Configuration Manual

MSc Research Project
Data Analytics

Tejas Sandeep Bafna
Student ID: x23211741

School of Computing
National College of Ireland

Supervisor: William Clifford

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Tejas Sandeep Bafna
Student ID:	X23211741
Programme:	Master's in data Analytics **Year:** 2024-2025
Module:	MSc. Research Project
Lecturer:	William Clifford
Submission Due Date:	12/12/2024
Project Title:	Detecting Adversarial Network Behaviors in IoT Environment
Word Count:	1798 **Page Count:** 24

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Tejas Sandeep Bafna

Date: 12/12/2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Tejas Sandeep Bafna
X23211741

1. Introduction

This configuration guide explains step to step how to set up and duplicate the system we engineered for the detection of IoT adversarial network behaviors and is therefore not exhaustive in detailing system requirements and library installations, data preprocessing, training, evaluation, and visualization.

2. System Requirements and Libraries

This section provides the details of Software and Hardware requirements to implement the research done.

Category	Requirement/Library
Operating System	Windows, macOS, or Linux
Processor	Intel Core i5 or higher
RAM	8 GB or higher
Storage	Minimum 10 GB free disk space
Python Version	Python 3.8 or higher
Libraries	
- numpy	For numerical computations
- pandas	For data manipulation and analysis
- matplotlib	For creating static, animated, and interactive visualizations
- seaborn	For statistical data visualization
- scikit-learn	For preprocessing, model building, and evaluation
- imblearn	For handling imbalanced datasets (SMOTE)
- tensorflow	For building and training deep learning models
- keras	For high-level deep learning API
- warnings	For suppressing warnings during execution
- collections	For counting occurrences in datasets
Other Tools	Jupyter Notebook (optional) for running code interactively

3. Data and its Execution

3.1 Importing Libraries and Modules

We start by importing the essential libraries needed for data processing, visualization, machine learning, and deep learning. Below is the breakdown:

- warnings: Suppresses warnings to keep the output clean.
- numpy: Handles numerical operations and arrays.
- pandas: Manages datasets and tabular data structures like CSV files.
- seaborn: Visualizes data with attractive and informative graphs.
- tensorflow: Builds and trains deep learning models.
- matplotlib.pyplot: Plots graphs for data visualization.
- collections.Counter: Counts occurrences of elements in a dataset.
- sklearn: Provides tools for preprocessing, dimensionality reduction, and evaluation.
- imblearn.SMOTE: Balances imbalanced datasets using oversampling techniques.

TensorFlow Keras Layers and Model Functions:

- layers: Includes different layer types for deep learning models (like Dense, Conv1D).
- Sequential: Combines layers sequentially to build models.
- load_model: Loads pre-trained models for reuse.

Preprocessing and Data Transformation Tools:

- LabelEncoder and OneHotEncoder: Convert categorical labels into numerical formats.
- StandardScaler: Standardizes features by removing the mean and scaling to unit variance.
- to_categorical: Converts class vectors into binary class matrices for categorical classification tasks.

Step 1: Importing Libraries and Modules

First, the libraries were imported to make easy execution of various tasks. Libraries that were used in handling data and machine learning included numpy, pandas, tensorflow, and sklearn.

Visualization libraries such as matplotlib and seaborn allowed for producing some really insightful graphs. Packages like warnings and collections were applied towards efficient scripting and debugging. That's the foundational step to ensure that the environment has been set up for subsequent analysis and modeling.

```python
import warnings
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from collections import Counter
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from sklearn.decomposition import PCA
from tensorflow.keras import Sequential
from imblearn.over_sampling import SMOTE
from tensorflow.keras.models import load_model
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.models import Sequential
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import classification_report, confusion_matrix
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense, Dropout, BatchNormalization, Lea


plt.rcParams['figure.dpi'] = 300
warnings.filterwarnings('ignore')
```

Figure 1: Importing all the necessary libraries

3.2 Dataset Information

We load the dataset named RT_IOT2022.csv using the pandas library. This dataset contains IoT network traffic data which we will analyze and use for detecting adversarial network behaviors.

Step 2: Loading the Dataset

The dataset was named RT_IOT2022.csv and loaded using the pandas library. This dataset contained IoT network traffic data that is important for analysis and detection of adversarial behaviors. The first five rows of the dataset were displayed by using head() to confirm a sucessful load

```python
# Load the dataset
data = pd.read_csv("RT_IOT2022.csv" )

# Display a message and the first few rows of the data
print("The first 5 rows of the IoT network traffic dataset are:")

display(data.head())
```

Figure 2: Loading the data into 'data' variable and then displaying it

We print a message indicating that we are showing the first few rows of the dataset. Then, we use the head() function to display the first five rows for inspection.

The first 5 rows of the IoT network traffic dataset are:

	no	id.orig_p	id.resp_p	proto	service	flow_duration	fwd_pkts_tot	bwd_pkts_tot	fwd_data_pkts_tot	bwd_
0	0	38667	1883	tcp	mqtt	32.011598	9	5	3	
1	1	51143	1883	tcp	mqtt	31.883584	9	5	3	
2	2	44761	1883	tcp	mqtt	32.124053	9	5	3	
3	3	60893	1883	tcp	mqtt	31.961063	9	5	3	
4	4	51087	1883	tcp	mqtt	31.902362	9	5	3	

5 rows × 85 columns

Figure 3: First few rows of the dataset

3.3 Statistical Analysis

Step 3: Statistical Analysis

Several statistical summaries were conducted to understand the dataset's structure:

- Dimensions: The shape function was used to retrieve the number of rows (samples) and columns (features).
- Column Names: columns provided a list of feature names for exploration.
- Data Types: The dtypes function identified whether columns contained numerical, categorical, or other types of data.
- Dataset Summary: info() provided a detailed summary, including non-null counts and memory usage.
- Descriptive Statistics: describe() calculated metrics like mean, standard deviation, and range for numerical features.

```
# Display the shape of the dataset

print(f"The shape of the dataset is: {data.shape}")
```

Figure 4: Illustration of dataset dimensions showing the number of rows (data points) and columns (attributes).

```
The shape of the dataset is: (123117, 85)
```

```
# Display the columns of the dataset

print("Columns in the dataset:")

display(data.columns)
```

