assuming tgt is initially zeros
    outputs = model(X_train, tgt)

    # Compute loss
    loss = criterion(outputs, y_train)

    # Backward pass and optimization
    loss.backward()
    optimizer.step()

    # Store training loss
    train_losses.append(loss.item())

    # Validation step
    model.eval()
    with torch.no_grad():
        tgt_val = torch.zeros_like(X_val)
        val_outputs = model(X_val, tgt_val)
        val_loss = criterion(val_outputs, y_val)
        val_losses.append(val_loss.item())

    print(f'Epoch {epoch+1}/{epochs}, Training Loss: {loss.item()}, Validation Loss: {val_loss.item()}')

# Test set evaluation
model.eval()
with torch.no_grad():
    tgt_test = torch.zeros_like(X_test)
    test_outputs = model(X_test, tgt_test)
    test_loss = criterion(test_outputs, y_test)
    print(f'Test Loss: {test_loss.item()}')

# Plot the loss curves
plt.figure(figsize=(10, 6))
plt.plot(range(1, epochs + 1), train_losses, label='Training Loss')
plt.plot(range(1, epochs + 1), val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-21-0c21a2dc6dd8> in <cell line: 14>()
     18     # Forward pass
     19     tgt = torch.zeros_like(X_train)  # Assuming tgt is initially zeros
---> 20     outputs = model(X_train, tgt)
     21
     22     # Compute loss

                                5 frames
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/transformer.py in forward(self, src, tgt, src_mask, tgt_mask,
memory_mask, src_key_padding_mask, tgt_key_padding_mask, memory_key_padding_mask, src_is_causal, tgt_is_causal,
memory_is_causal)
    213
    214         if src.size(-1) != self.d_model or tgt.size(-1) != self.d_model:
--> 215             raise RuntimeError("the feature number of src and tgt must be equal to d_model")
    216
    217         memory = self.encoder(src, mask=src_mask, src_key_padding_mask=src_key_padding_mask,

RuntimeError: the feature number of src and tgt must be equal to d_model
```

Mehtod 3

```
import pandas as pd

# Load GNSS data
gnss_data = pd.read_csv('/content/device_gnss 2.csv')

# Load IMU data
imu_data = pd.read_csv('/content/device_imu 5.csv')

# Load Ground Truth data
ground_truth = pd.read_csv('/content/ground_truth (1).csv')

# Display the first few rows of each dataset to confirm successful loading
print("GNSS Data Head:")
print(gnss_data.head())

print("\nIMU Data Head:")
print(imu_data.head())

print("\nGround Truth Data Head:")
print(ground_truth.head())

# Check for missing values in each dataset
print("\nMissing Values in GNSS Data:")
print(gnss_data.isnull().sum())

print("\nMissing Values in IMU Data:")
print(imu_data.isnull().sum())

print("\nMissing Values in Ground Truth Data:")
print(ground_truth.isnull().sum())
```

```
AccuracyMeters      0
BearingDegrees      0
UnixTimeMillis      0
dtype: int64
```

```
# Drop columns with excessive missing values (example: ArrivalTimeNanosSinceGpsEpoch and related columns)
columns_to_drop = ['ArrivalTimeNanosSinceGpsEpoch', 'RawPseudorangeMeters', 'RawPseudorangeUncertaintyMeters',
                   'SignalType', 'ReceivedSvTimeNanosSinceGpsEpoch', 'SvPositionXEcefMeters',
                   'SvPositionYEcefMeters', 'SvPositionZEcefMeters', 'SvElevationDegrees',
                   'SvAzimuthDegrees', 'SvVelocityXEcefMetersPerSecond', 'SvVelocityYEcefMetersPerSecond',
                   'SvVelocityZEcefMetersPerSecond', 'SvClockBiasMeters', 'SvClockDriftMetersPerSecond',
                   'IsrbMeters', 'IonosphericDelayMeters', 'TroposphericDelayMeters']

gnss_data_cleaned = gnss_data.drop(columns=columns_to_drop)

# Handle remaining missing values, for simplicity we will forward fill them
gnss_data_cleaned = gnss_data_cleaned.fillna(method='ffill')

# Merge GNSS and IMU data based on utcTimeMillis
merged_data = pd.merge_asof(gnss_data_cleaned.sort_values('utcTimeMillis'),
                            imu_data.sort_values('utcTimeMillis'),
                            on='utcTimeMillis')

