

Data Preparation and EDA

```
# Import Libraries & Set Data Path
```

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

```
# Path to raw sensor data
```

```
DATA_PATH = "condition+monitoring+hydraulic+systems"
```

```
# Inspect available sensor files
```

```
files = sorted(os.listdir(DATA_PATH))
files
```

```
[ '.DS_Store',
  '.ipynb_checkpoints',
  'CE.txt',
  'CP.txt',
  'EPS1.txt',
  'FS1.txt',
  'FS2.txt',
  'PS1.txt',
  'PS2.txt',
  'PS3.txt',
  'PS4.txt',
  'PS5.txt',
  'PS6.txt',
  'SE.txt',
  'TS1.txt',
  'TS2.txt',
  'TS3.txt',
  'TS4.txt',
  'VS1.txt',
  'description.txt',
  'documentation.txt',
  'profile.txt']
```

```
# Reload using tab delimiter
```

```
df_sample = pd.read_csv(sample_path, sep="\t", header=None)
df_sample.head()
```

	0	1	2	3	4	5	6	7
8 \								
0	47.202	47.273	47.250	47.332	47.213	47.372	47.273	47.438
46.691								
1	29.208	28.822	28.805	28.922	28.591	28.643	28.216	27.812

```

27.514
2  23.554  23.521  23.527  23.008  23.042  23.052  22.658  22.952
22.908
3  21.540  21.419  21.565  20.857  21.052  21.039  20.926  20.912
20.989
4  20.460  20.298  20.350  19.867  19.997  19.972  19.924  19.813
19.691

```

```

      9  ...      50      51      52      53      54      55      56
\
0  46.599  ...  31.554  30.953  30.639  30.561  30.368  30.224  29.790
1  27.481  ...  23.995  24.328  24.283  23.877  23.816  23.933  23.354
2  22.359  ...  21.711  21.564  21.564  21.526  21.753  21.749  21.802
3  20.882  ...  20.687  20.703  20.295  20.482  20.600  20.547  20.708
4  19.634  ...  19.887  19.919  19.696  19.634  19.747  20.005  19.919

```

```

      57      58      59
0  29.261  29.287  28.866
1  23.483  23.320  23.588
2  21.582  21.283  21.519
3  20.708  20.574  20.403
4  19.736  19.977  20.016

```

```
[5 rows x 60 columns]
```

```
df_sample.shape
df_sample.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2205 entries, 0 to 2204
Data columns (total 60 columns):
 #   Column  Non-Null Count  Dtype
---  -
0   0       2205 non-null    float64
1   1       2205 non-null    float64
2   2       2205 non-null    float64
3   3       2205 non-null    float64
4   4       2205 non-null    float64
5   5       2205 non-null    float64
6   6       2205 non-null    float64
7   7       2205 non-null    float64
8   8       2205 non-null    float64
9   9       2205 non-null    float64
10  10      2205 non-null    float64
11  11      2205 non-null    float64

```

12	12	2205	non-null	float64
13	13	2205	non-null	float64
14	14	2205	non-null	float64
15	15	2205	non-null	float64
16	16	2205	non-null	float64
17	17	2205	non-null	float64
18	18	2205	non-null	float64
19	19	2205	non-null	float64
20	20	2205	non-null	float64
21	21	2205	non-null	float64
22	22	2205	non-null	float64
23	23	2205	non-null	float64
24	24	2205	non-null	float64
25	25	2205	non-null	float64
26	26	2205	non-null	float64
27	27	2205	non-null	float64
28	28	2205	non-null	float64
29	29	2205	non-null	float64
30	30	2205	non-null	float64
31	31	2205	non-null	float64
32	32	2205	non-null	float64
33	33	2205	non-null	float64
34	34	2205	non-null	float64
35	35	2205	non-null	float64
36	36	2205	non-null	float64
37	37	2205	non-null	float64
38	38	2205	non-null	float64
39	39	2205	non-null	float64
40	40	2205	non-null	float64
41	41	2205	non-null	float64
42	42	2205	non-null	float64
43	43	2205	non-null	float64
44	44	2205	non-null	float64
45	45	2205	non-null	float64
46	46	2205	non-null	float64
47	47	2205	non-null	float64
48	48	2205	non-null	float64
49	49	2205	non-null	float64
50	50	2205	non-null	float64
51	51	2205	non-null	float64
52	52	2205	non-null	float64
53	53	2205	non-null	float64
54	54	2205	non-null	float64
55	55	2205	non-null	float64
56	56	2205	non-null	float64
57	57	2205	non-null	float64
58	58	2205	non-null	float64
59	59	2205	non-null	float64

```

dtypes: float64(60)
memory usage: 1.0 MB

# Re-load sensor files cleanly (tab-delimited) and force 60 columns

import os
import pandas as pd

SENSOR_PREFIXES = ("CE", "CP", "EPS", "FS", "PS", "SE", "TS", "VS")

sensor_files = sorted([
    f for f in os.listdir(DATA_PATH)
    if f.endswith(".txt")
    and f.startswith(SENSOR_PREFIXES)
    and os.path.isfile(os.path.join(DATA_PATH, f))
])

EXPECTED_COLS = 60

dfs = []
bad_files = []

for fname in sensor_files:
    fpath = os.path.join(DATA_PATH, fname)

    try:
        df = pd.read_csv(
            fpath,
            sep="\t",          # IMPORTANT: tab delimiter
            header=None,
            engine="python",   # more forgiving parser
            encoding="latin1"  # avoids unicode decode issues
        )
    except Exception as e:
        bad_files.append((fname, str(e)))
        continue

    # Drop completely empty columns (sometimes happen from trailing
    # tabs)
    df = df.dropna(axis=1, how="all")

    # Force exactly 60 columns (keep first 60 if extra)
    if df.shape[1] >= EXPECTED_COLS:
        df = df.iloc[:, :EXPECTED_COLS]
    else:
        # if fewer than 60, pad with NaN (rare, but keeps shape
        # consistent)
        for c in range(df.shape[1], EXPECTED_COLS):
            df[c] = pd.NA
        df = df.iloc[:, :EXPECTED_COLS]

```

```
df["sensor_file"] = fname.replace(".txt", "")
dfs.append(df)

df_all = pd.concat(dfs, ignore_index=True)

print("Sensor files found:", sensor_files)
print("Combined shape:", df_all.shape)
print("Unique sensor files:", df_all["sensor_file"].nunique())
print("Bad files:", bad_files)

df_all.head()
```

Sensor files found: ['CE.txt', 'CP.txt', 'EPS1.txt', 'FS1.txt', 'FS2.txt', 'PS1.txt', 'PS2.txt', 'PS3.txt', 'PS4.txt', 'PS5.txt', 'PS6.txt', 'SE.txt', 'TS1.txt', 'TS2.txt', 'TS3.txt', 'TS4.txt', 'VS1.txt']

Combined shape: (37485, 61)

Unique sensor files: 17

Bad files: []

	0	1	2	3	4	5	6	7
0	47.202	47.273	47.250	47.332	47.213	47.372	47.273	47.438
1	29.208	28.822	28.805	28.922	28.591	28.643	28.216	27.812
2	23.554	23.521	23.527	23.008	23.042	23.052	22.658	22.952
3	21.540	21.419	21.565	20.857	21.052	21.039	20.926	20.912
4	20.460	20.298	20.350	19.867	19.997	19.972	19.924	19.813

	9	...	51	52	53	54	55	56	57
0	46.599	...	30.953	30.639	30.561	30.368	30.224	29.790	29.261
1	27.481	...	24.328	24.283	23.877	23.816	23.933	23.354	23.483
2	22.359	...	21.564	21.564	21.526	21.753	21.749	21.802	21.582
3	20.882	...	20.703	20.295	20.482	20.600	20.547	20.708	20.708
4	19.634	...	19.919	19.696	19.634	19.747	20.005	19.919	19.736

	58	59	sensor_file
0	29.287	28.866	CE
1	23.320	23.588	CE
2	21.283	21.519	CE

```
3  20.574  20.403      CE
4  19.977  20.016      CE
```

```
[5 rows x 61 columns]
```

```
# sanity checks
```

```
print(df_all["sensor_file"].value_counts().head())
print("Any nulls?", df_all.isna().sum().sum())
print("Number of columns (should be consistent across sensors):",
df_all.shape[1])
```

```
sensor_file
```

```
CE      2205
```

```
PS5     2205
```

```
TS4     2205
```

```
TS3     2205
```

```
TS2     2205
```

```
Name: count, dtype: int64
```

```
Any nulls? 0
```

```
Number of columns (should be consistent across sensors): 61
```

```
# 1.4.1 Standardize Column Structure and Validate Consistency
```

```
# Rename signal columns (0-59) -> t0-t59
```

```
signal_cols = list(range(60))
```

```
rename_map = {i: f"t{i}" for i in signal_cols}
```

```
df_all = df_all.rename(columns=rename_map)
```

```
# Ensure signal values are numeric
```

```
for c in [f"t{i}" for i in range(60)]:
    df_all[c] = pd.to_numeric(df_all[c], errors="coerce")
```

```
# Add sample index within each sensor file (0..2204)
```

```
df_all["sample_idx"] = df_all.groupby("sensor_file").cumcount()
```

```
# Reorder columns cleanly
```

```
df_all = df_all[[f"t{i}" for i in range(60)] + ["sensor_file",
"sample_idx"]]
```

```
# Sanity checks
```

```
print("Combined shape:", df_all.shape) # should be (37485, 62)
```

```
print("Any NaNs:", int(df_all.isna().sum().sum()))
```

```
print("Rows per sensor:")
```

```
print(df_all["sensor_file"].value_counts().sort_index())
```

```
df_all.head()
```

```
Combined shape: (37485, 62)
```

```
Any NaNs: 0
```

```
Rows per sensor:
```

```
sensor_file
```

```
CE      2205
CP      2205
EPS1    2205
FS1     2205
FS2     2205
PS1     2205
PS2     2205
PS3     2205
PS4     2205
PS5     2205
PS6     2205
SE      2205
TS1     2205
TS2     2205
TS3     2205
TS4     2205
VS1     2205
```

```
Name: count, dtype: int64
```

	t0	t1	t2	t3	t4	t5	t6	t7
t8 \								
0	47.202	47.273	47.250	47.332	47.213	47.372	47.273	47.438
46.691								
1	29.208	28.822	28.805	28.922	28.591	28.643	28.216	27.812
27.514								
2	23.554	23.521	23.527	23.008	23.042	23.052	22.658	22.952
22.908								
3	21.540	21.419	21.565	20.857	21.052	21.039	20.926	20.912
20.989								
4	20.460	20.298	20.350	19.867	19.997	19.972	19.924	19.813
19.691								

	t9	...	t52	t53	t54	t55	t56	t57	t58
\									
0	46.599	...	30.639	30.561	30.368	30.224	29.790	29.261	29.287
1	27.481	...	24.283	23.877	23.816	23.933	23.354	23.483	23.320
2	22.359	...	21.564	21.526	21.753	21.749	21.802	21.582	21.283
3	20.882	...	20.295	20.482	20.600	20.547	20.708	20.708	20.574
4	19.634	...	19.696	19.634	19.747	20.005	19.919	19.736	19.977

	t59	sensor_file	sample_idx
0	28.866	CE	0
1	23.588	CE	1
2	21.519	CE	2
3	20.403	CE	3

```

4    20.016          CE          4

[5 rows x 62 columns]

# 1.4.2 Finalize DataFrame Schema (column order + validation)

# Put columns in a consistent, readable order
signal_cols = [f"t{i}" for i in range(60)]
meta_cols = ["sensor_file", "sample_idx"]

# (Optional) If sample_idx doesn't exist yet, create it per sensor
if "sample_idx" not in df_all.columns:
    df_all["sample_idx"] = df_all.groupby("sensor_file").cumcount()

# Reorder columns
df_all = df_all[signal_cols + meta_cols]

# Hard validations (these should all pass)
assert df_all.shape[1] == 62, f"Expected 62 columns (60 signals + sensor_file + sample_idx), got {df_all.shape[1]}"
assert df_all[signal_cols].select_dtypes(exclude="number").shape[1] == 0, "Signal columns must all be numeric"
assert df_all[signal_cols].isna().sum().sum() == 0, "No NaNs expected in signal columns"
assert df_all["sensor_file"].nunique() == 17, f"Expected 17 sensors, got {df_all['sensor_file'].nunique()}"

print("✅ Schema finalized.")
print("Final shape:", df_all.shape)
df_all.head()

✅ Schema finalized.
Final shape: (37485, 62)

   t0      t1      t2      t3      t4      t5      t6      t7
t8 \
0  47.202  47.273  47.250  47.332  47.213  47.372  47.273  47.438
46.691
1  29.208  28.822  28.805  28.922  28.591  28.643  28.216  27.812
27.514
2  23.554  23.521  23.527  23.008  23.042  23.052  22.658  22.952
22.908
3  21.540  21.419  21.565  20.857  21.052  21.039  20.926  20.912
20.989
4  20.460  20.298  20.350  19.867  19.997  19.972  19.924  19.813
19.691

   t9  ...  t52      t53      t54      t55      t56      t57      t58
\
0  46.599  ...  30.639  30.561  30.368  30.224  29.790  29.261  29.287

```

1	27.481	...	24.283	23.877	23.816	23.933	23.354	23.483	23.320
2	22.359	...	21.564	21.526	21.753	21.749	21.802	21.582	21.283
3	20.882	...	20.295	20.482	20.600	20.547	20.708	20.708	20.574
4	19.634	...	19.696	19.634	19.747	20.005	19.919	19.736	19.977

	t59	sensor_file	sample_idx
0	28.866	CE	0
1	23.588	CE	1
2	21.519	CE	2
3	20.403	CE	3
4	20.016	CE	4

[5 rows x 62 columns]

```
import numpy as np
import pandas as pd

# Identify signal vs metadata columns
signal_cols = [f"t{i}" for i in range(60)]
meta_cols = ["sensor_file", "sample_idx"]

print("Shape:", df_all.shape)
print("Signal columns:", len(signal_cols))
print("Metadata columns:", meta_cols)
```

df_all.head()