Figure 5: Illustration of the dataset's column names, representing the features and attributes available for analysis.

```
Columns in the dataset:

Index(['no', 'id.orig_p', 'id.resp_p', 'proto', 'service', 'flow_duration',
       'fwd_pkts_tot', 'bwd_pkts_tot', 'fwd_data_pkts_tot',
       'bwd_data_pkts_tot', 'fwd_pkts_per_sec', 'bwd_pkts_per_sec',
       'flow_pkts_per_sec', 'down_up_ratio', 'fwd_header_size_tot',
       'fwd_header_size_min', 'fwd_header_size_max', 'bwd_header_size_tot',
       'bwd_header_size_min', 'bwd_header_size_max', 'flow_FIN_flag_count',
       'flow_SYN_flag_count', 'flow_RST_flag_count', 'fwd_PSH_flag_count',
       'bwd_PSH_flag_count', 'flow_ACK_flag_count', 'fwd_URG_flag_count',
       'bwd_URG_flag_count', 'flow_CWR_flag_count', 'flow_ECE_flag_count',
       'fwd_pkts_payload.min', 'fwd_pkts_payload.max', 'fwd_pkts_payload.tot',
       'fwd_pkts_payload.avg', 'fwd_pkts_payload.std', 'bwd_pkts_payload.min',
       'bwd_pkts_payload.max', 'bwd_pkts_payload.tot', 'bwd_pkts_payload.avg',
       'bwd_pkts_payload.std', 'flow_pkts_payload.min',
       'flow_pkts_payload.max', 'flow_pkts_payload.tot',
       'flow_pkts_payload.avg', 'flow_pkts_payload.std', 'fwd_iat.min',
       'fwd_iat.max', 'fwd_iat.tot', 'fwd_iat.avg', 'fwd_iat.std',
       'bwd_iat.min', 'bwd_iat.max', 'bwd_iat.tot', 'bwd_iat.avg',
       'bwd_iat.std', 'flow_iat.min', 'flow_iat.max', 'flow_iat.tot',
       'flow_iat.avg', 'flow_iat.std', 'payload_bytes_per_second',
       'fwd_subflow_pkts', 'bwd_subflow_pkts', 'fwd_subflow_bytes',
       'bwd_subflow_bytes', 'fwd_bulk_bytes', 'bwd_bulk_bytes',
       'fwd_bulk_packets', 'bwd_bulk_packets', 'fwd_bulk_rate',
       'bwd_bulk_rate', 'active.min', 'active.max', 'active.tot', 'active.avg',
       'active.std', 'idle.min', 'idle.max', 'idle.tot', 'idle.avg',
       'idle.std', 'fwd_init_window_size', 'bwd_init_window_size',
       'fwd_last_window_size', 'Attack_type'],
      dtype='object')
```

```
# Display the data types of each column

print("Data types of each column:")

display(data.dtypes)
```

Figure 6: Illustration of the data types for each column in the dataset, showing whether they are numerical, categorical, or other types.

```
Data types of each column:

no                        int64
id.orig_p                 int64
id.resp_p                 int64
proto                    object
service                  object
                          ...
idle.std                float64
fwd_init_window_size      int64
bwd_init_window_size      int64
fwd_last_window_size      int64
Attack_type              object
Length: 85, dtype: object
```

```python
# Display the dataset information (number of entries, memory usage, etc.)

print("Dataset Information:")

display(data.info())
```

Figure 7: Illustration of the dataset summary, showing the number of rows, columns, non-null entries, data types, and memory usage.

```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 123117 entries, 0 to 123116
Data columns (total 85 columns):
 #   Column                   Non-Null Count   Dtype
---  ------                   --------------   -----
 0   no                       123117 non-null  int64
 1   id.orig_p                123117 non-null  int64
 2   id.resp_p                123117 non-null  int64
 3   proto                    123117 non-null  object
 4   service                  123117 non-null  object
 5   flow_duration            123117 non-null  float64
 6   fwd_pkts_tot             123117 non-null  int64
 7   bwd_pkts_tot             123117 non-null  int64
 8   fwd_data_pkts_tot        123117 non-null  int64
 9   bwd_data_pkts_tot        123117 non-null  int64
 10  fwd_pkts_per_sec         123117 non-null  float64
 11  bwd_pkts_per_sec         123117 non-null  float64
 12  flow_pkts_per_sec        123117 non-null  float64
 13  down_up_ratio            123117 non-null  float64
 14  fwd_header_size_tot      123117 non-null  int64
 15  fwd_header_size_min      123117 non-null  int64
 16  fwd_header_size_max      123117 non-null  int64
 17  bwd_header_size_tot      123117 non-null  int64
 18  bwd_header_size_min      123117 non-null  int64
...
 83  fwd_last_window_size     123117 non-null  int64
 84  Attack_type              123117 non-null  object
dtypes: float64(56), int64(26), object(3)
memory usage: 79.8+ MB
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

```python
# Display descriptive statistics for numerical columns

print("Descriptive Statistics of Numerical Columns:")

display(data.describe())
```

Figure 8: Illustration of descriptive statistics for numerical columns, including metrics such as mean, standard deviation, minimum, and maximum values.

Step 4: Handling Missing Values and Duplicates

To address data quality, missing values and duplicates were managed:

- The total count of missing values per column was calculated using isnull().sum(), sorted in descending order.
- Duplicates were identified using duplicated() and their count was displayed.

```
Descriptive Statistics of Numerical Columns:
```

	no	id.orig_p	id.resp_p	flow_duration	fwd_pkts_tot	bwd_pkts_tot	fwd_data_pkts_tot
count	123117.000000	123117.000000	123117.000000	123117.000000	123117.000000	123117.000000	123117.000000
mean	37035.089248	34639.258738	1014.305092	3.809566	2.268826	1.909509	1.471218
std	30459.106367	19070.620354	5256.371994	130.005408	22.336565	33.018311	19.635196
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	6059.000000	17702.000000	21.000000	0.000001	1.000000	1.000000	1.000000
50%	33100.000000	37221.000000	21.000000	0.000004	1.000000	1.000000	1.000000
75%	63879.000000	50971.000000	21.000000	0.000005	1.000000	1.000000	1.000000
max	94658.000000	65535.000000	65389.000000	21728.335578	4345.000000	10112.000000	4345.000000

8 rows × 82 columns

```
# Check for missing values and display their sum in descending order

print("Sum of Null (Missing) Values in Each Column (Descending Order):")

missing_values = data.isnull().sum().sort_values(ascending=False)

print(missing_values)
```

Figure 9: Illustration of missing value counts for each column, sorted in descending order, highlighting columns with the highest data gaps.

