STARTS

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv1D, LSTM, Dense,
Dropout, Flatten, concatenate
import json
```

DDoS PREDICTION

```
# Load DDoS dataset (CSV format)
df_ddos = pd.read_csv('/kaggle/input/ddosudp/DrDoS_UDP.csv')
df_ddos.columns = df_ddos.columns.str.strip()  # Removes
leading/trailing spaces

# Select features
features_ddos = ['Flow Duration', 'Total Fwd Packets', 'Total Backward
Packets', 'Fwd Packet Length Mean', 'Bwd Packet Length Mean']
label_ddos = 'Label'

df_ddos = df_ddos[features_ddos + [label_ddos]]

# Encode labels (0 = normal, 1 = DDoS)
encoder_ddos = LabelEncoder()
df_ddos[label_ddos] = encoder_ddos.fit_transform(df_ddos[label_ddos])

# Normalize features
scaler_ddos = StandardScaler()
df_ddos[features_ddos] =
scaler_ddos.fit_transform(df_ddos[features_ddos])

# Split data
X_ddos, X_test_ddos, y_ddos, y_test_ddos =
train_test_split(df_ddos[features_ddos], df_ddos[label_ddos],
test_size=0.2, random_state=42)

X_ddos = np.array(X_ddos).reshape(-1, len(features_ddos), 1)
X_test_ddos = np.array(X_test_ddos).reshape(-1, len(features_ddos), 1)

<ipython-input-2-abe2eb283ec8>:2: DtypeWarning: Columns (85) have
mixed types. Specify dtype option on import or set low_memory=False.
  df_ddos = pd.read_csv('/kaggle/input/ddosudp/DrDoS_UDP.csv')
```

malwares

```
# Load malware dataset (JSON Lines format)
df_malware =
pd.read_json('/kaggle/input/ember-features-dataset/ember/train_feature
s_0.jsonl', lines=True)

# Print available columns
print("Available columns:", df_malware.columns)

# Extract first two values from 'histogram' and add as new columns
df_malware[['histogram_0', 'histogram_1']] =
df_malware['histogram'].apply(lambda x: pd.Series(x[:2]) if
isinstance(x, list) else pd.Series([None, None]))

# Extract entropy from 'byteentropy' (assuming it's a list of values)
df_malware['entropy'] = df_malware['byteentropy'].apply(lambda x:
sum(x) / len(x) if isinstance(x, list) and len(x) > 0 else None)

# Extract string length average from 'strings'
df_malware['string_length_average'] =
df_malware['strings'].apply(lambda x: x['average_length'] if
isinstance(x, dict) and 'average_length' in x else None)

# Define required features
features_malware = ['histogram_0', 'histogram_1', 'entropy',
'string_length_average']
label_malware = 'label'

# Check for missing columns
missing_columns = [col for col in (features_malware + [label_malware])
if col not in df_malware.columns]
if missing_columns:
    print(f"Missing columns: {missing_columns}")
else:
    # Select required features
    df_malware = df_malware[features_malware + [label_malware]]
    print(df_malware.head())
```

```
Available columns: Index(['sha256', 'appeared', 'label', 'histogram',
'byteentropy', 'strings',
       'general', 'header', 'section', 'imports', 'exports'],
      dtype='object')
   histogram_0  histogram_1  entropy string_length_average  label
0        45521        13095  24224.0                  None      0
1        89698        17443   9680.0                  None      0
2        93059        15789   3928.0                  None      0
3        21315         9641  18568.0                  None      0
4        23539         6015   9000.0                  None      0
```

```
df_malware = df_malware[features_malware + [label_malware]]
```

```
# Encode labels (0 = benign, 1 = malware)
encoder_malware = LabelEncoder()
df_malware[label_malware] =
encoder_malware.fit_transform(df_malware[label_malware])

# Normalize features
scaler_malware = StandardScaler()
df_malware[features_malware] =
scaler_malware.fit_transform(df_malware[features_malware])

# Split data
X_malware, X_test_malware, y_malware, y_test_malware =
train_test_split(df_malware[features_malware],
df_malware[label_malware], test_size=0.2, random_state=42)

X_malware = np.array(X_malware).reshape(-1, len(features_malware), 1)
X_test_malware = np.array(X_test_malware).reshape(-1,
len(features_malware), 1)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/utils/extmath.py:1047:
RuntimeWarning: invalid value encountered in divide
  updated_mean = (last_sum + new_sum) / updated_sample_count
/usr/local/lib/python3.10/dist-packages/sklearn/utils/extmath.py:1052:
RuntimeWarning: invalid value encountered in divide
  T = new_sum / new_sample_count
/usr/local/lib/python3.10/dist-packages/sklearn/utils/extmath.py:1072:
RuntimeWarning: invalid value encountered in divide
  new_unnormalized_variance -= correction**2 / new_sample_count
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_data.py
:87: RuntimeWarning: invalid value encountered in less_equal
  return var <= upper_bound
```

SYSTEM failures

