Mini Project Report

on

Autoencoder Approach to Intrusion Detection in UAV Networks

SUBMITTED BY

|  |  |
| --- | --- |
| Name: Anjana Kumar Rao Parepalli | Name: Shrijani Patil |
| Registration No.: 235890228 | Registration No.: 235890036 |

A black text on a white background

Description automatically generated

School of Computer Engineering

October 2025

1. Abstract

Unmanned aerial vehicle (UAV) technology is being increasingly adopted in recent years for a variety of applications and across civilian and military sectors. UAVs often operate in UAV networks, with drones forming connections between each other in different topologies for purposes like information gathering in search and rescue missions or wide area coverage in large agricultural fields. However, UAV networks are vulnerable to cyberattacks perpetrated by malicious actors for motivations that include espionage, blackmail, and sabotage of defense or commercial interests. This paper addresses that concern by developing an Autoencoder model as an Intrusion Detection System (IDS) in UAV networks that can detect new and unseen kinds of attacks. The dataset used in this study is titled “Cyber-Physical Dataset for UAVs Under Normal Operations and Cyberattacks”, available in a public GitHub repository. Considering ‘attack’ as the positive class and ‘benign’ as the negative class, the results of the best Autoencoder model were 89.17187% accuracy, 94.75589% precision, and 90.56723% recall with an area under the Receiver Operating Characteristic (AUC) score of 0.89504 indicating competence in distinguishing between normal network conditions and intrusion. This study proves that an Autoencoder approach is effective in detecting intrusion in UAV networks.

***Keywords– UAVs, Machine Learning, Autoencoders, Intrusion Detection Systems***

1. Introduction

Unmanned aerial vehicles (UAVs) are used in a wide variety of applications, including but not limited to surveillance, search and rescue, and defense. The adoption of UAV technology becomes more popular year by year, with drones moving from military applications to commercial and civilian applications like agricultural monitoring and package delivery. With the rise of UAV technology, however, comes an increase in the number and type of cyberattacks perpetrated on UAV networks. Denial-of-Service (DoS) attacks overload and paralyze UAVs, while replay attacks spy on sensitive data being transmitted between UAVs in the same network. Because new kinds of attacks are invented very often, traditional machine learning models that focus on classifying the kind of attack fall short on detecting and classifying unseen (ground zero) cyberattacks. Instead, an Intrusion Detection System (IDS) is developed using the autoencoder algorithm to detect if any attack has taken place instead of classifying what kind of state the network is in.

The authors of a related paper titled “Cyber-Physical Intrusion Detection System for Unmanned Aerial Vehicles” developed the dataset utilized in this study by connecting drones to two computers and recording the conditions of this network under benign / normal operation and under simulated cyberattacks. Even though the primary aim of that paper was to compare using combined cyber-physical attack features to using only cyberattack or only physical attack features, the results provided performance measures of four algorithms: Support Vector Machine (SVM) achieved 89.96% accuracy and a 0.9104 AUC score, Feedforward Neural Network (FNN) achieved 90.62% accuracy and a 0.9452 AUC score, Long Short Term Memory Convolutional Neural Network (LSTM-CNN) achieved 93.22% accuracy and a 0.9661 AUC score, and One-Dimensional Convolution Neural Network (1D-CNN) achieved 94.53% accuracy and a 0.9695 AUC score. These models provide a reference to compare the performance of the Autoencoder model we develop.

Another research paper titled “Random search for hyper-parameter optimization” is about the process of randomly searching for the best set of hyperparameters for a given model. We use TensorFlow / Keras’s RandomSearch implementation to find the best set of hyperparameters for our Autoencoder model in a fraction of the time it takes to perform an exhaustive search like Grid Search. This paper provides details on how Random Search works.

Our study aims to discover if the Autoencoder algorithm is more or less viable in detecting intrusion / attack events. The expected result is an efficient Autoencoder model that can detect intrusion events with high precision and recall scores (> 90%). We aim for ease in adding more encoder or decoder blocks and other complexities for future research that aims to improve performance.

1. Methodology
   1. Dataset

The dataset used in this study is the “[Cyber-Physical Dataset for UAVs Under Normal Operations and Cyberattacks](https://github.com/uamughal/UAVs-Dataset-Under-Normal-and-Cyberattacks)”, a CSV file publicly available on GitHub by user uamughal. It contains a total of 54,784 observations split equally among five classes: Benign, Denial-of-Service Attack, Replay Attack, Evil Twin, and False Data Injection. Furthermore, each class’s attack samples are bifurcated into the Cyber or the Physical variety. For example, Cyber DoS Attack samples range from index 13,718 to index 25,389 while Physical DoS Attack samples range from index 25,390 to index 26,363.

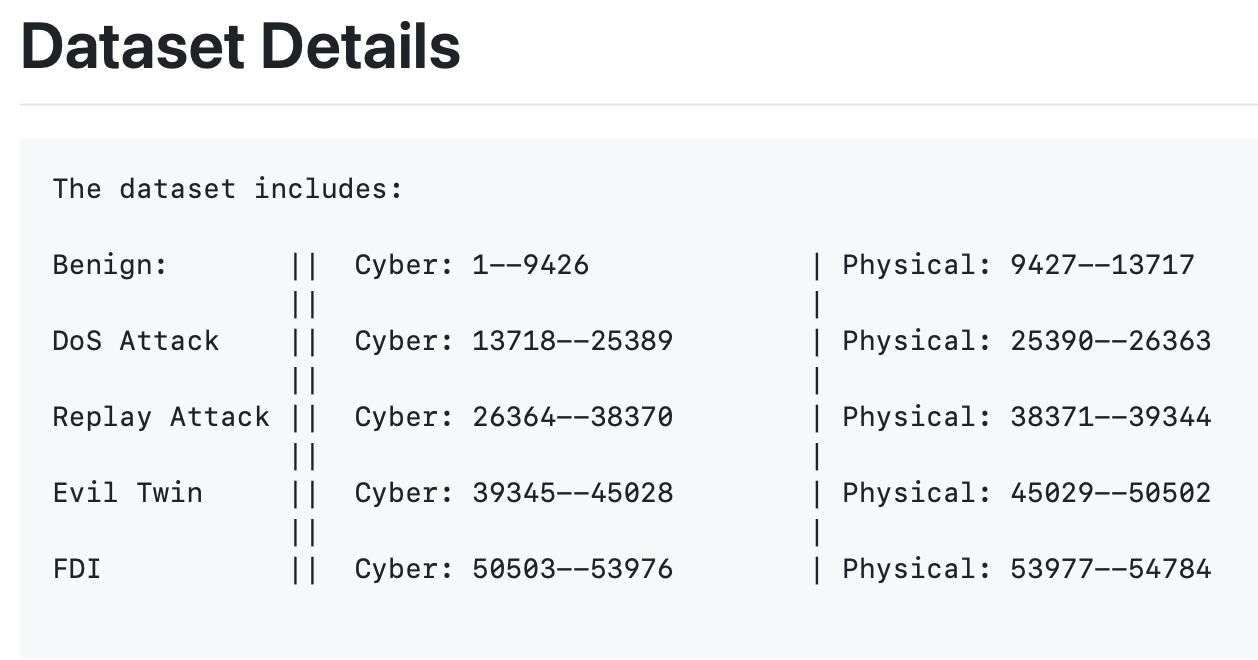


Figure 1: Authors’ description of the dataset they created, available at <https://github.com/uamughal/UAVs-Dataset-Under-Normal-and-Cyberattacks>.