```
Sum of Null (Missing) Values in Each Column (Descending Order):
no                          0
bwd_iat.std                 0
bwd_subflow_pkts            0
fwd_subflow_pkts            0
payload_bytes_per_second    0
                           ..
bwd_URG_flag_count          0
fwd_URG_flag_count          0
flow_ACK_flag_count         0
bwd_PSH_flag_count          0
Attack_type                 0
Length: 85, dtype: int64
```

```
# Check for exact duplicate rows

duplicates = data.duplicated().sum()

print(f"Number of Exact Duplicate Rows: {duplicates}")
```

Figure 10: Illustration of the total number of exact duplicate rows in the dataset, ensuring data quality and avoiding redundancy.

```
Number of Exact Duplicate Rows: 0
```

3.4 Exploratory Data Analysis

Step 5: Attack Type Distribution

The Attack_type column was analyzed to display the frequency of each attack category. This distribution was visualized using a bar plot (sns.countplot()), which highlighted the prevalence of different attack types and revealed potential class imbalances.

```
data["Attack_type"].value_counts()
```

```
Attack_type
DOS_SYN_Hping                 94659
Thing_Speak                    8108
ARP_poisioning                 7750
MQTT_Publish                   4146
NMAP_UDP_SCAN                  2590
NMAP_XMAS_TREE_SCAN            2010
NMAP_OS_DETECTION              2000
NMAP_TCP_scan                  1002
DDOS_Slowloris                  534
Wipro_bulb                      253
Metasploit_Brute_Force_SSH       37
NMAP_FIN_SCAN                    28
Name: count, dtype: int64
```

Figure 11: Illustration of the frequency distribution of attack types in the dataset, highlighting the prevalence of different attack categories.

```
# Attack type distribution

plt.figure(figsize=(10, 5))

sns.countplot(x='Attack_type', data=data)

plt.title("Distribution of Attack Types")

plt.xticks(rotation=90)

plt.show()
```

Figure 12: Visualization of the attack type distribution, showing the count of each attack type as a bar plot. This helps highlight class imbalances and the prevalence of specific attack types.

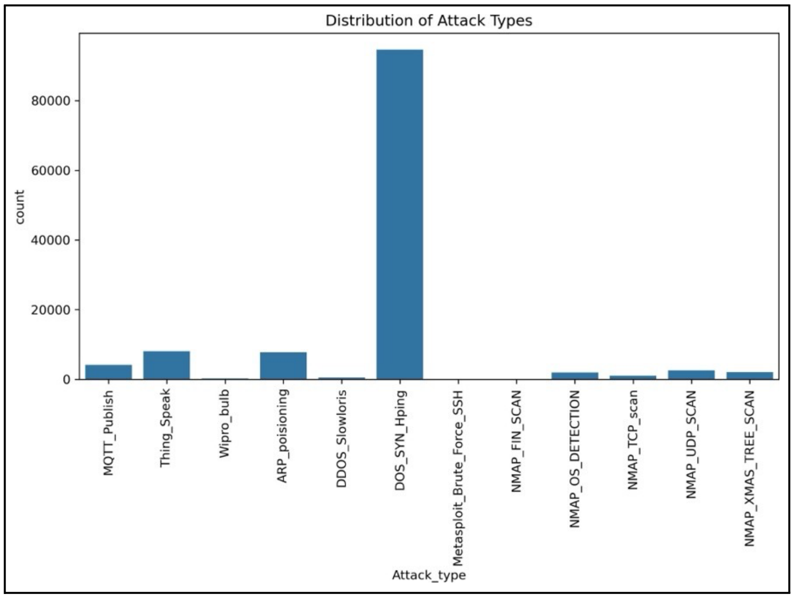


Figure 13: Visualization of the attack type distribution

Step 6: Protocol and Service Distributions

- Protocols: The proto column showed the frequency of different protocols in the dataset, visualized as a bar plot.
- Services: The service column displayed the usage frequency of various services, represented through a bar plot with rotated x-axis labels for readability.

```
# Protocol distribution

plt.figure(figsize=(8, 5))

sns.countplot(x='proto', data=data)

plt.title("Distribution of Protocols")

plt.show()
```

Figure 14: Visualization of protocol distribution in the dataset, showing the frequency of different protocols used in the IoT network as a bar plot.

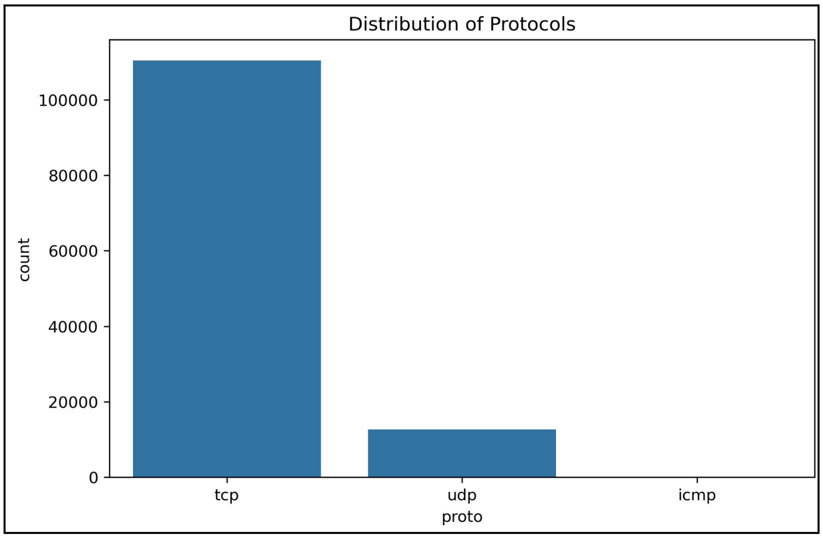


Figure 15: Visualization of protocol distribution in the dataset