# Merge the resulting dataset with the Ground Truth data based on utcTimeMillis
merged_data = pd.merge_asof(merged_data.sort_values('utcTimeMillis'),
                            ground_truth.sort_values('UnixTimeMillis'),
                            left_on='utcTimeMillis', right_on='UnixTimeMillis')

print("\nMerged Data Head:")
print(merged_data.head())

# Check for any remaining missing values after merging
print("\nMissing Values in Merged Data:")
print(merged_data.isnull().sum())
```

```
   AltitudeMeters  SpeedMps  AccuracyMeters  BearingDegrees  UnixTimeMillis
0      250.124001     0.007             0.1        55.88382   1638995330000
1      250.124001     0.007             0.1        55.88382   1638995330000
2      250.124001     0.007             0.1        55.88382   1638995330000
3      250.124001     0.007             0.1        55.88382   1638995330000
4      250.124001     0.007             0.1        55.88382   1638995330000

[5 rows x 45 columns]

Missing Values in Merged Data:
MessageType_x                           0
utcTimeMillis                           0
TimeNanos                               0
LeapSecond                          35155
FullBiasNanos                           0
BiasNanos                               0
BiasUncertaintyNanos                    0
DriftNanosPerSecond                     0
DriftUncertaintyNanosPerSecond          0
HardwareClockDiscontinuityCount         0
Svid                                    0
```

```
Speedmps                                0
AccuracyMeters                          0
BearingDegrees                          0
UnixTimeMillis                          0
dtype: int64
<ipython-input-14-7fae8eed5437>:12: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a futu
  gnss_data_cleaned = gnss_data_cleaned.fillna(method='ffill')
```

```
# Drop the LeapSecond column
merged_data = merged_data.drop(columns=['LeapSecond'])

# Select relevant features for the model
selected_features = ['MeasurementX', 'MeasurementY', 'MeasurementZ',  # IMU data
                     'Cn0DbHz', 'PseudorangeRateMetersPerSecond',  # GNSS data
                     'WlsPositionXEcefMeters', 'WlsPositionYEcefMeters', 'WlsPositionZEcefMeters',  # GNSS position
                     'LatitudeDegrees', 'LongitudeDegrees', 'AltitudeMeters']  # Ground truth

# Target labels (ground truth)
target_labels = ['LatitudeDegrees', 'LongitudeDegrees', 'AltitudeMeters']

# Extract selected features and target labels
X = merged_data[selected_features].values
y = merged_data[target_labels].values

print(f"Shape of X: {X.shape}")
print(f"Shape of y: {y.shape}")
```

```
Shape of X: (35155, 11)
Shape of y: (35155, 3)
```

```
import numpy as np

# Define sliding window function
def sliding_window(data, labels, window_size):
    X = []
    y = []
    for i in range(len(data) - window_size + 1):
        X.append(data[i:i+window_size])
        y.append(labels[i:i+window_size])
    return np.array(X), np.array(y)

# Apply sliding window
window_size = 50  # Example window size
X, y = sliding_window(X, y, window_size)

print(f"Shape of X after sliding window: {X.shape}")
print(f"Shape of y after sliding window: {y.shape}")
```

```
Shape of X after sliding window: (35106, 50, 11)
Shape of y after sliding window: (35106, 50, 3)
```

```
import torch

# Convert to PyTorch tensors if not already done
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

print(f'Shape of X_train_tensor: {X_train_tensor.shape}')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-16-5303a6fd4a65> in <cell line: 4>()
      2
      3 # Convert to PyTorch tensors if not already done
----> 4 X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
      5 X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
      6 X_test_tensor = torch.tensor(X_test, dtype=torch.float32)

