Graph Attention Network-Based Monitoring of Complex Operational Systems

Ivar Källander

Stanislaw Swirski

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6249
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2024

Abstract

Operational systems, such as industrial automation, autonomous vehicles, and larger air/sea/landcrafts, often contain a large number of heavily connected systems with real-time requirements for their functionality. For such systems, detecting and responding to anomalies is both challenging and crucial. Until recently, such anomalies were monitored using heuristic methods, or even humans monitoring the systems. Such approaches often fail to detect anomalies accurately due to the complexity of the systems. A continuous development of the systems also poses a significant challenge, as the current anomaly detectors have to be updated, and staff trained. Geometrical deep learning is a well known tool used for anomaly detection in applications where the data can be represented as a graph. However, to our knowledge it is yet to effectively be used for complex operational systems, currently only being used for simpler cases such as fraud detection. Recently, a new architecture named *Graph Attention Network* (GAT) has been studied as an anomaly detection method. Its ability to incorporate information in large networks makes it potentially useful for complex operational systems. In this thesis we evaluate different machine learning based methods for anomaly detection, trained on data from real operational systems, focusing on submarines. The methods evaluated include GCNs, GATs and Autoencoders. We also evaluate which data preprocessing methods that are best suited for our case. The results of this thesis provide a basis for further research and show that GATs could be successfully implemented as anomaly detectors for complex operational systems, though the usage may not be justified without sufficient data and complexity.

Acknowledgements

We want to thank Saab Kockums and our supervisor Mikael Lindberg for his invaluable guidance and support during our master thesis.

We would also like to thank our supervisor from LTH, Johan Eker for his contributions and assistance during our process.

Abbreviations

GNN Graph Neural Network

GAT Graph Attention Network

GCN Graph Convolutional Network

CNN Convolutional Neural Network

MTAD-GAT Multivariate Time-series Anomaly Detection via Graph Attention Network

GD Gradient Descent

SGD Stochastic Gradient Descent

ADAM Adaptive Moment Estimation

GRU Gated Recurrent Unit

MSE Mean Squared Error

MAE Mean Absolute Error

TE Total Error

TP True Positives

TN True Negatives

FP False Positives

FN False Negatives

Contents

1

Introduction

Submarines are underwater vessels, mostly used for military purposes. The first submarines were powered by hand, but as motors were invented, these were adopted mainly as generators for batteries that run the vessel while submerged. The first submarine powered by an electrical battery was invented by a polish engineer in 1881. With the additional invention of underwater torpedoes, submarines became important tools for Germany, UK, US and Russia, in the first and second World Wars.

Submarines have been in use by navies since the 19th century, both for offense towards enemy crafts, and as powerful reconnaissance operators. Due to their stealthy nature, communication with the outside need to be kept severely limited, making it exceedingly important with local maintenance capability. Another reinforcing factor is the duration of submersion, where a craft may not resurface for multiple weeks.

Anomaly detection is a crucial part of successful submarine operations, and small errors can potentially lead to critical situations. Historically, we have seen several submarine accidents of varying degrees, and to highlight the importance of proper security measures, we will mention a few:

- **HMS Thetis (1939):** A British submarine that sank during a trial, unfortunately causing the death of 99 people. The problem was an open torpedo tube, and the lack of sufficient warning systems lead to only four people being saved.
- **K-141 Kursk (2000):** A Russian submarine explosion caused the loss of the full crew of 118 people. Insufficient communication is thought to have been a contributing factor, and an anomaly detection system could potentially have prevented the catastrophe.

A submarine can be a stressful environment for the crew, and as technology has advanced more load has instead been shifted to computers. However, this has introduced new challenges that must be handled, such as maintenance complexity, cybersecurity threats and various other technical challenges related to the communication between components.

Operational systems, such as industrial automation, autonomous vehicles, and larger air/sea/landcrafts, often contain many heavily connected systems with real-time requirements for their functionality. Events and patterns leading to costly and dangerous problems can therefore be hard to detect. Additionally, the unique underwater environment, with high pressure and limited visibility contributes to the difficulty of the task for personnel.

Saab Kockums specializes in building sophisticated naval vessels for the Swedish

Armed Forces. Due to the increasing technical complexity of each new generation of naval vessels, technical maintenance at sea can be challenging. At the same time the safety and security requirements for these crafts are of utmost importance, as diverse problems can cause loss of human life or cause a risk for the security of the craft's mission.

When an accident occurs, it is critical to correctly identify the cause of the problem, as well as the solution. This often requires a deep understanding of how the systems work, and it can be difficult to educate the personnel of the craft to the extent needed to safely respond to an incident. Monitoring systems that could make detecting anomalies easier are therefore of great interest to Saab.

1.1 Motivation

Types of anomalies

- **Cybersecurity breaches**: These anomalies can often manifest in the form of unusual data traffic or data spikes that could signal a Distributed Denial-of-Service (DDoS) attack. Another example would be unusual data payloads which could for example signal code injections. Such anomalies are of uttermost importance to the safety of the crew, and potentially a larger organization, and should therefore be treated accordingly. (Further reading [Krishna Kishore et al., 2023])

- **Operational Anomalies**: This type of anomaly signals a malfunction or a breakdown in the system or process being monitored. This could be as simple as a cable breaking, a software bug, or a whole part of a system malfunctioning. This could manifest in high signal loss, corrupted data packets or complete absence of signals from a system. Although not as critical in normal scenarios, such errors could prove quite significant in stressful and demanding situation such as naval battle. (Further reading [Lutz and Carmen Mikulski, 2003])

- **Timing Anomaly**: This anomaly occurs when there is a lack of synchronization among critical systems, such as sensors or internal clocks. This could mean measurements in regards of the naval crafts surroundings, as well as out of sync steering/navigation, which could in turn could lead to inaccurate environmental assessments, compromised navigational accuracy, or even errors in tactical decisions. For example a non-synchronized navigation and autopilot system could result in incorrect course plotting, endangering the craft and its crew. Furthermore, timing anomalies during stealth operations could lead operational failures, for example if a craft is not synchronized with other crafts. (Further reading [Lisova et al., 2016])

- **Wear and Tear Anomaly**: This is associated with gradual degradation of submarine components, such as sensors or mechanical parts. due to wear and tear over time. Unlike other failures, wear and tear anomalies manifest as a slow decline in performance and reliability, which can be challenging to detect early. (Further reading [Michałowska et al., 2021])

The current operational processes

Warships and submarines have complex operating systems that control their navigation and automation. These systems require proper anomaly detection, but the existing solutions do not currently leverage artificial intelligence or machine learning, due to their complex structures.

Modern anomaly detection is mostly done manually by the crew of the seacraft. When something is wrong, it must firstly be detected by the crew, and later also diagnosed. This exposes the systems to human factor, which can in stressful, or more complex situations lead to errors in diagnosis and response to incidents/anomalies.

Graph Based Anomaly Detection

Graph-based anomaly detection involves identifying irregular patterns or behaviors in data that are structured as graphs. In this approach, nodes represent entities, and edges depict relationships between them. Anomalies could manifest as unexpected changes in the connectivity, structure, or attributes of these entities. The detection process leverages graph properties like centrality, clustering, and subgraph patterns to discern deviations from the norm. This method is particularly effective in contexts where relationships are complex and critical, such as in fraud detection, cybersecurity, and social network analysis. By revealing subtle and unexpected structural changes, graph-based anomaly detection provides valuable insights into the underlying data.

Graph Attention Networks

Graph Attention Networks (GAT) [Veličković, 2024] are a new machine learning architecture that can represent complex data with irregular structures. They allow for training on any type of graph, such as social or transportation networks. It is therefore reasonable to investigate their potential in complex operational systems.

Neural Networks work by taking numerical data points and feeding them to a number of hidden layers, in which nodes are trained by a set of independent weights. These nodes can produce an output result, e.g. a prediction. There are many variants of Neural Network architectures that work well for different applications, but relevant to this work is the class called Graph Neural Networks. One such architecture, Graph Convolutional Networks (GCN) is the first attempt towards a generalized CNN, but comes with a few limitations. Particularly GCNs lack the ability to learn

importance of single nodes, and treat all nodes equally. GATs introduce an attention mechanism, which allow for arbitrary importances between nodes.

1.2 Approach

With enough relevant data it should be possible to train a machine learning model that could detect anomalies. For this application, networks able to reconstruct data are suitable, and GNNs or trained GATs could be used for feature extraction. These models could potentially be used for anomaly detection.

1.3 Goal

The goal of this thesis is to dive deeper into Graph Attention Networks and how they can be used, specifically for complex operational systems. Our hypothesis is that the underlying system is not complex enough for GAT to provide a significant improvement over simple GCNs.

As there are simpler alternatives such as GCNs, we will compare the GAT's performance to different models. We will also examine if the complexity of the MTAD-GAT model (see Section 2.10) provides significant advantage over simpler models that use the GAT architecture.

Furthermore, we will explore different models such as simple autoencoders that would analyze individual packets. Our motivation for this is that these models could complement the GAT models in identifying exactly where an anomaly occurred, as the GAT/GCN models in this thesis can only detect when an anomaly occurred, but not where in the system.

Our goal is to understand more about Graph Attention Networks, more specifically to examine their use in complex operational systems, such as ship automation and navigation systems onboard warships and submarines. We aim to investigate whether GATs have an advantage over simpler GCNs, and whether the systems are suitable for such architectures.[1]

1.4 Related Work

Since 2018 there have been a substantial number of methods developed around Graph-based Time-Series Anomaly Detection, which were reviewed in a survey [Ho et al., 2024]. According to the study, these methods have been used in various

[1] The code is available at `https://github.com/ivkall/GAT-Monitoring-OpSys`

real-world and industrial applications, such as water systems, as well as traffic and social networks and video data. It mentions that regarding signal data, there are a few categories that are based on either autoencoders, Generative Adversarial Networks (GAN), prediction or self-supervision. One of the most recent contributions are MST-GAT: A Multimodal Spatial-Temporal Graph Attention Network for Time Series Anomaly Detection [Ding et al., 2023], which looks at diverse time series with spatial-temporal relationships and is trained on graph structures in both dimensions.

Another GAT method, targeted for water systems, is Graph Deviation Network (GDN) [Deng and Hooi, 2021]. It works by reading the time-series of N sensors, and associating with each one an embedding vector $\mathbf{v}_i$, which is initialized randomly. It builds a directed graph where nodes represent sensors and edges a dependency relation. We can choose to include all sensors, or a selection, based on prior knowledge of our data structure to be used as candidates $\mathscr{C}_i$ of sensors dependent on sensor i. Dependencies are learned using the dot product

$$e_{ji} = \frac{\mathbf{v}_i^\top \mathbf{v}_j}{\|\mathbf{v}_i\| \cdot \|\mathbf{v}_j\|} \quad \text{for } j \in \mathscr{C}_i$$

and we store 1 at the indices of the top k e_{ji} in the otherwise zero-valued learned adjacency matrix A. The next step is taking a time window of w sensor values that, using the GAT feature extraction, produces a prediction of the vector of sensor values at time t, $\hat{\mathbf{s}}^{(t)}$. The difference compared to the observation $\mathbf{s}^{(t)}$ is used to calculate the deviation score $A_s(t)$. An anomaly is regarded as a score exceeding a fixed threshold. The authors showed that GDN outperformed the baseline for accuracy.

More specifically related to this work is the original paper on Graph attention networks by Veličković et.al., from 2018, which lay out the foundation of the GAT. The main breakthrough with this model was the ability to assign different weights to different neighbors of nodes, unlike classical GCNs where all neighbors had the same weights. This allowed for the model to learn much more complex relations, and also learn the importance of single nodes in a graph. The result described in the paper proved that GATs were better at learning the structure of networks than many other models such as a simple multi-layered perceptron and GCN, even though the difference between the GAT and GCN was marginal. This paper covered mainly the clustering and classification of nodes.

Also related is the paper on Multivariate time-series anomaly detection via graph attention network by Zhao et.al., from 2020, which introduce the MTAD-GAT algorithm. This paper evaluated the use of GATs for analyzing the relations in time series, and the aim was to detect anomalies in these time series. The data consisted

of for example CPU, GPU and memory usage in a computer. This paper proved that MTAD-GAT outperformed state of art methods, however not significantly.

2

Background

2.1 Introduction

For a long time now, machine learning has been one of the captivating fields in human development. Its significance has, however, increased rapidly in the past years. Major improvement in computational power, universal data spread, and, most importantly, a series of new algorithmic approaches have accelerated machine learning's adoption in technology industry. In this field, the thesis majorly focuses on machine learning whereby a number of types of neural networks are used.

Neural networks have gained the center stage of attention because of their capability toward complex pattern recognition and predictive analysis.

Therefore, it is capable of solving problems across a large and diversified domain, for example, from image recognition, natural language processing, to predictive maintenance. The next chapter will set out the theoretical basis necessary to understand the methodology, experiments, and results laid down in this thesis. Namely, it will review some of the key concepts in the architecture of many other types of neural networks.

2.2 Types of Machine learning

Supervised Learning

Supervised learning represents a labeled training dataset; that is to say, in supervised learning, there is always a corresponding label for each of the training instances. The model should be able to predict the output for any new or unseen data after learning patterns from the training set. The main two types under this category of learning include classification, where the outputs are classes or categories, and regression, where the outputs are continuous values or set numbers. Basically, the performance of supervised learning models is majorly provided in terms of accuracy to predict the output of a test set that is separate from the data used in training.

Supervised learning finds applications in fields as diverse as image recognition or speech recognition.

Unsupervised Learning

Unsupervised learning is a class of machine learning techniques in which the algorithm learns patterns from unlabeled data. Most of the time, the goal is to find out the underlying structure of data with no, or at best very loose guidance by way of explicit outputs or labels. The most important tasks of unsupervised learning are clustering, or the division of data into sets with features that are even more alike among themselves, and dimensionality reduction, which subsequently reduces the high complexity of the data down to its essential aspects. In the case of anomaly detection, these unsupervised methods are also brought to bear for the finding of data points that exhibit large deviations away from the rest and in associative tasks to derive rules explaining large parts of the data. Unsupervised learning provides very useful techniques whenever labeled data is sparse or one tries to reveal the hidden patterns and relationships in the data by itself.

Semi-Supervised Learning

Semi-supervised learning stands as an intermediary approach in machine learning, utilizing both labeled and unlabeled data for training. Typically, a small amount of labeled data alongside a larger pool of unlabeled data is used. This method leverages the labeled data to learn a preliminary structure or pattern and then applies this understanding to categorize or derive insights from the unlabeled data. Semi-supervised learning is particularly useful when acquiring labeled data is expensive or labor-intensive but unlabeled data is abundant. Common applications include text and image classification, where labeling large datasets can be prohibitively costly. By harnessing both labeled and unlabeled data, semi-supervised learning can significantly improve learning accuracy and model performance compared to unsupervised learning, especially when labeled data is scarce. (Further reading [Chapelle et al., 2006])

2.3 Neural Networks

Multi-layered perceptron

Given a Multi-Layered Perceptron (MLP) [Cybenko, 1989] (see Figure 2.1) with L layers, where the first $L-1$ are hidden layers and layer L is the output layer, the mathematical operations of the MLP can be described as follows:

1. **Input Layer to Hidden Layers:** Let $\mathbf{x}$ denote the input vector. For the l-th layer ($1 \leq l < L$), the transformation can be represented as:

$$\mathbf{h}_l = f(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{b}_l)$$

where $\mathbf{h}_0 = \mathbf{x}$ is the input, $\mathbf{W}_l$ represents the weights, $\mathbf{b}_l$ is the bias, and f is the activation function.

2. **Propagation Through Hidden Layers:** The output of each layer serves as the input to the next, enabling the network to learn complex functions.

3. **Output Layer (Layer L):** The final output is computed as:

$$\mathbf{o} = g(\mathbf{W}_L \mathbf{h}_{L-1} + \mathbf{b}_L)$$

where g is the activation function suited for the output, which could be different from f.

4. **Learning Process:** The network adjusts its weights and biases to minimize a loss function, typically using gradient descent or its variants.

5. **Backpropagation:** Backpropagation computes gradients of the loss with respect to all weights and biases for updating the network parameters, facilitating learning from the data.

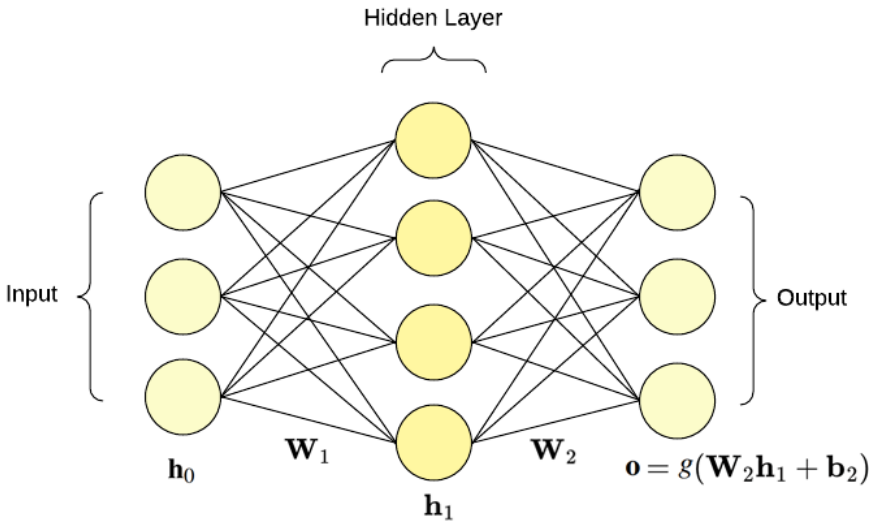


Figure 2.1 A Multi-Layered Perceptron with one hidden layer.

Recurrent Neural Network

Recurrent Neural Networks (RNN) are used for time sequence data, and can be represented as a MLP like structure, with as many layers as the length of the sequence. An example architecture can be seen in Figure 2.2.

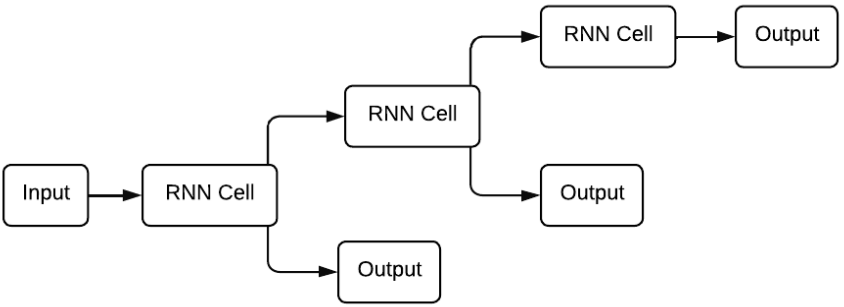


Figure 2.2 RNN architecture with a sequence length of three.

Gated Recurrent Unit

A Gated Recurrent Unit (GRU) [Cho et al., 2014] is an extension of a classical RNN that has two gates, an update gate that determines how much of the current value to keep, and a reset gate that determines how to combine the new input with the previous value.

- The **update gate** z_t is calculated as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

  where x_t is the input at time step t, h_{t-1} is the previous hidden state, W_z and U_z are weights for the input and previous hidden state, respectively, b_z is the bias, and σ is the sigmoid function.

- The **reset gate** r_t is computed by:

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

  with W_r, U_r, and b_r being the weights and bias associated with the reset gate.

- The **candidate hidden state** $\tilde{h}_t$ is given by:

$$\tilde{h}_t = \tanh(W x_t + U(r_t \odot h_{t-1}) + b)$$

  where W and U are the weights for the input and the gated hidden state, b is the bias, and $\odot$ denotes element-wise multiplication.

- The **actual hidden state** h_t at time t is updated as:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

  This equation linearly interpolates between the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$ based on the update gate z_t.

Long Short-Term Memory

Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] networks are another RNN extension. It uses a memory cell that allows it to selectively remember or forget information over varying time series. It contains an internal memory c_t that allows memory to flow through the network without much modification, and the following gates:

- **Forget gate** f: determines what information from the memory to discard or forget.
- **Input gate** i: decides which new information to store in the internal memory.
- **Output gate** o: chooses what information to output based on the current input and the memory of the cell.

The network is represented in Figure 2.3.

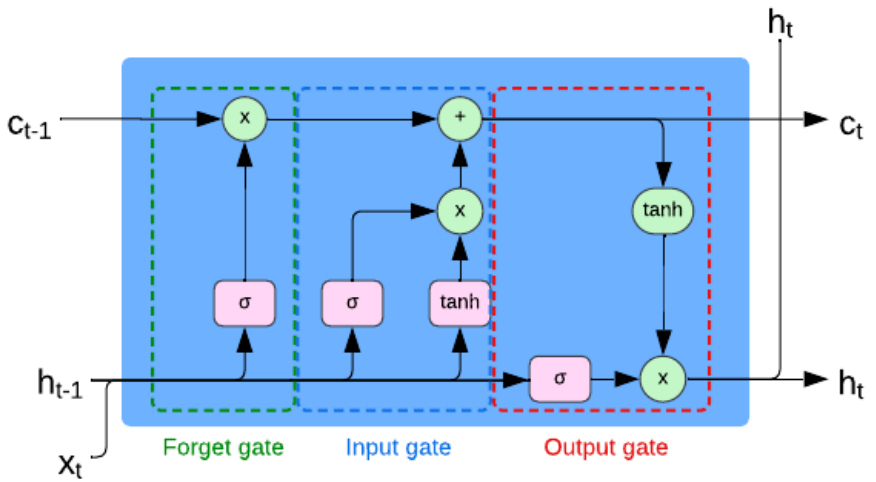


Figure 2.3 LSTM network overview.

2.4 Standard Autoencoders

Autoencoders are neural network architectures similar to multilayer perceptrons (MLPs) designed to learn how to reconstruct data. The primary objective of an autoencoder is to compress input information into a lower-dimensional representation and then reconstruct the original data from this compressed representation. The architecture of an autoencoder can be seen in Figure 2.4.

Encoder

The encoder in a standard autoencoder compresses the input data x into a lower-dimensional latent representation z. The encoder directly outputs the latent representation without trying to approximate any posterior distribution:

- The encoder transforms the input into a latent representation:

  $$z = f_\phi(x)$$

  where f_ϕ is a deterministic function parameterized by neural network parameters ϕ.

Decoder

The decoder part of the standard autoencoder aims to reconstruct the input data from the latent representation:

- The output of the decoder is the reconstruction $\hat{x}$, which tries to approximate the original input x:

  $$\hat{x} = g_\theta(z)$$

  where g_θ is a function parameterized by neural network parameters θ, designed to map the latent representation back to the data space.

Loss Function

The loss function of a standard autoencoder is the reconstruction error between the input and the output, measured as mean squared error (MSE) , mean absolute error (MAE) or total error (TE):

- The autoencoder loss L consists of a single term, which is the reconstruction loss:

  $$L(\theta, \phi; x) = ||x - \hat{x}||^2$$

  This loss measures the difference between the input data x and its reconstruction $\hat{x}$, often using the squared Euclidean distance.

2.5 Variational Autoencoders

Encoder

The encoder part of a Variational Autoencoder (VAE) tries to approximate the true posterior distribution of the latent variables z given an input x, with a variational approximation $q_\phi(z|x)$. This is often modeled as a Gaussian distribution with parameters (mean μ and variance σ^2) that are functions of the input:

- The encoder outputs parameters for the latent distribution:

$$q_\phi(z|x) = \mathcal{N}(z; \mu_\phi(x), \sigma^2_\phi(x))$$

where $\mu_\phi(x)$ and $\sigma^2_\phi(x)$ are outputs from neural networks with parameters ϕ.

Reparameterization Trick

To enable gradient descent through the sampling process, VAEs use the reparameterization trick:

- A sample z from the latent distribution is obtained by:

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \varepsilon$$

where $\varepsilon \sim \mathcal{N}(0, I)$ and $\odot$ denotes element-wise multiplication.

Decoder

The decoder part of the VAE generates data by taking samples from the latent space and mapping them back to the data space:

- The decoded output is given by:

$$p_\theta(x|z) = \mathcal{N}(x; \mu_\theta(z), \sigma^2_\theta(z))$$

where $\mu_\theta(z)$ and $\sigma^2_\theta(z)$ are determined by neural networks with parameters θ.