* 1. Preprocessing

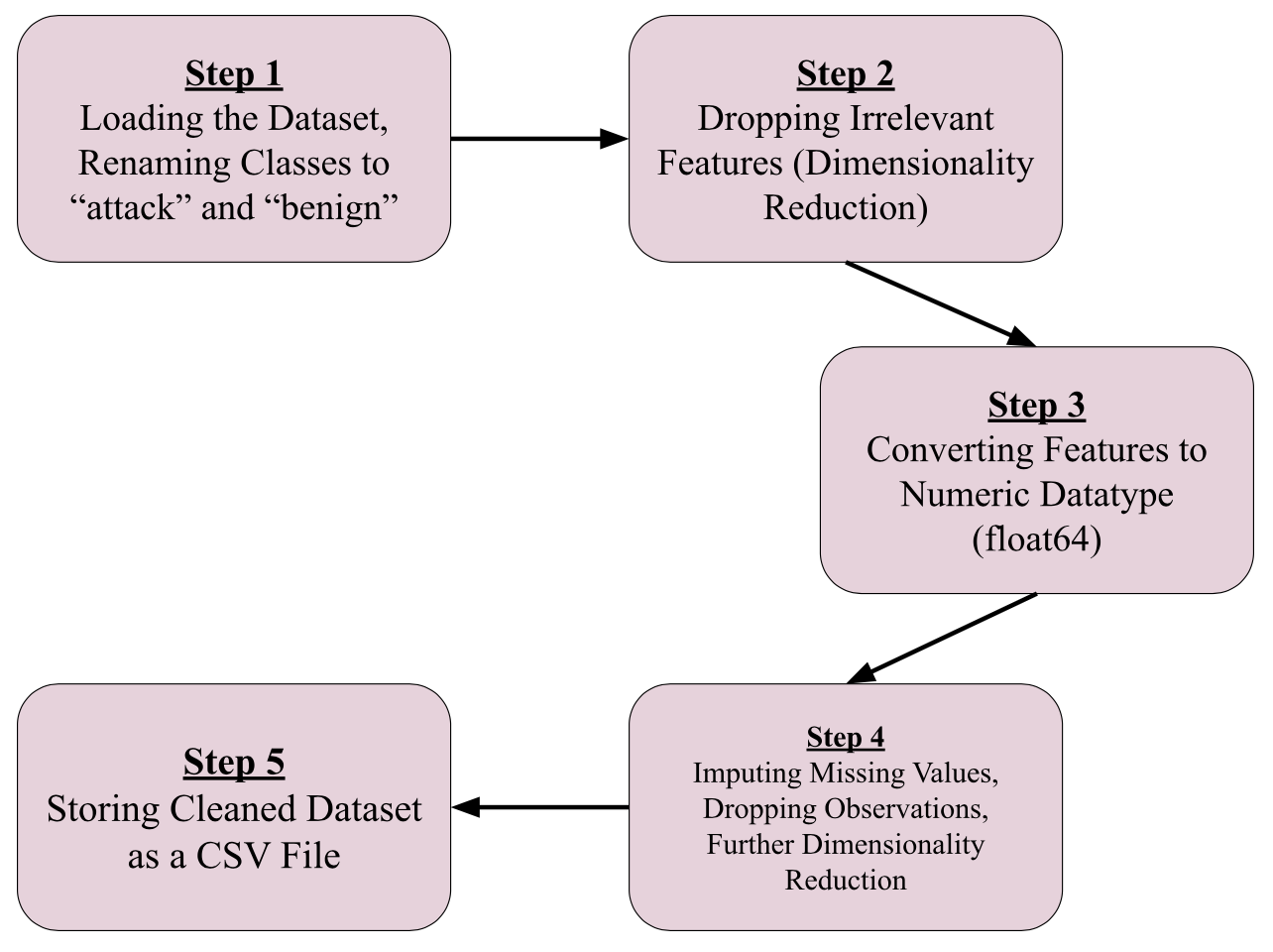
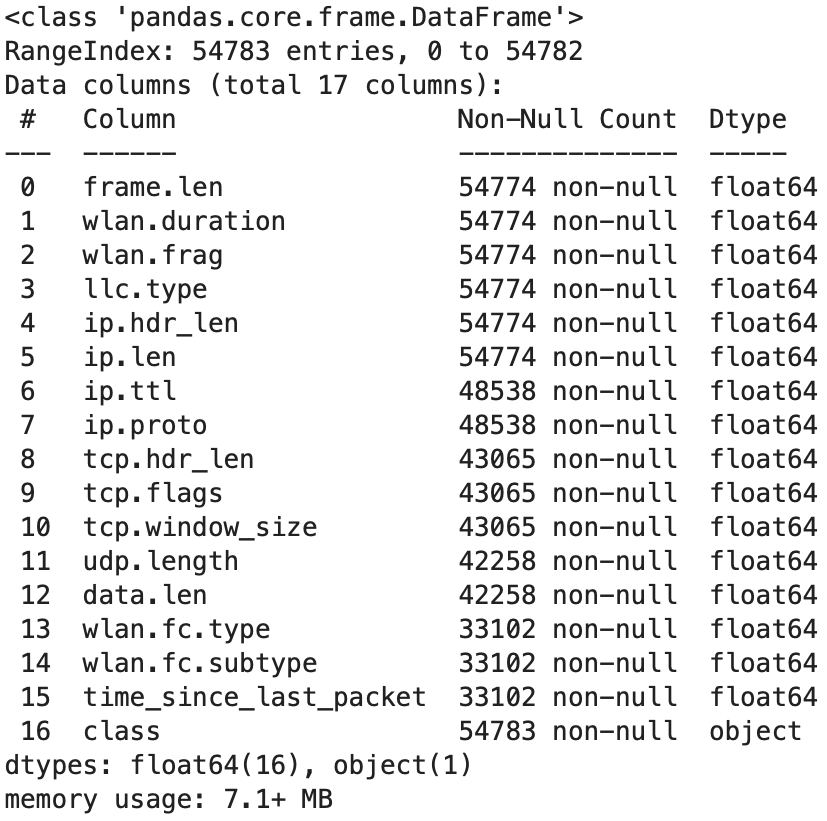
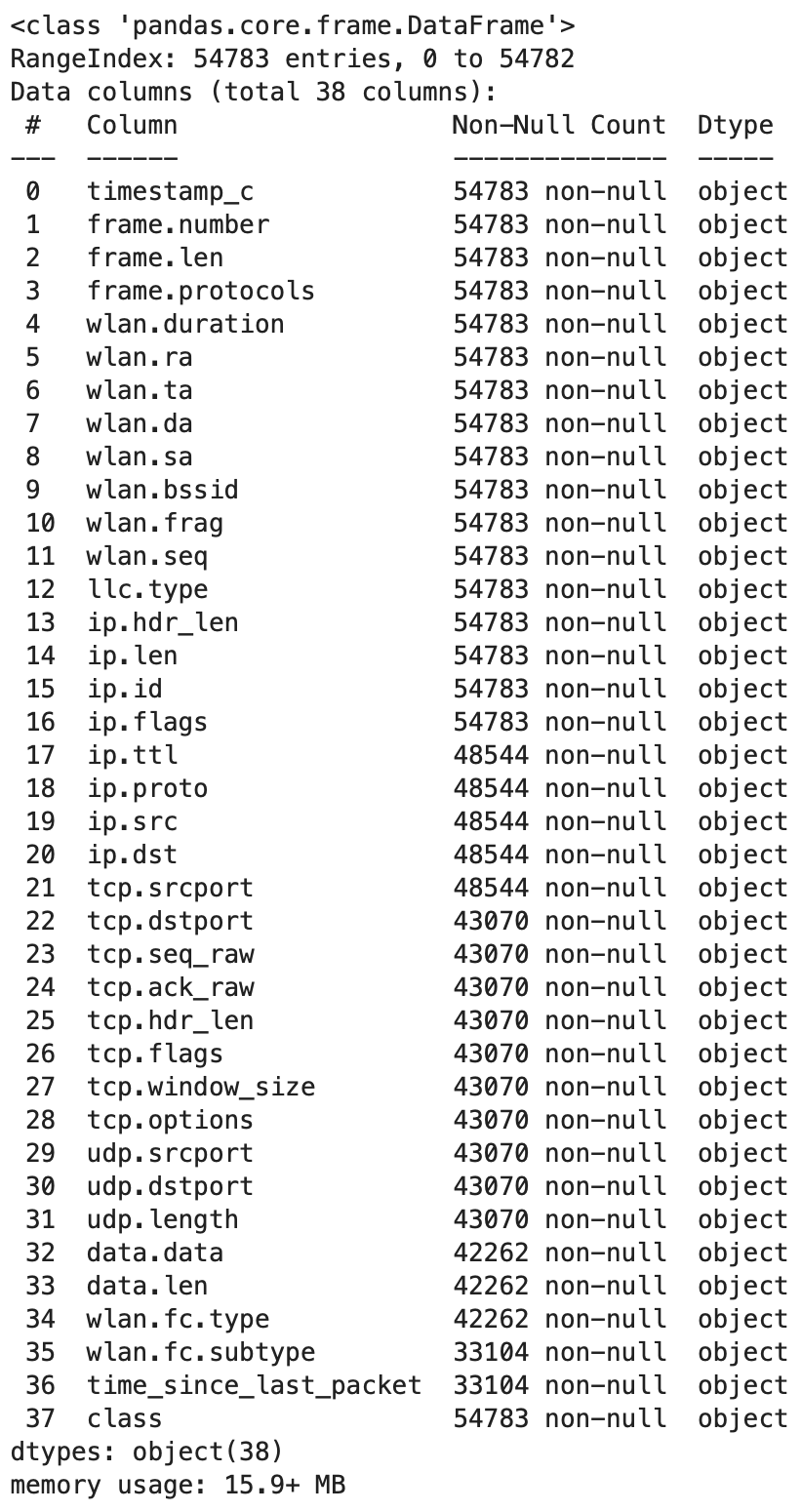
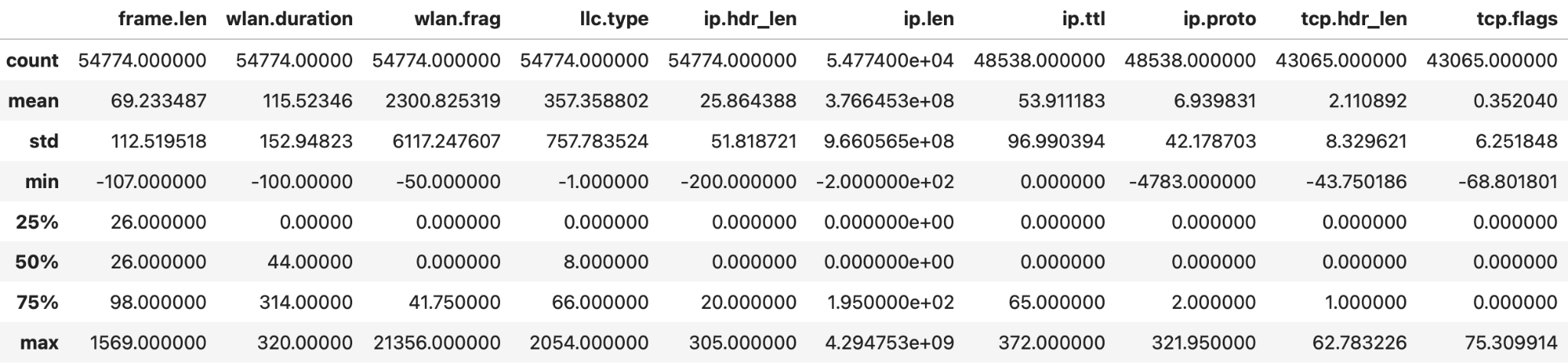


