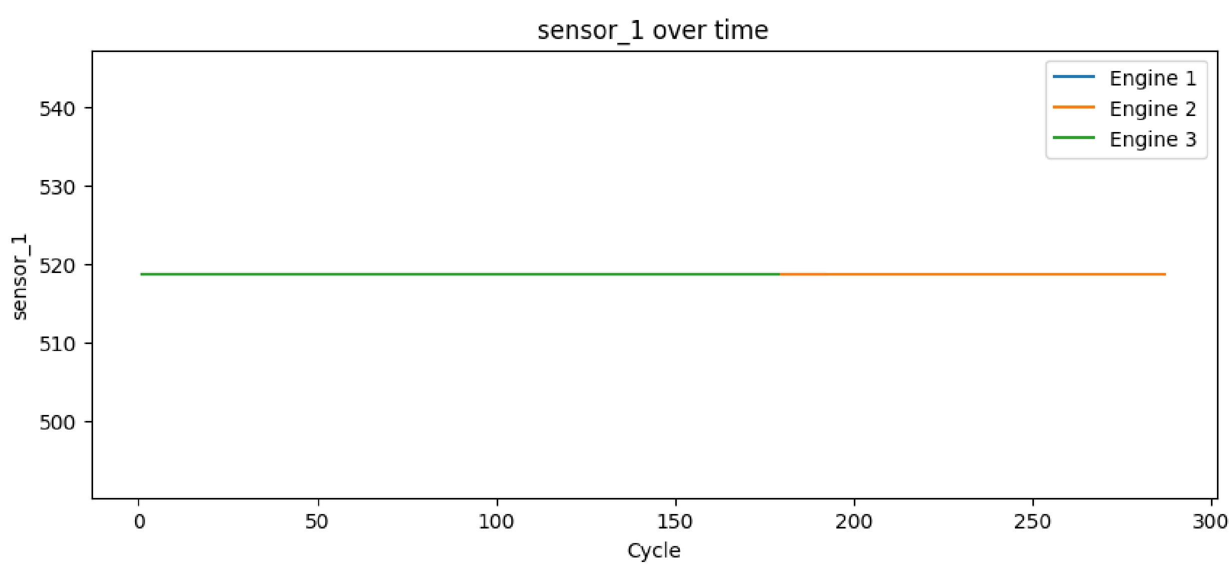
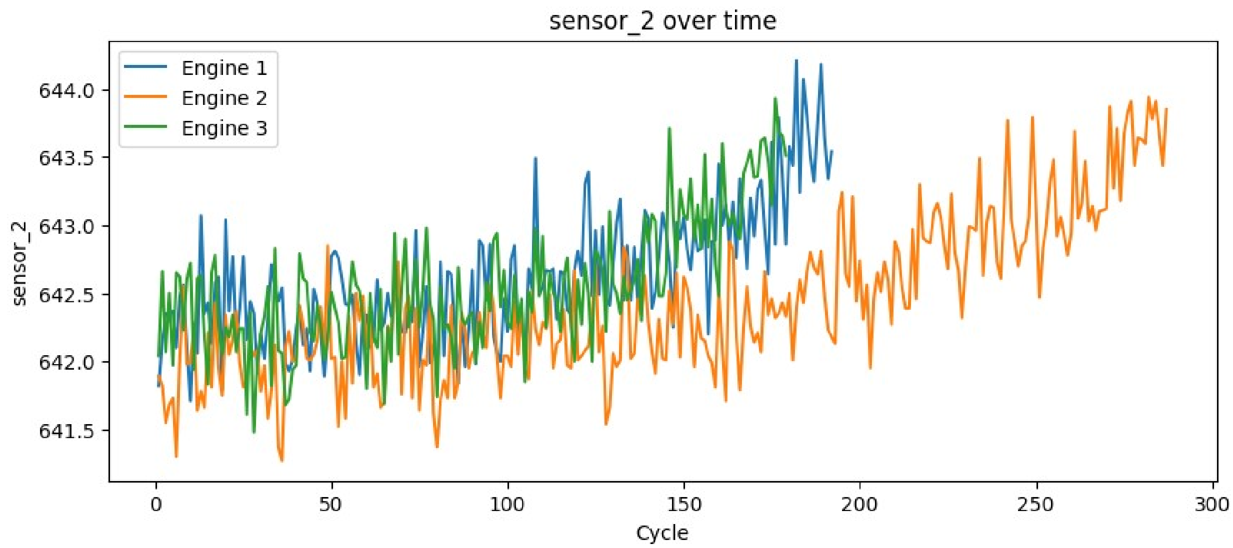
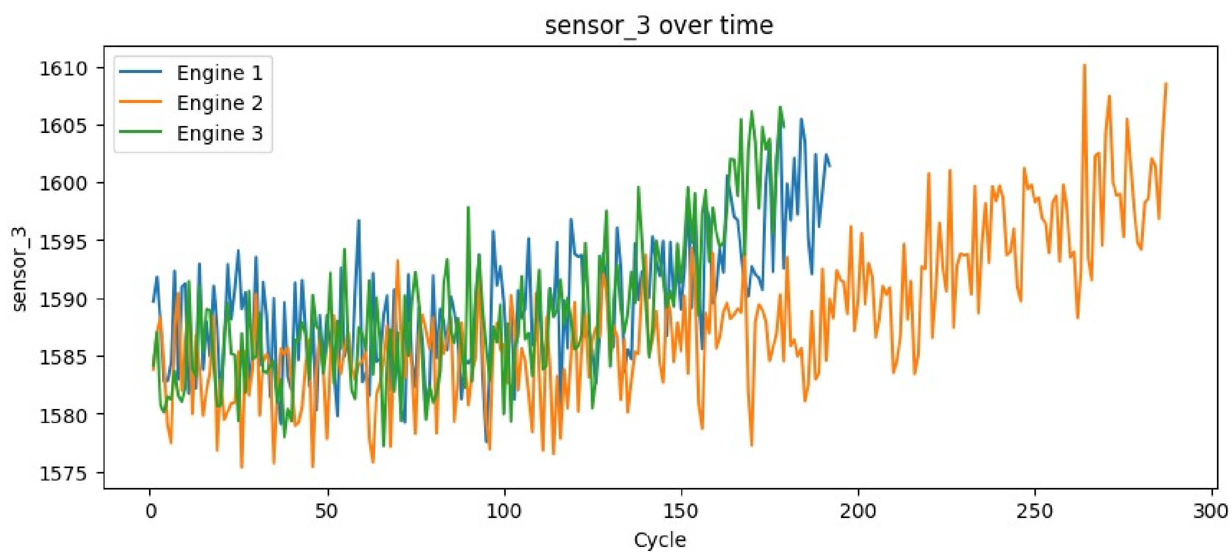
```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load FD001 dataset

df = pd.read_csv("D:\\dataset\\train_FD001.txt", sep=" ", header=None)
df.drop(columns=[26, 27], inplace=True)


# Assign column names
cols = ['unit_number', 'time_in_cycles'] + [f'op_setting_{i}' for i in
range(1, 4)] + [f'sensor_{i}' for i in range(1, 22)]
df.columns = cols

print(df['sensor_1'].describe())
print(df['sensor_1'].nunique())
print(df['sensor_2'].describe())
print(df['sensor_2'].nunique())
print(df['sensor_3'].describe())
print(df['sensor_3'].nunique())
print(df['sensor_4'].describe())
print(df['sensor_4'].nunique())

# Plot sensor degradation for a few engines
sample_units = df['unit_number'].unique()[:3]
for sensor in ['sensor_1','sensor_2', 'sensor_3', 'sensor_4']:
    plt.figure(figsize=(10, 4))
    for unit in sample_units:
        subset = df[df['unit_number'] == unit]
        plt.plot(subset['time_in_cycles'], subset[sensor],
label=f'Engine {unit}')
    plt.title(f'{sensor} over time')
    plt.xlabel('Cycle')
    plt.ylabel(sensor)
    plt.legend()
    plt.show()
```

