Unsupervised anomaly detection in network traffic with deep learning

Ivaylo Krumov
University of Luxembourg
Email: ivaylo.krumov.001@student.uni.lu

This report has been produced under the supervision of:
Salima Lamsiyah
University of Luxembourg
Email: salima.lamsiyah@uni.lu

Abstract

The purpose of this Bachelor Semester Project is to ascertain whether deep learning models can be used to detect anomalies in network traffic, which are frequently signs of cybersecurity concerns. In order to accomplish this, the project focuses on the domains of cybersecurity, machine learning, and data analysis. The project explores a variety of concepts and principles associated with unsupervised anomaly detection, establishing the scientific and technical context around which the research is centered. Several deep learning models have been utilized and optimized to demonstrate their potential in distinguishing between normal and anomalous network traffic. The results obtained from the implementation of these models are comprehensively presented in this report. The report's and the project's overall objective is to offer insightful information about how deep learning models and related machine learning techniques can be used to improve cybersecurity practices, especially in identifying and mitigating the impact of network-based attacks.

1. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
  8) Using another person's unpublished work without attribution and permission ('stealing')
  9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

2. Introduction

Cybersecurity dangers have increased in today's interconnected world due to the quick sharing of information via the internet. Cybercriminals are using increasingly sophisticated techniques to take advantage of weaknesses in network systems as technology develops. Finding anomalies in network data that can point to possible breaches or malicious activity is one of the most important cybersecurity concerns. The necessity for more advanced methodologies using machine learning and deep learning techniques arose from the ineffectiveness of traditional anomaly detection methods against attackers' shifting tactics.

The present study and report investigate the utilization of unsupervised deep learning models in the identification of network traffic anomalies. Without the requirement for

pre-labeled samples, these sophisticated neural network architectures employ large quantities of data to learn patterns and recognize unusual behaviors. We look at the underlying theories of neural networks and different deep learning architectures, their potential for anomaly and pattern detection, and the value of feature extraction, model evaluation, and data pre-processing in improving cybersecurity application performance. We also discuss relevant case studies and examples from existing research, which demonstrate the effectiveness of various unsupervised learning techniques in recognizing and mitigating cyber threats, to better understand how unsupervised deep learning models might be used for this task.

To aid in this research, three distinct deep learning models that have been optimized for network anomaly detection are implemented. The models consist of an autoencoder, variational autoencoder and generative adversarial network. These implementations combine the discussed concepts and ideas to give concrete examples of the utility of using unsupervised deep learning for network anomaly detection in real-world scenarios. Various criteria are employed to assess the performance of each model following training and testing on the UNSW-NB15 dataset in order to ascertain its potential. Ultimately, these results are compared with each other to determine which of the models and corresponding methods that have been described may be applied to network traffic anomaly detection with the highest degree of reliability.

3. Project description

3.1. Domains

3.1.1. Scientific. The project encompasses the following scientific domains:

- **Cybersecurity and network intrusion detection.** The field of cybersecurity focuses on the protection of digital systems, networks and programs from cyberattacks. It aims to prevent unauthorized access to sensitive data and ensure the integrity and availability of data. The subfield of network intrusion detection is particularly stressed upon throughout this project, as it is its main topic. This area of cybersecurity focuses on detecting and responding to unauthorized access or malicious activities within a computer network through the use of specialized systems that monitor network traffic and analyze patterns and behaviors to identify suspicious activity as potential threat.
- **Machine learning.** Machine learning is a branch of artificial intelligence that revolves around algorithms which allow computers to learn from data and make decisions as a result of that learning process. In this project, deep learning models are trained for the goal of detecting anomalies in data using machine learning methods. The architecture and algorithms behind these models are designed to identify unusual data points, enabling the detection of suspicious and potentially malicious activities. The training process iteratively improves the models' accuracy and effectiveness over time with the tuning of different model parameters and maximizes the ability of the models to successfully detect anomalous data within a dataset.

3.1.2. Technical. The project encompasses the following technical domains:

- **Deep learning models** Deep learning models are a subset of machine learning algorithms that utilize multi-layered neural networks to automatically learn complex patterns and representations from large amounts of data. Oftentimes these models are initially trained on large datasets to learn general features, which can then be fine-tuned on specific tasks with smaller datasets. In this project, deep learning models are utilized to analyze network traffic data and find out any outliers that exist within it, which are considered as anomalies and potentially malicious activities within the network. To achieve this, these models must be fine-tuned accordingly to specifically adapt to the task of network anomaly detection and maximize their performance in identifying outliers in the data.
- **Data science.** In order to derive useful information from raw data, pre-processing, analysis, and visualization are all tied into the application of data science. These aspects are required for this project so that the selected large dataset of network traffic data is handled properly, ensuring that the data is suitable for training the models in order for them to learn from it effectively. This specifically includes data transformation, data encoding and plotting of results for easier interpretation. Being the programming language put into use for the technical part of the project, Python offers different library utilities to support the different tasks; for example, Pandas and Numpy are suitable for pre-processing and transforming the data, whereas Matplotlib and Seaborn are tailored for data visualization.
- **Python.** Python serves as the selected programming language for the development of the technical portion of the project. This choice was made due to its high popularity and versatility, as well as ease of use and high-level syntax that makes it fairly accessible and understandable by beginners and experts alike. Numerous fields and domains are supported by Python's extensive library selection, including machine learning with Scikit-learn and data pre-processing with Pandas, Numpy and Matplotlib. These tools are essential for handling large datasets, developing, training and testing machine learning models and visualizing the resulting outcomes.

3.2. Targeted Deliverables

3.2.1. Scientific deliverable. The produced scientific deliverable aims to find an answer to the question "How can deep learning models be utilized to detect anomalies in network traffic?". In order to achieve this, it draws on the information and conclusions of several scientific papers and articles that

are relevant to the project's topic. Using these sources as a knowledge base, it discusses topics such as theory and architecture of deep learning models and the current state of cybersecurity in regard to network intrusion detection based on existing work. The information acquired during this process is expected to provide enough insights to answer the scientific question. After covering all the scientific aspects relevant to the question, a conclusive answer is finally developed based on the gained understanding. The deliverable is then concluded by a brief discussion that aims to evaluate the extent to which all of the requirements for the deliverable have been satisfied and what improvements could have been made, if any.

3.2.2. Technical deliverable. The produced technical deliverable is made up of several Jupyter notebook files, each of which contains a distinct deep learning model that has been trained and fine-tuned for anomaly detection on the same dataset of network traffic requests. This report describes thoroughly each step of the programs' code in the corresponding section, including the data pre-processing portion and the training and testing aspects of the utilized deep learning model. The final results for each model as well as their visualizations are then also discussed, which to some extent serve to support the answer to the scientific question given in the previously discussed scientific deliverable. In the end, a short assessment section evaluates whether the achieved outcomes meet the defined expectations for the deliverable, as well as discussing if there could be made any modifications to the programs that would potentially lead to improved results.

4. Pre-requisites

The complete realization of this project, both the scientific and the technical part, takes advantage of several skills and competencies that already have been acquired before the start of the project's research process. For each deliverable, these pre-requisites are respectively described in the 2 subsections that follow.

4.1. Scientific pre-requisites

The author is knowledgeable about the appropriate format and structuring for a work of writing within the scientific field. Furthermore, he is completely capable of comprehending essential information from scientific publications as long as the language is employed at a suitable level of complexity. Last but not least, he is well aware of and possesses basic knowledge about the scientific domains related to the project, specifically cybersecurity and network intrusion detection, with a focus on the utilization of deep learning models for the task of detecting anomalous activity in network traffic.

4.2. Technical pre-requisites

The author has prior experience in software development in different programming languages. As a result, he is well aware of the possibility of running into technical challenges during the development of the technical deliverable and is prepared to take the required actions to overcome them. The author is also competent in Python programming, has a broad range of Python library knowledge and can use his technical understanding in tasks involving data outlier detection. This includes the ability to process and analyze datasets, have a grasp of basic data visualization concepts that are necessary for analyzing and interpreting the final outcomes of the process and have practical knowledge and experience with fundamental machine learning concepts.

5. Scientific Deliverable

5.1. Requirements

The deliverables of this project must meet several requirements, which are organized into two categories: functional and non-functional requirements. The functional requirements are outlined first, followed by the non-functional requirements.