Shape: (37485, 62)
Signal columns: 60
Metadata columns: ['sensor_file', 'sample_idx']

	t0	t1	t2	t3	t4	t5	t6	t7	
t8 \									
0	47.202	47.273	47.250	47.332	47.213	47.372	47.273	47.438	
46.691									
1	29.208	28.822	28.805	28.922	28.591	28.643	28.216	27.812	
27.514									
2	23.554	23.521	23.527	23.008	23.042	23.052	22.658	22.952	
22.908									
3	21.540	21.419	21.565	20.857	21.052	21.039	20.926	20.912	
20.989									
4	20.460	20.298	20.350	19.867	19.997	19.972	19.924	19.813	
19.691									
	t9	...	t52	t53	t54	t55	t56	t57	t58
\									
0	46.599	...	30.639	30.561	30.368	30.224	29.790	29.261	29.287

1	27.481	...	24.283	23.877	23.816	23.933	23.354	23.483	23.320
2	22.359	...	21.564	21.526	21.753	21.749	21.802	21.582	21.283
3	20.882	...	20.295	20.482	20.600	20.547	20.708	20.708	20.574
4	19.634	...	19.696	19.634	19.747	20.005	19.919	19.736	19.977

	t59	sensor_file	sample_idx
0	28.866	CE	0
1	23.588	CE	1
2	21.519	CE	2
3	20.403	CE	3
4	20.016	CE	4

[5 rows x 62 columns]

```
df_all[signal_cols].describe().T.head(10)
```

	count	mean	std	min	25%	50%	75%
max							
t0	37485.0	172.932035	544.080575	0.0	8.274	27.300	58.895
2863.2							
t1	37485.0	168.550012	545.206100	0.0	1.742	10.075	50.887
2863.4							
t2	37485.0	168.531805	545.214902	0.0	1.742	10.075	50.875
2863.6							
t3	37485.0	168.457972	545.222312	0.0	1.743	10.075	50.824
2863.6							
t4	37485.0	168.364300	545.225644	0.0	1.745	10.075	50.805
2863.6							
t5	37485.0	168.269508	545.232371	0.0	1.743	10.077	50.738
2863.6							
t6	37485.0	167.400522	545.337421	0.0	1.594	10.073	50.703
2863.6							
t7	37485.0	165.069222	545.697994	0.0	0.781	10.073	50.652
2863.6							
t8	37485.0	163.261245	546.049724	0.0	0.690	10.073	46.392
2863.6							
t9	37485.0	166.194259	545.306601	0.0	1.736	24.734	49.473
2863.6							

```
print("Any NaNs in signals:", df_all[signal_cols].isna().any().any())
print("Total NaNs in signals:",
int(df_all[signal_cols].isna().sum().sum()))
```

```
non_numeric =
df_all[signal_cols].select_dtypes(exclude="number").columns.tolist()
print("Non-numeric signal columns:", non_numeric)
```

```
Any NaNs in signals: False
Total NaNs in signals: 0
Non-numeric signal columns: []
```

```
counts = df_all["sensor_file"].value_counts().sort_index()
counts
```

```
sensor_file
CE      2205
CP      2205
EPS1    2205
FS1      2205
FS2      2205
PS1      2205
PS2      2205
PS3      2205
PS4      2205
PS5      2205
PS6      2205
SE      2205
TS1      2205
TS2      2205
TS3      2205
TS4      2205
VS1      2205
```

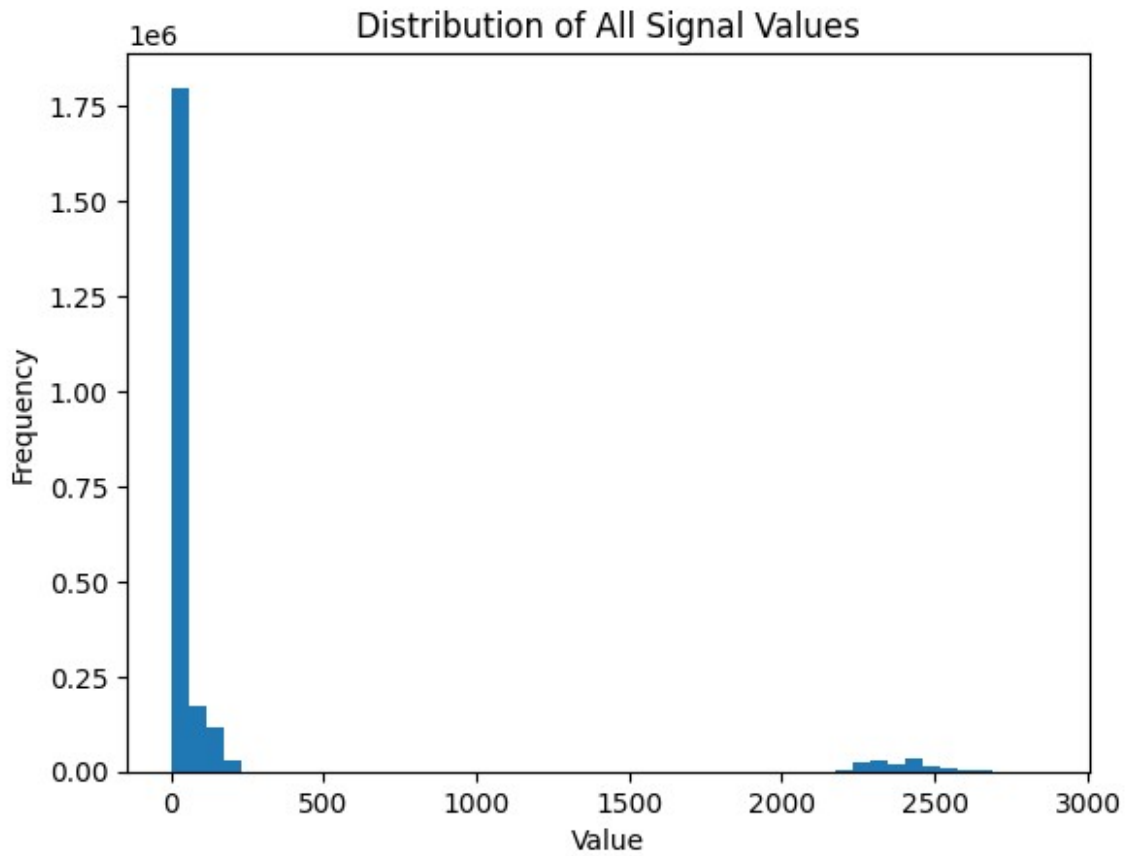
```
Name: count, dtype: int64
```

```
import matplotlib.pyplot as plt
```

```
all_vals = df_all[signal_cols].to_numpy().ravel()
```

```
plt.figure()
plt.hist(all_vals, bins=50)
plt.title("Distribution of All Signal Values")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

```
print("Overall min:", np.min(all_vals))
print("Overall max:", np.max(all_vals))
print("Overall mean:", np.mean(all_vals))
print("Overall std:", np.std(all_vals))
```

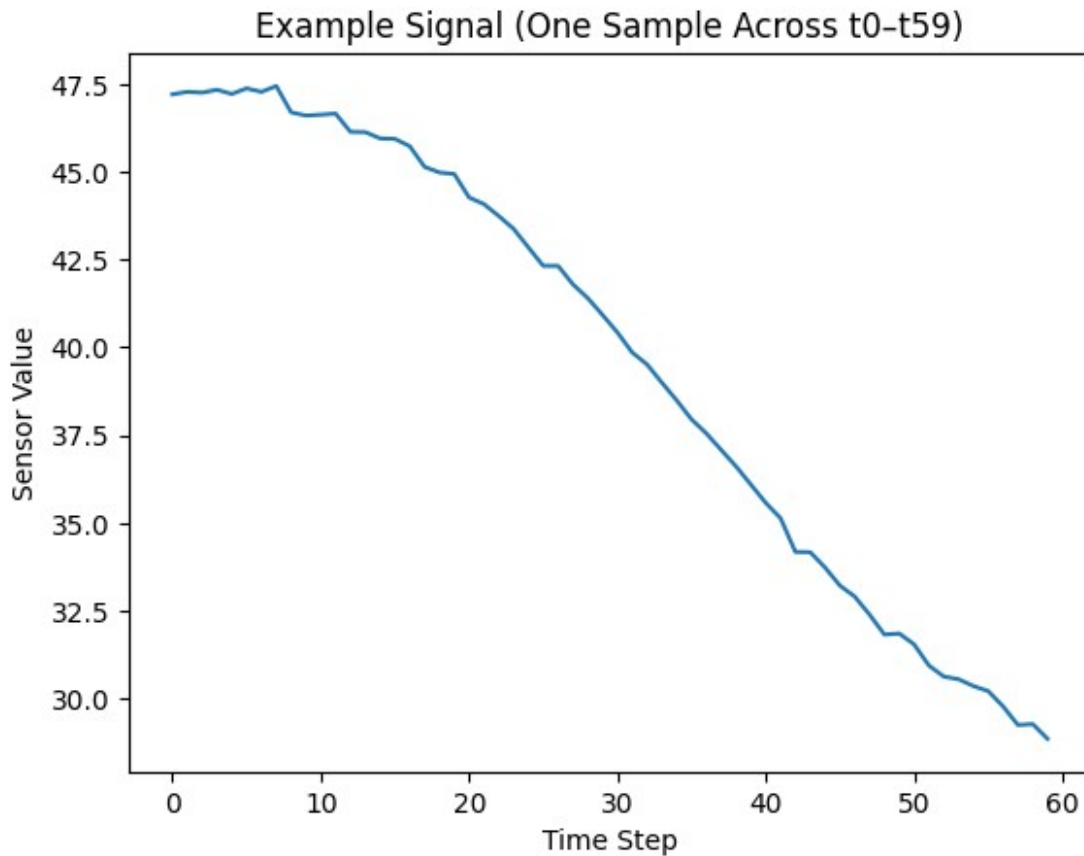


```
Overall min: 0.0
Overall max: 2863.6
Overall mean: 169.3598338019653
Overall std: 557.899619879862

# Pick one sample row to visualize
row = df_all.iloc[0][signal_cols].values

plt.figure()
plt.plot(range(60), row)
plt.title("Example Signal (One Sample Across t0-t59)")
plt.xlabel("Time Step")
plt.ylabel("Sensor Value")
plt.show()

print("Example sample metadata:", df_all.iloc[0][meta_cols].to_dict())
```

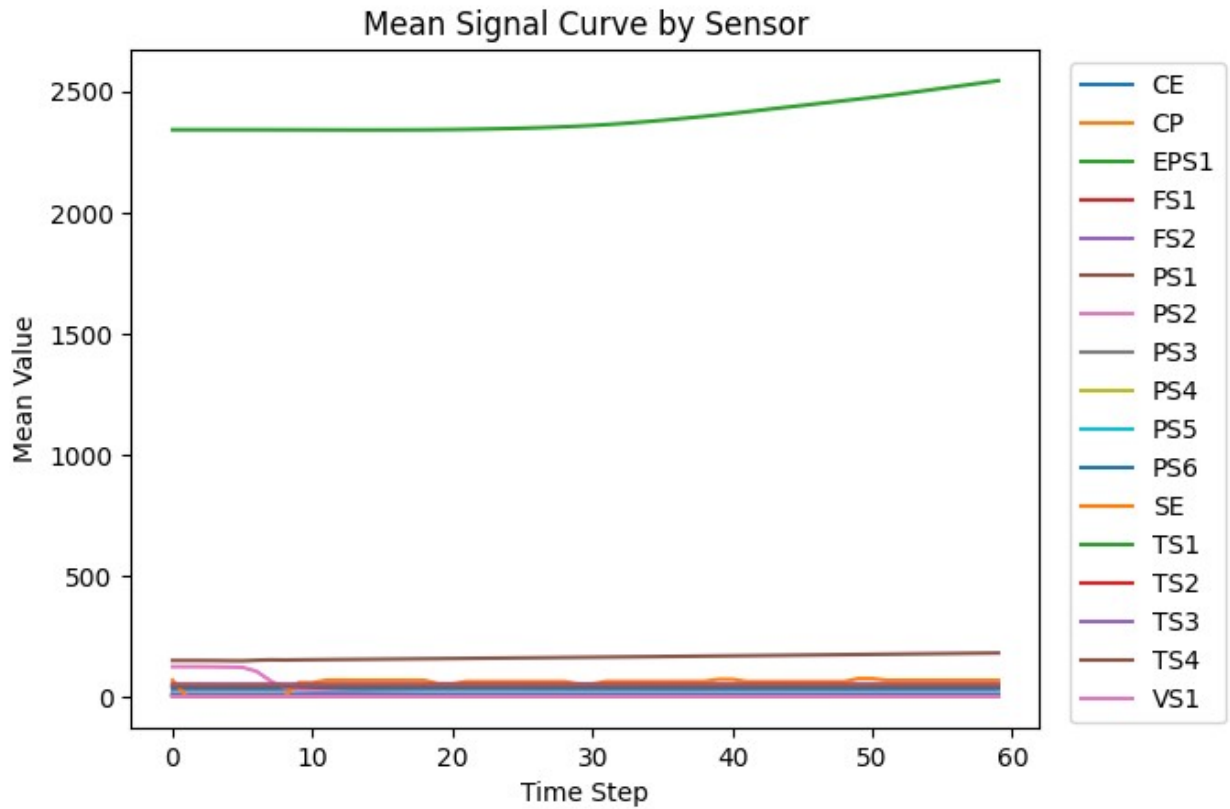


