Network Traffic Anomaly Detection with Variational Autoencoder

Ningrui Li, Matthew Cagle, Yukta Dandekar, Jack Kolenbrander

1. Abstract

Network anomaly detection is a commonly implemented security feature to detect threats within networks. Traditional models, however, often fail to detect previously undetected network intrusions. To address this issue, machine learning models can be used to determine variations from a normal baseline within a network. In this study, a variational autoencoder (VAE) will be utilized, which takes input into a convolutional neural network to a smaller latent space and then attempts to recreate the input through a decoder. The reconstruction loss and Kullback-Leibler divergence loss are combined to train the data. The VAE was trained on the normal network traffic data from the NSL KDD dataset to optimize the combined loss, along with tuning the learning rate and number of epochs. With the training completed, we were able to set a threshold for the combined loss that if exceeded would be categorized as an anomaly in the dataset. Testing the VAE, we passed in a combination of attacks and normal traffic and with the loss categorized the traffic. The results were then compared to a K-nearest neighbor (KNN) and support vector machine (SVM) models that showed VAE outperformed the KNN model but did not do as well as the SVM model. Overall, the VAE with a 0.8108 accuracy score and 0.8267 average precision may be better than the K-nearest neighbors model; it may not be a viable method for a stand-alone network traffic anomaly detection. As allowing for even a single network attack to get through could be extremely costly and our model had a recall of only 0.7289 meaning almost 25% of the attacks were not detected. Thus we conclude that other models such as SVMs may be better suited for network traffic anomaly detection.

2. Background

Computer networks experience large amounts of traffic and an essential part of ensuring the safety of these networks is to detect and address security threats. Security threats typically present as anomalies, or traffic that deviates from the typical baseline functionality of the network. [1] Due to the extensive amount of traffic and data present, there are a variety of ways that anomalies and security threats can manifest themselves within a network. As a result, developing a straightforward and general model to detect network anomalies is challenging and can be interrupted by minuscule changes within the network. [1]

Anomaly classification can be completed in two ways. The first method is to take previously known threat signatures and train a machine learning model to recognize those. This allows a model to recognize threats based on pre-associated identifiers. The second method, and the benefit of applying machine learning, is to recognize previously unencountered threats by detecting variation from a typical baseline within a network. By combining these methods, a robust anomaly detection system can be produced.

Machine learning offers an adaptable and transportable solution to anomaly detection. [2] Unlike traditional anomaly detection models, utilizing machine learning allows the implementation of an anomaly detection product that can learn to detect both threats that have been detected previously as well as new threats. Common machine learning models for anomaly detection include the Holt-Winters forecasting algorithm, the Gaussian mixture model, and the Principal Component Analysis (PCA) model. [1] In this project, we will apply a variational autoencoder, as described in the following sections, to implement an anomaly detection system.

3. Design and Methodology

The data to be used in this paper will be network traffic data that contains both normal connections and bad connections which represent different types of cyber security attacks. To classify the bad connections as anomalies we must train a model on what normal behavior is for our network. Our approach is to extract the most important features from the dataset, such as duration, destination ports, number of failed logins, and if it is flagged as urgent. The last step in preprocessing the data will be to one-hot encode the categorical variables so that our model can utilize the values.

The architecture we have chosen for our anomaly detection is a variational autoencoder (VAE). A VAE is made up of two main parts: an encoder, the inference model, and the decoder, the generative model. The VAE attempts to recreate the input data while learning the latent vector. This can be seen in Figure 1 as both the encoder and decoder are convolutional neural networks with layers that decrease in size for the encoder to only hold the most important information in the latent space. The decoder layers then increase in size up to the original input vector size generating the information based on the relationships and trends the neural network has learned.