Figure 2: Preprocessing pipeline steps followed in this study.

The first step was loading the dataset into Jupyter Notebook as a Pandas DataFrame. A small mistake made by the original authors was corrected (they forgot to label the physical benign network state as ‘benign’ and instead left it as NaN). All the different kinds of attacks were relabeled as one class called ‘attack’. This is because autoencoder will only be used to detect if an attack or intrusion event has occurred, not to classify the kind of attack. Irrelevant features were dropped from the DataFrame to reduce dimensionality. 16 columns were retained, including the prediction target. The effect of these changes was placed into a new DataFrame named ‘cleaned\_uav\_network\_df’ to avoid overwriting the original DataFrame. Using the info() method, we found that the datatype of all columns was ‘object’. This made sense for the prediction target (‘class’) because the values were Strings that either read ‘benign’ or the name of an attack. However, an Autoencoder model requires numeric features to function. Therefore we converted the values of every column except for the prediction target into a numeric datatype (float64). Any errors encountered were coerced into becoming a NaN / null value. The info() method on this cleaned DataFrame confirmed that all features were numeric, and the describe() method was called to provide a statistical summary of each feature column, listing the standard deviation, mean, percentiles, and minimum and maximum values.



Figures 3 and 4: Description of dataset columns before and after initial dimensionality reduction and type conversion of features (Steps 2 and 3).



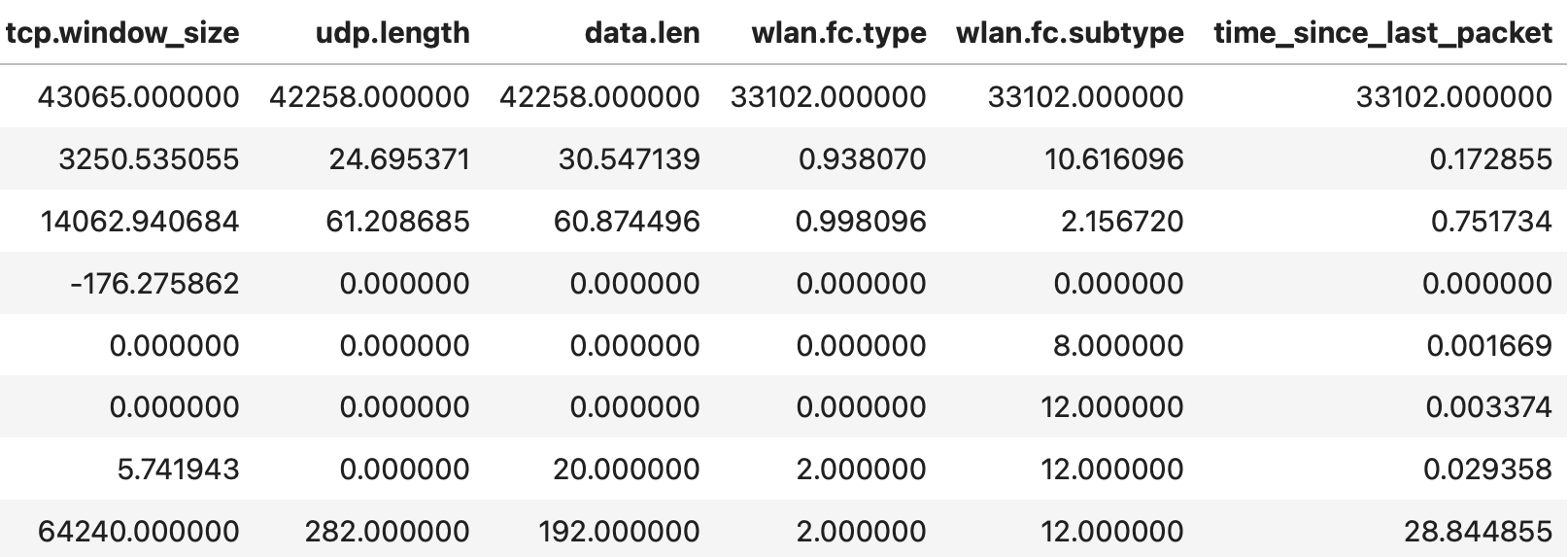


Figure 5: Statistical summary of feature columns before imputation and further dimensionality reduction (before Step 4).

The describe() method also revealed that some columns were missing a large number of sample observations. We used the isnull(), sum(), and mean() methods in our logic to find the number and percentage of missing samples per feature. Three features, ‘wlan.fc.type’, ‘wlan.fc.subtype’, and ‘time\_since\_last\_packet’, were missing 21,681 out of the total 54,784 samples (around 40% missing values). On the other end, ‘frame.len’, ‘wlan.duration’, ‘wlan.frag’, ‘llc.type’, ‘ip.hdr\_len’, and ‘ip.len’ were missing only nine samples. To deal with this problem, we had three options for each feature: drop the feature entirely, impute its median or mean value, or drop the observations missing a value for that feature.