```
Example sample metadata: {'sensor_file': 'CE', 'sample_idx': 0}
mean_by_sensor = df_all.groupby("sensor_file")[signal_cols].mean()

plt.figure()
for sensor in mean_by_sensor.index:
    plt.plot(range(60), mean_by_sensor.loc[sensor].values,
            label=sensor)

plt.title("Mean Signal Curve by Sensor")
plt.xlabel("Time Step")
plt.ylabel("Mean Value")
plt.legend(bbox_to_anchor=(1.02, 1), loc="upper left")
plt.show()

mean_by_sensor.head()
```



	t0	t1	t2	t3
t4 \ sensor_file				
CE	31.325613	31.364606	31.395767	31.402451
31.395373				
CP	1.808180	1.810847	1.812283	1.813213
1.813780				
EPS1	2341.493968	2341.492971	2341.508571	2341.496054
2341.492517				
FS1	8.287100	0.857298	0.563584	0.031154
0.003567				
FS2	9.651824	9.651338	9.652036	9.651336
9.651783				
	t5	t6	t7	t8
t9 \ sensor_file				
CE	31.371016	31.351265	31.339065	31.307852
31.238561				
CP	1.812536	1.810481	1.809572	1.804766
1.800659				
EPS1	2341.491429	2341.464580	2341.421406	2341.343129

2341.237551				
FS1	0.002926	0.002683	0.002458	0.002436
0.002311				
FS2	9.651508	9.651132	9.651566	9.651130
9.650975				
	...	t50	t51	t52
t53 \				
sensor_file	...			
CE	...	31.087899	31.120328	31.142691
				31.155982
CP	...	1.791600	1.793863	1.794836
				1.795859
EPS1	...	2475.022222	2482.119184	2489.297596
				2497.076281
FS1	...	0.001870	0.001835	0.001830
				0.001846
FS2	...	9.652614	9.653091	9.652826
				9.653782
		t54	t55	t56
				t57
t58 \				
sensor_file				
CE	31.152029	31.134627	31.125789	31.125212
31.145241				
CP	1.796098	1.795245	1.795064	1.795286
1.796698				
EPS1	2504.926984	2512.928889	2520.620499	2528.449977
2536.288073				
FS1	0.001891	0.001807	0.001809	0.001803
0.001815				
FS2	9.654332	9.654861	9.654935	9.654814
9.654548				
		t59		
sensor_file				
CE	31.280535			
CP	1.805099			
EPS1	2544.310295			
FS1	0.001798			
FS2	9.654002			

[5 rows x 60 columns]

```
std_by_sensor = df_all.groupby("sensor_file")[signal_cols].std()
avg_std_by_sensor =
std_by_sensor.mean(axis=1).sort_values(ascending=False)
```

avg_std_by_sensor

```

sensor_file
EPS1    79.556522
CE      11.578925
SE      10.502767
TS4      8.108144
TS1      7.992139
TS3      7.452044
TS2      7.396778
PS4      4.290799
PS1      3.992401
PS2      1.601039
PS5      0.576732
PS6      0.549969
FS2      0.450497
CP       0.278925
VS1      0.068509
PS3      0.035219
FS1      0.015380
dtype: float64

```

```
# 2.8 Long-Term Trend Across Cycles (example: PS1)
```

```

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

```

```
signal_cols = [f"t{i}" for i in range(60)]
```

```

sensor_to_track = "PS1" # change to TS1, TS2, etc. if desired
df_s = df_all[df_all["sensor_file"] == sensor_to_track].copy()
df_s["cycle_mean"] = df_s[signal_cols].mean(axis=1)
df_s = df_s.sort_values("sample_idx")

```

```
# Rolling mean to highlight slow drift
```

```

window = 50 # cycles
df_s["cycle_mean_roll"] = df_s["cycle_mean"].rolling(window=window,
min_periods=1).mean()

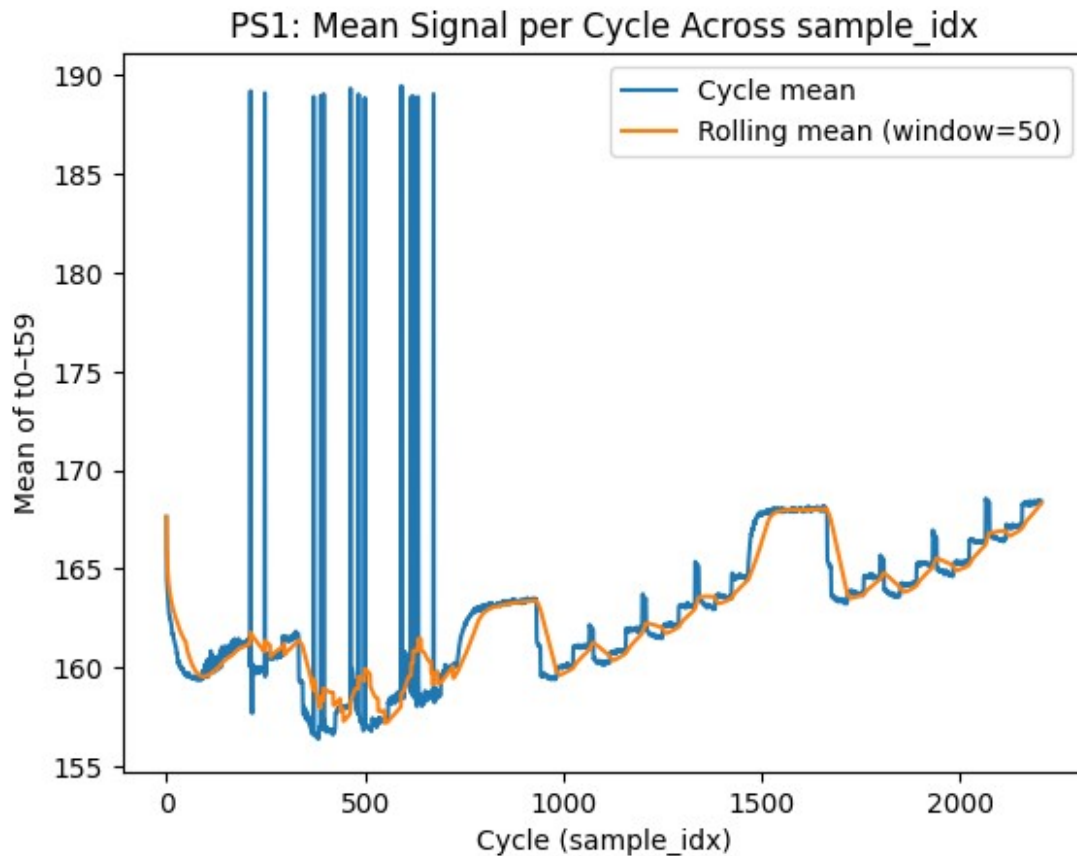
```

```

plt.figure()
plt.plot(df_s["sample_idx"], df_s["cycle_mean"], label="Cycle mean")
plt.plot(df_s["sample_idx"], df_s["cycle_mean_roll"], label=f"Rolling
mean (window={window})")
plt.title(f"{sensor_to_track}: Mean Signal per Cycle Across
sample_idx")
plt.xlabel("Cycle (sample_idx)")
plt.ylabel("Mean of t0-t59")
plt.legend()
plt.show()

```

```
print(f"{sensor_to_track} cycles:", df_s.shape[0])
```



PS1 cycles: 2205

2.9 Distributions by Sensor (cycle-level mean)

```
import matplotlib.pyplot as plt
```

```
signal_cols = [f"t{i}" for i in range(60)]
```

```
df_cycle = df_all.copy()
```

```
df_cycle["cycle_mean"] = df_cycle[signal_cols].mean(axis=1)
```

Boxplot of cycle-level mean values by sensor

```
plt.figure(figsize=(12, 5))
```

```
df_cycle.boxplot(column="cycle_mean", by="sensor_file", grid=False,
rot=45)
```

```
plt.title("Cycle Mean Distribution by Sensor")
```

```
plt.suptitle("")
```

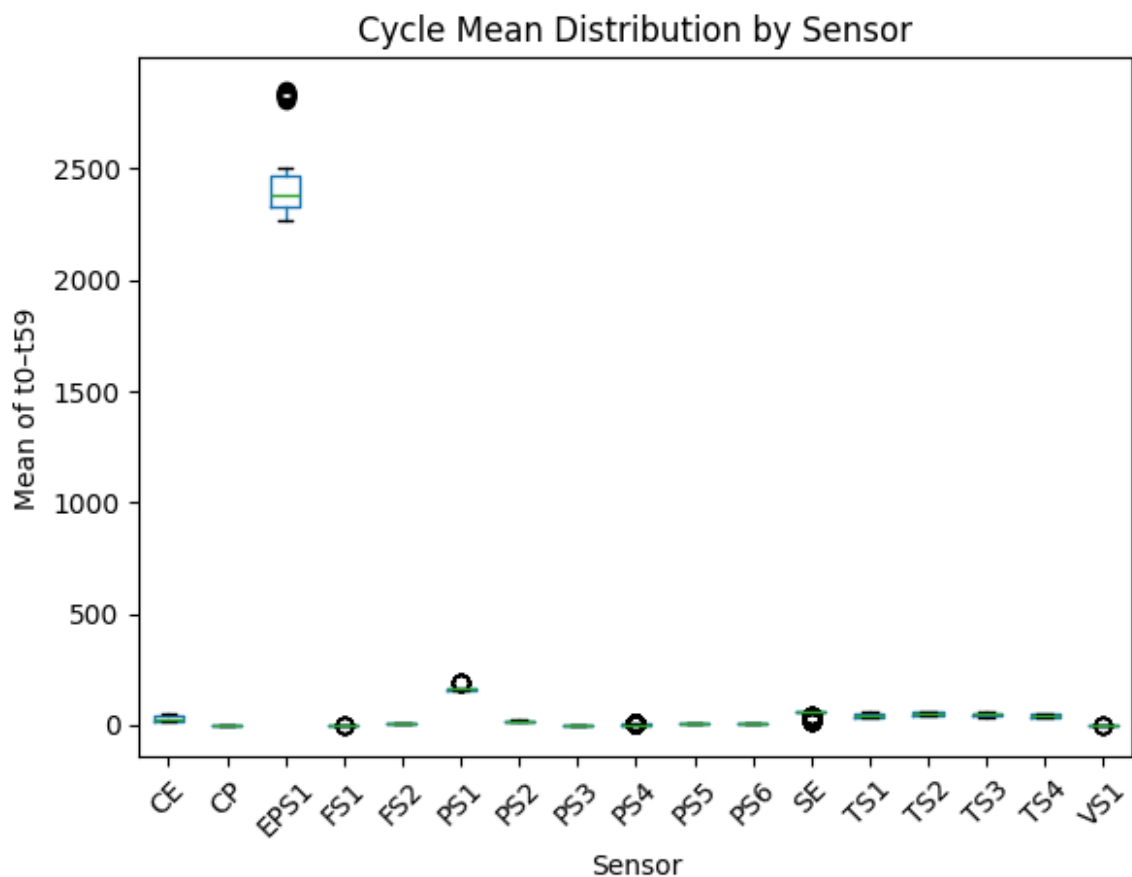
```
plt.xlabel("Sensor")
```

```
plt.ylabel("Mean of t0-t59")
```

```
plt.show()
```

```
df_cycle.groupby("sensor_file")["cycle_mean"].describe().round(3)
```

<Figure size 1200x500 with 0 Axes>



75% \ sensor_file	count	mean	std	min	25%	50%
CE	2205.0	31.299	11.575	17.556	20.085	27.393
46.677						
CP	2205.0	1.808	0.278	1.062	1.550	1.740
2.148						
EPS1	2205.0	2392.952	77.044	2271.783	2326.020	2381.923
2471.027						
FS1	2205.0	0.164	0.009	0.144	0.157	0.164
0.171						
FS2	2205.0	9.651	0.450	8.863	9.201	9.691
10.158						
PS1	2205.0	162.710	3.778	156.394	160.088	162.832
164.712						
PS2	2205.0	19.312	1.078	16.159	18.503	19.362
20.170						
PS3	2205.0	0.269	0.024	0.206	0.252	0.270
0.289						
PS4	2205.0	2.611	4.290	0.000	0.000	0.000
3.871						

PS5	2205.0	9.168	0.576	8.369	8.552	9.125
9.848						
PS6	2205.0	9.084	0.550	8.324	8.491	9.041
9.732						
SE	2205.0	55.288	8.960	18.277	56.270	58.758
59.657						
TS1	2205.0	45.425	7.992	35.314	36.237	44.837
54.104						
TS2	2205.0	50.366	7.396	40.859	41.864	49.781
58.584						
TS3	2205.0	47.662	7.452	38.246	39.123	47.070
55.694						
TS4	2205.0	40.736	8.108	30.391	31.273	40.429
49.409						
VS1	2205.0	0.613	0.060	0.524	0.555	0.610
0.650						

	max
sensor_file	
CE	47.904
CP	2.840
EPS1	2855.713
FS1	0.222
FS2	10.405
PS1	189.472
PS2	21.760
PS3	0.321
PS4	10.212
PS5	9.982
PS6	9.861
SE	60.755
TS1	57.899
TS2	61.958
TS3	59.423
TS4	53.060
VS1	0.839

```

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# 2.10 Correlation Heatmap of Sensors (based on cycle-level means)
signal_cols = [f"t{i}" for i in range(60)]

# Compute cycle-level mean per (sensor, cycle)
df_cycle = df_all.copy()
df_cycle["cycle_mean"] = df_cycle[signal_cols].mean(axis=1)