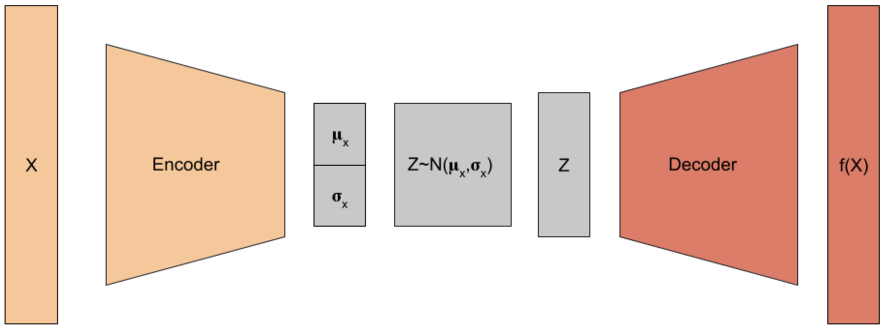

Figure 1. VAE Model Diagram

What makes the VAE different from a regular autoencoder is that the latent space is continuous. [3] This is possible because the mean and standard deviation are taken from the latent space, and then sampling from these distributions creates a new latent vector to be passed into the decoder. [5] Then the use of backpropagation is used to update the weights of the different nodes to better fit the data. The loss is calculated by combining the Kullback-Leibler divergence between the latent space and the normal distribution and the reconstruction loss. [3]

$$Loss = L(x,f(x)) + \sum_{j} KL(q_j(z|x) \, || \, p(z))$$

However, for the use of backpropagation in the training phase the sampling step must be an input thus the architecture utilizes the reparameterization trick, as seen in Figure 2.

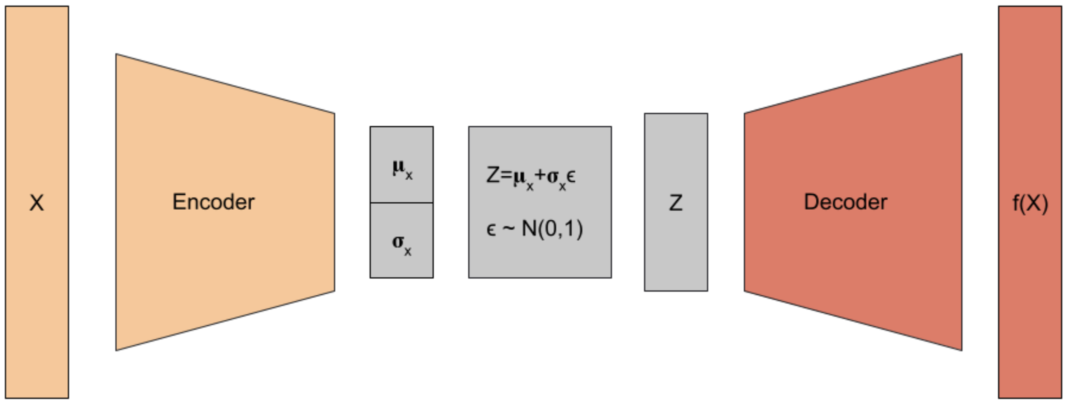

Figure 2. VAE Model Diagram with reparameterization trick

The specifics of the number of layers and the nodes of each layer will be determined based on the experimentations with the neural network.

The steps of our study are as follows:

1. **Literature review:**
    a. We will conduct a literature review on both network traffic anomaly detection and variational autoencoders.
2. **Data Preprocessing:**
    a. The NSL-KDD dataset will be cleaned and then feature extraction will be conducted.
3. **Creation of a variational autoencoder:**
    a. Program variational autoencoder in Python with the help of the library TensorFlow.
    b. Experiment with the design of the neural networks for the encoder and decoder sections.
4. **Training Phase:**
    a. At first, we will use the NSL-KDD dataset consisting of both normal and anomalous data points.
    b. K-nearest neighbors (KNN) and support vector machines (SVM) require both normal and anomalous data points for training while VAEs are trained exclusively on normal data.
5. **Testing Phase:**

    a. For each data point in the test dataset (including both normal and potentially anomalous data points), the different models will calculate a measure such as distance or loss to determine the classification.