5.1.1. Functional requirements:. The project's scientific deliverable should aim to answer the question: "How can deep learning models be utilized to detect anomalies in network traffic?" To achieve this, the deliverable must explain key concepts related to the project, such as data pre-processing, fine-tuning pre-trained deep learning models and the evaluation of model performance. Additionally, it should be grounded in relevant academic and scientific literature to provide a comprehensive overview of the relevant cybersecurity related field, namely network intrusion detection, and to contextualize the techniques and model types explored. To do so, it should discuss the fundamental principles of machine learning, the general architecture of deep learning models and specific characteristics of the anomaly detection models considered for this project's technical deliverable.

5.1.2. Non-functional requirements:. The content of the scientific deliverable should be presented in such a way that it is accessible to readers who do not necessarily have a deep background in machine learning, ensuring that all necessary technical terms are clearly defined. The structure of the deliverable must be coherent, with all sections logically connected to the main objective of evaluating the potential of deep learning models for anomaly detection in network traffic data. Proper citation of all references is essential to give credit to original authors and their work. The deliverable should be organized clearly and concisely to enhance readability and ensure that it effectively communicates the findings in an easily understandable manner.

5.2. Design

The scientific deliverable serves as a crucial piece of text that intends to complement the technical deliverable by providing the necessary context and background for its ideas,

methods and implementation. Although success of the project is largely dictated by the outcomes of the technical deliverable, the scientific deliverable is essential for establishing the foundational understanding required to fully grasp the technical work, therefore it can be considered as a vital prerequisite of sorts. In order to ensure a thorough exploration of the topics relevant to answering the scientific question, the research process for the scientific deliverable was conducted over approximately 2 weeks. Given the limited time granted to complete this particular project, this is considered as a satisfactory timespan.

The structure of the scientific deliverable is straightforward, with sections dedicated to defining specific concepts or subtopics related to the project's main question. After presenting the necessary definitions and explanations, the text synthesizes this information to formulate a well-supported answer to the scientific question. This concluding section integrates the key points from earlier sections, creating a cohesive and logical answer that connects the theoretical and practical components of the project.

Throughout the scientific deliverable, multiple sources are cited to assure that the presented information and findings are accurate and reliable. These sources provide formal definitions and theoretical insights and work to improve the clarity of the explanations. Where appropriate, concrete examples are included to illustrate concepts in greater detail, aiding the reader's understanding.

5.3. Production

In order to explore the scientific question, "How can deep learning models be utilized to detect anomalies in network traffic?" it is crucial to first understand the foundational concepts of both deep learning and unsupervised learning. These principles provide the basis for applying specific models such as autoencoders, variational autoencoders (VAEs) and generative adversarial networks (GANs) to the task of identifying anomalies in data. This report will begin by discussing the principles of deep learning and unsupervised learning, followed by a detailed exploration of the key architectures and techniques used for anomaly detection.

To begin with, we will acquaint ourselves with the concept of deep learning. Deep learning is a subset of machine learning that uses neural networks with many layers, often called deep neural networks, to model complex patterns in data. Unlike traditional machine learning models, which often require extensive feature engineering, deep learning models can automatically learn representations of the data, making them particularly powerful for tasks involving large datasets. Deep learning models are designed to mimic the human brain's ability to recognize patterns, and they have been successfully applied in areas such as image and speech recognition, natural language processing, and now, anomaly detection. In essence, anomaly detection defines a vital aspect of data analysis, focused on identifying data instances that significantly diverge from typical behavior. These unusual data points, known as anomalies or outliers, are often indicators of critical issues such as system malfunctions, fraud, or security breaches for network security in particular. Unlike conventional data analysis, which assumes uniformity across data points, anomaly detection specifically targets these rare and unusual occurrences. The challenge lies in the fact that anomalies are often rare and varied, making them difficult to detect with standard approaches. This is where the strengths of deep learning models are especially advantageous. These models are composed of multiple layers of interconnected neurons. These neurons apply transformations to the input data and pass the processed information to the next layer. Through this hierarchical structure, deep learning models can learn increasingly abstract representations of the data, which are essential for a wide range of tasks that require understanding intricate patterns.

Unsupervised learning is a type of machine learning where the model is trained on data without explicit labels. Unlike supervised learning, where models are trained on labeled datasets with known outputs, unsupervised learning focuses on identifying structural patterns within the data itself. This approach is particularly useful for anomaly detection, where labeled examples of anomalies are often rare or non-existent. Unsupervised learning techniques are utilized to discover hidden structures within the data, such as clusters or outliers. Among these techniques are clustering, dimensionality reduction, and density estimation, which help in understanding the underlying distribution of the data. When combined, deep learning and unsupervised learning form a powerful combination that, in theory, promises to be quite effective for anomaly detection. The deep neural networks used in these models are patterns with various degrees of complexity from large volumes of of data, while the unsupervised learning approach allows them to identify deviations from these patterns, i.e. anomalies, without requiring explicitly labeled samples.

Delving deeper into the project's topic, as already mentioned, unsupervised anomaly detection involves identifying data points that deviate significantly from the norm, without the need for labeled examples of anomalies. In the context of deep learning, several key techniques have been developed for this purpose, each utilizing the ability of neural networks to learn representations of the data. One prominent approach is reconstruction-based methods, which rely on the model's ability to reconstruct input data from a compressed representation. The underlying principle is that a well-trained model, such as the later discussed autoencoder, can accurately reconstruct data points that follow the learned patterns of the training set. However, if a data point significantly deviates from the norm, the model struggles to reconstruct it accurately, indicating that it might be an anomaly. This approach is effective because anomalies often exhibit patterns that differ from the majority of the data, making them difficult for the model to reproduce accurately. Another important technique is adversarial methods. In this approach, a model is trained to generate data that closely matches the distribution of the training set. The model consists of two networks: a generator,

which creates synthetic data, and a discriminator, which tries to distinguish between real and generated data. During training, the generator improves its ability to produce data that the discriminator cannot easily differentiate from the real data. Once trained, any data point that the discriminator identifies as unlikely to be real, meaning it does not fit the learned distribution, is flagged as an anomaly. Lastly, feature learning and dimensionality reduction are techniques that focus on reducing the dimensionality of the data while preserving the most relevant features for anomaly detection. Deep learning models, inherently perform dimensionality reduction as part of their architecture by compressing the input data into a lower-dimensional latent space. This reduced representation highlights the most important aspects of the data, making it easier to identify outliers.

Now that we have a foundational understanding of deep learning and unsupervised learning, we can explore the specific architectures that are commonly used for unsupervised anomaly detection and were hinted at in the previous paragraph. Starting off with the most basic models, autoencoders are neural networks designed to learn efficient representations of data, typically for the purpose of dimensionality reduction or feature learning. As described by [Ieracitano, C., Adeel, A., Morabito, F. C., & Hussain, A. (2020).], an autoencoder utilizes a reconstruction based method and consists of two main parts: an encoder, which compresses the input data into a latent space representation, and a decoder, which reconstructs the input from this compressed form. The key idea in using autoencoders for anomaly detection is that the model is trained on normal data, so it learns to reconstruct this type of data well. When the model encounters a data point that differs significantly from the normal data, the reconstruction error is high, indicating the presence of an anomaly. Autoencoders are particularly effective for scenarios where anomalies are expected to vary significantly in features and structure from normal data. A variation of the autoencoder is the promptly named variatonal autoencoder (VAE). VAEs extend the concept of autoencoders by introducing a probabilistic approach to the latent space. Instead of mapping the input to a single point in the latent space, VAEs learn to map the input to a distribution over the latent space, typically a Gaussian distribution. This probabilistic framework allows VAEs to better capture the uncertainty and variability in the data, making them more robust for the task at the cost of being more computationally expensive. In the context of anomaly detection, similarly to standard autoencoders, VAEs can identify anomalies by evaluating how likely a data point is under the learned distribution. If a data point is unlikely under this distribution, it is flagged as an