Loss Function

- The VAE loss L consists of two terms:

$$L(\theta, \phi; x) = -\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] + \mathrm{KL}[q_\phi(z|x)||p(z)]$$

where the first term is the reconstruction loss, and the second term is the Kullback-Leibler divergence between the approximated posterior $q_\phi(z|x)$ and the prior $p(z)$ over the latent variables.

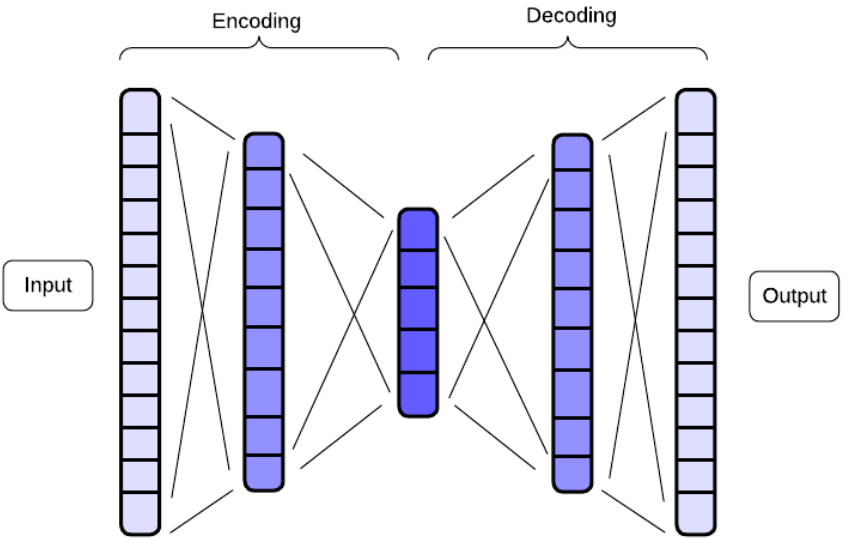


Figure 2.4 Autoencoder architecture

2.6 Graph Neural Networks and the Graph Attention Network variation

Graph Neural Network

Graph Neural Network (GNN) [Scarselli et al., 2009] is a form of geometric deep learning utilizing graph architectures. Parallels can be drawn between GNNs and Convolutional neural networks. In terms of computer vision, the graph used in convolutions would consist of pixels as nodes, and the neighborhood relations as edges. Graph neural networks let us apply "convolutions" on asymmetric structures where different nodes might have different amounts of neighbors.

Message passing layers is the name of the process of mapping the current graph to a new one using diverse mechanisms. This mechanism can come in form of averaging the neighbors' features, or taking the max or min.

Given a graph $G = (V, E)$ with nodes $v \in V$ having feature vectors x_v and edges E, the message passing mechanism in GNNs involves:

1. **Message Computation:** For each edge $(u, v) \in E$, the message m_{uv} is computed as:

$$m_{uv} = M(x_u, x_v, e_{uv}) \tag{2.1}$$

where M is the message function, and e_{uv} are the edge features, if available.

2. **Feature Update:** The feature vector of each node is updated by aggregating messages from its neighbors:

$$x'_v = U(x_v, \sum_{u \in N(v)} m_{uv}) \tag{2.2}$$

where x'_v is the updated feature vector, U is the update function, and $N(v)$ denotes the set of neighbors of node v.

After this process another machine learning architecture takes over, such as regular neural networks.

Graph Attention Network

A Graph Attention Network [Veličković et al., 2018] is essentially a message passing architecture which allows the edges of the graph to be trained, meaning more meaningful edges will get heavier weights, while insignificant ones will get lighter weights. Parallels can be drawn that in a social network graph two close friends would have a heavy weight, and two distant people would have a light weight (even though technically being friends). Figure 2.5 shows how neighboring features are aggregated into a new feature.

In GATs, the message function with an attention mechanism can be defined as follows for each edge $(i, j) \in E$:

$$e(\boldsymbol{h}_i, \boldsymbol{h}_j) = \text{LeakyReLU}\left(\boldsymbol{a}^T \cdot [\boldsymbol{W}\boldsymbol{h}_i \| \boldsymbol{W}\boldsymbol{h}_j]\right) \tag{2.3}$$

$$\alpha_{ij} = \frac{\exp\left(e(\boldsymbol{h}_i, \boldsymbol{h}_j)\right)}{\sum_{j' \in \mathcal{N}(i)} \exp\left(e(\boldsymbol{h}_i, \boldsymbol{h}_{j'})\right)} \tag{2.4}$$

where:

- $e(\boldsymbol{h}_i, \boldsymbol{h}_j)$ is a scoring function that indicates the importance of node i's features to node j.
- α_{ij} is the softmax normalized attention coefficient.
- $\boldsymbol{W}$ is a weight matrix applied to every node feature vector $\boldsymbol{h}_i$.
- $\boldsymbol{a}$ is a learnable weight vector of the attention mechanism.
- $\|$ denotes concatenation.
- LeakyReLU is the activation function, introducing non-linearity, defined as

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0, \\ 0.01x & \text{otherwise.} \end{cases}$$

- $\mathcal{N}(i)$ denotes the set of neighbors of node i in the graph.

This mechanism enables the network to dynamically assign importance to neighbors' information based on their feature vectors, enhancing the model's ability to capture complex patterns in the data.

Feature extraction: The new node features from the GAT are calculated by concatenating K outputs (Multi-head attention) from the activation function σ applied to the aggregated features across neighborhoods:

$$\boldsymbol{h}_i' = \Big\|_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \cdot \boldsymbol{W}^k \boldsymbol{h}_j\right)$$

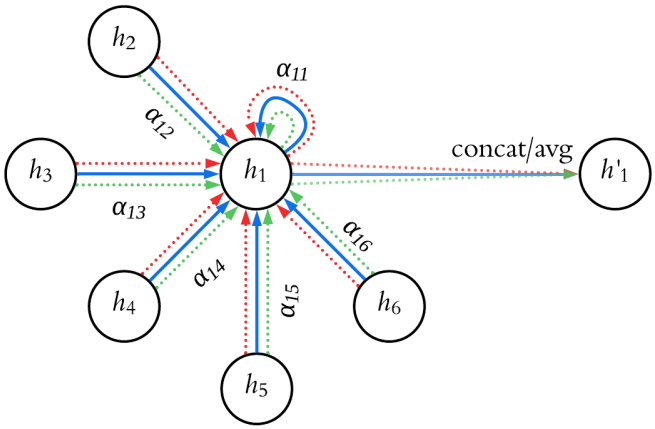


Figure 2.5 Multi-head attention for K=3.

Applications of GATs in Machine Learning

GATs are designed for processing data structured as graphs and are utilized in various complex data interconnection tasks such as in social networks, molecular structures, or communication networks. GATs are generally used as a way to improve other GNN models. The following are key applications of GATs in machine learning:

1. **Node Classification:** GATs excel in classifying nodes within a graph, leveraging an attention mechanism to weigh the influence of neighboring nodes. This is especially useful in applications like social network analysis for identifying individual roles, or in citation networks.

2. **Link Prediction:** This involves predicting potential links between nodes, crucial for applications in recommendation systems, social network analysis, and bioinformatics. For example, predicting potential friendships in social networks.

3. **Graph Classification:** GATs can classify entire graphs, applicable in fields such as chemistry for predicting molecular properties, or in biology for classifying protein interactions. For example, classifying water solvable molecules.

4. **Time-series Forecasting on Graphs:** In scenarios where nodes represent entities with temporal dynamics, GATs help in forecasting future states by considering both temporal changes and inter-node connections. Applications include traffic forecasting and dynamic pricing.

5. **Knowledge Graph Completion:** GATs aid in completing knowledge graphs by inferring missing relations or entities based on existing graph structures and node features.

6. **Fraud Detection:** In financial networks, GATs can detect unusual patterns suggesting fraudulent activities by analyzing transactions and relationships within the graph.

2.7 Optimization Methods

Gradient Descent

Gradient Descent (GD) is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In the context of machine learning, it is used to minimize the loss function:

- The update rule for GD is given by:

$$\theta = \theta - \eta \nabla_\theta J(\theta)$$

  where θ represents the parameters of the model, η is the learning rate, and $\nabla_\theta J(\theta)$ is the gradient of the loss function J with respect to the parameters θ.

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a variation of the gradient descent algorithm that updates the model's parameters using only a single sample (or a small batch of samples) at each iteration. This approach reduces computation time significantly, making it more scalable and faster compared to batch gradient descent:

- The update rule for SGD is:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$

where θ is the parameter vector, η is the learning rate, and $\nabla_\theta J(\theta; x^{(i)}, y^{(i)})$ is the gradient of the loss function with respect to θ, computed at a single data point $(x^{(i)}, y^{(i)})$.

ADAM Optimizer

ADAM (Adaptive Moment Estimation) is an optimization algorithm that combines the advantages of two other extensions of stochastic gradient descent: Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). It is designed to adjust the learning rate for each parameter dynamically, leveraging the concepts of momentum and scaling of the gradients:

- The update rules for ADAM are defined as follows:

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t}, \\
\theta_{t+1} &= \theta_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}.
\end{aligned}$$

Here, m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients, respectively. g_t represents the gradient at time step t, β_1 and β_2 are exponential decay rates for these moment estimates, and η is the learning rate. The term ε is a small scalar used to prevent division by zero, typically around 10^{-9}.

2.8 Performance Metrics

Performance Metrics for Autoencoders

The effectiveness of an autoencoder, particularly in terms of its reconstruction capabilities, can be evaluated using various performance metrics. These metrics assess how closely the reconstructed outputs match the original inputs. These metrics are often also used in the loss function. Key metrics include:

1. **Mean Squared Error (MSE):** A common metric for evaluating the reconstruction quality of an autoencoder. It computes the average of the squares of

the differences between the original inputs and the reconstructed outputs:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2$$

where x_i is the original input and $\hat{x}_i$ is the reconstructed output. A lower MSE indicates better reconstruction accuracy.

2. **Mean Absolute Error (MAE):** This metric computes the average of the absolute differences between the original inputs and the reconstructed outputs, providing a more intuitive measure of average error:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|x_i - \hat{x}_i|$$

3. **Total Error (TE):** Total Error aggregates the absolute differences between the original inputs and their reconstructions, offering a cumulative measure of the model's reconstruction discrepancies:

$$\text{TE} = \sum_{i=1}^{n}|x_i - \hat{x}_i|$$

Performance for standard Neural networks

For binary classification tasks (such as anomaly detection), there are four relevant fractions: True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). An important metric is accuracy, which is the fraction of correct predictions to the total number of predictions. Other metrics are for example the recall which measures the fraction of actual positives that were correctly predicted, and the precision which measures the fraction of positives that are true. The combined metric, F_1 score may be better than accuracy in imbalanced classes, and is particularly sensitive to anomalies:

$$F_1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

2.9 Regularization Methods in Neural Networks

In neural network training, various methods are employed to prevent overfitting, enhance generalization, and optimize performance. Among these, dropout and pruning are notable techniques:

1. **Dropout:** Dropout is a regularization technique used to prevent overfitting in neural networks. During training, dropout randomly sets a fraction of the

input units to 0 at each update phase of the training, reducing the reliance on any specific set of neurons and thus encouraging a more distributed representation. The probability of setting a neuron to zero is typically a hyperparameter that can be adjusted. Mathematically, for a neuron output x, dropout is applied by multiplying x with a random variable d drawn from a Bernoulli distribution with probability p:

$$x' = x \cdot d$$

where $d = 1$ with probability p and $d = 0$ with probability $1 - p$.

2. **Pruning:** Pruning is a process of simplifying the network by removing weights or neurons that contribute little to the output. This can be done statically after the training is complete or dynamically during training. Pruning helps in reducing the model size and improving computational efficiency while aiming to maintain or even enhance the model's performance. The criterion for pruning could be based on the size of weights, gradients, or other aspects.

3. **Batch Normalization:** This method accelerates deep network training by standardizing the inputs of each layer. For a given layer, batch normalization adjusts and scales the activations, which helps in stabilizing the learning process and reducing the number of epochs needed for training. Mathematically, if x is the input to a layer, batch normalization transforms it as:

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

where μ_B and σ_B^2 are the mean and variance of the batch, respectively, and ε is a small constant for numerical stability. This transformation is followed by a scale and shift operation: $y = \gamma\hat{x} + \beta$, where γ and β are learnable parameters.

4. **Max normalization:** Unlike batch normalization, min-max normalization is a scaling technique that shifts and rescales the data to a specified range, typically [0, 1]. This method is particularly useful for ensuring that all inputs or features have a are within the same range, which can be great for convergence in neural network training. The transformation is given by:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where x' is the normalized output, and $\min(x)$ and $\max(x)$ are the minimum and maximum values of the feature of the data set, respectively. Unlike batch normalization, min-max normalization does not center the data around zero

however ensures that each feature contributes proportionately to the final result.

5. **Early Stopping:** This is a form of regularization used to prevent overfitting. During training, the model's performance is monitored on a validation set. Training is stopped when the performance on the validation set starts to degrade, i.e., when the validation error begins to increase, even if the training error continues to decrease. This method helps in selecting the model that is neither underfit nor overfit.

2.10 Multivariate Time-series Anomaly Detection via Graph Attention Network (MTAD-GAT)

MTAD-GAT [Zhao et al., 2020] builds two networks: a feature-oriented and a time-oriented GAT. The results from these are concatenated and passed into a GRU layer.

Afterwards the result is passed through a Forecasting-based Model and a Reconstruction model (Variational Autoencoder) in parallel. The errors are calculated and together produce a inference score. One advantage with MTAD-GAT is that the anomaly score is represented as a time-series, which allows for identifying when and how long an anomaly has occurred.

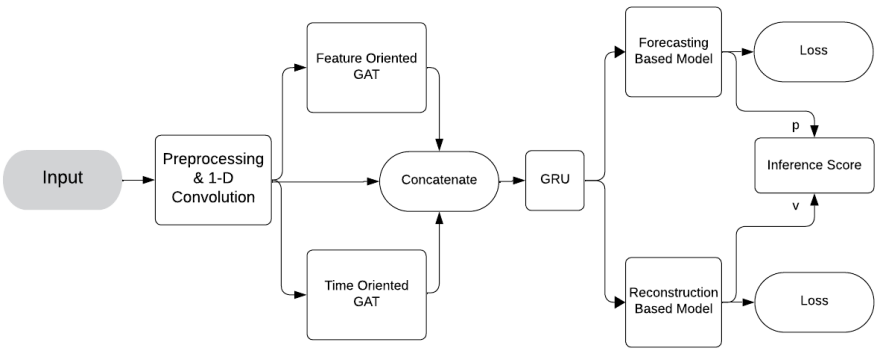


Figure 2.6 MTAD-GAT architecture

3

Methodology

3.1 Introduction

In this chapter, we describe the methods and their purposes used during this thesis. This includes the choice of models, preprocessing techniques, tests, metrics, and evaluation methods, along with the rationale behind most of these choices. The aim of this chapter is to make our results and discussion comprehensible.

3.2 Target Systems

Figure 3.1, although not the exact system studied in this work, illustrates how an operational system can look like. It includes sensors that feed data into units that distribute the data in a redundant manner. Such redundancy can potentially be a cause for synchronization errors. Another concern with the high degree of interconnectedness in the system is that local errors become global, which can drain the system resources and interfere with computations.

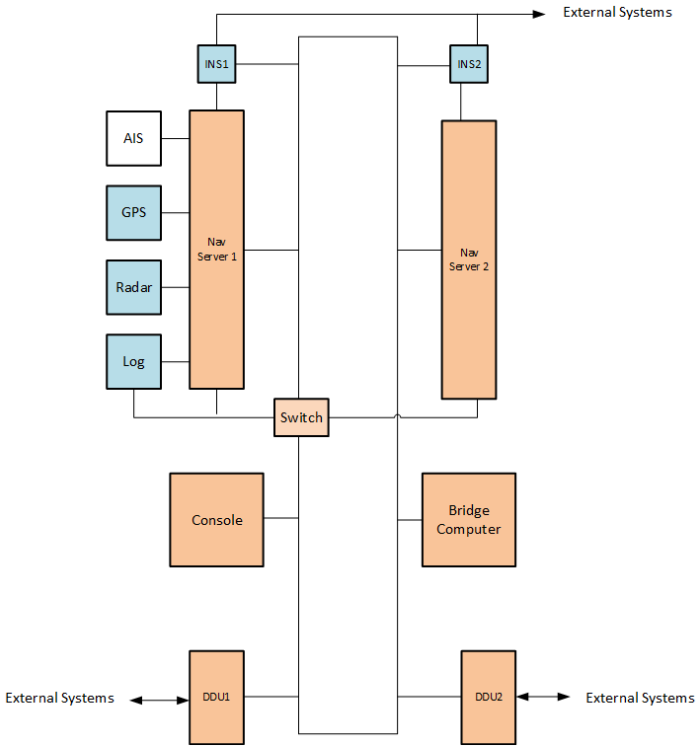


Figure 3.1 Navigation system example, illustrating the ring topology and redundancy.

Our network uses the Powerlink protocol [B&R, 2024] to provide real-time updates in a deterministic manner. The network uses broadcast to distribute data between nodes, which means that data will be sent to all nodes, as opposed to just the destination node.

3.3 Experimental Setup

Computing Environment: Our tests were conducted through the use of the WARA-Ops Dataportal, and its provided jupyterhub. This allows us to run our training and evaluation processes in a GPU-powered environment, utilizing Nvidia's CUDA interface for parallel computation.

Simulation Lab: We performed practical experiments in Saab Kockums' lab, where a number of input/output (IO) units and programmable logic controllers (PLC) communicate over a network. The network, which features 84 active nodes, uses a ring topology, meaning that each node is connected to two other nodes. Here

we carried out various simulated anomalies, such as delayed packages, cable rupture, killing nodes and introducing various degrees of package loss. We also collected extensive recordings of the normal state of the network, which is crucial for the model training. A computer was inserted in the ring network, and outputs emulated network traffic in which the errors are injected. The computer also received traffic, which was recorded in Wireshark. The process can be seen in Figure 3.2.

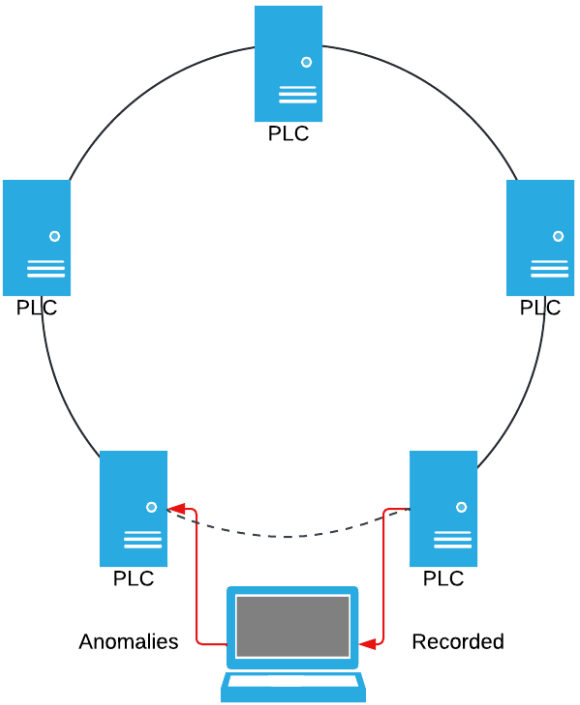


Figure 3.2 Anomaly injection and capturing in ring network.

3.4 Custom Graph Models

We have designed two simple models that we will later evaluate. The models are similar in architecture, only differing in the fact that one uses a GAT and one uses a GCN. In Figure 3.3 we can see the architecture. In the second box, we can either use a GAT, or a GCN, depending on the needs.

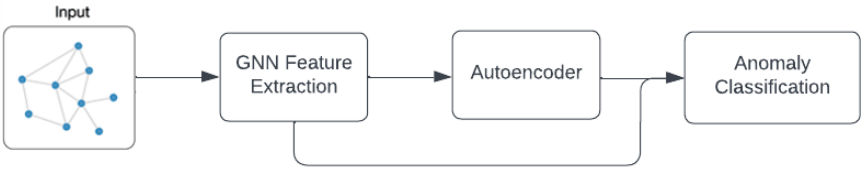


Figure 3.3 General graph based anomaly detection model.

The general procedure is as follows:

1. The model takes as input a graph, consisting of an adjacency matrix and a feature vector.

2. These arrays are fed into a GNN, either a GCN or GAT, which will extract a flattened list of intermediate features.

3. The new features are fed into an autoencoder that will learn the representation of these features, and try to reconstruct them as closely as possible.

4. Based on the loss between the original set of features fed into the autoencoder, and the reconstructed ones, we get an indication of the likelihood of an anomaly in the graph,

where input consists of accumulated data in our network over a time frame, as described in Section 3.9.

Why Simpler Models?

The purpose of such models is to compare how well a GCN and GAT would function on Saab's systems. GCN being a much simpler model should hypothetically provide less accurate results than a GAT. GAT is also trainable, and should therefore in theory perform better.

There is a possibility that for simpler systems, a GCN would provide more accurate results when dealing with less data. As autoencoder and anomaly detection models often require large amounts of data, a simpler model, not as dependent on learning would perhaps work better.

The models we designed for this purpose are therefore quite simple, the goal is not to design an effective architecture, but rather to compare GCN and GAT through a couple of tests, and evaluate overall whether such models are feasible for this application.

Training of the Models

The training for the GAT model was done separately to the training of the autoencoder. Meaning that we first learnt a proper representation of the graph, and then trained the autoencoder on that representation.

There was no training of the GCN in the GCN model, as it was only used for convolution. The convolution was simply done by averaging or summing the neighbors' features. The autoencoder was trained on the output of the GCN.

3.5 MTAD-GAT

MTAD-GAT can be used to analyze time series data from our network data, or the sensor data. For this model we need to transform our data into time series of different nodes. This can be done in several different ways explained in Section 3.10.

3.6 Standalone Autoencoder

A standalone autoencoder can be used to detect anomalies in single messages. Such autoencoder would take the raw message data as input as mentioned in Section 3.2.

This approach could be beneficial when trying to deduce exactly where in the system the anomaly happened, which the other models fail to incorporate, as they only give as a given time frame when an anomaly occurred.

3.7 Sensor Data

This dataset consists of various time series of sensor measurements. Sensors can for example mean salinity, pressure or temperature. The data is not completely synchronized, and therefore requires some processing to be used. The measurements are simulated in a test environment at Saab Kockums providing us with realistic data. This data can, after synchronization, be directly inserted into the MTAD-GAT model.

Datasets

This data consisted of one test recording from Saab Kockum's lab. The data included salinity and temperature, and a static value of one's was added for reference. Each metric is a time series, see Listing 3.1.

```
1 1707310867779 18.956205368041992  1
2 1707310868280 18.86391830444336   1
3 1707310868909 18.749839782714844  1
4 1707310869439 18.630634307861328  1
```

```
1707310870019 18.530654907226562  1
1707310870630 18.41273307800293   1
1707310871150 18.307626724243164  1
```

Listing 3.1 Sea water salinity time series. First column holds a time stamp and the second a sensor reading.

Processing of Sensor Measurements

The data is preprocessed by making interpolation, to achieve synchronization in regards of time. Other processing might include removing outliers, as sensors often measure natural phenomenon, extreme outliers should not be possible. In some cases, data might be averaged using for example a Gaussian bell, or a sliding window. This is to remove noise from data, which could negatively affect the output from our models.

3.8 Network Data

The datasets consist of network logs in the system, recorded in Wireshark [*Wireshark.org* 2024]. We are provided with network logs where messages contain the following:

1. **Source**
   The origin of the message.

2. **Destination**
   The recipient of the message.

3. **Protocol**
   The protocol used, typically Powerlink.

4. **Length**
   The length of the message.

5. **Information**
   Details about the message, including flags such as RD (ready), EA (exception acknowledge), EN (exception new), RS (request to send), and PR (priority).[1]

Listing 3.2 is an example of how the logs might look.

```
"No.","Time","Source","Destination","Protocol","Length","Info"
"1","0.000000000","B&RIndustria_4e:59:a1","EPLv2_SoC","POWERLINK
    ","60","240->255 SoC"
```

[1] For a comprehensive list of flags and their meanings, see `https://www.ethernet-powerlink.org/fileadmin/user_upload/Dokumente/Downloads/TECHNICAL_DOCUMENTS/EPSG_DS_301_V-1-3-0__4_.pdf`

```
3 "2","0.000246365","B&RIndustria_4e:59:a1","B&RIndustria_33
    :55:02","POWERLINK","1504","240->15 PReq [1480] F:RD=1,EA=0 V
    :0.0"
4 "3","0.000512378","B&RIndustria_33:55:02","EPLv2_PRes","POWERLINK
    ","1504","15->255 PRes [1480] F:RD=1,EN=0,RS=0,PR=0 V=0.0
    NMT_CS_OPERATIONAL"
```

Listing 3.2 Example of network traffic data in its pure form, as displayed by Wireshark.