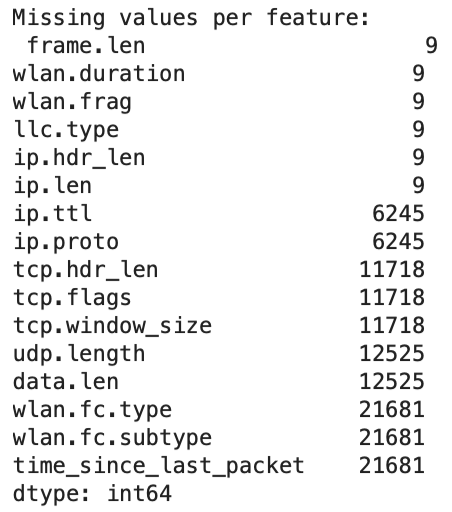
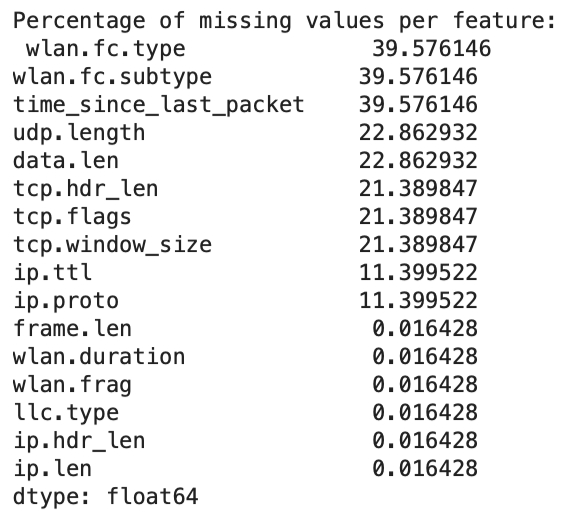
 

Figure 6: Number and percentage of missing values per feature.

We used MatPlotLib to plot the frequency histograms (50 bins each) of each feature to understand the underlying data distribution that informs our decision to use either median or mean when imputing feature values. If a given distribution is skewed, then we use the median. If a distribution roughly follows the Gaussian/Normal Distribution, then we use the mean. It turns out that all features had a skewed distribution so the median was used in all imputing calculations.

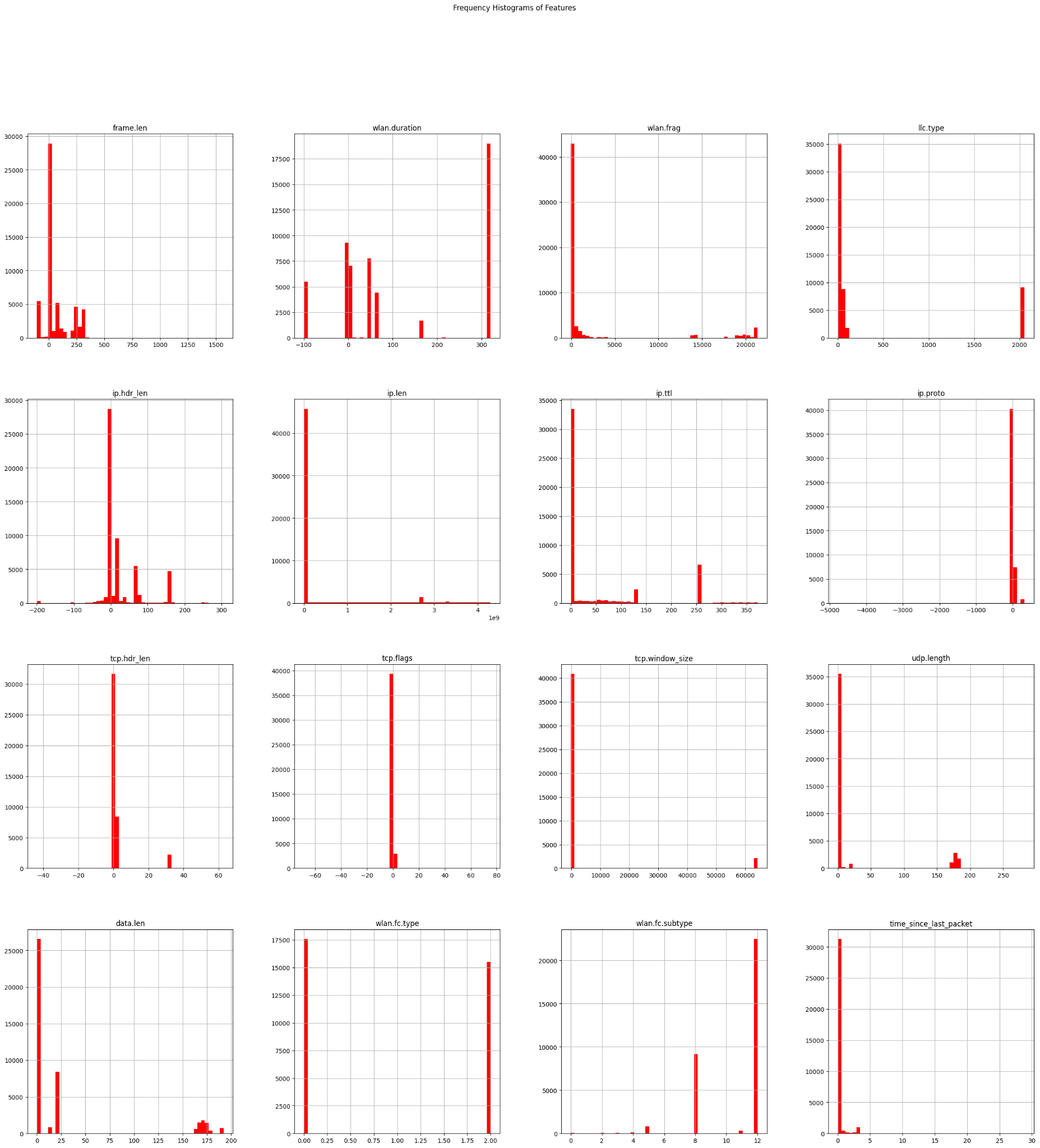


Figure 7: Sample data distributions of every feature, represented by histograms of 50 bins each.

We also took the opportunity to visualize the imbalance between the ‘benign’ and ‘attack’ classes. This imbalance is acceptable for an autoencoder approach because the autoencoder is only trained to reconstruct ‘benign’ data. Samples featuring new kinds of attacks can be added to the dataset and still detected using the trained autoencoder, showcasing the flexible nature of this approach.

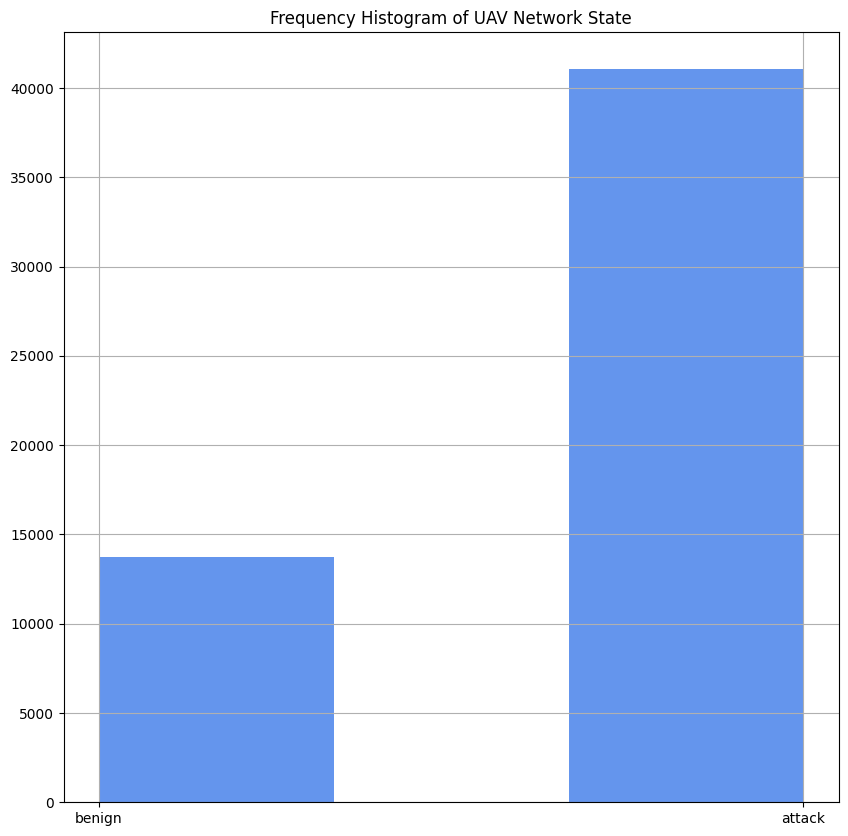


Figure 8: Visualized imbalance between ‘benign’ and ‘attack’ classes.