Key techniques mentioned in the different aforementioned models and used unsupervised anomaly detection in general include reconstruction, probability estimation, and adversarial training. Reconstruction error is commonly used in models like autoencoders and variational autoencoders, which learn to reconstruct the input data. The difference between the original input and its reconstruction is the reconstruction error and it serves as a measure of how well the model has learned the data distribution. When the model encounters an input that it does not recognize the reconstruction error tends to be high, suggesting that the input may be an anomaly. Probability estimation lies more in the context of VAEs. VAEs are designed to model the probability distribution of the training data, and during inference, they can calculate the likelihood that a new data point belongs to this distribution. Data points with low probability scores are considered anomalies because they do not conform to the patterns the model has learned from the training data, providing a direct way to assess the "normality" of a data point. Adversarial training is the particular method used in models like GANs. As mentioned earlier, this method involves a generator network that creates synthetic data, which then a discriminator network tries to distinguish from real data. Through this adversarial process, the generator learns to produce data that closely matches the real data distribution. When anomalies are introduced, the generator fails to replicate these outliers accurately, and the discriminator easily identifies them as anomalies. The drawback of this technique is that, in comparison with other methods, the whole process is more computationally intensive, requiring more resources as well as time, effectively leading to more accurate predictions at the cost of a slower than average training phase. In addition to these approaches, there exist various other machine learning principles that are integrated with one another. In regard to the task at hand, the most important of these are the regularization techniques that ensure effective model training, such as dropout and and weight decay. Dropout randomly disables neurons during training. This forces the network to rely on multiple neurons rather than any single one, which helps the model learn more generalizable features. This prevents the network from learning ineffectively from overfitting to the specific details of the training data and producing an extensive number of false positives and false negatives on new and unseen data later down the line. Meanwhile, weight decay is a regularization method that adds a penalty to large weights within the network. This further serves to reduce the likelihood of the model overfitting to the training data by encouraging it to maintain smaller, more general weights and adjust the model's parameters in a way that prioritizes generalization over memorization.

Building on the foundational principles of unsupervised anomaly detection with deep learning, we can examine a few notable use cases that showcase the effectiveness of these models across various domains. For instance, in the realm of industrial monitoring, the study by [Truong-Huu, T., Dheenadhayalan, N., Pratim Kundu, P., Ramnath, V., Liao, J., Teo, S. G., & Praveen Kadiyala, S. (2020, October).] additionally utilized deep autoencoding Gaussian mixture models (DAGMM) to detect anomalies in multivariate time series data from manufacturing processes. This model, combining dimensionality reduction with density estimation, was particularly effective at identifying subtle deviations in sensor data that could indicate equipment failures. The study demonstrated that DAGMM could outperform traditional methods, achieving

higher accuracy in detecting faults while maintaining a low false positive rate. Similarly, a study by [Zong, B., Song, Q., Min, M. R., Cheng, W., Lumezanu, C., Cho, D., & Chen, H. (2018, February).] employed a deep autoencoding Gaussian mixture autoencoder to uncover fraudulent activities in financial transactions, demonstrating the model's ability to handle high-dimensional data and detect anomalies that other statistical methods might miss. Another significant application is in network security, where [Vinayakumar, R., Alazab, M., Soman, K. P., Poornachandran, P., Al-Nemrat, A., & Venkatraman, S. (2019).] applied a variational autoencoder for anomaly detection in network traffic. By learning the distribution of normal traffic patterns, the VAE was able to identify unusual activity indicative of cyber threats, such as DDoS attacks, with great accuracy and efficiency. Similarly, research by [Sivasubramanian, A., Devisetty, M., & Bhavukam, P. (2024).] using the NSL-KDD dataset highlighted the importance of feature extraction in deep learning models for network intrusion detection. This study demonstrated that stacked autoencoders (SAEs) are particularly effective in scenarios where data points are highly correlated, and that convolutional approaches improve local feature understanding. Despite the challenges posed by class imbalance in training data, the models achieved strong performance, with a ROC-AUC score of 0.92 on test data and 0.81 on NSL-KDD Test+ data. Extending this work, [Fotiadou, K., Velivassaki, T. H., Voulkidis, A., Skias, D., Tsekeridou, S., & Zahariadis, T. (2021).] proposed using Long Short-Term Memory (LSTM) networks and one-dimensional convolutional neural networks (1D-CNN) for detecting anomalies in pfSense firewall logs. Their approach achieved high classification accuracy (97.29% for LSTM and 96.34% for 1D-CNN) in categorizing network events into six different types, outperforming standard machine learning approaches.

The in-depth theoretical exploration and case studies presented in this deliverable demonstrate that unsupervised deep learning models can effectively be utilized for anomaly detection across various domains. The examples provided, in combination with the discussed theoretical principles, offer a concrete answer to the research question —unsupervised models, such as autoencoders, variational autoencoders, and GANs, can successfully identify and analyze anomalies by making use of the discussed core principles of deep neural networks and unsupervised learning. The performance of these models is heavily reliant on the quality of data pre-processing, feature extraction, and the architecture design of the models themselves. As illustrated in the studies, the integration of diverse deep learning techniques and continuous refinement of models through hyperparameter tuning and architecture adjustments can lead to significantly more accurate and robust anomaly detection systems.

5.4. Assessment

The scientific deliverable thoroughly explains fundamental concepts like unsupervised deep learning, data preparation, and model assessment, creating a strong basis for comprehending anomaly detection systems. It applies this theoretical knowledge to real-world scenarios, showcasing how models such as autoencoders, variational autoencoders, and GANs are used for anomaly detection across different fields. This comprehensive analysis directly tackles the project's main scientific question and provides a clear, definitive answer. The deliverable is well-structured, with content that is clear, coherent, and logically aligned with its primary goal. Each part builds on the previous one, offering an in-depth exploration of the subject. Scientific paper references are effectively integrated throughout, strengthening the credibility of the information and providing deeper insights. Case studies and examples effectively demonstrate the practical applications of theoretical concepts, linking the scientific and technical aspects of the project. In summary, the scientific deliverable meets all requirements that had been laid out. It equips readers with a comprehensive understanding of key concepts and their practical applications, supported by a sophisticated scientific framework, preparing them for the technical deliverable that follows.

6. Technical Deliverable

6.1. Requirements

The technical deliverable of this project aims to satisfy several requirements, which are presented in this section. To provide a better structure, they are split into two different categories: functional and non-functional requirements. The functional requirements are described first, followed by a description of the non-functional ones.

6.1.1. Functional requirements. The technical deliverable should include the implementation of at least two deep learning models particularly suited for anomaly detection. These models must be trained and tuned specifically to network anomaly detection using a publicly available dataset. The dataset should contain recorded network traffic data with numerous features describing in detail the various requests it consists of. Each model should be relatively distinct from the others, employing unique techniques that characterize it. The models' effectiveness must be evaluated using relevant performance metrics in order to assess their ability to correctly identify outliers in the data. Lastly, the dataset should undergo any necessary pre-processing thoroughly using libraries like Pandas and Numpy to ensure that it is appropriately structured for model training. Visualization of the results, using tools such as Matplotlib, is also required to effectively present the outcomes and insights gained from the testing of the trained models.

6.1.2. Non-functional requirements. The implementation of the models should be implemented in a way that maximizes accuracy and precision of the results while guaranteeing that the training and evaluation procedures are finished in a reasonable amount of time. In order to process large amounts of training

and testing data without noticeably degrading performance, the implementation should also be either scalable or have imposed limitations on the volume of input data they can process. To improve readability, reproducibility, and user-friendliness, the code needs to be relatively well documented and arranged a proper format, such as Jupyter notebooks. Finally, in order to maximize anomaly detection performance, the implementation should be adaptable, enabling simple testing with the various hyperparameters and their values.

6.2. Design

The technical deliverable is divided into three Jupyter notebooks, each of which does on its own the handling the pre-processing, training, and testing processes of the various deep learning models for anomaly detection. In particular, the models chosen to perform this task are an autoencoder, a variational autoencoder and a generative adversarial network, all utilized directly from from PyOD, the Python outlier detection module. These decisions are primarily motivated by the desire to examine how many progressively more complicated models — which employ increasingly elaborate architectures and intricate deep learning techniques — perform on the particular anomaly detection task. The deliverable was developed over the span of roughly 3 weeks. A significant portion of this time was dedicated to creating and refining the code in each notebook, including data pre-processing and iterative improvements on model performance based on empirical testing with different hyperparameter values for each model.