Datasets

We recorded several different scenarios, as shown in Table 3.1. Three of them were standard training data, which can be used separately for training, or be concatenated to one big set. The other three scenarios are different test scenarios. The file size was on average 1 GB each.

Table 3.1 Datasets

Name	Nodes	Packages	Duration (s)	Description
train_1	48	747,334	102.8	Normal data
train_2	49	2,349,750	323.1	Normal data
train_3	51	1,022,652	133.4	Normal data
node_death_PLC	84	2,151,592	344.9	Killing of PLC
node_death_IO_PLC	52	364,852	49.0	Killing of IO and PLC
cable_rupture_IO	46	335,960	43.8	Disconnecting IO cable

- During **node_death_PLC** we first killed one PLC after 150,000 packages, then killed a second one after 420,000 packages. The first node was turned back on at 430,000 packages and the second one turned on at 1,000,000 packages.

- For **node_death_IO_PLC**, both a PLC and IO unit were turned off during an interval between 100,000 and 200,000 packages.

- During **cable_rupture_IO** a node was disconnected between 150,000 and 240,000 packages.

An example graph representation of the network data during one recording can be seen in Figure 3.4. The graph shows which nodes communicated with which nodes, and how much.

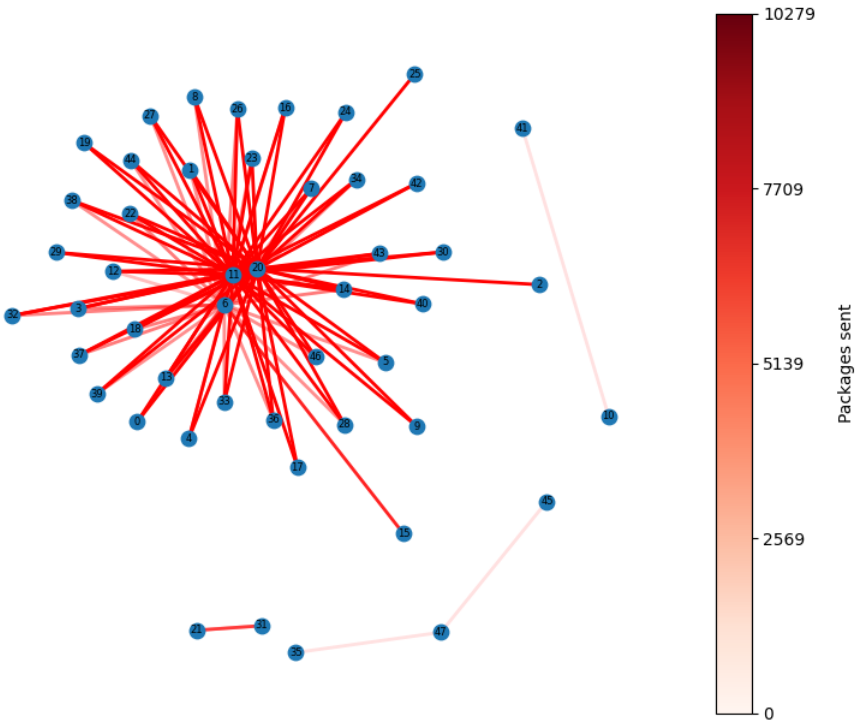


Figure 3.4 Graph representation of network data, where a node represents a source or destination and an edge shows the number of packages sent between them.

Across all the datasets, there are a total of 84 unique nodes. From our dataset we can see that a recording contains usually only $48-52$ nodes, with node_death_PLC containing 84. This can be explained by the fact that a "redundant system" is activated when the original system stops working properly, therefore the amount of nodes nearly doubles.

Looking at Figure 3.4 we can see that the nodes in the center are quite important for our graph. These are PLCs, while the rest of the nodes are standard IO nodes which are used for message relay.

3.9 Network Data Processing

Extracting Graphs for Own Models

As the data consists of one long recording, different methods can be used to extract data for training and inference for AI models. In order to be able to insert training

data into our own models we need the data to be in form of "graphs".

In all cases we essentially sum or take the average of all features for a node over the considered frame for a data points, implying that each data point, is a graph where the features of the nodes are for example the "total amount of messages received during that period".

Features and Graph Representation

A datapoint is represented as graph that includes the following:

- **Adjacency matrix:** An adjacency matrix describing the connections between nodes.
- **A list of nodes:** Each node has an array representation of features, according to either Feature set 1 or 2 below.

Features for nodes can be extracted in several ways, we mainly used two features sets in our experiments.

Feature set 1: The first approach was to have 4 features for each node, namely: the number of *packages received*, the number of *packages sent*, the average *length of received package* and the average *length of sent package*.

Feature set 2: As the feature set 1 does not specifically incorporate the relations between nodes, we can let our feature vector instead use the *sent* and *received* features, but for every other node in the graph specifically. Meaning that when having N nodes, each node would have $2N$ features. Now we will have graphs where each node has a feature vector that consists of number of packages sent and number of packages received for each other node in the graph.

The problems for this arise when we have a graph with a sparse number of connections, for example in the case of a "chain" graph (graph where each node has at most 2 edges), each node would only have 2 out of N features being nonzero. Another major problem is the scalability of this approach, as the number of features increases with $\mathcal{O}(N^2)$, whilst the previous approach gave us a complexity of $\mathcal{O}(N)$, this should however not pose a major problem with reasonable systems.

The systems we encountered had around 50 nodes, resulting in a total of 5000 features. With approximately 1000 data points in a dataset, this was manageable. However, in larger systems with more nodes and longer recordings, which are often necessary for training effective AI models, the time complexity should be taken into consideration.

Dividing the Recording into Graphs

As mentioned previously, in order to construct several graphs, our entire recording has to be cut into pieces, which we call intervals, and the information in those pieces aggregated by summing the number of packages sent between nodes, and taking the average of the message length. There are different ways to chop up the recording, described below.

Dividing Based on Time: One approach is to divide our recording based on time, meaning that we choose N intervals, which all should incorporate the same amount of time, however they can differ in the amount of data. This is because the message frequency in the network varies. Another factor in favor of this approach is the need for accurate time information of the anomaly occurrences.

Dividing Based on Amount: Another approach is to divide the total recording into parts by using the number of logs. This ensures that all data points contain the same amount of data, however they can represent different amounts of time, making it trickier to find out when an anomaly has occurred.

Number of Intervals: The number of intervals is based on a trade-off between data precision and data stability. By having more intervals, we can gain more insights. This approach could be bad when dealing with noise, as a too small interval can get overly affected by noise giving a unstable result. Having a too large interval, will not be affected by the noise as much, however it will obviously lead to data loss, which is not optimal.

Furthermore, the choice of interval should also depend on the data's regularity. Below is an example of a data vector and different choices for interval size (here we use the "divide based on amount approach").

Example of Interval Size Selection

In this example we will perform vector aggregation. Consider the data vector:

$$\mathbf{v} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

This data is regular, and the choice of interval size can significantly affect the summarized output. We will now analyze the effects of different window sizes.

Too Small Window Size: Choosing a too small window size, such as 2, might disrupt the regular pattern:

$$\text{Original: } \mathbf{v} \rightarrow \text{Windowed: } \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

This window size fails to capture the pattern accurately, leading to an inconsistent summary.

Too Large Window Size: Conversely, a too large window size, such as 5, could overly simplify the data:

$$\text{Original: } \mathbf{v} \rightarrow \text{Windowed: } \begin{bmatrix} 1 & 2 \end{bmatrix}$$

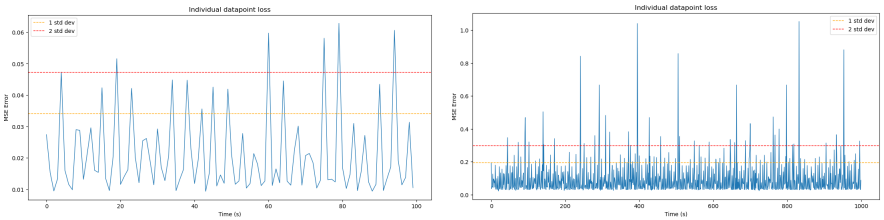
This window size results in a loss of detail that is also not optimal.

Optimal Window Size: An optimal window size for this data vector would be 3:

$$\text{Original: } \mathbf{v} \rightarrow \text{Windowed: } \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

This window size accurately maintains the regular occurrence of ones in the data.

Defining rules for this can be very tricky, also visualizing the data is not possible in many cases (especially when having 5000 features per data point, like in our case). Therefore, a trial and error approach is often needed for this parameter. Figure 3.5 are examples from training real data on one of the models (GAT + Autoencoder), and the corresponding intervals. Both show the training data errors trained with 50 epochs for our GAT model, and 100 for our autoencoder. The parameters were chosen after rigorous testing, as they provided us with the best results.



(a) Reconstruction loss for 100 intervals of standard training data

(b) Reconstruction loss for 1000 intervals of standard training data.

Figure 3.5

In this case we conclude that 100 intervals was a better choice, as 1000 intervals led to noise (as seen by the peaks) affecting the data too much, and making our training results worse.

Adjacency Matrix: The adjacency matrix for the graph is extracted from the network data by looping through a dataset file and creating two-directional edges between communicating nodes. This allows the GAT architecture to take advantage of pre-existing structure, which could hypothetically improve the training.

An alternative strategy could be to consider all nodes as interconnected, ignoring the existing network structure. This method would enable the Graph Attention

Network (GAT) to establish its own connections, potentially identifying relationships in areas where no direct links exist. For instance, it could link pressure and light sensors based on environmental conditions as deeper water exhibits lower light levels and higher pressure, even though these sensors are not directly connected through wires. This approach might reveal meaningful correlations between different types of measurements by learning from the data itself, despite the absence of direct physical links between the sensors.

Achieving Graphs of Equal Size: All recordings have different number of nodes. In order to test on different data than we train on, we need to ensure the graphs have equally many nodes. This is achieved by iterating through all the datasets and extracting their nodes, which are combined into one large list. This universal list is then used in all the graph extractions, meaning that some datasets will produce sparser adjacency matrices and features when some nodes are not used. Based on the datasets in Table 3.1, our standardized graphs will all have adjacency matrices of size 84×84, as we have 84 unique active nodes in our system.

3.10 Extracting Time Series Data from Network Data

Starting with a set of raw messages. We can create an array consisting of N elements, where N is the number of nodes in our network. We can now create such arrays for all timestamps in our recording. If a node has received or sent a package at a given timestamp, we can replace the node-corresponding element in the array with a value.

The value can be chosen in different ways. We chose to use the size of the package that was sent/received. The value could either be negative or positive, with positive values symbolizing values that were sent and negative values symbolizing values that were received. Here is a short example for clarity where we have a simplified recording.

Example

The raw network data is tabulated as follows:

TimeStamp	Source	Destination	PackageSize
1	1	2	8
2	2	1	3
3	0	1	3
4	1	2	5
4	2	0	4

Transformation to Time Series Format: To transform this data into a time series format, we create an array for each timestamp, corresponding to each node in the network. The length of each array is equal to the number of nodes, N. The position in the array corresponds to a specific node, and the value at each position is calculated as the net sum of packages sent (positive value) and received (negative value) by that node at the timestamp.

Here is the step-by-step transformation for each timestamp:

- **Timestamp 1:**
  Node 1 sends 8 to Node 2 $\Rightarrow [0, 8, -8]$
- **Timestamp 2:**
  Node 2 sends 3 to Node 1 $\Rightarrow [0, -3, 3]$
- **Timestamp 3:**
  Node 0 sends 3 to Node 1 $\Rightarrow [3, -3, 0]$
- **Timestamp 4:**
  Node 1 sends 5 to Node 2 and Node 2 sends 4 to Node 0 simultaneously $\Rightarrow [-4, 5, -1]$

Resulting Time Series Data: The resulting arrays for each timestamp represent the network activity as a time series, where each node's net traffic is captured. Below is the resulting time series:

```
[0, 8, -8]
[0, -3, 3]
[3, -3, 0]
[-4, 5, -1]
```

Sparsity of Data

Such dataset can become very sparse as each timestamp only gives information about one message being sent. When having more nodes, for example 81 as in our graph we might end up with too many zeroes. This might be hard for a machine learning model to process. There are several different ways we can come around it, as described below.

Smoothing out Time Series Data

In order to make the data more manageable for models such a MTAD-GAT, the data can be smoothed out so that signals "overlap". These regularizations can potentially help deep learning networks get better at spotting patterns in data by keeping signals important for a longer time. For example, the feature GAT used in MTAD-GAT looks at data one moment at a time. By stretching out how long we consider a signal active, we can give the network more context to work with. This means that even

after a bit of time has passed, the network still treats the signal as important. Below, we can see how we can adjust training data to make signals last longer, helping the network understand the data better. We can do it in 3 different ways.

Data Aggregation: Data can be aggregated using a sliding window. The deciding parameter is the number of messages aggregated. A higher number of aggregated messages provides cleaner but less specific data. Aggregating reduces the number of data points. The aggregation function is defined as:

$$\text{Aggregate}(x_i) = \sum_{j=0}^{n-1} x_{i+j} \tag{3.1}$$

The average aggregation is given by:

$$\text{Average Aggregate}(x_i) = \frac{1}{n} \sum_{j=0}^{n-1} x_{i+j} \tag{3.2}$$

Gaussian Bell: The Gaussian bell approach involves convolving the data with a Gaussian function, often referred to as a Gaussian filter. This method weights the data points near the target point more heavily than those farther away, with the weights following the characteristic bell-shaped curve of the Gaussian distribution. The parameter σ controls the standard deviation of the Gaussian bell, determining the breadth of the bell curve and thus the smoothness level of the output. A higher σ value results in more substantial smoothing, as it increases the effective "radius" of influence of each data point. The Gaussian function in one dimension is given by:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{3.3}$$

To apply the Gaussian filter to data $f(x)$, convolve the data with the Gaussian function:

$$(f * G)(x) = \int_{-\infty}^{\infty} f(t)G(x-t)\,dt = \int_{-\infty}^{\infty} f(t)\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-t)^2}{2\sigma^2}\right) dt \tag{3.4}$$

Moving Average: The moving average technique smooths data by replacing each data point with the average of neighboring points within a specified window size. This method is similar to sliding a window across the data; at each point, it calculates the mean of the points within the window and updates that value to the central point. Larger windows yield smoother data but can obscure quick changes, while smaller windows preserve more detail but smooth less. The formula for the moving average at point x_i is:

$$\mathrm{MA}(x_i) = \frac{1}{2k+1} \sum_{j=-k}^{k} x_{i+j} \tag{3.5}$$

Figure 3.6 is an example of how different data smoothing methods work on example data.

**

Figure 3.6 Different plots of time series data after and before smoothing. Original network data (the first row) is processed in different ways using different smoothing methods, and the outputs can be seen in the following three rows.

The benefit of smoothing data can be illustrated with the example of two data vectors. Consider two data vectors, where the second vector is a delayed version of the first:

$$\mathbf{v}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$
$$\mathbf{v}_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

A feature-based Graph Attention Network (GAT) might fail to incorporate the delay effectively, representing the relationship as follows:

$$\begin{array}{cccccccccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ | & | & | & | & | & | & | & | & | & | \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

Applying regularization (smoothing) to these vectors can help in capturing the delay, yielding:

$$\begin{array}{cccccccccc} 1.0 & 0.5 & 0.0 & 0.5 & 1.0 & 0.5 & 0.0 & 0.5 & 1.0 & 0.5 \\ | & | & | & | & | & | & | & | & | & | \\ 0.5 & 1.0 & 0.5 & 0.0 & 0.5 & 1.0 & 0.5 & 0.0 & 0.5 & 1.0 \end{array}$$

This structure allows networks to establish connections between non-adjacent data points, demonstrating temporal relationships not immediately obvious, potentially enhancing the network's learning and prediction capabilities. Figure 3.7 is a plot illustrating the timeseries.

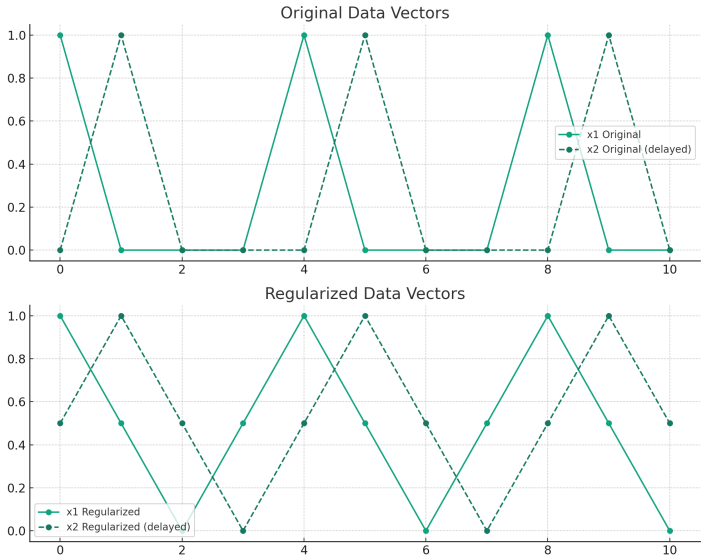


Figure 3.7 A simple example of how data can be smoothed out in order to be more manageable for neural networks.

We need to be careful with such an approach, as while it can improve the feature-based GAT in the MTAD-GAT model, it may potentially deteriorate the performance of the time-based GAT. What smoothing, and how much we smooth, is decided by trial and error that is done during training of the model.

Another problem is that regularization might lead to data loss, if we have two opposite signals close to each other (for example a node receiving and sending a signal at the same time), then some regularizations such as Gaussian curve might cancel those signals out). This is again tested by trial and error.

3.11 Extracting Raw Data for Autoencoder

The network data can also be left in its original form, meaning that we do not extract any graph from it, but use the full messages as input. The only processing we do to allow for training is to convert hexadecimal form into decimal form, and to pad each line with zeros to achieve equal lengths for all messages. The input then

consists of a vector containing bytes, which is the encoding for a message.

An example of the message is:

```
"1","0.000000000","B&RIndustria_4e:59:a1","EPLv2_SoC","POWERLINK
    ","60","240->255 SoC"
```

A hexadecimal encoding of this message would then be:

```
22 30 2e 30 30 30 30 30 30 30 30 22 20 2c 22 20 42 20 26 20
52 49 6e 64 75 73 74 72 69 61 5f 34 65 20 3a 35 39 3a 20 61 31
20 22 20 2c 22 20 45 50 4c 76 32 5f 53 6f 43 20 22 20 2c 22 20
50 4f 57 45 52 4c 49 4e 4b 20 22 20 2c 22 36 30 22 20 2c 22 32
34 30 20 2d 3e 32 35 35 20 53 6f 43 20 22
```

This can be later converted to decimal values, and used as input. This could be useful for standalone autoencoders or LSTMs analyzing series of raw data.

3.12 Anomaly Types

Due to the general nature of our goal model, which should be able to detect arbitrary anomalies in various systems in a submarine, we did not have a predefined set of errors that we want to evaluate. However, to test and compare the models we had to pick a few realistic anomalies to focus on.

The anomalies we decided on testing were the following:

1. **Increased network traffic**
   In this experiment we increase certain node package outputs. We try to vary the load intensity and number of affected nodes. The aim of this error is to simulate security breaches (DDOS) or operational anomalies. This also aims to be a sensitivity measure for how sensitive different models are to "different" traffic, as it is easily measurable.

2. **Removal/shut down of nodes**
   Here we try to make certain nodes stop sending and receiving packages. The aim of this error is to simulate wear and tear anomalies as well as operational anomalies. Such anomalies could in practical situations take form of for example faulty cables or broken receivers.

3. **Faulty packages**
   We try to send nonsense, or at least unusual, data. The data can both be completely meaningless simulating faulty signals, or the data can be malicious, which could indicate a cybersecurity breach.

These errors can be both simulated and induced in the data after its recording, or can be generated during the recording.

Qualitative vs Quantitative anomalies

Quantitative Anomalies: Anomalies that affect the whole system behavior over longer period of time can be called quantitative anomalies, meaning that the entire system will suddenly start acting differently. In this category we have the "increased network traffic" anomaly as well as "Removal/shut down of nodes.

Qualitative Anomalies: Another type of anomalies would be anomalies that do not necessarily change the entire systems behavior, but still count as an anomaly. It could for example be unrecognized code being injected through a message, or unauthorized access. The "Faulty packages" fall under this category.

3.13 Generating Anomalies for our Own Models

These anomalies are generated by changing the features of the extracted graphs that we use for our own models. Examples of how such changes can look like can be seen in Figure 3.8 and Figure 3.9, where feature values are increased and decreased, respectively.

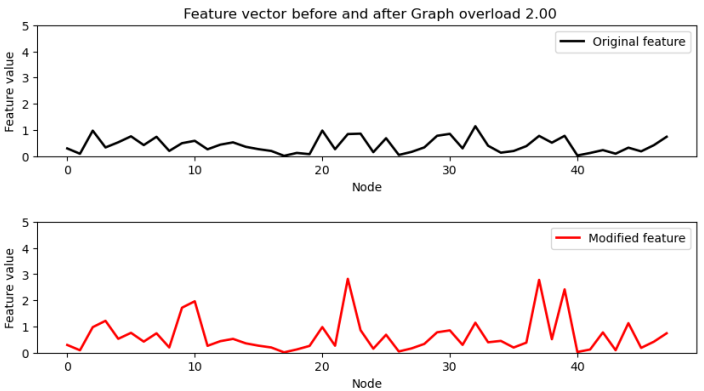


Figure 3.8 Upper figure shows an example feature for each node, lower figure shows the same feature after anomalies are generated using *graph overload* (factor 2), when changing 10 nodes with standard deviation of 3. In this case there are 48 nodes, and each node has only one feature.

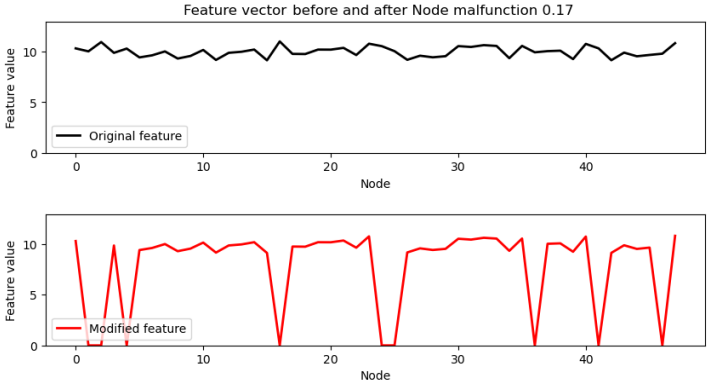


Figure 3.9 Upper figure shows an example feature for each node, lower figure shows the same feature after anomalies are generated using *node malfunction* (factor 0.17), when changing 17% of the nodes. In this case there are 48 nodes, and each node has only one feature.

Simulation of Network Traffic Overload

In order to assess the impact of the increasing load condition on network model performance, a special simulation function was developed: *graph_overload*. This feature is very central in simulating, through artificial creation, a state of traffic overload in a subset of nodes so as to give an outlook on the behavior of the network under stress-causing conditions. Underlying this simulation is the objective to show how our model reacts under abnormal traffic volume that, in real life, becomes an important scenario when evaluating the robustness of the network system.

Functionality Overview The function *graph_overload* emulates increased traffic on the network by artificially increasing the values of 'packages received' and 'packages sent' metrics for a certain subset of selected nodes in a graph. This is achieved by updating the features of these nodes based on a specified overload factor, as detailed in Algorithm 1. It is important to test network response when there is an unexpected data flow peak, which in this case might represent multiple real-world events, for example, cyber-attacks, system faults, or irregular user activities.

