Report: Predictive Maintenance Using LSTM and GRU Models

Table of Contents

1. Introduction

This project is part of the predictive maintenance assignment focused on estimating the Remaining Useful Life (RUL) of aircraft engines using machine learning models, specifically LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) neural networks. The project uses NASA's C-MAPSS dataset, which contains run-to-failure data for engines under various conditions.

The objective is to predict the RUL of engines to improve maintenance schedules and avoid unexpected failures.

2. Data Acquisition and Preprocessing (20 points)

Data Collection and Cleaning

- **Description:** Data was acquired from NASA on their Jet Engine Simulated dataset. This notebook focused mainly on the FDOO1.

  Data Set: FD001
  Train trjectories: 100
  Test trajectories: 100
  Conditions: ONE (Sea Level)
  Fault Modes: ONE (HPC Degradation)

- **Cleaning Process:** Before any feature engineering, I performed some data cleaning. I noticed that when doing train.describe(), 7 attributes were constant and had no variability. I removed these features because I wanted to reduce any noise before training and building a model.

```
# Identify columns with constant values
constant_columns = [col for col in train_df.columns if train_df[col].nunique() == 1]

# Print the constant columns identified
print("Constant columns:", constant_columns)

# Drop the constant columns from the DataFrame
train_df_cleaned = train_df.drop(columns=constant_columns)
```

```
Constant columns: ['setting3', 'sensor1', 'sensor5', 'sensor10', 'sensor16', 'sensor18', 'sensor19']
```

- i Footnote

Feature Engineering

- **Techniques Used:** There were two techniques that I used here.
  - **Aggregating Sensor Data:** I did this to summarize sensors to get overall engine performance. That is because all of these engines aren't brand new so performance across engines will be different. So I thought creating a baseline would help when predicting engine failure.

```
# Define aggregation functions you want to apply
aggregation_functions = ['mean', 'max', 'min', 'std']

# Apply the aggregation functions for sensor columns
# Aggregation per engine ('id') for train data
aggregated_train_df = train_df.groupby('id').agg({col: aggregation_functions for col in train_df.filter(like='sensor').columns})
aggregated_train_df.columns = ['_'.join(col).strip() for col in aggregated_train_df.columns.values]

# Merge aggregated features back to the original train dataframe
train_df = train_df.merge(aggregated_train_df, on='id', how='left')

# Aggregation per engine ('id') for test data
aggregated_test_df = test_df.groupby('id').agg({col: aggregation_functions for col in test_df.filter(like='sensor').columns})
aggregated_test_df.columns = ['_'.join(col).strip() for col in aggregated_test_df.columns.values]

# Merge aggregated features back to the original test dataframe
test_df = test_df.merge(aggregated_test_df, on='id', how='left')
```

- [ii] Footnote
- **Lag Features:** I did this so that each cycle could understand the past performance of the sensor readings. That way this feature for the record would understand prior readings on current conditions.

```
# Define the number of lags you want to create
lag = 3  # Adjust based on your needs

# Create a list to store lagged DataFrames
lagged_features = []

# Generate lag features for each sensor column in train_df
for col in train_df.filter(like='sensor').columns:
    # Create lagged columns for the current sensor and append to the list
    for i in range(1, lag + 1):
        lagged_df = train_df[[col]].shift(i).rename(columns={col: f'{col}_lag_{i}'})
        lagged_features.append(lagged_df)

# Concatenate all lagged features along columns
lagged_train_df = pd.concat(lagged_features, axis=1)

# Merge lagged features back to the original train_df
train_df = pd.concat([train_df, lagged_train_df], axis=1)

# Fill missing values caused by lagging
train_df.fillna(0, inplace=True)

# Repeat the same process for test_df
lagged_features_test = []
for col in test_df.filter(like='sensor').columns:
    for i in range(1, lag + 1):
        lagged_df = test_df[[col]].shift(i).rename(columns={col: f'{col}_lag_{i}'})
        lagged_features_test.append(lagged_df)

# Concatenate all lagged features for test data
lagged_test_df = pd.concat(lagged_features_test, axis=1)

# Merge lagged features back to the original test_df
test_df = pd.concat([test_df, lagged_test_df], axis=1)

# Fill missing values caused by lagging in test data
test_df.fillna(0, inplace=True)
```

- [iii] Footnote

Data Cleaning

- **Findings:** After performing feature engineering, I noticed that there were a lot of correlated features. So I removed any correlated features that were greater than .95. This reduced the size of my data frame to 86 columns.