The content of the notebooks generally follows the same structure for every one of them. Firstly, the same data is used and pre-processed to accommodate a proper format so that it is usable as input for the subsequent training and testing stages. After that, the data is divided into training and testing sets and the according model is initialized. This is followed by a training phase adjusted to the respective model in terms of hyperparameters and values used. After training the model for the anomaly detection task, its performance is evaluated, with a particular focus on metrics like accuracy, F1 score and most importantly receiver operating characteristic (ROC) curve, with the latter being well suited for anomaly detection due to its effectiveness in assessing the model's performance across different thresholds. Visualization tools are also used to display the results by creating suitable plots that show the performance metrics and provide a clear picture of the model's performance. Each notebook operates independently, from data pre-processing continuing through model training, evaluation, and result visualization. This lack of dependencies creates opportunities for better flexibility in experimenting with various parameters, values and techniques that do not require modifying third-party files to achieve this.

6.3. Production

Every Jupyter notebook file that comprises the final deliverable makes use of the exact same UNSW-NB15 data for training and evaluating the deep learning models, which already comes conveniently split into 2 parts - a training set and a testing set. Both datasets, which have been collected and openly provided by a research team from UNSW Canberra, are comprehensive collections of recorded network traffic particularly designed for academic use on research related to network intrusion detection. Each entry in the data is a separate network request which consists of 42 different features describing its structure and 2 labels - one about the type categorization of the request (e.g. whether it is considered "Normal", "Malware", etc.) and another giving a binary value (either 0 or 1) to the request depending on if it is considered an intrusion or not (0 corresponding to an intrusion and 1 to a "Normal" request). Throughout all files, the data is pre-processed in exactly the same way, since all 3 deep learning models require the same kind of basic data pre-processing and formatting before use. To avoid repetition, first the pre-processing portion will be explained in its entirety once, followed by separate descriptions of the training, testing and result visualizations for each model.

The pre-processing begins by loading the training and testing datasets from their respective CSV files. The training dataset contains 175,341 entries, while the testing dataset comprises 82,332 entries. These datasets are read into Pandas DataFrames, allowing for efficient manipulation and analysis of the data. Initial inspection of the datasets is performed using the *head()* method, which displays the first 5 rows (Fig. 1). This step helps to confirm that the data has been loaded correctly and provides a preliminary understanding of the structure and contents of the datasets. The contamination rate represents the proportion of anomalous data points within the dataset. For anomaly detection, understanding the proportion of anomalies is crucial for setting appropriate thresholds and model expectations. Since we will need to know what distribution of the data is outliers, i.e. intrusion requests, for the training and evaluation phases of the models, the contamination rate is computed straight away (Fig. 2). The contamination rate represents the proportion of anomalous data points within the dataset and is crucial to determine appropriate testing thresholds. It is computed for both the training and testing datasets by examining the *attack_cat* column, which is the same column that categorizes each network request into types such as "Normal," "Malware," etc. as mentioned earlier. This is a straightforward calculation that simply filters out the "Normal" entries and calculating the proportion of non-"Normal" entries. The train contamination rate is computed as roughly 68%, whereas the test one is roughly 55%. This shows that in both sets, the outlier entries are actually more than the inliers. Because of the restriction that PyOD imposes on the contamination values for training its models, namely contamination must be no more than 50%, this value cannot be used directly into the model. This is

done because an imbalance where anomalies are excessively, as is the case, prevalent can negatively affect model training and evaluation. The approach that is used to fix this issue involves considering the outliers as inliers and vice versa. Thus, the train contamination is adjusted by inverting it and the new contamination, being the one representing the proportion of "Normal" entries, is taken into account from this point forward. The same is done for the test contamination, as it is used to compute evaluation thresholds later on.

After calculating and adjusting the contamination rates, the training and testing datasets are merged into a single DataFrame for uniform pre-processing. This method guarantees that pre-processing processes are consistently applied to both datasets and makes data handling simpler. The *id* column, which serves as a unique identifier for each entry, and the *attack_cat* column are dropped. These columns are not needed for model training as they do not provide anything of value such as predictive features or target labels. An argument could be made to also drop any duplicate rows and rows with missing values to further avoid using redundant information, especially since the expectations are that due to the nature of network traffic, where the same request can be sent out multiple times, the number of duplicates would be large. However, this suggests performing some inconvenient pre-processing steps that also significantly tamper with the *label* column at the end when splitting the data back into training and testing sets. Therefore, for the sake of integrity and simplicity, duplicate values are left in. Inspecting the new combined dataset shows that there are in fact no missing values to worry about but that it includes categorical features such as *proto*, *service*, and *state* (Fig. 3). Machine learning algorithms normally require numerical input, so these categorical features need to be encoded into numerical values. This conversion is done through the use of Sklearn's LabelEncoder module. This process transforms the categorical labels into corresponding integers, making the data suitable for model training (Fig. 4).

With the main pre-processing steps completed, the combined dataset is split into features and target labels, where the binary *label* column is separated from the feature columns. Additionally, the target labels are inverted (i.e., 1 becomes 0 and 0 becomes 1) to accommodate for the consideration of treating "Normal"-labeled entries as outliers, aligning with the earlier adjustments to the contamination rates (Fig. 5). Finally, the dataset is divided back into training and testing sets through the use of the *iloc* method which splits the features and labels based on the original lengths of the training and testing datasets (Fig. 6). The two resulting datasets are fully pre-processed versions of the original two UNSW-NB15 training and testing sets, ready to be passed as input to the deep learning models for training and evaluation.

The next step is to initialize and train the models. For the autoencoder, this is done through PyOD's auto_encoder module. This module contains the AutoEncoder class itself, which creates a new autoencoder model instance. The initialization of the model is dependent on several hyperparameters, the more important of which will now be described (Fig. 7). It is noteworthy that all of the parameter values have been selected after extensive empirical testing with different combinations to maximize the performance reflected by the final metrics. First and foremost, a proper neural network architecture must be defined to serve as the underlying backbone of the model. PyOD's implementation directly alows to declare the structure of the hidden layers with the *hidden_neuron_list* parameter, which specifies the number of neurons in each hidden layer for the encoder, which is then mirrored for the hidden layers of the decoder. In this case, the autoencoder has two hidden layers in both its encoder and decoder. In the encoder, the first hidden layer, with 32 neurons, reduces the dimensionality of the input data, allowing the model to learn a compressed representation. The choice of 32 neurons is also motivated by the fact that the input data has 42 features and the general rule of thumb for such neural networks that the first hidden layer should start with a number of neurons no larger than the number of data features and each layer afterwards should gradually decrease its neuron count. As such, the second hidden layer, the bottleneck layer, further compresses the data with only 8 neurons. The decoder then takes the final representation and reconstructs it back into the original input's dimensions by passing it through the two mirrored hidden layers. Another essential parameter is *dropout_rate*, which is set to 0.4. This means that 40% of the neurons in the layers where dropout is applied will be randomly deactivated during each training step, preventing overfitting by forcing the model not to heavily rely on single neurons. The boolean *preprocessing* is also given a value of True, which significantly helps to improve model learning performance by applying additional pre-processing steps to the input data that optimize it further, notably standardization techniques, without requiring additional work than what has already been done.

Other major hyperparameters include the learning rate (*lr*), the optimizer (*optimizer_name*) and its additional parameter values (*optimizer_params*), as well as the train contamination rate (*contamination*), which simply uses the previously computed contamination value for the training data to set the model's expectations for the data distribution. The learning rate is set to a very small value of 0.00001. Generally, a smaller learning rate ensures that the model makes fine adjustments to its weights, which helps it converge more smoothly to an optimal solution without overshooting. The *optimizer_name* parameter is a string the defines optimizer that adjusts the model's weights during training for a smoother process. In the case of this autoencoder, the Adam optimizer has been chosen, since it is generally well known to be efficient and effective in training neural networks. This choice is reinforced by the *optimizer_params* parameter, which is a dictionary containing only a single element but can include more if desired. The element in question is the additional optimization parameter *weight_decay* with a value of 0.01. This gradually decreases the training weights and helps to prevent the model from becoming too complex and overfitting the data.