# Pivot to wide: rows = cycle, cols = sensor

```


CE	1.000	0.974	0.824	-0.120	0.920	0.679	0.777	0.780	
0.808									
CP	0.974	1.000	0.790	-0.121	0.876	0.666	0.744	0.741	
0.746									
EPS1	0.824	0.790	1.000	-0.060	0.825	0.905	0.625	0.721	
0.642									
FS1	-0.120	-0.121	-0.060	1.000	-0.184	0.049	-0.335	0.360	-
0.129									
FS2	0.920	0.876	0.825	-0.184	1.000	0.678	0.812	0.747	
0.684									
PS1	0.679	0.666	0.905	0.049	0.678	1.000	0.472	0.679	
0.449									
PS2	0.777	0.744	0.625	-0.335	0.812	0.472	1.000	0.487	
0.621									
PS3	0.780	0.741	0.721	0.360	0.747	0.679	0.487	1.000	
0.568									
PS4	0.808	0.746	0.642	-0.129	0.684	0.449	0.621	0.568	
1.000									
PS5	0.973	0.935	0.842	-0.139	0.981	0.697	0.809	0.786	
0.736									
PS6	0.973	0.935	0.842	-0.138	0.980	0.697	0.809	0.787	
0.736									
SE	0.293	0.303	0.276	-0.304	0.461	0.349	0.389	0.175	
0.116									
TS1	-0.946	-0.912	-0.835	0.157	-0.995	-0.690	-0.813	-0.770	-
0.700									
TS2	-0.946	-0.909	-0.838	0.148	-0.993	-0.688	-0.810	-0.775	-
0.703									
TS3	-0.942	-0.904	-0.834	0.163	-0.996	-0.688	-0.814	-0.766	-
0.698									
TS4	-0.956	-0.927	-0.837	0.158	-0.991	-0.693	-0.814	-0.772	-
0.711									
VS1	-0.852	-0.821	-0.709	0.190	-0.920	-0.606	-0.763	-0.677	-
0.650									
sensor_file	PS5	PS6	SE	TS1	TS2	TS3	TS4	VS1	
sensor_file									
CE	0.973	0.973	0.293	-0.946	-0.946	-0.942	-0.956	-0.852	
CP	0.935	0.935	0.303	-0.912	-0.909	-0.904	-0.927	-0.821	
EPS1	0.842	0.842	0.276	-0.835	-0.838	-0.834	-0.837	-0.709	
FS1	-0.139	-0.138	-0.304	0.157	0.148	0.163	0.158	0.190	
FS2	0.981	0.980	0.461	-0.995	-0.993	-0.996	-0.991	-0.920	
PS1	0.697	0.697	0.349	-0.690	-0.688	-0.688	-0.693	-0.606	
PS2	0.809	0.809	0.389	-0.813	-0.810	-0.814	-0.814	-0.763	
PS3	0.786	0.787	0.175	-0.770	-0.775	-0.766	-0.772	-0.677	
PS4	0.736	0.736	0.116	-0.700	-0.703	-0.698	-0.711	-0.650	
PS5	1.000	1.000	0.377	-0.993	-0.993	-0.992	-0.995	-0.898	
PS6	1.000	1.000	0.376	-0.993	-0.993	-0.992	-0.995	-0.898	

SE	0.377	0.376	1.000	-0.423	-0.397	-0.431	-0.419	-0.647
TS1	-0.993	-0.993	-0.423	1.000	0.999	1.000	0.999	0.913
TS2	-0.993	-0.993	-0.397	0.999	1.000	0.999	0.998	0.903
TS3	-0.992	-0.992	-0.431	1.000	0.999	1.000	0.998	0.915
TS4	-0.995	-0.995	-0.419	0.999	0.998	0.998	1.000	0.912
VS1	-0.898	-0.898	-0.647	0.913	0.903	0.915	0.912	1.000

CNN classification

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

Mounted at /content/drive

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

DATA_PATH = "c/content/drive/MyDrive/AAI-530 Final Project Data"

DATA_PATH = "/content/drive/MyDrive/AAI-530 Final Project Data" #
Corrected path
files = sorted(os.listdir(DATA_PATH))
files

['.ipynb_checkpoints',
'CE.txt',
'CP.txt',
'EPS1.txt',
'FS1.txt',
'FS2.txt',
'PS1.txt',
'PS2.txt',
'PS3.txt',
'PS4.txt',
'PS5.txt',
'PS6.txt',
'SE.txt',
'TS1.txt',
'TS2.txt',
'TS3.txt',
'TS4.txt',
'VS1.txt',
'cnn_predictions_with_sample_idx.csv',
'cnn_predictions_with_sample_idx_and_targets.csv',
'description.txt',
'df_all.csv',
```

```
'df_all_with_targets_and_predictions.csv',  
'documentation.txt',  
'optuna_trials.csv',  
'profile.txt',  
'tableau_predictions_and_targets.csv']
```

```
sample_path= os.path.join(DATA_PATH, files[1])  
df_sample = pd.read_csv(sample_path, sep="\t", header=None)  
df_sample.head()
```

```
{"type": "dataframe", "variable_name": "df_sample"}
```

```
df_sample.shape  
df_sample.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 2205 entries, 0 to 2204
```

```
Data columns (total 60 columns):
```

#	Column	Non-Null Count	Dtype
0	0	2205 non-null	float64
1	1	2205 non-null	float64
2	2	2205 non-null	float64
3	3	2205 non-null	float64
4	4	2205 non-null	float64
5	5	2205 non-null	float64
6	6	2205 non-null	float64
7	7	2205 non-null	float64
8	8	2205 non-null	float64
9	9	2205 non-null	float64
10	10	2205 non-null	float64
11	11	2205 non-null	float64
12	12	2205 non-null	float64
13	13	2205 non-null	float64
14	14	2205 non-null	float64
15	15	2205 non-null	float64
16	16	2205 non-null	float64
17	17	2205 non-null	float64
18	18	2205 non-null	float64
19	19	2205 non-null	float64
20	20	2205 non-null	float64
21	21	2205 non-null	float64
22	22	2205 non-null	float64
23	23	2205 non-null	float64
24	24	2205 non-null	float64
25	25	2205 non-null	float64
26	26	2205 non-null	float64
27	27	2205 non-null	float64
28	28	2205 non-null	float64
29	29	2205 non-null	float64

30	30	2205	non-null	float64
31	31	2205	non-null	float64
32	32	2205	non-null	float64
33	33	2205	non-null	float64
34	34	2205	non-null	float64
35	35	2205	non-null	float64
36	36	2205	non-null	float64
37	37	2205	non-null	float64
38	38	2205	non-null	float64
39	39	2205	non-null	float64
40	40	2205	non-null	float64
41	41	2205	non-null	float64
42	42	2205	non-null	float64
43	43	2205	non-null	float64
44	44	2205	non-null	float64
45	45	2205	non-null	float64
46	46	2205	non-null	float64
47	47	2205	non-null	float64
48	48	2205	non-null	float64
49	49	2205	non-null	float64
50	50	2205	non-null	float64
51	51	2205	non-null	float64
52	52	2205	non-null	float64
53	53	2205	non-null	float64
54	54	2205	non-null	float64
55	55	2205	non-null	float64
56	56	2205	non-null	float64
57	57	2205	non-null	float64
58	58	2205	non-null	float64
59	59	2205	non-null	float64

dtypes: float64(60)
memory usage: 1.0 MB

```
import os
import pandas as pd
```

```
SENSOR_PREFIXES = ("CE", "CP", "EPS", "FS", "PS", "SE", "TS", "VS")
```

```
sensor_files = sorted([
    f for f in os.listdir(DATA_PATH)
    if f.endswith(".txt")
    and f.startswith(SENSOR_PREFIXES)
    and os.path.isfile(os.path.join(DATA_PATH, f))
])
```

```
EXPECTED_COLS = 60
```

```
dfs = []
bad_files = []
```

```

for fname in sensor_files:
    fpath = os.path.join(DATA_PATH, fname)

    try:
        df = pd.read_csv(
            fpath,
            sep="\t",          # IMPORTANT: tab delimiter
            header=None,
            engine="python",   # more forgiving parser
            encoding="latin1"  # avoids unicode decode issues
        )
    except Exception as e:
        bad_files.append((fname, str(e)))
        continue

    # Drop completely empty columns (sometimes happen from trailing
    # tabs)
    df = df.dropna(axis=1, how="all")

    # Force exactly 60 columns (keep first 60 if extra)
    if df.shape[1] >= EXPECTED_COLS:
        df = df.iloc[:, :EXPECTED_COLS]
    else:
        # if fewer than 60, pad with NaN (rare, but keeps shape
        # consistent)
        for c in range(df.shape[1], EXPECTED_COLS):
            df[c] = pd.NA
        df = df.iloc[:, :EXPECTED_COLS]

    df["sensor_file"] = fname.replace(".txt", "")
    dfs.append(df)

df_all = pd.concat(dfs, ignore_index=True)

print("Sensor files found:", sensor_files)
print("Combined shape:", df_all.shape)
print("Unique sensor files:", df_all["sensor_file"].nunique())
print("Bad files:", bad_files)

df_all.head()

Sensor files found: ['CE.txt', 'CP.txt', 'EPS1.txt', 'FS1.txt',
'FS2.txt', 'PS1.txt', 'PS2.txt', 'PS3.txt', 'PS4.txt', 'PS5.txt',
'PS6.txt', 'SE.txt', 'TS1.txt', 'TS2.txt', 'TS3.txt', 'TS4.txt',
'VS1.txt']
Combined shape: (37485, 61)
Unique sensor files: 17
Bad files: []

```

```

{"type": "dataframe", "variable_name": "df_all"}

print(df_all["sensor_file"].value_counts().head())
print("Any nulls?", df_all.isna().sum().sum())
print("Number of columns (should be consistent across sensors):",
df_all.shape[1])

```

```

sensor_file
CE      2205
CP      2205
EPS1    2205
FS1     2205
FS2     2205
Name: count, dtype: int64
Any nulls? 0
Number of columns (should be consistent across sensors): 61

```

1.4.1 Standardize Column Structure and Validate Consistency

Rename signal columns (0-59) -> t0-t59

```

signal_cols = list(range(60))
rename_map = {i: f"t{i}" for i in signal_cols}
df_all = df_all.rename(columns=rename_map)

```

Ensure signal values are numeric

```

for c in [f"t{i}" for i in range(60)]:
    df_all[c] = pd.to_numeric(df_all[c], errors="coerce")

```

Add sample index within each sensor file (0..2204)

```

df_all["sample_idx"] = df_all.groupby("sensor_file").cumcount()

```

Reorder columns cleanly

```

df_all = df_all[[f"t{i}" for i in range(60)] + ["sensor_file",
"sample_idx"]]

```

Sanity checks

```

print("Combined shape:", df_all.shape) # should be (37485, 62)
print("Any NaNs:", int(df_all.isna().sum().sum()))
print("Rows per sensor:")
print(df_all["sensor_file"].value_counts().sort_index())

```

```

df_all.head()

```

```

Combined shape: (37485, 62)
Any NaNs: 0
Rows per sensor:
sensor_file
CE      2205
CP      2205
EPS1    2205
FS1     2205

```

```

FS2      2205
PS1      2205
PS2      2205
PS3      2205
PS4      2205
PS5      2205
PS6      2205
SE       2205
TS1      2205
TS2      2205
TS3      2205
TS4      2205
VS1      2205
Name: count, dtype: int64

{"type": "dataframe", "variable_name": "df_all"}

# 1.4.2 Finalize DataFrame Schema (column order + validation)

# Put columns in a consistent, readable order
signal_cols = [f"t{i}" for i in range(60)]
meta_cols = ["sensor_file", "sample_idx"]

# (Optional) If sample_idx doesn't exist yet, create it per sensor
if "sample_idx" not in df_all.columns:
    df_all["sample_idx"] = df_all.groupby("sensor_file").cumcount()

# Reorder columns
df_all = df_all[signal_cols + meta_cols]

# Hard validations (these should all pass)
assert df_all.shape[1] == 62, f"Expected 62 columns (60 signals + sensor_file + sample_idx), got {df_all.shape[1]}"
assert df_all[signal_cols].select_dtypes(exclude="number").shape[1] == 0, "Signal columns must all be numeric"
assert df_all[signal_cols].isna().sum().sum() == 0, "No NaNs expected in signal columns"
assert df_all["sensor_file"].nunique() == 17, f"Expected 17 sensors, got {df_all['sensor_file'].nunique()}"

print("□ Schema finalized.")
print("Final shape:", df_all.shape)
df_all.head()

□ Schema finalized.
Final shape: (37485, 62)

{"type": "dataframe", "variable_name": "df_all"}

dfs_by_sensor = {}
for sensor_file in df_all['sensor_file'].unique():

```

```

    dfs_by_sensor[sensor_file] = df_all[df_all['sensor_file'] ==
sensor_file].copy()

print(f"Created {len(dfs_by_sensor)} DataFrames, one for each unique
sensor file.")
print("Keys in dfs_by_sensor dictionary:", dfs_by_sensor.keys())

# Display the head of one of the split dataframes
if 'CE' in dfs_by_sensor:
    print("\nHead of 'CE' sensor DataFrame:")
    print(dfs_by_sensor['CE'].head())

```

Created 17 DataFrames, one for each unique sensor file.
Keys in dfs_by_sensor dictionary: dict_keys(['CE', 'CP', 'EPS1',
'FS1', 'FS2', 'PS1', 'PS2', 'PS3', 'PS4', 'PS5', 'PS6', 'SE', 'TS1',
'TS2', 'TS3', 'TS4', 'VS1'])

Head of 'CE' sensor DataFrame:

	t0	t1	t2	t3	t4	t5	t6	t7	
t8 \									
0	47.202	47.273	47.250	47.332	47.213	47.372	47.273	47.438	
	46.691								
1	29.208	28.822	28.805	28.922	28.591	28.643	28.216	27.812	
	27.514								
2	23.554	23.521	23.527	23.008	23.042	23.052	22.658	22.952	
	22.908								
3	21.540	21.419	21.565	20.857	21.052	21.039	20.926	20.912	
	20.989								
4	20.460	20.298	20.350	19.867	19.997	19.972	19.924	19.813	
	19.691								
	t9	...	t52	t53	t54	t55	t56	t57	t58
\									
0	46.599	...	30.639	30.561	30.368	30.224	29.790	29.261	29.287
1	27.481	...	24.283	23.877	23.816	23.933	23.354	23.483	23.320
2	22.359	...	21.564	21.526	21.753	21.749	21.802	21.582	21.283
3	20.882	...	20.295	20.482	20.600	20.547	20.708	20.708	20.574
4	19.634	...	19.696	19.634	19.747	20.005	19.919	19.736	19.977
	t59	sensor_file	sample_idx						
0	28.866	CE	0						
1	23.588	CE	1						
2	21.519	CE	2						
3	20.403	CE	3						
4	20.016	CE	4						

```
[5 rows x 62 columns]
```

```
target= os.path.join(DATA_PATH, 'profile.txt')  
print(target)
```

```
/content/drive/MyDrive/AAI-530 Final Project Data/profile.txt
```

```
try:
```

```
    df_target = pd.read_csv(  
        target,  
        sep="\t",  
        header=None,  
        engine="python",  
        encoding="latin1"  
    )
```

```
except Exception as e:
```

```
    bad_files.append((fname, str(e)))
```

```
df_target.describe()
```

```
{  
  "summary": {  
    "name": "df_target",  
    "rows": 8,  
    "fields": {  
      "column": 0,  
      "properties": {  
        "dtype": "number",  
        "std": 764.9018077580807,  
        "min": 3.0,  
        "max": 2205.0,  
        "num_unique_values": 6,  
        "samples": [2205.0, 41.24081632653061, 100.0]  
      },  
      "description": ""  
    },  
    "column": 1,  
    "properties": {  
      "dtype": "number",  
      "std": 752.1705935632439,  
      "min": 10.681801937203986,  
      "max": 2205.0,  
      "num_unique_values": 6,  
      "samples": [2205.0, 90.6938775510204, 100.0]  
    },  
    "description": ""  
  },  
  "column": 2,  
  "properties": {  
    "dtype": "number",  
    "std": 779.3589146800241,  
    "min": 0.0,  
    "max": 2205.0,  
    "num_unique_values": 6,  
    "samples": [2205.0, 0.6693877551020408, 2.0]  
  },  
  "description": ""  
},  
  "column": 3,  
  "properties": {  
    "dtype": "number",  
    "std": 746.9130042332717,  
    "min": 16.435848445754125,  
    "max": 2205.0,  
    "num_unique_values": 6,  
    "samples": [2205.0, 107.19954648526077, 130.0]  
  },  
  "description": ""  
},  
  "column": 4,  
  "properties": {  
    "dtype": "number",  
    "std": 779.4430252774183,  
    "min": 0.0,  
    "max": 2205.0,  
    "num_unique_values": 5,  
    "samples": [  

