Detecția anomaliilor în rețele folosind Machine Learning

1st Semester of 2023-2024

Author
gabriel-lucian.tilica@s.unibuc.ro

Abstract

This project focused on the detection of anomalies in network traffic using machine learning techniques. The key goal was to develop an AI-based system capable of identifying unusual or potentially malicious activities within network environments. The UNSW-NB15 dataset was employed to train and evaluate an Autoencoder model implemented in PyTorch, which identified anomalies based on reconstruction errors.

Throughout the project, significant learning outcomes were achieved, including data preprocessing, feature selection, and the practical implementation of neural network architectures. Notable methods such as Principal Component Analysis (PCA) and Recursive Feature Elimination (RFE) were used to enhance model efficiency and accuracy. The results highlighted the potential of Autoencoders for effective anomaly detection.

For future work, I aim to expand my knowledge by exploring Isolation Forests as an anomaly detection method and comparing them with Autoencoders to assess their strengths and weaknesses. Additionally, I plan to work with diverse datasets beyond UNSW-NB15 to evaluate the system's generalizability and enhance its adaptability to various network environments. Lastly, I intend to investigate ensemble learning techniques that combine multiple models to create more robust and scalable solutions for real-time anomaly detection.

1 Introduction

The problem I am trying to solve is the detection of anomalies in network traffic to identify unusual or potentially malicious activities. This issue is critical in cybersecurity and network management, as anomalies often indicate security breaches, malware activity, Distributed Denial of Service (DDoS) attacks, or misconfigurations within the network. The key aspects of the problem I am working at are the following.Modern networks generate massive amounts of traffic, which makes it challenging for traditional methods to effectively monitor and detect anomalies. At the same time, cyberattacks and malicious activities are becoming increasingly sophisticated, necessitating advanced detection mechanisms that go beyond static rule-based systems. Many network applications also require real-time or near-real-time anomaly detection to minimize potential damage. Moreover, existing systems often produce a high number of false positives, overwhelming administrators and making it difficult to focus on genuine threats. This problem is critically important because anomalies can serve as early indicators of security breaches or attacks. Promptly detecting and addressing network issues helps prevent downtime and improves overall network performance. Additionally, automated anomaly detection significantly reduces the need for manual monitoring, leading to considerable time and resource savings. By developing an AI-based system for anomaly detection, I aim to address these challenges using machine learning models that can learn patterns in network traffic and differentiate between normal and abnormal behavior. Since I'm the only contributor to this project, it was not easy, but I tried to approach this problem by presenting an implementation of code and documenting one machine learning model for anomaly detection: Autoencoder. Using the UNSW-NB15 dataset, I ensured that the model was reproducibly implemented, with clear instructions and comments in the code. The Autoencoder was used to detect anomalies by reconstructing input data and identifying deviations. Additionally, I created a detailed explanation and tutorial for the model, outlining it's structure, how it process data, and his suitability for anomaly detection. This includes insights gained from the results. The approach I am going to take involves a thorough phase of research and study before starting the implementation. During this phase, I dedicated time to documenting and planning the project. This included learning about network traffic flows and their characteristics, such as protocols (e.g., TCP/IP, HTTP, DNS) and essential parameters like IP addresses, ports, latency, and packet size. Understanding these elements was crucial for interpreting the dataset and identifying patterns in network behavior. I focused on studying the principles behind anomaly detection models, particularly Autoencoders for reconstructing input patterns. Additionally, I explored related methods, such as clustering techniques and recurrent neural networks (RNNs), to understand their potential applicability in this domain. While I also reviewed Isolation Forests conceptually, they were not part of the implementation in this project. Furthermore, I analyzed publicly available datasets like UNSW-NB15 to familiarize myself with their structure and relevance to the problem. This included identifying features critical for anomaly detection and understanding how these features align with the behavior of the Autoencoder model I implemented. In the implementation phase, I focused on developing an Autoencoder for detecting anomalies in network traffic. First, I preprocessed the UNSW-NB15 dataset by extracting relevant features and normalizing the data to ensure compatibility with the model. The Autoencoder was trained to reconstruct normal network traffic patterns, with anomalies identified as inputs that deviated significantly from the reconstruction. The implementation was done in Python, with detailed documentation and clear instructions provided to ensure reproducibility. I evaluated the performance of the Autoencoder using metrics such as precision, recall, and F1-score, and generated visualizations to illustrate its effectiveness in detecting anomalies. Lastly, I compiled the results into a concise tutorial and presentation, highlighting the strengths and limitations of the Autoencoder approach. I chose to approach this project because I have a strong passion for cybersecurity and plan to further my studies in this field by pursuing a master's degree in Security and Applied Logic. This project aligns perfectly with my interests, allowing me to explore practical applications of machine learning in enhancing network security. Cybersecurity is a critical domain where the complexity and volume of network traffic make traditional monitoring methods less effective. The integration of AI in this field is transformative, as it enables systems to adapt to evolving threats, identify anomalies in real-time, and reduce false positives through intelligent pattern recognition. By leveraging AI techniques like Autoencoders and Isolation Forests, this project demonstrates how machine learning can provide advanced, scalable solutions for detecting malicious activities in networks. This approach not only deepens my understanding of AI's role in cybersecurity but also serves as a foundation for further academic and professional growth in this domain. Throughout this project, I gained a deeper understanding of machine learning techniques and their applications in anomaly detection. I learned how to preprocess and analyze network traffic data, particularly using the UNSW-NB15 dataset. This involved understanding the structure of network traffic, selecting relevant features, and applying normalization and dimensionality reduction techniques to prepare the data effectively. Implementing an Autoencoder model in PyTorch allowed me to detect anomalies based on reconstruction errors, which further enhanced my understanding of neural network architectures, training processes, and evaluation metrics. I also developed a strong appreciation for the importance of feature selection and its impact on model performance. By applying techniques like Principal Component Analysis (PCA) and Recursive Feature Elimination (RFE), I was able to improve the system's efficiency and accuracy. Additionally, I gained valuable experience in visualizing data and model performance using techniques like kernel density plots and loss curves, which helped me interpret the results and refine the anomaly detection system. In the future, I would like to expand my knowledge by studying Isolation Forests, a powerful anomaly detection method that isolates data points in feature space to identify outliers. Learning about this technique will allow me to compare it with Autoencoders and assess its suitability for different scenarios. I also plan to explore diverse datasets beyond UNSW-NB15 to evaluate the generalizability of my system and adapt it to different network environments. Furthermore, I aim to delve into ensemble learning techniques, such as combining Isolation Forests with neural networks or other models, to create more accurate and robust systems for real-time anomaly detection. By reflecting on my experiences in this project and setting clear goals for future learning, I hope to continue improving my understanding and application of machine learning techniques in the field of cybersecurity.