The features ‘wlan.fc.type’ and ‘wlan.fc.subtype’ were dropped due to their large percentage of missing values. We chose to keep the feature ‘time\_since\_last\_packet’ due to its importance in predicting an attack, and imputed the missing values using the median. We also performed imputation on ‘ip.ttl’, ‘ip.proto’, ‘tcp.hdr\_len’, ‘tcp.flags’, ‘tcp.window\_size’, ‘udp.length’, ‘data.len’. Finally, for the features ‘frame.len’, ‘wlan.duration’, ‘wlan.frag’, ‘llc.type’, ‘ip.hdr\_len’, and ‘ip.len’, we dropped around 10 samples that lacked a value for them.

The result was a fully cleaned dataset containing 15 columns (14 numeric features) and no NaN or null values anywhere.

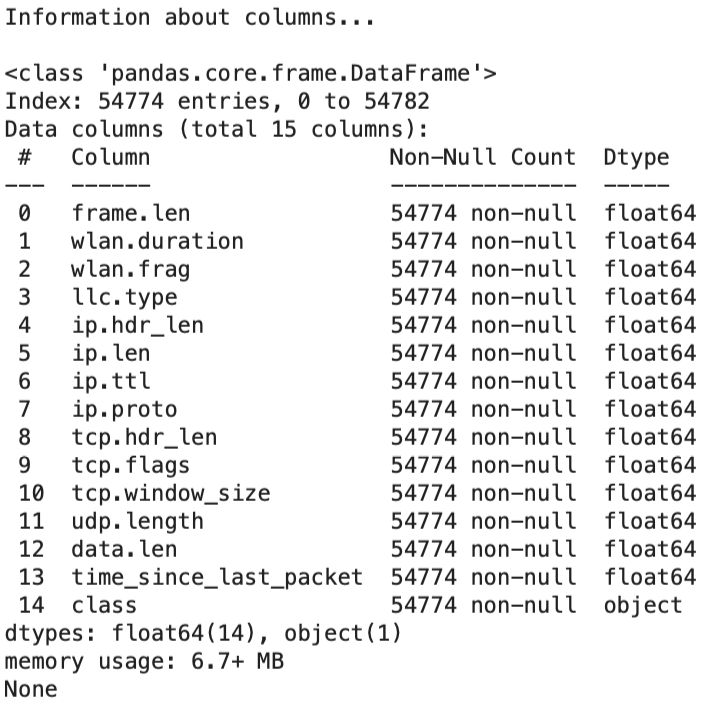


Figure 9: Columns of the final, cleaned dataset. Observe that there are no missing entries.

Results were stored in a new file titled ‘Cleaned\_UAV\_Network\_Dataset.csv’.

* 1. Building and Training an Autoencoder

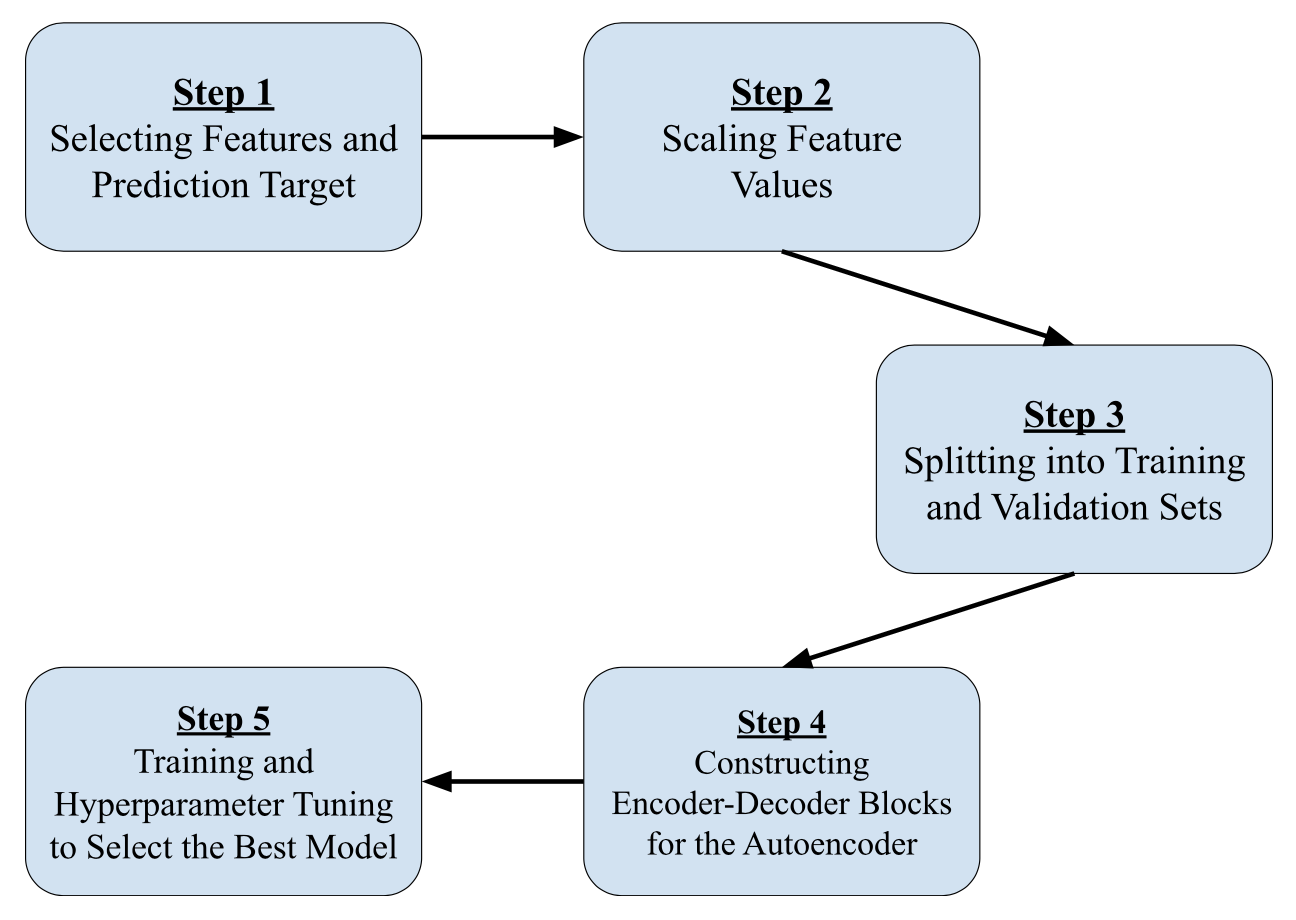


Figure 10: Steps to build and train an autoencoder.

Loading the cleaned dataset into a Pandas DataFrame, the first step in building an Autoencoder model is to select the features (designated as ‘X’) and prediction target (designated as ‘y’) columns. The training set for the model will consist entirely of benign feature values (‘X\_benign’). The training set of feature values is used to fit a StandardScaler object imported from Scikit-Learn. The scaler object then scales the feature values of the entire dataset. Next, we split the benign network training set into training and validation subsets. This lays the groundwork for training an Autoencoder model.