6. **Anomaly Score:**
    a. An anomaly score will be computed for each data point based on the distances or loss values.
    b. A lower anomaly score suggests that the data point is more similar to its neighbors and is likely a normal data point, while a higher anomaly score suggests that the data point is significantly different and may be an anomaly.
7. **Thresholding:**
    a. An anomaly occurs when a data point exceeds a threshold value. Statistical analysis, visualization, or domain expertise can be used to determine this threshold.
8. **Identifying Anomalies:**
    a. Any data point with an anomaly score above the chosen threshold is classified as an anomaly.
9. **Evaluation:**
    a. Performance assessment of our anomaly detection model using appropriate evaluation metrics of accuracy, precision and recall scores.
10. **Comparison:**
    a. The performance of the VAE will then be compared with KNNs and SVMs to determine the utility of VAEs in network traffic anomaly detection.

4. Datasets and Experiments

We make use of the NSL-KDD dataset to achieve our objectives. This database, prepared by Tavallaee et al, is a modified version of the KDD CUP 99 dataset. The KDD CUP 99 dataset is a standard set of data that can be audited, which includes intrusions that have been simulated into a military network [4]. This dataset has been widely used for the evaluation of anomaly detection methods. However, Tavallaee raised concerns over the KDDCUP 99 dataset as 75%-78% of the training and testing dataset are duplicated packets. The NSL-KDD dataset eliminates the duplicate packets and decreases the overall size of the dataset to 125,000 observations for the training dataset and 25,000 observations for the testing dataset. This more manageable size allows for the use of the full NSL-KDD dataset rather than having to rely on sampling. This dataset has now become the most widely used data set for the evaluation of anomaly detection methods. Hence we make use of this dataset to perform experiments below.

4.1 Dataset Analysis

In the total of 125,973 observations in the NSL-KDD dataset. The dataset contains 41 different explanatory variables and 1 response variable, the classification of the network traffic. Of these 41 explanatory variables, 4 of them are categorical variables. Neural networks cannot accept categorical variables thus these variables were one hot encoded. This process expands a feature into different features based on the number of unique values within the column, with each new column representing if that value is present or not in the observation. Looking now into the types of network traffic represented within the dataset there are besides normal status, 12 different classes of attack, including neptune, warezclient, ipsweep, teardrop, nmap, satan, portsweep, smurf, back, guess_passwd, pod, and F, respectively. Normal and Neptune comprise over 86% of the total samples, while the rest comprise about 14%. Refer to Figure 3 for more details.