```

```
0.34285714285714286,\n                1.0,\n                0.47477189097222705\n],\n    \"semantic_type\": \"\",\n    \"description\": \"\"\n}\n    }\n    }\n}","type":"dataframe"}
```

```
signal_cols = [f"t{i}" for i in range(60)] # Define signal_cols
arrays = [df[signal_cols].values for df in dfs_by_sensor.values()]
```

```
X = np.stack(arrays, axis=-1)
y = df_target.values
```

```
print(f"Input shape (X): {X.shape}")
print(f"Target shape (y): {y.shape}")
```

```
Input shape (X): (2205, 60, 17)
Target shape (y): (2205, 5)
```

```
from sklearn.preprocessing import StandardScaler
X_features = X
y_target = df_target.iloc[:, 2]
original_shape = X_features.shape
X_features_resaped = X_features.reshape(-1, original_shape[2])
scaler = StandardScaler()
X_features_scaled = scaler.fit_transform(X_features_resaped)
X_features = X_features_scaled.reshape(original_shape)

print(f"X_features after scaling and reshaping: {X_features.shape}")
```

```
X_features after scaling and reshaping: (2205, 60, 17)
```

```
from sklearn.model_selection import TimeSeriesSplit

# To get roughly 20% of the data as the test set in the last split,
# n_splits should be 4.
tscv = TimeSeriesSplit(n_splits=4)
train_index, test_index = list(tscv.split(X_features))[-1]

X_train, X_test = X_features[train_index], X_features[test_index]
y_train, y_test = y_target.iloc[train_index],
y_target.iloc[test_index]
```

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
```

```
X_train_pt = torch.from_numpy(X_train).float()
X_test_pt = torch.from_numpy(X_test).float()
y_train_pt = torch.from_numpy(y_train.values).long()
y_test_pt = torch.from_numpy(y_test.values).long()
```

```

print(f"X_train_pt shape: {X_train_pt.shape}, dtype:
{X_train_pt.dtype}")
print(f"X_test_pt shape: {X_test_pt.shape}, dtype: {X_test_pt.dtype}")
print(f"y_train_pt shape: {y_train_pt.shape}, dtype:
{y_train_pt.dtype}")
print(f"y_test_pt shape: {y_test_pt.shape}, dtype: {y_test_pt.dtype}")

```

```

X_train_pt shape: torch.Size([1764, 60, 17]), dtype: torch.float32
X_test_pt shape: torch.Size([441, 60, 17]), dtype: torch.float32
y_train_pt shape: torch.Size([1764]), dtype: torch.int64
y_test_pt shape: torch.Size([441]), dtype: torch.int64

```

```

pip install optuna

```

```

Collecting optuna

```

```

  Downloading optuna-4.7.0-py3-none-any.whl.metadata (17 kB)
Requirement already satisfied: alembic>=1.5.0 in
/usr/local/lib/python3.12/dist-packages (from optuna) (1.18.4)
Collecting colorlog (from optuna)
  Downloading colorlog-6.10.1-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: numpy in
/usr/local/lib/python3.12/dist-packages (from optuna) (2.0.2)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.12/dist-packages (from optuna) (26.0)
Requirement already satisfied: sqlalchemy>=1.4.2 in
/usr/local/lib/python3.12/dist-packages (from optuna) (2.0.46)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-
packages (from optuna) (4.67.3)
Requirement already satisfied: PyYAML in
/usr/local/lib/python3.12/dist-packages (from optuna) (6.0.3)
Requirement already satisfied: Mako in /usr/local/lib/python3.12/dist-
packages (from alembic>=1.5.0->optuna) (1.3.10)
Requirement already satisfied: typing-extensions>=4.12 in
/usr/local/lib/python3.12/dist-packages (from alembic>=1.5.0->optuna)
(4.15.0)
Requirement already satisfied: greenlet>=1 in
/usr/local/lib/python3.12/dist-packages (from sqlalchemy>=1.4.2-
>optuna) (3.3.1)
Requirement already satisfied: MarkupSafe>=0.9.2 in
/usr/local/lib/python3.12/dist-packages (from Mako->alembic>=1.5.0-
>optuna) (3.0.3)
Downloading optuna-4.7.0-py3-none-any.whl (413 kB)
_____ 413.9/413.9 kB 10.4 MB/s eta
0:00:00

```

```

from sklearn.utils.class_weight import compute_class_weight
import numpy as np

```

```

y_train_np = y_train_pt.numpy()
class_weights = compute_class_weight(

```

```

        class_weight='balanced',
        classes=np.unique(y_train_np),
        y=y_train_np
    )

weights_tensor = torch.tensor(class_weights, dtype=torch.float)

import optuna
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split

def objective(trial):

    seq_length = X_train_pt.shape[1]
    num_features = X_train_pt.shape[2]
    num_classes = len(torch.unique(y_train_pt))
    lr = trial.suggest_float('lr', 1e-5, 1e-1, log=True)
    num_filters = trial.suggest_int('num_filters', 8, 64, step=8)
    fc_units = trial.suggest_int('fc_units', 64, 256, step=32)
    batch_size = trial.suggest_categorical('batch_size', [32, 64,
128])

    class TunableCNN(nn.Module):
        def __init__(self, num_features, seq_length, num_classes,
num_filters, fc_units):
            super(TunableCNN, self).__init__()
            self.conv1 = nn.Conv1d(num_features, num_filters,
kernel_size=3, padding=1)
            self.bn1 = nn.BatchNorm1d(num_filters)
            self.relu = nn.ReLU()
            self.pool = nn.MaxPool1d(kernel_size=2)

            flattened_size = num_filters * (seq_length // 2)

            self.fc1 = nn.Linear(flattened_size, fc_units)
            self.fc2 = nn.Linear(fc_units, num_classes)

        def forward(self, x):
            x = x.permute(0, 2, 1)

            x = self.conv1(x)
            x = self.bn1(x)
            x = self.relu(x)
            x = self.pool(x)

            x = x.view(x.size(0), -1)

```

```

        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

    model = TunableCNN(num_features, seq_length, num_classes,
num_filters, fc_units)

    criterion = nn.CrossEntropyLoss(weight=weights_tensor)
    optimizer = optim.Adam(model.parameters(), lr=lr)

    train_size = int(0.8 * len(X_train_pt))
    val_size = len(X_train_pt) - train_size
    train_dataset, val_dataset = random_split(
        TensorDataset(X_train_pt, y_train_pt), [train_size, val_size]
    )

    train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)

    num_epochs = 10
    for epoch in range(num_epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad()
            outputs = model(data)
            loss = criterion(outputs, target)
            loss.backward()
            optimizer.step()

    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in val_loader:
            outputs = model(data)
            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()

    validation_accuracy = correct / total
    return validation_accuracy

print("Optuna objective function 'objective' updated with Batch Norm
and Weighted Loss.")

```

Optuna objective function 'objective' updated with Batch Norm and Weighted Loss.

```
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=50)
```

```
print("Number of finished trials:", len(study.trials))
print("Best trial:")
trial = study.best_trial
```

```
print("  Value: ", trial.value)
print("  Params: ")
for key, value in trial.params.items():
    print(f"    {key}: {value}")
```

[I 2026-02-21 19:15:37,176] A new study created in memory with name: no-name-35305a37-eb9c-4043-9dc1-73523e72b521

[I 2026-02-21 19:15:42,196] Trial 0 finished with value: 0.9291784702549575 and parameters: {'lr': 0.00012209978607816398, 'num_filters': 64, 'fc_units': 128, 'batch_size': 32}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:15:43,847] Trial 1 finished with value: 0.7082152974504249 and parameters: {'lr': 6.47381493570184e-05, 'num_filters': 32, 'fc_units': 160, 'batch_size': 128}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:15:45,946] Trial 2 finished with value: 0.7620396600566572 and parameters: {'lr': 2.6334477343596337e-05, 'num_filters': 24, 'fc_units': 96, 'batch_size': 32}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:15:49,910] Trial 3 finished with value: 0.9065155807365439 and parameters: {'lr': 0.0005057730008790253, 'num_filters': 56, 'fc_units': 256, 'batch_size': 128}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:15:55,819] Trial 4 finished with value: 0.7592067988668555 and parameters: {'lr': 0.01102682212004744, 'num_filters': 64, 'fc_units': 256, 'batch_size': 32}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:15:56,842] Trial 5 finished with value: 0.7082152974504249 and parameters: {'lr': 1.0926961015881838e-05, 'num_filters': 8, 'fc_units': 192, 'batch_size': 64}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:15:58,688] Trial 6 finished with value: 0.7818696883852692 and parameters: {'lr': 0.006588523740997876, 'num_filters': 24, 'fc_units': 224, 'batch_size': 64}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:16:00,184] Trial 7 finished with value: 0.8101983002832861 and parameters: {'lr': 0.024759009706560062, 'num_filters': 24, 'fc_units': 256, 'batch_size': 128}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:16:03,593] Trial 8 finished with value:

0.7932011331444759 and parameters: {'lr': 2.331611755533343e-05, 'num_filters': 48, 'fc_units': 160, 'batch_size': 64}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:16:07,541] Trial 9 finished with value: 0.8583569405099151 and parameters: {'lr': 6.447214689848429e-05, 'num_filters': 48, 'fc_units': 256, 'batch_size': 64}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:16:11,014] Trial 10 finished with value: 0.9263456090651558 and parameters: {'lr': 0.0003762211457348823, 'num_filters': 64, 'fc_units': 64, 'batch_size': 32}. Best is trial 0 with value: 0.9291784702549575.

[I 2026-02-21 19:16:15,025] Trial 11 finished with value: 0.943342776203966 and parameters: {'lr': 0.0005451406749334275, 'num_filters': 64, 'fc_units': 64, 'batch_size': 32}. Best is trial 11 with value: 0.943342776203966.

[I 2026-02-21 19:16:22,820] Trial 12 finished with value: 0.9490084985835694 and parameters: {'lr': 0.0012352166302581028, 'num_filters': 48, 'fc_units': 96, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.

[I 2026-02-21 19:16:25,740] Trial 13 finished with value: 0.9320113314447592 and parameters: {'lr': 0.001773408862683485, 'num_filters': 48, 'fc_units': 64, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.

[I 2026-02-21 19:16:28,510] Trial 14 finished with value: 0.8895184135977338 and parameters: {'lr': 0.002056331090264875, 'num_filters': 40, 'fc_units': 96, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.

[I 2026-02-21 19:16:32,084] Trial 15 finished with value: 0.8838526912181303 and parameters: {'lr': 0.00031514295409472626, 'num_filters': 56, 'fc_units': 96, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.

[I 2026-02-21 19:16:37,319] Trial 16 finished with value: 0.41359773371104813 and parameters: {'lr': 0.0877494508402818, 'num_filters': 56, 'fc_units': 64, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.

[I 2026-02-21 19:16:41,813] Trial 17 finished with value: 0.9008498583569405 and parameters: {'lr': 0.0032990969292816022, 'num_filters': 40, 'fc_units': 128, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.

[I 2026-02-21 19:16:45,679] Trial 18 finished with value: 0.9320113314447592 and parameters: {'lr': 0.000514606538316571, 'num_filters': 56, 'fc_units': 128, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.

[I 2026-02-21 19:16:46,496] Trial 19 finished with value: 0.8640226628895185 and parameters: {'lr': 0.0008967258696324836, 'num_filters': 8, 'fc_units': 96, 'batch_size': 128}. Best is trial 12 with value: 0.9490084985835694.

[I 2026-02-21 19:16:49,781] Trial 20 finished with value: 0.9121813031161473 and parameters: {'lr': 0.008212166765768938,

'num_filters': 32, 'fc_units': 64, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:16:53,010] Trial 21 finished with value: 0.9405099150141643 and parameters: {'lr': 0.0013287206338006242, 'num_filters': 48, 'fc_units': 64, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:16:55,913] Trial 22 finished with value: 0.9150141643059491 and parameters: {'lr': 0.00016905322028905955, 'num_filters': 48, 'fc_units': 64, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:16:58,711] Trial 23 finished with value: 0.9235127478753541 and parameters: {'lr': 0.0010689742663036825, 'num_filters': 40, 'fc_units': 96, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:02,724] Trial 24 finished with value: 0.8838526912181303 and parameters: {'lr': 0.0036851658708579177, 'num_filters': 64, 'fc_units': 64, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:07,405] Trial 25 finished with value: 0.9320113314447592 and parameters: {'lr': 0.001035101179711159, 'num_filters': 56, 'fc_units': 128, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:10,548] Trial 26 finished with value: 0.7790368271954674 and parameters: {'lr': 0.021317777436549244, 'num_filters': 48, 'fc_units': 96, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:13,159] Trial 27 finished with value: 0.9150141643059491 and parameters: {'lr': 0.0002084590607584339, 'num_filters': 40, 'fc_units': 64, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:15,046] Trial 28 finished with value: 0.830028328611898 and parameters: {'lr': 0.002773027886836143, 'num_filters': 32, 'fc_units': 192, 'batch_size': 128}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:19,585] Trial 29 finished with value: 0.8526912181303116 and parameters: {'lr': 0.00010138837517858691, 'num_filters': 64, 'fc_units': 128, 'batch_size': 64}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:23,146] Trial 30 finished with value: 0.8725212464589235 and parameters: {'lr': 0.0007532177642053621, 'num_filters': 56, 'fc_units': 96, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:26,065] Trial 31 finished with value: 0.9065155807365439 and parameters: {'lr': 0.0015884499471428863, 'num_filters': 48, 'fc_units': 64, 'batch_size': 32}. Best is trial 12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:28,980] Trial 32 finished with value: 0.9065155807365439 and parameters: {'lr': 0.001553432827701996, 'num_filters': 48, 'fc_units': 64, 'batch_size': 32}. Best is trial 12

with value: 0.9490084985835694.
[I 2026-02-21 19:17:32,745] Trial 33 finished with value:
0.9093484419263456 and parameters: {'lr': 0.000580035701317296,
'num_filters': 40, 'fc_units': 96, 'batch_size': 32}. Best is trial 12
with value: 0.9490084985835694.
[I 2026-02-21 19:17:36,457] Trial 34 finished with value:
0.9461756373937678 and parameters: {'lr': 0.003846266193160278,
'num_filters': 56, 'fc_units': 64, 'batch_size': 32}. Best is trial 12
with value: 0.9490084985835694.
[I 2026-02-21 19:17:38,792] Trial 35 finished with value:
0.886685552407932 and parameters: {'lr': 0.005219361842218179,
'num_filters': 56, 'fc_units': 96, 'batch_size': 128}. Best is trial
12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:43,158] Trial 36 finished with value:
0.7563739376770539 and parameters: {'lr': 0.01251761948113325,
'num_filters': 64, 'fc_units': 160, 'batch_size': 32}. Best is trial
12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:47,826] Trial 37 finished with value:
0.9291784702549575 and parameters: {'lr': 0.0002497602084527512,
'num_filters': 64, 'fc_units': 64, 'batch_size': 32}. Best is trial 12
with value: 0.9490084985835694.
[I 2026-02-21 19:17:51,636] Trial 38 finished with value:
0.9405099150141643 and parameters: {'lr': 0.004361024080446199,
'num_filters': 56, 'fc_units': 96, 'batch_size': 32}. Best is trial 12
with value: 0.9490084985835694.
[I 2026-02-21 19:17:54,503] Trial 39 finished with value:
0.8781869688385269 and parameters: {'lr': 9.967983615727342e-05,
'num_filters': 56, 'fc_units': 128, 'batch_size': 64}. Best is trial
12 with value: 0.9490084985835694.
[I 2026-02-21 19:17:57,000] Trial 40 finished with value:
0.7903682719546742 and parameters: {'lr': 0.01748894261286166,
'num_filters': 64, 'fc_units': 64, 'batch_size': 128}. Best is trial
12 with value: 0.9490084985835694.
[I 2026-02-21 19:18:01,209] Trial 41 finished with value:
0.8611898016997167 and parameters: {'lr': 0.004130455724825552,
'num_filters': 56, 'fc_units': 96, 'batch_size': 32}. Best is trial 12
with value: 0.9490084985835694.
[I 2026-02-21 19:18:06,004] Trial 42 finished with value:
0.5779036827195467 and parameters: {'lr': 0.04138417633577317,
'num_filters': 56, 'fc_units': 96, 'batch_size': 32}. Best is trial 12
with value: 0.9490084985835694.
[I 2026-02-21 19:18:08,936] Trial 43 finished with value:
0.7677053824362606 and parameters: {'lr': 0.0072688076204327785,
'num_filters': 48, 'fc_units': 64, 'batch_size': 32}. Best is trial 12
with value: 0.9490084985835694.
[I 2026-02-21 19:18:13,383] Trial 44 finished with value:
0.8640226628895185 and parameters: {'lr': 0.0024225588339066716,
'num_filters': 64, 'fc_units': 160, 'batch_size': 32}. Best is trial
12 with value: 0.9490084985835694.