NameError: name 'X_train' is not defined
```

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


class SpatialEncoder(nn.Module):
    def __init__(self, in_channels=3, out_channels=64, kernel_size=3, sequence_length=50):
        super(SpatialEncoder, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=in_channels, out_channels=out_channels, kernel_size=kernel_size, padding=1)
        self.bn1 = nn.BatchNorm1d(out_channels)
        self.conv2 = nn.Conv1d(in_channels=out_channels, out_channels=out_channels, kernel_size=kernel_size, padding=1)
        self.bn2 = nn.BatchNorm1d(out_channels)
        self.fc = nn.Linear(out_channels * sequence_length, 256)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = x.view(x.size(0), -1)  # Flatten the feature map
        x = self.fc(x)
        return x

# Apply spatial encoder to the IMU data
spatial_encoder = SpatialEncoder()
X_train_imu = X_train_tensor[:, :, :3].permute(0, 2, 1)  # (batch_size, channels, sequence_length)
X_val_imu = X_val_tensor[:, :, :3].permute(0, 2, 1)
X_test_imu = X_test_tensor[:, :, :3].permute(0, 2, 1)

# Get the encoded features
imu_encoded_train = spatial_encoder(X_train_imu)
imu_encoded_val = spatial_encoder(X_val_imu)
imu_encoded_test = spatial_encoder(X_test_imu)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-653ca9491ee6> in <cell line: 27>()
     25 # Apply spatial encoder to the IMU data
     26 spatial_encoder = SpatialEncoder()
---> 27 X_train_imu = X_train_tensor[:, :, :3].permute(0, 2, 1)  # (batch_size, channels, sequence_length)
     28 X_val_imu = X_val_tensor[:, :, :3].permute(0, 2, 1)
     29 X_test_imu = X_test_tensor[:, :, :3].permute(0, 2, 1)

NameError: name 'X_train_tensor' is not defined
```

```
class TemporalEncoder(nn.Module):
    def __init__(self, input_dim=256, hidden_dim=128, num_layers=2):
        super(TemporalEncoder, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, 256)  # Bidirectional LSTM doubles the hidden dimension

    def forward(self, x):
        lstm_out, _ = self.lstm(x)
        out = self.fc(lstm_out)
        return out

# Temporal encoder for GNSS data (assuming the last 8 channels are GNSS data)
X_train_gnss = X_train_tensor[:, :, 3:]  # (batch_size, sequence_length, channels)
X_val_gnss = X_val_tensor[:, :, 3:]
X_test_gnss = X_test_tensor[:, :, 3:]

# Concatenate encoded IMU features with GNSS data
X_train_combined = torch.cat((imu_encoded_train.unsqueeze(1).repeat(1, X_train_gnss.size(1), 1), X_train_gnss), dim=2)
X_val_combined = torch.cat((imu_encoded_val.unsqueeze(1).repeat(1, X_val_gnss.size(1), 1), X_val_gnss), dim=2)
X_test_combined = torch.cat((imu_encoded_test.unsqueeze(1).repeat(1, X_test_gnss.size(1), 1), X_test_gnss), dim=2)

# Initialize and apply temporal encoder
temporal_encoder = TemporalEncoder(input_dim=X_train_combined.shape[-1])
temporal_encoded_train = temporal_encoder(X_train_combined)
temporal_encoded_val = temporal_encoder(X_val_combined)
temporal_encoded_test = temporal_encoder(X_test_combined)
```

```
class TransformerFusionModel(nn.Module):
    def __init__(self, input_dim=256, output_dim=3):
        super(TransformerFusionModel, self).__init__()
        self.transformer = nn.Transformer(d_model=input_dim, nhead=4, num_encoder_layers=3, batch_first=True)
        self.fc = nn.Linear(input_dim, output_dim)

    def forward(self, src):
        transformer_output = self.transformer(src, src)
        output = self.fc(transformer_output)
        return output

# Initialize and apply the Transformer model
fusion_model = TransformerFusionModel(input_dim=temporal_encoded_train.shape[-1])
train_output = fusion_model(temporal_encoded_train)
val_output = fusion_model(temporal_encoded_val)
test_output = fusion_model(temporal_encoded_test)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-bc54fa03e6a8> in <cell line: 1>()
----> 1 class TransformerFusionModel(nn.Module):
      2     def __init__(self, input_dim=256, output_dim=3):
      3         super(TransformerFusionModel, self).__init__()
      4         self.transformer = nn.Transformer(d_model=input_dim, nhead=4, num_encoder_layers=3, batch_first=True)
      5         self.fc = nn.Linear(input_dim, output_dim)

NameError: name 'nn' is not defined
```

```
class TransformerModel(nn.Module):
    def __init__(self, input_dim, output_dim, embed_dim, num_heads):
        super(TransformerModel, self).__init__()
        # Assuming you want embed_dim to be divisible by num_heads
        assert embed_dim % num_heads == 0, "embed_dim must be divisible by num_heads"

        self.embedding = nn.Embedding(input_dim, embed_dim)
        self.transformer = nn.Transformer(d_model=embed_dim, nhead=num_heads)
        self.fc = nn.Linear(embed_dim, output_dim)

    def forward(self, x):
        x = self.embedding(x)
        x = self.transformer(x)
        x = self.fc(x)
        return x

# Example values
input_dim = 11
output_dim = 3
embed_dim = 12  # Adjust this value
num_heads = 3  # Ensure num_heads divides embed_dim

# Initialize the model
model = TransformerModel(input_dim=input_dim, output_dim=output_dim, embed_dim=embed_dim, num_heads=num_heads)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-84b343362f4a> in <cell line: 1>()
----> 1 class TransformerModel(nn.Module):
      2     def __init__(self, input_dim, output_dim, embed_dim, num_heads):
      3         super(TransformerModel, self).__init__()
      4         # Assuming you want embed_dim to be divisible by num_heads
      5         assert embed_dim % num_heads == 0, "embed_dim must be divisible by num_heads"

NameError: name 'nn' is not defined
```