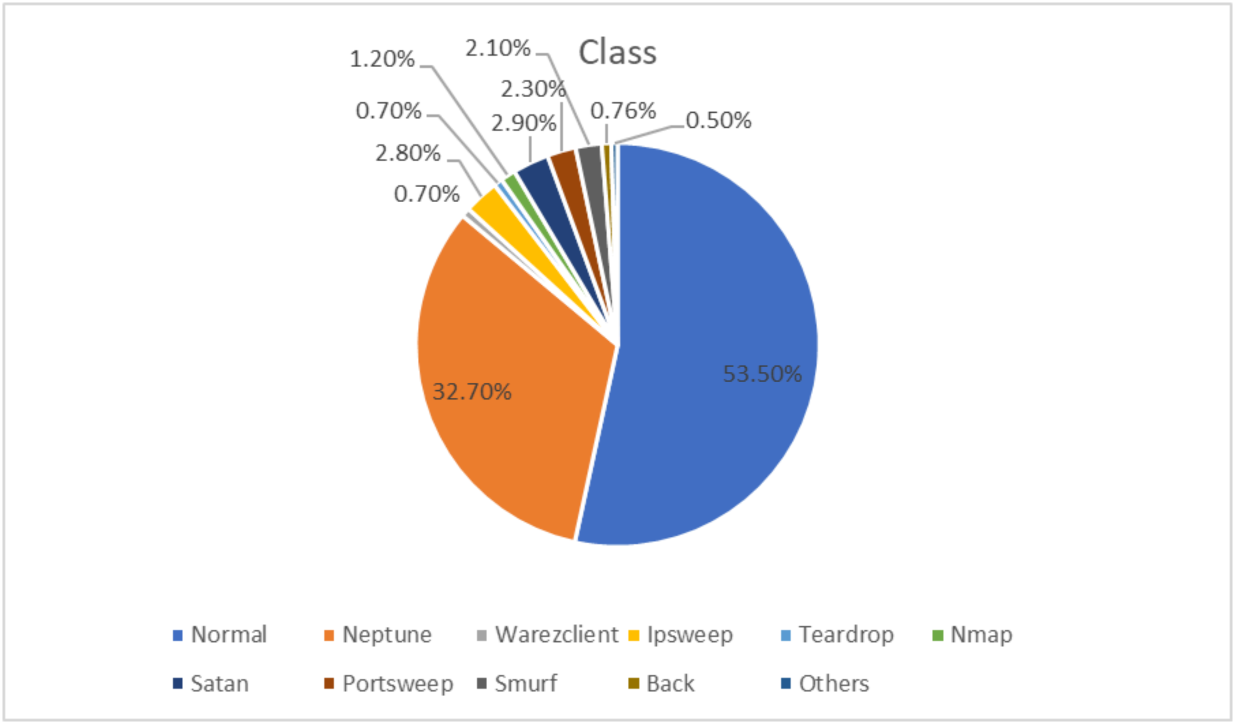

Figure 3: Dataset class of attack

4.1.1 Normal

In a normal class, variable dst_bytes are mostly positive values rather than 0 compared to other classes. In randomly selected 20 normal class samples, 19 have positive values. In normal activity, the dst_bytes field captures the data payload size. Positive values indicate the amount of data successfully transmitted. Also, positive dst_bytes values often indicate responses to requests and the establishment of a connection between different communication parties. That is why dst_bytes are mostly positive while other classes are 0 in most of the cases.

4.1.2 Ipsweep

In the Ipsweep attack, 4 error rate variables: serror_rate, srv_serror_rate, dst_host_serror_rate, dst_host_srv_serror_rate are significantly lower than other attacks. Randomly select 20 samples from the Ipsweep attack, 19 of them are 0 in 4 different variables. In Ipsweep attacks, the attackers will try to gather information about a range of IP addresses without raising suspicion. A lower error rate will help this type of attack undetected. A value being 0 means that the attacker's scanning attempts were successful, without disruptions or issues in communication with the targeted hosts.

4.1.3 Portsweep

In the portsweep attack, variables src_bytes and dst_bytes are significantly lower than other classes. In some samples, both of them are zero. A portsweep attack involves scanning different ports on a target system to find open ports and potential vulnerabilities. In those scanning processes, the amount of data exchanged (both sent and received bytes) is relatively low or zero because the portsweep attacks are focused on open ports rather than data transfer.

4.1.4 Guess_passwd

In the guess_passwd attack, variable num_failed_logins has positive correlations with variable hot. That being said, both hot and num_failed_logins will always be a non-zero value under the guess_passwd attack, and as num_failed_logins increase, hot will increase too. When the number of failed login attempts (num_failed_logins) increases, it triggers more hot indicators (hot) as the system detects the unusual amount of unsuccessful login activity, leading to a positive correlation.

4.1.5 Smurf

In the Smurf attack, both variables serror_rate and srv_serror_rate are extremely low (0 in the majority of the cases) compared to other attacks. In the random selection of 20 Smurf attack samples from the dataset, all 20 samples have 0 values for both variables. The reason could be no instances of SYN errors during the connections.