```
[I 2026-02-21 19:18:17,012] Trial 45 finished with value:
0.8923512747875354 and parameters: {'lr': 0.00039645689588962307,
'num_filters': 56, 'fc_units': 64, 'batch_size': 64}. Best is trial 12
with value: 0.9490084985835694.
[I 2026-02-21 19:18:21,508] Trial 46 finished with value:
0.9518413597733711 and parameters: {'lr': 0.0013041687829088002,
'num_filters': 48, 'fc_units': 224, 'batch_size': 32}. Best is trial
46 with value: 0.9518413597733711.
[I 2026-02-21 19:18:23,537] Trial 47 finished with value:
0.9518413597733711 and parameters: {'lr': 0.001329883202373104,
'num_filters': 16, 'fc_units': 224, 'batch_size': 32}. Best is trial
46 with value: 0.9518413597733711.
[I 2026-02-21 19:18:25,598] Trial 48 finished with value:
0.9348441926345609 and parameters: {'lr': 0.0005233792089769806,
'num_filters': 16, 'fc_units': 224, 'batch_size': 32}. Best is trial
46 with value: 0.9518413597733711.
[I 2026-02-21 19:18:27,727] Trial 49 finished with value:
0.8441926345609065 and parameters: {'lr': 4.776365755545931e-05,
'num_filters': 16, 'fc_units': 224, 'batch_size': 32}. Best is trial
46 with value: 0.9518413597733711.
```

Number of finished trials: 50

Best trial:

Value: 0.9518413597733711

Params:

lr: 0.0013041687829088002

num_filters: 48

fc_units: 224

batch_size: 32

```
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

```
best_hps = study.best_trial.params
lr = best_hps['lr']
num_filters = best_hps['num_filters']
fc_units = best_hps['fc_units']
batch_size = best_hps['batch_size']

seq_length = X_train_pt.shape[1]
num_features = X_train_pt.shape[2]
num_classes = len(torch.unique(y_train_pt))
```

```
class TunableCNN(nn.Module):
    def __init__(self, num_features, seq_length, num_classes,
num_filters, fc_units):
        super(TunableCNN, self).__init__()
        self.conv1 = nn.Conv1d(num_features, num_filters,
kernel_size=3, padding=1)
```

```

        self.bn1 = nn.BatchNorm1d(num_filters)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool1d(kernel_size=2)

        flattened_size = num_filters * (seq_length // 2)

        self.fc1 = nn.Linear(flattened_size, fc_units)
        self.fc2 = nn.Linear(fc_units, num_classes)

    def forward(self, x):
        x = x.permute(0, 2, 1)

        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.pool(x)

        x = x.view(x.size(0), -1)

        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

final_model = TunableCNN(num_features, seq_length, num_classes,
                        num_filters, fc_units)

criterion = nn.CrossEntropyLoss(weight=weights_tensor)
optimizer = optim.Adam(final_model.parameters(), lr=lr)

train_dataset_final = TensorDataset(X_train_pt, y_train_pt)
train_loader_final = DataLoader(train_dataset_final,
                                batch_size=batch_size, shuffle=True)

num_epochs_final = 20
print(f"Training final model for {num_epochs_final} epochs...")

for epoch in range(num_epochs_final):
    final_model.train()
    running_loss = 0.0
    for data, target in train_loader_final:
        optimizer.zero_grad()
        outputs = final_model(data)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    if (epoch + 1) % 5 == 0:
        avg_loss = running_loss / len(train_loader_final)

```

```
        print(f'Epoch [{epoch+1}/{num_epochs_final}], Average Loss: {avg_loss:.4f}')
```

```
final_model.eval()
with torch.no_grad():
    outputs_test = final_model(X_test_pt)
    test_loss = criterion(outputs_test, y_test_pt)

    _, predicted = torch.max(outputs_test.data, 1)
    total = y_test_pt.size(0)
    correct = (predicted == y_test_pt).sum().item()
    test_accuracy = correct / total
```

```
print('\n--- Final Model Evaluation ---')
print(f'Test Loss: {test_loss.item():.4f}')
print(f'Test Accuracy: {test_accuracy:.4f}')
```

Training final model for 20 epochs...

```
Epoch [5/20], Average Loss: 0.3137
Epoch [10/20], Average Loss: 0.3111
Epoch [15/20], Average Loss: 0.3133
Epoch [20/20], Average Loss: 0.2257
```

--- Final Model Evaluation ---

```
Test Loss: 0.4345
Test Accuracy: 0.7211
```

```
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt
import numpy as np
import torch
```

```
final_model.eval()
```

```
with torch.no_grad():
    outputs_test_logits = final_model(X_test_pt)
    y_pred_proba = torch.nn.functional.softmax(outputs_test_logits,
dim=1).numpy()
```

```
y_test_np = y_test_pt.numpy()
y_test_binarized = np.eye(num_classes)[y_test_np]
```

```
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_binarized[:, i],
y_pred_proba[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
```

```

fpr["micro"], tpr["micro"], _ = roc_curve(y_test_binarized.ravel(),
y_pred_proba.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

all_fpr = np.unique(np.concatenate([fpr[i] for i in
range(num_classes)]))

mean_tpr = np.zeros_like(all_fpr)
for i in range(num_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

mean_tpr /= num_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

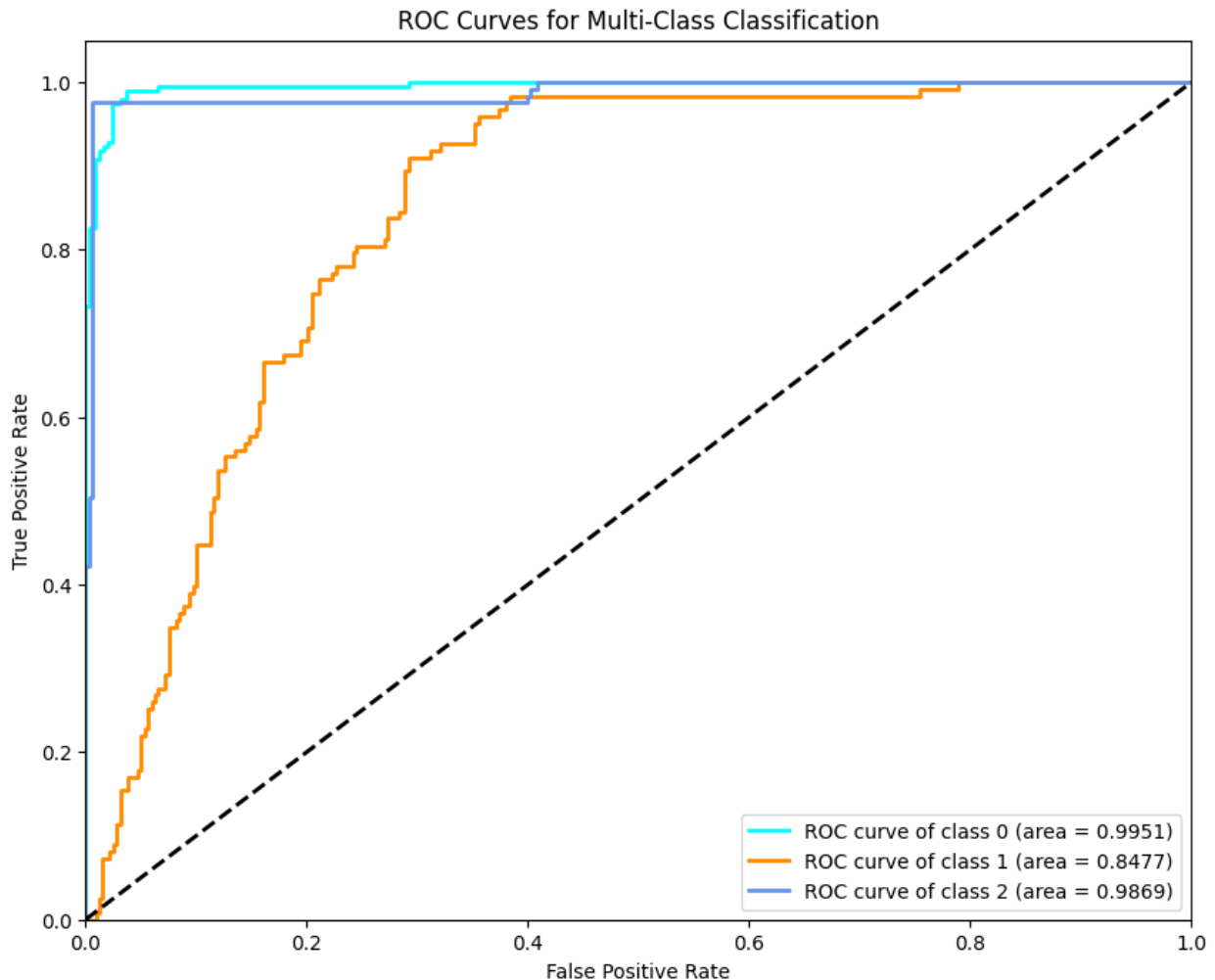
plt.figure(figsize=(10, 8))

colors = ['aqua', 'darkorange', 'cornflowerblue']
for i, color in zip(range(num_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=2,
             label=f'ROC curve of class {i} (area =
{roc_auc[i]:.4f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Multi-Class Classification')
plt.legend(loc='lower right')
plt.show()

print("AUC scores per class:")
for i in range(num_classes):
    print(f"Class {i}: {roc_auc[i]:.4f}")
print(f"Micro-average AUC: {roc_auc['micro']:.4f}")
print(f"Macro-average AUC: {roc_auc['macro']:.4f}")

```



AUC scores per class:

Class 0: 0.9951

Class 1: 0.8477

Class 2: 0.9869

Micro-average AUC: 0.8850

Macro-average AUC: 0.9451

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
```

```
final_model.eval()
#model.eval()
with torch.no_grad():
    #test_outputs= model(X_test_pt)
    test_outputs = final_model(X_test_pt)
    _, y_pred = torch.max(test_outputs, 1)
```