3. Model Building (30 points)

Hyperparameter Tuning

- **Approach:** Before building my model I did some hyperparameter tuning using keras tuner RandomSearch. I looked at the number of units in the layers and dropout rates to help prevent overfitting. To make this process more efficient due to limited computational resources, I implemented early stopping if the validation loss did not improve over 5 epochs. After getting the best parameters I had it print in my console.

LSTM & GRU Implementation

- **Model Architecture:** After hyperparameter tuning, I implemented only 50 epochs. I also did a validation split at .2. When building the model, I also implemented early stopping.
- **Training Process:** During the training process, early stopping was reached at 7 epochs, indicating that validation loss was not improving anymore.

4. Model Evaluation and Comparison (25 points)

Evaluation Metrics

- **Metrics Used:** The metrics used were accuracy, precision, recall, F1-score, and confusion matrix. Accuracy provides the overall correctness of predicting results. The F1-score balances the results between precision and recall.
- **Results:** Present the evaluation results for both LSTM and GRU models on the test set. Use tables or graphs to display the performance metrics clearly.

Results

- **LSTM Model:** On the training data, the LSTM achieved high precision (0.8499) and recall (0.9826), indicating it can effectively identify positive instances with minimal false negatives. However, on the validation set, the performance dropped significantly, with precision at 0.1868 and recall at 0.9444, suggesting overfitting or challenges in generalization.
- **GRU Model**: The GRU model performed slightly better, achieving a precision of 0.9441 and recall of 0.9594 on the training set, and precision of 0.1935 and perfect recall (1.0000) on the validation set. The confusion matrix shows the model's strengths in identifying true positives while struggling with false positives.

Comparative Analysis

- **Insights:** Basically, my GRU outperformed the LSTM model in both the test and validation sets. However, the precision in both models is terrible, indicating that I did not generalize well enough or perform good feature engineering. Basically, NASA should not hire me to create any models for there engines.

5. Conclusion and Future Directions (15 points)

Summary and Discussion

- **Key Findings:** Both of my models performed great on the training data, with high recall scores indicating that I was able to detect positive cases. However, on the validation set both models performed poorly. I saw a decreased precision indicating that the model was generating false positives. Overall, my models performed great on the training data indicating that the model tends to overfit as validation data performed poorly.
- **Challenges:** The primary challenges were feature engineering and hyperparameter tunning. If I wasn't limited to GPU units in Google Colab, I probably would have performed my hyperparameter tuning differently. Both models showed signs of overfitting so maybe I should have gotten rid of more correlated features in the Data Processing phase. I think another challenge that we may see with these models is that they overfit the training data. I think this shows the complexities when working with time series data.

Advantages and Disadvantages of LSTM and GRU

- **LSTM:** This model's gate architecture better captures long-term dependencies. Because of this, it can perform better on sequential data. However, computation is extensive and prone to overfitting, based on my results.
- **GRU:** This model is similar but has fewer gates in its architecture, making it simpler. As seen in my results, this model did better generalizing to the data than the LSTM, although it still performed poorly on the validation set.

Recommendations

- **Attention Mechanisms:** Maybe I could have somehow incorporated an attention layer to help the model focus more on relevant parts in the sequence. That way it could have captured the importance of certain features which is important in a task like this.
- **Feature Engineering and Regularization:** I probably could have done a better job at feature engineering. Maybe next time I do this I can remove than just .95 correlated features. Maybe I should have lowered that threshold or performed completely different types of feature engineering. I think I would have helped if I added L1/L2 in the dropout layers to help reduce the overfitting seen in my models.

6. Bonus Task: Application to FD002 Dataset (10 Bonus Points)

Complexity Handling

- **Approach:** I completely failed at trying to attempt this. But I tried to perform the same feature engineering steps and ensured that I had the same number of columns from when I trained the original LSTM and GRU.