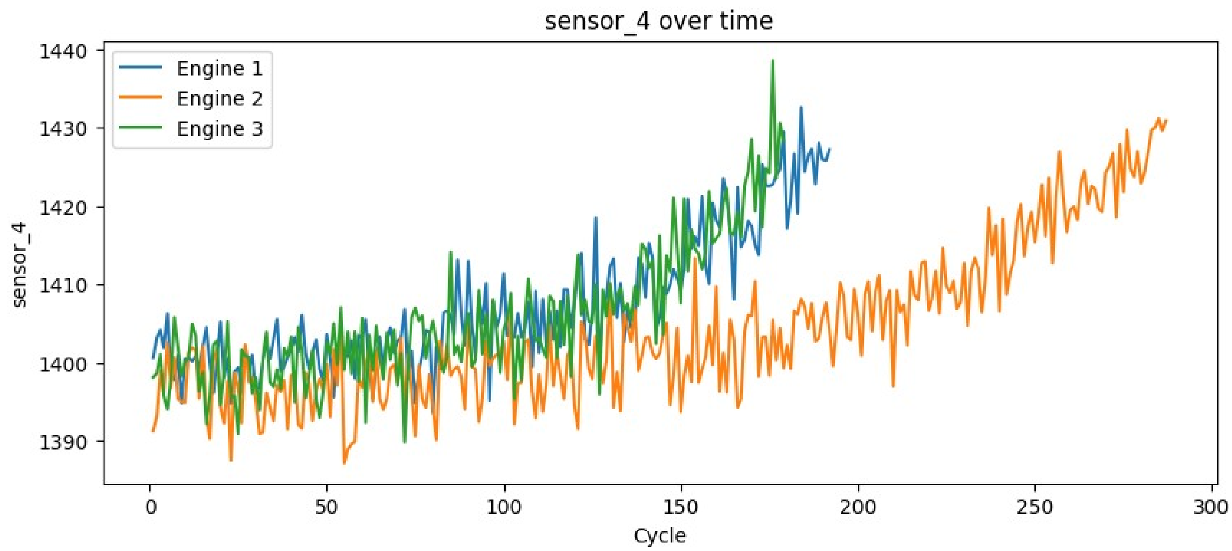
```
count    20631.00
mean       518.67
std          0.00
min        518.67
25%        518.67
50%        518.67
75%        518.67
max        518.67
Name: sensor_1, dtype: float64
1
count    20631.000000
mean       642.680934
```

```
std          0.500053
min        641.210000
25%        642.325000
50%        642.640000
75%        643.000000
max        644.530000
Name: sensor_2, dtype: float64
310
count    20631.000000
mean      1590.523119
std          6.131150
min       1571.040000
25%       1586.260000
50%       1590.100000
75%       1594.380000
max       1616.910000
Name: sensor_3, dtype: float64
3012
count    20631.000000
mean      1408.933782
std          9.000605
min       1382.250000
25%       1402.360000
50%       1408.040000
75%       1414.555000
max       1441.490000
Name: sensor_4, dtype: float64
4051
```




sensor_2 over time
Engine 1
Engine 2
Engine 3
644.0
643.5
643.0
642.5
642.0
641.5
sensor_2
0
50
100
150
200
250
300
Cycle