4.1.6 Compared to train data

In the normal class, train data behave similarly to the original dataset. Variable dst_bytes are mostly positive values rather than 0 compared to other attacks. In randomly selected 20 samples, 18 of them are positive values. In Ipsweep attack, train data to behave just like the original data, 4 error rate variables (serror_rate, srv_serror_rate, dst_host_serror_rate, dst_host_srv_serror_rate) are significantly lower than other attacks. Randomly select 20 samples of those 4 variables under ipsweep, all 20 of them are 0 at 4 variables. In portsweep attacks, variables src_bytes and dst_bytes are relatively low compared to other attacks, with the same result as the original dataset. Lastly, positive correlations between variables hot and num_failed_logins still exist in the training dataset.

Overall, this analysis showcases the variety of different network traffic captured by it. This is important as any network traffic anomaly detection system would need to be able to detect a variety of types of attacks as anomalies. In preprocessing our dataset we converted all of the different attack types into one class that our models will attempt to detect compared to the normal traffic data.

4.2 Experiment 1: Effect of Learning Rate and Number of Epochs on Model

The learning rate hyperparameter controls the rate or speed at which the model learns. In particular, it controls the amount of apportioned error that the weights of the model are updated with after each batch of training examples. In extreme cases, a large learning rate will result in a large weight update, which will cause the model's performance (such as its loss on the training dataset) to fluctuate. The oscillations in performance are the result of diverging weights (are divergent). A learning rate that is too small may never converge or may get stuck on a suboptimal solution. The model will never reach its optimal performance if the learning rate is too low. It is important to find the balance between learning and convergence. A too high learning rate will make the learning jump over minima but a too low learning rate will either take too long to converge or get stuck in an undesirable local minimum. The number of epochs represents the number of times the model is trained over the entire dataset. The larger the number of epochs is the more computational power and slower the training process is, as well as the possibility of overfitting the model to the training data. If the number of epochs is too small then the model may not capture all of the relationships within the data.

4.3 Experiment 2: Comparison to other models for Classification of Cyber Attacks

Utilizing a VAE requires a large amount of data and computing power for training the model along with its lack of transparency of why an observation is classified as an anomaly. These limitations may make the model a poor choice if there are other options that can detect network traffic anomalies with the same accuracy. To explore this we created KNN and SVM models to compare against the VAE. Both can be utilized for anomaly detection through one class encodings of the dataset.

5. Results and Discussions

After implementing each of the models we analyzed them based on three evaluation metrics: accuracy, precision, and recall. The formulas for these metrics are described below.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN},\ Precision = \frac{TP}{TP+FP},\ Recall = \frac{TP}{TP+FN}$$

With TP meaning true positive, TN meaning true negative, FP meaning false positive, and FN meaning false negative.

5.1 Variational Autoencoder Implementation and Analysis

In implementing the variational autoencoder model in Python, the TensorFlow library was utilized. The variational autoencoder was trained only on normal data to learn how to minimize the loss from encoding and decoding the data. In tuning the model the hyperparameters were adjusted to ensure that the model was not overfitting to the training data. Table 1 showcases the accuracy of each learning rate for the validation data.

Learning Rate	.00001	.00005	.0001	.0005	.001	.005	.01	.05
Accuracy	0.477	0.709	0.729	0.759	0.756	0.805	0.795	0.759

Figure 4. Learning rate effects on the accuracy of a validation dataset

Figure 4 supports the use of a learning rate of 0.005 to get the best accuracy on our validation dataset. The next step in tuning the hyperparameters was the number of epochs. Initially, we set the number of epochs the same for tuning the learning rate, but now that we have set the learning rate we will be selecting the number of epochs. Figure 4 shows the training loss value over all epochs, where we want to stop the training when the loss no longer decreases. We utilized the early stopping method within the TensorFlow library that stops training when the loss value stops improving.