7. References

---

[i] ChatGPT Prompt:

- ME: Correct me if I am wrong, but should I remove the columns that have the same values? For instance sensor18, sensor19 and many other sensors have the same values. seems like it would be irrelevant for my task to create an LSTM model. Correct me if I am wrong.
  index id cycle setting1 setting2 setting3 sensor1 sensor2 sensor3 sensor4 sensor5 sensor6 sensor7 sensor8 sensor9 sensor10 sensor11 sensor12 sensor13 sensor14 sensor15 count 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 20631.0 mean 51.506567786340945 108.80786195530997 -8.87014686636615e-06 2.3508312733265476e-06 100.0 518.67 642.6809335466046 1590.5231186079204 1408.9337816877514 14.620000000000001 21.60980320876352 553.367711211284 2388.0966516407348 9065.242940720276 1.3 47.54116814502448 521.4134700208425 2388.0961523920314 8143.752722117202 8.44214558189133 std 29.227632908799244 68.88099017721821 0.002187313449015146 0.0002930621245661 4496 0.0 0.0 0.5000532700606232 6.131149519690819 9.000604780543714 1.776399891637612e-15 0.0013889849127073112 0.8850922576634073 0.07098547889092417 22.082879525066232 0.0 0.2670873986396848 0.7375533922097114 0.07191891569857813 19.07617597595311 0.03750503795196652 min 1.0 1.0 -0.0087 -0.0006 100.0 518.67 641.21 1571.04 1382.25 14.62 21.6 549.85 2387.9 9021.73 1.3 46.85 518.69 2387.88 8099.94 8.3249 25% 26.0 52.0 -0.0015 -0.0002 100.0 518.67 642.325 1586.26 1402.36 14.62 21.61 552.81 2388.05 9053.1 1.3 47.35 520.96 2388.04 8133.245 8.4149 50% 52.0 104.0 0.0 0.0 100.0 518.67 642.64 1590.1 1408.04 14.62 21.61 553.44 2388.09 9060.66 1.3 47.51 521.48 2388.09 8140.54 8.4389 75% 77.0 156.0 0.0015 0.0003 100.0 518.67 643.0 1594.38 1414.5549999999998 14.62 21.61 554.01 2388.14 9069.42 1.3 47.7 521.95 2388.14 8148.31 8.4656 max 100.0 362.0 0.0087 0.0006 100.0 518.67 644.53 1616.91 1441.49 14.62 21.61 556.06 2388.56 9244.59 1.3 48.53 523.38 2388.56 8293.72 8.5848
- GPT: You're correct in your thinking—columns with constant values (i.e., the same value across all rows) do not provide useful information for modeling, especially in time-series models like LSTMs. These columns do not contribute to the learning process and can even introduce unnecessary noise, making the model less efficient.
  Why Remove Columns with Constant Values?
  Lack of Variability: Columns that do not change across rows provide no information about the variability in the data, which is essential for models to learn patterns.
  Model Efficiency: Keeping irrelevant columns increases the computational load without adding value. It can slow down training and complicate the model without improving accuracy.
  Noise Reduction: Removing such columns helps in reducing noise, making your model training cleaner and more interpretable.
  How to Identify and Remove Columns with Constant Values
  Here's how you can identify and remove columns with constant values in your DataFrame:
  Python Copy code

```
# Identify columns with constant values
constant_columns = [col for col in train_df.columns if train_df[col].nunique() == 1]

# Print the constant columns identified
print("Constant columns:", constant_columns)

# Drop the constant columns from the DataFrame
train_df_cleaned = train_df.drop(columns=constant_columns)
```

  In your specific case, sensors like sensor18 and sensor19 seem to have zero standard deviation (std), indicating that their values are constant. Removing them would be a good decision since they do not contribute to the variability needed for training your LSTM model. Let me know if you need help with the next steps in your LSTM model preparation!

[ii] ChatGPT Prompt:

- ME: Before I run the CODE below. Are there any techniques I can perform to meet this requirement of feature engineering? Implement feature engineering techniques, such as creating moving averages,