2 Approach

2.1 Learning Phase

Before starting the implementation, I dedicated time to understanding the foundational concepts and tools required for this project. This process was crucial to ensure that I could effectively develop and evaluate the anomaly detection system. Below, I outline the key aspects of this learning phase:

2.1.1 Understanding Network Traffic

I began by studying the characteristics of network traffic to understand the nature of the data I would be working with. This included:

- Learning about network protocols such as TCP/IP, HTTP, and DNS, which govern communication in networks.
- Understanding essential parameters of network traffic, including IP addresses, ports, packet size, and latency. These features play a critical role in identifying patterns and detecting anomalies.

2.1.2 Exploring Machine Learning for Anomaly Detection

Next, I focused on learning about machine learning models commonly used for anomaly detection:

- **Autoencoders:** I studied how autoencoders reconstruct input data and how deviations from reconstruction indicate anomalies. This included understanding the encoder-decoder architecture and its training process. Autoencoders is a type of neural network used in unsupervised learning to encode input data into a compressed representation and then reconstruct it. Anomalies are detected as inputs that deviate significantly from the reconstructed data, indicating unusual patterns.

- **Other Techniques:** Although not implemented in this project, I also explored related methods such as Isolation Forests, clustering techniques, and Recurrent Neural Networks (RNNs) to understand their strengths and limitations.

2.1.3 Familiarizing with Public Datasets

To ensure the system was trained on relevant and diverse data, I analyzed publicly available datasets like UNSW-NB15. This involved:

- Understanding the structure and features of the dataset.

- Identifying which features were most relevant for detecting anomalies in network traffic.

  The UNSW-NB15 dataset is a comprehensive collection of network traffic data designed to evaluate intrusion detection systems. It encompasses both normal activities and a variety of contemporary synthesized attack behaviors, making it suitable for developing and testing machine learning models for anomaly detection.

  Key Features of the UNSW-NB15 Dataset:

  Data Composition: The dataset includes a total of 2,540,044 records, divided into four CSV files. Specifically, the training set comprises 175,341 records, while the testing set contains 82,332 records.

  Attack Categories: It features nine distinct types of attacks:

  Fuzzers: Techniques that send unexpected or random input data to applications to discover vulnerabilities. Analysis: Methods involving the gathering and studying of information to exploit system vulnerabilities. Backdoors: Unauthorized access points created by inserting malicious code, allowing attackers to bypass security measures. DoS (Denial of Service): Attacks aimed at making a machine or network resource unavailable to its intended users. Exploits: Attacks that take advantage of software vulnerabilities to gain unauthorized access. Generic: Attacks that can be applied to various platforms without modification. Reconnaissance: Techniques used to gather information about a system to find ways to infiltrate it. Shellcode: Small pieces of code used as the payload in the exploitation of a software vulnerability. Worms: Malicious software programs that replicate themselves to spread to other computers. Feature Set: The dataset contains 49 features extracted using Argus and Bro-IDS tools, which are categorized into:

  Flow Features: Attributes related to the flow of data packets between source and destination. Basic Features: Fundamental attributes such as protocol type, service, and duration. Content Features: Information derived from the data portion of the packets, like the number of failed login attempts. Time Features: Attributes related to the timing of the connections, such as the time to live (TTL) of the packets. Additional Generated Features: Features created to provide more insights, like the number of compromised conditions.

2.2 Implementation Phase

After completing the learning phase, I proceeded to implement the project by setting up the necessary environment and adding core files. The following steps outline the implementation process:

2.2.1 Project Setup

The project was developed in Visual Studio Code (VSCode), providing a structured development environment to manage, edit, and execute the code. The initial setup focused on preparing the environment and incorporating essential files for anomaly detection.

- **Adding the Autoencoder Implementation:** The primary file added to the project is an existing implementation of an Autoencoder-based anomaly detection system tailored to the UNSW-NB15 dataset. This file, sourced from `https://github.com/alik604/cyber-security/blob/master/Intrusion-Detection/UNSW_NB15%20-%20PyTorch%20MLP%20and%20autoEncoder.ipynb`, contains:

- Data preprocessing steps specific to the UNSW-NB15 dataset, including feature extraction and normalization.
- An Autoencoder architecture implemented in PyTorch, designed to reconstruct network traffic patterns and detect anomalies based on reconstruction loss.
- Visualization and evaluation metrics to assess the model's effectiveness in identifying anomalies.

This file was integrated into the project as the core implementation for detecting anomalies.

- **Creating the `requirements.txt` File:** To manage dependencies and ensure reproducibility, a `requirements.txt` file was created. This file lists all the Python libraries necessary for running the project, including:
  - `torch` for building and training the Autoencoder.
  - `scikit-learn` for preprocessing and evaluation metrics.
  - `tabulate` for presenting results in a structured format.
  - `pandas` and `numpy` for handling and manipulating the dataset.
  - `matplotlib` for visualizations.
  - `ipykernel` to enable running the notebook in a Jupyter environment.

  This file ensures that all dependencies can be installed with a single command.

- **Creating the `setup.txt` File:** A `setup.txt` file was created to provide clear and concise instructions for setting up and running the project. The file includes:
  - Instructions to create a virtual environment using:
    ```
    python3 -m venv myenv
    ```
  - Steps to activate the virtual environment:
    ```
    source myenv/bin/activate
    ```
  - Commands to install the required libraries:
    ```
    pip3 install -r requirements.txt
    ```

  These steps ensure that the project environment can be replicated on any machine.

2.3 Understanding the Provided ipynb file

After setting up the project environment, the next step was to incorporate and understand the implementation provided in the notebook `UNSW_NB15 - PyTorch MLP and autoEncoder.ipynb`. This file contains the core implementation for anomaly detection using an Autoencoder. The following steps were undertaken to fully comprehend and adapt this notebook for the project:

2.3.1 Setup and Library Imports

The provided code begins by configuring the environment and importing the necessary libraries for developing an anomaly detection system. Each section of this setup plays a specific role in ensuring the proper execution of the project. First, the following line is used to enhance the Jupyter Notebook environment:

```
%config IPCompleter.greedy=True
```

This configuration enables comprehensive auto-completion features for methods, attributes, and variables, streamlining the development process and reducing potential errors.

Next, several essential Python libraries are imported:

* **Pandas:** Facilitates the manipulation and analysis of structured data in tabular form.
* **NumPy:** Provides tools for efficient numerical computations and array operations.
* **SciPy:** Adds advanced mathematical functions useful for scientific computing.
* **Matplotlib:** Includes tools for creating data visualizations. The `%matplotlib inline` command ensures that all plots are displayed directly within the notebook.
* **Scikit-learn:** Offers a suite of machine learning tools for

* preprocessing, model training, evaluation, and more. Key modules like `train_test_split`, `preprocessing`, and `ensemble` are used in this code.
* **PyTorch:** A deep learning library used to implement and train the Autoencoder model.