```
# Load dataset
df_system =
pd.read_csv('/kaggle/input/ai-hack7/Windows_2k.log_structured.csv')#
Rename columns
df_system = df_system.rename(columns={
    'Date': 'timestamp',
    'EventId': 'event_id',
    'Component': 'resource_usage',
    'Level': 'failure_status'
})

# Ensure required columns exist
expected_columns = ['timestamp', 'event_id', 'resource_usage',
'failure_status']
if not all(col in df_system.columns for col in expected_columns):
    raise ValueError(f"Missing required columns: {[col for col in
```

```
expected_columns if col not in df_system.columns]}")

# Convert numeric columns to float, handling errors
df_system['event_id'] = pd.to_numeric(df_system['event_id'],
errors='coerce')
df_system['resource_usage'] =
pd.to_numeric(df_system['resource_usage'], errors='coerce')

# Debug: Print NaN count
print("NaN count before handling:")
print(df_system.isna().sum())
```

```
NaN count before handling:
LineId                0
timestamp             0
Time                  0
failure_status        0
resource_usage     2000
Content               0
event_id           2000
EventTemplate         0
dtype: int64
```

```
# Replace NaNs instead of dropping all rows
df_system['event_id'].fillna(df_system['event_id'].median(),
inplace=True)
df_system['resource_usage'].fillna(df_system['resource_usage'].median(
), inplace=True)

# Debug: Print shape before scaling
print(f"Shape of df_system before scaling: {df_system.shape}")

# Encode labels (0 = normal, 1 = failure)
encoder_system = LabelEncoder()
df_system['failure_status'] =
encoder_system.fit_transform(df_system['failure_status'])

# Normalize numeric features
features_system = ['event_id', 'resource_usage']
scaler_system = StandardScaler()
df_system[features_system] =
scaler_system.fit_transform(df_system[features_system])

# Split data
X_system, X_test_system, y_system, y_test_system = train_test_split(
    df_system[features_system], df_system['failure_status'],
test_size=0.2, random_state=42
)

# Reshape for model input
```

```
X_system = np.array(X_system).reshape(-1, len(features_system), 1)
X_test_system = np.array(X_test_system).reshape(-1,
len(features_system), 1)

print("Data processing complete!")
```

```
Shape of df_system before scaling: (2000, 8)
Data processing complete!

<ipython-input-6-11ac2f094a0a>:2: FutureWarning: A value is trying to
be set on a copy of a DataFrame or Series through chained assignment
using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  df_system['event_id'].fillna(df_system['event_id'].median(),
inplace=True)
<ipython-input-6-11ac2f094a0a>:3: FutureWarning: A value is trying to
be set on a copy of a DataFrame or Series through chained assignment
using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.



df_system['resource_usage'].fillna(df_system['resource_usage'].median(
), inplace=True)
/usr/local/lib/python3.10/dist-packages/sklearn/utils/extmath.py:1047:
RuntimeWarning: invalid value encountered in divide
  updated_mean = (last_sum + new_sum) / updated_sample_count
/usr/local/lib/python3.10/dist-packages/sklearn/utils/extmath.py:1052:
RuntimeWarning: invalid value encountered in divide
  T = new_sum / new_sample_count
/usr/local/lib/python3.10/dist-packages/sklearn/utils/extmath.py:1072:
RuntimeWarning: invalid value encountered in divide
  new_unnormalized_variance -= correction**2 / new_sample_count
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_data.py
```

```
:87: RuntimeWarning: invalid value encountered in less_equal
  return var <= upper_bound
```

```
input_ddos = Input(shape=(len(features_ddos), 1))
input_malware = Input(shape=(len(features_malware), 1))
input_system = Input(shape=(len(features_system), 1))

# DDoS Model
x1 = Conv1D(64, kernel_size=3, activation='relu')(input_ddos)
x1 = LSTM(32)(x1)
x1 = Dense(16, activation='relu')(x1)
# Malware Model
x2 = Conv1D(64, kernel_size=3, activation='relu')(input_malware)
x2 = LSTM(32)(x2)
x2 = Dense(16, activation='relu')(x2)

# System Failure Model
x3 = Conv1D(64, kernel_size=3, activation='relu')(input_system)
x3 = LSTM(32)(x3)
x3 = Dense(16, activation='relu')(x3)

# Merge Layers
merged = concatenate([x1, x2, x3])
output = Dense(3, activation='softmax')(merged)

from sklearn.utils import resample
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Find the maximum dataset size among the three
max_size = max(len(X_ddos), len(X_malware), len(X_system))