extracting trend features, or aggregating sensor data, to enhance model performance. CODE: ####### # TRAIN ####### # Data Labeling - generate column RUL (Remaining Useful Life or Time to Failure) # TODO: Calculate the maximum cycle value for each engine (id) and store it in a new DataFrame (rul) rul = train_df.groupby('id')['cycle'].max().reset_index() # TODO: Rename the columns in the rul DataFrame rul.columns = ['id', 'max_cycle'] # TODO: Merge the rul DataFrame with the original train_df based on the 'id' column train_df = train_df.merge(rul, on='id', how='left') # TODO: Calculate the Remaining Useful Life (RUL) by subtracting the current cycle from the maximum cycle train_df['RUL'] = train_df['max_cycle'] - train_df['cycle'] # TODO: Remove the temporary column used to calculate RUL train_df.drop('max_cycle', axis=1, inplace=True) # Generate label columns for training data # We will only make use of "label1" for binary classification, # while trying to answer the question: is a specific engine going to fail within w1 cycles? w1 = 30 w0 = 15 # TODO: Create a binary label ('label1') indicating if the engine will fail within w1 cycles (1) or not (0) train_df['label1'] = (train_df['RUL'] <= w1).astype(int) # Replace with the correct threshold value and label values # TODO: Initialize a second label ('label2') as a copy of 'label1' train_df['label2'] = train_df['label1'].copy() # TODO: Update 'label2' to indicate if the engine will fail within w0 cycles (2) or not (0/1) train_df.loc[train_df['RUL'] <= w0, 'label2'] = 2 # Replace with the correct threshold value and label value # MinMax normalization (from 0 to 1) # TODO: Create a normalized version of the 'cycle' column (e.g., 'cycle_norm') using the original 'cycle' values train_df['cycle_norm'] = train_df['cycle'] / train_df['cycle'].max() # Replace with the correct normalization code # TODO: Select the columns to be normalized (all columns except 'id', 'cycle', 'RUL', 'label1', and 'label2') cols_normalize = train_df.columns.difference(['id', 'cycle', 'RUL', 'label1', 'label2']) # Replace with the correct column selection code # TODO: Initialize a MinMaxScaler object to scale values between 0 and 1 min_max_scaler = MinMaxScaler() # Replace with the correct scaler initialization code # TODO: Apply MinMaxScaler to the selected columns and create a new normalized DataFrame norm_train_df = pd.DataFrame(min_max_scaler.fit_transform(train_df[cols_normalize]), columns=cols_normalize, index=train_df.index) # Replace with the correct normalization code # TODO: Join the normalized DataFrame with the original DataFrame (excluding normalized columns) join_df = train_df[['id', 'cycle', 'RUL', 'label1', 'label2']].join(norm_train_df) # Replace with the correct join code # TODO: Reorder the columns in the joined DataFrame to match the original order train_df = join_df.reindex(columns=train_df.columns) # Replace with the correct reindexing code

- ChatGPT: To meet the feature engineering requirement of enhancing model performance through techniques like creating moving averages, extracting trend features, or aggregating sensor data, you can implement the following methods before executing your provided code. These steps will help capture more meaningful patterns in the data that can significantly improve your model's predictive capabilities, especially when working with sequential models like LSTM and GRU.
- **Recommended Feature Engineering Techniques:**
- **Creating Moving Averages and Rolling Statistics:**
- Moving averages and rolling statistics (like standard deviation) can help smooth out noise in the data and capture recent trends.
- python
- Copy code
- window_size = 5 # You can adjust this window size based on your analysis
- for col in train_df.filter(like='sensor').columns:
- train_df[f'{col}_ma'] = train_df[col].rolling(window=window_size).mean()
- train_df[f'{col}_std'] = train_df[col].rolling(window=window_size).std()
- **Extracting Trend Features (Slope Calculation):**
- Calculate the slope or trend over a rolling window to identify increasing or decreasing trends in sensor readings.
- python
- Copy code
- from scipy.stats import linregress
-
- def calculate_slope(series):
- if len(series) < 2:
- return 0
- x = list(range(len(series)))
- slope, _, _, _, _ = linregress(x, series)

- return slope
-
- for col in train_df.filter(like='sensor').columns:
- train_df[f'{col}_trend'] = train_df[col].rolling(window=window_size).apply(calculate_slope, raw=False)
- **Aggregating Sensor Data:**
- Aggregate sensor readings over different windows to create summarized features, which can help capture overall engine performance.
- python
- Copy code
- aggregation_functions = ['mean', 'max', 'min', 'std']
- aggregated_df = train_df.groupby('id').agg({col: aggregation_functions for col in train_df.filter(like='sensor').columns})
- aggregated_df.columns = ['_'.join(col).strip() for col in aggregated_df.columns.values]
- train_df = train_df.merge(aggregated_df, on='id', how='left')
- **Lag Features:**
- Add lag features to capture the impact of past sensor readings on current conditions.
- python
- Copy code
- lag = 3 # Number of lags; adjust as needed
- for col in train_df.filter(like='sensor').columns:
- for i in range(1, lag + 1):
- train_df[f'{col}_lag_{i}'] = train_df[col].shift(i)
- **Integration with Your Code:**
- **Step 1:** Add the above feature engineering steps right before the data labeling section of your provided code.
- **Step 2:** Continue with your existing code to label the data, normalize, and prepare it for training.
- **Benefits of These Techniques:**
- **Moving Averages and Rolling Stats:** Smooth out noise and highlight recent changes.
- **Trend Extraction:** Capture directionality, which can be critical for predicting failure.
- **Aggregation:** Provides summarized insights that can highlight overall performance trends.
- **Lag Features:** Enhance the temporal understanding of the model, providing context from past sensor data.
- Implementing these feature engineering techniques will add depth to your dataset, improving the model's ability to learn from the data, especially in the context of time-series analysis for predictive maintenance tasks.
- Let me know if you need any further help with integrating these steps!
-