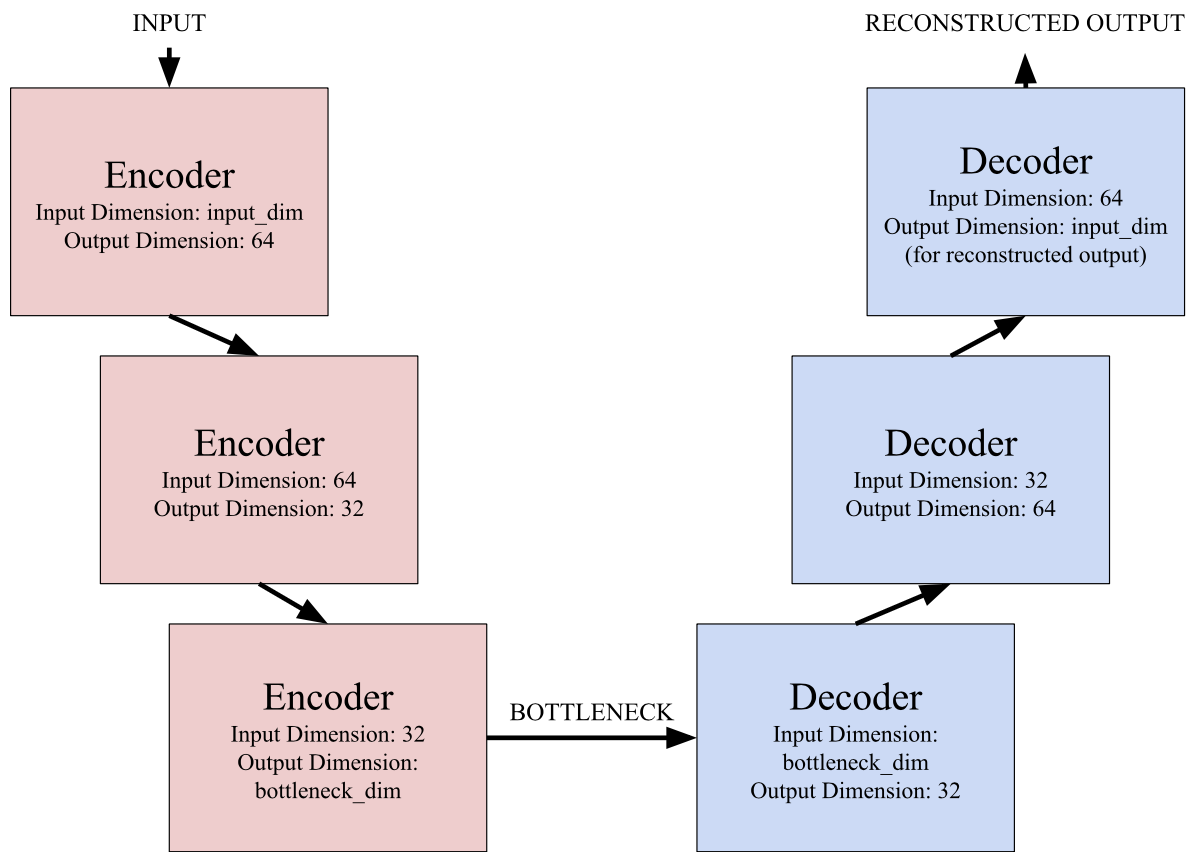


Figure 11: Block diagram architecture of the type of Autoencoder model utilized in this study.

We chose to build a “vanilla” or standard Autoencoder because it's the best kind of Autoencoder algorithm for working with tabular data like our CSV file. In order to build the Autoencoder, we lay down the blueprint for three encoder blocks and three decoder blocks. Each block is a fully connected layer (or dense layer) of a neural network, with the output dimensionality indicating the number of neurons in that block. The first encoder block takes in 14 features as inputs, and uses the Rectified Linear Unit (ReLU) activation function to learn 64 new features as a weighted combination of those original 14 features.

**Output = ReLU(Wx + b)**

x ∈ ℝ14 (input vector)

W ∈ ℝ14x64 (weight matrix)

b ∈ ℝ64 (bias terms)

It’s important to understand that we can have fewer features as the input dimension to learn more features as the output dimension in our first encoder block – Keras takes care of the expansion details for us.

The general responsibility of an encoder block is to compress the input vector into an output vector of a smaller dimension. Every time the Autoencoder model compresses a vector, it retains only the most relevant details while discarding irrelevant details. This means it learns only the most important details for reconstructing the input. At the mid-point of the Autoencoder, called the “bottleneck”, an encoder passes a vector to the first decoder block. This vector’s dimensionality, the bottleneck dimensionality, can be tuned as a hyperparameter.

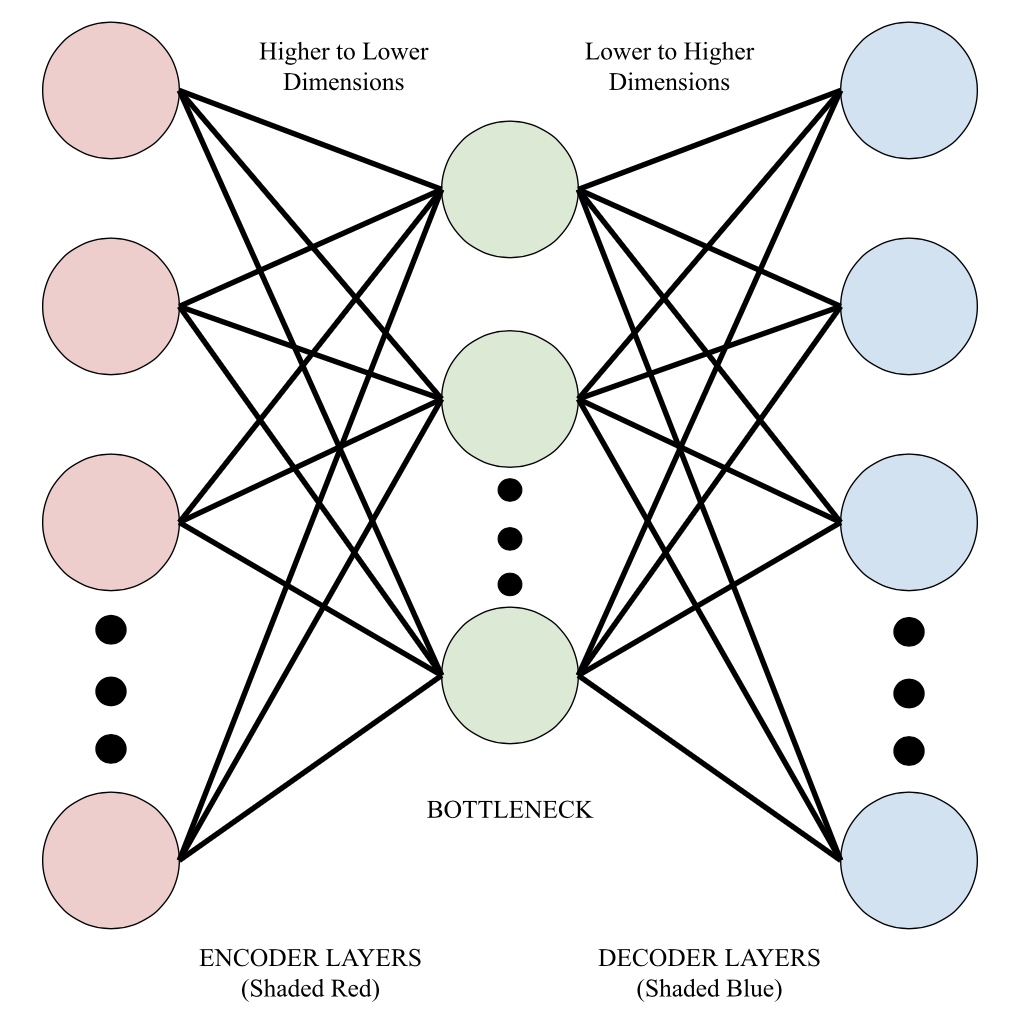


Figure 12: Representing the autoencoder as a neural network of fully connected (or dense) layers.

A decoder block functions as the inverse of an encoder block, by expanding input vectors into output vectors of larger dimensions. As we climb up the decoder blocks, we eventually end up with a reconstructed feature vector that, in the case of benign networks, will be incredibly similar to the original input. This is because a model with parameters learned from only benign network feature values will attempt to reconstruct any input as a “benign” output. If the input was the state of a network under attack, then the output (reconstructed vector) will be closer to a benign network state, and there will be noticeable reconstruction error between the input and output vectors. A threshold percentile of reconstruction error is used to determine which reconstructed outputs were derived from benign networks (anything below that percentile) and which were derived from networks under attack (anything above that percentile).

After building the Autoencoder architecture, there is one more important step; determining the most effective set of hyperparameters so that only the best model is used for evaluation. Earlier, Scikit-Learn’s train\_test\_split() method was used to divide the training set of scaled benign observations into training and validation subsets. Now, we use Keras’s implementation of Random Search (check references for details) to find the best set of hyperparameters and train and validate different models on those subsets. The specific details of our implementation are discussed in the Implementation Steps subsection of the Implementation section.