The last few hyperparameters consist of the number of training epochs, the batch size of the input data and the hardware

device to use during training. The *epoch_num* parameter is set to 10, meaning the model will iterate over the entire training dataset ten times during the training process. This relatively low number of epochs is chosen not only to limit the training time to a practical amount but also to prevent overfitting, which can occur when a model is trained for too long on the same data. The *batch_size* being set to 32 means that 32 samples from the training dataset will be processed at each step during training. This particular number is a common choice in for deep learning neural networks, as it finds a balance between memory efficiency and training speed, allowing the model to make steady progress towards optimal training results while being efficient in terms of memory usage. Finally, the *device* parameter has a default value, which indicates that the model makes use of the CPU's resources during training unless a compatible NVIDIA GPU is available, in which case the GPU's computational power is utilized. In practice, the device used throughout the process was always the GPU, as the code was run on a computer with NVIDIA GeForce GTX 1650 Ti, which is a CUDA-compatible GPU. This meant that it could make use of NVIDIA's CUDA platform to train the model using the GPU's processing power and achieve faster performance through more optimized computations. Nevertheless, the CPU option remains for the cases where the code might be executed on a machine without a CUDA-supported GPU or if a failure related to the GPU occurs. Two additional minor parameters are added in the end, which have no real impact on the actual training process - *verbose* simply assists to display the training progress, including epoch number, corresponding loss value and time taken to complete per epoch, and the *random_state* sets a fixed random seed to ensure reproducibility of the results each time the training loop is executed. The visualization provided by the given verbosity also plays a crucial role in determining when convergence occurs by tracking the gradual decrease of the loss function, as well as looking out for sudden increases, which is a sign of overfitting.

Once the autoencoder is configured with these parameters, the model is trained on the train data using the *fit()* method. This method initiates the training process, where the autoencoder learns to compress and reconstruct the input data while identifying anomalies. Note that since all the models integrate unsupervised learning, their *fit()* method does not take the training labels as input and only makes use of the training features to learn to distinguish outliers on its own. The training is done over the specified number of epochs, during which the model adjusts its weights based on the reconstruction error to minimize the difference between the input and output. The output given by the function displays the progress according to the aforementioned verbosity (Fig. 8). From it, it can be seen that the loss function value starts off at 1.1444 and steadily decreases to 0.7733 over the 10 epochs, with the decrease itself becoming lower after each epoch. The negligible decrease in the later epochs is an indication of the model reaching its convergence point, meaning an optimally minimized loss value. During the hyperparameter tuning procedure it has become evident that past the tenth epoch sudden increases and fluctuations in this value begin to occur, i.e. the model overfits the data, confirming that the number of training epochs has indeed been suitably chosen in regard to the rest of the selected parameter values.

Once training is complete, the evaluation phase is initiated. The first step in the evaluation process involves using the trained autoencoder to predict labels for the test data. This is done *predict()* method, which represents the model's initial predictions on whether each data point in the testing set is normal or anomalous. However, this initial prediction is not the final step. Instead, the focus shifts to computing anomaly scores using the decision function. In anomaly detection, this is a special function that through which the autoencoder computes scores based on the reconstruction error for each data point — the difference between the original input and the reconstructed output. Higher scores indicate a higher likelihood that the data point is an anomaly and this provides a continuous scale of anomaly scores rather than binary predictions for a less biased decision-making. After calculating the anomaly scores, the next step would be to compute the metrics that will evaluate the model's performance. PyOD provides a direct method for this, *evaluate_print()*, which only requires the predicted labels and the anomaly scores as inputs and relies on the receiver operating characteristic (ROC) curve as its main metric for output. This is in fact a suitable performance metric when it comes to anomaly detection, due to its effectiveness in assessing the model's performance across different thresholds, and is the one used for the final performance evaluation. Despite this, in order to gain a more detailed and complete understanding, we have decided to also include the values of standard metrics such as classification matrix, accuracy and F1-score with the help of Sklearn's metrics module. To achieve this, the labels will need to explicitly be adjusted to account for the value threshold. Thus, the next step becomes determining the threshold that will be used to classify data points as either normal or anomalous. It is calculated based on the expected contamination rate in the test data. Numpy's *percentile()* function is used to find the score value below which a certain percentage of the data falls. The particular formula selected for this computation is $100 * (1 - test\_contamination)$. For example, since the test contamination rate is 45%, the threshold is set at the 55th percentile of the anomaly scores. Any data point with a score above this threshold is classified as an anomaly. With the threshold determined, the next step involves classifying each data point based on its anomaly score. To achieve this, the previously predicted labels are reassigned by comparing each score to the threshold. Data points with scores higher than the threshold are labeled as anomalies (True), while those below the threshold are labeled as normal (False). This label reassignment through binary classification enables the evaluation of the model's performance using the standard Sklearn metrics alongside PyOD's usage of the ROC metric.

To avoid any confusion in the following metrics analysis, it is good to place a reminder here that as a result

of the label swapping during pre-processing, the "normal" class, i.e. the inliers, of the data refers to be the anomalous data points, whereas the anomaly class, i.e. the outliers, in reality refers to the non-anomalous entries, and this will be retained for consistency. The confusion matrix shows that out of 45,332 normal instances, 31,777 were correctly identified (true positives), while 13,555 were misclassified as anomalies (false negatives). Similarly, out of 37,000 outliers, the model correctly identified 23,445 (true negatives) but mistakenly labeled 13,555 as inliers (false positives). Sklearn's classification report provides a deeper look into the model's metrics (Fig. 9). For the normal class (0), the model achieved a precision and recall of 0.70, resulting in an F1-score of 0.70. This means that 70% of the instances predicted as normal were correct, and 70% of all actual normal instances were correctly identified. For the anomaly class (1), the precision, recall, and F1-score are slightly lower at 0.63, indicating a slightly reduced ability to accurately identify anomalies compared to normal instances. Overall, the model's accuracy stands at 0.67, meaning it correctly classified 67% of the total data points. The macro and weighted averages, both at 0.67, reflect consistent performance across both classes. The ROC AUC score provided by PyOD as the main metric shows a value of 0.7302, suggesting that the model has a reasonably good ability to distinguish between normal and anomalous instances, predicting correctly 73% of the time, though there is clear room for improvement. The secondary metric, the precision at rank n is calculated at 0.6336, which indicates that among the top n% of instances identified as most anomalous, around 63% are correctly classified as such. These results are supported by 2 visualizations. Matplotlib is used to plot the ROC curve, which is shown to be consistently above the diagonal that represents the performance of an average model (Fig. 10). This is indicative of the autoencoder's moderate predictive power, performing better than random but with potential for refinement. PyOD is further utilized to create a figure of expected predictions versus actual predictions through the use of its *visualize()* function, plotting the inliers and outliers as points in a diamond shape (Fig. 11). Admittedly the sheer number of data points on this figure make it slightly difficult to read, but the key observations from it are that the model correctly identifies the majority of points as outliers in both cases, while there exists a slight discrepancy in detecting inliers along the edges, albeit a minimal one. The figure also plays an additional role of visualizing the ROC curve score, evident in how well the prediction plots match the ground truth, especially in maintaining the overall diamond shape and class distribution.

Moving on to the VAE model, its implementation largely follows the same steps, with the data pre-processing and evaluation phases being exactly the same. The only notable difference is in the tuning of the model's hyperparameters, where specific adjustments are made to accommodate its distinct architecture. In particular, most of the parameters defining the autoencoder model are applied here as well, though with slightly changed values for some of them as it would be expected, but there are also 3 additional ones - *encoder_neuron_list*, *decoder_neuron_list* (both of which replace the autoencoder's *hidden_neuron_list*) and *latent_dim* (Fig. 12). These set the VAE apart from the standard autoencoder by introducing a probabilistic approach to encoding the input data. They are absent in the autoencoder model because generally autoencoders rely on a simpler, deterministic mapping of the input data to a fixed-size latent vector, without the need for modeling a probabilistic latent space, therefore they would not need a very detailed definition to their architecture compared to VAEs. The two lists, one for the encoder neurons and another for the decoder neurons, specify the architecture of the encoder and decoder networks, respectively. They define the number of neurons in each layer separately, allowing for more precise control over the model's complexity. Due to the increased complexity and the time and resource constraints of this project, the default values for each have been selected ([128, 64, 32] and [32, 64, 128] respectively), which also appear to be some of the better performers when it comes to all considered value options. Meanwhile, the dimensionality of the probabilistic latent space is defined by the *latent_dim* parameter. Testing with the various value possibilities shows that the model performs very similarly for all of them, especially 2, 6 and 8. To encourage the model to learn a more compact and interpretable representation, the choice of the lowest value, namely 2 dimensions, has been favored, as it simplifies the latent space as much as possible while still capturing the essential features of the data. From the *fit()* method's output, the fitting process is indicated to be slower than that of the autoencoder, both in time per epoch and decrease value of the loss function, the latter of which begins at 0.9090 and only goes down until 0.7274 with a negligible reduction in between the last 4 epochs (Fig. 13).