[iii] ChatGPT Prompt

- ME: When performing the Lag got this: <ipython-input-38-2409d03a36b9>:7: PerformanceWarning: DataFrame is highly fragmented. This is usually the result of calling frame.insert many times, which has poor performance. Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use newframe = frame.copy() train_df[f'{col}_lag_{i}'] = train_df[col].shift(i) <ipython-input-38-2409d03a36b9>:7: PerformanceWarning: DataFrame is highly fragmented. This is usually the result of calling frame.insert many times, which has poor performance. Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use newframe = frame.copy() train_df[f'{col}_lag_{i}'] = train_df[col].shift(i) <ipython-input-38-2409d03a36b9>:7: PerformanceWarning: DataFrame is highly fragmented. This is usually the result of calling frame.insert many times, which has poor performance. Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use newframe = frame.copy() train_df[f'{col}_lag_{i}'] = train_df[col].shift(i) <ipython-input-38-2409d03a36b9>:7: PerformanceWarning: DataFrame is highly fragmented. This is usually the result of calling frame.insert many times, which has poor performance. Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use newframe = frame.copy() train_df[f'{col}_lag_{i}'] = train_df[col].shift(i)
- ChatGPT: The warning you encountered indicates that the DataFrame is becoming highly fragmented due to repeatedly inserting new columns. This fragmentation occurs because each shift operation

followed by adding the result as a new column causes the DataFrame to internally reorganize its memory structure multiple times, leading to performance degradation.

- **Solution: Use pd.concat for Efficient Column Addition**
- Instead of adding each lag feature directly to the DataFrame within the loop, you can create a separate DataFrame for the lag features and then concatenate all at once. This approach significantly improves performance and avoids fragmentation.
- Here's an optimized way to implement lag features using pd.concat:
- python
- Copy code
- # Define the number of lags you want to create
- lag = 3 # Adjust based on your needs
-
- # Create a list to store lagged DataFrames
- lagged_features = []
-
- # Generate lag features for each sensor column in train_df
- for col in train_df.filter(like='sensor').columns:
- # Create lagged columns for the current sensor and append to the list
- for i in range(1, lag + 1):
- lagged_df = train_df[[col]].shift(i).rename(columns={col: f'{col}_lag_{i}'})
- lagged_features.append(lagged_df)
-
- # Concatenate all lagged features along columns
- lagged_train_df = pd.concat(lagged_features, axis=1)
-
- # Merge lagged features back to the original train_df
- train_df = pd.concat([train_df, lagged_train_df], axis=1)
-
- # Fill missing values caused by lagging
- train_df.fillna(0, inplace=True)
-
- # Repeat the same process for test_df
- lagged_features_test = []
- for col in test_df.filter(like='sensor').columns:
- for i in range(1, lag + 1):
- lagged_df = test_df[[col]].shift(i).rename(columns={col: f'{col}_lag_{i}'})
- lagged_features_test.append(lagged_df)
-
- # Concatenate all lagged features for test data
- lagged_test_df = pd.concat(lagged_features_test, axis=1)
-
- # Merge lagged features back to the original test_df
- test_df = pd.concat([test_df, lagged_test_df], axis=1)
-
- # Fill missing values caused by lagging in test data
- test_df.fillna(0, inplace=True)
- **Key Changes:**
- **Use pd.concat()**: This method efficiently combines multiple DataFrames, avoiding the overhead of repeated inserts.
- **Fill Missing Values**: After concatenation, fill NaN values that result from shifting operations.
- **Performance Optimization**: By handling all lag columns at once, you avoid fragmentation and significantly improve processing speed.
- This approach ensures that the performance is optimized, and you won't encounter the fragmentation warning during execution. Let me know if you need further assistance!
-