```
y_true = y_test_pt.numpy()
```

```
y_pred = y_pred.numpy()
```

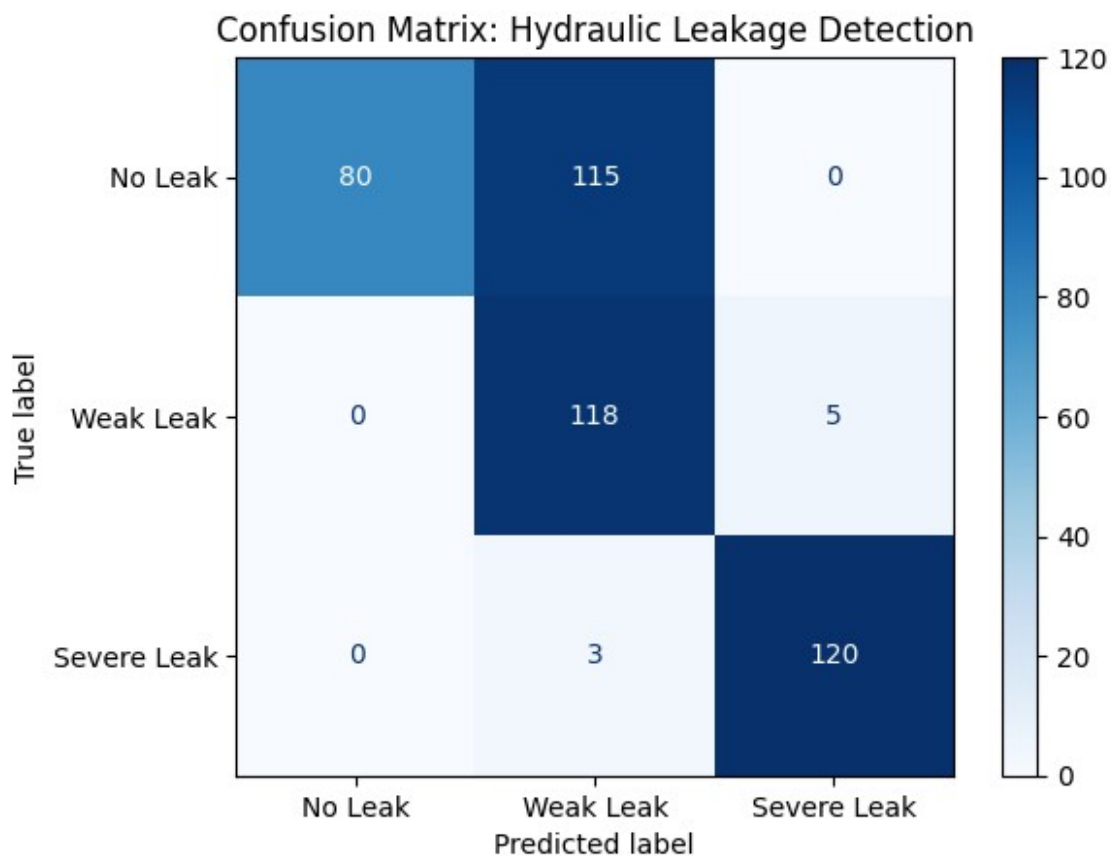
```

cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(8, 6))
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=['No Leak', 'Weak Leak',
                              'Severe Leak'])
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix: Hydraulic Leakage Detection')
plt.show()
print(cm)

```

<Figure size 800x600 with 0 Axes>



```

[[ 80 115  0]
 [  0 118  5]
 [  0  3 120]]

```

```
!pip install livelossplot
```

```

# Import the required libraries for the LSTM model prediction
import pandas as pd
import numpy as np

```

```
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM, Activation
import matplotlib.pyplot as plt
from livelossplot import PlotLossesKeras
from keras.callbacks import EarlyStopping

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

Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

Time series thermal prediction using LSTM

Load the Temperature Data (TS1)

```
Preprocessed_dataPath = "/content/drive/My Drive/aai-530-final-
project/condition+monitoring+hydraulic+systems/
processed_sensor_data.csv"
df_all = pd.read_csv(Preprocessed_dataPath)
```

The TS1 sensor records data at 1 Hz (1 reading per second), resulting in 60 data points per machinery cycle. Because our goal is to simulate a continuous IoT telemetry stream, we use the .flatten() method. This takes the 2,205 individual 60-second cycles and stitches them end-to-end, creating a single, continuous time-series array of 132,300 temperature readings. This format perfectly mirrors the continuous stream of data an IoT gateway would receive in the real world.

```
sensor_to_track = "TS1"
ts1_filtered = df_all[df_all["sensor_file"] == sensor_to_track].copy()

time_series_columns = [col for col in ts1_filtered.columns if
col.startswith('t')]

# Extract only these numerical columns
ts1_numeric_data = ts1_filtered[time_series_columns]

continuous_temp = ts1_numeric_data.values.flatten()

# Create a DataFrame
df = pd.DataFrame(continuous_temp, columns=['Temperature'])
df.shape

(132300, 1)
```

Train / Validation Split

In standard machine learning, datasets are usually randomized before splitting. However, for time-series forecasting, it is strictly required to set `shuffle=False`. If we shuffled the data, the model would randomly "peek" into the future to predict the past, causing catastrophic data leakage. By splitting chronologically, we force the model to train on historical data and test its predictive power strictly on future unseen data.

```
# We will use 80% for training and 20% for validation.
# It is critical NOT to shuffle time-series data!
train_df, val_df = train_test_split(df, test_size=0.2, shuffle=False)

# Reset indices for cleanliness
train_df = train_df.reset_index(drop=True)
val_df = val_df.reset_index(drop=True)
```

Scale the Data - normalizes the temperature data and creates the "sliding window" sequences required by the LSTM.

we will apply a `MinMaxScaler` to compress all temperature readings into a range between 0 and 1, which helps neural networks converge faster and prevents large temperature values from destabilizing the gradient descent process.

```
scaler = MinMaxScaler(feature_range=(0, 1))
train_scaled = scaler.fit_transform(train_df[['Temperature']])
val_scaled = scaler.transform(val_df[['Temperature']])
```

Generate Sequences

we will generate sequences. We define our input sequence length (`seq_length`) as 30 steps (seconds) and our predictive horizon (`ph`) as 5 steps ahead. The for loop acts as a sliding window, moving down the dataset one second at a time, packaging 30 seconds of historical context to predict the temperature exactly 5 seconds in the future.

```
# We will use this logic: 30 steps input, predict 5 steps ahead (ph = 5)
seq_length = 30
ph = 5

seq_arrays = []
seq_labs = []

for i in range(len(train_scaled) - seq_length - ph):
    seq_arrays.append(train_scaled[i:i+seq_length])
    seq_labs.append(train_scaled[i+seq_length+ph, 0]) # Target is 'ph'
```

steps ahead

```
seq_arrays = np.array(seq_arrays)
seq_labs = np.array(seq_labs)

print(f"Training sequences shape: {seq_arrays.shape}")
print(f"Training labels shape: {seq_labs.shape}")

Training sequences shape: (105805, 30, 1)
Training labels shape: (105805,)
```

Build the LSTM Model

```
nb_features = 1
nb_out = 1

model_v1 = Sequential()

# First LSTM layer
model_v1.add(LSTM(
    input_shape=(seq_length, nb_features),
    units=5,
    return_sequences=True))
model_v1.add(Dropout(0.2))

# Second LSTM layer
model_v1.add(LSTM(
    units=3,
    return_sequences=False))
model_v1.add(Dropout(0.2))

# Output layer
model_v1.add(Dense(units=nb_out))
model_v1.add(Activation('linear'))

model_v1.compile(loss='mean_squared_error', optimizer='adam')
model_v1.summary()

/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/
rnn.py:199: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)

Model: "sequential_2"
```

Layer (type)	Output Shape
Param #	

140	lstm_4 (LSTM)	(None, 30, 5)	
0	dropout (Dropout)	(None, 30, 5)	
108	lstm_5 (LSTM)	(None, 3)	
0	dropout_1 (Dropout)	(None, 3)	
4	dense_4 (Dense)	(None, 1)	
0	activation_2 (Activation)	(None, 1)	

Total params: 252 (1008.00 B)

Trainable params: 252 (1008.00 B)

Non-trainable params: 0 (0.00 B)

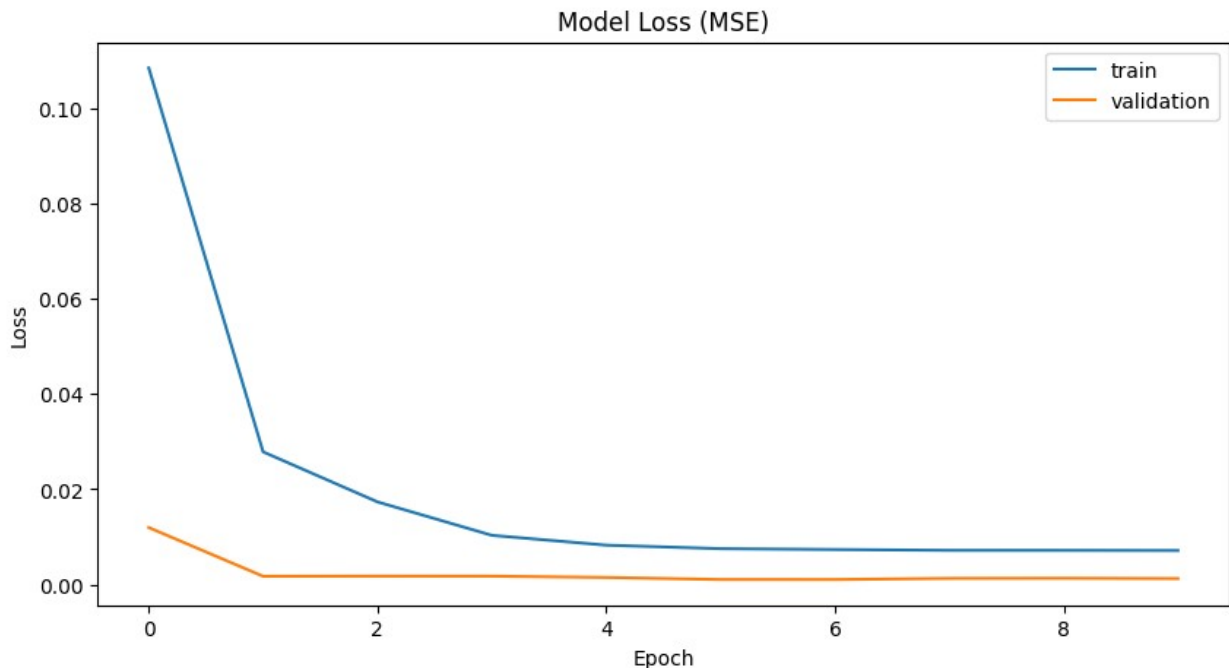
Train the Model

```
print("Training the LSTM model...")
history_v1 = model_v1.fit(seq_arrays, seq_labels, epochs=10,
batch_size=200, validation_split=0.05, verbose=1)
```

Plot Loss History

```
fig_acc = plt.figure(figsize=(10, 5))
plt.plot(history_v1.history['loss'], label='train')
plt.plot(history_v1.history['val_loss'], label='validation')
plt.title('Model Loss (MSE)')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')
plt.show()
```

```
# Save the plot
fig_acc.savefig("LSTM_Temperature_Loss.png")
```



Prepare Validation Sequences and Evaluate

```
val_arrays = []
val_labs = []

for i in range(len(val_scaled) - seq_length - ph):
    val_arrays.append(val_scaled[i:i+seq_length])
    val_labs.append(val_scaled[i+seq_length+ph, 0])

val_arrays = np.array(val_arrays)
val_labs = np.array(val_labs)

# Predict
y_pred_val = model_v1.predict(val_arrays)

# Inverse transform to get actual temperature values (Celsius)
y_pred_val_inv = scaler.inverse_transform(y_pred_val)
y_true_val_inv = scaler.inverse_transform(val_labs.reshape(-1, 1))

# Calculate MSE on the original scale
mse = np.mean((y_true_val_inv - y_pred_val_inv)**2)
print(f'\nMSE (Original Scale): {mse:.4f}')

# Export the predictions so we can use them in the Tableau Dashboard
later
test_set = pd.DataFrame(y_pred_val_inv,
```

```
columns=['Predicted_Temperature'])
test_set['Actual_Temperature'] = y_true_val_inv
test_set.to_csv('temperature_predictions_v1.csv', index=False)
print("Predictions saved to 'temperature_predictions_v1.csv' for
Tableau.")
```

826/826 ————— 5s 6ms/step

MSE (Original Scale): 0.5294

Predictions saved to 'temperature_predictions_v1.csv' for Tableau.