Algorithm 1 Update Features for a Subset of Nodes

Input: *datapoint* (Data point representing the network, see Section 3.9)
Input: *overloadFactor* (Factor by which to update features)
Output: Modified *datapoint* with updated features
numNodes ← length of $datapoint.nodes$
selectedNodes ← randomSelectionOfNodes(*numNodes*)
for each *node* in *selectedNodes* **do**
  for each $feature$ in $node.features$ **do**
    $updateFactor \leftarrow$ generateUpdateFactor($overloadFactor$)
    $feature \leftarrow feature + feature \times updateFactor$
  end for
end for

We can now create a dataset with increasing overloads, using Algorithm 2

Algorithm 2 Apply Graph Overload with Variable Step Sizes

Input: *datapoint* (Data point representing the network)
Input: *maxFactor* (Maximum overload factor)
Input: *stepSize* (Increment size for each step)
Output: List of *datapoints* each with varied overload factors
results ← empty list
for $factor \leftarrow 0$ to *maxFactor* step *stepSize* **do**
  $overloadedDatapoint \leftarrow$ graph_overload($datapoint, factor$)
  Append *overloadedDatapoint* to *results*
end for
return *results*

Simulation of Node Malfunction

An integral part of assessing network robustness is understanding how the network behaves under scenarios of node malfunctions. To simulate such conditions, the *node_malfunction* function was developed. This function is designed to artificially induce malfunctions in a subset of nodes within the network, thereby enabling an analysis of the network's fault tolerance and recovery mechanisms. The simulation aims to mimic the effects of node failures or degraded performance, which are common in real-world networks due to various factors like hardware failures, software bugs, or external attacks. The process of simulating node malfunctions is detailed in Algorithm 3.

Algorithm 3 Simulate Node Malfunction Based on Malfunction Rate and Factor

Input: *datapoint* (Data point representing the network)
Input: *malfunctionRate* (Rate at which nodes malfunction, between 0 and 1)
Output: Modified *datapoint* with malfunctioning nodes
numNodes ← length of *datapoint.nodes*
numMalfunctionNodes ← round(*numNodes* × *malfunctionRate*)
malfunctionNodes ← selectRandomNodes(*numNodes*, *numMalfunctionNodes*)
for each *node* in *malfunctionNodes* **do**
  for each *feature* in *node.features* **do**
    feature ← *feature* × *overloadFactor*
  end for
end for
return *datapoint* with updated node features

To extend this functionality, the *Batch Node Malfunction Simulation* function simulates node malfunctions at different rates, allowing for a comprehensive analysis across a range of malfunction scenarios. This extended simulation is described in Algorithm 4.

Algorithm 4 Batch Node Malfunction Simulation

Input: *datapoint* (Data point representing the network)
Output: List of *datapoints* each with nodes malfunctioned at different rates
results ← empty list
for *i* ← 0 to length of *datapoint.features* − 1 **do**
  malfunctionRate ← *i*/length of *datapoint.features*
  modifiedDatapoint ← node_malfunction(*datapoint*, *malfunctionRate*, *overloadFactor*)
  Append *modifiedDatapoint* to *results*
end for
return *results*

Simulation of Faulty Packages and Wrong Messages on Raw Data

Faulty packages and wrong messages can indicate a cyber-attack or faulty systems. The hexadecimal encoding of a random message is extracted, and then some byte values are changed. Depending on the requirements, we can change the message content or source/destination, introducing a faulty source destination, or sending messages to wrong nodes. Here, we used a similar approach to the node malfunction, but instead changed individual bytes with various intensities. The procedure for simulating package malfunctions is detailed in Algorithm 5.

Algorithm 5 Simplified Package Malfunction Simulation

Input: *packageLine* (Network traffic data)
Input: $malfunctionRate$ (Percentage of data to modify)
Input: $adjustmentRange$ (Possible adjustment values)
Output: Updated *packageLine*
$numBytes \leftarrow$ length of *packageLine*
$malfunctionCount \leftarrow \text{round}(numBytes \times malfunctionRate)$
$indicesToModify \leftarrow$ random sample of $malfunctionCount$ indices from $numBytes$
for i in $indicesToModify$ **do**
    $packageLine[i] \leftarrow packageLine[i] +$ random choice from $adjustmentRange$
end for
return *packageLine*

3.14 Evaluation Metrics

Custom Model Comparison Metrics

We use a number of metrics to evaluate our models. For the custom models where we introduce generated errors, we want to compare how sensitive the models are. This can be done by measuring the loss as a result of the size of the errors, or how big portion of the graph is being altered. Since the training loss can vary between the models, comparing the MSE values straight off, may not be the best sensitivity metric, especially when using different variations of datasets. Instead, we introduce a threshold k, which represents a certain level above the MSE we get on unchanged data. When running multiple iterations of our tests, we can then take note of the ratio of tests that produced MSE values above the model-dataset specific threshold. A higher ratio would then indicate a more sensitive model. This can be seen in Figure 3.10.

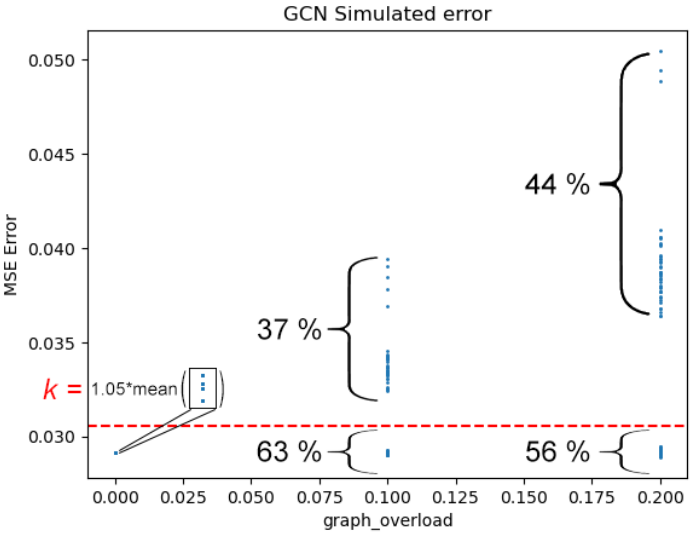


Figure 3.10 A visualization of the k-metric used to compare GAT to GCN. High ratios above the threshold indicate a sensitive model. k is taken as 5 % above the mean of the unchanged value (0 graph overload) MSE.

Correlation Another value we can look at is the correlation coefficient for the errors, which represents the stability of our model. A higher correlation might imply that the model manages to learn something.

Classification Metrics

When using labeled test data, we can calculate the F_1 scores as described in Section 2.8. We could also try to measure the responsivity, i.e., how fast the models respond to an anomaly. Another interesting metric could be to measure how much data is required to correctly identify an anomaly.

3.15 Finding a Threshold for Anomaly Classification

Finding a correct threshold can be challenging. The typical approach would be to use the error from train data and somehow base a threshold on it. However, a problem occurs when the test data overall looks different, and therefore produces a higher mean error. In such cases anomalies can still be detected by observing the error, and how it varies throughout the time series. It essentially boils down to detecting anomalies in the time series consisting of errors. For this approach we have the following alternatives:

Mean and STD Based on Test Data

We can calculate the mean and standard deviation of the test data, and exclude outliers. This is done by setting percentiles above and below which the data is excluded and then, the recalculating the mean and STD, see Algorithm 6. This helps us to avoid incorporating the anomaly data in the threshold, however the percentiles should be set carefully, and will depend on how big of a portion of the dataset the anomaly data takes up.

Algorithm 6 Calculate Threshold

Input: data (a list of numerical values)
Input: exclusion_percentile (percentile for exclusion of extreme values)
Output: A tuple (mean_val, std_val) or None
if length of data == 0 **then**
    return None ▷ Return None if no data is available
end if
$lower\_bound \leftarrow$ percentile(data, exclusion_percentile)
$upper\_bound \leftarrow$ percentile(data, 100 - exclusion_percentile)
$filtered\_data \leftarrow$ empty list
for each value x in data **do**
    if $x \geq lower\_bound$ and $x \leq upper\_bound$ **then**
        append x to filtered_data
    end if
end for
if filtered_data is empty **then**
    return None ▷ Return None if no data remains after filtering
end if
$mean\_val \leftarrow$ mean(filtered_data)
$std\_val \leftarrow$ standard deviation(filtered_data)
return $(mean\_val, std\_val)$

By using this method, we can now create several different thresholds with different magnitudes, seen in Algorithm 7.
Having several different thresholds provides more insights about the data. For example, magnitudes can be: $1, 2, 10$ or $2, 4, 8$, depending on the data used.

Threshold Based on Maximizing the F_1 Score or Accuracy

Another way to find a threshold is by optimizing the F_1 score or accuracy. This gives us the optimal threshold. It is important to mention that this provides us the theoretical optimal threshold, and such a result would not be realistic in real life uses of such AI models, in such cases different methods should be considered.

We use a binary search to find the optimal threshold, see Algorithm 8. This method

Algorithm 7 Generate More Thresholds

Input: threshold_magnitude (multiplier for standard deviation)
Input: data (a list of numerical values)
Output: threshold (calculated threshold value)
$(mean, std) \leftarrow$ Calculate_Statistics_Excluding_Extremal_Values(data)
if *mean* is None or *std* is None **then**
    return None ▷ Return None if statistics cannot be calculated
end if
$threshold \leftarrow mean + std \times threshold\_magnitude$
return *threshold*

is not optimal because it can miss local minima, as it assumes that the F_1 score will change linearly according to the Y-axis, which is not the case. However, it gives us a solid chance of finding a reasonable threshold.

The code in Algorithm 8 maximizes the F_1 score; in order to maximize accuracy, we simply change the F_1 score function to one that calculates accuracy.In this case we might create more thresholds from the initial threshold by for example multiplying it with different factors.

Algorithm 8 Optimal Threshold Using Binary Search

Input: *labels* (ground truth binary labels)
Input: *anomaly_scores* (anomaly scores from a model)
Input: $tolerance$ (minimum difference to stop the search, default 1×10^{-4})
Output: $(best\_threshold, best\_f1)$
$lower\_bound \leftarrow \min(anomaly\_scores)$
$upper\_bound \leftarrow \max(anomaly\_scores)$
$best\_f1 \leftarrow 0$
$best\_threshold \leftarrow lower\_bound$
while $upper\_bound - lower\_bound > tolerance$ **do**
    $mid\_point \leftarrow (lower\_bound + upper\_bound)/2$
    predictions_mid $\leftarrow$ list of 1 if score $\geq$ *mid_point* else 0 for each score in *anomaly_scores*
    $f1\_mid \leftarrow \text{f1\_score}(labels, predictions\_mid)$
    $slightly\_higher \leftarrow mid\_point + tolerance$
    predictions_higher $\leftarrow$ list of 1 if score $\geq$ *slightly_higher* else 0 for each score in *anomaly_scores*
    $f1\_higher \leftarrow \text{f1\_score}(labels, predictions\_higher)$
    if $f1\_mid > best\_f1$ **then**
        $best\_f1 \leftarrow f1\_mid$
        $best\_threshold \leftarrow mid\_point$
    end if
    if $f1\_higher > best\_f1$ **then**
        $best\_f1 \leftarrow f1\_higher$
        $best\_threshold \leftarrow slightly\_higher$
    end if
    if $f1\_higher > f1\_mid$ **then**
        $lower\_bound \leftarrow mid\_point$
    else
        $upper\_bound \leftarrow mid\_point$
    end if
end while
return $(best\_threshold, best\_f1)$

3.16 Underlying System and its Impact on Anomaly Detection

The underlying system can react in different ways, and even though an anomaly officially is said to occur during a certain period of time, the system can still be affected by the anomaly sometime after, for example when trying to reset itself.

This can affect all sorts of metrics, for example when deciding a threshold based on a F_1 score. Therefore, in this case it is hard to find quantitative metrics to evaluate our model, and we often have to resort to qualitative comparison instead.

In the case of finding a threshold, we often have to adapt the values of the true anomalies (but only for finding the threshold) in our data in order to find a reasonable threshold, which is then evaluated visually.

3.17 Experiments Performed

Comparing GCN to GAT

Our first Experiment will consist of testing whether a CGN or GAT performs better, using our own models. Graphs will be created in different ways described in Section 3.9. We will use the functions *node_malfunction* and *graph_overload* described in Section 3.13 And the results will be evaluated using the custom k-metric described in 3.14.

Furthermore, variations of these base models will be evaluated using both feature set 1 and feature set 2. We will also test whether connecting all nodes, or using the underlying systems connections will provide better result.

We will also test these models on the recorded test data set, and find thresholds using the methods described in 3.15.

Testing MTAD-GAT

Our first test aims to find out if the algorithm works correctly, and will make us accustomed to its workflow, we will therefore use the simpler sensor dataset. We will also try a few different parameter settings to see how they affect the result. Next, we will run the algorithm on transformed network data, and see how it compares to our custom graph models, which are trained on the same original dataset. The three simulated test cases will be evaluated, where we will try to find an optimal threshold method using the anomaly scores in comparison to our true anomaly labels.

Testing Standalone Autoencoder

In this test we will train an autoencoder on (normalized) hexadecimal encoding of raw data. Later we will use the *node_malfunction* variant for raw data and generate errors in our encodings with increased intensities.

This test aims to evaluate the performance of autoencoders in detecting anomalies in network data.

4

Results

4.1 Algorithm Selections

Along with our custom GAT and reference GCN model, we chose the MTAD-GAT as an existing algorithm to compare with. The code base [Harstad and Kvaale, 2024] with the PyTorch implementation worked out of the box, apart from minor adjustments to align with our own data, mainly regarding its dimensions. This also meant the data had to be formatted according to Section 3.10.

4.2 Preliminary Tests on Simulated Data

GAT/GCN and Autoencoder Models

The experiments simulating node malfunction and graph overload as described in Section 3.13 were tested on both the GAT and GCN variation of our own models. The evaluation metric from Section 3.14 was used where the sensitivity of the models was tested. In these tests we ignored the aspect of time, and only used the "divide by amount" method for dividing the recording into intervals. This because it did not matter much for these tests.

Explanation of Model Names

1. **GAT:** This is a GAT using Feature set 1 (as described in Section 3.2), and adjacency matrix only consisting of underlying structure.
2. **GAT-Ones:** This is a GAT using Feature set 1, and an adjacency matrix consisting of all ones, meaning all nodes are interconnected (again described in Section 3.2).
3. **GAT-Big:** This is a GAT using Feature set 2 and adjacency matrix only consisting of underlying structure.
4. **GAT-BigOnes:** This is a GAT using Feature set 2, and an adjacency matrix consisting of all ones.

5. **GCN:** This is a GCN using Feature set 1.
6. **GCN-Ones:** This is a GAT using Feature set 1, and an adjacency matrix consisting of all ones.
7. **GCN-Big:** This is a GCN using Feature set 2, and adjacency matrix only consisting of underlying structure.
8. **GCN-BigOnes:** This is a GCN using Feature set 2, and an adjacency matrix consisting of all ones.

All graph models are connected to an autoencoder which is trained to reconstruct the input data.

Graph Overload

Below are the tables with the results when testing with an overload factor equal to 0.0-1.0, and affecting 15% of the nodes in the network. The highlighted models are the ones that performed best when taking both correlation and the ratios into consideration.

Table 4.1 Results when testing different models using the Graph overload algorithm

	Iteration ratio > k=105% MSE of training data										
Overload factor	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	k	Corr.
GAT	0.70	0.87	0.88	0.91	0.91	0.93	0.98	1.0	1.0	0.0088	0.5167
GAT-Ones	0.43	0.51	0.58	0.57	0.8	0.75	0.74	0.82	0.84	0.0129	0.2975
GAT-Big	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.0645	0.3000
GAT-BigOnes	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.2143**	**0.8523**
GCN	0.22	0.40	0.31	0.39	0.41	0.41	0.40	0.38	0.54	0.1476	0.3542
GCN-Ones	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.0298	0.5343
GCN-Big	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0002	-
GCN-BigOnes	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.1927	0.7961

Because of the similarity of GAT-Big, Gat-BigOnes, GCN-Ones and GCN-BigOnes we decided to lower the overload factor to $0 - 0.02$, in order to compare those models more accurately, in this case also affecting 15% of the nodes in the network.

Table 4.2 Results when testing different models using the Graph overload algorithm

	Iteration ratio > k=105% MSE of training data										
Overload factor	0.002	0.004	0.006	0.008	0.01	0.012	0.014	0.016	0.018	k	Corr.
GAT	0.00	0.00	0.00	0.00	0.00	0.01	0.09	0.07	0.12	0.0166	0.5638
GAT-Ones	0.67	0.61	0.63	0.65	0.73	0.69	0.76	0.73	0.71	0.0047	0.3542
GAT-Big	0.00	0.00	0.00	0.00	0.19	0.50	0.78	0.93	0.99	0.1118	0.8719
GAT-BigOnes	0.35	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	0.0419	0.7054
GCN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.0351	0.3709
GCN-Ones	0.73	0.82	0.96	0.98	0.99	0.97	0.98	1.00	0.99	0.0436	0.4795
GCN-Big	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0005	-
GCN-BigOnes	**0.59**	**0.99**	**0.99**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.3540**	**0.7823**

The amount of nodes, 15% in this scenario was chosen as it provided the most variation between results for different models. However the amount of nodes affected did not matter much, as long as it was larger than 10% and smaller than 50%.

Node Malfunction

We tested the reaction of the models to nodes "shutting" down, where malfunction rate refers to how many nodes were shut down, 0.5 meaning 50% were shut down.

Table 4.3 Node malfunction

	Iteration ratio > k=105% MSE of training data												
Malfunction rate	0.083	0.167	0.25	0.333	0.417	0.5	0.583	0.667	0.75	0.833	0.917	k	Corr.
GAT	0.00	0.02	0.05	0.05	0.04	0.01	0.00	0.03	0.01	0.01	0.00	0.0163	-0.6947
GAT-Ones	0.33	0.43	0.61	0.77	0.89	0.94	0.98	0.99	1.00	1.00	1.00	0.0016	0.3744
GAT-Big	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.8389	0.2552
GAT-BigOnes	0.38	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.1243	0.6529
GCN	0.01	0.03	0.04	0.09	0.09	0.02	0.08	0.08	0.05	0.04	0.00	0.0798	-0.4333
GCN-Ones	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.0207	0.9059
GCN-Big	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0004	-
GCN-BigOnes	0.99	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.2973**	**0.9588**

Plots

The plots of the results in Table 4.1–4.3 in are shown in Figure A1–A6 in the appendix. The format of the plots is also described in Section 3.14. In these plots, the x-axis represents either the fraction of nodes turned off (in the case of node malfunction) or the overload factor (in the case of graph overload). The y-axis represents the reconstruction error. Each individual point is one simulation, as described in Section 3.13

4.3 GCN and GAT on Simulated Errors

All the below models are trained with 200 autoencoder epochs, and the GAT models are trained with 100 epochs. These parameters are chosen after testing various numbers, and provided the best results overall. The results from each class of models use the same trained model, both for GCN and GAT. We use the Big feature set, as its results in our preliminary tests were slightly better, meaning each node has a feature vector two times the number of nodes; the sizes of each dataset and their graphs can be seen in Table 3.1.

Furthermore, we divide the datasets based on time, each with 100 intervals, which is high enough to get variation, but low enough to reduce noise. The thresholds used are based on Algorithm 6. The levels we chose are 1, 2 and 10 standard deviations from the mean of non-anomalous data, with an exclusion percentile of 10. The 1 and 2 levels represent values close to the mean, and the 10 level an extreme value.

In both models our autoencoder splits the original training data into 80% training data and 20% validation data. The GAT batch size is 32, providing a balance between stability and training efficiency.

Table 4.4 contains the F_1 scores of the models on each dataset, where the classification threshold is 1, 2 or 10 standard deviations from the mean.

Table 4.4 F_1 scores

	Node death PLC			Node death IO & PLC			Cable rupture IO		
Std.dev.	1	2	10	1	2	10	1	2	10
GAT-zeros	0.5176	0.5205	**0.4706**	**0.6420**	0.0	0.0	0.2593	0.1778	0.0571
GAT-ones	0.6593	0.6591	**0.4706**	**0.6420**	0.0	0.0	**0.2963**	0.1364	0.0
GCN-zeros	0.2778	0.2857	0.2462	0.2727	**0.2727**	**0.0606**	0.1463	0.1463	**0.1081**
GCN-ones	**0.7021**	**0.6818**	**0.4706**	**0.6420**	0.0	0.0	0.2759	**0.2222**	0.0

Plots

All results can be found in Figure A7–A18 in the appendix. Some examples are shown in Figure 4.1–4.3. The figures are split into two different plots.

In the top plot, the x-axis represents time, while the y-axis represents the reconstruction error. The blue line indicates the reconstruction error from our model. The red area highlights the ground truth for the anomaly, showing when the anomaly occurred. We also have three lines, which are the threshold based on the mean and standard deviation of non anomalous data. In the bottom plot, we observe when the reconstruction error exceeds a specific threshold.

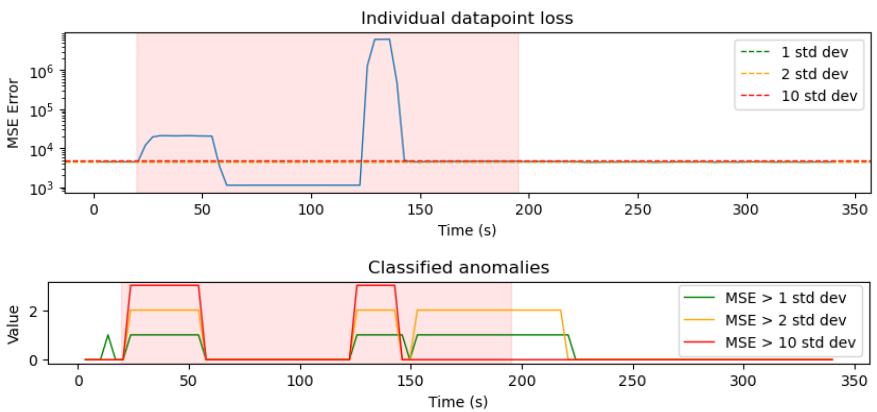


Figure 4.1 The reconstruction loss when using the GAT-ones model on the Node death PLC data

.

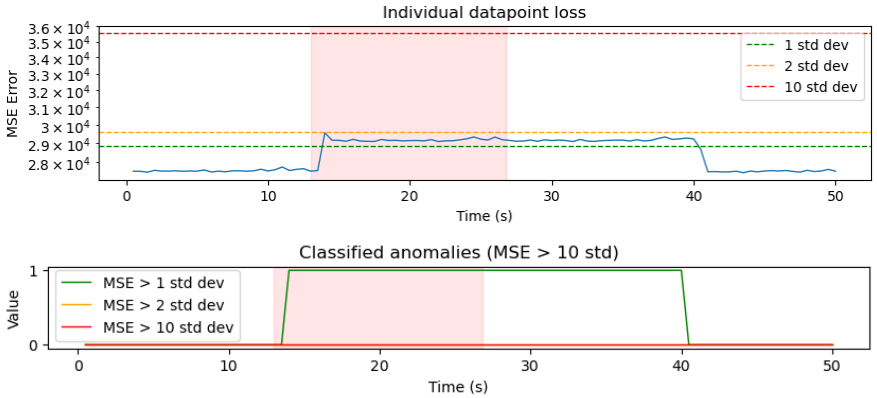


Figure 4.2 The reconstruction loss when using the GCN-ones on Node death IO & PLC data

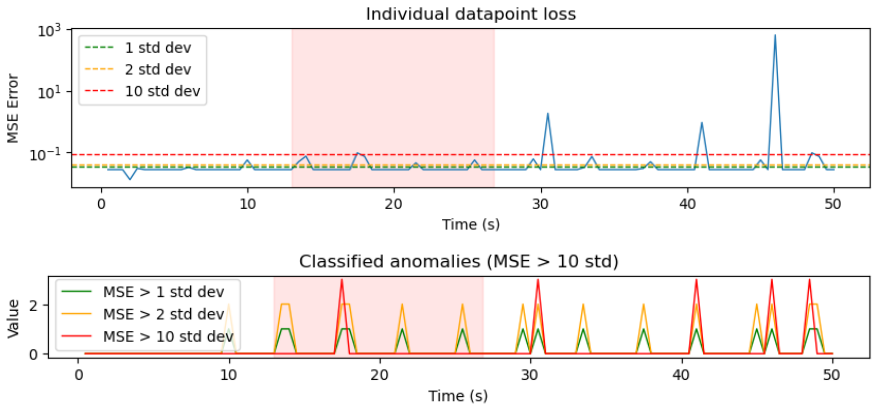


Figure 4.3 The reconstruction loss when using the GCN on cable rupture data

4.4 Custom Graph Models Implementation Details

The following applies to both our GCN and GAT:

- **Graph representation:** We decided to use adjacency matrices to represent our graphs, mostly because they are intuitive and easy to create. Although some graphs will not utilize the entire matrix size when achieving graphs of equal size, we do not think the sparsity will affect performance significantly.