Additionally, the configuration `InteractiveShell.ast_node_interactivity` is set to display all outputs of a Jupyter Notebook cell. This is particularly useful for debugging and monitoring progress during execution. A variable, `DLed`, is also initialized with a default value, though its role is not immediately apparent in this snippet.

This setup phase ensures that the environment is prepared with all the necessary tools and configurations for successfully analyzing network traffic data and applying machine learning techniques.

Loading and Preparing the Dataset

The provided code is responsible for downloading, combining, and preparing the UNSW-NB15 dataset for analysis. It begins by checking if the dataset has already been downloaded. If not, it retrieves the training and testing datasets from their respective online links. These datasets are then merged into a single DataFrame to streamline subsequent processing. Unnecessary columns, such as "id" and "label," are removed to focus only on the relevant features for analysis. Lastly, the code outputs the dimensions and a preview of the dataset, ensuring the data is correctly loaded and structured for further operations.

Calculating Proportion of Normal Traffic in Datasets

This code calculates and prints the proportion of "Normal" attack categories in both the training and testing datasets of the UNSW-NB15 dataset. It does so by filtering rows where the attackcat column equals "Normal" and then computing the ratio of these rows to the total rows in each dataset. The results indicate how much of the data in the training and testing sets corresponds to normal (non-attack) traffic. Lastly, there is a commented-out line to delete the train and test datasets from memory, likely intended for freeing resources after their use.

Encoding Categorical Features and Describing Attack Types

This code block processes the categorical features within the dataset and provides descriptive insights into the attack categories. Initially, it uses a ‘LabelEncoder‘ to transform textual categorical columns (‘attackcat‘, ‘proto‘, ‘service‘, and ‘state‘) into numeric values, facilitating further machine learning tasks.

The attack categories (‘attackcat‘) are printed as a list to showcase the distinct types present in the dataset. Subsequently, the mode (most frequent value) of the ‘attackcat‘ column is calculated and displayed, along with its percentage occurrence in the dataset. These steps help identify the prevalence of "normal" traffic (assigned the encoded value of 6) and highlight its relative rarity within the dataset.

Overall, this portion of the code standardizes categorical variables and begins exploratory data analysis by examining the distribution of attack types.

Counting and Displaying Attack Categories

This code segment utilizes the ‘collections.Counter‘ module to count the occurrences of each unique attack type in the ‘attackcat‘ column. By leveraging the ‘tabulate‘ library, the counts are formatted into a tabular representation with headers ‘Type‘ and ‘Occurrences‘, making it easier to interpret the distribution of attack categories.

The resulting table provides an overview of the frequency of each attack type in the dataset, aiding in understanding the data’s class imbalance and guiding further analysis or preprocessing steps.

Identifying Low-Variance and Low-Correlation Features

This code segment focuses on identifying features in the dataset with minimal variance and low correlation to the target variable, ‘attackcat‘.

First, a copy of the ‘combined$_{data}$‘$'Dataframe'iscreatedtoensuretheoriginaldatasetremainsunchangedduringprocessing$

The low-variance features are identified using the standard deviation of each column. The ‘std()‘ method computes the standard deviation for all features, and the ‘nsmallest()‘ function selects the seven features with the smallest values. These features are stored in the ‘lowSTD‘ list.

Next, the code calculates the absolute correlation of each feature with the ‘attackcat‘ column using the ‘corr()‘ method. Features with the lowest correlation to ‘attackcat‘ are identified using the ‘nsmallest()‘ function and stored in the ‘lowCORR‘ list. This step helps pinpoint features that are least relevant to predicting the target variable.

The identified low-variance and low-correlation features may be considered for removal or further analysis to improve model performance by reducing noise and redundancy.

Performing Dimensionality Reduction and Updating the Dataset

This section of the code is responsible for performing dimensionality reduction on selected features in the dataset and subsequently updating the dataset with reduced features. The key steps are as follows:

Initially, a list named `exclude` is created by combining features with low correlation (`lowCORR`) and low standard deviation (`lowSTD`). The `attack_cat` column, which serves as the target variable, is explicitly removed from this list if it exists. This ensures that the target variable is not included in the dimensionality reduction process.

The code then prints the shape of the dataset before dimensionality reduction and displays the features identified for replacement with Principal Component Analysis (PCA).

Next, PCA is applied to the columns specified in the `exclude` list, with the number of principal components set to three. This process reduces the dimensionality of the dataset while retaining as much variance as possible. The `explained_variance_ratio_` is calculated and printed, providing insight into how much variance is captured by the reduced features.

After PCA is performed, the features listed in `exclude` are dropped from the dataset using the `drop` method. The reduced features are stored in a DataFrame and joined with the updated dataset to maintain a consistent structure.

Finally, the code prints the shape of the dataset after dimensionality reduction, showcasing the changes made to the dataset. The updated dataset now includes the reduced features in place of the original ones, which simplifies the dataset while preserving its predictive power.

Scaling the Duration Feature

This section of the code performs a scaling operation on the `dur` (duration) feature within the dataset. The primary objective is to transform the values of this feature to a larger scale for better numerical representation during model training and analysis.

The operation scales the `dur` feature by multiplying its values by 10,000. This scaling ensures that the feature has a more significant numerical range, which might help certain machine learning models that are sensitive to the magnitude of feature values.

After scaling, the code prints a message indicating that the `dur` feature has been scaled up by a factor of 10,000. Additionally, the `head()` method is called on the `combined_data` DataFrame to display the first few rows of the dataset, allowing verification of the scaling operation.

Preparing the Data for Model Training

This section of the code prepares the dataset for model training by splitting it into features (`data_x`) and labels (`data_y`), followed by normalizing the feature values.

The process begins by printing the shape of the original `combined_data` DataFrame. The `data_x` variable is created by dropping the `attack_cat` column, which serves as the label for classification. The `data_y` variable is assigned the `attack_cat` column, representing the target variable for the machine learning model. The shapes of both `data_x` and `data_y` are printed to verify the separation.

Normalization is applied to the `data_x` features using a lambda function. This function scales each feature to the range [0, 1] by subtracting the minimum value and dividing by the range (maximum - minimum). This ensures that all features are on a consistent scale, which is essential for machine learning models sensitive to feature magnitudes.

Two commented-out lines provide alternative normalization techniques:

* `MinMaxScaler()`: Normalizes data to the range [0, 1] and is noted as better suited for `VotingClassifier`.
* `StandardScaler()`: Standardizes data to have a mean of 0 and a standard deviation of 1.

These options provide flexibility depending on the requirements of different machine learning models or algorithms.

Splitting the Data into Training and Testing Sets

This portion of the code splits the prepared dataset into training and testing subsets using the `train_test_split` function from the `sklearn.model_selection` module. The variables `X_train` and `X_test` represent the feature data for training and testing, while `y_train` and `y_test` represent the corresponding labels.