```
# Service distribution

plt.figure(figsize=(8, 5))

sns.countplot(x='service', data=data)

plt.title("Distribution of Services")

plt.xticks(rotation=45)

plt.show()
```

Figure 16: Visualization of service distribution in the dataset, displaying the frequency of different services used in the IoT network as a bar plot.

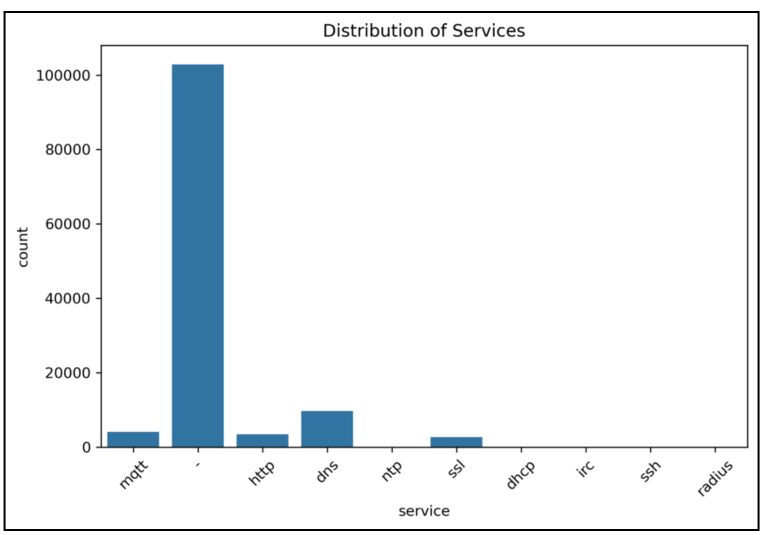


Figure 17: Visualization of service distribution in the dataset

3.5 PCA

Step 7: Principal Component Analysis (PCA)

PCA was applied to reduce the dataset's dimensionality while retaining 95% of its variance. A cumulative explained variance plot highlighted the required number of components

```python
# Select only numerical features from the dataset
numeric_data = data.select_dtypes(include=['float64', 'int64'])

# Standardize the data before applying PCA
scaler = StandardScaler()
scaled_data = scaler.fit_transform(numeric_data)

# Apply PCA and keep enough components to explain 95% of the variance
pca = PCA(n_components=0.95)
pca_result = pca.fit_transform(scaled_data)

# Display the explained variance ratio for each component
print("Explained variance ratio for each principal component:")
print(pca.explained_variance_ratio_)

# Cumulative explained variance plot
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(pca.explained_variance_ratio_) + 1),
         pca.explained_variance_ratio_.cumsum(), marker='o')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA - Cumulative Explained Variance')
plt.grid()
plt.show()

# Optional: Visualize the first two principal components
plt.figure(figsize=(8, 5))
plt.scatter(pca_result[:, 0], pca_result[:, 1], c=data['Attack_type'].astype('category').cat.codes)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA - Scatter Plot of First Two Principal Components')
plt.colorbar(label='Attack Type')
plt.show()
```

Figure 18: Cumulative explained variance plot showing how many principal components are needed to explain 95% of the variance in the dataset.

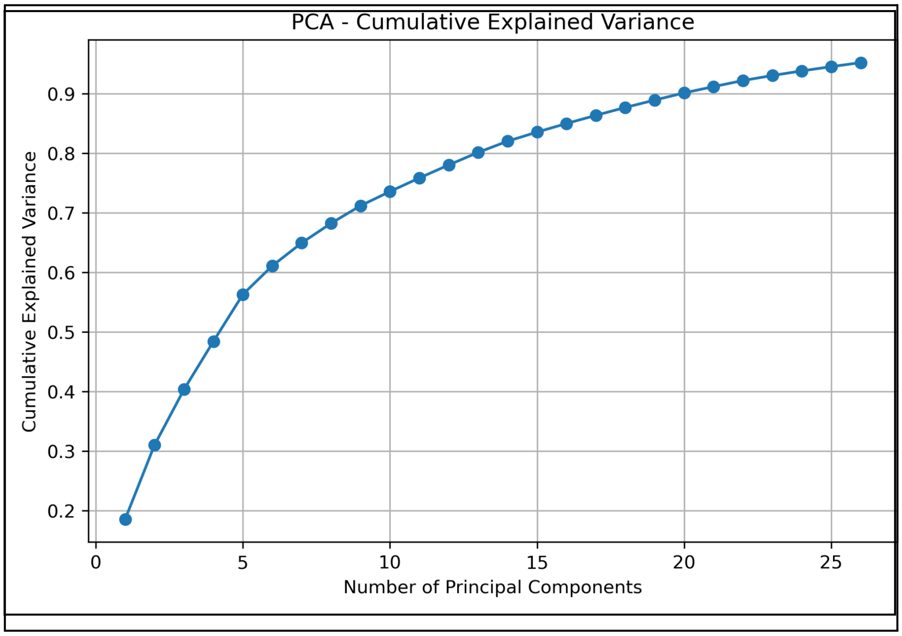


Figure 20: Scatter plot of the first two principal components, visualizing clusters and variations in the data based on attack types.

Step 8: Data Preparation Pipeline

The data was scaled, encoded, and reshaped for compatibility with the CNN model. This step ensured consistency across training, validation, and test datasets.

```python
# Identify and apply label encoding to object-type features (excluding target column)
label_encoders = {}
for column in data.select_dtypes(include=['object']).columns:
    if column != 'Attack_type':  # Avoid encoding the target label at this stage
        le = LabelEncoder()
        data[column] = le.fit_transform(data[column])
        label_encoders[column] = le  # Store the encoder for future use if needed

# Separate features and target
X = data.drop('Attack_type', axis=1)  # Features
y = data['Attack_type']  # Target (unencoded at this stage)

# Split the data into initial training (70%) and test (30%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

# Apply SMOTE to the initial training data
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

# Split the resampled training data into final training (80%) and validation (20%) sets
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train_resampled, y_train_resampled,
                                                              test_size=0.2, random_state=42, stratify=y_t

# Display the class distribution after SMOTE and splitting
print("Class distribution after applying SMOTE and splitting:")
print("Training set class distribution:")
print(pd.Series(y_train_final).value_counts())
print("\nValidation set class distribution:")
print(pd.Series(y_val).value_counts())
print("\nTest set class distribution:")
print(pd.Series(y_test).value_counts())
```

Figure 21: Class distribution in the final training set after applying SMOTE, illustrating balanced classes to mitigate biases during model training.