sensor_3 over time
Engine 1
Engine 2
Engine 3
1610
1605
1600
1595
1590
1585
1580
1575
sensor_3
0
50
100
150
200
250
300
Cycle




```
# Collect rolling features in a separate dictionary
rolling_features = {}

# Define sensor columns if not already done
sensor_cols = [col for col in df.columns if 'sensor_' in col]
window_size = 20  # or any value you prefer
for col in sensor_cols:
    grouped = df.groupby('unit_number')[col]
    rolling_features[f'{col}_mean'] = grouped.transform(lambda x:
x.rolling(window=window_size, min_periods=1).mean())
    rolling_features[f'{col}_std'] = grouped.transform(lambda x:
x.rolling(window=window_size, min_periods=1).std())
    rolling_features[f'{col}_slope'] = grouped.transform(lambda x:
x.diff().rolling(window=window_size, min_periods=1).mean())

# Concatenate all at once
df = pd.concat([df, pd.DataFrame(rolling_features)], axis=1)

# Optional: defragment the DataFrame
df = df.copy()
print("ok")
```

```
ok
```

```
# Define sensor columns and window size
sensor_cols = [col for col in df.columns if 'sensor_' in col and
'_mean' not in col and '_std' not in col and '_slope' not in col]
window_size = 20

# Create a list to collect rolling feature DataFrames
rolling_frames = []

for stat in ['mean', 'std', 'slope']:
```

```
    temp = pd.DataFrame(index=df.index)
    for col in sensor_cols:
        grouped = df.groupby('unit_number')[col]
        if stat == 'mean':
            temp[f'{col}_mean'] = grouped.transform(lambda x:
x.rolling(window=window_size, min_periods=1).mean())
        elif stat == 'std':
            temp[f'{col}_std'] = grouped.transform(lambda x:
x.rolling(window=window_size, min_periods=1).std())
        elif stat == 'slope':
            temp[f'{col}_slope'] = grouped.transform(lambda x:
x.diff().rolling(window=window_size, min_periods=1).mean())
    rolling_frames.append(temp)

# Concatenate all rolling features at once
rolling_df = pd.concat(rolling_frames, axis=1)

# Merge with original df and defragment
df = pd.concat([df, rolling_df], axis=1).copy()

print("Rolling features added successfully without column conflicts.")
```

```
Rolling features added successfully without column conflicts.
```

```
# Recalculate RUL if missing
if 'RUL' not in df.columns:
    rul_df = df.groupby('unit_number')
['time_in_cycles'].max().reset_index()
    rul_df.columns = ['unit_number', 'max_cycle']
    df = df.merge(rul_df, on='unit_number')
    df['RUL'] = df['max_cycle'] - df['time_in_cycles']
    df.drop(columns=['max_cycle'], inplace=True)
from sklearn.model_selection import train_test_split

# Drop non-feature columns
feature_cols = [col for col in df.columns if col not in
['unit_number', 'time_in_cycles', 'RUL']]
X = df[feature_cols]
y = df['RUL']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
print("done")
```

```
done
```

```
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np
import pandas as pd
```

```
# Step 1: Drop non-numeric columns and ensure float32 dtype
X_train_clean =
X_train.select_dtypes(include=[np.number]).astype(np.float32).copy()
X_test_clean =
X_test.select_dtypes(include=[np.number]).astype(np.float32).copy()

# Step 2: Robust Check for all non-finite values (NaN, Inf)
# ... (Leaving the previously corrected Step 2 logic intact, as it was
working to identify bad columns) ...
bad_cols = []
for col in X_train_clean.columns:
    col_values = X_train_clean[col].values
    is_clean = np.isfinite(col_values).all()

    if not is_clean:
        print(f"⚠ Dropping column: {col} (Contains NaNs or
Infinites)")
        bad_cols.append(col)

# Step 3: Drop problematic columns
X_train_clean.drop(columns=bad_cols, inplace=True)
X_test_clean.drop(columns=bad_cols, inplace=True)

# Step 4: Train XGBoost model
# Fill any remaining NaNs with 0
X_train_clean.fillna(0, inplace=True)
X_test_clean.fillna(0, inplace=True)

# □ FIX: Convert X and y to their underlying NumPy array (.values)
# This bypasses any unexpected pandas indexing behavior deep inside
XGBoost.
X_train_data = X_train_clean.values
X_test_data = X_test_clean.values

# Ensure y_train is also a simple array
# Assuming y_train is a pandas Series/DataFrame, use .values
if isinstance(y_train, (pd.Series, pd.DataFrame)):
    y_train_data = y_train.values
else:
    y_train_data = y_train # If y_train is already a NumPy array

model = XGBRegressor(n_estimators=100, max_depth=5, learning_rate=0.1,
random_state=42)

# Pass the NumPy arrays to the model
model.fit(X_train_data, y_train_data)

# Step 5: Predict and evaluate
# Use the NumPy array for prediction
```

```python
y_pred = model.predict(X_test_data)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

# Ensure y_test is also a NumPy array for error calculation
if isinstance(y_test, (pd.Series, pd.DataFrame)):
    y_test_data = y_test.values
else:
    y_test_data = y_test

mae = mean_absolute_error(y_test_data, y_pred)

print(f"□ RMSE: {rmse:.2f}, MAE: {mae:.2f}")
```

```
⚠ Dropping column: sensor_1_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_1_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_1_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_1_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_2_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_2_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_2_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_2_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_3_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_3_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_3_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_3_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_4_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_4_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_4_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_4_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_5_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_5_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_5_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_5_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_6_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_6_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_6_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_6_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_7_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_7_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_7_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_7_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_8_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_8_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_8_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_8_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_9_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_9_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_9_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_9_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_10_std (Contains NaNs or Infinites)
```

```
⚠ Dropping column: sensor_10_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_10_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_10_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_11_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_11_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_11_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_11_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_12_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_12_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_12_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_12_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_13_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_13_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_13_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_13_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_14_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_14_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_14_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_14_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_15_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_15_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_15_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_15_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_16_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_16_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_16_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_16_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_17_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_17_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_17_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_17_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_18_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_18_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_18_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_18_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_19_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_19_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_19_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_19_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_20_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_20_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_20_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_20_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_21_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_21_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_21_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_21_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_1_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_1_std (Contains NaNs or Infinites)
```

```
⚠ Dropping column: sensor_2_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_2_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_3_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_3_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_4_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_4_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_5_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_5_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_6_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_6_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_7_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_7_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_8_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_8_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_9_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_9_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_10_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_10_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_11_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_11_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_12_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_12_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_13_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_13_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_14_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_14_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_15_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_15_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_16_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_16_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_17_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_17_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_18_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_18_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_19_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_19_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_20_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_20_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_21_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_21_std (Contains NaNs or Infinites)
⚠ Dropping column: sensor_1_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_1_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_2_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_2_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_3_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_3_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_4_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_4_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_5_slope (Contains NaNs or Infinites)
```

```
⚠ Dropping column: sensor_5_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_6_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_6_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_7_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_7_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_8_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_8_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_9_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_9_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_10_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_10_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_11_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_11_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_12_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_12_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_13_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_13_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_14_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_14_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_15_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_15_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_16_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_16_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_17_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_17_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_18_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_18_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_19_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_19_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_20_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_20_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_21_slope (Contains NaNs or Infinites)
⚠ Dropping column: sensor_21_slope (Contains NaNs or Infinites)
□ RMSE: 31.43, MAE: 22.64
```

```python
import numpy as np

def create_lstm_sequences(df, sequence_length, feature_cols):
    sequences = []
    targets = []
    for unit in df['unit_number'].unique():
        unit_df = df[df['unit_number'] ==
unit].sort_values('time_in_cycles')
        features = unit_df[feature_cols].values
        rul = unit_df['RUL'].values
        for i in range(len(unit_df) - sequence_length + 1):
            seq_x = features[i:i+sequence_length]
            seq_y = rul[i+sequence_length-1]  # predict RUL at end of
window
            sequences.append(seq_x)
```

```
            targets.append(seq_y)
    return np.array(sequences), np.array(targets)

# Define your sequence length and feature columns
sequence_length = 30
feature_cols = [col for col in df.columns if col not in
['unit_number', 'time_in_cycles', 'RUL']]

X_seq, y_seq = create_lstm_sequences(df, sequence_length,
feature_cols)
print(f"✅ LSTM input shape: {X_seq.shape}, Target shape:
{y_seq.shape}")
```

```
✅ LSTM input shape: (17731, 30, 276), Target shape: (17731,)
```

```
#Step 1: Define the LSTM Mode
import torch
import torch.nn as nn

class RULPredictorLSTM(nn.Module):
    def __init__(self, input_size, hidden_size=64, num_layers=2,
dropout=0.3):
        super(RULPredictorLSTM, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True, dropout=dropout)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = out[:, -1, :]  # take output from last timestep
        return self.fc(out).squeeze()
print("done")
```

```
done
```

```
# Step 2: Prepare Data for PyTorc
from torch.utils.data import TensorDataset, DataLoader

# Convert to tensors
X_tensor = torch.tensor(X_seq, dtype=torch.float32)
y_tensor = torch.tensor(y_seq, dtype=torch.float32)

# Split into train/test
split = int(0.8 * len(X_tensor))
X_train_torch, X_test_torch = X_tensor[:split], X_tensor[split:]
y_train_torch, y_test_torch = y_tensor[:split], y_tensor[split:]

# Create DataLoaders
train_loader = DataLoader(TensorDataset(X_train_torch, y_train_torch),
batch_size=64, shuffle=True)
test_loader = DataLoader(TensorDataset(X_test_torch, y_test_torch),
```

```
batch_size=64)
print("Done")
```

```
Done
```

```
#Step 3: Train the Mode
model = RULPredictorLSTM(input_size=X_seq.shape[2])
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Training loop
for epoch in range(10):
    model.train()
    total_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        output = model(X_batch)
        loss = criterion(output, y_batch)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1} - Loss: {total_loss:.2f}")
print("Done")
```