The split is configured with the following parameters:

* `test_size=0.50`: Specifies that 50% of the data will be allocated to the test set, and the remaining 50% will be used for training. The comment suggests that the `test_size` value can be adjusted depending on computational constraints and how long the user is willing to wait for processing.
* `random_state=42`: Ensures reproducibility of the split by using a fixed random seed.

This step is crucial for evaluating the model's performance, as it creates separate datasets for training the model and testing its accuracy on unseen data. The split ensures that the evaluation is unbiased and reflects the model's generalization capabilities.

Implementing and Evaluating Ensemble and Individual Models

This portion of the code implements three individual classifiers and an ensemble model to evaluate their performance on the training and testing datasets:

1. Classifiers Used:

* `DecisionTreeClassifier (DTC)`: A simple decision tree-based model that splits the data recursively based on features to make predictions.
* `RandomForestClassifier (RFC)`: An ensemble method that combines multiple decision trees, with 50 trees specified via the `n_estimators` parameter and a random seed for reproducibility (`random_state=1`).
* `ExtraTreesClassifier (ETC)`: Another ensemble method similar to `RandomForestClassifier`, with 75 trees (`n_estimators=75`) and customized hyperparameters, such as the `gini` criterion for splitting and no bootstrap sampling (`bootstrap=False`).

2. Voting Classifier: The ensemble model, `VotingClassifier`, combines the predictions of the three classifiers mentioned above. It uses `voting='hard'`, which means the final prediction is based on majority voting among the classifiers.

3. Training and Evaluation: The `for` loop iterates through the three individual classifiers (`DTC`, `RFC`, and `ETC`) and the ensemble model (`eclf`). For each classifier:

* `clf.fit(X_train, y_train)`: Trains the classifier on the training data.
* `clf.score(X_test, y_test)`: Evaluates the classifier on the test data and returns the accuracy score.
* The accuracy score is printed with the classifier's name for comparison: `Acc: %0.7f [%s]`.

This step allows for a direct comparison of the performance of individual classifiers and the ensemble model, providing insights into which approach performs best for the given dataset.

Feature Selection and Model Evaluation

This code combines feature selection techniques with ensemble and individual model evaluations to improve classification performance.

1. Feature Selection with Recursive Feature Elimination (RFE): The `RFE` (Recursive Feature Elimination) method is used to select the top 10 most important features from the dataset:

* `RFE(DecisionTreeClassifier(), 10).fit(X_train, y_train)`: This initializes an `RFE` object with a `DecisionTreeClassifier` as the estimator and identifies the 10 most significant features by iteratively removing the least important ones.
* `np.where(rfe.support_==True)[0]`: Identifies the indices of the selected features.
* The selected feature names are stored in the variable `whitelist`, and the training and testing datasets are reduced to these features using `X_train[whitelist]` and `X_test[whitelist]`.

2. Model Initialization: Three classifiers are defined:

* `DecisionTreeClassifier (DTC)`: A simple decision tree model.
* `RandomForestClassifier (RFC)`: An ensemble of 50 decision trees with a random seed for reproducibility.
* `ExtraTreesClassifier (ETC)`: An ensemble method with 75 decision trees, using `gini` as the splitting criterion and no bootstrap sampling.

3. Evaluation with Original and Reduced Features:

* The classifiers are trained and evaluated twice: once with the full feature set (`X_train` and `X_test`) and once with the reduced feature set (`X_train_RFE` and `X_test_RFE`).
* The accuracy scores for the classifiers are compared to evaluate the impact of feature selection on performance.

4. Voting Classifier: The ensemble model `VotingClassifier` combines the predictions of the three classifiers using majority voting (`voting='hard'`). The performance of the ensemble model is also evaluated.

5. Output: For each classifier (including the ensemble), the accuracy score is printed in the format `Acc: %0.7f [%s]` to compare the effectiveness of the classifiers on the full and reduced feature sets.

This code demonstrates the integration of feature selection techniques with machine learning models to enhance classification accuracy and reduce model complexity.

Inspecting Feature-Reduced Dataset and Target Labels

This code performs two actions to inspect the data after applying feature selection and to understand the distribution of target labels:

1. Checking the Shape of the Feature-Reduced Training Dataset:

* `X_train_RFE.shape` prints the dimensions of the feature-reduced training dataset after applying Recursive Feature Elimination (RFE). This provides insight into how many features and samples are retained for model training.

2. Inspecting the Unique Labels in the Target Variable:

* `set(y_train)` returns the unique values in the target variable `y_train`. This reveals the distinct classes present in the training labels, ensuring that all expected classes are accounted for.

These actions are essential for verifying the data preparation process and confirming that the feature reduction and label assignments are correctly handled.

Defining the Neural Network Architecture

This segment of code defines a fully connected neural network (FCNN) for anomaly detection, specifying its architecture, parameters, and behavior.

1. Selecting the Device: The code checks if a CUDA-compatible GPU is available for computation. If a GPU is available, the device is set to `'cuda'`; otherwise, it defaults to `'cpu'`. The second assignment, `device = 'cpu'`, explicitly overrides this, ensuring that the model runs on the CPU.

2. Setting Hyperparameters: The code initializes several hyperparameters critical to training the neural network:

- `input_size`: The number of input features, set to 10.
- `hidden_size` and `hidden_size_2`: The sizes of the two hidden layers, each with 64 neurons.
- `num_classes`: The number of output classes, set to 10.
- `num_epochs`: The number of training iterations, set to 40.
- `batch_size`: The size of mini-batches used during training, set to 32.
- `learning_rate`: The step size for the optimizer, set to 0.001.

3. Defining the Neural Network Class: The `NeuralNet` class defines a custom neural network architecture using the PyTorch `nn.Module` base class:

- The `__init__()` method initializes the layers and activation functions:
  - `self.l1`: A fully connected layer (`nn.Linear`) connecting the input layer to the first hidden layer.
  - `self.l2`: A fully connected layer connecting the first hidden layer to the second hidden layer.
  - `self.l3`: A fully connected layer connecting the second hidden layer to the output layer.
  - `self.relu`: A Rectified Linear Unit (ReLU) activation function applied after the first and second layers to introduce non-linearity.
  - `self.elu`: An Exponential Linear Unit (ELU) activation function, though it is defined but not currently used in the `forward()` method.
- The `forward()` method defines the forward pass of the network:
  - The input passes through the first layer (`l1`) and then through the ReLU activation function.
  - The output of the first layer is passed through the second layer (`l2`) and another ReLU activation.
  - Finally, the data is passed through the third layer (`l3`) to produce the network's output. Note that no activation function or softmax is applied at the output, as it is likely handled separately, depending on the task.

This modular architecture facilitates easy experimentation with different activation functions, layer sizes, and other hyperparameters.

Training the Neural Network

This segment of the code is responsible for training the neural network model on the reduced feature set using the selected hyperparameters, loss function, and optimizer.