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```
import torch

# Convert X_train, X_val, and X_test to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)

y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)



# Training loop
epochs = 10
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    # Convert tgt to the same shape as X_train_tensor, initially filled with zeros
    tgt = torch.zeros_like(X_train_tensor)  # Assuming tgt is initially zeros

    # Forward pass
    outputs = model(X_train_tensor, tgt)

    # Compute loss
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()

    # Validation step
    model.eval()
    with torch.no_grad():
        tgt_val = torch.zeros_like(X_val_tensor)
        val_outputs = model(X_val_tensor, tgt_val)
        val_loss = criterion(val_outputs, y_val_tensor)

    print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {loss.item()}, Validation Loss: {val_loss.item()}')

print("Training complete.")
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-29-d4d9be057ee4> in <cell line: 3>()
      9
     10     # Forward pass
---> 11     outputs = model(X_train_tensor, tgt)
     12
     13     # Compute loss

                                 10 frames
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/rnn.py in check_input(self, input, batch_sizes)
    238                 f'input must have {expected_input_dim} dimensions, got {input.dim()}')
    239         if self.input_size != input.size(-1):
--> 240             raise RuntimeError(
    241                 f'input.size(-1) must be equal to input_size. Expected {self.input_size}, got
{input.size(-1)}')
    242

RuntimeError: input.size(-1) must be equal to input_size. Expected 256, got 264
```

```
class TransformerModel(nn.Module):
    def __init__(self):
        super(TransformerModel, self).__init__()
        self.transformer = nn.Transformer(d_model=256, nhead=4, num_encoder_layers=3, batch_first=True)
        self.fc = nn.Linear(124, 3)

    def forward(self, src, tgt):
        transformer_output = self.transformer(src, tgt)
        output = self.fc(transformer_output)
        return output

transformer_model = TransformerModel()

train_output = transformer_model(X_train_temporal, X_train_temporal)
val_output = transformer_model(X_val_temporal, X_val_temporal)
test_output = transformer_model(X_test_temporal, X_test_temporal)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-552408453438> in <cell line: 1>()
----> 1 class TransformerModel(nn.Module):
      2     def __init__(self):
      3         super(TransformerModel, self).__init__()
      4         self.transformer = nn.Transformer(d_model=256, nhead=4, num_encoder_layers=3, batch_first=True)
      5         self.fc = nn.Linear(124, 3)

NameError: name 'nn' is not defined
```

Generate print hello world using rot13 Close

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)
torch.manual_seed(42)

# Generate random data
X_train = np.random.rand(1000, 10)  # 1000 samples, 10 features
y_train = np.random.rand(1000, 1)   # 1000 samples, 1 target

X_val = np.random.rand(200, 10)  # 200 samples, 10 features
y_val = np.random.rand(200, 1)   # 200 samples, 1 target

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32)

# Define a simple feedforward neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Initialize the model, loss function, and optimizer
model = SimpleNN()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
epochs = 10
train_losses = []
val_losses = []

for epoch in range(epochs):
    # Training phase
```

Kalman Filter Implementation

```
    loss = criterion(outputs, y_train_tensor)
```