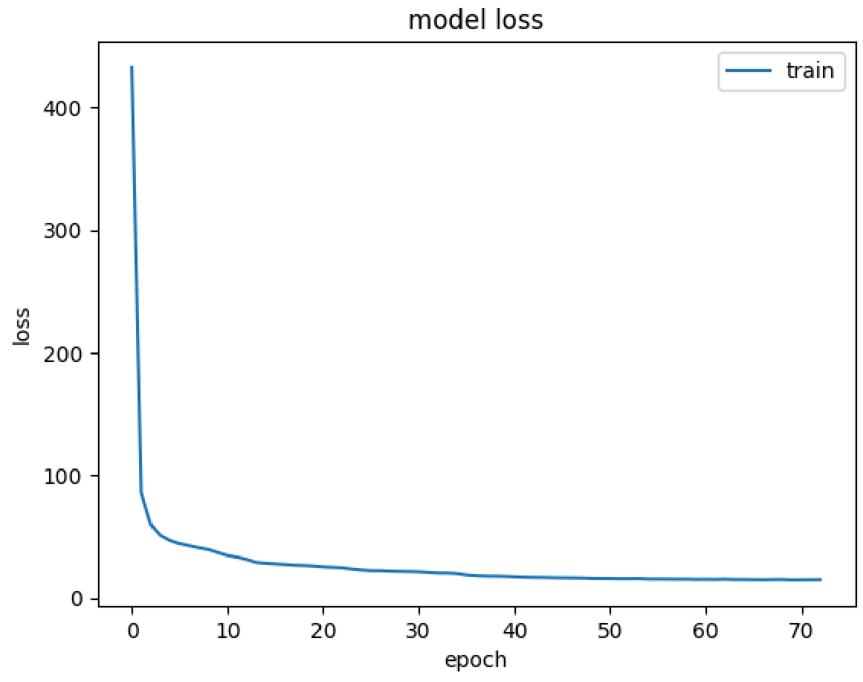

Figure 5. Model Loss over Number of Epochs

Figure 5 indicates that around the 50th epoch is when the model's loss is staying consistent. The early stopping method however allowed the number to grow to 73 epochs before stopping as three loss values in a row did not decrease. Thus the exact value of 70 epochs to utilize in training our dataset.

Now that we have experimented with both the learning rate and number of epochs, we will test the model's ability to detect anomalies within the data. For our dataset, we classify any type of cyber attack as an anomaly and the model determines if an observation is an anomaly if its loss value is outside of the 99% quantile of the loss values calculated in training, which we found to be 0.013 for our dataset. With this methodology, the model had a 0.8108 accuracy score, a 0.7289 recall value, and a 0.8267 precision score.

5.2 K-Nearest Neighbors Implementation and Analysis

To implement the K-Nearest Neighbors algorithm, Python, along with the Sklearn library, was used. The Sklearn KNN feature was then utilized to generate a fit for the KNN algorithm, using the 3 nearest neighbors. Following the training of the KNN algorithm utilizing the training data, the implementation loops through the test data set and forms a prediction for each element.

To analyze the effectiveness of the implementation, the precision and recall values were calculated using the Sklearn libraries. As shown in the table below, the model results in an accuracy of 0.7822, a precision of 0.8115, and a recall of 0.8029.

5.3 Support Vector Machine Implementation and Analysis

We want to classify the dataset and to do so we will need decision boundaries but our main task is how to find the best decision boundary. It is important to note that plotting data points in a two-dimensional graph is a straight line but when we have more dimensions we call it a hyperplane. A vector is a quantity that has magnitude as well as direction and just like numbers we can use mathematical operations such as addition and multiplication. The dot product can be defined as the projection of one vector along with another, multiplied by the product of another vector.