- **Number of layers:** For our graph, we have found that the maximum number of edges between two nodes is three. Therefore, we found it reasonable to limit the number of layers of both the GCN and GAT to three.

- **Data normalization:** Through rigorous testing and inspection of feature values, we concluded that the data should not be normalized, as it was generally imbalanced: a few extreme values would cause most of the values to be scaled down, resulting in values close to zero. Instead, we standardized the data (mean=0, std=1) which was sufficient for our purposes.

GCN Implementation

The GCN is not trained, as it is not necessary when only used for feature extraction. The model is implemented by hand using the Tensorflow [TensorFlow, 2024] library in Python. The following parameters are used:

- **Activation functions:** The choice of activation function is implementation specific, and we go with the ReLU for its sparse nature.

- **Layer sizes:** Testing different combinations, we came up with the following dimensions for our hidden layers (for N features per node): $8N$, $4N$, $3N$.

GAT Implementation

The GAT is trained, as attention scores should not be left to the initial values, however training is done separately from the receiving autoencoder. We use the GATConv module from the Deep Graph Library [Zhang et al., 2024] in Python to implement the GAT. Our parameter choices were:

- **Number of heads:** We decided to use 8 heads, to hopefully stabilize the learning process enough, and it seemed to provide the best results.

- **Layer sizes:** For our two layers we use the sizes 64 and N respectively (for N node features), after experimentation.

- **Activation functions:** Our implementation uses the Exponential Linear Unit, as it is continuous.

- **Loss function:** We use MSE (section 2.8.1) as it is preferred for unsupervised training.

- **Optimizer:** We include the previously described Adam optimizer 2.7, with a learning rate set to 0.01.

4.5 MTAD-GAT

Preliminary Tests on Sensor Data

Initial tests were done on Salinity and Temperature sensor data that were "simulated". The data essentially consisted of a sinus function, and a alternating linear function. These tests are similar to the data used in the original MTAD-GAT paper [Zhao et al., 2020].

Testing on sensor data gave reasonable results, as shown in Figure 4.4. Some synthetically created anomalies that were made by simple data manipulation were detected. In the plots we can see the anomaly scores before and after gaussian blur, the actual anomalies, as well as train and test data.

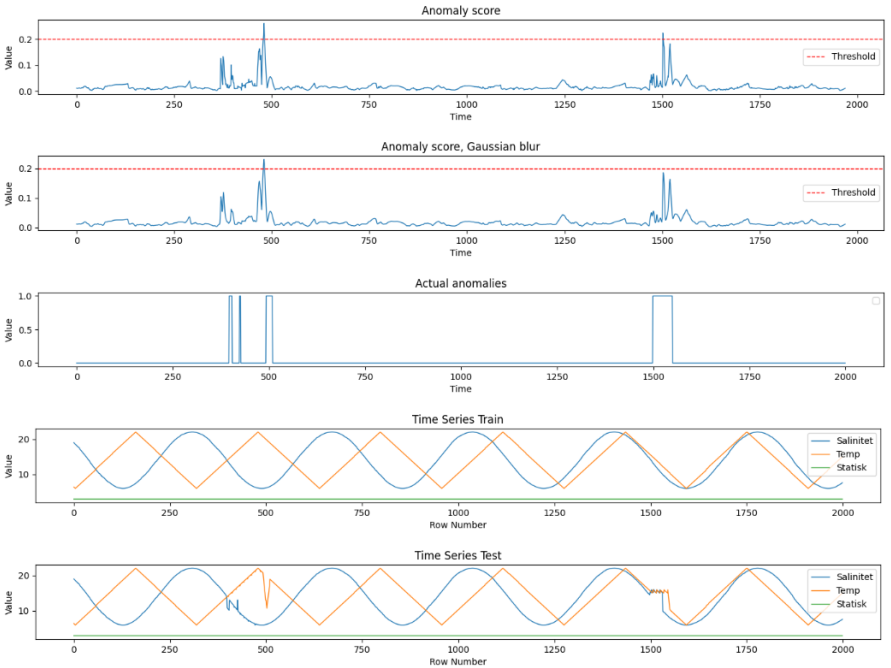


Figure 4.4 Time series of anomaly scores, where the detected anomalies are above the threshold; the actual anomalies; the training data, containing three sensors: salinity, temperature and a static value; and the testing data identical to the training data except for the three introduced anomalies.

When adding noise to the training and test data, by varying the values by a certain random amount,the anomaly scores were initially not very readable. However, that was fixed by running the through a Gaussian blur, as shown in the figures. When introducing noise the performance decreased as shown in Figure 4.5. However, the errors introduced were quite benign, and could themselves be considered as noise. Introducing larger anomalies gave more significant results, see Figure 4.6.

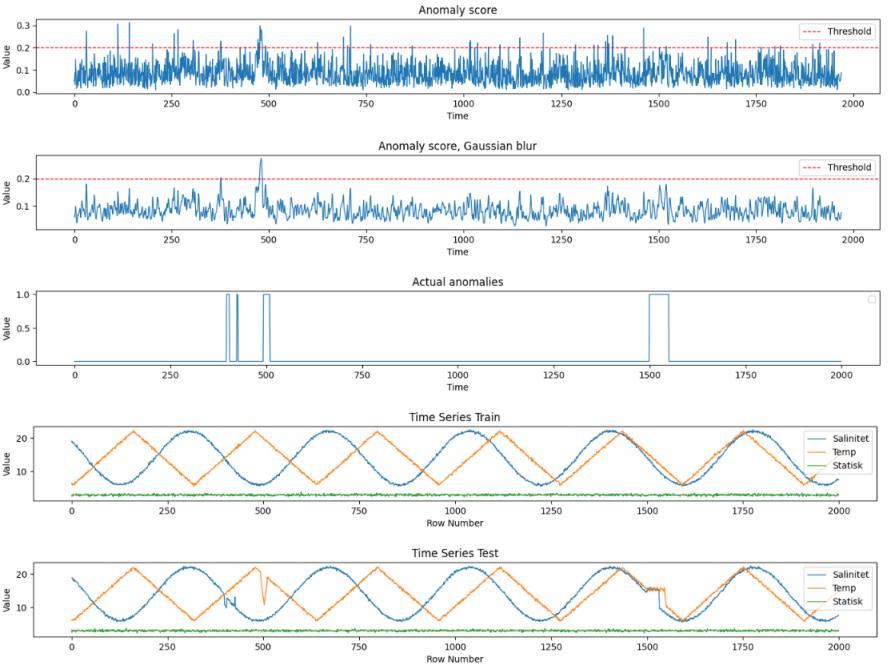


Figure 4.5 Training and test data with added noise. Time series of anomaly scores, where the detected anomalies are above the threshold; the actual anomalies; the training data, containing three sensors: salinity, temperature and a static value; and the testing data identical to the training data except for the three introduced anomalies.

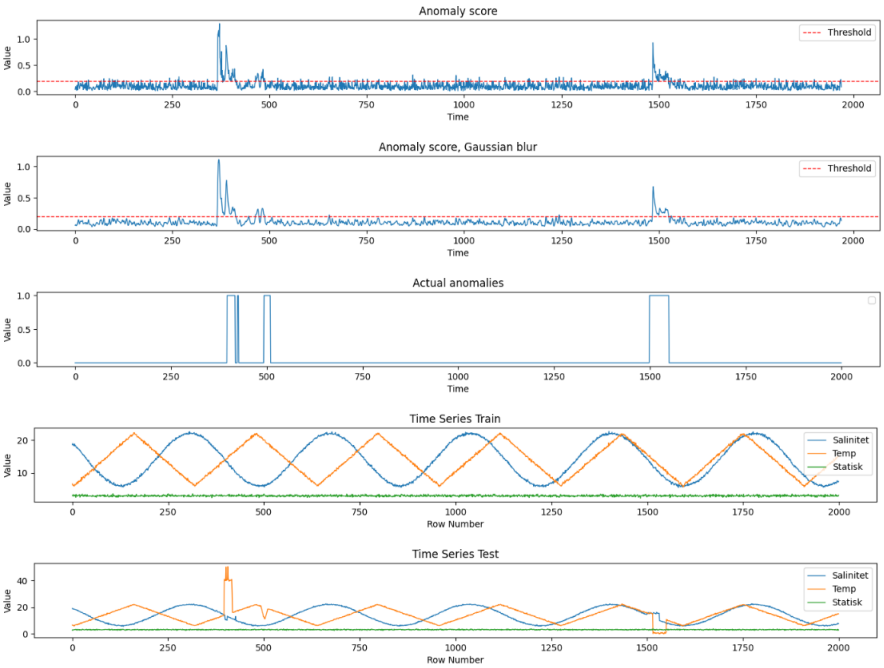


Figure 4.6 Time series of anomaly scores, where the detected anomalies are above the threshold; the actual anomalies; the training data, containing three sensors: salinity, temperature and a static value; and the testing data identical to the training data except for the three introduced anomalies.

This training was done using a *lookback* of 10, meaning that the time dependent GAT considered the last 10 values when predicting a new value. Increasing the lookback did not improve performance much, however it significantly increased the size of the model with the complexity $\mathcal{O}(N^2)$ as every new node calculates connections with all previous ones.

The anomaly detection here was performed by an advanced method for detecting anomalies in time series data [NetManAIOps-OmniAnomaly, 2024], which we did not use any further in our own tests.

We All models were trained with around 20 epochs, as the training loss stopped decreasing after this amount. This can be seen in Figure 4.7.

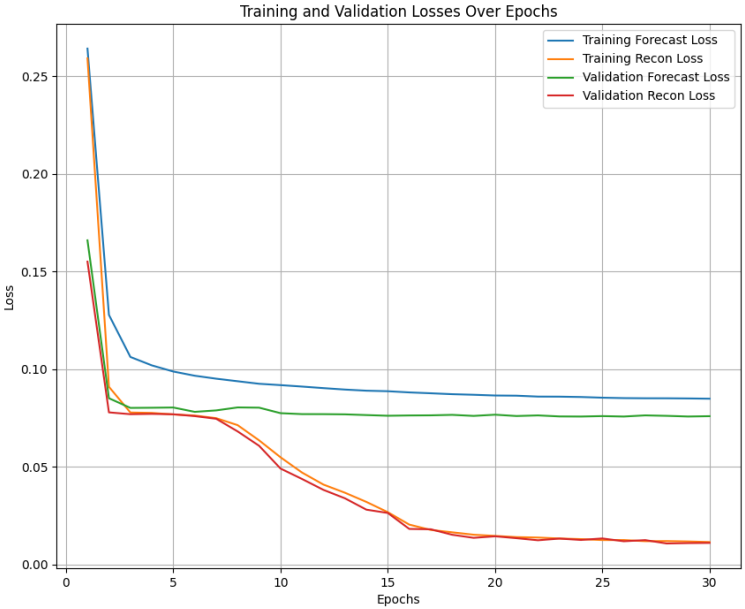


Figure 4.7 The resulting loss when training MTAD-GAT on salinity and temperature data

Network Data

We tested the smoothing methods in Section 3.10 on each test case. We also tried both approaches for threshold setting in Section 3.15. For the standard deviation method, we have set the levels for each test that seemed to make the classified anomalies resemble the actual anomalies the most. For the performance optimizing threshold, we chose to optimize the accuracy as it gave slightly better results than F_1-scores; we also used a lower threshold of 50 percent (green) and a higher threshold of 150 percent (red), of the optimal threshold (yellow). Figure 4.8–4.10 are the best results we managed to get for each test case. The rest of the results are Figure A19–A30 in the appendix.

The figures are divided into three plots. The top plot shows the reconstruction error (blue) and three thresholds. The x-axis shows the package number, and the y-axis shows the reconstruction error. The middle plot shows when the reconstruction error exceeded the thresholds. Finally, the bottom plot indicates when the actual anomalies occurred.

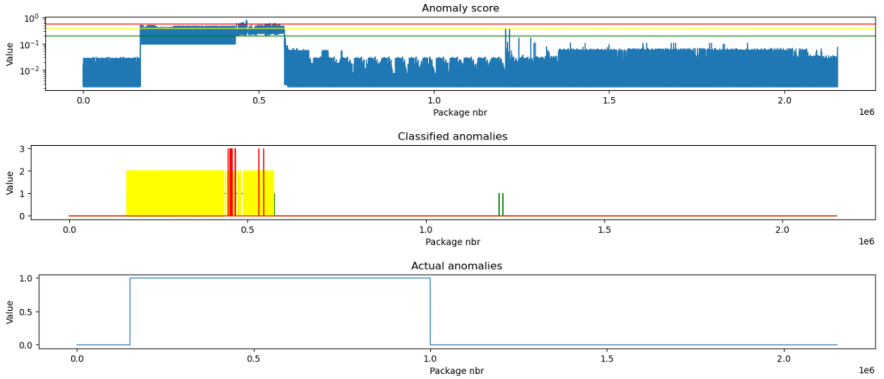


Figure 4.8 Node death PLC gaussian blur 5, accuracy-based threshold.

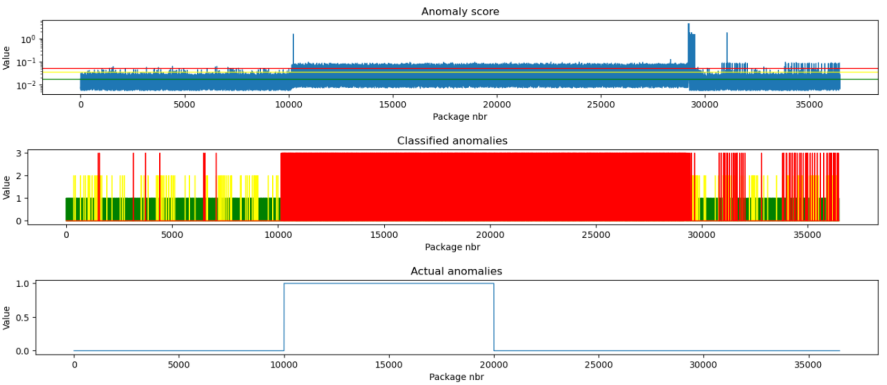


Figure 4.9 Node death IO PLC aggregated 10, accuracy-based threshold.

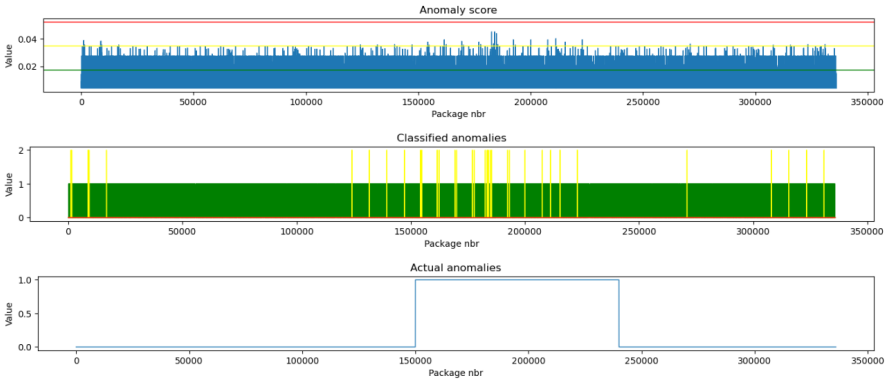


Figure 4.10 Cable rupture IO moving average 10, accuracy-based threshold.

4.6 Standalone Autoencoder

Modified Test Data

We performed an experiment on 145 logs from the train_1 recording. After training the model, logs $60-80$ were picked out, and 20% of each the bytes in each message were assigned a random value, providing us with corrupted messages. In Figure 4.11 we can see the reconstruction error from our autoencoder for each package, and we can see that the error increased dramatically for packages $60-80$.

We cannot guarantee that the messages still are invalid after change, as randomly changing the bytes might still produce a valid message. However, we deemed the probability of such scenarios to be quite low. This test proved the potential to use autoencoders for detecting anomalies in single message logs.

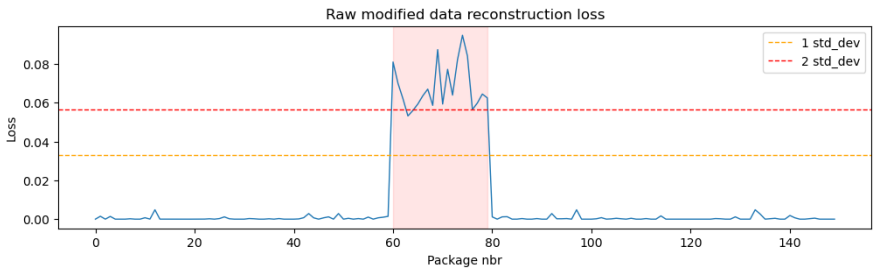
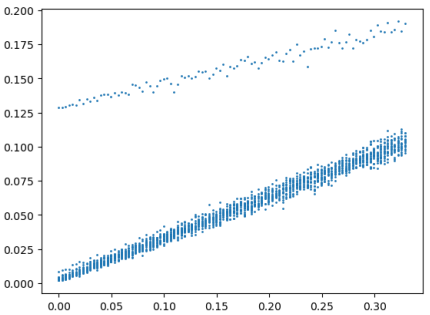


Figure 4.11 Reconstruction error of standalone autoencoder on raw data. Each input datapoint contains a network package converted into decimal values. The red area is the modified range, where each package randomly gets values changed.

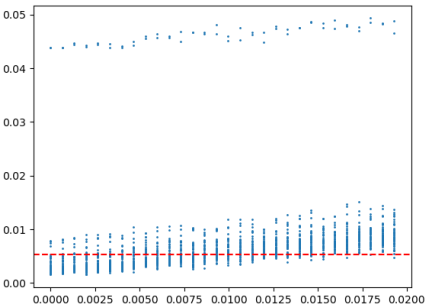
Testing Sensitivity

We used a similar approach here that we did when evaluating the GCN/GAT models with node malfunction. But instead of changing the features of each node we decided to change bytes in messages, as described in Algorithm 5.

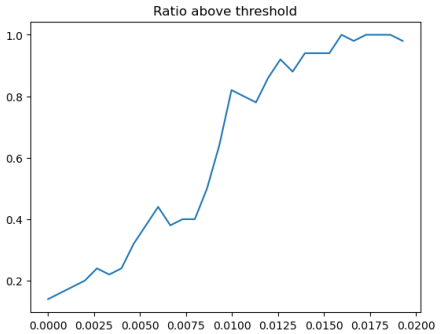
When randomly picking out bytes with different intensities (increasing the amount for each step) and assigning a completely new value to each byte we received the results in Figure 4.12.



(a) Autoencoder loss for node malfunction, changing 1-30%.



(b) Autoencoder loss for node malfunction, changing 1-2%



(c) ratios of points above $mean \cdot 1.05$ in Figure 4.12b.

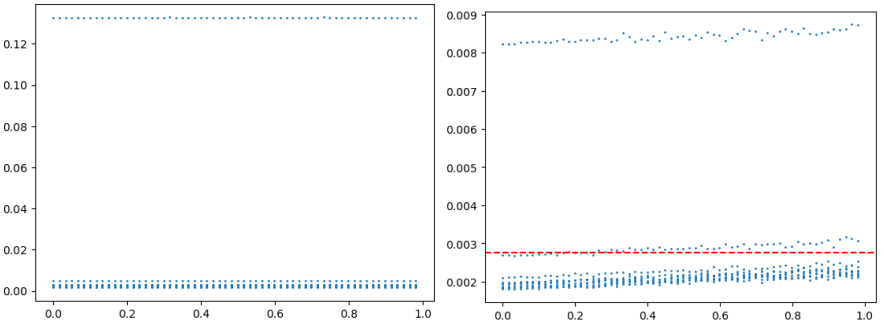
Figure 4.12

We can clearly see that the error increases linearly in Figure 4.12a when we change 0% to 30% bits. Now when we tried to find, similar to the experiment with GCN/-GAT, where the sensitivity had its limits, it turned out to be when changing around $1-2\%$ of bytes, which can be seen in Figure 4.12b. The ratios above the threshold $mean \cdot 1.05$ can be seen in Figure 4.12c.

We can from the ratio see that the anomaly detection become meaningful when dealing with changes that affect around 1% of the bytes, which is quite good.

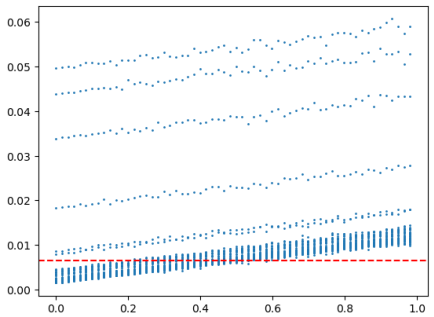
As we now changed the bytes randomly, we wanted to test how much we have to change each byte $(0-256)$ in order to get meaningful results. When adding 1 or subtracting 1 from a random byte in the data point for a certain percentage of nodes we got the results seen in Figure 4.13a. Even when changing 100% of values, we did not achieve any meaningful error.

When trying to change the values with either -5 or 5, we achieved the results that can bee seen in Figure 4.13b, which were close but not good enough. Finally, it seemed that when adding or subtracting 25 from random values proved to give a good result, as seen in Figure 4.13c. The ratios above $mean \cdot 1.05$ can be seen in Figure 4.13d.

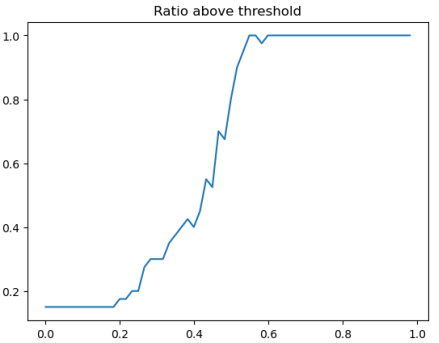


(a) changing the values with 1 or −1

(b) changing the values with 5 or −5



(c) changing the values with 25 or −25



(d) changing the values with 25 or −25

Figure 4.13

Simulated Data

While trying to test a standalone autoencoder on recorded data, we did not manage to achieve any significant results, despite trying different model parameters. Two example plots can be seen in Figure 4.14–4.15.

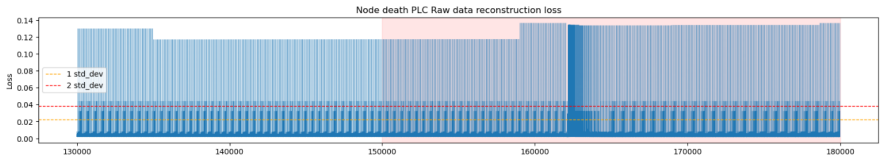


Figure 4.14 50 epochs, 512 batch size. Reconstruction error of standalone autoencoder on raw data. Tested on node death PLC. The real anomaly is represented by the red area.

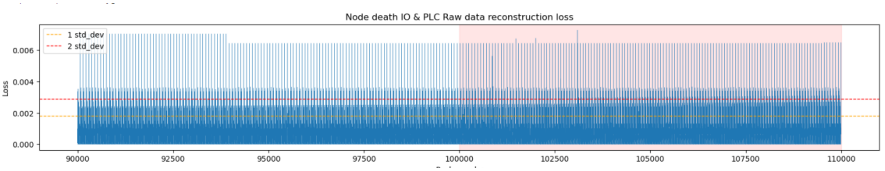


Figure 4.15 10 epochs, 256 batch size. Reconstruction error of standalone autoencoder on raw data. Tested on node death IO PLC. The real anomaly is represented by the red area.

We can see that the reconstruction error does neither increase nor decrease in a meaningful way when the anomaly occurs. This can be seen in the figures, as the loss does not increase when the anomaly occurs. In Figure 4.14 we can see a slight difference, in the way that the frequency of the high reconstruction error (the peaks) increases, however this was not deemed to be significant enough to be classified as an anomaly.

4.7 Autoencoder Model Architecture

The autoencoder model designed for this study consists of two principal components: an encoder and a decoder, both implemented using the Keras framework [Keras, 2024].

Encoder

The encoder comprises the following layers:

- **Input Layer:** Accepts input vectors of size `max_length`.
- **First Hidden Layer:** Consists of 256 neurons using ReLU activation, followed by batch normalization and LeakyReLU activation for added stability.
- **Second Hidden Layer:** Composed of 128 neurons, also using ReLU activation with L1 regularization to encourage sparsity, followed by batch normalization and LeakyReLU activation.
- **Third Hidden Layer:** Includes 64 neurons with ReLU activation to compress the data into a lower-dimensional representation.