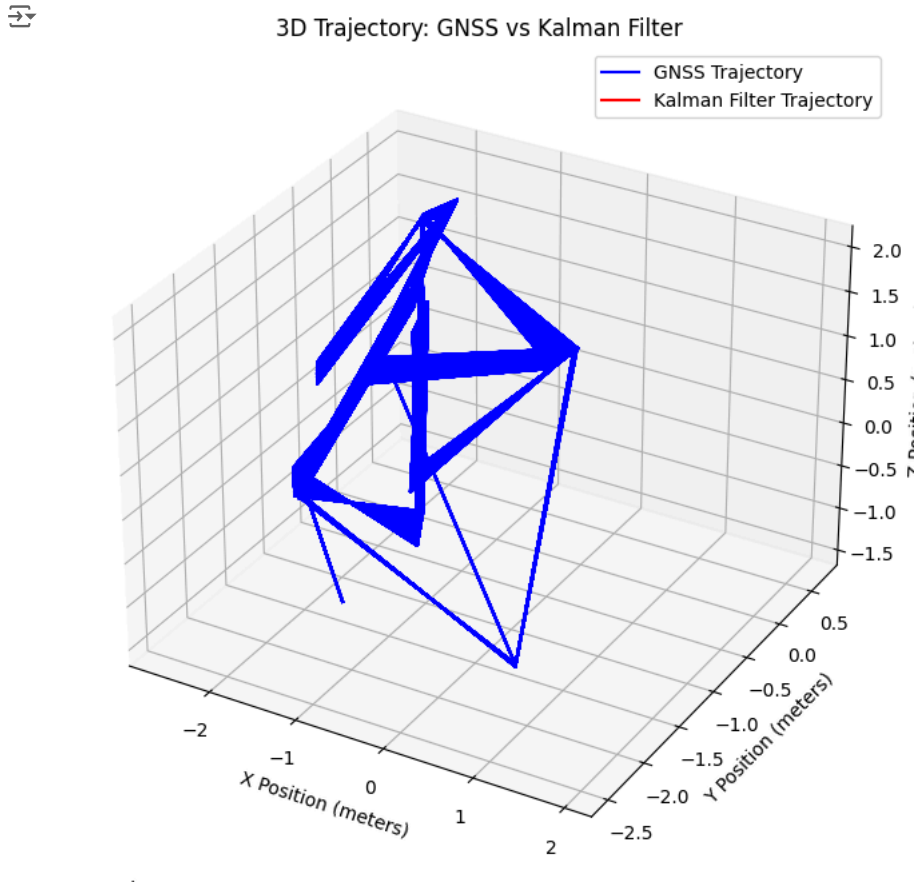
```
import numpy as np
import matplotlib.pyplot as plt

class KalmanFilter:
    def __init__(self, dt, u_x, u_y, u_z, std_acc, std_meas):
        self.dt = dt
        self.u = np.array([[u_x], [u_y], [u_z]])
        self.std_acc = std_acc
        self.std_meas = std_meas

        self.A = np.array([[1, 0, 0, self.dt, 0, 0],
                           [0, 1, 0, 0, self.dt, 0],
                           [0, 0, 1, 0, 0, self.dt],
                           [0, 0, 0, 1, 0, 0],
                           [0, 0, 0, 0, 1, 0],
                           [0, 0, 0, 0, 0, 1]])

        self.B = np.array([[(self.dt**2)/2, 0, 0],
                           [0, (self.dt**2)/2, 0],
                           [0, 0, (self.dt**2)/2],
                           [self.dt, 0, 0],
                           [0, self.dt, 0],
                           [0, 0, self.dt]])

        self.H = np.array([[1, 0, 0, 0, 0, 0],
                           [0, 1, 0, 0, 0, 0],
                           [0, 0, 1, 0, 0, 0]])

        self.P = np.eye(self.A.shape[1])
        self.Q = np.eye(self.A.shape[1]) * self.std_acc**2
        self.R = np.eye(self.H.shape[0]) * self.std_meas**2
        self.x = np.zeros((self.A.shape[1], 1))

    def predict(self):
        self.x = np.dot(self.A, self.x) + np.dot(self.B, self.u)
        self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q

    def update(self, z):
        y = z - np.dot(self.H, self.x)
        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R
        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))
        self.x = self.x + np.dot(K, y)
        I = np.eye(self.H.shape[1])
        self.P = np.dot((I - np.dot(K, self.H)), self.P)

    def get_state(self):
        return self.x.flatten()
```