1. Model Initialization:

- The `NeuralNet` model is instantiated using the previously defined architecture and configured with the `input_size`, `hidden_size`, and `num_classes` parameters. The model is moved to the specified `device` (CPU in this case).

2. Loss Function:

- The loss function used is `nn.CrossEntropyLoss`, which combines `nn.LogSoftmax` and `nn.NLLLoss` in one class. It is well-suited for multi-class classification problems.

3. Optimizer:

- The `torch.optim.Adam` optimizer is selected to update the model parameters. It uses an adaptive learning rate (`learning_rate=0.001`) to optimize the training process efficiently.

4. Training Loop:

- The training process runs for a specified number of epochs (`num_epochs=40`).
- The training data is iterated in mini-batches, with the batch size set to `batch_size=32`.
- For each mini-batch:
  - The input features (`x`) and corresponding labels (`y`) are converted

into PyTorch tensors and moved to the specified device.
- The model performs a forward pass by calculating the predictions (`outputs`) based on the input features.
- The loss between the predicted and actual labels is computed using the `criterion`.
- Gradients of the loss with respect to the model parameters are calculated using `loss.backward()`.
- The model parameters are updated using `optimizer.step()`.

5. Progress Logging:

* Every 10 epochs, the training progress is logged, showing the epoch number, the number of steps completed, and the current loss value.

This iterative process allows the model to learn patterns in the data by minimizing the loss function over successive epochs.

Evaluating the Neural Network

This section of the code evaluates the trained neural network's performance on the test dataset. It calculates the model's accuracy using the test data while ensuring efficient memory usage by disabling gradient computations.

1. Preparing the Test Data:

* The test data features (`X_test_RFE_vals`) and labels (`y_test_vals`) are loaded and stored as NumPy arrays.

2. Disabling Gradient Calculations:

* The `torch.no_grad()` context manager is used to disable gradient calculations during the test phase. This reduces memory usage and speeds up computations.

3. Iterating Through Test Data:

* The test data is processed in mini-batches, with the batch size set to `batch_size=32`.
* For each mini-batch:
  - Input features (`x`) and corresponding labels (`y`) are converted into PyTorch tensors and moved to the specified device.
  - The model performs a forward pass on the input features to generate predictions (`outputs`).

4. Prediction and Accuracy Calculation:

* For the model outputs, the predicted class for each sample is determined using `torch.max(outputs.data, dim=1)`, which returns the index of the maximum value along the specified dimension.
* The total number of samples (`n_samples`) and the number of correct predictions (`n_correct`) are updated.

5. Error Handling:

* If the output data is empty, an error message is printed, and the input features (`x`) and outputs are displayed for debugging purposes.

6. Accuracy Calculation:

* The accuracy of the model is computed as:

$$Accuracy = \frac{NumberofCorrectPredictions}{TotalNumberofSamples} \times 10$$

* The computed accuracy is displayed in percentage format.

This evaluation process provides a quantitative measure of the model's performance on unseen data.

Defining the Autoencoder Components: Encoder and Decoder

This segment of the code defines the Encoder and Decoder classes, which together form the building blocks of an autoencoder neural network. These components are implemented using the PyTorch `nn.Module` base class.

1. Setting the Device:

* The `device` variable is explicitly set to `'cpu'`, indicating that all computations will be performed on the CPU.

2. Encoder Class: The `Encoder` class processes the input data to generate a compressed representation in a lower-dimensional space.

* `__init__()`: Initializes two fully connected layers and an activation function:

        - `self.l1`: A linear layer mapping the input size to the `hidden_size`.
        - `self.l2`: A linear layer mapping the `hidden_size` to the `center_size`, which represents the bottleneck or compressed feature space.
        - `self.acti`: An Exponential Linear Unit (ELU) activation function introducing non-linearity.
    - ∗ `forward()`: Defines the forward pass of the encoder:
        - Input `x` is passed through the first linear layer (`l1`).
        - The activation function (`acti`) is applied to introduce non-linearity.
        - The transformed data is passed through the second linear layer (`l2`), producing the compressed representation.

3. Decoder Class: The `Decoder` class reconstructs the original data from the compressed representation.

- ∗ `__init__()`: Initializes two fully connected layers and an activation function:
    - `self.l1`: A linear layer mapping the `center_size` back to the `hidden_size`.
    - `self.l2`: A linear layer mapping the `hidden_size` to the `output_size`, which matches the original input dimension.
    - `self.acti`: An Exponential Linear Unit (ELU) activation function for non-linearity.
- ∗ `forward()`: Defines the forward pass of the decoder:
    - Input `x` (compressed representation) is passed through the first linear layer (`l1`).
    - The activation function (`acti`) is applied.
    - The transformed data is passed through the second linear layer (`l2`) to reconstruct the original input.

4. Summary:

- ∗ The `Encoder` reduces the dimensionality of the input data, while the `Decoder` attempts to reconstruct the original input from this compressed representation.
- ∗ These classes are essential for building and training an autoencoder, a type of neural network used for unsupervised learning and dimensionality reduction.

Defining the Recurrent Autoencoder Class

The `RecurrentAutoencoder` class combines the previously defined `Encoder` and `Decoder` classes into a complete autoencoder architecture. It is built using the PyTorch `nn.Module` base class.

1. Class Initialization:

- ∗ The `__init__()` method initializes the encoder and decoder components of the autoencoder:
    - `self.encoder`: An instance of the `Encoder` class, which compresses the input data into a lower-dimensional representation.
    - `self.decoder`: An instance of the `Decoder` class, which reconstructs the original input from the compressed representation.
    - Both the encoder and decoder are moved to the specified `device` (e.g., `'cpu'`).
- ∗ The architecture of the encoder and decoder is printed for verification using `print(self.encoder)` and `print(self.decoder)`.

2. Forward Pass:

- ∗ The `forward()` method defines the forward propagation logic:
    - The input data (`x`) is passed through the `encoder` to obtain a compressed representation (`encoded`).
    - The compressed representation is then passed through the `decoder` to produce the reconstructed data (`dencoded`).
    - The output of the decoder (`dencoded`) is returned as the final result.

3. Summary:

- ∗ The `RecurrentAutoencoder` encapsulates the complete autoencoder structure by combining an

encoder and a decoder. It processes input data, compresses it into a latent representation, and reconstructs the original input.
    * This class serves as the core architecture for training and testing the autoencoder on a given dataset.

Training the Autoencoder Model

The `train_model` function is responsible for training the autoencoder model using the provided training and validation datasets. This function includes mechanisms for optimization, loss evaluation, and tracking performance across epochs.

Function Parameters:

* `model`: The autoencoder model to be trained.
* `train_dataset`: The dataset used for training the model.
* `val_dataset`: The dataset used for validating the model's performance.
* `n_epochs`: The number of epochs for training.
* `batch_size`: The size of each batch used during training and validation.
* `lr`: The learning rate for the optimizer.