We built an SVM classifier to classify the NSL KDD dataset. SVM is a powerful machine learning algorithm for classification, regression, and outlier detection.
Our main objective is to select a hyperplane with the maximum possible margin between support vectors in the given dataset. SVM searches for the maximum margin hyperplane in the following two-step process:

1. Segregate classes as best as possible using hyperplanes. There are many hyperplanes that might classify the data. We should look for the best hyperplane that represents the largest separation, or margin, between the two classes.
2. We choose the hyperplane that maximizes the distance from it to each of the support vectors. Maximum margin hyperplanes define linear classifiers that define maximum margins, and maximum margin classifiers are known as maximum margin hyperplanes.

The SVC class from the Sklearn library in Python allows us to build a kernel SVM model (linear as well as non-linear). The default value of the kernel is 'rbf', because it is nonlinear and gives better results as compared to linear. The results of accuracy, precision, and recall are shown in Figure 6.

	Actual Positive	Actual Negative
Predicted Positive	105201	22
Predicted Negative	1824	16458

Figure 6. Confusion Matrix for SVM

From the confusion matrix we can extrapolate the accuracy, precision, and recall of the model which are 0.985, 0.862, and 0.798 respectfully. Out of 106,105 instances of normal traffic, the model correctly classified 105,201 as normal and 22 as anomalous. In summary, the SVM model has high accuracy and performs well in classifying both normal and anomalous traffic.

5.4 Model Comparison

We now want to compare the models on how well they were able to detect the anomalies based on the accuracy, precision, and recall scores.

	VAE	KNN	SVM
Accuracy	0.8108	0.782	0.985
Precision	0.8267	0.812	0.862
Recall	0.7289	0.803	0.798

Figure 7. Comparison of Model Metrics

Comparing the three models in Figure 7, the VAE outperforms the KNN in accuracy and average precision but has a lower recall score. The SVM outperforms VAE and KNN in accuracy and precision but has a slightly lower recall score than KNN.

6. Conclusions

This paper presents a method for anomaly detection in network traffic through the use of a VAE. The NSL KDD dataset was utilized for the experiment as it is an improved version of the KDD CUP 99 dataset that is the standard set of data for testing anomaly detection in network traffic. The dataset includes various types of intrusions that have been simulated into a military network to mitigate the risk of missing any type of potential threat. After training and testing the various models, we found that the SVM model outperforms both VAE and KNN in accuracy and precision, demonstrating its strength in accurately classifying both normal and anomalous traffic and minimizing false positives. Nevertheless, SVM has a lower recall score than KNN, indicating that it is less effective than KNN at capturing all actual anomalies, potentially leading to more false negatives. In conclusion, a VAE can be utilized for network traffic anomaly detection but there are other models such as SVMs that may be better suited for the problem.

Contributions:

Abstract - Jack and Matthew
Aims - Ningrui and Jack
Background - Jack
Design and Methodology - Matthew and Yukta
Datasets and Experiments - Matthew and Yukta
Data Analysis - Ningrui
Variational Autoencoder Results - Matthew
K-Nearest Neighbor Results - Jack
Support Vector Machine Results - Yukta
Conclusion - Matthew

7. References

[1] “Machine learning approaches to network anomaly detection,” USENIX, https://www.usenix.org/legacy/event/sysml07/tech/full_papers/ahmed/ahmed_html/sysml07CR_07.html (accessed Oct. 1, 2023).

[2] S. Wang, J. F. Balarezo, S. Kandeepan, A. Al-Hourani, K. G. Chavez and B. Rubinstein, "Machine Learning in Network Anomaly Detection: A Survey," in IEEE Access, vol. 9, pp. 152379-152396, 2021, doi: 10.1109/ACCESS.2021.3126834.

[3] Rowel Atienza. 2018. Advanced Deep Learning with Keras: Apply deep learning techniques, autoencoders, GANs, variational autoencoders, deep reinforcement learning, policy gradients, and more. Packt Publishing.

[4] M. Tavallaee, E. Bagheri, W. Lu and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications, Ottawa, ON, Canada, 2009, pp. 1-6, doi: 10.1109/CISDA.2009.5356528.