Apllying Kalman Filters to GNSS data

```
# Assuming gnss_data has columns 'utcTimeMillis', 'SvPositionXEcefMeters', 'SvPositionYEcefMeters', 'SvPositionZEcefMeters'
# Initialize the Kalman Filter
dt = 1  # Assuming 1 second between measurements, adjust as necessary
u_x, u_y, u_z = 0, 0, 0  # No control input
std_acc = 0.2  # Process noise standard deviation (adjust based on data)
std_meas = 20.0  # Measurement noise standard deviation (adjust based on data)

kf = KalmanFilter(dt, u_x, u_y, u_z, std_acc, std_meas)

# Applying Kalman Filter to the GNSS data
positions = []
for i in range(len(gnss_data)):
    z = np.array([[gnss_data['SvPositionXEcefMeters'][i]],
                  [gnss_data['SvPositionYEcefMeters'][i]],
                  [gnss_data['SvPositionZEcefMeters'][i]]])
    kf.predict()
    kf.update(z)
    positions.append(kf.get_state())

# Convert to NumPy array for easier plotting
positions = np.array(positions)
```

Visualization of Kalman Filter Output

```
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Ground Truth Trajectory (GNSS Data)
ax.plot(gnss_data['SvPositionXEcefMeters'],
        gnss_data['SvPositionYEcefMeters'],
        gnss_data['SvPositionZEcefMeters'], label='GNSS Trajectory', color='blue')

# Kalman Filter Trajectory
ax.plot(positions[:, 0], positions[:, 1], positions[:, 2], label='Kalman Filter Trajectory', color='red')

ax.set_xlabel('X Position (meters)')
ax.set_ylabel('Y Position (meters)')
ax.set_zlabel('Z Position (meters)')
ax.set_title('3D Trajectory: GNSS vs Kalman Filter')
ax.legend()
plt.show()
```



Position Comparision

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class KalmanFilter:
    def __init__(self, dt, u, std_acc, std_meas):
        # Time step
        self.dt = dt

        # Control input model
        self.u = u

        # Initial state estimate
        self.x = np.zeros((4, 1))

        # State transition matrix
        self.A = np.array([[1, 0, self.dt, 0],
                           [0, 1, 0, self.dt],
                           [0, 0, 1, 0],
                           [0, 0, 0, 1]])

        # Control input matrix
        self.B = np.array([[0.5 * self.dt ** 2],
                           [0.5 * self.dt ** 2],
                           [self.dt],
                           [self.dt]])

        # Measurement model
        self.H = np.array([[1, 0, 0, 0],
                           [0, 1, 0, 0],
                           [0, 0, 1, 0]])

        # Process noise covariance
        self.Q = np.array([[std_acc ** 2, 0, 0, 0],
                           [0, std_acc ** 2, 0, 0],
                           [0, 0, std_acc ** 2, 0],
                           [0, 0, 0, std_acc ** 2]])

        # Measurement noise covariance
        self.R = np.array([[std_meas ** 2, 0, 0],
                           [0, std_meas ** 2, 0],
                           [0, 0, std_meas ** 2]])

        # Error covariance matrix
        self.P = np.eye(4)

    def predict(self):
        # State prediction
        self.x = np.dot(self.A, self.x) + np.dot(self.B, self.u)

        # Covariance prediction
        self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q

    def update(self, z):
        # Kalman gain
        S = np.dot(np.dot(self.H, self.P), self.H.T) + self.R
        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))

        # State update
        self.x = self.x + np.dot(K, (z - np.dot(self.H, self.x)))

        # Covariance update
        I = np.eye(4)
        self.P = np.dot((I - np.dot(K, self.H)), self.P)

    def get_state(self):
        return self.x
```

```
# Load GNSS data (assuming you have it loaded in a DataFrame)
gnss_data = pd.read_csv('/content/device_gnss 2.csv')

# Remove rows with NaN values for simplicity
gnss_data_cleaned = gnss_data.dropna(subset=['SvPositionXEcefMeters', 'SvPositionYEcefMeters', 'SvPositionZEcefMeters'])