The evaluation metrics show slightly moderate performance, which actually falls below the performance of the previous autoencoder (Fig. 14). While the model correctly identified 29,269 normal instances and 20,919 anomalies it still made 16,063 false positives and 16,081 false negatives. Precision and recall are both 0.65 for normal instances, but drop to 0.57 for anomalies, indicating weaker performance in detecting anomalies ("Normal" requests). The F1-scores follow this pattern, with the model performing better on normal data. The overall accuracy is 61%, and the ROC score of 0.6495 suggests the model has limited ability to distinguish between inliers and outliers. The precision at rank n of 0.5657 also highlights its struggle to accurately identify top anomalies. The same couple of visualizations are plotted and further support that the VAE is having a difficult time in effectively performing anomaly detection. The predictions figure shows close results to the autoencoder, but with a reduced number of correctly predicted outliers (Fig. 15). Similarly, the ROC curve, while still being consistently above the diagonal, has a less pronounced "bow" shape (the ideal curving shape) and more deviations towards the diagonal at several points (Fig. 16).

The final model implementation, that of the generative adversarial network (GAN) provided by PyOD's *anoGAN*

module, also fully reuses the data pre-processing stage, and thus the same data in identical format as input. When it comes to tuning the model's hyperparameters, it is essential to note that the GAN has a vastly different architecture and objectives from the two autoencoders, meaning that the parameters themselves and their values also differ (Fig. 17). The GAN model's architecture consists of two neural networks: a generator and a discriminator. The generator creates synthetic data samples, while the discriminator evaluates whether a given sample is real or generated. This adversarial process helps the GAN learn to generate data that closely mimics the normal patterns in the training set. To keep it concise, key parameters include the lists of number of nodes in the hidden layers of generator and the discriminator, which are left at their respective default values ([20, 10, 3, 10, 20] and [20, 10, 5] respectively), as well as a ReLU activation function in the hidden layers for effective learning within them. With 1000 epochs and a low learning rate of 0.00001, the model is expected to accommodate for gradual stable learning. The batch size of 32 ensures efficient updates, while a dropout rate of 0.1 helps prevent overfitting. Additionally, the *learning_rate_query* is a specific to anoGAN's implementation that needs to match the value of the aforementioned learning rate for most efficient results. During the fitting process, this variation of GAN searches for a point in the latent space that, when passed through the generator, produces an output close to an arbitrary unseen data point, i.e. the query. This process involves optimizing the latent space representation of the query, which is what this parameter controls. It represents the learning rate for the backpropagation steps needed to find a point in the latent space of the generator that approximate the query sample, i.e. how quickly the model reaches the optimal query representation. After model initialization, the *fit()* method once again provides training information over the epochs and additionally for each query sample processed during the querying phase (Fig. 18). Specifically, it logs the progress of the model's training through increments of 100 iterations, indicating the number of epochs completed. Following the training phase, the model proceeds to evaluate query samples from the training data, providing progress updates on the processing of each sample. An issue encountered at this point is that the model would not be able to query the entire dataset in a practical time due to the sheer volume of it (175,341 data points). To achieve a more manageable processing time, only the first 5000 entries of the training data are considered. The reduction in samples comes at the cost of potentially less efficient model performance, but the increased speed of the training makes the tradeoff worth it in the end.

Since computing the predictions and the decision anomaly scores once more requires going through a querying phase for each of the two steps, the time is again optimized by reducing the number of samples by taking only the first 1000 samples of the test data, while retaining the 1:5 ratio in terms of the original test to train data sizes. The evaluation results on these 1000 samples are as follows (Fig. 19): The confusion matrix shows 498 true negatives, 259 false positives, 55 false negatives, and 188 true positives. The precision for normal samples is a high 0.90, indicating that 90% of predicted normal samples are indeed normal, while the precision for anomalies is disappointingly low at 0.42. The recall for normal samples is 0.66, meaning 66% of actual normal samples are correctly identified, whereas the recall for anomalies is higher at 0.77, reflecting the model's ability to detect 77% of actual anomalies. The overall accuracy of the model is 0.69, while The F1-scores are 0.76 for normal samples and 0.54 for anomalies, balancing the trade-off between precision and recall. The ROC score of 0.7997 is indicative of the model's exceptionally good ability to distinguish between normal and anomalous samples in comparison to the two autoencoders before it. The plot of the curve has a slow initial start that collides with the random guess diagonal, but very soon it sharply increases and quickly achieves a high true positive rate with minimal false positives, while keeping a more pronounced "bow" shape within a large distance of the diagonal (Fig. 20). The great performance of the GAN can also be seen in the prediction expectations figure, which is notably more readable because of the reduction in data points. It portrays the model as able to capture almost all patterns in both sets accurately with a minimal error rate, maintaining the overall diamond shape and class distribution (Fig. 21).

6.4. Assessment

The initial goal of the technical deliverable was to implement and evaluate at least two deep learning models tailored for network anomaly detection, using different techniques such as autoencoders, variational autoencoders, and generative adversarial networks. Given the constraints of a one-month timeframe, the developed notebooks and their results successfully accomplish this goal, providing comprehensive insights into the performance of these models on the UNSW-NB15 dataset. The deliverable effectively uses the dataset information to train and evaluate the models, thereby providing a conclusive answer to the project's objective of assessing the effectiveness of deep learning models in network anomaly detection. Each model was tested and evaluated using standard metrics like accuracy, F1 score, precision, and most importantly, the ROC AUC score.

A notable finding is that, despite its complexity, the GAN model managed to outperform both autoencoder variants. The GAN achieved a ROC AUC score of 0.7997, compared to the autoencoder's 0.7302 and the VAE's 0.6495. However, it is important to note that the GAN's performance was evaluated on a reduced dataset due to computational and time constraints, which may have influenced its results, so further work will need to be done to confirm this. The autoencoder, with its simpler architecture, demonstrated robust performance and could be considered a good balance between complexity and effectiveness for this task. One area that showed potential for improvement was the implementation of the VAE model. Although successfully implemented, it unexpectedly underperformed compared to the standard autoencoder. This suggests

that with more time and resources, further tuning of the VAE's hyperparameters or architecture could potentially enhance its performance. Additionally, the need to reduce the dataset for the GAN model highlights the scalability challenges when dealing with large datasets, especially under time constraints.

Overall, considering the limited time and resources, the technical deliverable meets the specified functional and non-functional requirements and produces informative results. The implementations clearly demonstrate the feasibility and effectiveness of using different deep learning architectures for network anomaly detection, with varying performance metrics. Despite the challenges encountered with the VAE and the dataset reduction for the GAN, the overall success in achieving the main goals of the deliverable is evident. There is clear potential for further improvement, particularly in optimizing model architectures and addressing scalability issues for larger datasets, which could be explored with additional time and resources.

7. Conclusion

This report highlights the viability and effectiveness of unsupervised deep learning models for anomaly detection in diverse fields. The scientific analysis covered key concepts like unsupervised learning, data preparation, and model evaluation, which are essential for applying deep learning to anomaly detection tasks. The report also examined previous applications, providing context for using models such as autoencoders, variational autoencoders, and GANs to detect and address unusual data patterns.

The technical component reinforced these scientific findings by implementing and assessing several deep learning models designed for anomaly detection. It provided explanations of model structures, training methods, and evaluation processes, demonstrating in practice how to adapt these models for specific anomaly detection needs. Despite the challenges concerning the time constraints, given the circumstances the models performed relatively well for the most part, confirming first hand the practicality and efficiency of unsupervised deep learning in this area.

To summarize, the project effectively demonstrated the application of unsupervised deep learning models in anomaly detection. The research clearly showed that combining theoretical knowledge with practical implementation, along with proper adjustment and optimization, can transform these models into powerful tools for identifying anomalies and minimizing potential risks across various domains.