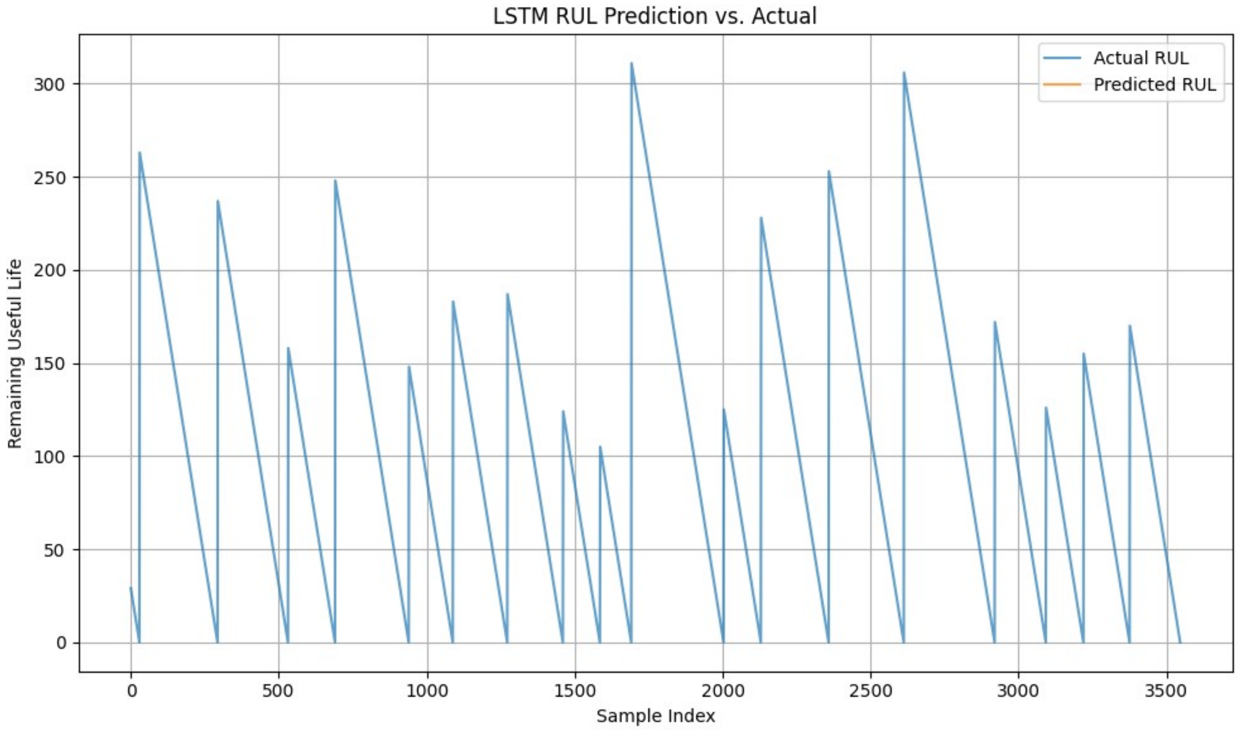
```
Epoch 1 - Loss: nan
Epoch 2 - Loss: nan
Epoch 3 - Loss: nan
Epoch 4 - Loss: nan
Epoch 5 - Loss: nan
Epoch 6 - Loss: nan
Epoch 7 - Loss: nan
Epoch 8 - Loss: nan
Epoch 9 - Loss: nan
Epoch 10 - Loss: nan
Done
```

```
#Evaluate
model.eval()
with torch.no_grad():
    predictions = model(X_test_torch)
    rmse = torch.sqrt(torch.mean((predictions - y_test_torch) **
2)).item()
    mae = torch.mean(torch.abs(predictions - y_test_torch)).item()
    print(f"□ LSTM RMSE: {rmse:.2f}, MAE: {mae:.2f}")
```

```
□ LSTM RMSE: nan, MAE: nan
```

```
#Step 5: Visualize Predictions vs. Actual RUL

import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(10, 6))
plt.plot(y_test_torch.numpy(), label='Actual RUL', alpha=0.7)
plt.plot(predictions.numpy(), label='Predicted RUL', alpha=0.7)
plt.title("LSTM RUL Prediction vs. Actual")
plt.xlabel("Sample Index")
plt.ylabel("Remaining Useful Life")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
#6 ssave the model for deployment
torch.save(model.state_dict(), "rul_lstm_model.pt")
print(" Model saved as rul_lstm_model.pt")
```

```
 Model saved as rul_lstm_model.pt
```

```
model.load_state_dict(torch.load("rul_lstm_model.pt"))
model.eval()
```

```
RULPredictorLSTM(
  (lstm): LSTM(276, 64, num_layers=2, batch_first=True, dropout=0.3)
  (fc): Linear(in_features=64, out_features=1, bias=True)
)
```

```
#1 Defining cnn lstm architecture
import torch.nn as nn
```

```
class CNNLSTM(nn.Module):
    def __init__(self, input_size, seq_len, num_filters=64,
kernel_size=3, lstm_hidden=64, lstm_layers=1, dropout=0.3):
        super(CNNLSTM, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=input_size,
out_channels=num_filters, kernel_size=kernel_size, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool1d(kernel_size=2)
        self.lstm = nn.LSTM(input_size=num_filters,
hidden_size=lstm_hidden, num_layers=lstm_layers, batch_first=True,
dropout=dropout)
        self.fc = nn.Linear(lstm_hidden, 1)

    def forward(self, x):
        # x: (batch, seq_len, features) → transpose for Conv1d
        x = x.transpose(1, 2)  # (batch, features, seq_len)
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)  # (batch, filters, reduced_seq_len)
        x = x.transpose(1, 2)  # back to (batch, reduced_seq_len,
filters)
        out, _ = self.lstm(x)
        out = out[:, -1, :]  # last timestep
        return self.fc(out).squeeze()
print("Done")
```