Training Procedure:

1. **Initialization:**
    * An Adam optimizer is initialized with the model parameters and the specified learning rate (`lr`).
    * A learning rate scheduler (`ReduceLROnPlateau`) adjusts the learning rate based on validation loss, reducing it if no improvement is observed for 3 epochs.
    * The loss function used is Mean Squared Error (`MSELoss`) with reduction set to `sum`.
    * `history`: A dictionary is created to track training and validation losses across epochs.
    * `best_model_wts`: A copy of the model's initial weights is stored to retain the best-performing model.
2. **Epoch Loop:** The training process iterates for the specified number of epochs (`n_epochs`):
    * **Training Phase:**
        · The model is set to training mode using `model.train()`.
        · For each batch in the training dataset:
        · The true sequence (`seq_true`) is fed into the model to generate predictions (`seq_pred`).
        · The loss between predicted and true sequences is calculated using `criterion`.
        · Gradients are computed via backpropagation, and the optimizer updates the model's weights.
        · The training loss for each batch is recorded and averaged for the epoch.
    * **Validation Phase:**
        · The model is set to evaluation mode using `model.eval()`.
        · For each batch in the validation dataset:
        · The true sequence is passed through the model to generate predictions.
        · The loss is computed and recorded for each batch.
        · The validation loss for each batch is averaged for the epoch.
3. **Loss Tracking and Scheduler Update:**
    * Training and validation losses are stored in the `history` dictionary.
    * The learning rate scheduler adjusts the learning rate based on the validation loss.
    * If the current validation loss is lower than the best observed loss, the model's weights are saved as the best weights.
4. **Model Finalization:** After all epochs, the model's weights are restored to the best-performing configuration, ensuring the returned model represents the best version encountered during training.

Output:

- The function returns the trained model in evaluation mode and the `history` dictionary containing training and validation loss trends.

2.4 Splitting and Normalizing the Dataset

In this section of the code, the dataset is split into two subsets: one representing *normal* network traffic and the other representing *attack* traffic. Additionally, normalization is applied to ensure that features are scaled for consistency in subsequent analysis. An optional experimental feature selection process is included, controlled by the `EXPERIMENT` flag.

Overview of the Code:

- The `EXPERIMENT` variable determines whether to apply additional filtering to the dataset.
- If `EXPERIMENT = True`, a statistical analysis of feature differences between *normal* and *attack* traffic is conducted to retain only significant features.
- If `EXPERIMENT = False`, the dataset is simply split into two subsets, and min-max normalization is applied.

Steps When `EXPERIMENT = True`:

1. The dataset is divided into two groups:
   - *Normal traffic:* Rows where `attack_cat` equals 6.
   - *Attack traffic:* Rows where `attack_cat` is not 6.
2. Both groups are normalized using min-max scaling.
3. Statistical summaries (`mean` and `std`) are computed for each feature in the two subsets.
4. The absolute differences in mean and standard deviation between the two subsets are calculated.
5. Features with a mean difference greater than 0.1 are retained. These features are then used to filter the *normal* and *attack* datasets.

Steps When `EXPERIMENT = False`:

1. The dataset is split into *normal* and *attack* subsets based on the value of `attack_cat`.
2. Min-max normalization is applied to both subsets, scaling all features to a range of 0 to 1.

Final Output:

- The variable `normal` contains rows corresponding to normal network traffic with the `attack_cat` column removed.
- The variable `attack` contains rows corresponding to attack traffic with the `attack_cat` column removed.
- The dimensions of the `normal` subset are printed for verification.

This step ensures that the dataset is prepared for further analysis or modeling by segregating the data and applying normalization. Additionally, the optional feature selection process helps reduce dimensionality and retain only the most relevant features for anomaly detection.

2.5 Preparing Data for Training and Testing

In this section, the dataset is further divided and converted into tensors for use in a PyTorch-based model. The code processes the `normal` and `attack` subsets to create training and testing datasets specifically for normal traffic.

Steps in the Code:

1. The `normal` and `attack` datasets, initially stored as pandas DataFrames, are converted to NumPy arrays using `.values`. This conversion is required to prepare the data for PyTorch tensor operations.
2. The `normal` dataset is split into two parts:

    * `normal_train`: 90% of the data is allocated for training.
    * `normal_test`: The remaining 10% is reserved for testing.

3. The training and testing subsets are converted to PyTorch tensors using `torch.as_tensor()`, with the data type specified as `float` for numerical computations. These tensors are moved to the device (CPU or GPU) specified earlier in the script.

4. Additionally, the entire `normal` and `attack` datasets are converted into PyTorch tensors (`normal_tensor` and `attack_tensor`), which may be used for evaluation or analysis purposes.

Output:

* The shapes of `attack_tensor`, `normal`, `x_train`, and `x_test` are printed to verify the dimensions of the data at each stage of processing.

* This ensures that the datasets are correctly divided and converted into tensors, which are compatible with PyTorch models.

This step prepares the data for subsequent training and testing phases, ensuring that the model receives appropriately formatted input for anomaly detection.

2.6 Training the Recurrent Autoencoder Model

This section describes the setup and training of the Recurrent Autoencoder model. The code initializes model parameters, trains the model, and saves it for future use.

Steps in the Code:

1. **Input and Output Dimensions:** The `input_size` and `output_size` are both set to the number of features in the `normal` dataset, obtained using `normal.shape[1]`. This ensures that the input and output layers of the autoencoder match the dimensionality of the data.

2. **Hidden and Center Sizes:**
    * `hidden_size` is set to 7, defining the size of the hidden layers in the encoder and decoder.
    * `center_size` is set to 2, representing the size of the latent representation learned by the autoencoder.

3. **Training Hyperparameters:**
    * `num_epochs`: The number of epochs for training is set to 25.
    * `batch_size`: The batch size for training is set to 16, defining how many samples are processed together in a single forward and backward pass.
    * `lr`: The learning rate is set to 0.01, controlling the step size of the optimizer.

4. **Model Initialization:** A `RecurrentAutoencoder` instance is created using the defined parameters for `input_size`, `output_size`, `hidden_size`, and `center_size`. The model is moved to the specified device (CPU or GPU).

5. **Training the Model:** The `train_model` function is called with the following arguments:
    * `model`: The initialized Recurrent Autoencoder.
    * `x_train` and `x_test`: Training and testing datasets prepared earlier.
    * `n_epochs`, `batch_size`, and `lr`: The defined training hyperparameters.

This function trains the model using Mean Squared Error (MSE) as the loss function and returns the trained model along with a history of training and validation losses for analysis.

6. **Saving the Model:** After training, the model is saved to a file named `model.pth` using `torch.save`. This allows the trained model to be loaded and reused without retraining.

Purpose of this Step: This step trains the Recurrent Autoencoder to learn latent representations of the `normal` network traffic. The trained model will be used in subsequent phases to detect anomalies based on deviations from normal patterns.