```
Class distribution after applying SMOTE and splitting:
Training set class distribution:
NMAP_XMAS_TREE_SCAN          53009
NMAP_UDP_SCAN                53009
DOS_SYN_Hping                53009
ARP_poisioning               53009
Thing_Speak                  53009
NMAP_FIN_SCAN                53009
Metasploit_Brute_Force_SSH   53009
Wipro_bulb                   53009
DDOS_Slowloris               53009
NMAP_OS_DETECTION            53008
NMAP_TCP_scan                53008
MQTT_Publish                 53008
Name: Attack_type, dtype: int64

Validation set class distribution:
NMAP_OS_DETECTION            13253
MQTT_Publish                 13253
NMAP_TCP_scan                13253
Thing_Speak                  13252
Wipro_bulb                   13252
NMAP_FIN_SCAN                13252
NMAP_UDP_SCAN                13252
NMAP_XMAS_TREE_SCAN          13252
...
Wipro_bulb                      76
Metasploit_Brute_Force_SSH      11
NMAP_FIN_SCAN                    8
Name: Attack_type, dtype: int64
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Figure 22: Class distribution in the validation and test sets, ensuring consistent evaluation metrics.

Data Preparation

```
# Apply standard scaling to the training, validation, and testing data separately
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_final)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

# Convert scaled features back to DataFrames for consistency
X_train_prepared = pd.DataFrame(X_train_scaled, columns=X.columns)
X_val_prepared = pd.DataFrame(X_val_scaled, columns=X.columns)
X_test_prepared = pd.DataFrame(X_test_scaled, columns=X.columns)

# Display the shapes of the prepared train, validation, and test sets with the specified format
print(f"Training features shape: {X_train_prepared.shape}")
print(f"Validation features shape: {X_val_prepared.shape}")
print(f"Testing features shape: {X_test_prepared.shape}")
print(f"Training labels shape: {y_train_final.shape}")
print(f"Validation labels shape: {y_val.shape}")
print(f"Testing labels shape: {y_test.shape}")

# Apply label encoding to the labels instead of one-hot encoding
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train_final)
y_val_encoded = label_encoder.transform(y_val)
y_test_encoded = label_encoder.transform(y_test)

# Determine the number of classes
num_classes = len(np.unique(y_train_encoded))

# Reshape data for CNN input (CNN expects 3D input: (samples, timesteps, features))
X_train_reshaped = np.expand_dims(X_train_prepared, axis=2)
X_val_reshaped = np.expand_dims(X_val_prepared, axis=2)
X_test_reshaped = np.expand_dims(X_test_prepared, axis=2)

# Display the shapes of the prepared train, validation, and test sets with the specified format for CNN mo
print(f"\nCNN Training features shape: {X_train_reshaped.shape}")
print(f"CNN Validation features shape: {X_val_reshaped.shape}")
print(f"CNN Testing features shape: {X_test_reshaped.shape}")
```

Figure 23: Data preparation pipeline, including scaling, label encoding, and reshaping, tailored for CNN model input.

```
Training features shape: (636105, 84)
Validation features shape: (159027, 84)
Testing features shape: (36936, 84)
Training labels shape: (636105,)
Validation labels shape: (159027,)
Testing labels shape: (36936,)

CNN Training features shape: (636105, 84, 1)
CNN Validation features shape: (159027, 84, 1)
CNN Testing features shape: (36936, 84, 1)
```

Figure 24: Data preparation pipeline

3.6 CNN Model

Step 9: Model Architecture

The CNN architecture included convolutional layers, batch normalization, max-pooling, and dropout layers, followed by dense layers with ReLU and softmax activations for multi-class classification.

```python
# Build the CNN model with added measures to prevent overfitting
CNN_model = Sequential([
    Conv1D(64, kernel_size=3, activation='relu', input_shape=(X_train_reshaped.shape[1], 1)),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),
    Dropout(0.4),
    Conv1D(128, kernel_size=3, activation='relu'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),
    Dropout(0.5),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.6),
    Dense(num_classes, activation='softmax')
])

# Compile the model with sparse categorical cross-entropy
CNN_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print model summary
print("CNN Model Summary:")
CNN_model.summary()
```

Figure 25: Architecture of the CNN model, showing convolutional layers, pooling layers, dropout measures, and fully connected layers for multi-class IoT network behavior classification.

```
CNN Model Summary:
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 82, 64)            256

 batch_normalization (Batch  (None, 82, 64)            256
 Normalization)

 max_pooling1d (MaxPooling1  (None, 41, 64)            0
 D)

 dropout (Dropout)           (None, 41, 64)            0

 conv1d_1 (Conv1D)           (None, 39, 128)           24704

 batch_normalization_1 (Bat  (None, 39, 128)           512
 chNormalization)

 max_pooling1d_1 (MaxPoolin  (None, 19, 128)           0
 g1D)

 dropout_1 (Dropout)         (None, 19, 128)           0

...
Total params: 338700 (1.29 MB)
Trainable params: 338316 (1.29 MB)
Non-trainable params: 384 (1.50 KB)
```

Figure 26: Architecture of the CNN model and its output

Step 10: Training and Evaluation

- The model was trained using early stopping to prevent overfitting, and training/validation metrics were plotted over epochs.
- The model was saved for future use, and evaluation metrics (accuracy, precision, recall, F1-score) were computed.

```python
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Train the model with early stopping
history = CNN_model.fit(X_train_reshaped, y_train_encoded, epochs=5, batch_size=32,
                        validation_data=(X_val_reshaped, y_val_encoded),
                        callbacks=[early_stopping])
```

Figure 27: Illustration of the training process, showing how early stopping monitors the validation loss and halts training when no significant improvement is observed, preventing overfitting.