1. Implementation
   1. Tools and Libraries

Tools:

* Jupyter Notebook
  + Used to write all the logic for preprocessing the dataset and building and training the Autoencoder model
* Pip3
  + Python package installer, can be used to install packages of or below a certain version
* PyCharm
  + Interactive Development Environment (IDE) used to quickly perform sanity checks on dataset CSV files and to launch Jupyter Notebook with the correct kernel

Libraries:

* Pandas
  + Data science library used for loading and creating datasets as CSV files and DataFrames, essential for both the preprocessing and the model creation notebooks
* NumPy (< Version 2)
  + Mathematical library for mathematical operations while building the model, and in calculating the reconstruction scores and determining the threshold percentile
* MatPlotLib
  + Plotting library used to visualize data during exploratory data analysis or results when evaluating the model
* Seaborn
  + Similar to MatPlotLib, provided aesthetic and stylistic plots for visualization
* Scikit-Learn
  + Machine learning library that provided the scaler object and ability to split data into training and validation subsets as well as the code for evaluating the model’s performance
* TensorFlow and Keras
  + Machine learning / deep learning library used to build the encoder / decoder architecture of the Autoencoder model, provided an implementation of Random Search for quick hyperparameter tuning and the ability to select the best model (model with the lowest validation loss)
  1. Environment Configuration

The environment to launch Jupyter Notebook was first created and modified in PyCharm. A virtual environment (venv) running Python 3.11.9 was used for compatibility with TensorFlow. The venv was activated in PyCharm terminal as a kernel for Jupyter Notebook, named “ML Lab Mini Project Kernel”. Afterwards, the command “jupyter notebook” was typed into the PyCharm terminal to open Jupyter Notebook in browser and create the notebooks for processing data and building the Autoencoder.

The version of NumPy installed was older than version 2, to ensure compatibility with TensorFlow. Pip3 installed all the necessary packages for the environment, called within Jupyter Notebook cells as “!pip3 install …”.

* 1. Data Pipeline

The original dataset was downloaded as a CSV file from a GitHub repository and operated on using Pandas to produce a cleaned dataset, which was subsequently stored as another CSV file. The cleaned dataset was employed in building and training the Autoencoder model. For more details and a flowchart depicting the steps, refer to the Dataset and Preprocessing subsections under the Methodology section of this paper.

GitHub Repo: <https://github.com/uamughal/UAVs-Dataset-Under-Normal-and-Cyberattacks>\

* 1. Implementation Steps

This subsection is intimately related with the Building and Training an Autoencoder subsection of the Methodology section. Please consult that subsection before reading this one.

The Autoencoder model in this study is composed of three encoder and three decoder blocks. There are five tunable hyperparameters in our implementation: ‘bottleneck\_dim’, ‘learning\_rate’, ‘EPOCHS’, ‘BATCH\_SIZE’, and ‘THRESHOLD\_PERCENTILE’. The hyperparameters which the user sets are named in all capital letters. The hyperparameters that are searched using the Random Search algorithm are ‘bottleneck\_dim’ and ‘learning\_rate’. Our code implementation provides potential ‘bottleneck\_dim’ values of 8 to 64 in steps of 8 and potential ‘learning\_rate’ values as 0.0001, 0.001, and 0.005. Note that the validation loss metric used during the search is mean-squared error (MSE). The best model returned has the least validation loss at some point during training. To understand how Random Search works, consult the References section for the research paper that introduced it.

‘bottleneck\_dim’ refers to the dimensionality of the “bottleneck” part of the autoencoder where it transitions from an encoder block to a decoder block. Smaller dimensions like 4 or 8 focus more on compression, which causes anomalies (attacks in this context) to stand out more. Larger dimensions like 32 or 64 allow less compression, which means the model will reconstruct both benign and subtle attack data very well. Larger bottleneck dimensionality can potentially cause overfitting by memorizing the benign data instead of learning the important underlying patterns, and reduces the ability of the model to distinguish between a benign network and a network that’s being intruded upon. The best bottleneck dimensionality found was 48.

‘learning\_rate’ is the learning rate of the Adaptive Moment Estimator (Adam) optimizer, a type of stochastic gradient descent that can avoid falling into local minima traps. For more details on Adam, consult the References section for its introductory research paper. A learning rate that is too high (e.g.: 0.01) leads to oscillation and divergence from the minimum. A learning rate that is too low (e.g.: 0.00001) leads to very slow convergence and the solver possibly getting stuck in a local minimum. A moderate learning rate is required for the best results. The best learning rate found was 0.01.

‘EPOCHS’ are the number of epochs for training a model– the number of times the scaled benign training set is split into disjoint batches of benign observations in this study’s context. Too few epochs (< 50) means that the model may not learn the benign pattern well enough, leading to underfitting. Too many epochs (> 200) means that the model will begin memorizing benign states and loses the ability to generalize, leading to overfitting. It will begin reconstructing attack too well, causing a lot of attack observations to be labeled as benign (Type II Error as ‘attack’ is the positive class and ‘benign’ is the negative class). The default value provided in our code is 100.

‘BATCH\_SIZE’ is the size or number of observations per batch. When a batch is processed by the model, the weights are updated. A small batch size of 16 or 32 causes frequent weight updates which improves generalization but slows the training process. A large batch size of 128 or 256 causes smoother and faster training but runs the risk of overfitting. To balance the tradeoff between training time and consistent weight updates, our code uses a batch size of 64 by default.

‘THRESHOLD\_PERCENTILE’ is used to determine whether an input sample is labeled as ‘attack’ or as ‘benign’. Anything above the given threshold percentile of reconstruction error is labeled as ‘attack’, the positive class. Higher percentile thresholds cause fewer false positives (Type I Errors) to occur, but may miss subtle attacks. Lower percentile thresholds are more sensitive to subtle attacks but may generate false alarms (Type II Error). Through experimentation, we found that a threshold percentile of 0.85 provides the best precision and recall tradeoff (~90% for each, discussed in the Results and Discussion section). Note that reconstruction error was implemented in code as show below:

**# Compute reconstruction errors to detect intrusion**

**X\_pred = best\_model.predict(X\_scaled)**

**mse = np.mean(np.square(X\_scaled - X\_pred), axis = 1) # Using mean squared error**

1. Results and Discussion
   1. Visualizations

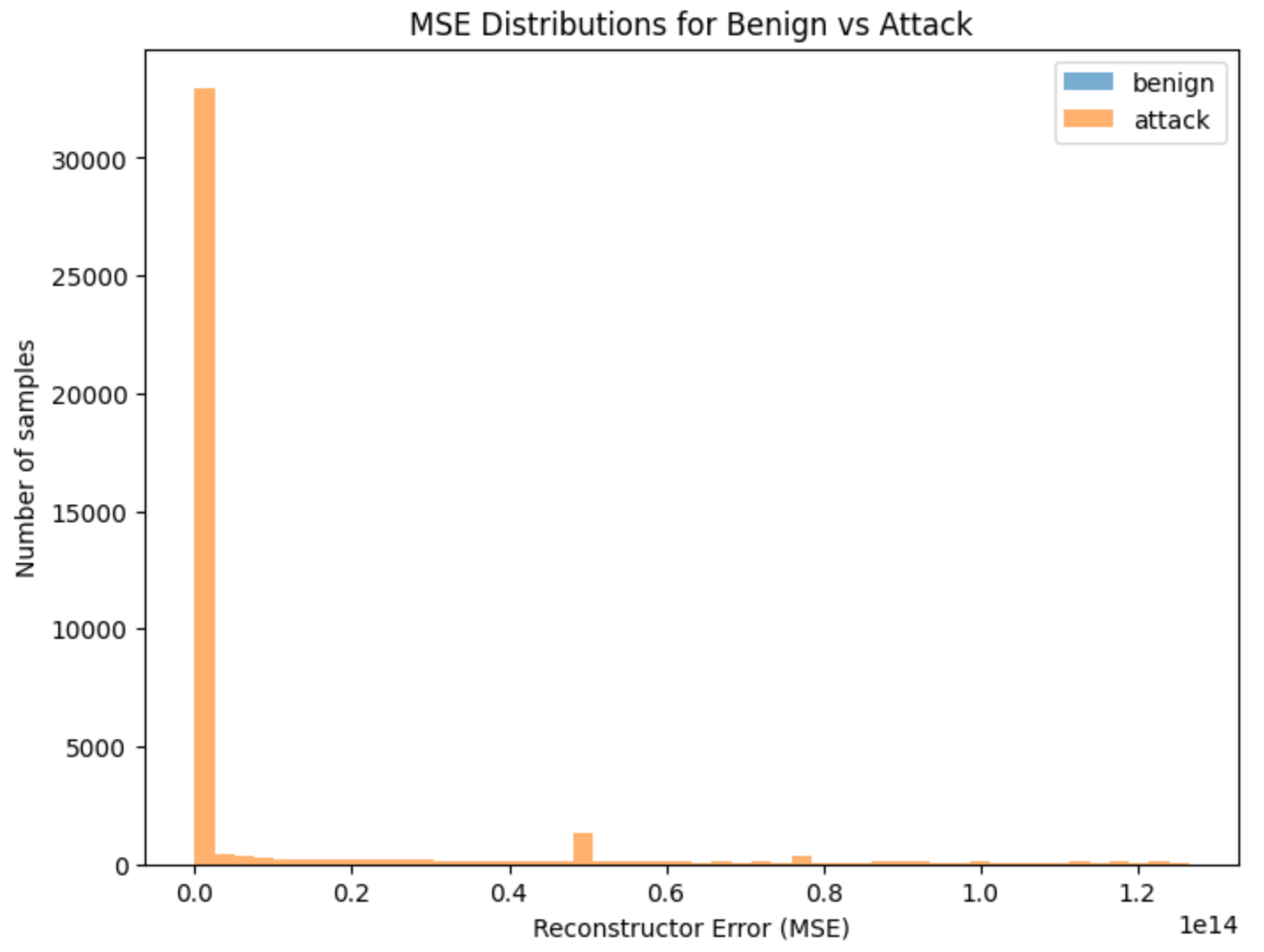


Figure 13: Visualizing reconstruction error distributions.