# Debug: Check for missing data after cleaning
print("Remaining missing values:\n", gnss_data_cleaned.isna().sum())

# Set the time step and measurement uncertainty (these are example values)
dt = 1.0
std_acc = 1.0
std_meas = 5.0

# Initialize the Kalman Filter
kf = KalmanFilter(dt=dt, u=0, std_acc=std_acc, std_meas=std_meas)
```

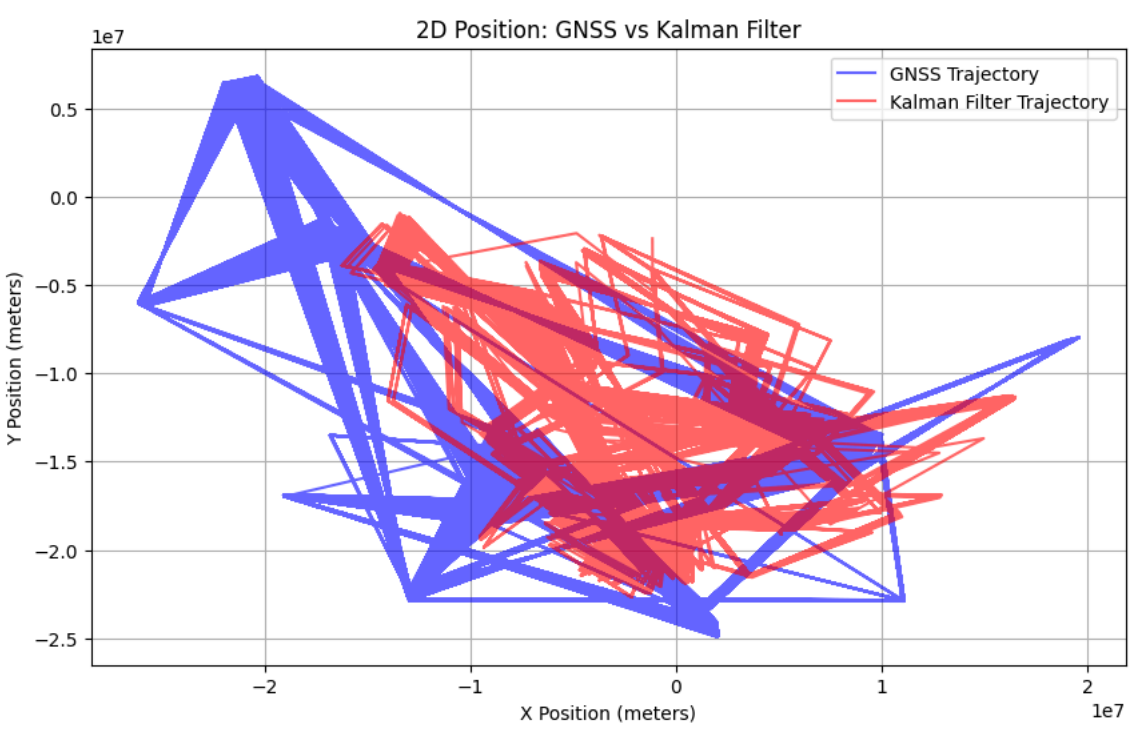
```
Remaining missing values:
 MessageType                                     0
utcTimeMillis                                    0
TimeNanos                                        0
LeapSecond                                   13127
FullBiasNanos                                    0
BiasNanos                                        0
BiasUncertaintyNanos                             0
DriftNanosPerSecond                              0
DriftUncertaintyNanosPerSecond                   0
HardwareClockDiscontinuityCount                  0
Svid                                             0
TimeOffsetNanos                                  0
State                                            0
ReceivedSvTimeNanos                              0
ReceivedSvTimeUncertaintyNanos                   0
Cn0DbHz                                          0
PseudorangeRateMetersPerSecond                   0
PseudorangeRateUncertaintyMetersPerSecond        0
AccumulatedDeltaRangeState                       0
AccumulatedDeltaRangeMeters                      0
AccumulatedDeltaRangeUncertaintyMeters           0
CarrierFrequencyHz                               0
MultipathIndicator                               0
ConstellationType                                0
CodeType                                         0
ChipsetElapsedRealtimeNanos                      0
ArrivalTimeNanosSinceGpsEpoch                    0
RawPseudorangeMeters                             0
RawPseudorangeUncertaintyMeters                  0
SignalType                                       0
ReceivedSvTimeNanosSinceGpsEpoch                 0
SvPositionXEcefMeters                            0
SvPositionYEcefMeters                            0
SvPositionZEcefMeters                            0
SvElevationDegrees                               0
SvAzimuthDegrees                                 0
SvVelocityXEcefMetersPerSecond                   0
SvVelocityYEcefMetersPerSecond                   0
SvVelocityZEcefMetersPerSecond                   0
SvClockBiasMeters                                0
SvClockDriftMetersPerSecond                      0
IsrbMeters                                       0
IonosphericDelayMeters                           0
TroposphericDelayMeters                          0
WlsPositionXEcefMeters                           0
WlsPositionYEcefMeters                           0
WlsPositionZEcefMeters                           0
dtype: int64
```