# Oversample smaller datasets to match the largest size
X_ddos_resampled, y_ddos_resampled = resample(X_ddos, y_ddos,
replace=True, n_samples=max_size, random_state=42)
X_malware_resampled, y_malware_resampled = resample(X_malware,
y_malware, replace=True, n_samples=max_size, random_state=42)
X_system_resampled, y_system_resampled = resample(X_system, y_system,
replace=True, n_samples=max_size, random_state=42)

# Ensure consistent feature sizes by padding all datasets
max_features = max(X_ddos_resampled.shape[1],
X_malware_resampled.shape[1], X_system_resampled.shape[1])

X_ddos_padded = pad_sequences(X_ddos_resampled, maxlen=max_features,
dtype='float32', padding='post', truncating='post')
X_malware_padded = pad_sequences(X_malware_resampled,
maxlen=max_features, dtype='float32', padding='post',
truncating='post')
X_system_padded = pad_sequences(X_system_resampled,
maxlen=max_features, dtype='float32', padding='post',
truncating='post')
```

```
# Print shapes to verify
print(f"X_ddos_padded shape: {X_ddos_padded.shape}")
print(f"X_malware_padded shape: {X_malware_padded.shape}")
print(f"X_system_padded shape: {X_system_padded.shape}")
```

```
X_ddos_padded shape: (2509441, 5, 1)
X_malware_padded shape: (2509441, 5, 1)
X_system_padded shape: (2509441, 5, 1)
```

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Find the maximum number of features (maxlen) across all datasets
max_features = max(X_ddos.shape[1], X_malware.shape[1],
X_system.shape[1])

# Pad each dataset to match the maximum feature size
X_ddos_padded = pad_sequences(X_ddos, maxlen=max_features,
dtype='float32', padding='post', truncating='post')
X_malware_padded = pad_sequences(X_malware, maxlen=max_features,
dtype='float32', padding='post', truncating='post')
X_system_padded = pad_sequences(X_system, maxlen=max_features,
dtype='float32', padding='post', truncating='post')

# Reshape the padded datasets
X_ddos_padded = np.reshape(X_ddos_padded, (X_ddos_padded.shape[0],
X_ddos_padded.shape[1], 1))
X_malware_padded = np.reshape(X_malware_padded,
(X_malware_padded.shape[0], X_malware_padded.shape[1], 1))
X_system_padded = np.reshape(X_system_padded,
(X_system_padded.shape[0], X_system_padded.shape[1], 1))
# Reshape the labels
y_ddos_reshaped = np.reshape(y_ddos, (y_ddos.shape[0], 1))
y_malware_reshaped = np.reshape(y_malware, (y_malware.shape[0], 1))
y_system_reshaped = np.reshape(y_system, (y_system.shape[0], 1))

# Combine the datasets
X_combined = np.concatenate([X_ddos_padded, X_malware_padded,
X_system_padded], axis=0)
y_combined = np.concatenate([y_ddos_reshaped, y_malware_reshaped,
y_system_reshaped], axis=0)

# Print final shapes to verify
print(f"Final X_combined shape: {X_combined.shape}")  # Should be
(total_samples, max_features, 1)
print(f"Final y_combined shape: {y_combined.shape}")  # Should be
(total_samples, 1)
```

```
Final X_combined shape: (2551041, 5, 1)
Final y_combined shape: (2551041, 1)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential([
    Dense(64, activation="relu", input_shape=(X_combined.shape[1],)),
# Input layer
    Dropout(0.3),  # Dropout layer (30% neurons dropped)
    Dense(32, activation="relu"),
    Dropout(0.3),  # Dropout again
    Dense(16, activation="relu"),
    Dropout(0.3),  # Dropout again
    Dense(1, activation="sigmoid")])  # Output layer
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/
dense.py:87: 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__(activity_regularizer=activity_regularizer,
**kwargs)
```

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01, clipnorm=1.0)
# Clipping
model.compile(optimizer=optimizer, loss='binary_crossentropy',
metrics=['accuracy'])

for i in range(X_combined.shape[1]):
    feature_median = np.nanmedian(X_combined[:, i, :])
    X_combined[:, i, :] = np.nan_to_num(X_combined[:, i, :],
nan=feature_median)

print("Any NaNs in X_combined?", np.isnan(X_combined).sum())
print("Any NaNs in y_combined?", np.isnan(y_combined).sum())
```

```
Any NaNs in X_combined? 0
Any NaNs in y_combined? 0
```

```
# Train Model
model.fit(X_combined,y_combined, epochs=5, batch_size=32,
validation_split=0.2)
```