A screenshot of a graph

AI-generated content may be incorrect.

Figure 14: Color-coded Seaborn heatmap used to visualize the confusion matrix.

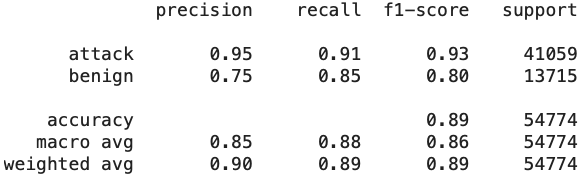


Figure 15: Precision and recall scores of ‘attack’ (positive class) and ‘benign’ (negative class) labeling.

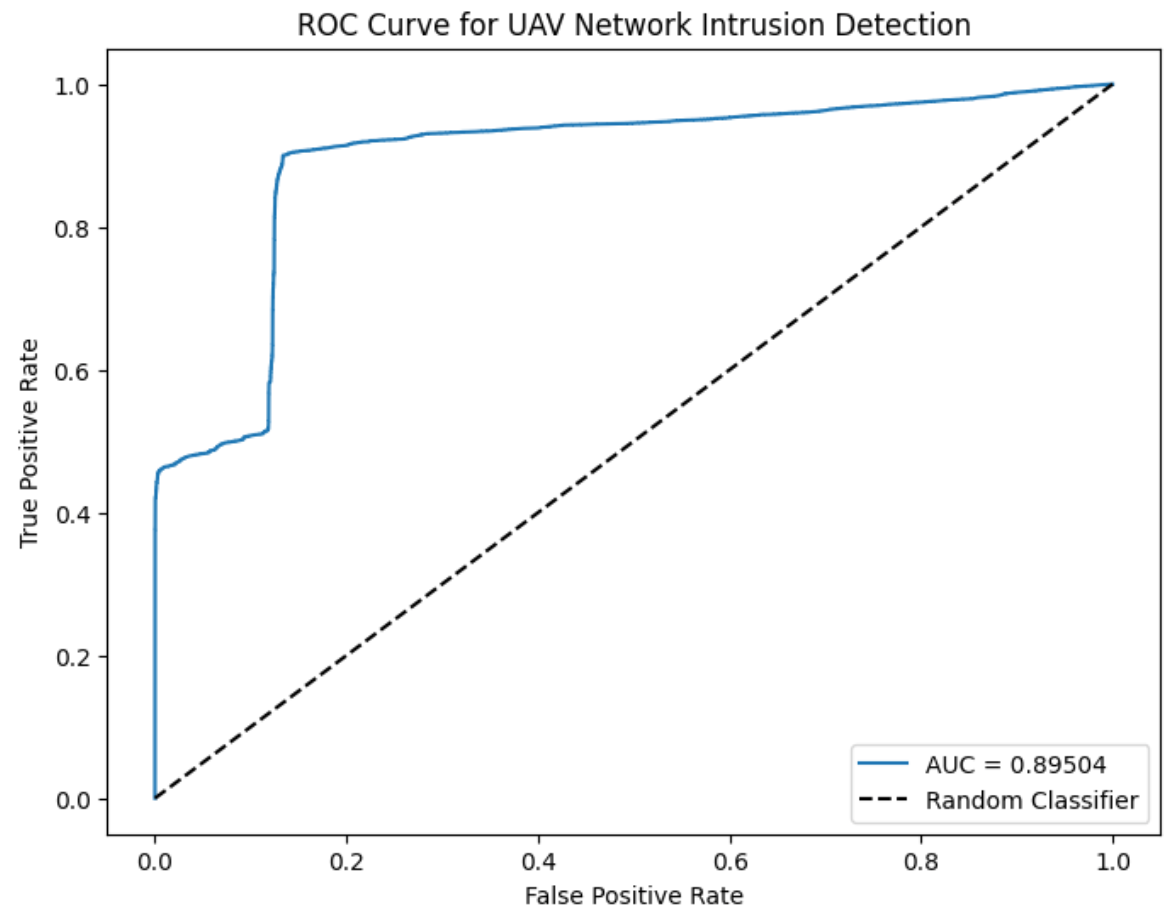


Figure 16: Receiver Operating Characteristic (ROC) curve of the Autoencoder algorithm.

* 1. Interpretation and Challenges

The majority of attacks were subtle in nature, resembling the benign network status almost exactly. This is representative of the real world where attackers aim to be as subtle as possible in most attacks, like replay attacks. This fact caused us to lower the threshold percentile to 0.85 from 0.95 when evaluating our model in order to detect as many attacks as possible. The consequence was that there were more False Positives (Type I Errors) at a lower threshold (consult the confusion matrix). This is preferred in our context because it is better to label a benign network as under attack and take precautions rather than classify an attack as benign and expose the UAV network to threats. Accordingly, the precision of classifying attacks was around 95% while that of labeling benign network states correctly was around 75%. The ROC curve featured above shows this reality, but the high AUC score of 0.89504 proves that the model can distinguish between benign networks and networks being intruded upon.

1. Conclusion and Future Work
   1. Summary

In conclusion, Autoencoder models have great potential in detecting intrusions in UAV networks and even networks of other types, like MANETs and VANETs. The Autoencoder approach has the distinct advantage of being able to detect any kind of attack, even those never seen before or included in the dataset, at the cost of not being able to distinguish between different kinds of attacks. The Autoencoder model has proven itself competent in the case of UAV networks and a pretrained autoencoder can be deployed as lightweight software on top of UAV networks as an Intrusion Detection System.

* 1. Future Research

Future research based off of this work should endeavor to include more encoder decoder blocks in our vanilla architecture. It is encouraged to tweak the input and output dimensionalities of blocks for potentially better results. The type of solver / optimizer, the dataset, and features can be switched or augmented according to the users’ needs. In the Jupyter Notebook files linked in my GitHub, I provided extensive comments that specify where changes can be made. For example, in the data preprocessing notebook there is a cell where potentially beneficial features that were dropped in our cleaned dataset were listed for use in future implementations.

Access my repository here: <https://github.com/Anjana-Parepalli/Autoencoder-Approach-to-Intrusion-Detection-in-UAV-Networks>.

1. References

[1] Mughal, U. A., Hassler, S. C., & Ismail, M. (Year). *UAVs-Dataset-Under-Normal-and-Cyberattacks [Dataset]*. GitHub. [https://github.com/uamughal/UAVs-Dataset-Under-Normal-and-Cyberattacks](https://github.com/uamughal/UAVs-Dataset-Under-Normal-and-Cyberattacks?utm_source=chatgpt.com) accessed on October 5th, 2025.

[2] [Hassler, S. C., Mughal, U. A., & Ismail, M. (2024). Cyber-Physical Intrusion Detection System for Unmanned Aerial Vehicles. *IEEE Transactions on Intelligent Transportation Systems, 25*(6), 6106-6117. https://doi.org/10.1109/TITS.2023.3339728](https://ieeexplore.ieee.org/abstract/document/10368002)

[3] Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research, 13*(Feb), 281–305. <http://www.jmlr.org/papers/v13/bergstra12a.html>

[4] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. In Proceedings of the International Conference on Learning Representations (ICLR). <https://arxiv.org/abs/1412.6980>