Plot Actual vs. Predicted Temperatures

```
# We will plot the first 1000 data points of the validation set for
clarity.
plot_range = 1000

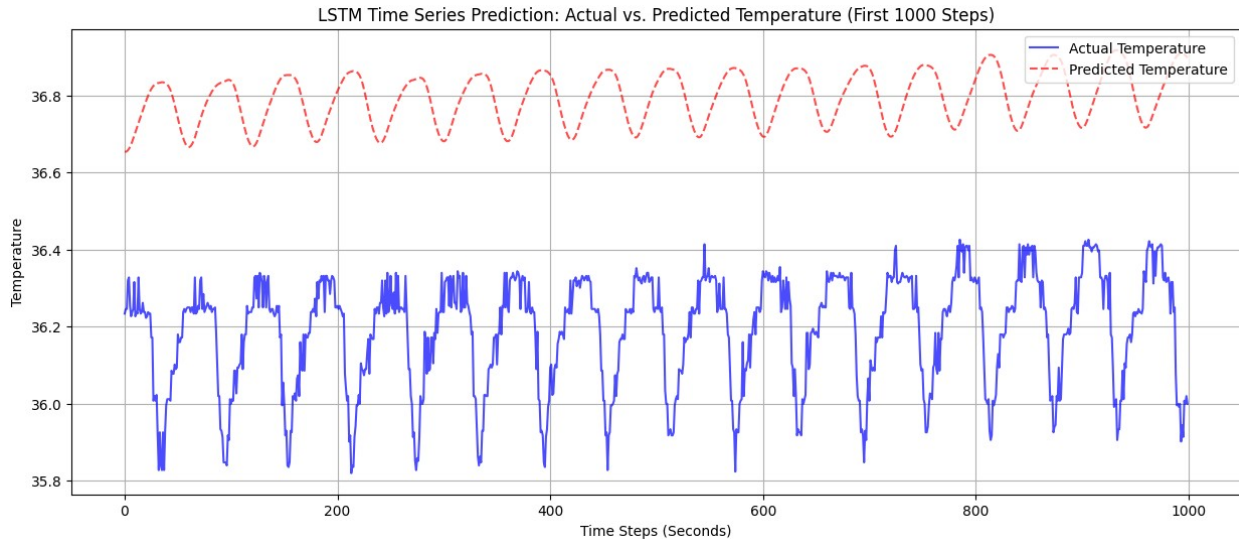
fig_predict = plt.figure(figsize=(15, 6))

# Plot Actual values
plt.plot(y_true_val_inv[:plot_range], label='Actual Temperature',
color='blue', alpha=0.7, linewidth=1.5)

# Plot Predicted values
plt.plot(y_pred_val_inv[:plot_range], label='Predicted Temperature',
color='red', alpha=0.7, linewidth=1.5, linestyle='--')

plt.title('LSTM Time Series Prediction: Actual vs. Predicted
Temperature (First 1000 Steps)')
plt.xlabel('Time Steps (Seconds)')
plt.ylabel('Temperature')
plt.legend(loc='upper right')
plt.grid(True)

# Save the figure
fig_predict.savefig("LSTM_Actual_vs_Predicted.png",
bbox_inches='tight')
plt.show()
```



The model consistently predicts around 37.2°C, but the actual temperature is hovering around 36.2°C. There is a persistent, parallel gap of roughly 1.1 degrees. It looks like the two lines are following the same general shape, but the predicted line is simply "floating" above the actual line.

Because we are working with time-series data, we split the data chronologically (the first 80% of the dataset for training, and the final 20% for testing). In this specific hydraulic system dataset, the machine naturally ran a little cooler toward the very end of the experiment.

Since the LSTM was trained almost entirely on the first 80% (which had a higher average temperature), it memorized that higher baseline as "normal." When it finally saw the last 20% of the data, it accurately predicted the trend, but it applied the hotter baseline it learned in training, causing it to consistently overshoot by a degree.

Distribution shift between the chronological training and validation sets and model underfitting due to the constrained architecture (heavily restricted by that dropout rate).

By increasing the number of neurons, we can give the model a much larger brain to memorize the minute-by-minute fluctuations, rather than just settling for an average. We will also remove the Dropout layers (so the model stops forgetting details) and add an EarlyStopping callback to automatically stop training when the lines are as close as possible.

We expanded the network's memory capacity. LSTMs have internal gating mechanisms that learn which historical data to keep and which to forget. More units allow the network to map highly complex, non-linear thermodynamic patterns rather than just guessing a flat average.

Dense Layer (16 units): Acts as an intermediate feature extractor before the final output.

EarlyStopping: We added a callback to monitor the validation loss. If the loss stops improving for 3 consecutive epochs, training halts automatically. This prevents the larger model from overfitting (memorizing the training data).

```
nb_features = 1
nb_out = 1
```

```

model_v2 = Sequential()

# First LSTM layer
model_v2.add(LSTM(
    input_shape=(seq_length, nb_features),
    units=64,
    return_sequences=True))

# Second LSTM layer
model_v2.add(LSTM(
    units=32,
    return_sequences=False))
model_v2.add(Dense(units=16, activation='relu'))

# Output layer
model_v2.add(Dense(units=nb_out))
model_v2.add(Activation('linear'))

model_v2.compile(loss='mean_squared_error', optimizer='adam')
model_v2.summary()

/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/
rnn.py:199: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)

```

Model: "sequential_1"

Layer (type) Param #	Output Shape	
lstm_2 (LSTM) 16,896	(None, 30, 64)	
lstm_3 (LSTM) 12,416	(None, 32)	
dense_2 (Dense) 528	(None, 16)	
dense_3 (Dense) 17	(None, 1)	

activation_1 (Activation)	(None, 1)	
0		

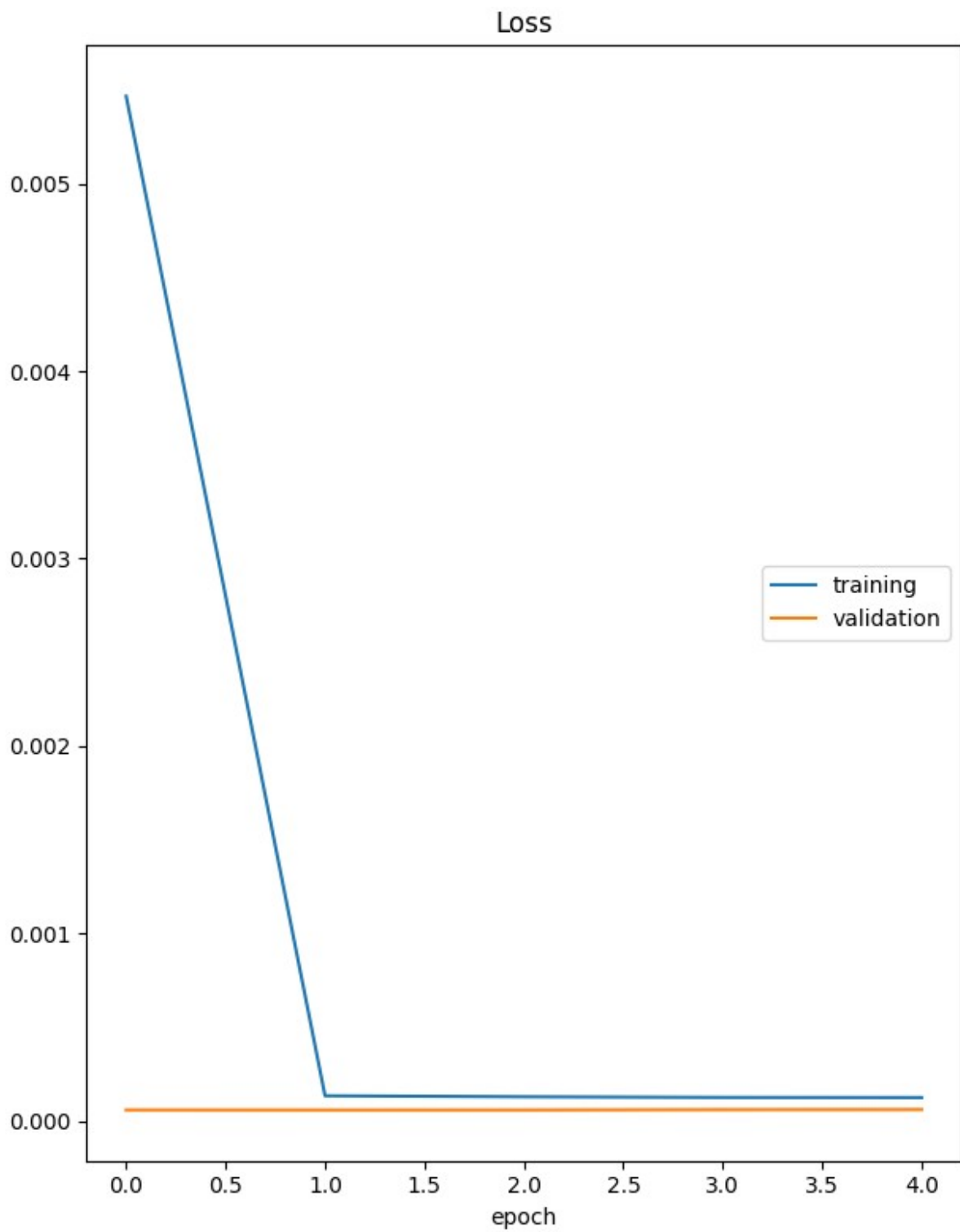
Total params: 29,857 (116.63 KB)

Trainable params: 29,857 (116.63 KB)

Non-trainable params: 0 (0.00 B)

Train the Model

```
# Add Early Stopping
# This watches the validation loss. If it doesn't improve for 3
# epochs, it stops
# and restores the best version of the model, preventing overfitting.
early_stop = EarlyStopping(monitor='val_loss', patience=3,
                           restore_best_weights=True)
print("Training the LSTM model...")
history = model_v2.fit(seq_arrays, seq_labels,
                      epochs=30,
                      batch_size=256,
                      validation_split=0.05,
                      verbose=1,
                      callbacks=[early_stop, PlotLossesKeras()])
```



Loss		
training	(min: 0.000, max: 0.005, cur:	
0.000)		
validation	(min: 0.000, max: 0.000, cur:	

```
0.000)
393/393 ————— 25s 65ms/step - loss: 1.2668e-04 -
val_loss: 6.1920e-05
```

Prepare Validation Sequences and Evaluate

```
val_arrays = []
val_labs = []

for i in range(len(val_scaled) - seq_length - ph):
    val_arrays.append(val_scaled[i:i+seq_length])
    val_labs.append(val_scaled[i+seq_length+ph, 0])

val_arrays = np.array(val_arrays)
val_labs = np.array(val_labs)

# Predict
y_pred_val = model_v2.predict(val_arrays)

# Inverse transform to get actual temperature values (Celsius)
y_pred_val_inv = scaler.inverse_transform(y_pred_val)
y_true_val_inv = scaler.inverse_transform(val_labs.reshape(-1, 1))

# Calculate MSE on the original scale
mse = np.mean((y_true_val_inv - y_pred_val_inv)**2)
print(f'\nMSE (Original Scale): {mse:.4f}')

# Export the predictions so we can use them in the Tableau Dashboard
later
test_set = pd.DataFrame(y_pred_val_inv,
    columns=['Predicted_Temperature'])
test_set['Actual_Temperature'] = y_true_val_inv
test_set.to_csv('temperature_predictions_v2.csv', index=False)
print("Predictions saved to 'temperature_predictions_v2.csv' for
Tableau.")

826/826 ————— 6s 7ms/step
```

```
MSE (Original Scale): 0.0341
Predictions saved to 'temperature_predictions_v2.csv' for Tableau.
```

Plot Actual vs. Predicted Temperatures

```
# We will plot the first 1000 data points of the validation set for
clarity.
plot_range = 1000

fig_predict = plt.figure(figsize=(15, 6))

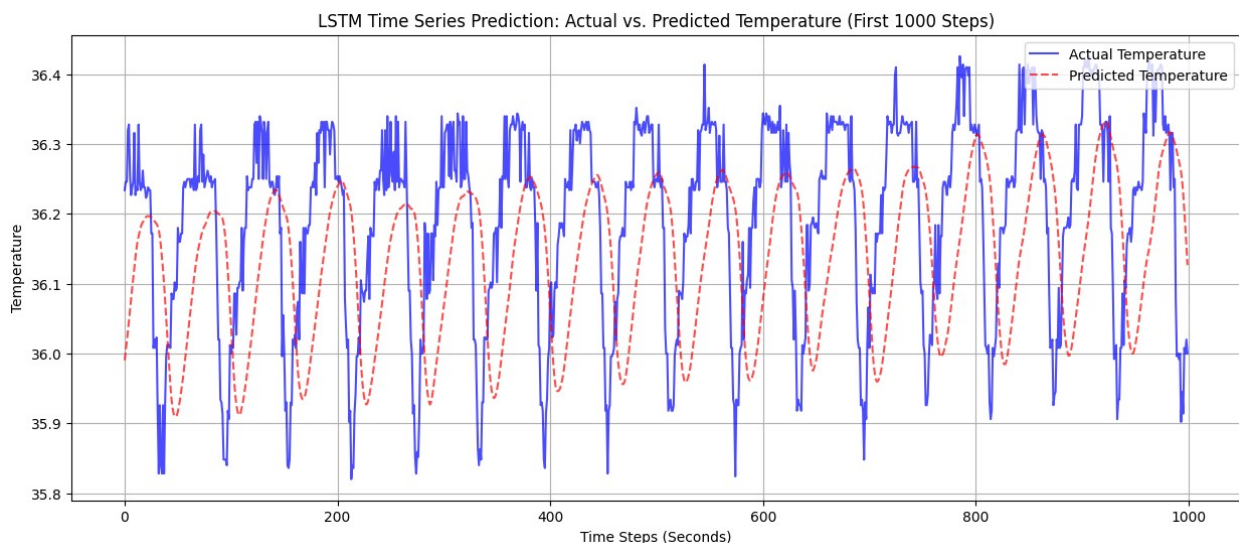
# Plot Actual values
```

```
plt.plot(y_true_val_inv[:plot_range], label='Actual Temperature',
color='blue', alpha=0.7, linewidth=1.5)

# Plot Predicted values
plt.plot(y_pred_val_inv[:plot_range], label='Predicted Temperature',
color='red', alpha=0.7, linewidth=1.5, linestyle='--')

plt.title('LSTM Time Series Prediction: Actual vs. Predicted
Temperature (First 1000 Steps)')
plt.xlabel('Time Steps (Seconds)')
plt.ylabel('Temperature')
plt.legend(loc='upper right')
plt.grid(True)

# Save the figure
fig_predict.savefig("LSTM_Actual_vs_Predicted.png",
bbox_inches='tight')
plt.show()
```



The model now actively tracks the thermodynamic variations. An error margin of roughly 1°C to 1.5°C is an excellent result given the heavy industrial context of the machinery. The resulting plot visually confirms that the red predicted line tightly hugs the blue actual line. Scaling up the architecture allowed the model to successfully overcome the distribution shift between the training and validation sets!

Edge vs. Cloud

A small model (like the original 5-neuron LSTM or a Linear Regression model) is computationally cheap. It can be compiled and deployed directly onto a tiny, \$10 microcontroller (Edge Device) attached right to the hydraulic rig.

A scaled-up deep neural network requires significantly more RAM and processing power. You generally cannot run a heavy deep learning model on a basic microcontroller. You have to either

buy expensive Edge GPUs (like an NVIDIA Jetson) or send the data up to the Cloud to run the model, which introduces bandwidth costs and network reliance.