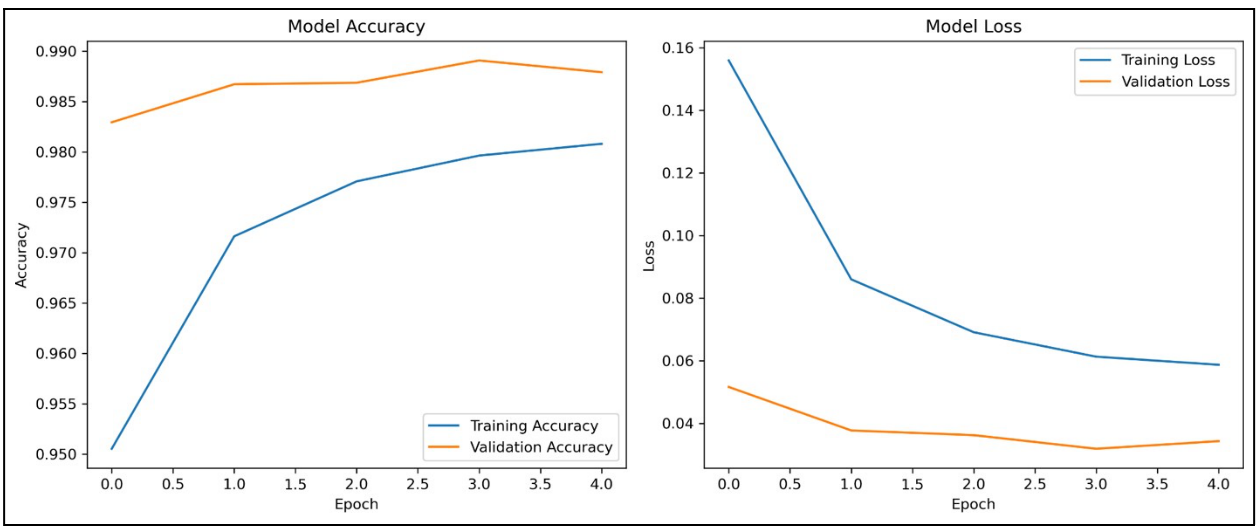
```
Epoch 1/5
19879/19879 [==============================] - 306s 15ms/step - loss: 0.1559 - accuracy: 0.9505 - val_loss: 0.
Epoch 2/5
19879/19879 [==============================] - 294s 15ms/step - loss: 0.0860 - accuracy: 0.9716 - val_loss: 0.
Epoch 3/5
19879/19879 [==============================] - 295s 15ms/step - loss: 0.0691 - accuracy: 0.9771 - val_loss: 0.
Epoch 4/5
19879/19879 [==============================] - 295s 15ms/step - loss: 0.0613 - accuracy: 0.9797 - val_loss: 0.
Epoch 5/5
19879/19879 [==============================] - 295s 15ms/step - loss: 0.0587 - accuracy: 0.9808 - val_loss: 0.
```

Figure 28: Illustration of the training process and its outcome

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

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='best')
```

Figure 29: Visualization of the training and validation accuracy and loss over epochs.




```python
CNN_model.save('cnn_trained_model.h5')
print("Model saved as 'cnn_trained_model.h5'")
```

Figure 30: Snapshot showing the process of saving the trained CNN model to a file named cnn_trained_model.h5, preserving the model for deployment or further use.

- Saves the model in the HDF5 format (.h5 file), which includes the model architecture, trained weights, and optimizer configuration.
- The file name is specified as 'cnn_trained_model.h5'.

CNN Model Evaluation

```python
# Load the previously saved model
loaded_model_cnn = load_model('cnn_trained_model.h5')
print("Model loaded successfully.")

# Predict the classes for the test set
y_test_pred = loaded_model_cnn.predict(X_test_reshaped)
y_test_pred_classes = np.argmax(y_test_pred, axis=1)
y_test_true_classes = y_test_encoded

# Convert numerical labels back to original class names
y_test_pred_labels = label_encoder.inverse_transform(y_test_pred_classes)
y_test_true_labels = label_encoder.inverse_transform(y_test_true_classes)

# Convert label names to strings for compatibility with classification_report
class_labels = [str(label) for label in label_encoder.classes_]

# Print the classification report with actual labels
print("Classification Report:")
print(classification_report(y_test_true_labels, y_test_pred_labels, target_names=class_labels, digits=4))

# Plot the confusion matrix with actual labels
conf_matrix = confusion_matrix(y_test_true_labels, y_test_pred_labels, labels=label_encoder.classes_)

plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_labels, yticklabels=class_labels)

plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

Figure 31: Classification report table summarizing precision, recall, F1-score, and support for each attack type in the IoT dataset.

```
Model loaded successfully.
1155/1155 [==============================] - 5s 4ms/step
Classification Report:
```

	precision	recall	f1-score	support
ARP_poisioning	0.9860	0.9389	0.9619	2325
DDOS_Slowloris	0.7500	0.9938	0.8548	160
DOS_SYN_Hping	1.0000	1.0000	1.0000	28398
MQTT_Publish	1.0000	0.9976	0.9988	1244
Metasploit_Brute_Force_SSH	0.0935	0.9091	0.1695	11
NMAP_FIN_SCAN	0.8750	0.8750	0.8750	8
NMAP_OS_DETECTION	1.0000	1.0000	1.0000	600
NMAP_TCP_scan	0.9934	1.0000	0.9967	301
NMAP_UDP_SCAN	1.0000	0.9305	0.9640	777
NMAP_XMAS_TREE_SCAN	1.0000	0.9967	0.9983	603
Thing_Speak	0.9774	0.9782	0.9778	2433
Wipro_bulb	0.7979	0.9868	0.8824	76
accuracy			0.9930	36936
macro avg	0.8728	0.9672	0.8899	36936
weighted avg	0.9958	0.9930	0.9941	36936

Figure 32: Classification report table summary and results

```
test_loss, test_accuracy = loaded_model_cnn.evaluate(X_test_reshaped, y_test_encoded)

print(f"Test accuracy: {test_accuracy * 100:.2f}%")
```

Figure 33: The test accuracy score presented as a percentage, indicating the model's effectiveness in predicting attack types on unseen IoT network data.

- Evaluates the model's performance on the test data.
- Computes the test loss and accuracy.
- X_test_reshaped: Scaled and reshaped test features.
- y_test_encoded: Encoded true labels for the test set.