Decoder

The decoder mirrors the encoder's architecture in reverse order to reconstruct the input data:

- **First Layer:** Expands the representation to 128 neurons, uses ReLU activation, followed by batch normalization and LeakyReLU.
- **Second Layer:** Further expands the data to 256 neurons with the same activations and regularizations as the previous layer.
- **Output Layer:** Reconstructs the original input using a sigmoid activation across `max_length` neurons.

Training and hyperparameters

The model is compiled with the Adam optimizer and trained using mean squared error as the loss function. Training is performed over 50 epochs with a batch size of 512. The model also utilizes data shuffling and validation with separate data sets. Upon completion of training, the model is saved to ensure reproducibility and for future evaluations.

5

Discussion

5.1 Own Models

Simulated Errors

GAT based Models In the case of GAT it can be clearly seen that using more features provided much better results, this was also expected, as more meaningful features often provide better results, therefore concluding that GAT-big and GAT-BigOnes gave the best results.

The performance of GAT-Big and Gat-BigOnes was similar when testing with the $0.0 - 0.1$ interval, however GAT-BigOnes had better correlations, and therefore proved to be better.

When iterating through the $0 - 0.02$ interval the correlation was better on GAT-Big, however it was not as sensitive as the GAT-BigOnes, therefore the latter one being better. Another thing to notice is that the sensitivity between the GAT-ones model did not change much when switching from $0.0 - 0.1$ to $0 - 0.02$.

This experiment showed that it can be beneficial to let the GAT learn its own structures, instead of only using the predefined ones.

GCN based Models In the case of GCN we can also see that the GCN and GCN-Big performed the worst, and in the interval of $0 - 0.02$ both failed to produce any meaningful result. This can be attributed to the fact that the GCN is not being explicitly trained, and just performs convolutions. The adjacency matrix is quite sparse in the GCN with the predefined structure as input. This leads to the convolution not producing anything meaningful, especially in the case of GCN-Big.

The other models, GCN-BigOnes and GCN-Ones proved to be sensitive to the graph overload, performing in a similar fashion to the GAT models.

Comparing CGN and GAT We could also see that the best models from both categories performed similarly well, reinforcing out hypothesis about the underlying network not being complex enough for a GAT model to provide better results.

Recorded Data

In the Node death PLC test, GCN-Ones performed the best. In Node death IO & PLC, all models except GCN-Zeros, which had low values, were more or less the same. In cable rupture, the same three models were about equally bad. Overall, it is safe to say that GCN-Zeros was inferior to the other models.

Perhaps the most surprising result was that GAT, despite its level of sophistication, did not perform better than it did, especially with respect to its feature extraction process being trained unlike GCN.

The GAT performing in the same manner as GCN could be attributed to the simplicity of our system, as shown in Figure 5.1.

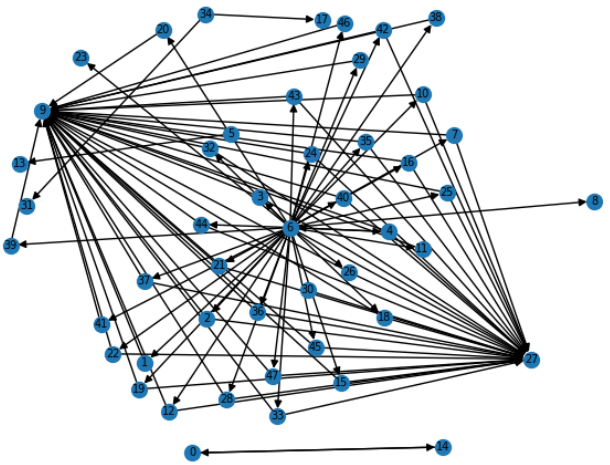


Figure 5.1 Communications between nodes.

There are no complex connections between nodes, from the image we can conclude that there are 3 central nodes having a lot a connections, and the rest of the nodes having only around 2-3 connections.

In a case of a more complex system, GAT could potentially perform better, however in this case GCN did suffice.

The fact that GCN was as good as the GAT model can also be attributed to the quality of data. The data used was simply the system in a "steady state" where nothing was happening. This is not the typical scenario, and perhaps with training data covering more complex scenarios, a GAT would be better in distinguishing between anomalies and non-anomalies.

This is however system dependent, some systems might or should act the same and not vary a lot during runtime, here according to our tests as GCN would suffice. However, some systems might vary a lot, in those cases a GAT would potentially be a better choice.

General Observations

Overall, we could see GAT and GCN performed similarly, however there was a clear advantage of letting GAT learn its own connections for all nodes instead of limiting it to the predefined structures. In the case of GCN it was also more advantageous to let all nodes be connected to all nodes. This being said, the best models were GCN-BigOnes and GAT-BigOnes. The cable rupture test was not being identified, which makes sense as we are operating in a ring network, which should not be affected if one node is disconnected.

Improving Feature Extraction

Our feature vector only incorporated the data regarding the number of packages sent, as well as the average lengths of them. The time aspect was incorporated when *dividing by time*. This approach is dependent on which intervals we use, and does not incorporate time perfectly. Some more complex feature extraction might provide better results.

An example of a complex feature vector might be one that uses the information contained in the message as a feature, instead of only using the length and size as in our case.

Improving Training of the Models

The GAT model and its autoencoder were trained separately. This is not the optimal solution, as they could be trained in the same feedback loop. This would ensure the models are adapting together to our data, instead of doing it each on their own.

In regards of the GCN, the weights between layers could and should also be trained. Preferably in the same feedback loop as the autoencoder.

Another relevant improvement could be to utilize more of the regularization techniques described in Section 2.9. Specifically, dropout is mentioned as particularly useful for GAT when applied to attention coefficients [Veličković, 2024]. It means that neighborhoods will be stochastically sampled and dependencies reducing generalization are minimized. This is especially useful for small training datasets, such as our own.

Improved GAT Model

Our GAT model could possibly be improved by not training the GAT and autoencoder separately, but instead using feedback from the autoencoder to the GAT. This would mean that the GAT will be adapted when the total loss changes.

GATv2: There is a recent improvement to the GAT attention mechanism, called GATv2 [Brody et al., 2022]. The idea is that the original GAT calculates the scoring function in a static manner, i.e. the ranking of attention coefficients α is shared between all nodes, see Equation 2.3.

Instead, GATv2 proposes a different order of operations in the scoring function, preventing $\boldsymbol{a}$ and $\boldsymbol{W}$ from collapsing into a single linear layer:

$$e(\boldsymbol{h}_i, \boldsymbol{h}_j) = \boldsymbol{a}^T \text{LeakyReLU}\left(\boldsymbol{W} \cdot [\boldsymbol{h}_i \| \boldsymbol{h}_j]\right)$$

which they prove will compute dynamic attention, meaning that nodes will have different rankings of neighbors. This is shown to outperform GAT in their benchmarks, while having the same time complexity and parametric cost. However, the importance of this attention improvement is thought to depend on how complex the node interactions are.

Importance of Nodes in the System

While testing our own models using graph_overload and node_malfunction we noticed that in some cases our results took the form of two separate lines which points gathered around. Those lines came in different forms depending on the model and tests, however in most cases there were exactly two lines that could be identified.

Graph Overload

When testing graph overload, we in many times got results where the MSE errors could be split in two exponential lines, distinct from each other. An example can be seen in Figure 5.2.

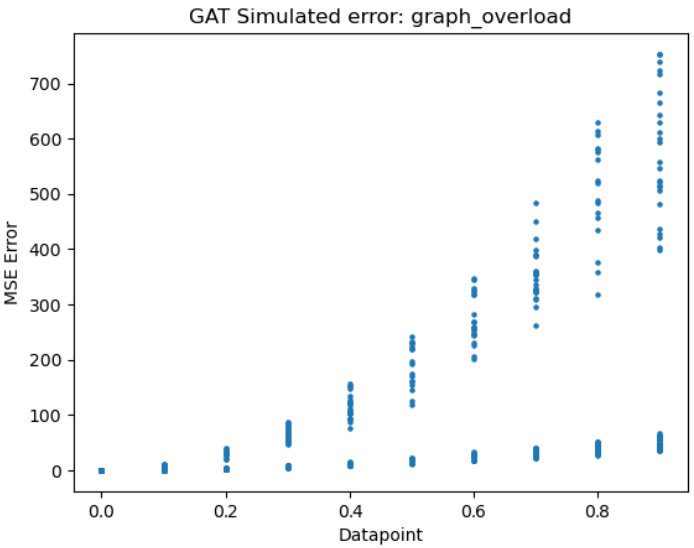


Figure 5.2 Graph overload, x-axis values are overload factors. The values at the x-axis represent by how many percent the nodes feature was increased. In this scenario 10 nodes were affected.

This occurred quite often with many different models during testing, and occurred much more often when using the GAT variation than it did when using the GCN variation.

One possible explanation to this phenomenon is the relative importance of nodes. The graph had a few central nodes, that received a lot more messages than the rest of the nodes. As our sampling method was stochastic, and we picked random nodes each iteration, the split of results could be explained by which nodes were picked.

Our hypothesis is that when we happened to pick and modify the features of an important node the error became much higher, which could symbolize the upper line. However, in the case that we managed to not pick an important node, we got a much lower error as seen in the bottom line.

This explanation aligns well with the fact that GAT learns the relative importance of nodes in a graph, and GCN does not. This phenomenon was much more frequent for GAT models, than for GCN during experiments.

Node Malfunction

When testing Node Malfunction there was a large variation in the graphs it produced. However, in many cases we got an "X" shape when training a GAT model,

which can be seen in Figure 5.3.

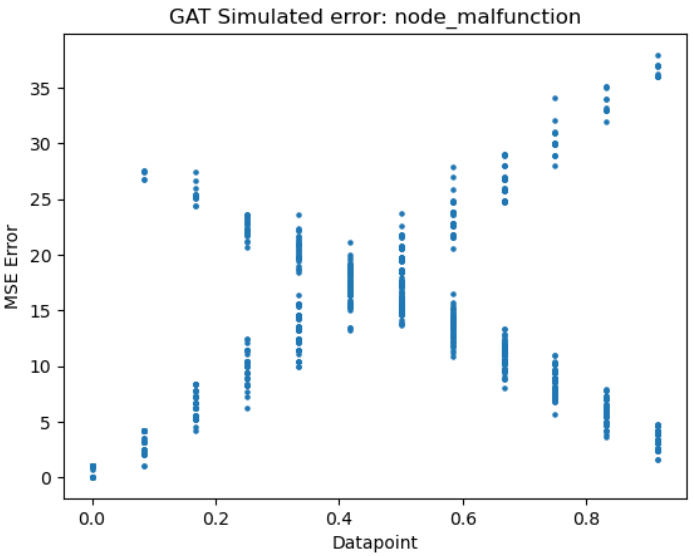


Figure 5.3 Node malfunction. x-axis values are the fraction of nodes that were turned off.

We can see two distinct lines, where the number of points for each line is larger at the bottom portion of both the increasing and the decreasing line. This is also likely attributed to the importance of certain nodes, and how GAT learns about them.

In the beginning, when shutting off a small percentage of nodes, there is a high chance that one of the few central nodes will not get shutdown, therefore the majority of the simulations have a low MSE, however in the few cases when a central node gets shut down, we got a higher MSE.

When we approach the end of our testing interval things kind of reverse. Here the model seems to learn that shutting off all nodes means that the system behaves normally, therefore the low MSE error. However, we still manage to get some points with a high error. It is quite unclear why it is this way.

It could be argued that the large MSE errors occur when only some, but not all central nodes get shut off, meaning that the network learned that when all central nodes are on, the system works as usual, and when only some of the nodes work, an anomaly has occurred.

These are of course only conjectures, and this should be explored further. It is

important to mention that we did not get an X for all models we trained, however it occurred frequently enough to justify trying to understand the reason behind.

5.2 MTAD-GAT

This method being based on the paper [Zhao et al., 2020] was far more complex than our initial models and was thought to realistically produce better results.

Recorded Data

When testing recorded data we got satisfactory result for Node death PLC and Node death IO & PLC, ignoring the potential measurement error (or system error). The thresholds were mainly found by maximizing the accuracy, and for that we had to change the ground truth in order for our algorithm to work properly, by extending the ground truth so it better matched the detected error.

For Node death PLC, we got a more even anomaly detection, instead of getting two peaks as in our own models. which we deemed a better result as the error did not decrease unlike the case when testing our own models.

For Node death IO & PLC, we got a similar result to our own models, and this result was also satisfactory.

This model also failed, just like our own simpler models, to detect the cable rupture in any meaningful way. We heavily doubt that changing the parameter and training with an extreme lookback, say 100 instead of 20 would improve the system a lot. The maximal lookback we tried was 50, this due to the space and time complexity of the model being $\mathcal{O}(N^2)$.

Improving Feature Extraction

The methods we used for extracting features were satisfactory, but could be improved. The time series only included if a signal was sent or received during a given time, and the size of the signal. We did not incorporate from whom the signal was received, we also did not incorporate the content of the message.

Using the content of the message would be quite tricky in this case, but could be done by encoding the message and using its value as the input (instead of the size of the message). We do not know how such a model would behave.

Including who the messages were sent from and to could be done by having N time series for each node, assuming N is the number of nodes. This would however provide us with N^2 time series, which in our case would be $81^2 = 6561$ time series, which would produce a very sparse time series matrix, and aggregation would most

certainly be needed. We do not know how well the model would perform with such amount of time series, but we do know that it would be very computationally heavy, both for inference and training, we therefore deemed exploring such model not worth the time during this master thesis.

Time and computing power limitation should however not apply when designing safety critical systems, especially for submarines and naval crafts designed to keep a society safe, such feature extractions should therefore definitely be evaluated in the future.

5.3 Standalone Autoencoder

We could see that the standalone autoencoder did not detect quantitative anomalies very well, not all actually. This is logical due to the autoencoder only analyzing single message format, and as we "turn off" nodes as we did in our tests, the format and type of the messages did not necessarily change, depending on the system they either were sent through a different node or were not sent at all.

This approach would be better suited for qualitative errors, where the contents of messages are changed, either completely as represented by the tests replacing bytes completely, symbolizing different data content, or bytes simply being changed slightly (with for example one bit), symbolizing faulty packages.

However, it is worth mentioning that this method of byte manipulation theoretically can be unreliable, as we cannot be sure that the new data replacing the previous data actually represents an anomaly; although we consider it unlikely that replacing a significant number of bytes would consistently produce normal data. Nonetheless, the autoencoder was able to detect such manipulations, indicating that the changes made are not normal, at least in relation to the training data. As we see in the results, these errors were detected quite easily.

Analyzing Frequency Instead of Error Size

As mentioned, the results on the simulated data were not very good. However, when looking at the result from testing the standalone autoencoder, see Figure 4.15, we can see that although the error does not increase significantly during the anomaly, the frequency of high errors increases at one point. Analyzing the frequency of the errors could potentially provide some insights. Another method would be to for example use a Gaussian blur and inspect the results after such transformation.

Improving Feature Extraction

The feature vector we used simply consisted of the raw data's hexadecimal encodings that were normalized. This vector included the source, destination, protocol, message content and other things. While some things were highly relevant such as

destination, source and message content, some of the contents were not as useful, such as protocol (assuming the whole system uses the same protocol). Removing unneeded features/bytes from the feature vector could prove beneficial as the model would have less data to work with, therefore potentially increase its performance.

Yet another way would be to extract numerical data from the raw data, for example replacing the hexadecimal encoding of the source/destination mac address, with a simple integer that would be mapped to that specific mac address. Similar mapping could be performed for message contents, in the case that there is a standard set of such in the system. This approach would simplify the data, and potentially improve the models' performance.

RNN Model

Another potentially good approach could be implementing a RNN model analyzing raw data. Such model could potentially be able to detect quantitative changes, for example changes in frequency of certain messages, or increased number of messages in a specific timeframe. The most suitable architecture would most likely be an LSTM. However, it would require a quite big sequence length, to capture enough relevant data, and with our vectors being around 1500 bytes long, it would produce quite a large network. However as mentioned before different things could be done to improve the feature extraction, which would make the model possibly a lot smaller and manageable.

5.4 Observational Errors

Labeling Errors

In all our tests, Node death IO & PLC displayed a behavior where the anomaly signal was almost exactly double the length of our written down anomaly duration, see e.g. Figure A11, A12, A14 and Figure A23–A26. We suspect that this could be due to two reasons. Either the system has a delay that is only present when switching on the device, as the start times agree, but not the end times; or, perhaps more likely, there have been a clerical error and the end package number 200,000 should in fact be 300,000. The main concern with this is the possibility that our performance metrics (F_1 score) are lower than they should be. With this in mind, we adjusted the ending value for our accuracy optimizing thresholds (Algorithm 8), which increased maximum performance.

System Specific Behavior

What is considered an anomaly heavily depends on the specific system and its operator. Therefore, it is difficult and sub-optimal to set a static threshold that will classify an event as an anomaly. An illustrating example of this is Figure A7, where

the error exceeds the threshold when the actual anomaly begins, but then descends far below it until the state is restored. It is thus important to emphasize that knowledge about the system behavior is needed to draw conclusions from the anomaly signals, and operators should be able to adjust thresholds and not solely rely on the system monitoring.

5.5 Finding Thresholds

In this study the thresholds for classifying anomalies were often based on the correct labels for anomalies, which would not be possible in real life scenarios. Instead, an expert user could determine the threshold based on relevancy. The results in this thesis are presented as potential maximal results that could be achieved with different models, given that a correct threshold method was chosen.

One problem that occurred was when our test data was different than our training data, meaning that the overall error between data and reconstruction was much larger for test data than for training data. This case was not hopeless however, as anomalies could still easily be detected when analyzing the series of errors, this because the "error" produced peaks in the time series, which could be distinguished from "normal" data.

The initial and most logical approach for finding a threshold would be to base it on the training data's loss. As mentioned above this was not possible in our cases, and we instead resolved to maximizing F_1 score or basing our threshold on the test data's mean and standard deviation.

These methods can definitely be expanded in some form adaptable threshold finding, where the actual threshold also has to be learned from training and test data. This would further add another step to the complexity of the problem, and increase the amount of data needed. However, it is essential for real life scenarios.

Another important thing, as mentioned several times, is the system's behavior in the case of anomaly. A system can behave differently after an anomaly for several hours, or simply fix itself mere seconds after the anomaly, even though an anomaly is still technically present. It is important to map the ground truth correctly to the machine learning models, in order to accurately train or calculate a satisfactory threshold.

5.6 Acquiring Better Data

Better Training Data

The main reason we should train on extensive data is that it allows for better generalization. This is because more training data can contain more variations, which means that the model will not be locked to one specific type of behavior when assuming what is normal. This is the same phenomenon that reduces overfitting: without a large enough sample, we will not have a representative distribution of our data. This becomes especially important with more complex models, such as GAT, which can easily overfit without enough data. For example, the more weights a model has the higher its ability is to predict based on its training data, but if this data only represents a small sample, it will inevitably be biased. This will, when testing on new unseen data, affect performance negatively.

In order to fully take advantage of the complexity and test the GAT in relation to simpler models, we estimate that the models used should in an ideal scenario be trained on multiple times the current amount of data. To achieve this, we suggest that the simulations of the normal state should be performed over a longer time period, while capturing an increased variation of events.

Better Test Data

More test data would allow us to evaluate the models with higher certainty. To accurately assess generalization, we need more than a small sample of unseen test data. The other aspect is that we would like to test more types of anomalies, to see what the model is capable of detecting, and what its limitations are. For example, we would like to test more of the "slow" errors, such as increasing network traffic, and various levels of packet loss, which are more difficult to spot and where the GAT may be more valuable.

The amount of test data also limited our experiments to qualitative evaluation, instead of quantitative.

Better Labels

As mentioned in the Section observational errors, knowledge of the underlying system is needed in order to do proper evaluation and produce proper labels. Improper labeling makes quantitative evaluation (such as using F_1 scores, or accuracy) almost useless, as even though the F_1 score might be bad, the result is good, and looks good. This can be seen on Table 4.4, where F_1 scores were bad, even though technically an anomaly was detected properly, however the inaccurate labeling skewed the results.

In this case it was not the end of the world. However, when using the labels for

dynamic and adaptable thresholds (such as ones that are learned during inference) this can be a major issue.

6

Conclusion

6.1 Can GATs be Used to Monitor Complex Operational Systems?

GCN vs GAT

We did not find a strong reason to use a GAT instead of a classical GCN, however the cause of such results deducted to be the simplicity of the systems and quality of data. An advanced model such as MTAD-GAT worked slightly better that the simpler models, however needed much more computing power for training and inference.

To conclude, GAT did not increase the performance of the model enough to justify its complexity, this is only applicable to the data we trained and tested on, which as mentioned before, was not of top-notch quality.

Standalone Autoencoder

As our own models, and the MTAD-GAT have a hard time locating the exact location of the error in the graph we examined the use of a standalone autoencoder, which would be a complement to the main GAT-based model.

This approach proved successful when analyzing qualitative errors of single messages, but it did not however give any promising results when analyzing quantitative errors.

A proposed solution for the quantitative errors was to explore the use of RNNs, which analyze sequences of data.

Positive Findings

Despite the need for further research, there are a few significant findings:

- GAT, GCN and standalone autoencoder were all sensitive to increasing errors that are synthetically induced in simulated data.
- Both GAT and GCN running on simulated data were able to correctly indicate that anomalies were occurring, at least for Node death PLC and Node death IO & PLC.
- MTAD-GAT showed promising results, and was able to somewhat indicate a cable rupture.

6.2 Limitations of our Study

The biggest limitation of this study was the lack of data, which mainly caused us to perform qualitative evaluation instead of quantitative. There was not enough varying test data or training data, which also severely limited our model to only learning the most usual case of the systems behavior. Anomaly detection usually require astronomical amounts of data, and creating such datasets was not feasible during this study.

Another limitation was the bad labeling of the data. A proper labeling was hindered by our lack of knowledge of the underlying system and its behaviour during different events. It was hard to deduce for how long the system was behaving anomalously after an abnormality was induced.

The feature extraction we performed for different models was sub-optimal and, in many cases, failed to incorporate all dimensions of the data. This means that the results could potentially have been better, for example in the case of Cable rupture IO if we used all available dimensions of the data.

Another major limitation was that we trained our GAT and Autoencoder model separately, which may not be optimal, and using the same feedback for both models could improve the performance significantly. The GCN model should also have been trained, as we only used it for feature extraction, without training the weights between layers.

6.3 Future Improvements

Better Data

As mentioned before, it is imperial to acquire better data, both in terms of amount, quality and labeling.

More Complex Systems

In order to fully take advantage of GAT, larger systems could be analyzed that have more complex relations between nodes and also vary a lot more.

Feature Extraction

Extracting features better in order to capture the dimensions of the systems in a better way is also a major area of improvement.

Better Models

More complex and different models could be analyzed. One approach would be to for example use a standalone autoencoder together with the MTAD-GAT model, in order to identify when an anomaly occurred (through MTAD-GAT) and identify where an anomaly occurred using the autoencoder. Another model that can be worth comparing is GATv2, especially if more complex relations are present in the graph structure. For our GAT model, adding dropout can also be worth-while in order to increase generalization.

Other Comparisons

Interesting subjects for future research could be to look at the machine learning models discussed in this work compared to the built-in system monitoring.

Bibliography

B&R (2024). *Powerlink*. URL: https://www.br-automation.com/sv/teknologier/powerlink/ (visited on 2024-05-07).

Brody, S., U. Alon, and E. Yahav (2022). *How attentive are graph attention networks?* arXiv: 2105.14491 [cs.LG].

Chapelle, O., B. Scholkopf, and A. Zien, (Eds.) (2006). *Semi-Supervised Learning*. MIT Press.

Cho, K., B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). *Learning phrase representations using rnn encoder-decoder for statistical machine translation*. arXiv: 1406.1078 [cs.CL].

Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function". *Mathematics of Control, Signals, and Systems* **2**, pp. 303–314. DOI: 10.1007/bf02551274. URL: https://link.springer.com/article/10.1007%2FBF02551274.

Deng, A. and B. Hooi (2021). *Graph neural network-based anomaly detection in multivariate time series*. arXiv: 2106.06947 [cs.LG].