Acknowledgment

The authors would like to thank the BiCS management and education team for the amazing work done.

References

[Ieracitano, C., Adeel, A., Morabito, F. C., & Hussain, A. (2020).] Network traffic anomaly detection via deep learning. Information, 12(5), 215.

[Truong-Huu, T., Dheenadhayalan, N., Pratim Kundu, P., Ramnath, V., Liao, J., Teo, S. G., An empirical study on unsupervised network anomaly detection using generative adversarial networks. In Proceedings of the 1st ACM Workshop on Security and Privacy on Artificial Intelligence (pp. 20-29).

[Zong, B., Song, Q., Min, M. R., Cheng, W., Lumezanu, C., Cho, D., & Chen, H. (2018, F Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In International conference on learning representations.

[Vinayakumar, R., Alazab, M., Soman, K. P., Poornachandran, P., Al-Nemrat, A., & Venkat Deep learning approach for intelligent intrusion detection system. Ieee Access, 7, 41525-41550.

[Sivasubramanian, A., Devisetty, M., & Bhavukam, P. (2024).] Feature Extraction and Anomaly Detection Using Different Autoencoders for Modeling Intrusion Detection Systems. Arabian Journal for Science and Engineering, 1-13.

[Fotiadou, K., Velivassaki, T. H., Voulkidis, A., Skias, D., Tsekeridou, S., & Zahariadis, T. Network traffic anomaly detection via deep learning. Information, 12(5), 215.

8. Appendix

	id	dur	proto	service	state	spkts	dpkts	sbytes	dbytes	rate	...
0	1	0.121478	tcp	-	FIN	6	4	258	172	74.087490	...
1	2	0.649902	tcp	-	FIN	14	38	734	42014	78.473372	...
2	3	1.623129	tcp	-	FIN	8	16	364	13186	14.170161	...
3	4	1.681642	tcp	ftp	FIN	12	12	628	770	13.677108	...
4	5	0.449454	tcp	-	FIN	10	6	534	268	33.373826	...

5 rows × 45 columns

	id	dur	proto	service	state	spkts	dpkts	sbytes	dbytes	rate	...
0	1	0.000011	udp	-	INT	2	0	496	0	90909.0902	...
1	2	0.000008	udp	-	INT	2	0	1762	0	125000.0003	...
2	3	0.000005	udp	-	INT	2	0	1068	0	200000.0051	...
3	4	0.000006	udp	-	INT	2	0	900	0	166666.6608	...
4	5	0.000010	udp	-	INT	2	0	2126	0	100000.0025	...

5 rows × 45 columns

Fig. 1: Initial entries of training and testing sets

```
Train contamination: 0.6806223302022916
Test contamination: 0.5506000097167565

Train contamination is too high: 0.6806223302022916
New train contamination: 0.31937766979770843
Test contamination is too high: 0.5506000097167565
New test contamination: 0.4493999902832435
```

Fig. 2: Computing and adjusting train and test contaminations

```
Index: 257673 entries, 0 to 82331
Data columns (total 43 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   dur                257673 non-null  float64
 1   proto              257673 non-null  object
 2   service            257673 non-null  object
 3   state              257673 non-null  object
 4   spkts              257673 non-null  int64
 5   dpkts              257673 non-null  int64
 6   sbytes             257673 non-null  int64
 7   dbytes             257673 non-null  int64
 8   rate               257673 non-null  float64
 9   sttl               257673 non-null  int64
 10  dttl               257673 non-null  int64
 11  sload              257673 non-null  float64
 12  dload              257673 non-null  float64
 13  sloss              257673 non-null  int64
 14  dloss              257673 non-null  int64
 15  sinpkt             257673 non-null  float64
 16  dinpkt             257673 non-null  float64
 17  sjit               257673 non-null  float64
 18  djit               257673 non-null  float64
 19  swin               257673 non-null  int64
...
 41  is_sm_ips_ports    257673 non-null  int64
 42  label              257673 non-null  int64
dtypes: float64(11), int64(29), object(3)
```

Fig. 3: Combined dataset details

```
Index: 257673 entries, 0 to 82331
Data columns (total 43 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   dur                257673 non-null  float64
 1   proto              257673 non-null  int32
 2   service            257673 non-null  int32
 3   state              257673 non-null  int32
 4   spkts              257673 non-null  int64
 5   dpkts              257673 non-null  int64
 6   sbytes             257673 non-null  int64
 7   dbytes             257673 non-null  int64
 8   rate               257673 non-null  float64
 9   sttl               257673 non-null  int64
 10  dttl               257673 non-null  int64
 11  sload              257673 non-null  float64
 12  dload              257673 non-null  float64
 13  sloss              257673 non-null  int64
 14  dloss              257673 non-null  int64
 15  sinpkt             257673 non-null  float64
 16  dinpkt             257673 non-null  float64
 17  sjit               257673 non-null  float64
 18  djit               257673 non-null  float64
 19  swin               257673 non-null  int64
...
 41  is_sm_ips_ports    257673 non-null  int64
 42  label              257673 non-null  int64
dtypes: float64(11), int32(3), int64(29)
```

Fig. 4: Combined dataset after applying encoding

```python
# Separate features (X_combined) and target labels (y_combined) from the dataset
# 'label' is dropped from features and target labels are inverted (1 -> 0, 0 -> 1)
#  due to excessive contamination
X_combined = combined_data.drop(['label'], axis=1)
y_combined = 1 - combined_data['label']
```

Fig. 5: Separation of features from labels and label inversion

```python
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((175341, 42), (82332, 42), (175341,), (82332,))
```

Fig. 6: Final shapes of the feature and label sets (training labels not used)

```python
autoencoder = auto_encoder.AutoEncoder(contamination=train_contamination,
                                       preprocessing=True,
                                       hidden_neuron_list=[32, 8],
                                       epoch_num=10,
                                       device='cuda',
                                       verbose=2,
                                       lr=0.00001,
                                       batch_size=32,
                                       optimizer_name='adam',
                                       optimizer_params={'weight_decay': 0.01},
                                       dropout_rate=0.4,
                                       random_state=42)
autoencoder.fit(X_train)
```

Fig. 7: Autoencoder initialization

```
Epoch 1/10, loss=1.1444, time=24.32s
Epoch 2/10, loss=1.0125, time=24.34s
Epoch 3/10, loss=0.9329, time=24.44s
Epoch 4/10, loss=0.8797, time=24.73s
Epoch 5/10, loss=0.8446, time=24.13s
Epoch 6/10, loss=0.8211, time=24.35s
Epoch 7/10, loss=0.8044, time=24.59s
Epoch 8/10, loss=0.7920, time=23.79s
Epoch 9/10, loss=0.7819, time=22.06s
Epoch 10/10, loss=0.7733, time=21.95s
```

Fig. 8: Training progress of autoencoder

```
[[31777 13555]
 [13555 23445]]
              precision    recall  f1-score   support

           0       0.70      0.70      0.70     45332
           1       0.63      0.63      0.63     37000

    accuracy                           0.67     82332
   macro avg       0.67      0.67      0.67     82332
weighted avg       0.67      0.67      0.67     82332

Autoencoder ROC:0.7302, precision @ rank n:0.6336
```