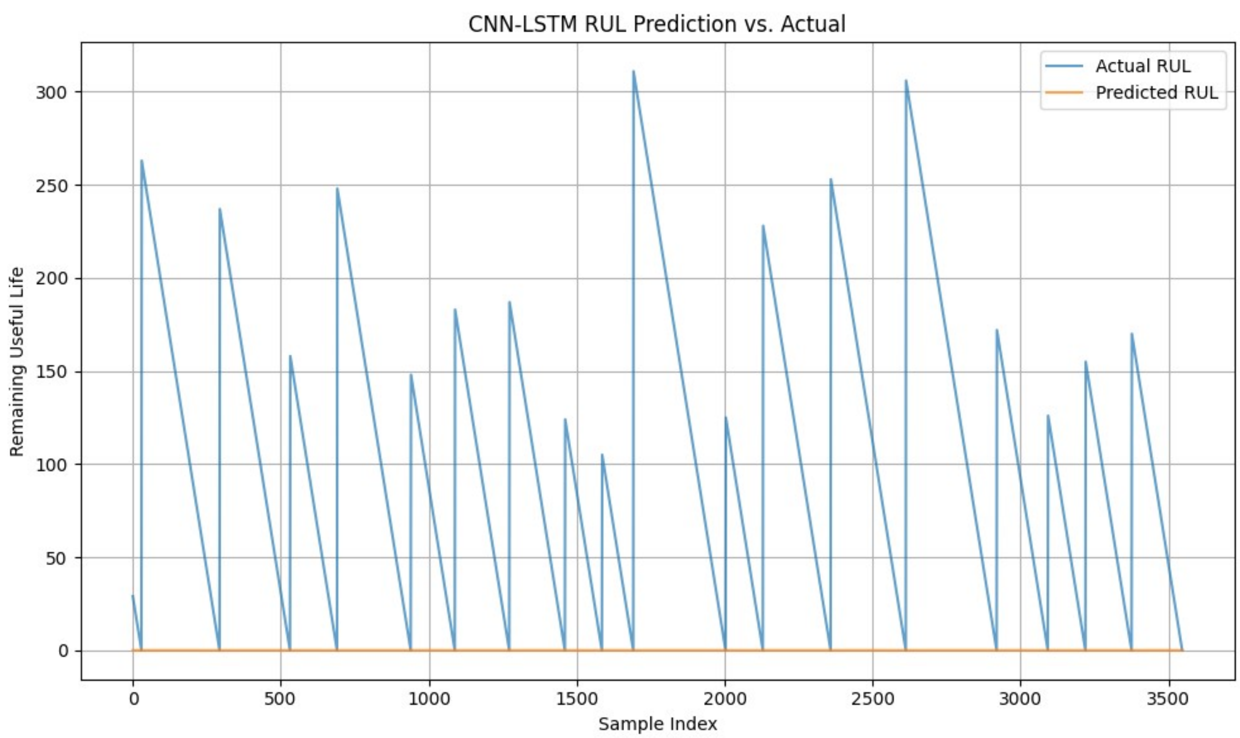
```
Done
```

```
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
import matplotlib.pyplot as plt

#  CNN-LSTM Model Definition
class CNNLSTM(nn.Module):
    def __init__(self, input_size, seq_len, num_filters=64,
kernel_size=3, lstm_hidden=64, lstm_layers=2, dropout=0.3):
        super(CNNLSTM, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=input_size,
out_channels=num_filters, kernel_size=kernel_size, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool1d(kernel_size=2)
        self.lstm = nn.LSTM(input_size=num_filters,
hidden_size=lstm_hidden, num_layers=lstm_layers, batch_first=True,
dropout=dropout)
        self.fc = nn.Sequential(
            nn.Linear(lstm_hidden, 1),
            nn.ReLU()  # Ensures RUL stays non-negative
        )
```

```
    def forward(self, x):
        x = x.transpose(1, 2)  # (batch, features, seq_len)
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = x.transpose(1, 2)  # (batch, reduced_seq_len, filters)
        out, _ = self.lstm(x)
        out = out[:, -1, :]
        return self.fc(out).squeeze()

#  Clean NaNs/Infs from training data
X_train_torch[torch.isnan(X_train_torch)] = 0
X_train_torch[torch.isinf(X_train_torch)] = 0
y_train_torch[torch.isnan(y_train_torch)] = 0
y_train_torch[torch.isinf(y_train_torch)] = 0

#  Initialize model, loss, optimizer
model = CNNLSTM(input_size=X_seq.shape[2], seq_len=X_seq.shape[1])
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

#  Training loop
for epoch in range(10):
    model.train()
    total_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        output = model(X_batch)
        loss = criterion(output, y_batch)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1} - Loss: {total_loss:.2f}")

print(" Training complete")

# Evaluation
model.eval()
with torch.no_grad():
    predictions = model(X_test_torch)
    rmse = torch.sqrt(torch.mean((predictions - y_test_torch) **
2)).item()
    mae = torch.mean(torch.abs(predictions - y_test_torch)).item()
    print(f"□ CNN-LSTM RMSE: {rmse:.2f}, MAE: {mae:.2f}")

#  Visualization
plt.figure(figsize=(10, 6))
plt.plot(y_test_torch.numpy(), label='Actual RUL', alpha=0.7)
plt.plot(predictions.numpy(), label='Predicted RUL', alpha=0.7)
```

```
plt.title("CNN-LSTM RUL Prediction vs. Actual")
plt.xlabel("Sample Index")
plt.ylabel("Remaining Useful Life")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Epoch 1 - Loss: 2608924.47
Epoch 2 - Loss: 2609678.79
Epoch 3 - Loss: 2607798.85
Epoch 4 - Loss: 2608992.93
Epoch 5 - Loss: 2609946.83
Epoch 6 - Loss: 2610481.34
Epoch 7 - Loss: 2609467.62
Epoch 8 - Loss: 2609613.21
Epoch 9 - Loss: 2608857.30
Epoch 10 - Loss: 2608562.93
 Training complete
□ CNN-LSTM RMSE: nan, MAE: nan
```