Ding, C., S. Sun, and J. Zhao (2023). "Mst-gat: a multimodal spatial–temporal graph attention network for time series anomaly detection". *Information Fusion* **89**, pp. 527–536. ISSN: 1566-2535. DOI: 10.1016/j.inffus.2022.08.011. URL: http://dx.doi.org/10.1016/j.inffus.2022.08.011.

Harstad, A. and W. Kvaale (2024). *Ml4its/mtad-gat-pytorch*. URL: https://github.com/ML4ITS/mtad-gat-pytorch (visited on 2024-05-07).

Ho, T. K. K., A. Karami, and N. Armanfard (2024). *Graph anomaly detection in time series: a survey*. arXiv: 2302.00058 [cs.LG].

Hochreiter, S. and J. Schmidhuber (1997). "Long Short-Term Memory". *Neural Computation* **9**:8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

Keras (2024). *Keras*. URL: https://keras.io/ (visited on 2024-05-07).

Krishna Kishore, P., S. Ramamoorthy, and V. Rajavarman (2023). "Artp: anomaly based real time prevention of distributed denial of service attacks on the web using machine learning approach". *International Journal of Intelligent Networks* **4**, pp. 38–45. ISSN: 2666-6030. DOI: https://doi.org/10.1016/j.ijin.2022.12.001. URL: https://www.sciencedirect.com/science/article/pii/S2666603022000380.

Lisova, E., M. Gutiérrez, W. Steiner, E. Uhlemann, J. Akerberg, R. Dobrin, and M. Björkman (2016). "Protecting clock synchronization: adversary detection through network monitoring". *Journal of Electrical and Computer Engineering* **2016**, pp. 1–13. DOI: 10.1155/2016/6297476.

Lutz, R. R. and I. Carmen Mikulski (2003). "Operational anomalies as a cause of safety-critical requirements evolution". *Journal of Systems and Software* **65**:2, pp. 155–161. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(02)00057-2. URL: https://www.sciencedirect.com/science/article/pii/S0164121202000572.

Michałowska, K., S. Riemer-Sørensen, C. Sterud, and O. M. Hjellset (2021). "Anomaly detection with unknown anomalies: application to maritime machinery". *IFAC-PapersOnLine* **54**:16. 13th IFAC Conference on Control Applications in Marine Systems, Robotics, and Vehicles CAMS 2021, pp. 105–111. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2021.10.080. URL: https://www.sciencedirect.com/science/article/pii/S2405896321014828.

NetManAIOps-OmniAnomaly (2024). *Ml4its/mtad-gat-pytorch*. URL: https://github.com/NetManAIOps/OmniAnomaly (visited on 2024-05-07).

Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini (2009). "The graph neural network model". *IEEE Transactions on Neural Networks* **20**:1, pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

TensorFlow (2024). *Tensorflow*. URL: https://www.tensorflow.org/ (visited on 2024-05-07).

Veličković, P. (2024). *Graph attention networks*. URL: https://petar-v.com/GAT/ (visited on 2024-05-07).

Veličković, P., G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio (2018). *Graph attention networks*. arXiv: 1710.10903 [stat.ML].

Wireshark.org (2024). URL: https://www.wireshark.org/ (visited on 2024-05-07).

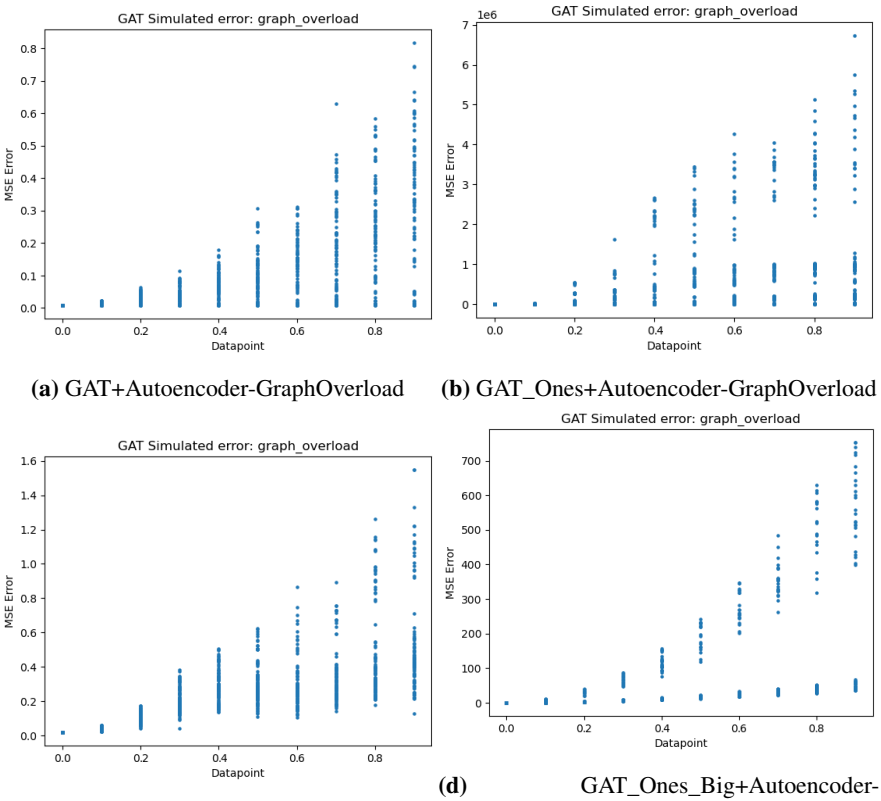
Zhang, H., M. Li, M. Wang, and Z. Zhang (2024). *Understand graph attention network*. URL: https://docs.dgl.ai/en/0.8.x/tutorials/models/1_gnn/9_gat.html (visited on 2024-05-07).

Zhao, H., Y. Wang, J. Duan, C. Huang, D. Cao, Y. Tong, B. Xu, J. Bai, J. Tong, and Q. Zhang (2020). *Multivariate time-series anomaly detection via graph attention network*. arXiv: 2009.02040 [cs.LG].

Appendix

A1 Figures

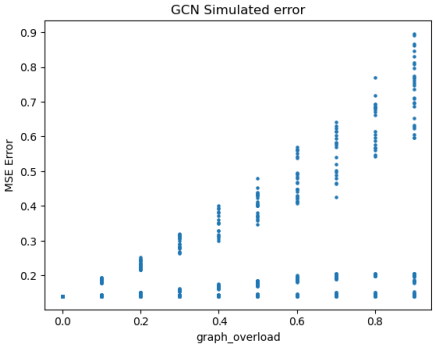
Generated Error Sensitivity



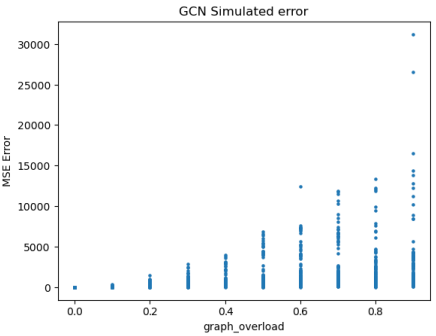
(a) GAT+Autoencoder-GraphOverload **(b)** GAT_Ones+Autoencoder-GraphOverload

(c) GAT_Big+Autoencoder-GraphOverload **(d)** GAT_Ones_Big+Autoencoder-GraphOverload

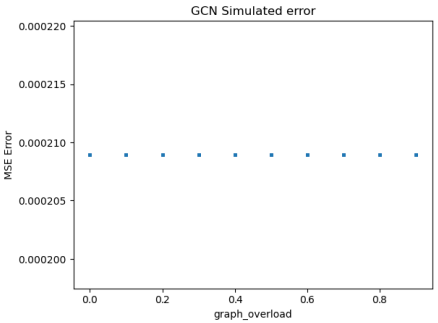
Figure A1 Comparative results of GAT variations with Autoencoder-GraphOverload



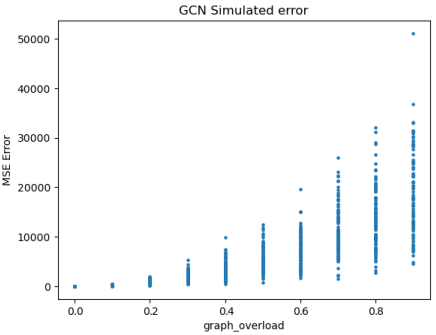
(a) GCN+Autoencoder-GraphOverload



(b) GCN_Ones+Autoencoder-GraphOverload

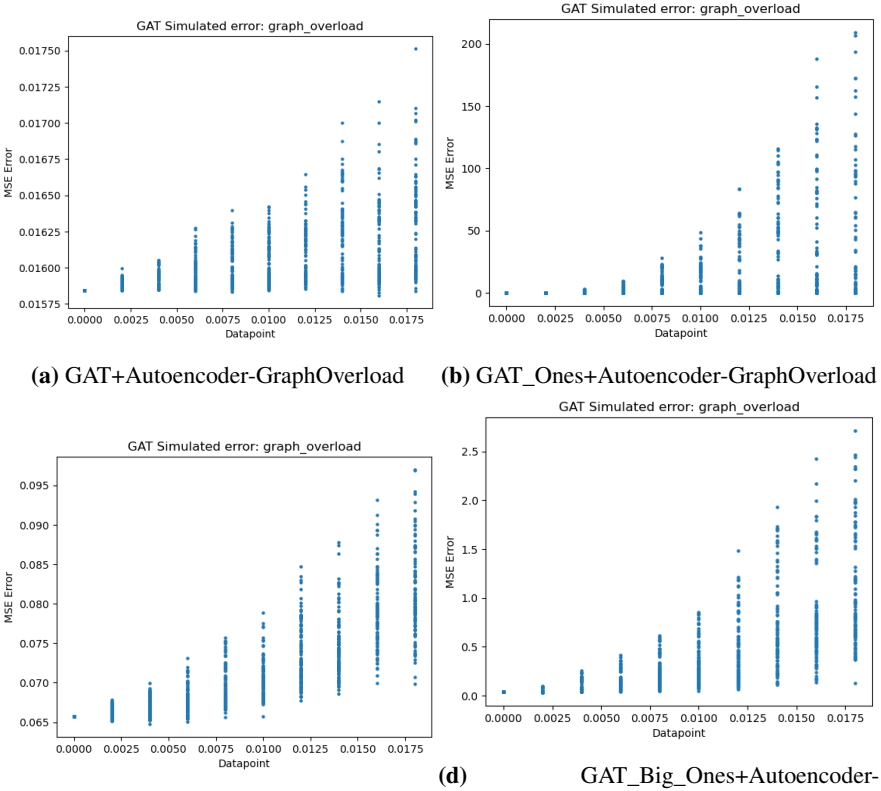


(c) GCN_Big+Autoencoder-GraphOverload



(d) GCN_Big_Ones+Autoencoder-GraphOverload

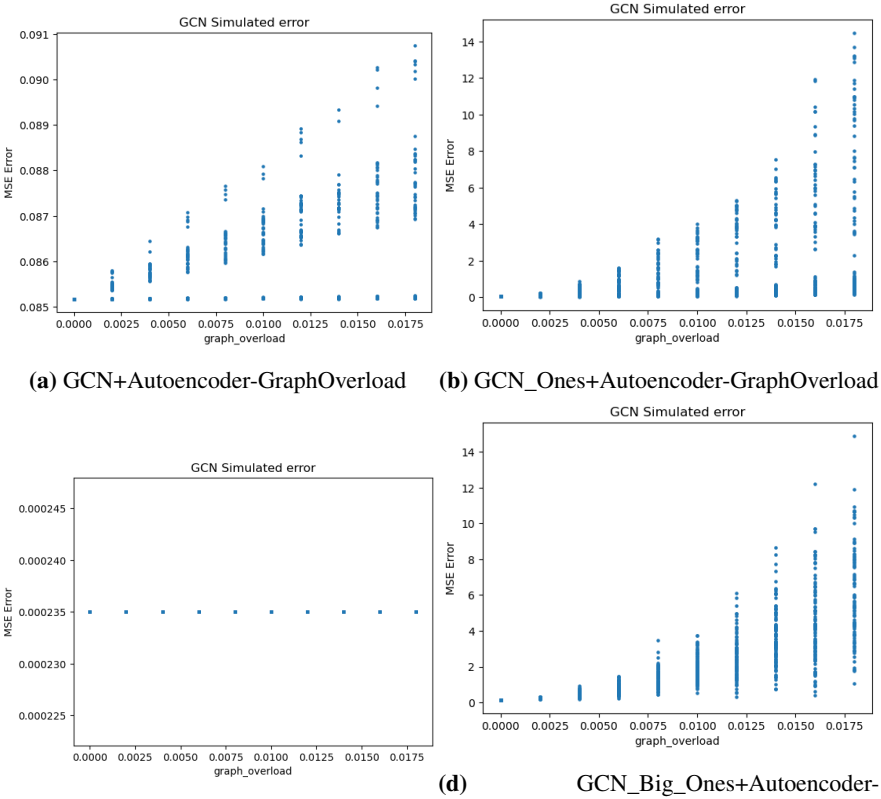
Figure A2 Comparative results of GCN variations with Autoencoder-GraphOverload



(a) GAT+Autoencoder-GraphOverload **(b)** GAT_Ones+Autoencoder-GraphOverload

(c) GAT_Big+Autoencoder-GraphOverload **(d)** GAT_Big_Ones+Autoencoder-GraphOverload

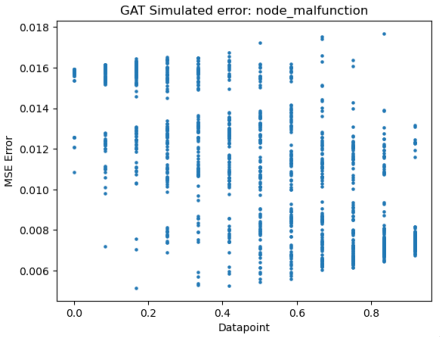
Figure A3 Comparative results of GAT variations with Autoencoder-GraphOverload



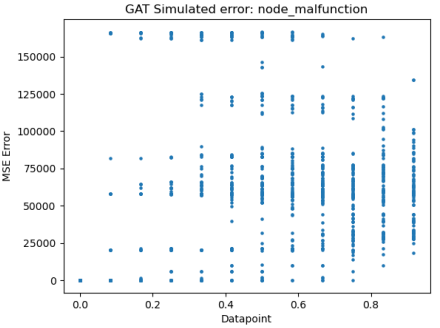
(a) GCN+Autoencoder-GraphOverload **(b)** GCN_Ones+Autoencoder-GraphOverload

(c) GCN_Big+Autoencoder-GraphOverload **(d)** GCN_Big_Ones+Autoencoder-GraphOverload

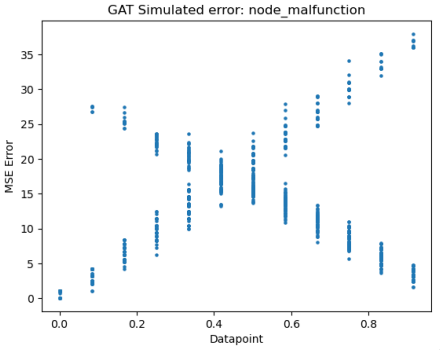
Figure A4 Comparative results of GCN variations with Autoencoder-GraphOverload



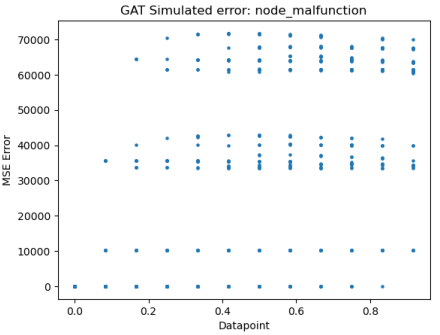
(a) GAT+Autoencoder-NodeMalfunction



(b) GAT_Ones+Autoencoder-NodeMalfunction

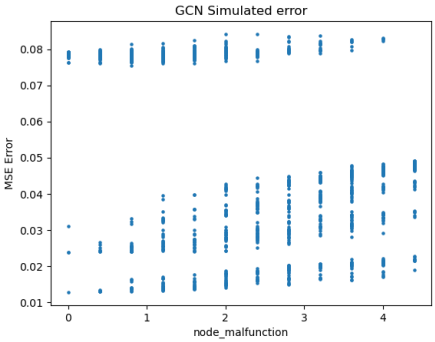


(c) GAT_Big+Autoencoder-NodeMalfunctionNodeMalfunction

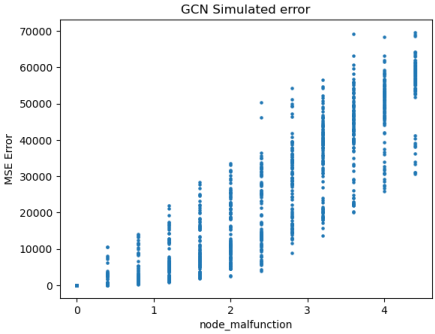


(d) GAT_Big_Ones+Autoencoder-NodeMalfunction

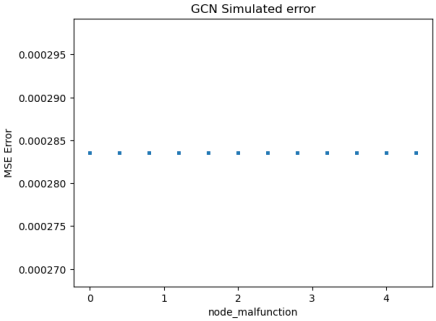
Figure A5 Comparative results of GAT variations with Autoencoder-NodeMalfunction