CNN Model Prediction and Actual Label Comparison

```
# Get unique classes from the test set
unique_classes = np.unique(y_test_encoded)

# Collect 5 samples from each class in the test set
sample_indices = []
for class_label in unique_classes:
    class_indices = np.where(y_test_encoded == class_label)[0]
    selected_indices = np.random.choice(class_indices, size=5, replace=False)
    sample_indices.extend(selected_indices)

# Extract the samples and corresponding labels
X_sample = X_test_reshaped[sample_indices]
y_sample_true = y_test_encoded[sample_indices]

# Predict the labels for the samples
y_sample_pred_probs = loaded_model_cnn.predict(X_sample)
y_sample_pred = np.argmax(y_sample_pred_probs, axis=1)

# Convert the reshaped X_sample back to original feature form for readability
X_sample_original = X_test.iloc[sample_indices].reset_index(drop=True)

# Create a DataFrame with features, actual labels, and predicted labels
a_p_labels = X_sample_original.copy()
a_p_labels['Actual Label'] = label_encoder.inverse_transform(y_sample_true)
a_p_labels['Predicted Label'] = label_encoder.inverse_transform(y_sample_pred)

# Display the DataFrame
print("Comparison of Actual and Predicted Labels for 5 Samples from Each Class:")
display(a_p_labels)
```

Figure 34: Tabular view of IoT features, actual labels, and predicted labels for 5 samples from each class, providing a detailed comparison of the model's predictions.

.

3.7 GAN Model

```python
# Check class distribution in y_train
print("Class Distribution in y_train:")
display(Counter(y_train))

# Determine the maximum class size (for balancing)
max_class_size = max(Counter(y_train).values())

# Calculate how many samples to generate for each underrepresented class
samples_to_generate = {cls: max_class_size - count for cls, count in Counter(y_train).items() if count
print("Samples to Generate Per Class:")
display(samples_to_generate)
```

```python
# GAN parameters
latent_dim = 100  # Dimension of random noise vector
num_classes = len(np.unique(y_train))  # Number of classes

# Build Generator
def build_generator(latent_dim, num_classes, feature_dim):
    noise_input = layers.Input(shape=(latent_dim,))
    label_input = layers.Input(shape=(num_classes,))
    merged_input = layers.Concatenate()([noise_input, label_input])

    x = layers.Dense(128, activation='relu')(merged_input)
    x = layers.Dense(256, activation='relu')(x)
    output = layers.Dense(feature_dim, activation='tanh')(x)

    return tf.keras.Model([noise_input, label_input], output)

# Build Discriminator
def build_discriminator(num_classes, feature_dim):
    data_input = layers.Input(shape=(feature_dim,))
    label_input = layers.Input(shape=(num_classes,))
    merged_input = layers.Concatenate()([data_input, label_input])

    x = layers.Dense(256, activation='relu')(merged_input)
    x = layers.Dense(128, activation='relu')(x)
    output = layers.Dense(1, activation='sigmoid')(x)

    return tf.keras.Model([data_input, label_input], output)

# Instantiate models
feature_dim = X_train.shape[1]  # Number of features in X_train
generator = build_generator(latent_dim, num_classes, feature_dim)
discriminator = build_discriminator(num_classes, feature_dim)

# Compile discriminator
discriminator.compile(optimizer=tf.keras.optimizers.Adam(0.0002, 0.5), loss='binary_crossentropy', metrics

# Build and compile GAN
noise = layers.Input(shape=(latent_dim,))
label = layers.Input(shape=(num_classes,))
generated_sample = generator([noise, label])
discriminator.trainable = False  # Freeze discriminator during generator training
gan_output = discriminator([generated_sample, label])
gan = tf.keras.Model([noise, label], gan_output)
gan.compile(optimizer=tf.keras.optimizers.Adam(0.0002, 0.5), loss='binary_crossentropy')
gan.summary()
```

Figure 35: Code depicting the workings of the GAN model

GAN Data-CNN Model Training

```
# Split the balanced data into training and validation sets
GAN_X_train_final, GAN_X_val, GAN_y_train_final, GAN_y_val = train_test_split(GAN_X_train_balanced, GAN_y_
                                                              random_state=42, stratify=GAN_y_train_balanc

# Display the shapes of the splits
print("X_train_final shape:", GAN_X_train_final.shape)
print("X_val shape:", GAN_X_val.shape)
print("y_train_final shape:", GAN_y_train_final.shape)
print("y_val shape:", GAN_y_val.shape)

# Apply standard scaling to the training, validation, and testing data separately
scaler = StandardScaler()
GAN_X_train_scaled = scaler.fit_transform(GAN_X_train_final)
GAN_X_val_scaled = scaler.transform(GAN_X_val)

# Convert scaled features back to DataFrames for consistency
GAN_X_train_prepared = pd.DataFrame(GAN_X_train_scaled, columns=X.columns)
GAN_X_val_prepared = pd.DataFrame(GAN_X_val_scaled, columns=X.columns)

# Display the shapes of the prepared train, validation, and test sets with the specified format
print(f"\nGAN Generated Training features shape: {GAN_X_train_prepared.shape}")
print(f"GAN Generated Validation features shape: {GAN_X_val_prepared.shape}")
print(f"GAN Training labels shape: {GAN_y_train_final.shape}")
print(f"GAN Validation labels shape: {GAN_y_val.shape}")
# Apply label encoding to the labels instead of one-hot encoding
label_encoder = LabelEncoder()
GAN_y_train_encoded = label_encoder.fit_transform(GAN_y_train_final)
GAN_y_val_encoded = label_encoder.transform(GAN_y_val)

# Determine the number of classes
num_classes = len(np.unique(GAN_y_train_encoded))

# Reshape data for CNN input (CNN expects 3D input: (samples, timesteps, features))
GAN_X_train_reshaped = np.expand_dims(GAN_X_train_prepared, axis=2)
GAN_X_val_reshaped = np.expand_dims(GAN_X_val_prepared, axis=2)

# Display the shapes of the prepared train, validation, and test sets with the specified format for CNN mo
print(f"\nGAN-CNN Training features shape: {GAN_X_train_reshaped.shape}")
print(f"GAN-CNN Validation features shape: {GAN_X_val_reshaped.shape}")
```

Figure 36: Code showing the merger of the CNN and the GAN and the training process

```
# Build the CNN model with added measures to prevent overfitting
GAN_CNN_model = Sequential([
    Conv1D(64, kernel_size=3, activation='relu', input_shape=(GAN_X_train_reshaped.shape[1], 1)),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),
    Dropout(0.4),
    Conv1D(128, kernel_size=3, activation='relu'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),
    Dropout(0.5),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.6),
    Dense(num_classes, activation='softmax')
])

# Compile the model with sparse categorical cross-entropy
GAN_CNN_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Print model summary
print("GAN - CNN Model Summary:")
GAN_CNN_model.summary()
```