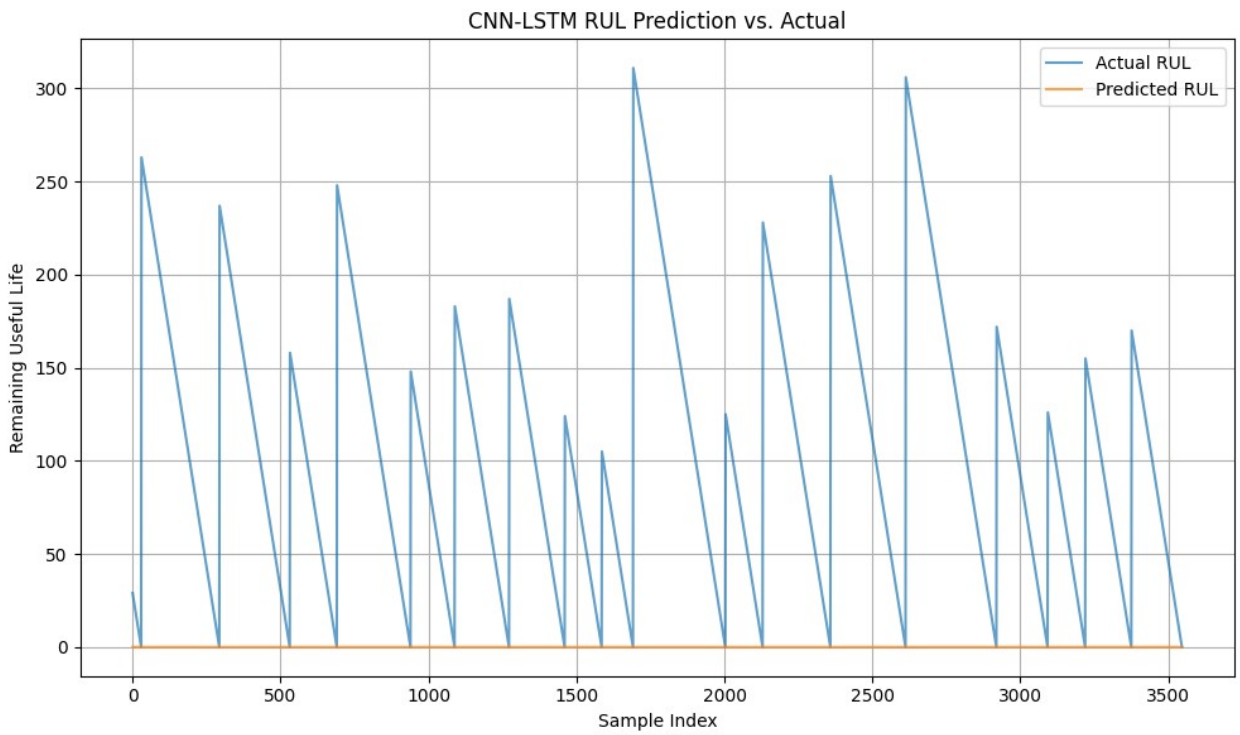



```
model.eval()
with torch.no_grad():
    predictions = model(X_test_torch)
    rmse = torch.sqrt(torch.mean((predictions - y_test_torch) **
2)).item()
```

```
    mae = torch.mean(torch.abs(predictions - y_test_torch)).item()
    print(f" CNN-LSTM RMSE: {rmse:.2f}, MAE: {mae:.2f}")
```

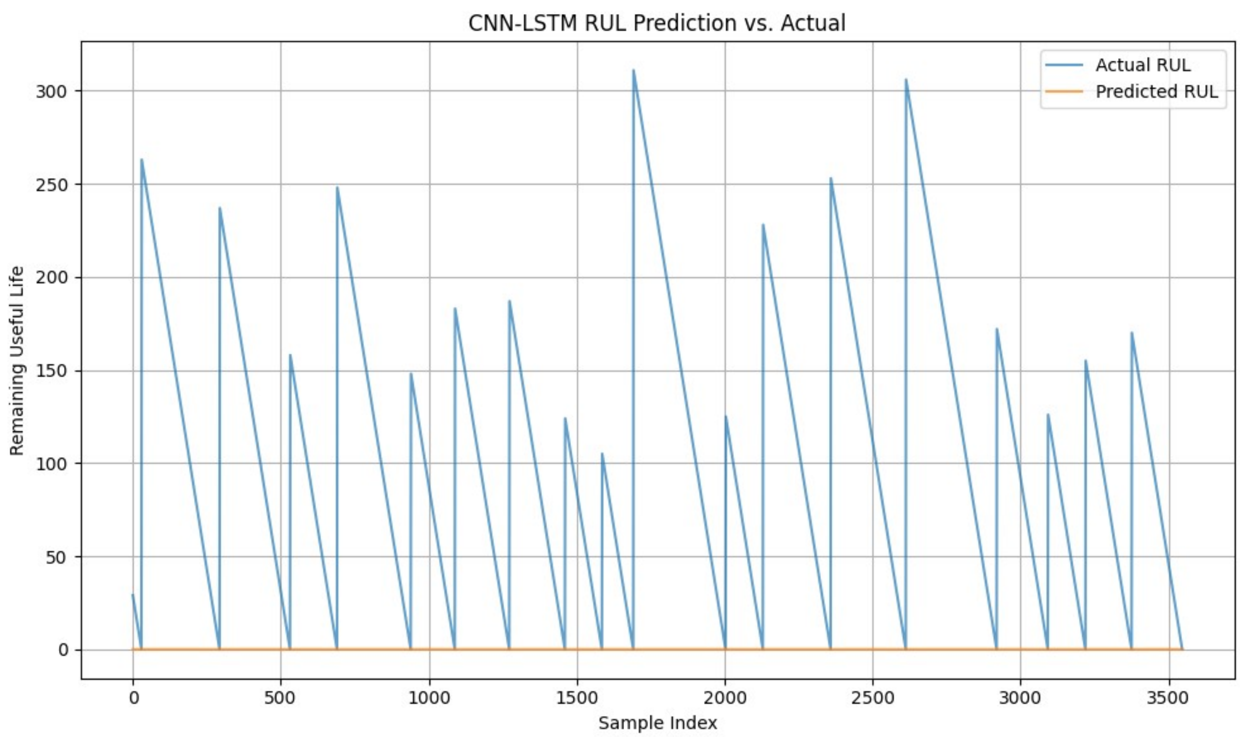
```
 CNN-LSTM RMSE: nan, MAE: nan
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(y_test_torch.numpy(), label='Actual RUL', alpha=0.7)
plt.plot(predictions.numpy(), label='Predicted RUL', alpha=0.7)
plt.title("CNN-LSTM RUL Prediction vs. Actual")
plt.xlabel("Sample Index")
plt.ylabel("Remaining Useful Life")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
#3 model evaluation
model.eval()
with torch.no_grad():
    predictions = model(X_test_torch)
    rmse = torch.sqrt(torch.mean((predictions - y_test_torch) **
2)).item()
    mae = torch.mean(torch.abs(predictions - y_test_torch)).item()
    print(f" CNN-LSTM RMSE: {rmse:.2f}, MAE: {mae:.2f}")
```

```
CNN-LSTM RMSE: nan, MAE: nan

#Visualize Predictions Plot actual vs. predicted RUL to see how well
it tracks degradation:

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(y_test_torch.numpy(), label='Actual RUL', alpha=0.7)
plt.plot(predictions.numpy(), label='Predicted RUL', alpha=0.7)
plt.title("CNN-LSTM RUL Prediction vs. Actual")
plt.xlabel("Sample Index")
plt.ylabel("Remaining Useful Life")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
torch.save(model.state_dict(), "cnn_lstm_rul_model.pt")
print(" CNN-LSTM model saved as cnn_lstm_rul_model.pt")

CNN-LSTM model saved as cnn_lstm_rul_model.pt

model.eval()
with torch.no_grad():
    predictions = model(X_test_torch)
    rmse = torch.sqrt(torch.mean((predictions - y_test_torch) **
```

```
2)).item()
    mae = torch.mean(torch.abs(predictions - y_test_torch)).item()
    print(f" CNN-LSTM RMSE: {rmse:.2f}, MAE: {mae:.2f}")
```

```
 CNN-LSTM RMSE: nan, MAE: nan
```

```
import pandas as pd

def load_cmapss_data(file_path):
    df = pd.read_csv(file_path, sep=" ", header=None)
    df.dropna(axis=1, how='all', inplace=True)
    df.columns = ['unit', 'cycle'] + [f'op_{i}' for i in range(1, 4)]
+ [f'sensor_{i}' for i in range(1, df.shape[1]-5)]
    return df

rul_df = df.groupby('unit_number')
['time_in_cycles'].max().reset_index()
rul_df.columns = ['unit_number', 'max_cycle']
df = df.merge(rul_df, on='unit_number')
df['RUL'] = df['max_cycle'] - df['time_in_cycles']
df.drop(columns=['max_cycle'], inplace=True)
print("Done")
```

```
Done
```