```
positions = []

for i in range(len(gnss_data_cleaned)):
    z = np.array([[gnss_data_cleaned['SvPositionXEcefMeters'].iloc[i]],
                  [gnss_data_cleaned['SvPositionYEcefMeters'].iloc[i]],
                  [gnss_data_cleaned['SvPositionZEcefMeters'].iloc[i]]])
    kf.predict()
    kf.update(z)
    positions.append(kf.get_state().flatten()[:2])  # Only store X and Y positions

# Convert positions to a numpy array
positions = np.array(positions)

# Debug: Ensure the positions array is populated
if len(positions) == 0:
    print("Error: The positions array is empty. Please check the Kalman Filter implementation.")
else:
    print("Kalman Filter output (first 5 positions):")
```

```
print(positions[:5])
```

```
Kalman Filter output (first 5 positions):
 [[ -1138094.09288817  -2375351.09749027]
  [ -1122955.43644148  -5359811.03217854]
  [ -4828287.19088794  -2077014.39321477]
  [-12460091.97047768  -3725125.93160223]
  [-12978042.93270141  -2840531.68229273]]
```

```
plt.figure(figsize=(10, 6))

# Plot GNSS trajectory
plt.plot(gnss_data_cleaned['SvPositionXEcefMeters'], gnss_data_cleaned['SvPositionYEcefMeters'], label='GNSS Trajectory', c

# Plot Kalman Filter trajectory
plt.plot(positions[:, 0], positions[:, 1], label='Kalman Filter Trajectory', color='red', alpha=0.6)

plt.xlabel('X Position (meters)')
plt.ylabel('Y Position (meters)')
plt.title('2D Position: GNSS vs Kalman Filter')
plt.legend()
plt.grid(True)
plt.show()
```



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class KalmanFilter:
    def __init__(self, dt, u, std_acc, std_meas):
        # Time step
        self.dt = dt

        # Control input model
        self.u = u

        # Initial state estimate
        self.x = np.zeros((6, 1))  # [x, y, z, vx, vy, vz]

        # State transition matrix
        self.A = np.array([[1, 0, 0, self.dt, 0, 0],
                           [0, 1, 0, 0, self.dt, 0],
                           [0, 0, 1, 0, 0, self.dt],
                           [0, 0, 0, 1, 0, 0],
                           [0, 0, 0, 0, 1, 0],
                           [0, 0, 0, 0, 0, 1]])

        # Control input matrix
        self.B = np.array([[0.5 * self.dt ** 2],
                           [0.5 * self.dt ** 2],
                           [0.5 * self.dt ** 2],
                           [self.dt],
                           [self.dt],
                           [self.dt]])

        # Measurement model
        self.H = np.array([[1, 0, 0, 0, 0, 0],
                           [0, 1, 0, 0, 0, 0],
                           [0, 0, 1, 0, 0, 0]])

        # Process noise covariance
        self.Q = np.array([[std_acc ** 2, 0, 0, 0, 0, 0],
                           [0, std_acc ** 2, 0, 0, 0, 0],
                           [0, 0, std_acc ** 2, 0, 0, 0],
                           [0, 0, 0, std_acc ** 2, 0, 0],
                           [0, 0, 0, 0, std_acc ** 2, 0],
                           [0, 0, 0, 0, 0, std_acc ** 2]])

        # Measurement noise covariance
        self.R = np.array([[std_meas ** 2, 0, 0],
                           [0, std_meas ** 2, 0],
                           [0, 0, std_meas ** 2]])

        # Error covariance matrix
        self.P = np.eye(6)

    def predict(self):
        # State prediction
        self.x = np.dot(self.A, self.x) + np.dot(self.B, self.u)

        # Covariance prediction
        self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q

    def update(self, z):
        # Kalman gain
        S = np.dot(np.dot(self.H, self.P), self.H.T) + self.R
        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))

        # State update
        self.x = self.x + np.dot(K, (z - np.dot(self.H, self.x)))

        # Covariance update
        I = np.eye(6)
        self.P = np.dot((I - np.dot(K, self.H)), self.P)

    def get_state(self):
        return self.x
```