2.7 Visualizing Training and Validation Loss

This code snippet generates a visualization of the training and validation loss over the course of the training epochs. The plot helps evaluate how well the model has learned during training and whether overfitting or underfitting has occurred.
Description of the Code:

* A figure and its axes are created using `plt.figure().gca()`, where `gca` returns the current axes.
* The training loss, stored in `history['train']`, is plotted over the epochs.
* The validation loss, stored in `history['val']`, is plotted for comparison.
* Labels for the x-axis (`Epoch`) and y-axis (`Loss`) are added to describe the plot.
* A legend is added to differentiate between the training and validation loss curves.
* A title, `Loss over training epochs`, is included to specify the purpose of the plot.
* The plot is displayed using `plt.show()`.

Purpose of this Visualization: This plot is crucial for understanding the model's performance during training:

* It helps identify if the model is converging by observing whether the loss values decrease over epochs.
* A significant gap between training and validation loss may indicate overfitting.
* Similar trends in training and validation loss suggest the model generalizes well to unseen data.

2.8 Prediction and Loss Calculation

The following code defines a function named `predict`, which is responsible for generating predictions and calculating the corresponding reconstruction loss for a given dataset. This function is typically used during the evaluation phase to assess the model's performance.
Description of the Code:

* The function `predict` takes three arguments:
  - `model`: The trained autoencoder model used for predictions.
  - `batch_size`: The number of samples processed in a single forward pass, which helps manage memory usage.
  - `dataset`: The input data for which predictions and losses are computed.
* Two lists, `predictions` and `losses`, are initialized to store the predicted outputs and corresponding reconstruction losses for each batch.
* The loss function is defined using `nn.L1Loss`, which calculates the mean absolute error (MAE) between the input and the reconstructed output. The `reduction='sum'` argument ensures that the loss is summed across all elements.
* The model is set to evaluation mode using `model.eval()` to disable dropout and batch normalization, ensuring consistent predictions.

- The dataset is processed in batches using a loop. For each batch:
  - The true input sequence, `seq_true`, is passed through the model to generate the predicted output, `seq_pred`.
  - The reconstruction loss between `seq_true` and `seq_pred` is calculated using the specified criterion.
  - The predicted outputs are appended to the `predictions` list, and the corresponding loss values are appended to the `losses` list.
- The function returns two outputs:
  - `predictions`: A list of the predicted outputs generated by the model.
  - `losses`: A list of the reconstruction loss values for each batch.

Purpose of the Function: This function is crucial for evaluating the model's reconstruction ability:

- By computing predictions and losses, it helps identify how well the model reconstructs input data.
- Loss values can be analyzed to detect anomalies, as higher losses often correspond to unusual or anomalous inputs.
- It provides a batch-wise processing mechanism to handle large datasets efficiently.

2.9 Loss Calculation for Different Data Sets

This code snippet calculates the reconstruction losses for different subsets of data using the trained autoencoder model. These losses are essential for understanding how well the model reconstructs data and for identifying anomalies.

Description of the Code:

- `x_train.size()`: This line retrieves the size (dimensions) of the training dataset, providing insight into the number of samples and features used for training.
- The `predict` function is used to calculate reconstruction losses for four subsets of data:
  - `x_train`: The training data used to train the autoencoder.
  - `x_test`: The testing data used to evaluate the autoencoder's generalization capability.
  - `attack_tensor`: Data representing anomalous samples (e.g., attacks) from the dataset.
  - `normal_tensor`: Data representing normal samples from the dataset.
- For each subset, the `predict` function is called:
  - The first returned value (_) contains the predicted outputs, which are not used here.
  - The second returned value (`losses_train`, `losses_test`, `losses_attack`, `losses_normal`) stores the reconstruction losses for each subset.

Purpose of the Code:

- The reconstruction losses provide a quantitative measure of how well the autoencoder performs on each subset of data.
- Higher reconstruction losses for `attack_tensor` compared to `normal_tensor` are expected, as anomalies typically deviate from the patterns learned by the autoencoder.
- Losses for `x_test` indicate the model's performance on unseen data, highlighting its generalization capability.
- These loss values can be further analyzed to define thresholds for anomaly detection or visualized for better interpretation.

2.10 Visualizing Loss Distributions

This code snippet uses the `seaborn` library to visualize the reconstruction loss distributions for different subsets of data. Visualization is a critical step in understanding the model's performance and in differentiating normal and anomalous data.

Description of the Code:

* The `sns.kdeplot` function is employed to plot kernel density estimates (KDE) of the reconstruction losses. This technique visualizes the probability density function of the data, providing insights into the distribution of reconstruction losses.
* Specifically, two distributions are plotted:
  - `losses_normal`: The reconstruction losses for normal data are plotted in blue, representing how well the autoencoder reconstructs non-anomalous samples.
  - `losses_attack`: The reconstruction losses for anomalous data are plotted in red, highlighting the deviations from normal patterns.
* Commented-out lines include alternative visualization options, such as histograms (`plt.hist`) and combined KDE and histogram plots (`sns.distplot`), which are not used in this final version.

Purpose of the Code:

* The KDE plots provide a smooth, intuitive visualization of loss distributions, enabling the comparison of normal and anomalous data.
* Differences in the distributions help identify potential thresholds for anomaly detection. For instance:
  - If the red curve (anomalies) significantly deviates from the blue curve (normal data), this indicates that the autoencoder effectively distinguishes between normal and anomalous samples.
  - Overlapping regions may suggest areas where the model struggles to differentiate anomalies from normal data.
* These plots can guide the selection of an appropriate loss threshold to classify data as normal or anomalous.

2.11 Brute-Force Optimization of Cutoff Thresholds

This section of the code brute-forces the computation of optimal cutoff thresholds to maximize the accuracy of anomaly detection. The goal is to identify the lower and upper reconstruction loss bounds that best differentiate between normal and anomalous data.

Description of the Code:

* The code defines two ranges of values:
  - `lower_list`: A sequence of candidate lower bounds, generated by adding small increments around an initial `lower` value (23 in this case).
  - `upper_list`: A sequence of candidate upper bounds, similarly generated around an initial `upper` value (58 in this case).
* Two empty lists, `ls_1` and `ls_2`, are used to store accuracy scores and their corresponding threshold pairs.
* For each combination of `lower` and `upper` thresholds:
  - True positives (`TP`) are computed as anomalous samples with reconstruction losses within the threshold range (`lower` to `upper`).
  - False negatives (`FN`) are anomalous samples outside this range.

- False positives (`FP`) are normal samples within the threshold range.
- True negatives (`TN`) are normal samples outside this range.
- Accuracy is calculated as the ratio of correctly classified samples (`TP + TN`) to the total number of samples (`TP + TN + FP + FN`).

* The accuracy scores are appended to `ls_1`, and their corresponding threshold pairs (`lower, upper`) are stored in `ls_2`.

Purpose of the Code:

* This brute-force approach explores a wide range of potential threshold pairs to empirically determine the combination that yields the highest accuracy.