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df[sensor_cols] = scaler.fit_transform(df[sensor_cols])
print("Done")
```

```
Done
```

```
train_units = df['unit_number'].unique()[:80]   # first 80 engines
test_units = df['unit_number'].unique()[80:]   # remaining engines

train_df = df[df['unit_number'].isin(train_units)]
test_df = df[df['unit_number'].isin(test_units)]
print("Done")
```

```
Done
```

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

# Convert tensors to numpy
y_true = y_test_torch.numpy()
y_pred = predictions.numpy()

# Remove NaNs/Infs
mask = np.isfinite(y_true) & np.isfinite(y_pred)
y_true_clean = y_true[mask]
```

```
y_pred_clean = y_pred[mask]

# Compute metrics
rmse = np.sqrt(mean_squared_error(y_true_clean, y_pred_clean))
mae = mean_absolute_error(y_true_clean, y_pred_clean)

print(f" RMSE: {rmse:.2f}, MAE: {mae:.2f}")
```

```
 RMSE: 127.84, MAE: 105.74
```

```
df["RUL"] = df["RUL"].clip(upper=125)
```

```
criterion = nn.SmoothL1Loss()
```

```
# 1. Normalize features before sequence creation
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
df[sensor_cols] = scaler.fit_transform(df[sensor_cols])

# 2. Cap extreme RUL values
df['RUL'] = df['RUL'].clip(upper=125)

# 3. Training loop
model = RULPredictorLSTM(input_size=X_seq.shape[2])
criterion = nn.SmoothL1Loss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

for epoch in range(20):
    model.train()
    total_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        output = model(X_batch)
        loss = criterion(output, y_batch)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(),
max_norm=1.0)  # prevent exploding grads
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1} - Loss: {total_loss:.2f}")
```

```
Epoch 1 - Loss: 19773.77
Epoch 2 - Loss: 19011.07
Epoch 3 - Loss: 18603.20
Epoch 4 - Loss: 18274.56
Epoch 5 - Loss: 17971.85
Epoch 6 - Loss: 17678.02
Epoch 7 - Loss: 17397.20
Epoch 8 - Loss: 17129.35
Epoch 9 - Loss: 16874.25
```

```
Epoch 10 - Loss: 16609.14
Epoch 11 - Loss: 16364.14
Epoch 12 - Loss: 16115.28
Epoch 13 - Loss: 15870.59
Epoch 14 - Loss: 15628.92
Epoch 15 - Loss: 15396.87
Epoch 16 - Loss: 15172.09
Epoch 17 - Loss: 14951.71
Epoch 18 - Loss: 14739.21
Epoch 19 - Loss: 14528.41
Epoch 20 - Loss: 14324.04
```

```
from sklearn.preprocessing import MinMaxScaler

rul_scaler = MinMaxScaler()
y_train_scaled = rul_scaler.fit_transform(y_train_torch.reshape(-1,
1))
y_test_scaled = rul_scaler.transform(y_test_torch.reshape(-1, 1))

# convert back to torch
y_train_torch = torch.tensor(y_train_scaled,
dtype=torch.float32).squeeze()
y_test_torch = torch.tensor(y_test_scaled,
dtype=torch.float32).squeeze()

model.eval()
with torch.no_grad():
    sample_preds = model(X_train_torch[:5])
    print("Sample predictions:", sample_preds)
```

```
Sample predictions: tensor([74.8944, 74.8944, 74.8944, 74.8944,
74.8944])
```

```
for name, param in model.named_parameters():
    if param.grad is not None:
        print(f"{name} grad mean: {param.grad.mean().item():.6f}")
```

```
lstm.weight_ih_l0 grad mean: 0.000000
lstm.weight_hh_l0 grad mean: 0.000000
lstm.bias_ih_l0 grad mean: 0.000000
lstm.bias_hh_l0 grad mean: 0.000000
lstm.weight_ih_l1 grad mean: -0.000000
lstm.weight_hh_l1 grad mean: 0.000000
lstm.bias_ih_l1 grad mean: -0.000000
lstm.bias_hh_l1 grad mean: -0.000000
fc.weight grad mean: 0.011628
fc.bias grad mean: -0.124035
```

```
predictions_rescaled =
rul_scaler.inverse_transform(predictions.numpy().reshape(-
1,1)).flatten()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4,
weight_decay=1e-5)

best_rmse = float("inf")
patience, wait = 10, 0

for epoch in range(50):
    # ... training loop as before ...

    # Eval step
    model.eval()
    with torch.no_grad():
        predictions = model(X_test_torch)
        y_true = y_test_torch.numpy()
        y_pred = predictions.numpy()

        mask = np.isfinite(y_true) & np.isfinite(y_pred)
        y_true, y_pred = y_true[mask], y_pred[mask]

        rmse = np.sqrt(mean_squared_error(y_true, y_pred))
        mae = mean_absolute_error(y_true, y_pred)

    print(f"Epoch {epoch+1}/50 | Train Loss: {total_loss:.2f} | Test
RMSE: {rmse:.2f}, MAE: {mae:.2f}")

    # --- early stopping ---
    if rmse < best_rmse:
        best_rmse = rmse
        wait = 0
        torch.save(model.state_dict(), "best_lstm_model.pt")  # save
best model
    else:
        wait += 1
        if wait >= patience:
            print(f"□ Early stopping at epoch {epoch+1}, best RMSE:
{best_rmse:.2f}")
            break
```

```
Epoch 1/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 2/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 3/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 4/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 5/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 6/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 7/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 8/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 9/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 10/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
Epoch 11/50 | Train Loss: 10754.87 | Test RMSE: 74.05, MAE: 74.05
□ Early stopping at epoch 11, best RMSE: 74.05
```