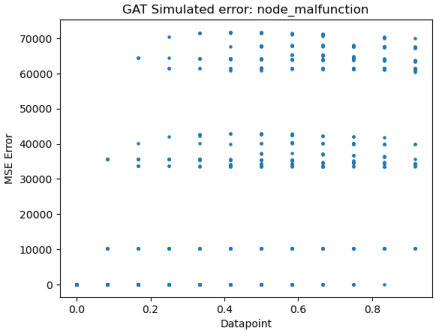
(a) CGN+Autoencoder-NodeMalfunction



(b) CGN_Ones+Autoencoder-NodeMalfunction



(c) CGN_Big+Autoencoder-NodeMalfunction



(d) CGN_Big_Ones+Autoencoder-NodeMalfunction

Figure A6 Comparative results of CGN variations with Autoencoder-NodeMalfunction

Simulated Error

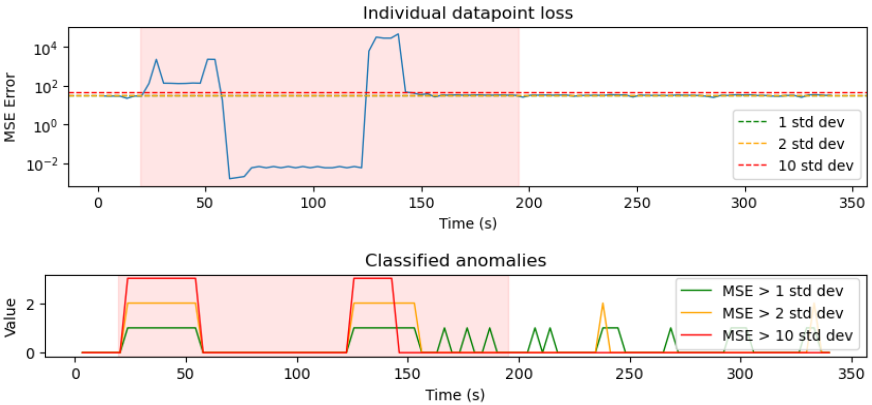


Figure A7 GAT: Node death PLC

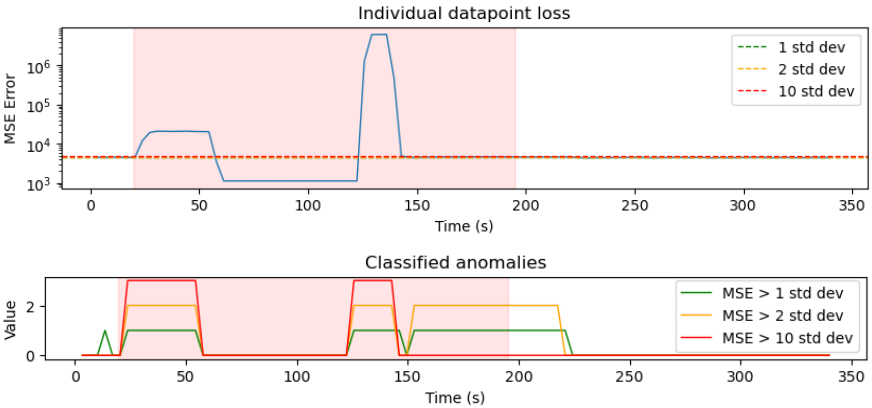


Figure A8 GAT-ones: Node death PLC

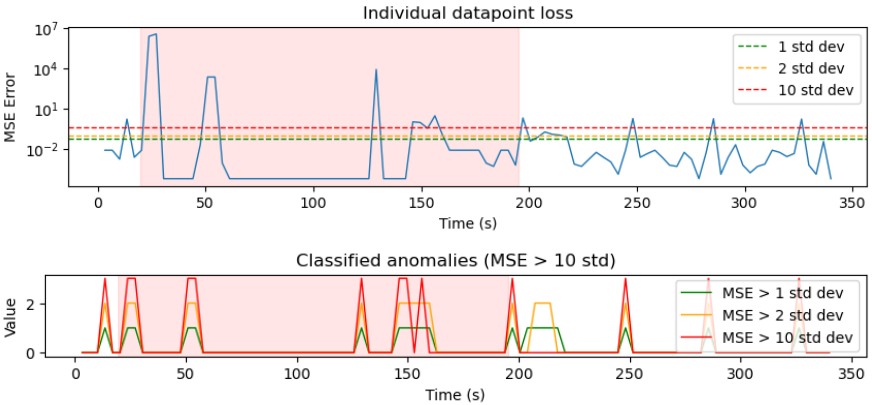


Figure A9 GCN: Node death PLC

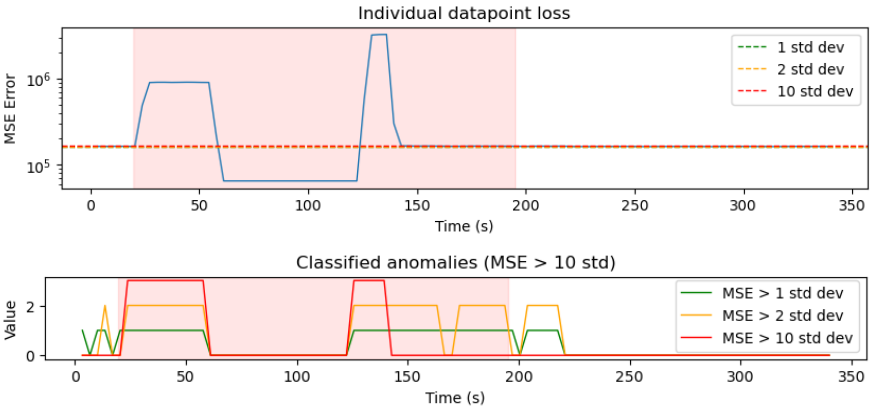


Figure A10 GCN-ones: Node death PLC

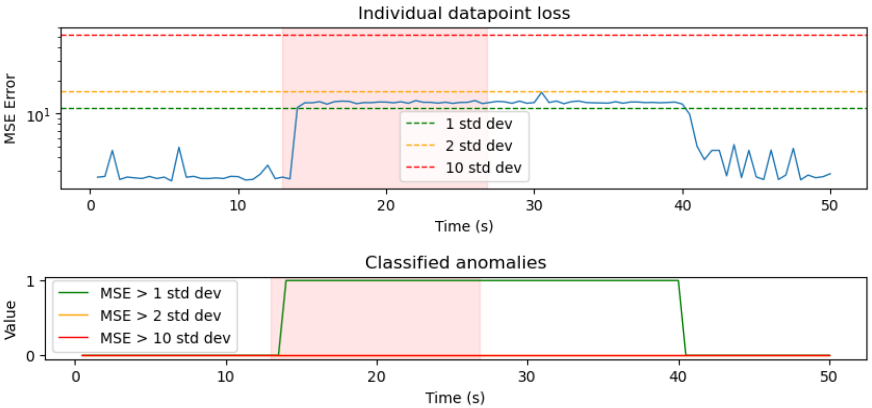


Figure A11 GAT: Node death IO & PLC

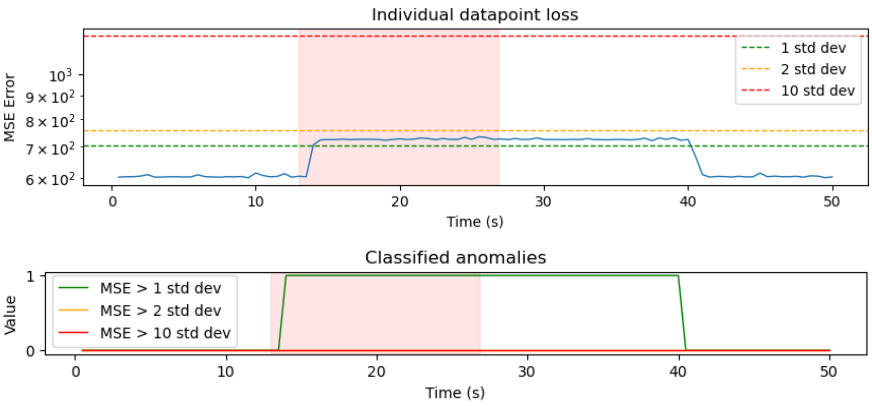


Figure A12 GAT-ones: Node death IO & PLC

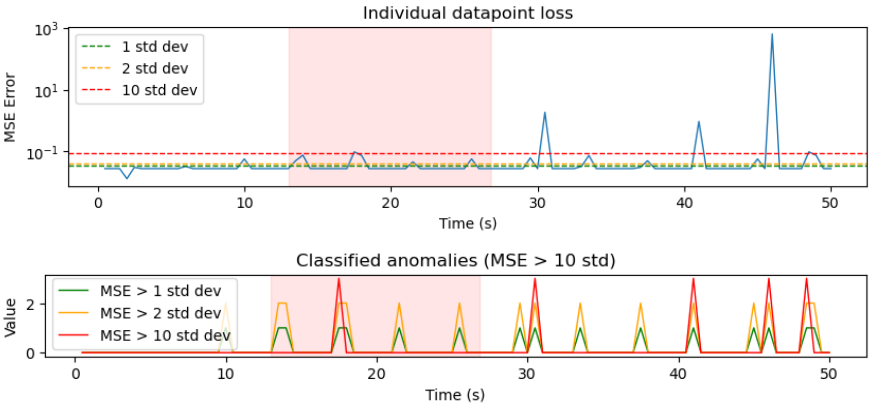


Figure A13 GCN: Node death IO & PLC

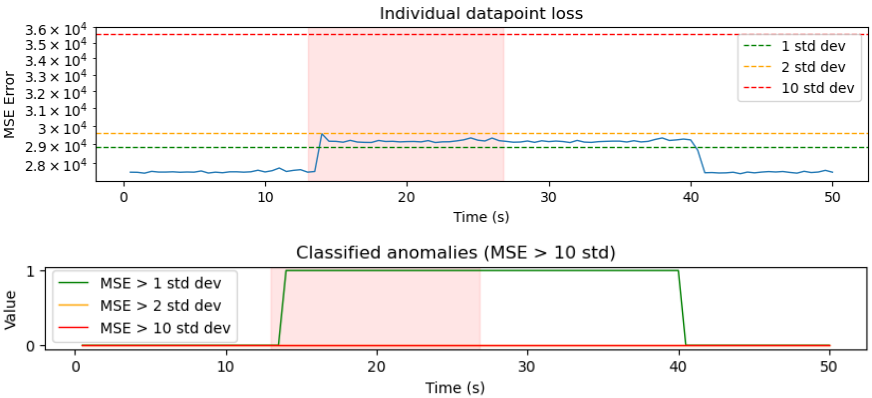


Figure A14 GCN-ones: Node death IO & PLC

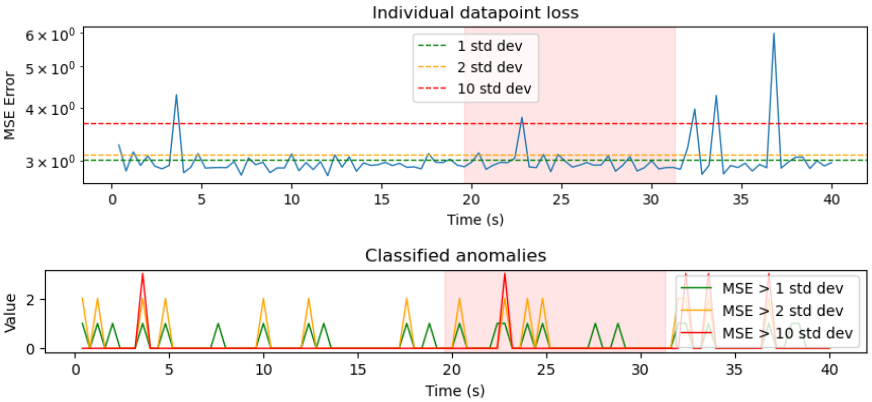


Figure A15 GAT: Cable rupture IO

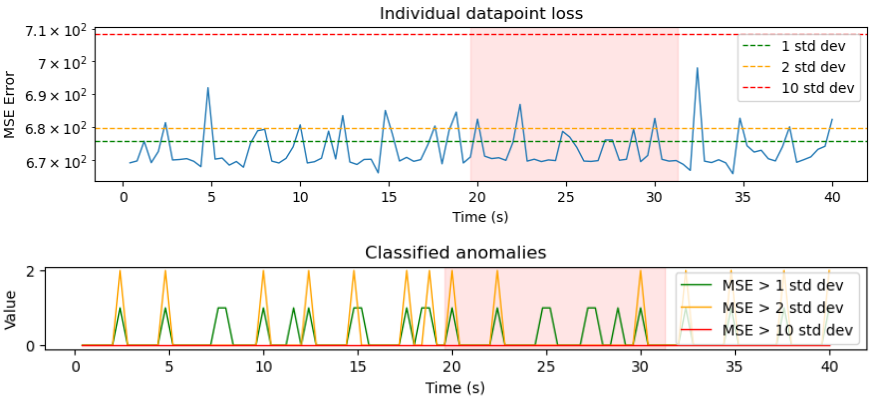


Figure A16 GAT-ones: Cable rupture IO

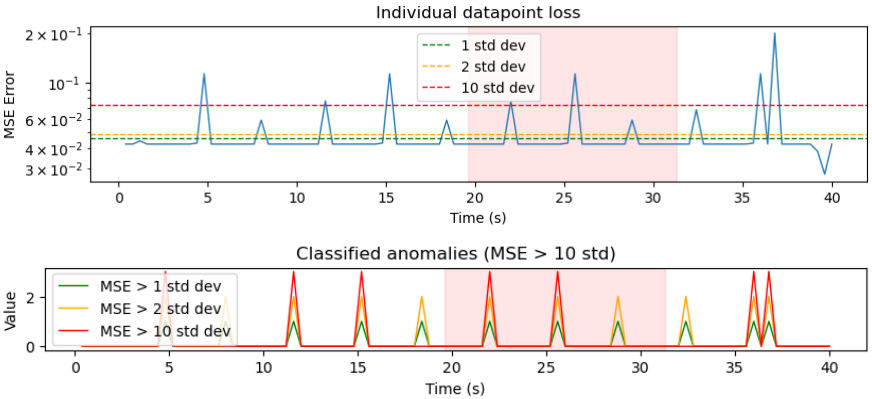


Figure A17 GCN: Cable rupture IO

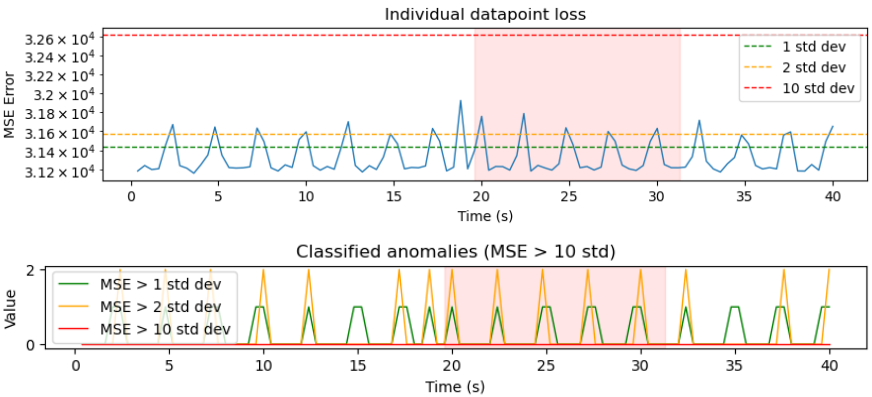


Figure A18 GCN-ones: Cable rupture IO

MTAD-GAT

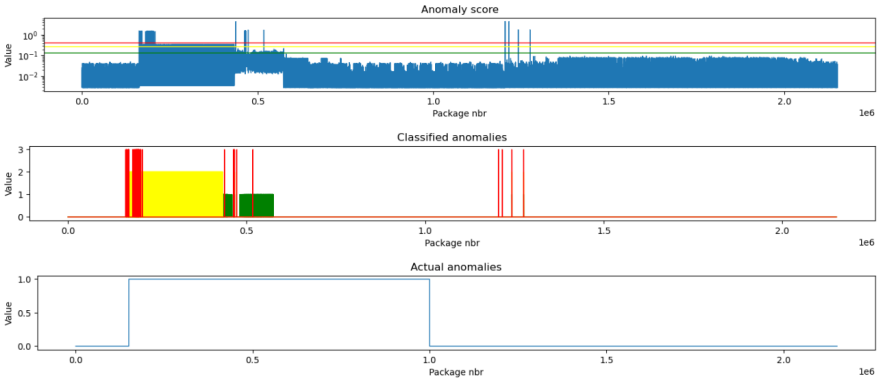


Figure A19 MTAD-GAT: Node death PLC no smoothing, accuracy-based threshold.

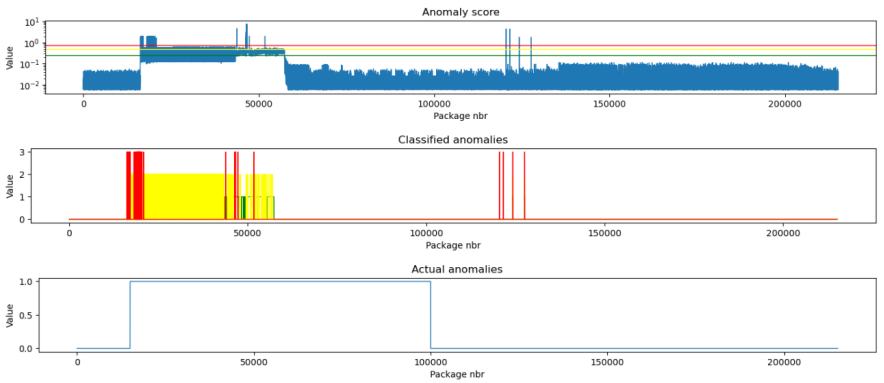


Figure A20 MTAD-GAT: Node death PLC aggregated 10, accuracy-based threshold.

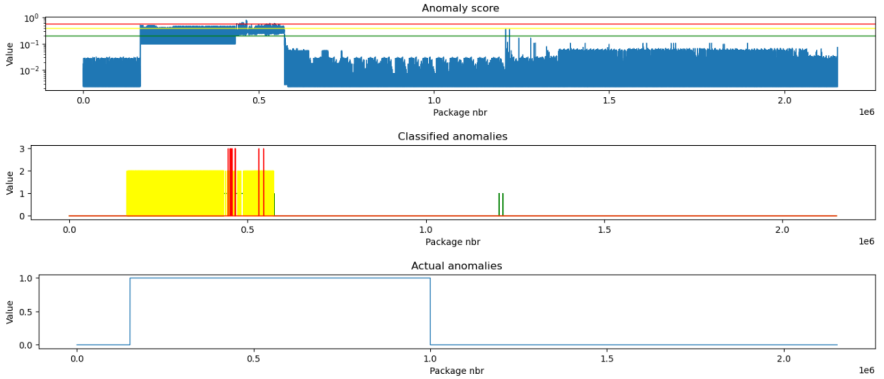


Figure A21 MTAD-GAT: Node death PLC gaussian blur 5, accuracy-based threshold.

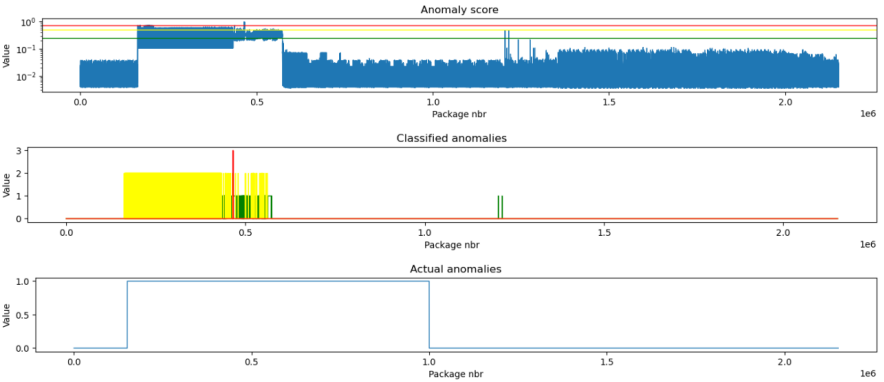


Figure A22 MTAD-GAT: Node death PLC moving average 10, accuracy-based threshold.

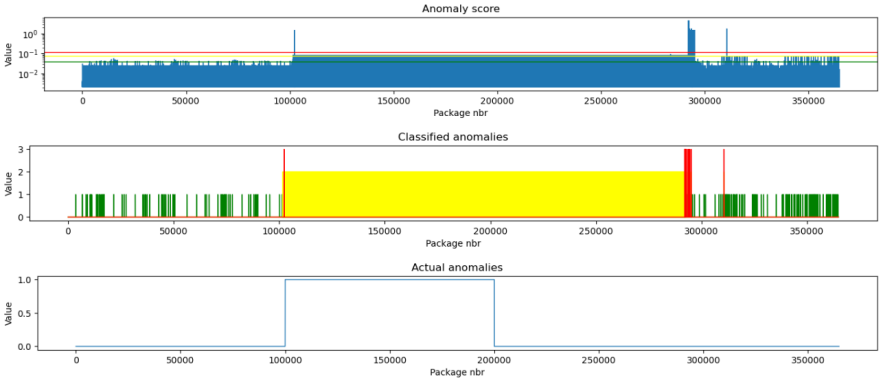


Figure A23 MTAD-GAT: Node death IO PLC no smoothing, accuracy-based threshold.

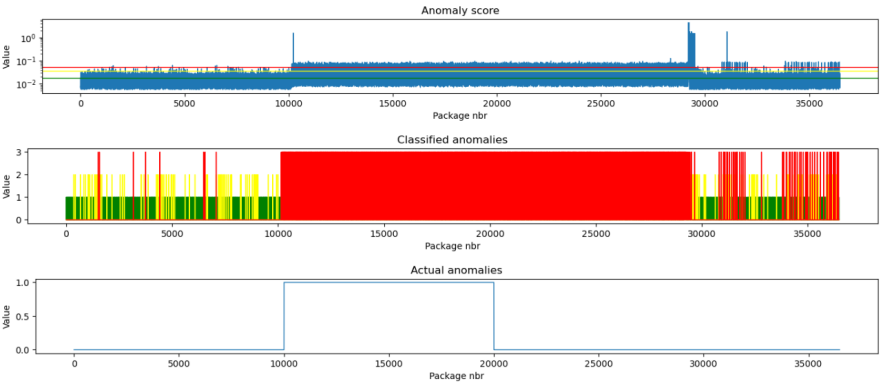


Figure A24 MTAD-GAT: Node death IO PLC aggregated 10, accuracy-based threshold.

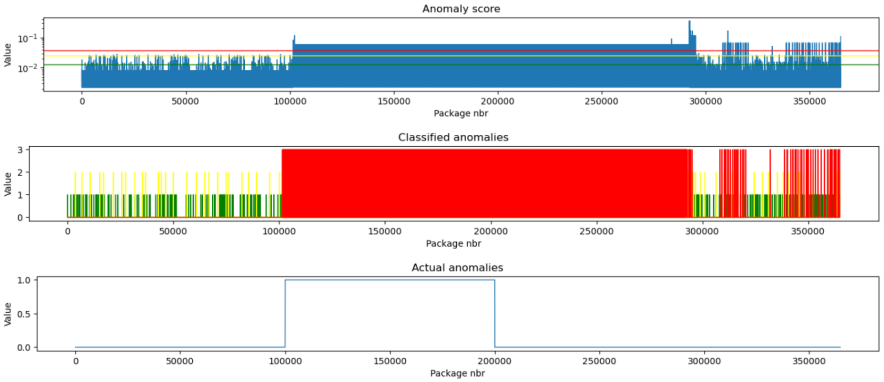


Figure A25 MTAD-GAT: Node death IO PLC gaussian blur 5, accuracy-based threshold.

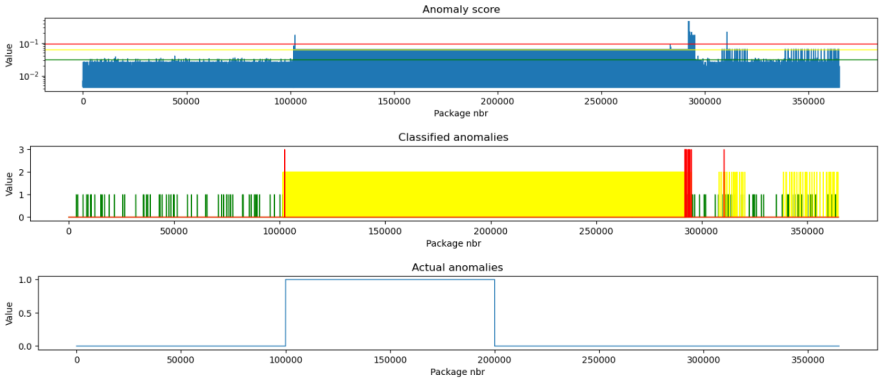


Figure A26 MTAD-GAT: Node death IO PLC moving average 10, accuracy-based threshold.

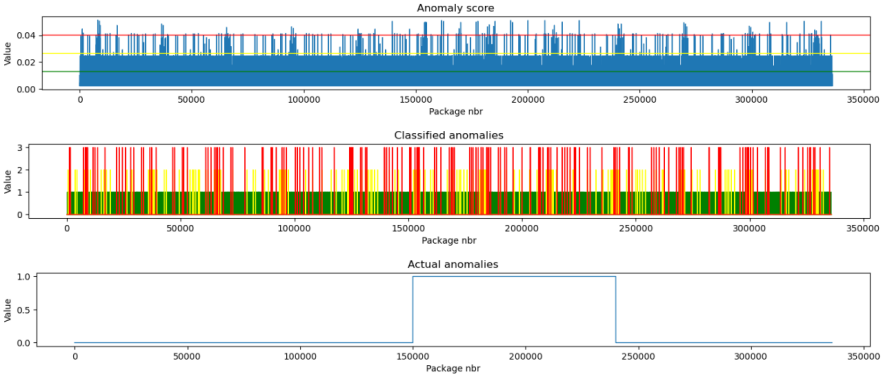


Figure A27 MTAD-GAT: Cable rupture IO no smoothing, accuracy-based threshold.

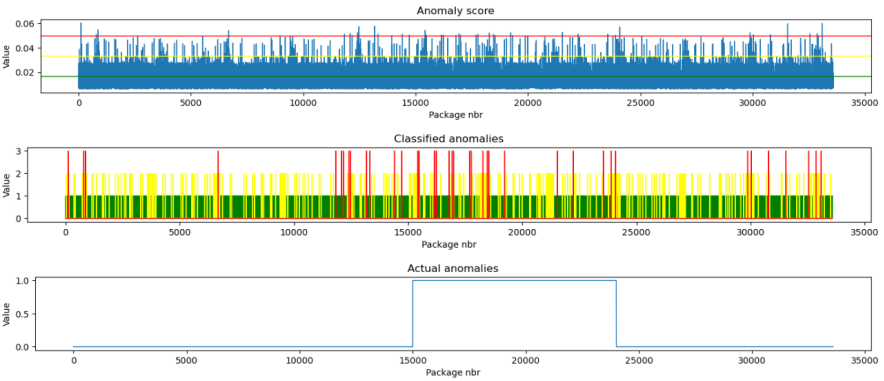


Figure A28 MTAD-GAT: Cable rupture IO aggregated 10, accuracy-based threshold.

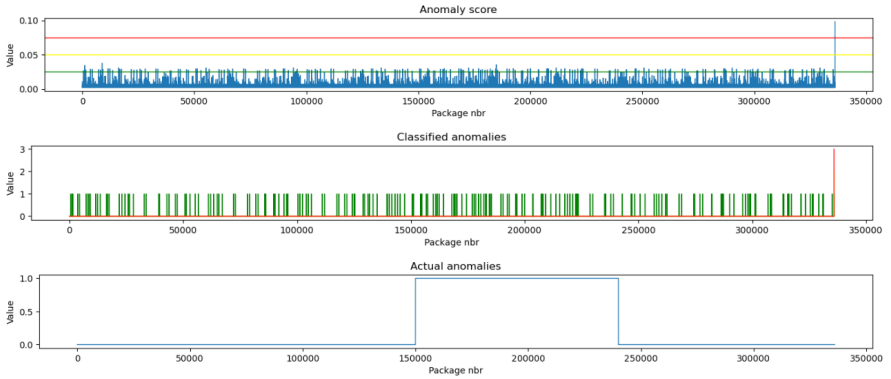


Figure A29 MTAD-GAT: Cable rupture IO gaussian blur 5, accuracy-based threshold.

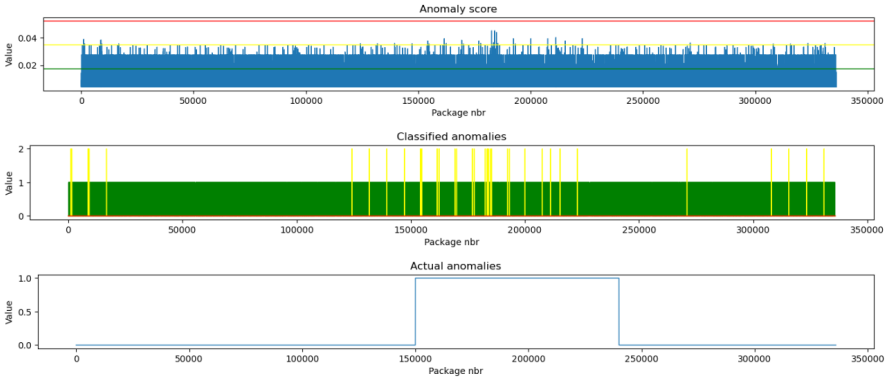


Figure A30 MTAD-GAT: Cable rupture IO moving average 10, accuracy-based threshold.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2024

Document Number
TFRT-6249

Author(s)
Ivar Källander
Stanislaw Swirski

Supervisor
Mikael Lindberg, Saab Kockums, Sweden
Johan Eker, Dept. of Automatic Control, Lund University
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
Graph Attention Network-Based Monitoring of Complex Operational Systems

Abstract

Operational systems, such as industrial automation, autonomous vehicles, and larger air/sea/landcrafts, often contain a large number of heavily connected systems with real-time requirements for their functionality. For such systems, detecting and responding to anomalies is both challenging and crucial. Until recently, such anomalies were monitored using heuristic methods, or even humans monitoring the systems. Such approaches often fail to detect anomalies accurately due to the complexity of the systems. A continuous development of the systems also poses a significant challenge, as the current anomaly detectors have to be updated, and staff trained. Geometrical deep learning is a well known tool used for anomaly detection in applications where the data can be represented as a graph. However, to our knowledge it is yet to effectively be used for complex operational systems, currently only being used for simpler cases such as fraud detection. Recently, a new architecture named Graph Attention Network (GAT) has been studied as an anomaly detection method. Its ability to incorporate information in large networks makes it potentially useful for complex operational systems. In this thesis we evaluate different machine learning based methods for anomaly detection, trained on data from real operational systems, focusing on submarines. The methods evaluated include GCNs, GATs and Autoencoders. We also evaluate which data preprocessing methods that are best suited for our case. The results of this thesis provide a basis for further research and show that GATs could be successfully implemented as anomaly detectors for complex operational systems, though the usage may not be justified without sufficient data and complexity.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-113

Recipient's notes

Security classification

http://www.control.lth.se/publications/