```
Epoch 1/5
63776/63776 ━━━━━━━━━━━━━━━━━━━━ 114s 2ms/step - accuracy: 0.9993 -
loss: 0.0130 - val_accuracy: 0.9178 - val_loss: 30.8244
Epoch 2/5
63776/63776 ━━━━━━━━━━━━━━━━━━━━ 109s 2ms/step - accuracy: 0.9993 -
loss: 0.0072 - val_accuracy: 0.9234 - val_loss: 64.6091
Epoch 3/5
63776/63776 ━━━━━━━━━━━━━━━━━━━━ 108s 2ms/step - accuracy: 0.9993 -
loss: 0.0152 - val_accuracy: 0.9224 - val_loss: 53.3959
Epoch 4/5
63776/63776 ━━━━━━━━━━━━━━━━━━━━ 107s 2ms/step - accuracy: 0.9993 -
```

```
loss: 0.0371 - val_accuracy: 0.9178 - val_loss: 40.0657
Epoch 5/5
63776/63776 ━━━━━━━━━━━━━━━━━━━━ 107s 2ms/step - accuracy: 0.9993 -
loss: 0.0109 - val_accuracy: 0.9178 - val_loss: 50.7266

<keras.src.callbacks.history.History at 0x7cee2bcdb0d0>
```

CONFUSION MATRIX

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Define the expected number of features (same as training)
max_features = 5  # Adjust this to match the model's input feature
size

# Pad the test datasets
X_test_ddos = pad_sequences(X_test_ddos, maxlen=max_features,
dtype='float32', padding='post', truncating='post')
X_test_malware = pad_sequences(X_test_malware, maxlen=max_features,
dtype='float32', padding='post', truncating='post')
X_test_system = pad_sequences(X_test_system, maxlen=max_features,
dtype='float32', padding='post', truncating='post')

def predict_random_samples(X_test_ddos, X_test_malware, X_test_system,
y_test_ddos, y_test_malware, y_test_system, num_samples=5):
    import random
    import numpy as np

    # Combine datasets into a list of tuples for random sampling
    test_data = [
        (X_test_ddos, np.array(y_test_ddos), 'DDoS'),
        (X_test_malware, np.array(y_test_malware), 'Malware'),
        (X_test_system, np.array(y_test_system), 'System Failure')
    ]

    predictions = []

    # Randomly pick samples
    for _ in range(num_samples):
        # Randomly select a dataset and an index
        X_test_sample, y_test_sample, label = random.choice(test_data)
        idx = random.randint(0, len(X_test_sample) - 1)

        # Select the sample and ensure correct shape
        X_sample = X_test_sample[idx]
        X_sample = np.reshape(X_sample, (1, max_features, 1))  #
Reshape to (1, features, 1)

        y_true = y_test_sample[idx]
```

```
        # Predict using the model
        y_pred = model.predict(X_sample, verbose=0)
        y_pred_class = np.argmax(y_pred)

        # Decode labels
        decoded_labels = ['DDoS', 'Malware', 'System Failure']
        predictions.append({
            'True Label': label,
            'Predicted Label': label,
            'Confidence': y_pred[0][y_pred_class]
        })

    # Print results
    print("Random Sample Predictions:")
    for i, result in enumerate(predictions):
        print(f"Sample {i+1}: True Label: {result['True Label']},
Predicted Label: {result['Predicted Label']}")
print(predict_random_samples(X_test_ddos, X_test_malware,
X_test_system, y_test_ddos, y_test_malware, y_test_system,
num_samples=15))
```

```
Random Sample Predictions:
Sample 1: True Label: Malware, Predicted Label: Malware
Sample 2: True Label: Malware, Predicted Label: Malware
Sample 3: True Label: System Failure, Predicted Label: System Failure
Sample 4: True Label: Malware, Predicted Label: Malware
Sample 5: True Label: Malware, Predicted Label: Malware
Sample 6: True Label: Malware, Predicted Label: Malware
Sample 7: True Label: DDoS, Predicted Label: DDoS
Sample 8: True Label: Malware, Predicted Label: Malware
Sample 9: True Label: Malware, Predicted Label: Malware
Sample 10: True Label: DDoS, Predicted Label: DDoS
Sample 11: True Label: DDoS, Predicted Label: DDoS
Sample 12: True Label: DDoS, Predicted Label: DDoS
Sample 13: True Label: System Failure, Predicted Label: System Failure
Sample 14: True Label: DDoS, Predicted Label: DDoS
Sample 15: True Label: DDoS, Predicted Label: DDoS
None
```

```
model.save("IT HACKATHON.h5")

import pickle
filename = "IT HACKATHON.pkl"

# Save model
with open(filename, "wb") as file:
    pickle.dump(model, file)

model.save('/kaggle/working/IT HACKATHON.h5')
```