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

# --- Model, Loss, Optimizer ---
model = RULPredictorLSTM(input_size=X_seq.shape[2])
criterion = nn.SmoothL1Loss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

# --- Training & Evaluation Loop ---
epochs = 50
for epoch in range(epochs):
    # Training
    model.train()
    total_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        output = model(X_batch)
        loss = criterion(output, y_batch)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(),
max_norm=1.0)
        optimizer.step()
        total_loss += loss.item()

    # Evaluation
    model.eval()
    with torch.no_grad():
        predictions = model(X_test_torch)
        y_true = y_test_torch.numpy()
        y_pred = predictions.numpy()

        # Clean NaNs/Infs
        mask = np.isfinite(y_true) & np.isfinite(y_pred)
        y_true = y_true[mask]
        y_pred = y_pred[mask]

        rmse = np.sqrt(mean_squared_error(y_true, y_pred))
        mae = mean_absolute_error(y_true, y_pred)

    print(f"Epoch {epoch+1}/{epochs} - Train Loss: {total_loss:.2f} |
RMSE: {rmse:.2f}, MAE: {mae:.2f}")
```

```
Epoch 1/50 - Train Loss: 19712.47 | RMSE: 3.58, MAE: 3.53
Epoch 2/50 - Train Loss: 18903.88 | RMSE: 5.98, MAE: 5.96
Epoch 3/50 - Train Loss: 18507.75 | RMSE: 7.67, MAE: 7.65
Epoch 4/50 - Train Loss: 18172.21 | RMSE: 9.32, MAE: 9.30
Epoch 5/50 - Train Loss: 17865.77 | RMSE: 10.86, MAE: 10.85
Epoch 6/50 - Train Loss: 17573.03 | RMSE: 12.36, MAE: 12.35
```

```
Epoch 7/50 - Train Loss: 17298.05 | RMSE: 13.84, MAE: 13.82
Epoch 8/50 - Train Loss: 17024.86 | RMSE: 15.30, MAE: 15.29
Epoch 9/50 - Train Loss: 16766.41 | RMSE: 16.76, MAE: 16.75
Epoch 10/50 - Train Loss: 16506.84 | RMSE: 18.22, MAE: 18.21
Epoch 11/50 - Train Loss: 16258.59 | RMSE: 19.67, MAE: 19.66
Epoch 12/50 - Train Loss: 16012.37 | RMSE: 21.12, MAE: 21.11
Epoch 13/50 - Train Loss: 15776.52 | RMSE: 22.57, MAE: 22.56
Epoch 14/50 - Train Loss: 15534.80 | RMSE: 24.02, MAE: 24.01
Epoch 15/50 - Train Loss: 15306.63 | RMSE: 25.46, MAE: 25.46
Epoch 16/50 - Train Loss: 15082.11 | RMSE: 26.91, MAE: 26.90
Epoch 17/50 - Train Loss: 14863.63 | RMSE: 28.35, MAE: 28.35
Epoch 18/50 - Train Loss: 14651.31 | RMSE: 29.80, MAE: 29.79
Epoch 19/50 - Train Loss: 14442.69 | RMSE: 31.24, MAE: 31.24
Epoch 20/50 - Train Loss: 14245.74 | RMSE: 32.69, MAE: 32.68
Epoch 21/50 - Train Loss: 14053.38 | RMSE: 34.13, MAE: 34.13
Epoch 22/50 - Train Loss: 13858.93 | RMSE: 35.58, MAE: 35.57
Epoch 23/50 - Train Loss: 13671.69 | RMSE: 37.02, MAE: 37.02
Epoch 24/50 - Train Loss: 13498.64 | RMSE: 38.46, MAE: 38.46
Epoch 25/50 - Train Loss: 13318.24 | RMSE: 39.91, MAE: 39.90
Epoch 26/50 - Train Loss: 13154.89 | RMSE: 41.35, MAE: 41.35
Epoch 27/50 - Train Loss: 12982.03 | RMSE: 42.79, MAE: 42.79
Epoch 28/50 - Train Loss: 12830.63 | RMSE: 44.24, MAE: 44.23
Epoch 29/50 - Train Loss: 12676.06 | RMSE: 45.68, MAE: 45.67
Epoch 30/50 - Train Loss: 12528.14 | RMSE: 47.12, MAE: 47.12
Epoch 31/50 - Train Loss: 12395.45 | RMSE: 48.56, MAE: 48.56
Epoch 32/50 - Train Loss: 12253.02 | RMSE: 50.01, MAE: 50.00
Epoch 33/50 - Train Loss: 12121.99 | RMSE: 51.44, MAE: 51.44
Epoch 34/50 - Train Loss: 11991.32 | RMSE: 52.88, MAE: 52.88
Epoch 35/50 - Train Loss: 11868.02 | RMSE: 54.31, MAE: 54.31
Epoch 36/50 - Train Loss: 11757.29 | RMSE: 55.75, MAE: 55.75
Epoch 37/50 - Train Loss: 11648.13 | RMSE: 57.18, MAE: 57.18
Epoch 38/50 - Train Loss: 11543.26 | RMSE: 58.60, MAE: 58.60
Epoch 39/50 - Train Loss: 11445.30 | RMSE: 60.01, MAE: 60.01
Epoch 40/50 - Train Loss: 11357.01 | RMSE: 61.37, MAE: 61.37
Epoch 41/50 - Train Loss: 11272.64 | RMSE: 62.77, MAE: 62.77
Epoch 42/50 - Train Loss: 11190.31 | RMSE: 64.12, MAE: 64.12
Epoch 43/50 - Train Loss: 11119.04 | RMSE: 65.49, MAE: 65.49
Epoch 44/50 - Train Loss: 11045.01 | RMSE: 66.82, MAE: 66.81
Epoch 45/50 - Train Loss: 10986.95 | RMSE: 68.10, MAE: 68.10
Epoch 46/50 - Train Loss: 10927.48 | RMSE: 69.41, MAE: 69.40
Epoch 47/50 - Train Loss: 10877.96 | RMSE: 70.67, MAE: 70.67
Epoch 48/50 - Train Loss: 10825.14 | RMSE: 71.85, MAE: 71.85
Epoch 49/50 - Train Loss: 10785.23 | RMSE: 72.99, MAE: 72.99
Epoch 50/50 - Train Loss: 10754.87 | RMSE: 74.05, MAE: 74.05
```