Figure 37: Code depicting the workings of the GAN_CNN_model

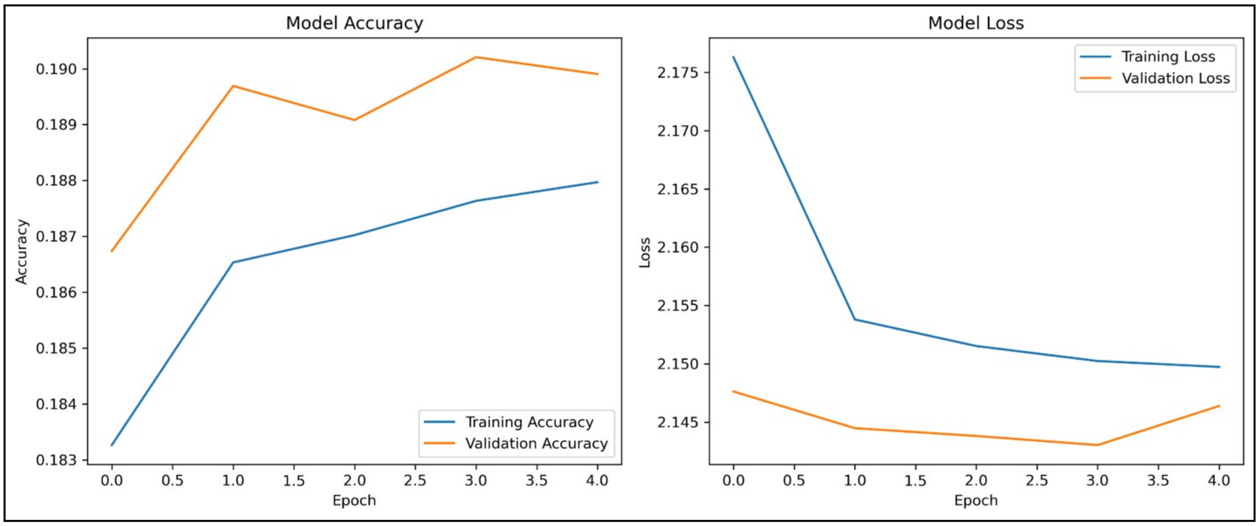


Figure 38: Plot training & validation accuracy values

3.8 GAN-CNN Model Evaluation

```python
# Load the previously saved model
loaded_model_gan_cnn = load_model('gan_cnn_trained_model.h5')
print("Model loaded successfully.")

# Predict the classes for the test set
y_test_pred = loaded_model_gan_cnn.predict(X_test_reshaped)
y_test_pred_classes = np.argmax(y_test_pred, axis=1)
y_test_true_classes = y_test_encoded

# Convert numerical labels back to original class names
y_test_pred_labels = label_encoder.inverse_transform(y_test_pred_classes)
y_test_true_labels = label_encoder.inverse_transform(y_test_true_classes)

# Convert label names to strings for compatibility with classification_report
class_labels = [str(label) for label in label_encoder.classes_]

# Print the classification report with actual labels
print("Classification Report:")
print(classification_report(y_test_true_labels, y_test_pred_labels, target_names=class_labels, digits=4))

# Plot the confusion matrix with actual labels
conf_matrix = confusion_matrix(y_test_true_labels, y_test_pred_labels, labels=label_encoder.classes_)

plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Greens',
            xticklabels=class_labels, yticklabels=class_labels)

plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

Figure 39: Classification report displaying precision, recall, F1-score, and support for each IoT attack type.

3.9 GAN-CNN Model Prediction and Actual Label Comparison

```python
# Get unique classes from the test set
unique_classes = np.unique(y_test_encoded)

# Collect 5 samples from each class in the test set
sample_indices = []
for class_label in unique_classes:
    class_indices = np.where(y_test_encoded == class_label)[0]
    selected_indices = np.random.choice(class_indices, size=5, replace=False)
    sample_indices.extend(selected_indices)

# Extract the samples and corresponding labels
X_sample = X_test_reshaped[sample_indices]
y_sample_true = y_test_encoded[sample_indices]

# Predict the labels for the samples
y_sample_pred_probs = loaded_model_gan_cnn.predict(X_sample)
y_sample_pred = np.argmax(y_sample_pred_probs, axis=1)

# Convert the reshaped X_sample back to original feature form for readability
X_sample_original = X_test.iloc[sample_indices].reset_index(drop=True)

# Create a DataFrame with features, actual labels, and predicted labels
a_p_labels = X_sample_original.copy()
a_p_labels['Actual Label'] = label_encoder.inverse_transform(y_sample_true)
a_p_labels['Predicted Label'] = label_encoder.inverse_transform(y_sample_pred)

# Display the DataFrame
print("Comparison of Actual and Predicted Labels for 5 Samples from Each Class:")
display(a_p_labels)
```

Figure 40: Tabular comparison of IoT features, actual attack labels, and predicted labels for 5 samples from each attack type, illustrating model predictions and their accuracy.

References

NumPy Documentation (n.d.) *NumPy provides tools for numerical computations and handling large datasets.* Available at: https://numpy.org/doc/

Pandas Documentation (n.d.) *Pandas is used for data manipulation and analysis.* Available at: https://pandas.pydata.org/pandas-docs/stable/

Matplotlib Documentation (n.d.) *Matplotlib is used for creating detailed visualizations and graphs.* Available at: https://matplotlib.org/stable/contents.html

Seaborn Documentation (n.d.) *Seaborn simplifies statistical data visualizations.* Available at: https://seaborn.pydata.org/

Scikit-learn Documentation (n.d.) *Scikit-learn provides tools for preprocessing, model building, and evaluation.* Available at: https://scikit-learn.org/stable/documentation.html

Imbalanced-learn Documentation (n.d.) *Imbalanced-learn offers techniques for handling imbalanced datasets.* Available at: https://imbalanced-learn.org/stable/

TensorFlow Documentation (n.d.) *TensorFlow provides an open-source platform for building deep learning models.* Available at: https://www.tensorflow.org/

RT-IOT2022 Dataset (n.d.) *RT-IOT2022 dataset contains IoT network traffic data for research purposes.* Available at: https://archive.ics.uci.edu/dataset/942/rt-iot2022

GAN Documentation (n.d.) *Generative Adversarial Networks (GANs) are used for generating synthetic data.* Available at: https://paperswithcode.com/method/gan