* By systematically varying `lower` and `upper` bounds, this method ensures thorough coverage of possible threshold combinations, helping to identify optimal cutoff values for classification.

Significance:

* The identified thresholds can significantly enhance the model's performance in distinguishing between normal and anomalous data, which is critical in anomaly detection systems.

* This approach, while computationally intensive, provides insights into the sensitivity of the model's accuracy to different threshold values.

2.12 Threshold Optimization and Performance Metrics

This section of the code finalizes the threshold optimization process by selecting the optimal thresholds and evaluating the performance of the anomaly detection system.

Description of the Code:

* The index of the maximum accuracy score is identified using the `np.argmax` function applied to the `ls_1` list, which stores accuracy scores for different threshold pairs.

* The optimal `lower` and `upper` thresholds corresponding to the highest accuracy are extracted from `ls_2`.

* Using these thresholds:
  - True Positives (`TP`) are computed as anomalous samples with reconstruction losses between the `lower` and `upper` thresholds.
  - False Negatives (`FN`) are anomalous samples outside this range.
  - False Positives (`FP`) are normal samples within this range.
  - True Negatives (`TN`) are normal samples outside this range.

* The True Positive Rate (`TPR`) is calculated as the ratio of `TP` to the sum of `TP` and `FN`.

* The True Negative Rate (`TNR`) is calculated as the ratio of `TN` to the sum of `TN` and `FP`.

* Accuracy is calculated as the ratio of correctly classified samples (`TP + TN`) to the total number of samples (`TP + TN + FP + FN`).

Purpose of the Code:

* This code evaluates the effectiveness of the optimized thresholds in separating normal and anomalous data based on reconstruction loss.

* By calculating key performance metrics such as TPR, TNR, and accuracy, it provides a comprehensive assessment of the model's ability to detect anomalies.

Significance:

* The metrics calculated in this code (TPR, TNR, accuracy) are essential for understanding the strengths and weaknesses of the anomaly detection system.

* These metrics highlight the trade-offs between detecting anomalies (sensitivity) and avoiding false alarms (specificity), which are critical in applications like cybersecurity.

Reference:

* The precision and recall concepts are further elaborated in the Wikipedia article on Precision and Recall.

3 Limitations

This section outlines the key limitations of the current project and its implementation:

- **Dependency on Dataset Quality:** The performance of the anomaly detection system is highly dependent on the quality and representativeness of the dataset. While the UNSW-NB15 dataset provides diverse attack scenarios, it may not fully capture real-world network traffic variations or evolving cyber threats.

- **Limited Generalizability:** The system has been primarily tested on the UNSW-NB15 dataset, which may limit its ability to generalize effectively to other datasets or real-world environments without significant retraining and fine-tuning.

- **Scalability Challenges:** The use of Autoencoders and neural networks requires substantial computational resources, particularly for large-scale networks with high traffic volumes. This may limit the scalability of the solution in real-time network environments without optimization or additional hardware.

- **High Computational Requirements:** Training the Autoencoder model is computationally intensive, requiring access to GPUs or high-performance CPUs for efficient processing. This makes the approach less accessible for organizations with limited computational resources.

- **Imbalanced Detection Capabilities:** The Autoencoder is trained on normal traffic data and relies on reconstruction errors to identify anomalies. This approach may lead to higher false positives for benign but unusual traffic patterns, as well as false negatives for cleverly disguised malicious activities.

- **Lack of Real-Time Evaluation:** The project has not yet implemented real-time monitoring and anomaly detection, which is a critical requirement for many practical applications in cybersecurity. The current implementation is better suited for batch processing and offline analysis.

- **Threshold Selection Sensitivity:** The anomaly detection system depends on predefined thresholds for reconstruction loss, which can be sensitive to small variations in data. Manually or heuristically determining these thresholds might not always result in optimal performance.

- **Limited Exploration of Alternative Models:** While Autoencoders were implemented, other potentially effective anomaly detection models, such as Isolation Forests or ensemble methods, were only conceptually explored and not implemented or evaluated in this project.

4 Conclusions and Future Work

This project has been a rewarding experience, allowing me to explore the application of machine learning techniques in anomaly detection for network security. However, there are a few reflections on what could have been done differently, areas for improvement, and plans for the future.

Looking back, starting the project earlier would have been beneficial. This would have allowed me to implement more of the code myself and gain a deeper understanding of the underlying processes and algorithms. While

I managed to grasp the core concepts and implementation details, having more time to experiment with and fine-tune the models could have enhanced my learning experience.

There are several ways to improve this project. One significant enhancement would be to integrate real-time anomaly detection capabilities, enabling the system to analyze network traffic as it happens. Additionally, exploring alternative machine learning models, such as Isolation Forests or ensemble techniques, could provide valuable insights and potentially improve the system's performance. Expanding the range of datasets used for evaluation would also enhance the system's generalizability and robustness in real-world scenarios.

I genuinely enjoyed this project. It allowed me to understand the importance of preprocessing data, designing and training models, and analyzing their effectiveness. Working with the UNSW-NB15 dataset and Autoencoders helped me develop practical skills in Python programming and the application of machine learning frameworks like PyTorch.

Through this project, I learned a great deal about anomaly detection, network traffic analysis, and the challenges of developing scalable machine learning solutions. It deepened my understanding of the complexities involved in cybersecurity and the potential of AI-driven methods to address them.

For future projects in this course, I plan to focus on Isolation Forests and evaluate their performance on diverse datasets. Comparing the results of these models with Autoencoders could provide a more comprehensive understanding of their strengths and weaknesses. Additionally, incorporating ensemble methods and advanced feature selection techniques could make the system even more robust.

Finally, the project was thoroughly enjoyable, and having more time to experiment and collaborate with peers could have made the experience even better. Overall, this project has not only been a valuable academic exercise but also an exciting opportunity to delve into the practical applications of machine learning in cybersecurity.

5 References

- **GitHub Repository:** Implementation of the Autoencoder for anomaly detection using the UNSW-NB15 dataset. Available at: https://github.com/alik604/cyber-security/blob/master/Intrusion-Detection/UNSW_NB15%20-%20PyTorch%20MLP%20and%20autoEncoder.ipynb.
- **UNSW-NB15 Dataset Description:** Comprehensive details about the UNSW-NB15 dataset for network intrusion detection research. Available at: https://www.unb.ca/cic/datasets/cic-unsw-nb15.html?utm_source=chatgpt.com.
- **UNSW-NB15 Dataset Files:** Training and testing datasets hosted by UNSW. Available at: https://unsw-my.sharepoint.com/personal/z5025758_ad_unsw_edu_au/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Fz5025758%5Fad%5Funsw%5Fedu%5Fau%2FDocuments%2FUNSW%2DNB15%20dataset%2FCSV%20Files%2FTraining%20and%20Testing%20Sets&ga=1.