Fig. 9: Evaluation metrics of autoencoder prediction performance

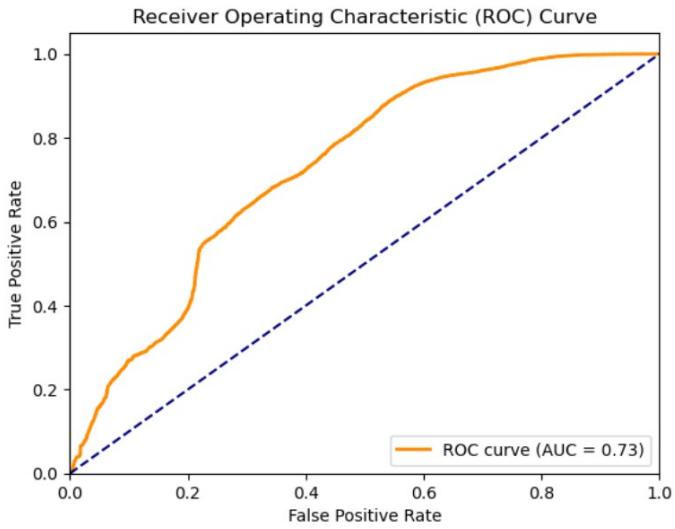

Fig. 10: Autoencoder ROC curve

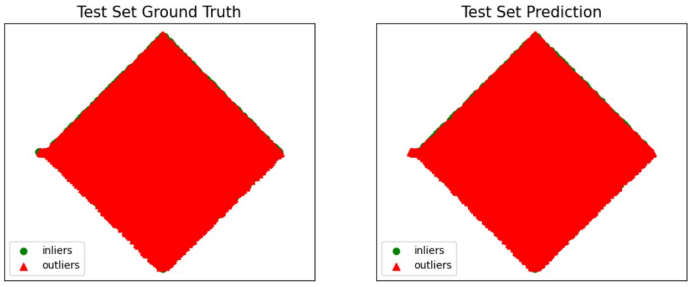

Fig. 11: Autoencoder prediction distribution

```
vae = vae.VAE(contamination=train_contamination,
              preprocessing=True,
              encoder_neuron_list=[128, 64, 32],
              decoder_neuron_list=[32, 64, 128],
              epoch_num=10,
              device='cuda',
              verbose=2,
              lr=0.001,
              batch_size=32,
              optimizer_params={'weight_decay': 0.0001},
              dropout_rate=0.4,
              latent_dim=2,
              random_state=42)
vae.fit(X_train)
```

Fig. 12: VAE initialization

```
Epoch 1/10, loss=0.9090, time=37.28s
Epoch 2/10, loss=0.8769, time=37.03s
Epoch 3/10, loss=0.8497, time=36.95s
Epoch 4/10, loss=0.7625, time=37.15s
Epoch 5/10, loss=0.7406, time=36.98s
Epoch 6/10, loss=0.7366, time=37.22s
Epoch 7/10, loss=0.7295, time=37.13s
Epoch 8/10, loss=0.7286, time=37.37s
Epoch 9/10, loss=0.7279, time=37.31s
Epoch 10/10, loss=0.7274, time=37.17s
```

Fig. 13: Training progress of VAE

```
[[29269 16063]
 [16081 20919]]
              precision    recall  f1-score   support

           0       0.65      0.65      0.65     45332
           1       0.57      0.57      0.57     37000

    accuracy                           0.61     82332
   macro avg       0.61      0.61      0.61     82332
weighted avg       0.61      0.61      0.61     82332

VAE ROC:0.6495, precision @ rank n:0.5657
```

Fig. 14: Evaluation metrics of VAE prediction performance

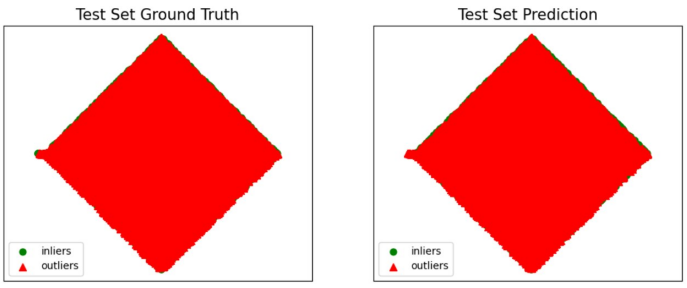

Fig. 15: VAE prediction distribution

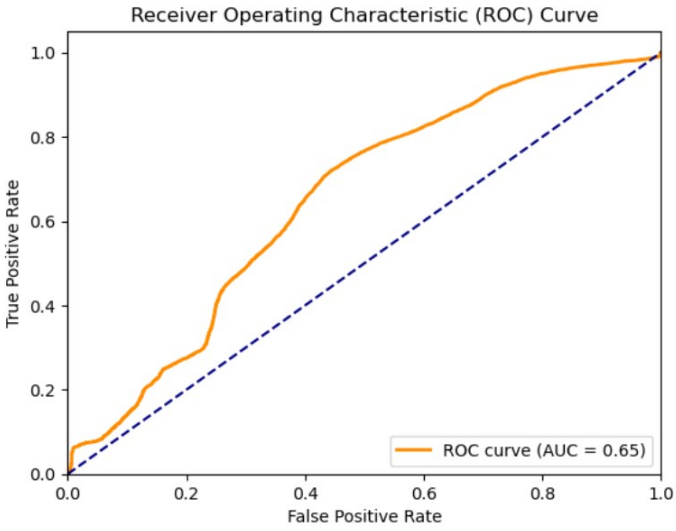

Fig. 16: VAE ROC curve

```
gan = anogan.AnoGAN(contamination=train_contamination,
                    activation_hidden='relu',
                    preprocessing=True,
                    epochs=1000,
                    learning_rate=0.00001,
                    learning_rate_query=0.00001,
                    batch_size=32,
                    dropout_rate=0.1,
                    verbose=1)
gan.fit(X_train[:5000])
```

Fig. 17: GAN initialization

```
Train iter:100
Train iter:200
Train iter:300
Train iter:400
Train iter:500
Train iter:600
Train iter:700
Train iter:800
Train iter:900
query sample 1 / 5000
iter: 0
query sample 2 / 5000
iter: 0
query sample 3 / 5000
iter: 0
query sample 4 / 5000
iter: 0
query sample 5 / 5000
iter: 0
query sample 6 / 5000
iter: 0
query sample 7 / 5000
iter: 0
query sample 8 / 5000
iter: 0
...
query sample 4999 / 5000
iter: 0
query sample 5000 / 5000
iter: 0
```

Fig. 18: Training progress of GAN, followed by query sampling phase

```
query sample 992 / 1000
iter: 0
query sample 993 / 1000
iter: 0
query sample 994 / 1000
iter: 0
query sample 995 / 1000
iter: 0
query sample 996 / 1000
iter: 0
query sample 997 / 1000
iter: 0
query sample 998 / 1000
iter: 0
query sample 999 / 1000
iter: 0
query sample 1000 / 1000
iter: 0
[[498 259]
 [ 55 188]]
              precision    recall  f1-score   support

           0       0.90      0.66      0.76       757
           1       0.42      0.77      0.54       243

    accuracy                           0.69      1000
   macro avg       0.66      0.72      0.65      1000
weighted avg       0.78      0.69      0.71      1000

GAN ROC:0.7997, precision @ rank n:0.5967
```

Fig. 19: Evaluation metrics of GAN prediction performance

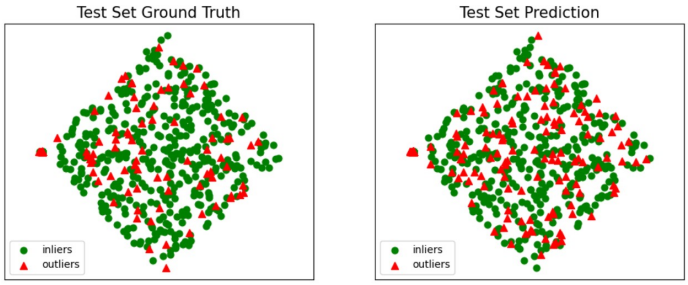

Fig. 21: GAN prediction distribution

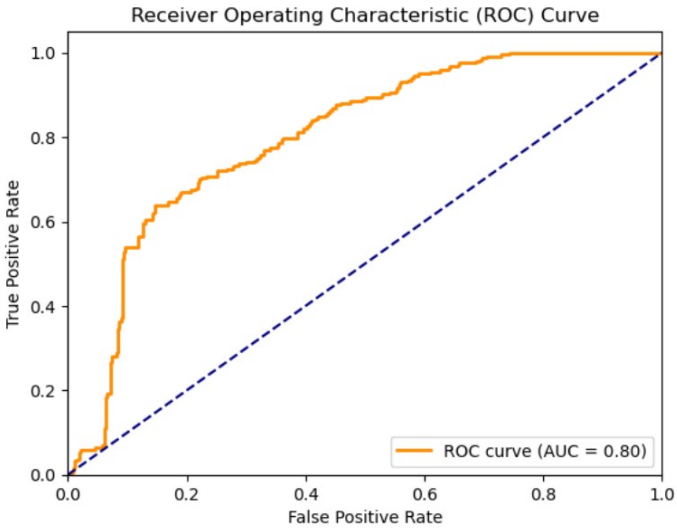

Fig. 20: GAN ROC curve